Design of the Transform Phase of the Digital Fortran Compiler

Carl Offner
Alex Nelson
John Bircsak
Jonathan Harris
Shin Lee
David Offner

April 9, 2001

Abstract

This is the design report for the Transform components of the Digital Fortran compiler, later inherited by the Compaq and HP Fortran compilers for Tru64 Unix running on Alpha.

Transform is that part of the compiler which deals with High Performance Fortran (HPF) data mapping directives and parallel constructs. This report describes the data structures and internal transformations needed to turn HPF code into SPMD code with message-passing, and describes the plethora of optimizations introduced in order to generate efficient code. It also contains discussions of the main design decisions that were made, both for compiler efficiency and for high-quality code generation, with explanations of the trade-offs that were considered.

The same technology was later extended to apply to OpenMP Fortran code with data mapping directives added for the purpose of generating efficient code on non-uniform memory access (NUMA) machines; that extension is also described here.

Contents

Notation Used in this Report

1. The term “node” in this report always refers to a node in the compiler’s internal representation of an expression tree. What is often referred to as a node of a MIMD machine is here called a “processor”. Such a processor may in fact be composed of several physical computing units, some of which may even compute in parallel. Such internal structure, however, is invisible to the compiler described here; a processor is seen by this compiler as an atomic object.

   An exception is made in Chapter 16, where a processor is referred to as an “hpf processor”, and its internal structure is visible to the compiler.

2. If g is a function, the notation $g \uparrow$ indicates that g is an increasing function; and similarly, $g \downarrow$ indicates that g is a decreasing function. The functions for which this is used are all data or iteration allocation functions, of the form

   $$g(x) = \mathit{alloc\_stride} * x + \mathit{alloc\_offset}$$

   For such functions, $g \uparrow \iff \mathit{alloc\_stride} > 0$ and $g \downarrow \iff \mathit{alloc\_stride} < 0$.

3. $\vee$ is the boolean OR, and $\wedge$ is the boolean AND. An empty OR is FALSE; an empty AND is TRUE.

References to Future Work

Some future development work and optimizations for this compiler are indicated in this report. Descriptions of this future work are distinguished from the main text by being set in italics in paragraphs with wider margins.

> *This is an example of how a description of future work might appear in this report.*

Page references to these descriptions are collected in the index, under the heading “future tasks”.

Chapter 1

The Programmer's View of the Compiler

1.1 The source language

This report describes the Transform components of the compiler whose source language is Digital Fortran Version 5.0 [5], and whose target machine is either a farm of Digital Alpha workstations or an SMP machine based on Alpha processors.

Digital Fortran Version 5.0 is in turn based on the latest version of standard Fortran ([6], often referred to as "Fortran 95"), and also includes all of High Performance Fortran ("HPF") Version 2.0 ([7]) with the following exceptions:

- nested FORALLs and WHERE statements within FORALLs.
- passing CYCLIC(N) arguments to local (i.e., HPF_LOCAL) procedures.
- accessing non-local data within PURE functions called within a FORALL.
- the SORT_UP and SORT_DOWN library routines.

In addition, the language supports the following Approved Extensions described in Part III of the HFP Version 2.0 specification:

- mapped pointers (Section 8.8).
- mapped components of derived types (Section 8.9)
- the SHADOW directive (Section 8.12).
- the HOME and RESIDENT clauses of the ON directive (parts of Sections 9.2 and 9.3) applied to INDEPENDENT DO loops.
- the HPF_LOCAL and HPF_SERIAL extrinsic kinds, with their supporting libraries, as specified in Sections 11.1, 11.3, 11.5, and 11.7, with the exception of the LOCAL_BLKCNT, LOCAL_LINDEX, and LOCAL_UINDEX routines.

- the extended HPF intrisic and library procedures as specified in Section 12.2 except for the HPF_MAP_ARRAY and HPF_NUMBER_MAPPED routines.

This design report mainly describes how the compiler handles the HPF constructs in the language. These constructs support *data-parallel* applications; i.e., applications containing large amounts of data, typically in the form of arrays, which are to be operated on in parallel.

These techniques, originally developed to handle HPF, have been adapted to support data layout directives in OpenMP Fortran, for use in compiling code to be executed on NUMA machines. These data layout directives are based on the corresponding HPF directives. The way the compiler manages these directives in processing OpenMP Fortran code is explained in Chapter 16.

1.1.1 Some comments on the language

Sequence association In (non-HPF) Fortran, any method of argument passing other than by assumed shape implies sequence association between the actual and the dummy arguments. This is *not* true in HPF, however: sequence association is enforced only in certain specified circumstances (e.g., there is a SEQUENCE directive visible to the compiler).

In particular, in Fortran, without an explicit interface, an actual argument which is an array element designator would be assumed to represent the first element of an array which is sequence associated with its corresponding dummy. On a distributed-memory machine, this in principle would lead to data motion which would really be unnecessary if the dummy were not declared (via the HPF SEQUENCE directive) to be sequential.

However, HPF does not honor sequence association without such a directive, and so we do not support this normal Fortran interpretation. Without a SEQUENCE directive visible to the compiler, a call from a global HPF routine to another global HPF routine with an actual argument of $A(3)$, say, will result in the called routine being able to access just the element $A(3)$, and no other elements of A.

DO loop indexes Variables which are used as DO loop indexes should not have any HPF data distribution directives applied to them. If they do, performance will most likely be degraded. See Section 3.1.1, page 17.

Aliased arrays Many array sections are passed copy-in/copy-out to subroutines. There are some unusual cases involving aliasing where this leads to different program semantics than the conventional Fortran pass-by-reference. Such cases are actually forbidden by the Fortran Standard—see Section 12.5.2.9 of the Standard ([6]), and also Section 12.5.2.3 of [1]—but were known to occur in previous Fortran 77 code (where they were also illegal).

Implicit reshaping By default, HPF does not support the linear memory model for distributed data. In particular, without using the SEQUENCE directive, there can be no implicit reshaping via COMMON blocks or by passing an argument into a function and having the function receive it in a different shape.

1.1.2 Some language restrictions

We currently insist on four restrictions to the HPF language; those restrictions are set in bold face type below:

CYCLIC(n) arguments We currently restrict the language so that

> **Arguments passed to HPF_LOCAL routines cannot be distributed CYCLIC(n).**
>
> See page 34.

PURE functions The meaning of a PURE function in HPF is further restricted by adding the following constraint to the conditions in the HPF Language report:

> **No name in the scope of a PURE procedure and no entity that is associated other than by argument association (that is: use, host, pointer, storage, or sequence association) with any name in the scope of a PURE procedure may be explicitly (by user-written HPF directives) mapped (aligned or distributed).**
>
> The purpose of this restriction is to allow iterations of FORALL constructs containing PURE functions to be distributed over processors in a distributed-memory machine where message-passing is used for remote memory accesses. This restriction is capable of being checked by the compiler at the time the PURE function itself is compiled. See page 17 for some discussion of this restriction.
>
> In addition, we interpret the HPF standard to to say that a PURE function may not modify any of its arguments. (A PURE subroutine may, however; but note that PURE subroutines cannot be called directely from inside a FORALL construct.)

mapped components of derived types Fortran requires that each bound in an *explicit-shape-spec* of a field of a derived type must be a *constant* specification expression. (Outside of a derived type, it does not have to be constant.)

> HPF needs to require this also of expressions in any mappings applied to such fields of derived types. For instance,

```
subroutine foo(n)
   integer n
   type t
      real, dimension(10) :: a
      !hpf$ distribute(cyclic(n)) :: a
   end type t
   ...
end subroutine foo
```

> is illegal, because although n is a specification expression, it is not a constant specification expression.

SEQUENCE and mapped components of derived types The NOSEQUENCE directive may not be specified for a component of a derived type which has the SEQUENCE attribute. Thus, the following is regarded as non-conforming:

```
type, sequence :: mytype
   integer :: i, j
   real :: x
   real, dimension(15) :: A
   !hpf$ distribute(block), nosequence :: A
end type mytype
```

1.1.3 Addition to the language

We have added one feature to the HPF Language: an HPF subprogram can be called from a non-parallel main program or subprogram. For this to work correctly, the non-parallel main program or subprogram must be compiled with the `-nowsf_main` switch. The details are set out in Section 6.7 of [5]; also see section 1.3.1 below (page 7).

1.2 The machine model

As far as the transformation components of the compiler are concerned, the target machine is a distributed memory MIMD machine. The processors are numbered linearly starting from 0, and are fully interconnected. That is, the cost of sending a message between any two distinct processors is independent of their processor numbers.

Each processor knows its own processor number; this number is returned from a call to the function *my_processor*`()`. Processors communicate via paired SEND and RECEIVE calls.

The startup cost for a message is large compared to the incremental cost of sending each word in the message, by a factor of at least 10,000. For this reason, a great advantage is gained by packaging the messages between each two processors together to make the number of messages as small as possible, and the size of each message as large as possible.

We currently also generate code that runs on a shared-memory machine. However, we do this by treating the machine as if it were distributed-memory, so that each processor has its own private memory. SEND and RECEIVE calls are then implemented as memory-to-memory copies.

> *This actually generates quite good code in many cases. It is not, however, idiomatic code generation for a shared-memory machine, and we are in the process of changing this model.*

1.3 Programming models

There are three programming models which this compiler supports:

Data Parallel Model The source code contains aggregate data which is to be operated on in as parallel a fashion as possible. The language contains directives which the programmer may use to tell the compiler how data is distributed across processors. Parallelism is attained by having processors operate simultaneously on different portions of this data. From the programmer's viewpoint, however, there is only a single thread of control.

The source code will not ordinarily contain SEND or RECEIVE calls. If they are present in the source, they refer to communication with some exterior program (operating on another set of processors).

Routines written in this fashion are referred to as "global HPF" (or simply "HPF") routines.

Explicit SPMD Model A program is written containing explicit SENDs and RECEIVEs and references to *my_processor*`()`, and is loaded onto all processors and run in parallel. Storage for all arrays and scalar variables in the program are replicated; that is, identical storage for each such data object exists on each processor. The intent is that in general, the storage for an array A on distinct processors represents distinct slices through a global array which the programmer is really concerned with. (Only the programmer is aware of these global arrays, however. They have no representation in the source code.)

Thus, this is a multi-threaded programming model, where the same program runs on each processor. Different processors will in general, however, execute different instructions, because the program on each processor is parametrized by references to *my_processor*`()`, and because parallel data on different processors in general has different values. Since in addition the data on each processor consists of different slices of global arrays, this programming model is referred to as "single-program, multiple-data", or SPMD.

Each array element really has two kinds of addresses: on one hand it has a global address, and on the other hand, it has a two-part address which consists of a particular processor and a local memory address in that processor. It is up to the programmer to handle the translation between these two forms of addressing. Similarly, it is up to the programmer to insert whatever SENDs and RECEIVEs are necessary; this involves figuring out which array elements have to be sent/received and where they have to go or come from. Again, this involves explicit translation between global and local addressing.

The explicit SPMD model can take two forms within the context of Digital Fortran:

vanilla explicit SPMD This is just as described above.

HPF_LOCAL routines These are routines coded as above, but having available to them inquiry library functions which enable the programmer to retrieve information about global arrays corresponding to dummy arrays in the program.

Single Processor Model This is the conventional Fortran programming model. A program is written to execute on one processor which has a single linear memory. Sequence and storage association holds in this model, since it implements conventional Fortran.

In our implementation, a single processor routine may execute on any processor, with one exceptional case: if it is the main program, it always executes on processor 0.

A subprogram written in the single processor model may additionally be declared to be EXTRINSIC(HPF_SERIAL). This is just an indication that it may be called from a global HPF routine, and that the compiler will ordinarily generate code in the global routine to move all the arguments to processor 0 before the call, and back afterwards. The only place where this does not happen is when the HPF_SERIAL subprogram is called from inside an INDEPENDENT DO loop and is declared to be RESIDENT. In this case, no data motion is needed, and the call simply happens on whatever processor is executing the loop iteration.

The compiler can be invoked in several ways, corresponding to these different programming models:

1. When invoked with the `-hpf` switch but without an EXTRINSIC declaration, the compiler expects a global HPF source program, and generates the correct addressing and message-passing in the emitted object code.

   Thus, the compiler will emit code containing explicit SENDs and RECEIVEs and references to *my_processor*`()`. In effect, the compiler accepts a global HPF program and emits explicit SPMD object code.

   This implements the data parallel programming model.

2. without the `-hpf` switch and without an EXTRINSIC declaration, the compiler generates addressing for a single linear memory. Such a routine can be used to implement the explicit SPMD programming model, by containing programmer-written message passing as needed. It can also be used to implement the single processor model. There is no way to discover by inspecting the source code of such a routine which programming model is intended. The programming model is entirely determined by how the routine is invoked: on one processor, or simultaneously on many.

3. If the routine is declared as EXTRINSIC(HPF_LOCAL) (note: if this routine is called from a global HPF routine, this declaration must also be visible in an explicit interface in the calling routine), then the routine by that fact becomes an explicit SPMD routine. The HPF_LOCAL declaration also has two other effects:

    - If the calling routine is global HPF, it is a signal to the compiler when compiling the calling routine to localize the passed parameters. That is, only the part of each passed parameter which lives on a given processor is passed as the actual parameter to the instance of the HPF_LOCAL routine which is called on that processor.
    - It makes available to the HPF_LOCAL routine a library of inquiry functions (the HPF Local Routine Library) which enable the local routine to extract information about the global HPF data object corresponding to a given local dummy parameter. This information can then be used (explicitly, by the programmer) to set up interprocessor communication as needed.

4. If the routine is declared as EXTRINSIC(HPF_SERIAL) the compiler processes it as any other single-processor Fortran routine. If the routine is called from a global HPF routine, then the EXTRINSIC(HPF_SERIAL) declaration must be visible to the calling routine in an explicit interface. The compiler ordinarily[1] generates code in the calling routine that moves all arguments to the routine to processor 0 before the subroutine call, and copies them back after the call if necessary.

An EXTRINSIC(HPF_LOCAL) or an EXTRINSIC(HPF_SERIAL) declaration in a routine overrides the `-hpf` switch—it is as if that switch were not invoked for the routine. This makes it possible to have an HPF_LOCAL or HPF_SERIAL subroutine which is internal to a global HPF routine or in the same file as that routine. The compiler would be invoked with the `-hpf` switch, but that switch would have no effect on the compilation of the HPF_LOCAL or HPF_SERIAL subroutine.

The following table illustrates how the different compiler modes are invoked:

	with `-hpf` switch	without `-hpf` switch
with EXTRINSIC(HPF_SERIAL) directive	single processor	single processor
with EXTRINSIC(HPF_LOCAL) directive	SPMD	SPMD
without any EXTRINSIC directive	data parallel	single processor or SPMD

[1] except, as mentioned above, for a RESIDENT routine called inside an INDEPENDENT DO loop

1.3.1 Who can call whom

A single processor routine which is the main program always executes on processor 0. Other than that, there is no restriction on where a single processor routine executes.

- A single processor routine can call
  - a single processor routine.
  - a global HPF routine, provided that
    1. the single processor routine is executing on processor 0.

       This restriction on the calling routine is made because the called routine is responsible for distributing the data (as explained below in section 3.4.2), and therefore has to know which processor to get that data from.
    2. the single processor routine is compiled with the `-nowsf_main` switch.

       This is so that the single processor routine can know to call a special version of the called routine that is set up to distribute the data from processor 0, as described above.
    3. the (global HPF) called routine does not contain any assumed size dummy arguments.

    The reason for the restriction on the called routine is that an assumed-shape array is sequence-associated and cannot be explicitly mapped. Hence it will have the default layout (see Section 3.1, page 16), which in our case is a replicated layout. But the called routine has no way of replicating the passed array if it is assumed size, because the routine does not know the size of the array being passed. This second restriction can be enforced by the compiler (see Section 3.4.2, page 19).
- A global HPF routine can call
  - a global HPF routine.
  - an HPF_LOCAL routine.
  - an HPF_SERIAL routine.
- An HPF_LOCAL routine can call
  - an explicit SPMD routine, which might be another HPF_LOCAL routine.
  - a single processor routine. Such a routine will run on the processor from which it is called. An example of this could be a scalar library routine.
- An explicit SPMD routine which is not an HPF_LOCAL routine can call
  - another explicit SPMD routine. This routine, however, cannot be an HPF_LOCAL routine.
  - a single processor routine. Such a routine will run on the processor from which it is called, as above.

With a few exceptions, explicit SPMD routines and single processor routines will not be discussed further in this report. The programming of such routines is straightforward Fortran programming, and the Transform phases of the compiler are not invoked for such routines.

1.4 General parallelism issues

1.4.1 Parallel constructions

The following types of constructions are candidates for being run in parallel:

- Statements containing Fortran array expressions.
- FORALL constructs.
- INDEPENDENT DO loops.

To the extent allowed by the programmer's specification of data layout, and by the number of processors in the target machine, iterations of these constructions will be distributed over the processors.

In this release of the compiler, the iterations of Fortran DO loops are not distributed over processors unless the loop is marked as INDEPENDENT. The processing of non-INDEPENDENT DO loops is described in Section 3.2.2, page 17.

1.4.2 Communication and synchronization

Since this report confines itself to global HPF programming, when we refer to communication, we mean communication generated by the compiler. In this first release of the compiler, all communication is in the form of synchronous SEND and RECEIVE calls. Specifically,

- If processor A executes a SEND of a message to processor B, processor B must also execute a RECEIVE for that message. Thus, SENDs and RECEIVEs always occur in pairs.
- An executed SEND call does not return until the information in the SEND buffer has been transmitted; hence the buffer can be re-used or overwritten as soon as the SEND returns. The fact that the SEND has returned does not mean, however, that the information has arrived at the receiving processor.
- An executed RECEIVE call does not return until the information has arrived and been placed in the RECEIVE buffer. Thus, as soon as the RECEIVE returns, the information can be extracted from the RECEIVE buffer and used.

Because of the handshaking imposed by the SEND/RECEIVE protocol, no additional synchronization of any sort is necessary. If a processor needs data in another processor's memory it will execute a RECEIVE for that data. The other processor will execute a SEND of that data only when the data is ready to be sent (i.e. when the data has been updated if necessary to have the proper values). Thus, the RECEIVE can really be executed at any time, and will not return until the correct data has been received.

Similarly, a SEND can be executed at any time a processor is ready to send data. The data will not be received by the receiving processor until that processor is ready to receive the data and executes a RECEIVE to do so.

For this reason, the compiler does not generate calls to any global synchronization routines.

> *We plan to investigate the use of asynchronous SENDs and RECEIVEs in future releases of this compiler.*

1.4.3 Communication costs

The cost of transmitting a message from one processor to another can be divided into two parts:

- A startup cost. This can be thought of as the cost necessary to establish the communication path between the two processors. It is a constant cost.
- The cost of transmitting the data once the communication path has been established. This cost is proportional to the amount of data being transmitted.

In this machine, the startup cost is huge compared to the incremental cost of data transmission. Therefore we can only get good performance from this machine by vectorizing communication. In other words, if one processor has a number of data elements to send to another processor, it is important to generate one long message rather than a number of shorter ones.

This consideration has affected much of the design. In particular, it has made it necessary for the compiler to generate code to pre-process communications where possible to package up data into large messages. It also has meant that we raise what otherwise might have been data motion in the interior of an expression up to the statement level, so that this packaging can be performed.

> *Future work along these lines will include packaging up motion contained in distinct iterations of a loop nest; and also packaging motion contained in more than one statement, where possible.*

1.4.4 Data distribution

In compiling code to be run in parallel, a key issue is how the data is to be distributed across the processors. Without the use of data optimization techniques (which enable the compiler to assist in this determination), data distribution is determined completely and explicitly by the programmer through the use of directives. In any case, the following considerations are important:

1. Increasing the extent of the distribution (for instance, by distributing more of the dimensions of an array) in principle increases the parallelism in a program, which is a good thing to do.
2. On the other hand, in applications which are not completely (or "embarrasingly") parallel, too much distribution can lead to an unacceptable increase in interprocessor communication costs. It may be the case that allocating one or more dimensions of an array serially (i.e. not across processors) may drastically reduce communication costs.
3. How dimensions are allocated (serially, cyclically, block, etc.) can affect load balancing, which we want to maximize. It can also affect the ability of the compiler to generate efficient code—a good example of this is the processing of nearest-neighbor computations; see Chapter 15.

> *In the future, this project will look at the possibility of implementing automatic data optimization techniques, as set out in the papers [2, 8, 9, 10].*

1.5 The number of processors

The number of processors will not in general be known at compile time. The user can, however, specify the number of processors at compile time by using an optional parameter to the `-hpf` switch. If the switch is not used, the compiler does not make any assumptions about the number of processors.

The user may specify any number of processors by means of the switch. In particular, the number of processors does not need to be a power of two.

If the number of processors is not a power of two, or at least highly composite, or if the number of processors is not known at compile time, the efficiency of the compiled code may be noticeably degraded.

For example, suppose that a large 2-dimensional array is to be distributed in a (BLOCK, BLOCK) fashion over a machine which is known to contain 64 processors. The compiler might distribute the array over an 8×8 grid. If on the other hand, one more processor were added to make 65 processors, then since the only factorization of 65 is 5×13, the compiler would have to distribute the array over a 5×13 grid, and this might well degrade performance. And of course, if the number of processors were a prime number, only one of the dimensions of the array could be distributed over the processors, and all the other dimensions would have to be allocated serially (i.e. "down memory"). Note that the compiler does not attempt to second-guess the user: if the user specifies 65 processors, the compiler will use all 65, even if it had reason for "believing" that 64 might be better.

In addition, if the number of processors is not known at compile time, then the compiler cannot assume that the number of processors on which the program will execute is composite. Therefore, only one dimension of any array is distributed by the compiler in such a circumstance, and the others are allocated serially, no matter what the programmer has specified in mapping directives.

The number of processors specified on the command line must be equal to the number of processors specified in any non-scalar HPF PROCESSORS directive; this number must also equal the number of processors on which the program is executed. The compiler generates code to check both these constraints at the time the program is run.

Chapter 2

The Compiler Architecture

This short chapter gives a brief overview of the compiler architecture. A more detailed description of the various Transform phases is given in Chapters 5–15.

2.1 Where the Transform section fits into the compiler

Figure 2.1 illustrates the high-level architecture of the compiler. The curved path is the path taken when the `-hpf` switch is not used, or when the scoping unit being compiled (see Section 2.3) is declared as EXTRINSIC(HPF_LOCAL).

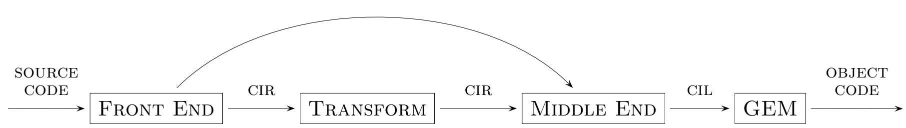


Figure 2.1: Where the transform component fits into the compiler.

The parts of the compiler in Figure 2.1 have the following functions:

Front End Parses the input code and produces an internal representation called the Common Intermediate Representation (CIR). The CIR consists of an abstract syntax tree and a symbol table. Extensive semantic checking is performed.

Transform Performs the transformation from global HPF to explicit SPMD form. To do this, it localizes the addressing of data, inserts motion where necessary, and distributes parallel computations over processors. Transform inputs and outputs CIR.

Middle End Translates the CIR into another form of internal representation called the Compact Intermediate Language (CIL).

GEM Performs local and global optimization, code generation, and register allocation on the CIL representation; emits binary object code.

2.2 The Transform component is not source-to-source

Since the Transform component takes CIR as input and yields CIR as output, and since the CIR is very close to the input source code, it may appear that Transform could really be implemented as a source-to-source translator. This is not actually true, however. The advantage of having Transform be part of the compiler rather than a pre-processor is that it can generate constructs which either cannot be expressed in the source language or could be expressed only by fighting the language. There is in principle an advantage in being able to generate these constructs, if later phases of the compiler can recognize them and handle them efficiently.

There are indeed some cases in which the output of Transform is not equivalent to Fortran code:

- Transform generates some operators, itemized in Chapter 18, which are specific to Transform and are never generated by the Front End. The purpose of generating these operators is to encapsulate certain operations which should not be exposed until later in the compilation process. (If they were exposed in Transform they would inhibit, or at least make much more difficult, various optimizations that Transform needs to make.) These operators could not easily be translated back into a source form; or if they were, the source would be even more unreadable than conventional source-to-source output is at present.

- Dope vectors are created by the Middle End, based on information left around by Transform. In order for these dope vectors to be set up properly, Transform has to "lie" a little—it has to change the values in these data structures so that the Middle End will generate correct dope vectors. This is in essence due to the fact that dope vectors need to contain information about the local layout of passed arrays (which the Middle End is prepared to deal with), and also about the global layout of passed arrays (which in principle the Middle End knows nothing about). So while what we are doing leads to correct generated code, the data structures after Transform might look inconsistent if viewed naively.

- Array-valued functions require still more special handling. (This is described in Section 7.3.) A special AFUNC operator has been introduced to convey information to the Middle End about special processing it needs to perform in this case. This operator is peculiar to the interface between Transform and the Middle End.

2.3 Program units and scoping units

The terms *program unit* and *scoping unit* are Fortran terms. The main driver of the compiler deals in program units: it first calls the Front End on each program unit, and then calls Transform on that same unit.

However, Transform does not operate on program units, but on scoping units; and there may be several scoping units within one program unit. (For instance, a module may contain several functions; or a main program may contain several internal subroutines.) Therefore, the driver for the Transform components walks the CIR tree for the program unit and calls Transform separately on each scoping unit within that tree. At the end of Transform, the CIR tree for that program unit is reconstituted as a single tree.

There is one small inaccuracy with this notation: when Transform processes a scoping unit, it has to be aware of any interface blocks within that unit. Such interface blocks technically constitute separate scoping units. Nevertheless, Transform is not called separately on interface blocks. Rather than define a new term, however, we shall merely say that Transform operates on scoping units, and note that this is almost, but not strictly speaking, true. Internally, such an "almost scoping unit" is represented as a `dt_psm` node (see page 37), where "psm" stands for "program, subroutine, or module".

2.4 Data structures used by Transform

The main data structures used by Transform are:

The symbol table This is the symbol table created by the Front End. It is extended by the Transform phases to include dope information for array and scalar symbols.

The dotree Although Transform accepts and emits CIR, the Transform phases themselves operate on a somewhat different data structure called the dotree. The dotree is really a fairly straightforward image of the abstract syntax tree produced by the front end.

The dependence graph This is a graph whose nodes are expression nodes in the dotree and whose edges represent dependence edges.

Data and iteration spaces A data space is associated with each data symbol. The data space information describes how each data object is distributed over the processors. This information is derived from HPF directives.

An iteration space is associated with each computational node in the dotree. The iteration space information describes how computations are distributed over the processors. This information is not specified in the source code, but is produced by the compiler.

2.5 The Transform phases

The Transform phase order is illustrated in Figure 2.2. The different Transform phases perform the following tasks:

CIR2DTR Translates the CIR to the dotree.

DATA Fills in the data space information for each symbol from the HPF directives.

LOWER Does some further translating. Creates dope information for each entry in the symbol table. Lowers FORALL statements. Inlines DATA statements and statement functions. Replaces SPREAD of a scalar by the scalar.

ITER Fills in the iteration space information for each computational expression node. This determines where each computation takes place, and indicates where motion is necessary.

ARG Pulls functions in the interior of expressions up to the statement level.

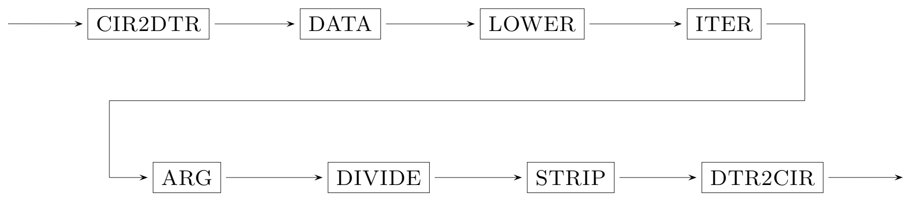


Figure 2.2: The Transform Phases

DIVIDE Pulls all motion inside expressions (identified by ITER) up to the statement level.

STRIP Turns global HPF code into explicit SPMD code by localizing the addressing of all data objects and inserting explicit SEND and RECEIVE calls to make motion explicit. In the process, performs strip mining and loop optimizations, vectorizes data motion, and optimizes nearest-neighbor computations.

DTR2CIR Translates the dotree back to the CIR.

There are two additional packages of routines which are not really phases but are used by the Transform phases:

DOTREE Utility routines, including the access package for the dotree data structure, and a dumper.

DEPEND Performs dependence analysis. The dependence analyzer is not described in this report. The algorithmic kernel of the dependence analyzer is based on Pugh's Omega test (see [16]), and is described in more detail in [15]

Each component of Transform is referred to in the compiler source code by a three-character abbreviation, as follows:

CIR2DTR	`c2d`
DATA	`dat`
LOWER	`low`
ITER	`itr`
ARG	`arg`
DIVIDE	`div`
STRIP	`stp`
DTR2CIR	`d2c`
DOTREE	`dtr`
DEPEND	`dep`

These abbreviations are used in the following ways:

- They are the names of the sub-directories where the files containing the code for these components live.

- They are prefixes for the names of the files containing the code for these components.
- They are prefixes for the names of global functions in these components.

For instance, the function `dat_make_sym_dope`, which creates the dope information for an entry in the symbol table, is contained in the file `dat_sizes.c`, which in turn lives in the sub-directory `dat/master`.

Chapter 3

The Run-Time Model

3.1 The mapping of data

Each data element is associated with a certain memory address in a certain processor. The user is able, by using data mapping directives, to specify the *mapping* of data objects over processors. This mapping is also referred to as *data layout*.

Data objects include not only arrays, but also scalars, which may be thought of as 0-dimensional arrays—this is consistent with the usage of the HPF directives. Thus, a scalar may "live on" (i.e., be mapped to) 1 processor, or may be totally or partially replicated over the processor array. All these possibilities can be expressed in HPF.

When the mapping of a data object is not specified, the compiler distributes the data according to a *default layout*. This default layout of data is not intended to be efficient; nor is it the intention of the compiler development team that the default layout should be under user control. HPF already provides in its directives a way for the programmer to completely specify data layout.

It is important that the programmer not rely on the default layout implemented in any release of the compiler, as this layout may change without notice. Further, there is not even a long-term guarantee that the default layout will be the same in all scoping units. While the default layout will be capable of being described in terms of HPF directives, it will not be proper for the user to try to describe the default layout as a way of getting the compiler to avoid data motion. The purpose of the default layout is simply to give the compiler a consistent way of laying out data when information is not provided to it in the source.

The default data layout implemented in this version of the compiler generally has each data element replicated over all the processors. Thus, if A is an array which is not specified in any directives, storage for a complete copy of A will be allocated on each processor, and the compiler will insure that corresponding elements of A on each processor always contain the same value.

Sequential variables (see 7.1.1(5) of the HPF Language Specification) are always given the default layout. In particular, this includes assumed-size arrays, as well as data declared to be sequential by means of the HPF SEQUENCE directive.

There are two exceptions to the default handling of unmapped data:

3.1.1 Privatized variables

Scalars which are used as loop indexes are privatized. That is to say, each processor has its own private version of the loop index, and these private versions are not kept in step with each other.

For this reason, the user should not apply data mapping directives to scalar variables used as loop indexes. If a loop index has such directives applied to it, the compiler will have to generate code to make sure that that index is replicated (or actually, privatized) over all the processors before any loop in which it is used—motion which really should be unnecessary.

Some other compiler-generated scalars are also privatized.

3.1.2 Data layout in PURE functions: the problem of callee ignorance

On page 3 we referred to an additional restriction (beyond those in the HPF report) we place on a function in order that it may be specified as PURE. The point of the restriction is this: normally a global HPF function is called on all processors. If the function is contained in a FORALL construct, however, and the iterations of that construct are to be distributed over the processors, then each iteration of the FORALL construct will call that function on only 1 processor (the processor executing that iteration). The code the compiler generates for this function, however, is the same as if it were to be invoked on all processors. Anthropomorphically, the function does not know that it is only being invoked on only 1 processor. We refer to this as "the problem of callee ignorance".

Calling a global function on only one processor can only work if all the data which that function references is available locally. So our restriction on PURE functions really amounts to saying that the compiler has the freedom to map the data accessed by the pure function so that it is all available where it is needed; i.e. that no data motion has to take place during the execution of the PURE function.

The precise way that the compiler remaps the data in a PURE function is specified later in this report.

3.2 The mapping of computations

3.2.1 The owner-computes rule

The owner-computes rule is used as a general guide to determine how iterations of parallel computations are distributed[1] over processors. Significant optimizations are made in the implementation of this rule, particularly by DIVIDE (see the discussion on page 86) and in processing nearest-neighbor computations (see Section 3.7.1).

3.2.2 Fortran DO loops

The iterations of a Fortran DO loop are distributed over processors provided the loop is marked as **!hpf$ independent**. The way we do this is specified in Chapter 11.

[1] The correct term is really "mapped", but "distributed" is often used as a synonym in this and other cases.

The iterations of a non-independent loop are not distributed. (That is, such a DO loop does not add to the dimensionality of the iteration space.) A non-independent DO loop is implemented as follows:

- Each processor owns its own copy of the loop index, and there is no communication generated when incrementing the loop index. That is, the loop index is a privatized variable.
- Each processor executes all iterations of the loop iteratively. In a typical case, this means that on any iteration of the loop only a few processors are active.

In any case, there is no synchronization generated between loop iterations: different processors may execute different iterations of the loop concurrently, subject only to whatever implicit synchronization is imposed by the sending and receiving of data between different processors.

3.2.3 Restricting the set of active processors

There are some conditions under which the compiler can determine that one or more processors will not participate in a computation. In such cases, the STRIP phase of Transform inserts IF guards where necessary into the generated code. When the code is embedded in a nest of DO loops, the IF guard is placed outside of the outermost loop. (See Section 12.3, page 129.)

In most cases, however, such IF guards will not be necessary. This is because the normal zero-trip loop tests will be sufficient to restrict the set of processors on which the loop bodies are executed.[2]

3.3 Control flow

Control flow is implemented as follows: the conditional expression for each IF in the source text must be available on all processors. In general, it is in fact available everywhere, simply because scalar variables and expressions are typically replicated over all processors. If it is not, however, a broadcast to all processors is generated. This then has the effect of inducing a barrier synchronization. For instance, the code

```
if (scalar)
  xxx(:)
```

would become (in the worst case, where the scalar was not already replicated)

[2]Using terminology to be defined below, an IF guard will be necessary around a DO loop nest only when the iteration space contains a non-trivial completion subspace.

```
processor holding scalar:
  broadcast its value

each processor:
  receive scalar
  if (scalar)
    do whatever part of xxx(:) is appropriate
```

3.4 Processor 0 as a distinguished processor

In general, there is no distinguished processor in the processor array. However, there are certain circumstances under which processor 0 is treated differently from the others.

3.4.1 Special intrinsics

Fortran intrinsics such as `DATE_AND_TIME` which could in principle return different numbers on different processors are executed only on processor 0, and the result is then propagated to any processor that needs it.

3.4.2 Entry points to global HPF routines

Since single processor routines execute only on processor 0, an HPF routine which is called from a single processor routine has to arrange to map any dummy arguments on entry. This is different from what happens when an HPF routine is called from another HPF routine (or is the main program).

To handle this correctly, every global HPF routine will be compiled to have two entry points for each entry in the source, and also two exits for each exit in the source. When such a routine is called from another global HPF routine, or is started up as the main program, the normal entry point and the normal exit will be used.

When a global HPF routine is called from a single processor routine (as could happen when the main program is a single processor routine), the alternate entry will be used. This will cause code to be executed that will move all passed parameters from processor 0 to their location as specified by data directives in the global HPF routine. Then the normal code of the routine itself will be executed. When the routine exits, an alternate exit will be taken which will cause code to be executed that will move all passed parameters back to processor 0.

3.4.3 I/O

All I/O is performed on processor 0. Any mapped data object which has to be written, therefore, is first moved by the compiler to processor 0, and then a write on processor 0 is generated. For data which is to be read, the data is first copied in to processor 0, read, and then copied back out. The copy in to processor 0 for reads is necessary because if there is an early exit from the read (e.g.

on an error), the copy out must still take place, and we want to make sure that we don't copy out garbage.

There is an optimization we make in performing I/O of replicated data. Of course replicated data which is to be output does not have to be moved to processor 0 since it is already there. Further, if replicated data is to be read, we do not generate a temporary on processor 0 to read it into followed by a broadcast of this temporary to the actual array. Instead, we read directly into the processor 0 copy of the array. Our data motion code then updates the rest of the array (i.e. on the other processors) directly from the processor 0 values (see the REPLICATE entry on page 103).

NAMELIST constructions need special handling, for the following reason: When a NAMELIST containing an array A is output, not only are the values of the individual elements of A output, the name "A" is as well. If A were copied to a temporary on processor 0 and then the NAMELIST processing were invoked, the name of the temporary would be output, rather than "A".

Fortunately, the run-time write routine called from the Middle End takes three arguments:

1. the base address in memory
2. the name to be used in a NAMELIST construction
3. the bounds to be read or written

So we can manage this by holding the name of the associated NAMELIST temporary (i.e. the temporary on processor 0) in a field in the dope for A (see Section 4.3). When I/O involves a NAMELIST, this temporary is used for item 1, and the array name A is used for item 2.

3.4.4 Memory on processor 0

Because of the way we handle I/O, it is important that processor 0 have enough memory to hold any arrays which we may need to move there. This will also be true (regardless of how the compiler handles I/O) if the main program starts on processor 0 and then calls an HPF routine, passing large arrays as parameters—those arrays will start out entirely on processor 0.

3.5 Punting

There are some constructs which this release of our compiler does not handle in a parallel manner. These include

- Array constructors. (However, many array constructors can be rewritten as simple FORALL constructs, which we can handle perfectly well.)
- Reduction intrinsics whose DIM argument is not known at compile time.
- The Fortran intrinsics PACK, RESHAPE, and UNPACK.

Transform handles such constructs essentially by remapping the data if necessary so that the Middle End can handle them entirely by itself. Since any data in such constructs (in order to be handled correctly by the Middle End) would have to be sequence associated, there are really only two layouts that we could use for this purpose:

- The default layout: all data is replicated.
- The layout in which all data lives only on processor 0.

With the exception of the Fortran intrinsics, we use the default layout. (There is no real reason for the exception; it is just that way because of the history of the implementation.) Precisely, we perform the following processing:

Any data involved in one of these constructs (e.g. the result of a reduction intrinsic whose DIM argument is not known at compile time) is given the default layout.

Statements involving these constructs are tagged as punted statements.

This processing happens in LOWER.

STRIP ignores these statements. They are thus passed through to the Middle End for processing; and since all the data in them has the default layout, it is sequentially allocated, and the normal Fortran processing can handle it.

3.6 Data motion: basic techniques

By data motion we mean potential interprocessor communication. Our compiler insures that all data motion occurs at the statement level. That is, an interior node of an expression which needs to be moved is moved and assigned to a temporary. A fetch of that temporary is then substituted in the expression for that node.

There are two reasons for doing this:

1. Motion inherently involves a change of context (i.e. a possible change of which processors are active). Since context is expressed as IF guards, it must be exposed at the statement level.
2. By exposing motion at the statement level, we can minimize the communication cost by, for instance, bundling up several sends to the same processor into one send. This second reason is of critical importance.

The language enables the programmer to specify several kinds of data motion. We can describe these motions in terms of source and target data elements:

1-to-1 Motion Each source element has a unique target element. Each target element comes from a unique source element. There are several kinds of such motion from the machine's point of view:

A Remapping Store In this kind of motion the sending processor knows what it will be sending and where it is sending to; and the receiving processor knows what it will be receiving (and could calculate where from). An example is a simple array assignment:

$$A(:) = B(:)$$

A Remote Store In this kind of motion the sending processor knows what it is sending, and where it is sending to, but the receiving processor doesn't know anything. An example is

$$A(V(:)) = B(:)$$

A Remote Fetch In this kind of motion the receiving processor knows what it needs to receive, and where it wants to get it from, but the sending processor doesn't know anything. An example is

$$A(:) = B(V(:))$$

1-to-Many Motion: A Partial Broadcast In this kind of motion the sending processor sends the same data to a number of processors. The sending processor knows what it will be sending and where to; and the receiving processors know what they will be receiving (and where they will be getting it from). This kind of motion is also sometimes referred to as a *multicast* operation.

Many-to-1 Motion: Reductions and Combining SCATTER functions This kind of motion is characteristic of reduction intrinsics, such as SUM, where one processor accumulates information contained on a number of other processors. Again, sending processors know where they are sending, and receiving processors know what they are receiving, and where it is coming from.

Another example of many-to-1 motion is that implied by a Combining SCATTER function (e.g. COPY_SCATTER). These routines can be handled by a variant of the way in which remote stores are handled; this is discussed in Section 14.2.

Many-to-Many Motion A statement such as

$$A(:) = \text{COPY\_SCATTER}(B(W(:)), 0, V(:), 0)$$

is equivalent to the Fortran statement

$$A(V(:)) = B(W(:))$$

except that it does not have the restriction that no element of A may be assigned to twice. This amounts to a non-deterministic store: many elements of B could be sent to a single element of A. Since also there is a vector-valued subscript W of B, a single element of B could be sent to a many elements of A. So this gives what could be described as many-to-many motion.

We now describe each of these cases in more detail:

3.6.1 A remapping store

We say that two nodes in the dotree are within the same *region* if on each processor, both nodes define the same number of elements, and if those elements correspond. This is the condition needed for the two nodes to be acted on in parallel. This condition implies that any array references which are represented by the nodes (or by children of those nodes) are aligned identically.

Actually, this definition of "same region" is not quite correct. The actual condition is that an expression node is in the same region as that of one of its children iff every value it needs from that child is available locally, i.e., with no interprocessor communication necessary to retrieve it.

If a parent and child in an expression tree are not in the same region, then motion will be necessary between them. It is the job of DIVIDE to raise this motion to the statement level.

Once the motion is exposed at the statement level, it may look like this:

$$A(:) = B(:),$$

where arrays A and B are laid out as follows:

Array A

processor number	1	2
$mem[A_{base} + 0]$		7
$mem[A_{base} + 1]$		8
	1	9
$\vdots$	2	10
	3	11
	4	12
	5	
	6	

Array B

	5	6	7	8
$mem[B_{base} + 0]$		4	8	12
$mem[B_{base} + 1]$	1	5	9	
	2	6	10	
$\vdots$	3	7	11	

where in each diagram, processors go across, and memory within a processor goes down.

Each processor holding elements of array B will calculate the corresponding target processor, and will generate a sequence of SEND instructions. Each such SEND will contain the following information:

1. The source processor
2. A sequence of data values.

Each receiving processor will compute which data elements are being sent from each sending processor, and will generate a sequence of RECEIVE instructions, one for each sending processor. Since the receiving processor knows how long each message will be, it can allocate the correct buffer space for each message. It then assigns each received data element to its proper address (which it has computed locally).

We denote the receiving buffer in each processor by *buf*; it should be thought of as a 0-based array. Note that the translation from global to local addressing has been performed in these SEND and RECEIVE statements: this translation happens in the STRIP phase of Transform. Also note that the base address of the array is the same on each processor, even if there is no array element stored at that local address.

Processor 1

- RECEIVE $buf[0:2]$ from processor 5 (buffer length = 3);

$$mem[A_{base} + 2 : A_{base} + 4] \leftarrow buf[0:2]$$

- RECEIVE $buf[0:2]$ from processor 6 (buffer length = 3);

$$mem[A_{base} + 5 : A_{base} + 7] \leftarrow buf[0:2]$$

Processor 2

- RECEIVE $buf[0]$ from processor 6 (buffer length = 1);

$$mem[A_{base}] \leftarrow buf[0]$$

- RECEIVE $buf[0:3]$ from processor 7 (buffer length = 4);

$$mem[A_{base}+1 : A_{base}+4] \leftarrow buf[0:3]$$

- RECEIVE $buf[0]$ from processor 8 (buffer length = 1);

$$mem[A_{base}+5] \leftarrow buf[0]$$

Processor 5

- SEND to processor 1 $(mem[B_{base}+1 : B_{base}+3])$;

Processor 6

- SEND to processor 1 $(mem[B_{base} : B_{base}+2])$;
- SEND to processor 2 $(mem[B_{base}+3])$;

Processor 7

- SEND to processor 2 $(mem[B_{base} : B_{base}+3])$;

Processor 8

- SEND to processor 2 $(mem[B_{base}])$;

The compiler does not generate a SEND from a processor to itself.

3.6.2 A remote store or combining SCATTER

There are at least four ways of implementing a remote store:

1. The source processor sends a message to the target processor. The target processor has no way of knowing that it is expecting a message, so the message must interrupt the target processor. This requires synchronization before the send, and also some synchronization after. The "after" synchronization could be delayed until the stored value is actually needed, however. The cost is $O(p)$ messages (where p is the number of processors) in typical cases, plus the cost of synchronization.

2. The source processor sends a message to every other processor with information about what that particular processor is to receive—the message is empty unless the source processor is intending to send something to that processor. Every processor receives that message and then sets up to receive what (if anything) the source processor is going to send. The receive could be an asynchronous receive, giving the same effect as delaying the final synchronization in the first method. The advantage of this method is that there is no additional synchronization. The cost, however is $O(p^2)$ messages in typical cases. It therefore appears that this method suffers from the drawback that it will not scale well—that contention will become significant as the number of processors grows.

3. As a variant of method 2, the source processor broadcasts to all processors all the information about what it will send to each one. (That is, the same message is sent to all processors.) The receiving processors unpack this information and find out what part of it applies to them, then set up to receive if appropriate. Again, no additional synchronization is required. The cost is $O(p \log p)$ messages in typical cases (because the broadcast can be optimized). This method (and method 2) can be optimized by including the data to be sent in the original message. This does not affect the relative costs, however. A disadvantage of this method is that each processor has to allocate $p-1$ buffers to hold incoming messages, and this space can be quite large. In addition, since $p-1$ large messages will be coming in at once, there is a danger that the operating system's input buffers will become overloaded.

4. A fourth method is to imagine the processors as arranged in a ring, and have each processor send the information that it would have sent in method 3 to its neighbor "on the right". Any message a processor receives from its neighbor "on the left" is examined to extract any information intended for that processor, and is then passed along to the neighbor on the right, until each message has had a chance to go around the entire ring. The advantage of this method is that there is no contention. The cost is $O(p)$ in time (since we have to allow time for each message to go around the entire ring). This compares unfavorably with method 3, which optimally would have a cost in time of $O(\log p)$. (Of course there is some contention in method 3, so this is probably not really attainable.) On the other hand, only one buffer needs to be allocated per processor, so memory use is not as great, and there is no danger of the operating system's input buffers becoming overloaded.

We are not pursuing method 1 for two reasons:

- There are no current primitives which implement interrupting sends or receives.
- Even if there were, the need to put in additional explicit synchronization would complicate the design.

Method 2 we regard as too expensive. We are currently implementing method 4, although we do not know how it actually would compare on this machine with method 3.

Given the actual interconnection of the different processors in our target machine, it does not seem to be significant how we determine each processor's neighbor on the right. Therefore, the processor on the right of a given processor is simply implemented as that processor whose processor id is one more than that of the given processor (with wrapping around). That is, the processor id of the processor on the right is just

$$(\mathit{my\_processor}() + 1) \bmod P$$

where P is the total number of processors.

Combining SCATTER library routines are handled as a variant of remote stores; see Section 14.2.

3.6.3 A remote fetch

Remote fetches are a little more complicated than remote stores, because the fetching processor has to send a request to the source processor and then receive the information. Thus, the optimization mentioned in method 3 above cannot be used. We are implementing a method which is similar to method 4 for remote fetches:

Each processor sends to the processor "on its right" all the information about what it wants to receive from every other processor. Each processor examines the message it receives, finds the fields in that message which ask for data from it, and fills in the requested data in those fields. It then sends the complete message on to the next processor on its right. After p such steps, each message is back at the original processor, with all the requested data filled in. Each processor then extracts that data and stores it at the proper address.

3.6.4 A partial broadcast

Partial broadcasts are used to implement spreads, which may be explicit in the source text or generated by the compiler. A partial broadcast for us is actually a little more general than a Fortran SPREAD, in that we allow broadcasting into more than 1 dimension. A typical partial broadcast might look like this:

$$\mathrm{PBRDCST}(A(:,:,:,:), \mathrm{DIMS} = \{1, 3\}, B(:,:))$$

by which we mean that B (which in general would actually be an expression) is being spread into the first and third dimensions of A. Of course this means that the first and second dimensions of B would conform to the second and fourth dimensions of A, respectively.

Each processor holding values of B calculates for each such value the processors to which that value is to be sent (using information about the mapping of A) and generates the appropriate SENDs. Each processor holding values of A similarly calculates the processors holding the corresponding values of B which are to be sent to it, and generates the appropriate RECEIVEs. Since these calculations happen before any SENDs or RECEIVEs are generated, the SENDs and RECEIVEs can be packaged up together so that between each pair of processors there is at most one SEND and one RECEIVE.

It turns out that the fundamental distinction in dealing with motion code is not between 1-1 and 1-many, but is between kinds of motion classified by what the sender and receiver each know.

As a consequence, the implementation of code for a partial broadcast is really the same as that for a remapping store, and therefore there is no PBRDCST operator in the dotree. The operator is simply a remapping store operator, and the broadcast behavior of the operation will be encoded in the iteration space structures of the store operator and the expression for the right-hand side of the statement.

There is, however, a multicast primitive in the message-passing library called by the compiler, and at run-time, this primitive is used to optimize remapping stores where the same element is sent to more than one destination.

3.6.5 Reductions

Let us consider the simple case

$B = \mathbf{sum}(A,\ \mathbf{dim} = d)$

where A is an n-dimensional array and B is an $n-1$-dimensional array. Conceptually, this is what happens:

- If B does not align with A in all dimensions except dimension d, then the sum is computed into an array T which has that property. Then T is remapped into B. So let us assume that B is so aligned.
- On each processor, there will in general be several elements of B. Corresponding to each of those elements, there will be a collection of elements of A *on each processor* whose sum must be accumulated into that element of B. We could accomplish this by having each processor accumulate those elements into a privatized scalar temporary S, and then have a partial global reduction performed to accumulate each copy of S into the element of B.

  Actually, however, we do this in parallel: We create an n-dimensional global temporary array S, the sections of which on each processor contain one element for each element of B which the values of A on that processor have to be accumulated into. Then each processor accumulates all its elements of A locally into the appropriate element of S, after which a global partial reduction is performed to accumulate S into B.

3.6.6 Many-to-many motion

Statements such as

$A(:) = \text{COPY\_SCATTER}(B(W(:)),\ 0,\ V(:),\ 0)$

are implemented by breaking them apart, effectively into the pair of statements

$T(:) = B(W(:))$
$A(:) = \text{COPY\_SCATTER}(T(:),\ 0,\ V(:),\ 0)$

3.6.7 An example: vector-valued subscripts

As an example, let us consider the problem of generating code for vector-valued subscripts. This problem will be taken up again later in this report in more detail.

Expressions with vector-valued subscripts can of course be arbitrarily complicated. We consider the following example, which contains the essential kernel of what we have to do:

$$A(V(:), W(:)) = B(:,:)$$

The right-hand side of this statement is computed into what might be described as an anonymous 2-dimensional array C. (In this simple case, C would be laid out just as B is, but in general, the right-hand side would be more complex.)

A one dimensional example would be even simpler:

$$A(V(:)) = B(:)$$

Here is what the compiler does with these two cases:

1. In the one-dimensional example, it makes sure that V is aligned with C, and moves it if it is not. In the two-dimensional example, it performs a partial broadcast of V to both align it with the first dimension of C and spread it parallel to the second dimension of C. The

processors receiving this partial broadcast can compute the fact that they are due to receive a value of V and generate the appropriate receive.

2. In the two-dimensional example, similarly, it performs a partial broadcast of the values of W to both align it with the second dimension of C and spread it parallel to the first dimension of C.

3. Thus, each processor "holding a value of C" receives its appropriate value of V (and of W in the two-dimensional example). These values are used (together with the addressing information—known as data space information—of the array A) to generate an address to which the value of C is sent. This information is packaged up and sent around the ring of processors as outlined in the section on remote stores. Each receiving processor is able to retrieve the information intended for it from each subsequent message, and store it at the proper location.

3.6.8 Scalars

Motion for scalars will be driven by the same machinery (i.e. the data and iteration space structure) as motion for arrays, and similar SEND and RECEIVE operators will be generated.

3.7 Data motion: additional optimizations

3.7.1 Nearest-neighbor computations

A nearest-neighbor computation is a vector assignment statement which is almost parallel in that the individual array references contain subscripts which differ by very little so that only a small amount of communication is necessary. An example is the usual 4-point average computation:

$$
\begin{aligned}
A(2{:}N{-}1,\ 2{:}N{-}1) \quad = \quad & 0.25 * (A(1{:}N{-}2,\ 2{:}N{-}1) \\
& + A(3{:}N,\ 2{:}N{-}1) \\
& + A(2{:}N{-}1,\ 1{:}N{-}2) \\
& + A(2{:}N{-}1,\ 3{:}N))
\end{aligned}
$$

A naive implementation (using the owner-computes rule) would create four temporaries for the array sections on the right-hand side. Motion would be generated to move the proper values of these arrays into these temporaries. While in this case, only those elements which need to be moved between processors would be so moved, a large and unnecessary amount of memory would be used in creating these temporaries, and there would also be a large and unnecessary amount of local motion involved in moving most of the array sections on the right locally into the appropriate temporaries.

A more complicated, but still realistic example, would contain even more slices of A on the right hand side. Again, a naive implementation would generate a motion statement and a temporary for each of these slices. But in general, there is high degree of overlap between these slices, and much less motion really needs to be generated. We can do much better than the naive implementation.

We have two goals in generating code for such computations:

1. Move only the data that has to be moved, and move it only once.
2. Bundle up the motion so that between each pair of processors there is only one SEND and only one RECEIVE.

How we do this is explained in detail in Chapter 15. The general idea is this:

1. We recognize certain statements as nearest-neighbor computations. (The programmer does not have to identify these statements in any way—the compiler will recognize them automatically.) The same-region check is suppressed for such statements.
2. We compute exactly what data has to be moved. The data that has to be moved corresponds to what we call "shadow edges" around the sections of the left-hand array reference. We allocate additional space locally around the left-hand side array in which to store these shadow edges. This changes the addressing calculations for the left-hand array.
3. We generate motion to load the shadow edges into place.
4. Once the shadow edges are loaded into place, all computations are local, and we generate code to express them.

3.7.2 Irregular data accesses

An irregular data access always amounts to a remote fetch or a remote store. We can further optimize such data accesses when the following two conditions hold:

- The access occurs more than once in a scoping unit. Typically this means the access occurs inside a loop.
- The pattern of communication is the same for all these instances of the data access. For instance, this happens when the data access is through a vector-valued subscript, and when the indirection vector is unchanged between successive accesses.

An example is shown in Figure 3.1.

---

call setup(U, V, W)
do $i = 1$, n_time_steps, 1
    A = **sum_scatter**($B(W(1{:}n))$, A, $V(1{:}n)$, **mask** = .true.)
    C = **sum_scatter**($D(W(1{:}n))$, C, $V(1{:}n)$, **mask** = .true.)
    E = **sum_scatter**($F(W(1{:}n))$, E, $V(1{:}n)$, **mask** = .true.)
end do

Figure 3.1: Code with Reusable Indirect Accesses

In this case, we do not have to use the whole machinery of remote fetches or stores each time through the loop: instead, we can generate code to precompute the data paths once, at the top of the loop. Then each irregular data access inside the loop can use this precomputed information to generate at most one send and one receive between each pair of processors. Thus, in effect, we pay the price of the indirection once, and then each data access becomes equivalent to a simple remapping store.

The details of this are laid out in Section 13.4.

3.8 HPF intrinsic and library functions

The implementation of these functions is as follows:

Elemental Intrinsic Functions. These include the usual elemental intrinsics such as SIN; and also the HPF elemental bit manipulation funcions ILEN, POPCNT, POPPAR, and LEADZ. These functions appear in the dotree as operators and are handled no differently from ordinary vector operators such as vUMINUS, for example. They are stripped and ultimately reduced to scalar operators which will be turned into local function calls or inlined code on each node by the code generator. (Actually, the stripped operators may not actually be scalar, but they will refer to operations on only one processor. This is entirely analogous to generating a vector PLUS operation on local data, which we may do in a future release.)

System Inquiry Intrinsic Functions

NUMBER_OF_PROCESSORS

At the beginning of the generated code, an internal `numnodes` variable is created and the number of processors on which the program is executing is stored into this variable. Of course if the program is compiled for a specific number of processors, the compiler knows this and makes use of this information.

NUMBER_OF_PROCESSORS() is turned into a FETCH of this variable (or a constant if the compiler knows its value).

PROCESSORS_SHAPE

This turns into a reference to a 1-dimensional array of size 1 whose only element holds a FETCH of the `numnodes` variable mentioned above, or a contstant if the number of processors is known at compile time.

Mapping Inquiry Subroutines

HPF_ALIGNMENT

HPF_DISTRIBUTION

HPF_TEMPLATE

These three intrinsics are passed as operators to the Middle End, which turns them into calls to routines in the HPF run-time library.

Array Reduction Functions

ALL

ANY

COUNT

IALL

IANY

IPARITY

PARITY

MAXLOC

MAXVAL

MINLOC

MINVAL

PRODUCT

SUM

These functions are inlined by the compiler.

Array Combining Scatter Functions

XXX_SCATTER

Here XXX can be replaced by the name of any of the reduction intrinsics. The XXX_SCATTER functions are inlined by the compiler.

Array Prefix and Suffix Functions

XXX_PREFIX

XXX_SUFFIX

Here XXX can be replaced by the name of any of the reduction intrinsics. These functions are turned into library calls.

Array Sort Functions

GRADE_UP

GRADE_DOWN

SORT_UP

SORT_DOWN

These functions are turned into library calls.

Computational Intrinsics

DOT_PRODUCT

MATMUL

RANDOM_NUMBER

These functions are inlined by the compiler.

Routines that move data

CSHIFT

EOSHIFT

SPREAD

TRANSPOSE

These functions are inlined by the compiler.

PACK

RESHAPE

UNPACK

These functions are implemented as follows: all their arguments are replicated, and the intrinsic Fortran routine is called on each processor. The result is then remapped over the processor array. This is an example of *punting* (see Section 3.5, page 20).

There is one exception to this specification: intrinsic functions are automatically PURE in HPF. If such a function is called within a FORALL construct it is treated as any other PURE function. In particular, it generally becomes transformed by the compiler to a library call on a single processor. See page 97 for an example of this.

3.9 Storage allocation

3.9.1 Implicit allocation

Fortran traditionally has used static data allocation. This has the advantage of allowing fast addressing, but cannot be used for data that is local to recursive subroutines. Since Fortran includes such subroutines, the compiler allocates data statically or on a stack as appropriate.

This data allocation is handled completely by the Middle End and GEM; Transform sees no difference in the representation of references to statically or dynamically allocated arrays.

3.9.2 Explicit allocation

In addition, the language includes ALLOCATE and DEALLOCATE statements, which have the semantics of heap allocation. These are represented in the CIR by opALLOC and opDEALLOC operators.

Further, Transform itself creates short-lived temporaries. These are always stack temporaries. The dotree operators vaNEWSCOPE, vaSALLOCATE, and vaENDSCOPE manage stack allocation of these temporaries. For more details, see Section 4.2.3.

3.10 Argument passing: dope information

Here we describe generally how and where dope vectors are created. Transform does not actually create these dope vectors; the Middle End, which runs after Transform, will create them. But Transform has to leave enough information around so that the Middle End will be able to do this.

The contents of a dope vector are detailed below, in Chapter 4, Section 4.6.

3.10.1 Calls from global routines

At each call site of a global subroutine, each processor computes a dope vector for each actual argument passed assumed-shape. All other actual arguments are passed by reference.

If the called routine is also a global routine, an identical copy of this dope lives on each processor. (That is, the dope is replicated across all the processors.)

If the called routine is an HPF_LOCAL routine, then each actual argument corresponding to a dummy array argument must be passed assumed-shape. This entails passing a dope vector on each processor. The dope information is in general *different* on different processors. Here is an example which shows the kind of phenomenon which can occur:

Suppose we have an array of 16 processors, and an array $A(50, 50)$ which is distributed (BLOCK,BLOCK) in such a way that the *strip_length* of each dimension is 4. The array will be mapped over the processors as shown in Figure 3.2.

$A(1:13,\ 1)$ $A(1:13,\ 2)$ $\vdots$ $A(1:13,\ 13)$	$A(1:13,\ 14)$ $A(1:13,\ 15)$ $\vdots$ $A(1:13,\ 26)$	$A(1:13,\ 27)$ $A(1:13,\ 28)$ $\vdots$ $A(1:13,\ 39)$	$A(1:13,\ 40)$ $A(1:13,\ 41)$ $\vdots$ $A(1:13,\ 50)$
$A(14:26,\ 1)$ $A(14:26,\ 2)$ $\vdots$ $A(14:26,\ 13)$	$A(14:26,\ 14)$ $A(14:26,\ 15)$ $\vdots$ $A(14:26,\ 26)$	$A(14:26,\ 27)$ $A(14:26,\ 28)$ $\vdots$ $A(14:26,\ 39)$	$A(14:26,\ 40)$ $A(14:26,\ 41)$ $\vdots$ $A(14:26,\ 50)$
$A(27:39,\ 1)$ $A(27:39,\ 2)$ $\vdots$ $A(27:39,\ 13)$	$A(27:39,\ 14)$ $A(27:39,\ 15)$ $\vdots$ $A(27:39,\ 26)$	$A(27:39,\ 27)$ $A(27:39,\ 28)$ $\vdots$ $A(27:39,\ 39)$	$A(27:39,\ 40)$ $A(27:39,\ 41)$ $\vdots$ $A(27:39,\ 50)$
$A(40:50,\ 1)$ $A(40:50,\ 2)$ $\vdots$ $A(40:50,\ 13)$	$A(40:50,\ 14)$ $A(40:50,\ 15)$ $\vdots$ $A(40:50,\ 26)$	$A(40:50,\ 27)$ $A(40:50,\ 28)$ $\vdots$ $A(40:50,\ 39)$	$A(40:50,\ 40)$ $A(40:50,\ 41)$ $\vdots$ $A(40:50,\ 50)$

Figure 3.2: A 50×50 array distributed (BLOCK,BLOCK) over 16 processors.

Now the processor "at the upper left" contains an array which looks like a 13×13 array, and this array is laid out sequentially in that processor's memory. The processor "at the lower left", however, contains what should be an 11×13 array, except that there are gaps in the processor's memory. For this reason, the dope vector that is passed to an HPF_LOCAL subroutine includes an access function, the coefficients of which reflect the actual mapping of the array in the local processor's memory.

In this case, the local dope vector for the processor at the lower left contains the following information (all contained in the F90 ARRAY SECTION of the dope vector; see Section 4.6):

`addr_a0` $= base$

dimension 1

> `lower_bound` $= 1$
>
> `upper_bound` $= 11$
>
> `inter_element_spacing` $= 1$

dimension 2

$\texttt{lower\_bound} = 1$

$\texttt{upper\_bound} = 13$

$\texttt{inter\_element\_spacing} = 13$

Thus, that processor would compute the local memory address of the local array element which it knows as $A(i, j)$ to be

$$\begin{aligned}\texttt{addr\_a0} + \texttt{inter\_element\_spacing}_1 * (i - \texttt{lower\_bound}_1) \\ + \texttt{inter\_element\_spacing}_2 * (j - \texttt{lower\_bound}_2) \\ = \textit{base} + (i-1) + 13 * (j-1)\end{aligned}$$

(Just to be absolutely clear, note that the local array element $A(1, 1)$ on this processor is really the global array element $A(40, 1)$.)

Passing this dope vector enables the HPF_LOCAL subroutine to generate correct addressing for each local array section, and saves us from generating code in the global HPF caller to perform a local remapping prior to calling the HPF_LOCAL subroutine.

Arrays which are distributed CYCLIC(n) in any dimension cannot be passed assumed shape to an HPF_LOCAL subroutine. (See for instance the picture on page 134 for an example of what can happen. The part of that array on each processor has no simple description even in terms of our extended dope structure.) Therefore, if such an array is to be used in a HPF_LOCAL function, it either has to be remapped first by the calling function, or the user will have to write a loop of calls, one call for each partial BLOCK of the array. In effect, this passes a BLOCK distributed array section on each iteration of the loop.

Dope information is the only way we have of passing information about mapped actuals to the called routine.

3.10.2 Calls from HPF_LOCAL routines

At each call site of an HPF_LOCAL subroutine, each processor computes a dope vector for each actual argument which needs it. The information in such a dope vector will differ from processor to processor.

Chapter 4

The Transform Data Structures

With the exception of the Common IR, the main data structures in Transform are constructed using tools originally developed at Massachusetts Computer Associates (Compass, Inc.). In particular, these include

ldo A specification language used for describing recursive data structures such as parse trees or more general graph structures. The structures are composed of *nodes*. Each node has *components* (which can be thought of as the children of the node if the structure is a tree) and *attributes*. The language is described in [13].

dtb A source code generator that builds an access package for a data structure specified in ldo. The nature of this package is described in [12]. Since dtb is used to create the access package, the data structure is often referred to as a "dtb data structure", even though its specification is written in the ldo language.

4.1 The Common IR and the symbol table

Unlike the other main data structures in Transform, the Common IR is not a dtb data structure. It is a family of ordinary data structures in C, with a hand-written access package. The data structure itself is defined in the file `cir/cir_tables.h`. The CIR operator names themselves (e.g., opPLUS) are enumerated in the file `cir/cir_op.h`.

The general Common IR node is called a `pIlNode` ("pointer to an intermediate language node"). Terminal pIlNodes are known as *tokens*. Each token is one of the following types:

SYMTOK A symbol.

TMPTOK A compiler-generated temporary. Actually, these are only generated by the Front End. TRANSFORM generates SYMTOKs when it needs to create temporaries.

CONTOK A constant occurring in the source text.

VALTOK An immediate constant generated by the Front End.

LABTOK A label from the source text.

EDTTOK An edit descriptor, from a FORMAT statement.

NAMTOK A name which is in the source text but is not inserted in the symbol table because its scope is so small; used for instance for dummies in ALIGN directives.

KEYTOK Used for keywords for optional parameters in OPEN and INQUIRE statements.

A SYMTOK is actually a `pIlNode` with the operator `opSYMTOK`, and so on.

The CIR is built mainly as a binary tree. This is for historical reasons: the first implementation of the CIR was on a machine where memory was limited, and keeping the size of a general CIR node small was important. Since most CIR nodes in fact had 1 or 2 children, this was a reasonable design decision. It does mean, however, that the representation of ternary or list structures is somewhat convoluted. There are some exceptions to this: some nodes (such as opFAINDEX) which have been added more recently are ternary nodes.

There are actually a number of tables which collectively are referred to as the Symbol Table:

Table	Entry	Corresponding CIR node	Use
Symbol Table	`pSymNode`	SYMTOK	symbols
Temp Table	`pTmpNode`	TMPTOK	temporaries
Constant Table	`pConNode`	CONTOK	literal constants
Label Table	`pLabNode`	LABTOK	labels (i.e. targets of jumps)
Edit Descriptor Table	`pEdtNode`	EDTTOK	edit descriptors
Name Table	`pNamNode`	NAMTOK	e.g. ALIGN dummies
Keyword Table	`pKeyNode`	KEYTOK	a few keywords
Type Table	`pAtyNode`	—	types (real, complex, etc.)

In this report, the phrase *symbol table entry* generally refers to a `pSymNode`.

Each symbol table entry of course has many fields. One which is used often in transform is a table of “handy bits”. Each handy bit specifies whether the symbol has a particular attribute. Some common handy bits are

btALLOCATABLE An allocatable array.

btASHARY An assumed shape array.

btAUTOMATIC An automatic data object.

btESHARY An explicit shape array.

btNEAR_NEIGHBOR Array is a nearest-neighbor array (see Chapter 15); in particular, it has shadow edges.

btNOSEQUENCE Array has the NOSEQUENCE attribute.

btNUMA_STACK Array is allocated on the numa stack (see Section 16.9.1).

btPOINTER90 Symbol has the POINTER attribute.

btSAVE Symbol has the SAVE attribute.

btTARGET Symbol has the TARGET attribute.

There are currently several hundred handy bits. They are defined and documented in the file `cir/cir_handy.h`.

There are no CIR nodes corresponding to entries in the type table—these entries are attributes of CIR nodes. There is no table to manage VALTOKs, since these represent immediate data which is completely represented in inlined executable code.

Attached to each `pSymNode` are a large number of fields. Among these is a field containing dimension information. This field, used for arrays, contains CIR expressions which represent the lower bound and the extent of the array in each of its dimensions.

4.2 The dotree

The dotree is specified as a recursive data structure by an ldo description (`dtb/master/dotree.ldo`) and accessed by a package of routines produced from this description by dtb. In the following description, the terms *child* and *attribute* refer to dtb usage. Dotree nodes may be *structural* nodes or *expression* nodes. Expression nodes cannot be children (in the sense that they would be dtb components) of structural nodes—they can only be dtb attributes of such nodes. However, they may be thought of as children, and for the purposes of this discussion are referred to as such. That is to say, no real distinction is made between dtb components and attributes in this discussion.

In the ldo specification, a structural node is a `do_tree_node` and an expression node is an `acomp` (i.e., *abstract computation*). `acomp` is actually a disjunction of two node classes: `VC` (*value computation*) and `FL` (*flow computation*). These in turn are disjunctions of actual node types, which are described below.

4.2.1 Structural nodes

The structural nodes form a tree whose root is a `dt_psm` node and whose leaves are `dt_astmt` nodes. The different kinds of structural nodes are:

`dt_psm` A node representing a program, subroutine, or module. This corresponds to the Fortran notion of a "scoping unit", except for the modification noted previously in section 2.3 (page 12).

The root of the dotree is a `dt_psm` node, but these nodes can also occur elsewhere in the dotree.

This node has one child:

`tree` A structural node which represents the executable code in the dotree. This node must be a `dt_seq` node.

In addition, this node has four specific attributes:

`name` A `pSymNode` holding the declared name of this program, subroutine, or module.

`cir` The `pIlNode` that is the root of the CIR tree of which this is the dotree translation.

`entries` A list of structural nodes that specifies all the entries to the psm.

`exits` A list of structural nodes that specifies all the exit nodes of the psm.

`dt_seq` Has one child which is a list of structural nodes. The flow of control is from one node to the next. Thus, there can be no jumps out of a `dt_seq` node except from the last node in the list, and no jumps into a `dt_seq` node except to the initial node of the list.

`dt_astmt` An "abstract statement". This node has no children per se. However, it has an attribute which is a statement root; for instance, a STORE operator. Every node below a `dt_astmt` node is an expression node.

Each `dt_astmt` also has a `stmt_kind` attribute describing the kind of statement it is. This attribute triggers different kinds of processing by different phases of TRANSFORM. The complete list of values for this attribute is given on page 102.

`dt_simple_if` Has three children:

`cond` An expression node, which must evaluate to TRUE or FALSE.

`then_node` A `dt_seq`, representing the code executed if `cond` is TRUE.

`else_node` A `dt_seq`, representing the code executed if `cond` is FALSE.

`dt_dowhile` Has four children:

`cond` An expression node, which must evaluate to TRUE or FALSE. This node represents the test at the head of the loop.

`prolog` A structural node. Initially empty. This is a place where loop-invariant code can be moved to before the first iteration of the loop.

`body` A `dt_seq`, representing the body of the loop.

`epilog` A structural node. Initially empty. This is a place where loop-invariant code can be moved to after the last iteration of the loop.

`dt_case` Has three children:

`expression` An expression node, the value of which is the value switched on.

`selectors` A table of expression nodes, representing the possible values of `expression`.

`blocks` A parallel table of structural nodes, representing the corresponding code.

`dt_tangle` Has one child, which is a list of structural nodes. Unless there is explicit flow of control in these nodes, control passes from one node to the next in the list. Thus, this node is a weak form of a `dt_seq`, but can be used in its full generality to represent any otherwise unclassifiable structure.

`dt_do_loop` Has three children:

`prolog` A structural node. Initially empty. This is a place where loop-invariant code can be moved to before the first iteration of the loop.

`body` A `dt_seq` representing the body of the loop.

`epilog` A structural node. Initially empty. This is a place where loop-invariant code can be moved to after the last iteration of the loop.

When code is generated for a dt_do_loop, a zero-trip test is inserted at the top of the loop. This is just a bounds check to see if there are any valid iterations of the loop to perform. The `prolog` represents code which is executed once, before the first iteration of the loop body, if the zero-trip test is true (i.e. if the loop is actually entered). The `epilog` represents code which is executed once, after the last iteration of the loop body, if the zero-trip test was true. If the zero-trip test is false, none of the code in the `prolog`, `epilog`, or `body` is executed.

A `dt_do_loop` is guaranteed to have no explicit (via GOTO statements) or implicit (via error exits of I/O statements) exits from the body other than by the test at the bottom of the loop.

In addition, a `dt_do_loop` has the following attributes, all of which are expression nodes:

`do_variable` The name of the do loop index.

`initial_bd` Expression for the initial bound of the loop.

`terminal_bd` Expression for the terminal bound of the loop.

`increment` Expression for the increment of the loop.

The initial assignment of the `initial_bd` to the `do_variable`, and the test and increment of the `do_variable` using the `terminal_bd` and the `increment` are not represented in the dotree. (They are made explicit after the Transform component of the compiler, in the Middle End.)

There are many other attributes as well; in fact, the attributes of do loops are the most rapidly changing part of the data structures of TRANSFORM.

`dt_loop` A `dt_loop` node is structurally the same as a `dt_do_loop` node, with the exception that it cannot be guaranteed that there are no additional exits from the body. Both nodes (as they come into Transform) come from DO loop constructions in the source code. Transform creates additional `dt_do_loop` nodes in the process of strip mining, but no more `dt_loop` nodes.

`dt_where` Has three children:

`mask_expr` An expression node representing a mask (i.e. boolean parallel data).

`true_stmts` A `dt_seq` node representing a sequence of parallel assignment statements, each conformant with the expression node.

`false_stmts` A `dt_seq` node, similar to `true_stmts`, representing the parallel assignments under the ELSEWHERE clause.

`dt_forall_block` Comes from a FORALL construct in the source. Has two children:

`forindx` An expression node whose operator is vaFORINDXS, having two children. Its first child is a list representing the list of FORALL indexes and their bounds in the FORALL header. Its second child represents the mask expression in the FORALL header.

`stmts` A `dt_seq` node representing the executable code.

`dt_group_forall` Comes from a FORALL construct in the source where the body of the FORALL contains an array-valued PURE function. It has one child, which is a list of structural nodes, each of which is a `dt_astmt`. This node is created where necessary by ARG, and is deleted by STRIP.

`dt_io` Represents an I/O operation. This is the top node of such an operation. It has two attributes:

`op` The operator name. For example, this could be vaSFREAD.

`cond` An expression node representing a Boolean guard condition. This typically represents the condition "if *my_processor*() = 0".

It has four children:

`pre_stmts` A `dt_seq` node. This node is empty unless error exits are specified in the source code, in which case generated code is inserted here to manage that exception handling. See Section 9.2.

`specs` A `dt_seq` node containing a list of `dt_io_item` nodes (see below). Each of these nodes is either NULL or represents a specification (e.g., UNIT number, label of FORMAT statement, IOSTAT variable, etc.).

`items` A `dt_seq` node containing a list of actual arguments (e.g., to be read or written). Each of these arguments is represented as a `dt_io_item` or a `dt_io_loop`, both of which are described below.

`post_stmts` Either a `dt_seq` or a `dt_tangle`. It is normally empty unless error exits are specified in the source code. See Section 9.2.

`dt_io_loop` This node is used to describe an implied do loop in an I/O statement. It is one of the `items` of a `dt_io` node. It has some attributes, of which the most important ones are the four obvious ones needed for any loop:

`do_variable` An expression node representing a `pSymNode` or `pTmpNode` naming the loop variable of the implied do loop.

`initial_bd` An expression node representing the initial bound of the implied do loop.

`terminal_bd` An expression node representing the terminal bound of the implied do loop.

`increment` An expression node representing the increment of the implied do loop.

It has three children, which are entirely similar to the children of the same name of a `dt_io` node. Each is a `dt_seq` node:

`pre_stmts` Copy-in code (if any) for the loop bounds.

`items` A list of `dt_io_item` nodes specifying the values being written or read.

`post_stmts` Copy-out code (if any) for the loop bounds.

`dt_io_item` This node is used in the two previous I/O nodes to hold the actual values being written or read. As above, it has two children to manage any temporaries that may be necessary:

`pre_stmts` Copy-in code (if any) needed for the value being read or written.

`post_stmts` Copy-out code (if any) needed for the value being read or written.

And finally, there is an attribute:

`item` The value being written or read. Since this is an expression node, it cannot be a child, so it is an attribute.

`dt_entry` Represents an entry node of the scoping unit. Has two children;

`preamble` A `dt_seq` which contains initialization code inserted by the Front End, and which may be added to by Transform.

`comp` An expression node holding the entry statement itself. (If there is no ENTRY statement in the source, this would be the PROGRAM or SUBROUTINE statement, or the equivalent.)

`dt_exit` Represents an exit node of the scoping unit. Has two children:

`preexit` A `dt_seq`, currently empty, but reserved for possible use to hold exit code.

`comp` An expression node, holding the EXIT, RETURN, or STOP statement.

`dt_hpf_on` Represents an HPF ON directive. It has one child, `statements`, which is a `dt_seq` node containing a list of statements governed by this directive. In addition, it has four attributes:

`home_expr` An expression node representing a variable, template element, or processors element.

`has_resident` A Boolean attribute specifying whether or not a RESIDENT clause is part of this ON directive.

`resident_vars` A (possibly empty) list of expression nodes representing the variables defined to be resident in the RESIDENT clause, if it is present.

`new_vars` A (possibly empty) list of expression nodes specifying any variables declared to be NEW (i.e., privatized) in the ON directive.

`dt_hpf_resident` Represents a stand-alone RESIDENT directive. It has one child, `statements`, which is a `dt_seq` node containing a list of statements governed by this directive. In addition, it has one attribute, `vars`, which is a list of expression nodes specifying the variables (if any) that are declared to be resident by this directive.

`dt_region` Represents an Open MP region. It has one child:

`stmts` This is a `dt_seq` node that contains the statements and other dotree nodes inside this region.

It also has three attributes:

`op` This is a `dotree_op` that names the kind of region. A few of the many possible operators that can occur here are:

- `vaOMP_DIR_PARALLEL`
- `vaOMP_DIR_PARALLELDO`
- `vaOMP_DIR_PARALLEL_SECTIONS`
- `vaOMP_DIR_SECTIONS`
- `vaOMP_DIR_SECTION`

`modifiers` This a list or table of modifiers—typically, the options of the region directive. Some typical options are:

- `vaOMP_OPTION_FIRSTPRIVATE` This is an expression node with 1 child. The child is a list of names of the variables being declared as FIRSTPRIVATE.
- `vaOMP_OPTION_LASTPRIVATE` Same for LASTPRIVATE.
- `vaOMP_OPTION_PRIVATE` Same for PRIVATE.
- `vaOMP_OPTION_COPYIN` This is an expression node with 1 child. The child is a list of THREADPRIVATE variables to be copied from the master thread to each thread in the team of the region.

- `vaOMP_OPTION_IF` This is an expression node with 1 child. The child is the logical expression for the IF clause.

`closers` This is a list or table of options that can appear at the end of the region. An example is

`vaOMP_NOWAIT`

A region whose operator is `vaOMP_PARALLELDO` and one of whose options is `vaOMP_OPTION_NUMA` is referred to as a *numa region*. The closers attribute of a numa region consists of a list of two elements:

- If there is an associated LASTPRIVATE list, TRANSFORM creates a private Boolean variable which is initialized (inside the region just before entry to the loop) to FALSE. Inside the loop, it is set to an expression that evaluates to TRUE precisely for that thread that executes the logically "last" iteration of the loop. The name of that variable is the first element of the closers list.

  If there is no LASTPRIVATE clause, this element is `vaNULL`.
- The second element of the closers list is the (address of) the `numa_config descriptor` that describes the team of threads that is started up on entry to the HPF_NUMA loop.

4.2.2 Expression nodes

Expression nodes in the dotree fall into two classes: value computations (the dtb disjunction is `VC`) and flow computations (`FL`).

Every `VC` node has at least the following three attributes:

`Op` The operator name (e.g. vaPLUS). The complete list of operator names is defined in the file `dtr/master/dtr_op.h` It also occurs in the generated file `dtr/master/dtr_opdef.h`.

`Aty` The type; this is an entry in the type table, referred to as a pAtyNode.

`Cir` A corresponding CIR node (referred to as a pIlNode). This attribute will of course in general be nil for new `VC` nodes created by Transform.

Every `VC` node is one of the following:

`VC0` Has no children. vaNEWSCOPE is an example.

`VC1` Has 1 child. vaFETCH is an example.

`VC2` Has 2 children. vaPLUS and vavFETCH are examples.

`VC3` Has 3 children. vavPLUS is an example.

`VCN` Has one child which is a table; this is used for any operator having more than 3 children. An example would be vaBLOCK_FAST_SIMPLE_BD.

`VCarrayref` Represents an array reference. The operator is always vaARRAY or vavARRAY. The children are

  `Base` A `VC` which is a symbol holding the array name.

`Subs` A table of `VC` nodes holding the successive subscripts of the array reference.

`VClist` Has 1 child which is a list of `VC` nodes.[1] The operator is always vaLIST. This node is really "structural sugar"; by itself it performs no computation.

`VCcon` A terminal node representing a constant. The operator is always vaCONTOK. Has a pConNode as an attribute.

`VCsym` A terminal node representing a symbol. The operator is always vaSYMTOK. Has a pSymNode as an attribute.

`VCtmp` A terminal node representing a scalar temporary generated by the Front End. The operator is always vaTMPTOK. Has a pTmpNode as an attribute. This node is mainly vestigial, but must be included as long as the Front End keeps generating pTmpNodes. Transform uses pSymNodes for the temporaries it creates.

`VCterm` A catch-all terminal representing either a vaLABTOK, vaNAMTOK, or vaEDTTOK operator. Such a node always comes from the Front End: it is never created by Transform and always has a corresponding CIR operator, so the `Cir` attribute is enough to characterize it completely.

Every flow computation has the following attribute:

`flop` ("flow operator") The operator name. While operator names corresponding to value computations begin with "va", those corresponding to flow computations begin with "fl". These operator names are also included in the file dtr_opdef.h.

The flow computations are the following:

`FLugoto` An unconditional GOTO. The possible operators are flGOTO, flCYCLE, and flEXIT. Has one child, the `target`, whose value is a pLabNode (i.e. the label which is the target of the unconditional GOTO).

`FLagoto` An assigned GOTO. The operator is flAGOTO. Has two children:

`symbol` A `VCsym` representing the symbol which has been assigned a label which will be the target of the node.

`labs` A `VClist` consisting of all the labels which are somewhere in the scoping unit assigned to the symbol.

`FLstop` The operator is flSTOP. Has one child, (`StopMark`), which is the optional constant value specified in the STOP statement.

`FLcgoto` The operator is flCGOTO. Has two children:

`expr` An expression node whose value is used to determine the label to jump to.

[1] There is a technical dtb wrinkle here. A dtb object can be used to form tables or lists, but not both; this is determined in the ldo specification file `dtb/master/dotree.ldo`. `VC` nodes can only form tables. Hence actually the child of a `VClist` node is a list of `acomp` nodes—an `acomp` node is defined to form lists, rather than tables, and each `VC node` is automatically an `acomp` node.

- `lablist` A list of labels. The value of the expression is used as a 1-based index into this list to find the target of the node.

`FLreturn` The operator is flRETURN. Has one child, the optional parameter from the source text.

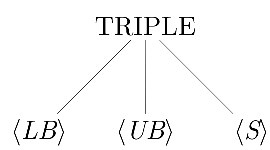
`FLarif` An arithmetic IF. The operator is flARIF. Has four children:

- `expr` The expression to be evaluated.
- `lt0_lab` The target label if the expression is < 0.
- `eq0_lab` The target label if the expression $= 0$.
- `gt0_lab` The target label if the expression is > 0.

4.2.3 Some common operators

TRIPLE and vINDEX nodes

Triples such as $LB\!:\!UB\!:\!S$ (i.e., "lower bound" : "upper bound" : "stride") are represented as `VC3` nodes whose operator is a vaTRIPLE:

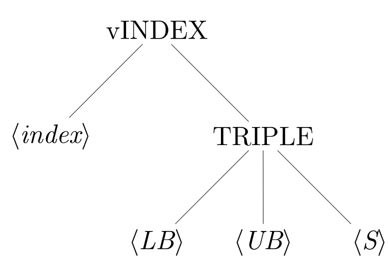


TRIPLE nodes in an expression are *conformant*: each term in the expression must have an equivalent set of triples in the same order. That is, triples in corresponding dimensions must represent the same number of elements.

A FORALL index can be thought of as a triple where the conformance restriction is relaxed. Instead, the triple itself is given a name. This allows the user to identify triples which are "out of order", as in

forall $(I = 1\!:\!100,\ J = 1\!:\!100)$
$A(I,\ J) = B(J,\ I)$

Such nodes are represented as vavINDEX nodes:



where the *index* child is the position of the index in the FORALL header—in effect, the name of the triple. A vINDEX node can thus be thought of as a named triple. In general, FORALL indexes are said to represent *named dimensions*, while triples represent *unnamed dimensions*.

Whole arrays

The term “whole array” in Fortran has a special, somewhat unintuitive significance. If an array A is declared as

dimension $A(2:10)$

then any reference to A in the program counts as a whole array, while a reference to $A(2:10)$ counts as an array section, even though it includes all the elements of the array. This has semantic significance in Fortran: LBOUND(A) evaluates to 2, while LBOUND($A(2:20)$) evaluates to 1. Therefore, it is important to know in the compiler which array references denote whole arrays in this sense.

In order to manage this, a `VCarrayref` node is tagged with a boolean attribute `whole_array` which is TRUE iff the node comes from a whole array reference in the source program. In addition, all temporary arrays created by the compiler are tagged as being whole arrays.

Stack temporaries

The Transform phase generates stack temporaries by first declaring the start of a new scope with a statement-level vaNEWSCOPE operator. Within this scope, any number of stack temporaries can be allocated, using statement-level vaSALLOCATE operators. The end of the scope is indicated by a statement-level vaENDSCOPE operator. Ending a scope has the effect of popping all temps allocated within that scope.

vaNEWSCOPE and vaENDSCOPE take no children. vaSALLOCATE has one child, which is a vaSYMTOK representing the name of the array being allocated.

NEWSCOPE/ENDSCOPE pairs are always placed in the dotree so that they are both children of the same `dt_seq` or `dt_tangle` node. (Of course, the vaENDSCOPE follows the vaNEWSCOPE node, with other nodes in between. They do not have to be the first and last nodes of the list of children of the `dt_seq` or `dt_tangle` node, however.) The reason for allowing `dt_tangle` nodes here is that it is actually possible for an early exit to take place out of a scope: the GEM component of the compiler will handle any necessary cleanup correctly.

Array-valued function calls

An array-valued function call is represented by means of a vaAFUNC operator. This operator has two children:

- A SYMTOK holding the name of an array temporary into which the result of the function is to be computed.
- A representation of the actual function call.

Array-valued functions are implemented by the Middle End to take a hidden argument, passed assumed-shape, which is a temporary holding the return value of the function. The AFUNC operator tells the Middle End that its first child is to be used as that hidden argument.

4.3 Dope information

The term *dope* is used in two contexts in our compiler:

dope information or **symbol table dope** This is declarative information about data objects containing, for instance, the rank of an array. It is really part of the symbol table. Ordinary arrays have declarative information that never changes as a routine executes. However, the declarative information for allocatable arrays and pointers can be modified in the course of execution, and for pointers, this can happen in two very different ways—allocation and pointer assignment. To handle these two ways correctly, we maintain two versions of dope information for pointers:

regular dope information This is usually just called "dope". It exists for every array, whether or not it is a pointer.

alternative dope information This is called `alt_dope` in our compiler. It is used only in situations in which an object needs to be allocated, but there is not enough information available at the point of allocation to fully describe the object. Currently, alternative dope information is maintained for two kinds of objects:

- pointers
- allocatable dummies with the INHERIT attribute.

dope vector Dope vectors are used in several situations:

When an argument is passed assumed-shape to a subroutine, the calling routine needs to pass quite a lot of information about the actual argument—in particular, more than just its base address. So what happens is that all of this information is packaged up into a vector and the vector itself is passed, rather than the base address. This is called a dope vector, and in our compiler, it is extended to include information about the distribution of the array. This is dynamic information that may change as the code executes. (A subroutine may be called twice, for instance, with two different actuals corresponding to a specific dummy.)

Dope vectors are also used to manage pointers and allocatable arrays, as explained below.

The way these versions of dope interact is this:

An ordinary array (i.e., not a pointer, allocatable array, or assumed-shape dummy argument) has its regular dope information filled in from the declarative information in the program. This information may consist of expressions. For instance, an explicit-shape dummy may have its size specified in terms of another dummy argument. This would be reflected in the regular dope information.

Pointers, allocatable arrays, and assumed-shape dummy arguments have dope vectors associated with them. On allocation, pointer assignment, or subroutine entry, these dope vectors are filled in. The regular dope information of such objects consists largely of pointers to fields in the dope vector; it is the dope vector fields that actually change on execution.

When a pointer is used as the left-hand side of a pointer assignment, its dope vector is modified to reflect the information of the target of the assignment. When a pointer is used for allocation, however, there is no right-hand side to take information from. In this case, the compiler will in general need to make some decisions. For instance, if a rank 2 pointer was declared simply as having a (BLOCK, BLOCK) distribution, the DATA phase of Transform will by default allocate cell places to the two dimensions of the pointer. These cell places can be overriden by a pointer assignment, but will be used if the pointer is allocated. These cell places are stored in the `alt_dope`. On pointer allocation, this information is copied into the dope vector for the pointer. In this way, the `alt_dope` stores information created by the DATA phase of Transform that is used only for allocation of pointer arrays.

The only reason this is necessary is that pointers are used both for pointing (i.e., aliasing) and for allocation. Allocatable arrays, which cannot be pointer assigned, generally do not have this problem. Their dope vectors are largely filled in by DATA in the normal fashion, and those fields are never changed. The one exception to this (as mentioned above) concerns allocatable dummies with the INHERIT attribute. When such a dummy is allocated, its mapping is taken from a compiler default, which is stored in the `alt_dope` structure[2].

In this compiler, the Middle End handles the actual procedure calling mechanism, and therefore is responsible for creating dope vectors. The responsibility of Transform is simply to leave enough information around so that the Middle End can do this consistently. The information contained in dope vectors is outlined below in Section 4.6.

When we refer to dope, or dope information, in this report, we always mean symbol table dope. Dope vectors are always referred to as such.

A `dope` structure holding symbol table dope information is attached to every `pSymNode` by LOWER. This dope structure is the repository for all declarative information about array symbols. It is the place from which all such information is retrieved by Transform code. The access functions for the dope information are generated by dtb from an ldo description given in the file `dtb/master/dope.ldo`.

The fields of a `dope` structure are:

`irank` The rank; 0 for a scalar.

`addr` A `VC` holding the base address of the array. This provides a way to get back from the dope structure to the name of the data object it describes. Actually, it would be clearer if this field were a symbol table entry instead of being a `VC`. But such an entry might be a pSymNode or a pTmpNode, and using a `VC` avoids the need to make a disjunction of non-dtb types.

`charlen` a `VC` holding the length of a character type symbol.

`lbound` An array from `[1..irank]` of declared lower bounds.

`ubound` Same for upper bounds.

`pntsym` A `pSymNode` holding the name of the replicated temporary used to copy the array into when it is used in a punted construction.

`nmltmp` A `pSymNode` holding the name of the processor 0 temporary into which the array is copied for I/O purposes if the array is part of a NAMELIST; otherwise this field is NULL.

`base` Holds the local stack base of the array in the NUMA compiler (see page 228) if this array is allocated on the numa stack.

`extent` Holds the extent field for this array in the NUMA compiler (see page 228) if this array is allocated on the numa stack.

[2] This is not the only way such an allocation could be handled. But the language seems to be silent on this issue, and what is described here is the choice we have made in our implementation.

The file `dtb/master/occurrence.ldo` contains an extension to this dope structure, adding the following four fields:

`subspaces` A table of data subspaces (see page 51).

`complete` The completion structure (see page 52).

`replicated` The replication structure (see page 52).

`cells` The cell structure. This field exists only if the dope describes a symbol naming a template (see Section 4.5.7, page 53).

The fields `lbound` and `ubound` are sometimes referred to as *global dope*. The `num_folds` field of a `subspace` entry (see page 50) is referred to as *local dope*; it represents the local extent of a global array.

4.3.1 Symbol table dope processing

Here we give a synopsis of the way in which (symbol table) dope information is processed. All this information is also repeated in the appropriate sections of this report which deal with individual Transform components.

Dope information for an array is managed in several different ways, depending on the attributes of the array. There are three different cases to consider:

arrays in the source code that are not assumed-shape

assumed-shape arrays These are by definition dummy arguments.

compiler-generated stack allocatable arrays Compiler generated temporary arrays are all stack allocatable. They are used in supporting array-valued functions (see Section 7.3), in removing strip mining obstacles, in handling copy-in/copy-out of arguments at subroutine calls, as well as in many other places.

For all arrays, the Front End fills in dimension information attached to the symbol table entry for that array. (For assumed-shape arrays, the extent field is made NULL; subsequent compiler phases retrieve information from the corresponding dope vector.)

For arrays in the source code that are not assumed-shape, LOWER converts the dimension information into global dope. For all assumed-shape arrays, LOWER inserts into the global dope fields expressions referring to fields in the dope vector. (See Section 7.5.8, page 84.)

For compiler-generated stack allocatable arrays, global dope is filled in by Transform at the point of allocation.

For all arrays, DATA uses global dope together with distribution information about arrays to compute local dope, and fills in those local dope fields. (See Section 6.3, item 2, page 72.)

For all arrays, DTR2CIR changes the dimension information attached to their symbol table entries to reflect the local dope. This is because after Transform, all arrays in the compiler are really local, and need to be seen in terms of their local size. (see Section 5.2, page 60.)

4.4 The dependence graph

The dependence graph consists of (directed) arcs between dotree nodes. Each arc is characterized as a *true*, *anti*, *input*, or *output* dependence, and each arc has associated with it a direction vector and a distance vector. Only plausible dependences (that is, dependences whose direction vectors are lexicographically non-negative) are represented as arcs.

4.5 Data and iteration spaces

Data and iteration spaces are the data structures which enable Transform to manage parallel data and parallel computations. A general introduction to these data structures can be found in the paper [14]. Some more details are given here; but most of the details in the paper are not repeated.

The access functions for the data structures described here are generated by dtb from an ldo description in the file `dtb/master/occurrence.ldo`.

4.5.1 Cells

Cells give a representation of the coordinates of a virtual processor space. The cells representing a particular virtual processor space are given some numbering, say from $j = 1$ to $j = n$, where n is the dimension of the virtual processor space. (This would be the dimension of the template if the virtual processor space was defined by a TEMPLATE directive.)

Every cell has the following attributes:

`dist` The name of the distribution map from that cell to a dimension of the logical processor space. Possible values for `dist` are:

`dist_block`

`dist_cyclic`

`dist_serial`

`dist_block` and `dist_cyclic` in turn have a `size` attribute corresponding to the parameter n in a **block**(n) or **cyclic**(n) directive.

`strip_len` The *strip_length* of the cell. Referred to in this report as $strip\_length_j$.

`multiplier` The *multiplier* of the cell. Referred to in this report as $multiplier_j$. The *strip_length* and *multiplier* together constitute what is referred to in [14] as the *cell place*.

`v_bar` A privatized variable holding the coordinate in the j^{th} dimension of the logical processor space of the processor returned by *my_processor*`()`. Thus, this field is an expression involving *my_processor*`()`. This is what is denoted in this report by $\bar{v}_j$.

(Note that the subscript j used in this report is not explicitly represented in these elements of the data structure, because j is just a way of distinguishing between different dimensions of the virtual processor space, which are here distinguished by the different cells.)

The value returned by *my_processor*`()` (that is, the address of the processor itself) is denoted by $\bar{v}$. Except for the distribution map, these attributes are related as follows:

1. The product of the strip lengths of the cells in each logical processor space must equal the number of processors.

2. Strip lengths and multipliers are related by

$$\mathit{multiplier}_n = \prod_{j=1}^{n-1} \mathit{strip\_length}_j.$$

   In particular, $\mathit{multiplier}_1 = 1$. Equivalently, of course, *multiplier* can be computed recursively by first setting $\mathit{multiplier}_1 = 1$, and then successively calculating

$$\mathit{multiplier}_n = \mathit{strip\_length}_{n-1} * \mathit{multiplier}_{n-1}.$$

3.
$$\bar{v} = \sum \bar{v}_j * \mathit{multiplier}_j$$

   where the sum is taken over all the cells in the logical processor space. To go the other way, we have

$$\bar{v}_j = \left\lfloor \frac{\bar{v}}{\mathit{multiplier}_j} \right\rfloor \bmod \mathit{strip\_length}_j$$

   (All numbers involved are non-negative.) Note that $\bar{v}_j$ is 0-based.

4.5.2 Subspaces

Data and iteration spaces are products of subspaces. Each subspace has the following attributes:

`alloc_stride` The allocation stride. *alloc_stride* in this report.

`alloc_offset` The allocation offset. *alloc_offset* in this report.

The *alloc_stride* and *alloc_offset* together constitute the *data allocation function* or *iteration allocation function* mapping a dimension of the data space to a corresponding dimension of the virtual processor space.

`cell` The cell representing the subspace of the virtual processor space which the data or iteration space is mapped to by its allocation function.

`num_folds` This represents the number of foldings (or the "multiplicity") of this cell when it is folded onto a dimension of the logical processor space by the distribution map.

4.5.3 Data spaces and maps

A data space is associated with each data object. Since the data space is declarative, it is attached (as a dtb attribute) to the dope for that object. Specifically, there is an attribute `subspaces` on the dope for each object which represents the table of data subspaces for that object.

The subspaces may be used in different ways, however, and this will vary from occurrence to occurrence in the source text. (The use of a data subspace refers to the kind of subscript that occurs in that place in the array reference.) Hence attached to each `VCarrayref` in the dotree is a table of subspace `uses` (a subspace use is represented in the ldo description as `SSp_use`) which is parallel to the table of data subspaces for the array. The different kinds of data subspace uses are:

primary The subscript is either

- A Fortran TRIPLE, or
- An affine function of a single iteration space index, with the additional restriction that (calling the iteration space index i), this is the first subscript (going left to right through the subscript list of the array reference) which is an affine function of i.

  Thus, in the example

forall $(i = 1\!:\!N)$
  $A(2{*}i{-}3,\ 3{*}i{+}2) = \ldots$

the subspace corresponding to the first subscript $(2{*}i{-}3)$ is primary,while that corresponding to the second subscript $(3{*}i{+}2)$ is complex (see below).

Each primary use in turn has two integer-valued attributes:

`index` The number of the associated iteration space index.

`correspond` For data space uses, this is the index into the iteration subspace table (see below) of the corresponding iteration subspace (i.e. the iteration subspace coming from the same iteration space index). For iteration space uses, this is the inverse map, i.e., the index in the data subspace table of the corresponding data subspace.

scalar The subscript involves neither TRIPLEs nor iteration space indexes, and is therefore constant over the iteration space. There are actually two kinds of scalar uses:

local scalar The subscript must be computable locally on each processor. If the subscript is a constant, or if it is a replicated or privatized variable which is not an iteration space index, then the use is local scalar. In particular, a subscript which is a function of a DO loop index creates a local scalar use.

complex scalar The subscript is constant over the iteration space but is not a priori computable locally.

For example, in the array reference $A(M, V(N))$, where M and N are not indexes of the iteration space, the subscript M is a local scalar subscript, while $V(N)$ is a complex scalar subscript.

complex The subscript is of any other form. For example, the subspace for $A(V(:))$ has a complex use, as does the subspace for $A(i + j)$ where i and j are both FORALL indexes.

The purpose of characterizing subspaces by their uses is to help in determining where data motion may be required.

4.5.4 Mapping data space to virtual processor space

The data allocation functions determine how data space is mapped to virtual processor space with the exception of two pieces of information:

1. The map may not be onto.
2. The map may be 1-to-many.

To handle the fact that the map may not be onto, a *completion structure*, denoted by `complete`, is attached to the dope for each data object. This structure consists of two parallel tables, one of which is a table of subspaces and the other one of which is a corresponding table of values.

Let us refer to the subspaces described in the previous section of this report as *regular subspaces*.

A *completion subspace* of an array A is not a regular data subspace of A (although it always comes from a template or another array with which A is aligned). But one can think of it as an "invisible" extension to the data space for A with scalar subscript given by the corresponding entry in the value table mentioned above. This value is mapped by the data allocation function of the completion subspace to the cell (called a *completion cell*) (also determined by that subspace) representing a dimension of the virtual processor space which is not mapped into. In this way, a constant value for the address in that virtual processor space dimension is arrived at. (Typically, the constant value is 0.)

To handle the fact that the map from data space to virtual processor space may be 1-to-many, a `replicated` attribute is also associated with the dope for each data object. This attribute specifies a table of cells (called *replicated cells*).. Each replicated cell represents a dimension of the virtual processor space over which the data object is replicated.

For such cells, the `dist` (i.e. distribution) attribute is ignored.

As a matter of nomenclature, the complete set of data subspaces, completion structure, and replicated structure is sometimes called the *data space structure* or *data space information*. When we are using the data space information that refers specifically to the data virtual processor space, we sometimes call it the *data VP space*. Similar terms apply to iteration space data structures.

4.5.5 Mapping virtual processor space to logical processor space

The coordinatewise mapping of dimensions of the virtual processor space to corresponding dimensions of the logical processor space is determined by the distribution associated with each dimension. Since each virtual processor dimension (with the exception of those which are replicated dimensions) is associated with a data subspace, for which the distribution is given, we have all the information at hand to compute these maps, with one exception: for BLOCK distributions, we need to know the block size. This may be specified in the directive (via the BLOCK(n) syntax), or it may be computed (as will be explained below in Section 6.4, page 73). The value is stored in the attribute `num_folds` of each data subspace.

4.5.6 Iteration spaces and maps

With the exception of the subspace uses, the data spaces and maps are all declarative and therefore can be attached to the symbol table dope information for each data object. The iteration spaces and maps, however, may differ at each computational node in the program.

Therefore, to each `VC` in the dotree is attached an attribute `ispace` of dtb type `iter_space`. An `iter_space` in turn has four attributes:

`spaces` The table of iteration subspaces. Each iteration subspace has all the attributes of a data subspace enumerated above (i.e. `dist`, `strip_length`, `multiplier`, and `v_bar`), and in addition has the following attributes:

`lower` The iteration lower bound.

`upper` The iteration upper bound.

`stride` The iteration stride.

These three attributes are derived from the TRIPLE or the FORALL bounds for the corresponding iteration indexes. (Note that the corresponding data bounds are simply the declared lower and upper bounds, and are already present in the dope for the data object, so these attributes are not needed for data subspaces.)

`complete` The completion structure.

`replicated` The table of replicated cells in the virtual processor space.

`uses` A table of subspace uses, parallel to the subspace table given by the `spaces` attribute. Actually, all iteration subspaces in this list by definition have primary uses. The uses are included because the primary use is tagged with the `index` and `correspond` attributes (see above).

In addition, each `VCarrayref` has an attribute `refspace`, also of dtb type `iter_space`. For any array reference not of a derived type, the `refspace` is the same as the `ispace`. The `refspace` attached to an array reference that is a *structure-component* reflects only the mapping of that component and not of any other components of the structure. For instance, the reference $A(i)\%B(j)$ has three iteration spaces associated with it:

- The `ispace` of the expression $A(i)\%B(j)$.
- The `refspace` of $A(i)$.
- The `refspace` of $B(j)$.

4.5.7 Templates

Template names are entered into the symbol table. Like declared arrays, they have dope attached to them which specifies their declared bounds. Attached to this dope (only in the case of templates) is also an attribute `cells` representing a table of cells which make up the virtual processor space named by the template.

4.6 Dope vectors

Dope vectors are not really part of the Transform data structures, but we include their description here, both for completeness, and also because Transform has to make sure that it leaves information in the internal representation of the program in such a form that the Middle End can create dope vectors correctly.

Dope vectors can be thought of as having four sections:

F90 ARRAY SECTION This section contains "normal" dope vector information, which would need to be included even without the HPF extensions to the language. The size information included in this section is all local size information, suitable for use by a scalar Fortran compiler back end. This section includes the following fields (among others):

`rank` The number of dimensions of the dummy array.

`dtype` A data type code.

`length` Maximum length of a single dummy array element.

`arrsize` The number of elements in the dummy.

`pointer` The base address of the dummy.

`info` This is a table, with successive elements corresponding to successive dimensions of the dummy. Each element contains the following fields, which are used in generating a single address in a processor's linear memory from the collection of the dummy's subscripts.

`inter_element_spacing` See the example on page 33 for the use of this field.

`upper_bound`

`lower_bound`

In a single-processor or explicit SPMD routine (e.g. an HPF_LOCAL routine), the `upper_bound` and `lower_bound` fields hold the values returned by the UBOUND and LBOUND intrinsics.

HPF ARRAY SECTION This section contains global information about arrays. For each dimension of the dummy, it contains the following fields:

`lower_bound` The lower bound of the global HPF actual array argument associated with the dummy.

`upper_bound` The upper bound of the global HPF actual array argument associated with the dummy.

In a global routine, the regular dope lbound and ubound arrays (for an array with a dope vector) contain fetches of the elements in these arrays. The values in these fields are what are ultimately retrieved (after some semantic checking) by the LBOUND and UBOUND intrinsics.

In an HPF_LOCAL routine, these arrays contain the values returned by the GLOBAL_BOUNDS inquiry subroutine.

ALIGN SECTION This section contains information about the alignment of the global HPF actual array argument associated with the dummy. For each dimension of the dummy, it contains the following fields:

`alloc_offset` The *alloc_offset*.

`alloc_stride` The *alloc_stride*.

`templ_index` The corresponding template dimension. 0 for collapsed dimensions.

TEMPLATE SECTION This section contains information about the distribution of the global HPF actual array argument associated with the dummy. It contains a field

`rank` The template rank; not always the same as the dummy's rank.

and for each dimension of the template, it contains the following fields:

`templ_lo_bound` Declared lower bound of template with which the global actual argument is aligned.

`templ_hi_bound` Declared upper bound of that template.

`strip_length` The extent of the logical processor array in this dimension. Comes from the PROCESSORS directive.

`num_folds` The size of this dimension in the logical memory space. Computed as in Section 6.4.

`dist` Block, cyclic, or replicated.

`axis_kind` Normal (primary use), single (scalar use), or replicated.

`align_index` Corresponding dimension in the ALIGN SECTION. 0 if `axis_kind` is single or replicated.

`cyclic_size` n if the distribution of this dimension is CYCLIC(n); else 0.

`scal_exp` Scalar value if `axis_kind` is single; else garbage.

Accesses to symbol table dope information and alignment and distribution information for dummies which inherit this information via dope vectors are turned transparently in the compiler into accesses to the appropriate fields of the dope vector for that dummy. This happens in LOWER (Section 7.5.8, page 84.)

4.6.1 Dope vector processing

Dope vectors are needed to support the following constructions:

- to support parameters passed assumed-shape. Such dope vectors are filled in by the caller at the call site.
- to support compiler generated stack allocatable temporary arrays. Such dope vectors are filled in on allocation of these temporaries.
- to support allocatable arrays and pointers. Such dope vectors are filled in as arrays are allocated, and whenever a pointer assignment takes place.

Dope vectors are filled in as follows:

In the first two cases, the Middle End fills in the dope vector. It fills in the F90 section from symbol table information. For mapped actuals, this symbol table information has been modified by DTR2CIR to reflect localized sizes (see pages 48 and 60). The rest of the dope vector is filled in by the use of functions that the Middle End calls which use Transform utilities to access the symbol table dope information to get global dope. These utilities are contained in the file `dtr/master/dtr_hpf_dv.c`.

In the third case, Transform fills in the entire dope vector.

Chapter 5

Conversion Between CIR and Dotree Formats

We now describe the separate phases of Transform. This chapter describes the two filters at the beginning and end of Transform.

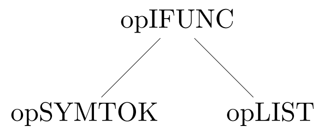
5.1 CIR2DTR

Most of the processing here is straightforward data structure translation. Some points should be noted:

- Sometimes in the CIR small compiler-generated integers are represented as VALTOK nodes. CIR2DTR converts these nodes to CONTOK nodes for uniformity and ease of handling by later phases of Transform.
- Non-structured DO loops are identified by scanning for explicit flow of control statements and computing for each statement in the program its list of predecessors and successors. DO loops which are identified as having abnormal exits become `dt_loop` nodes in the dotree; all other DO loops become `dt_do_loop` nodes.
- Entry and exit nodes are identified. Entry nodes are nodes at the statement level coming from the ENTRY keyword. Exit nodes are nodes at the statement level representing RETURN, STOP, or END.

  Entry nodes become `dt_entry` nodes in the dotree, and exit nodes become `dt_exit` nodes in the dotree. The collection of entries and exits is also placed under the `dt_root` node.

  Between each ENTRY statement in the CIR and the next following opLINE operator may be a sequence of "preamble" statements. CIR2DTR collects these statements and moves them to the `preamble` child of the corresponding `dt_entry` node.
- Intrinsic functions in the CIR are represented as binary opIFUNC nodes (or opIFUNCEL, if the function is an elemental intrinsic). The first child of such a node is an opSYMTOK representing the name of the function. The second child is an opLIST holding the list of actual arguments to the function.



CIR2DTR eliminates the opIFUNC or opIFUNCEL node and replaces it by the dotree node corresponding to the actual intrinsic function. For example, if the opSYMTOK represented the intrinsic function SUM, then the CIR structure would be replaced by a vaSUM node whose children are the (dotree equivalents of the) arguments in the argument list of the opIFUNC node.

- OMP directives are turned into `dt_region` nodes in the dotree.

5.1.1 Translation of array expressions

Consider the following program fragment:

> **dimension** $A(20,\ 20)$
>
> $A(:,\ :) = \ldots$

The CIR for this statement is shown in Figure 5.1. In this figure, the nodes are identified by their operator names. Each operator name is really prefixed by "op", so the actual operator names are the CIR operator names opARCS, opVARRAY, etc. The node "A" is really an opSYMTOK whose symbol table entry is that of the array A; and the literal numbers are opCONTOK nodes.

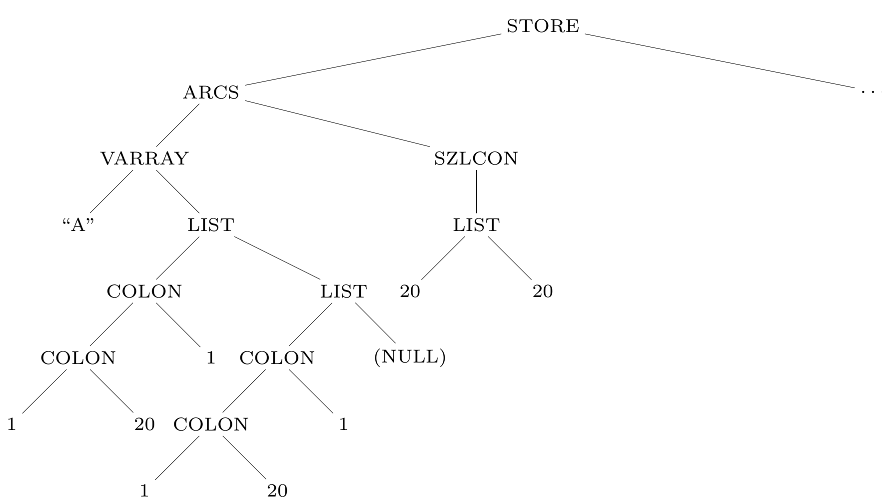


Figure 5.1: CIR for $A(:,\ :) = \ldots$

Note the way the ternary triple expressions 1 : 20 : 1 are represented in terms of binary opCOLON operators; and similarly, lists are represented in terms of binary opLIST operators. opARCS refers to an *array cross section*, and opSZLCON refers to a *constant size list*. The SZLCON nodes are attached to nodes representing array expressions where appropriate to aid semantic analysis in verifying argument compatibility and shape conformance in expressions.

After CIR2DTR, the statement looks like Figure 5.2. In such figures, a table of subscripts with more than one element is enclosed in a dotted box for clarity.

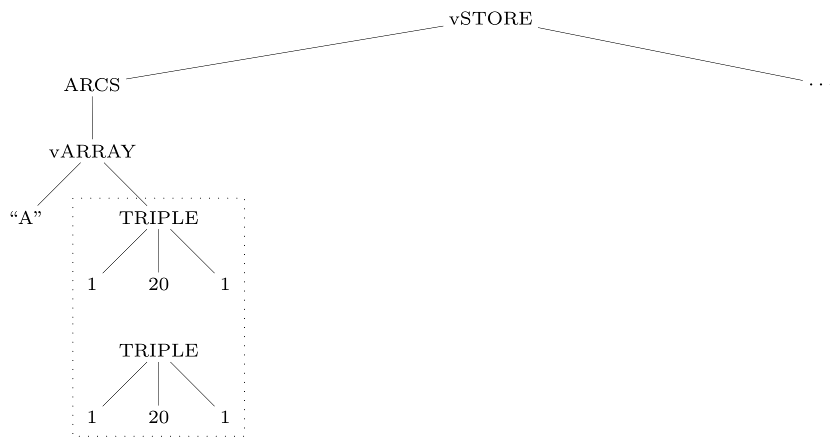


Figure 5.2: $A(:,\ :) = \ldots$ after CIR2DTR

Again, the nodes are identified by their operator names, but here the names in the figure are prefixed by "va", so the real names are vavSTORE, vaARCS, vavARRAY, etc. The convention for these value computations is that a small initial "v" (after the initial "va") denotes a vector operator; thus vavPLUS refers to the addition of two vector expressions, whereas vaPLUS would refer to a scalar addition. In this figure, vavSTORE and vavARRAY have this form.

The second child of a vARRAY node is now a table whose two elements are the two triples.

In addition, the SZLCON nodes are thrown away, and ARCS nodes now have one operand, rather than two.

The CIR for the right-hand side of the following expression

dimension $B(20,\ 20,\ 20)$, $C(20)$

$\cdots = B(3,\ :,\ 1) + C$

is shown in Figure 5.3. AREX is a node which is used in the CIR to cover an *array expression*.

The dotree for this expression after CIR2DTR is shown in Figure 5.4. The (NIL) nodes in this figure are intended to be place holders for context or mask nodes which will later be added in these places, based on any enclosing WHERE statement. (In particular, they are *not* the residues of SZLCON nodes.)

5.1.2 Translation of FORALL statements

The CIR for a typical FORALL statement

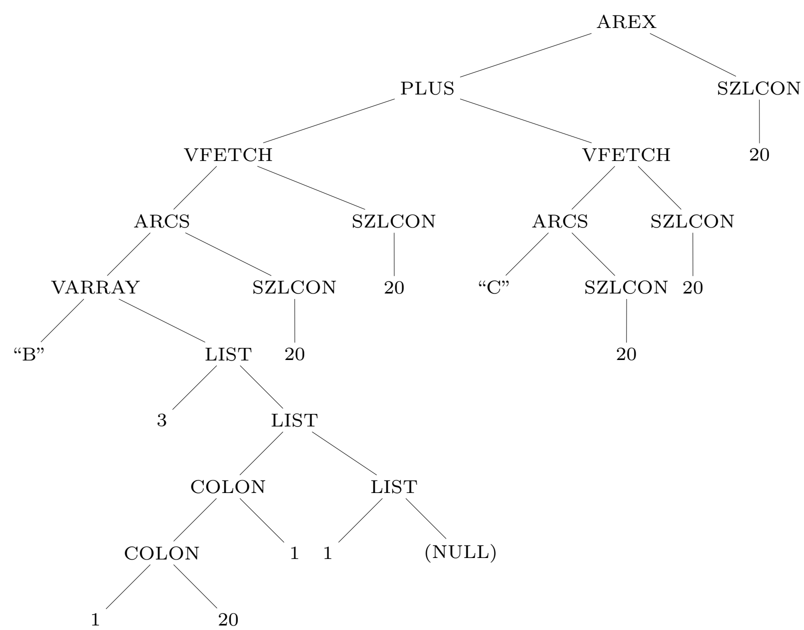


Figure 5.3: CIR for $B(3,\ :,\ 1) + C$

dimension $A(20,\ 20),\ C(20)$

forall $(I = 1\!:\!10,\ J = 3\!:\!17)\quad A(I,\ J) = C(I{+}J)$

is shown in Figure 5.5. The first child of the FORALL node is the FORINDXS node. This node has two children. The first is a list of forall indexes and their bounds, and the second is the optional mask. In this case, there is no mask in the source, so that child is NULL. The second child of the FORALL node is the assignment statement. Within this statement, each forall index is represented by an FAINDEX node. The first child of each FAINDEX node is the SYMTOK holding the name of the forall index, the second child is the order in which that index appears in the FORALL header, and the third child is the triple, also derived from the FORALL header. In this figure, the third children are indicated as being dagged nodes—$\langle 1\rangle$ refers to the COLON$\langle 1\rangle$ node, and so on.

Figure 5.6 shows the same FORALL statement after it has been processed by CIR2DTR. The left and right-hand sides of the assignment statement are now the second and third children of the vaFORALL node. None of the nodes are dagged. The subscripts of the array A now form a table consisting of two FAINDEX nodes. The subscript of the array C is a table consisting of one PLUS node.

5.1.3 Function return values

We will use the program in Figure 5.7 as a running example that will be referred to later in this document. After semantic analysis—and this is not changed in any significant respect by

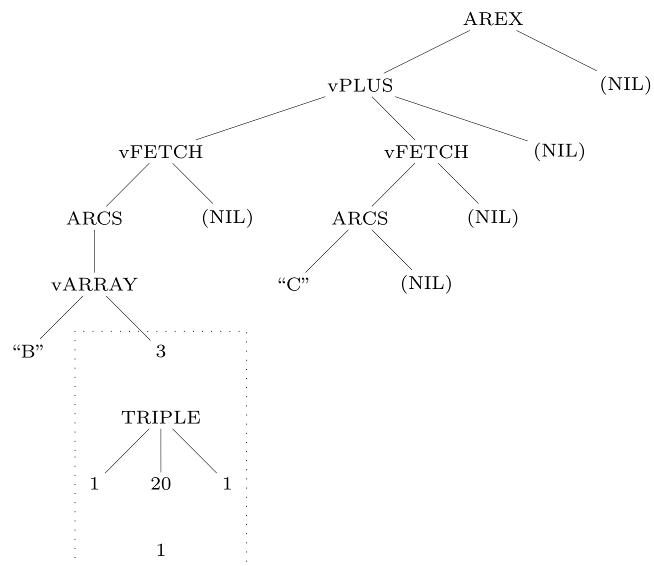


Figure 5.4: $B(3,\ :,\ 1) + C$ after CIR2DTR

CIR2DTR—the essential parts of the representation of this program in the dump is shown in Figure 5.8. If no result variable was declared in the program, the front end would create a result variable named `array_func_0`, and this is the variable that would appear in the symbol table, and which would be assigned to in the executable code.

One thing to keep in mind about this example is that the actual a is mapped differently than the dummy d. This is not reflected in the processing after CIR2DTR, but it will be taken account of later, in ARG.

5.2 DTR2CIR

Again, this is fairly straightforward translation. The following is worth noting:

- If the programmer has redefined an intrinsic function which Transform has used (such as ABS, for instance), then the single name "ABS" will stand for two separate notions after Transform. To handle this correctly, the name entry "ABS" in the symbol table will have a chain of symbol table entries attached to it, each entry with a different *kind* field (user function, intrinsic function, etc.). This chain of entries is set up by DTR2CIR and suffices to disambiguate the references to ABS.

- Localized size information from the dope attached to each symbol is used to reset the dimension information in each symbol table entry. This is done in a way such that all arrays now appear as being 0-based in each dimension.

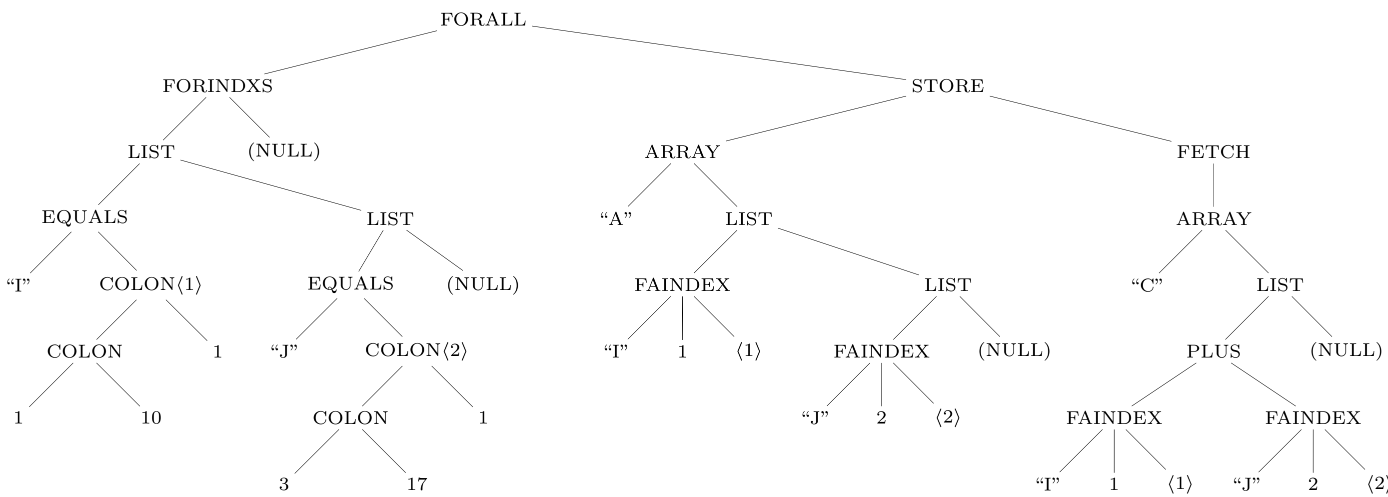


Figure 5.5: CIR for **forall** $(I = 1\!:\!10,\ J = 3\!:\!17) \quad A(I,\ J) = C(I{+}J)$

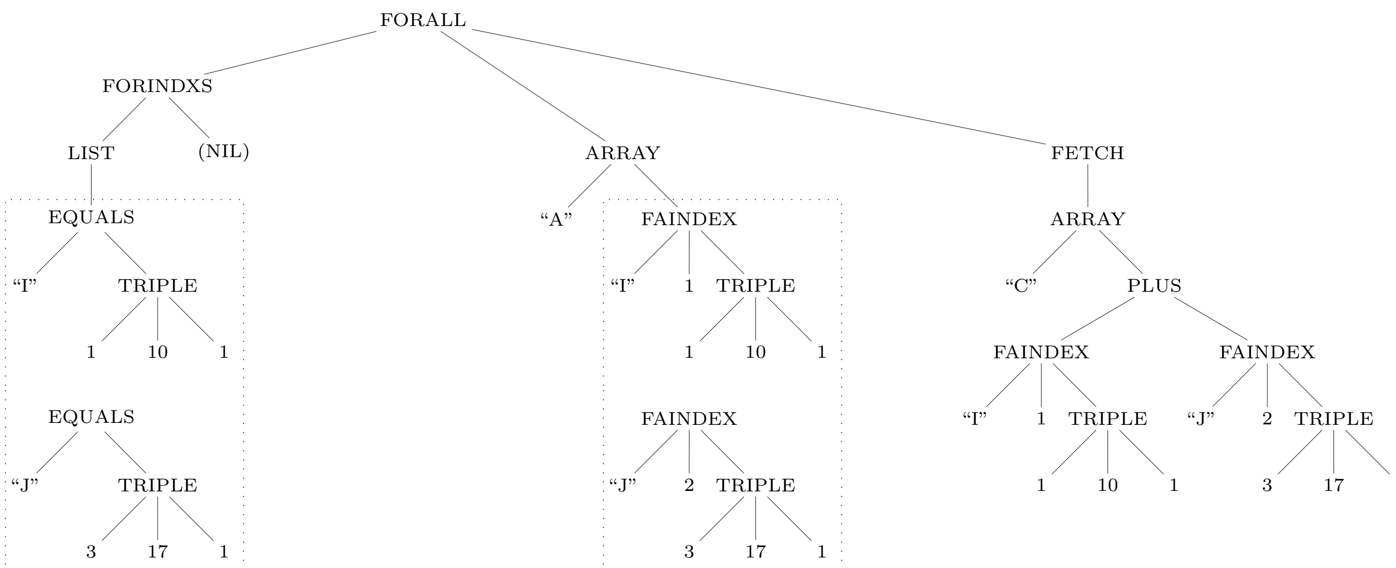


Figure 5.6: **forall** $(I = 1{:}10,\ J = 3{:}17) \quad A(I,\ J) = C(I{+}J)$ after CIR2DTR

```
program af
    integer s
    integer, dimension(5) :: a
    !hpf$ distribute(cyclic) :: a
        ...
    s = sum(array_func(a, scalar_func(3)))
        ...
contains
    function array_func(d, n) result(array_func_result)
        integer n
        integer, dimension(:) :: d
        integer array_func(d(n))
            ...
        array_func_result = d(1:n)
            ...
    end function array_func
    function scalar_func(k)
        integer k
        integer scalar_func
            ...
    end function scalar_func
end program af
```

Figure 5.7: A function (array_func) with a return value specified.

This information is used by later phases of the compiler to generate array addressing and dope vectors. (See Section 4.3.1, page 48.)

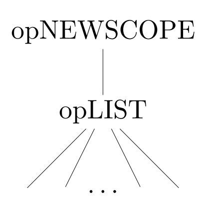
This is the one operation in Transform which is not done on an individual scoping unit basis. This is because the Transform phases expect global dimension information as input. So if DTR2CIR changed this information on a symbol in one scoping unit which was then used in another, that use would receive inaccurate global information. Therefore, this processing occurs once for each program unit, after all its contained scoping units have been processed by Transform.

This operation is suppressed for symbols which have the PUNT attribute. (See Section 3.5, page 20.)

- vaNEWSCOPE/vaENDSCOPE pairs are replaced by the construct

```
program af:

s = sum(LFUNC(array_func,
              (argpos(1, a),
               argpos(2, LFUNC(scalar_func,
                               (argpos(1, 3))
                              )
                     )
              )
             )
       )

function array_func:

DT_ENTRY
  PREAMBLE
    @1 = n
    @2 = d(@1)
array_func_result = d(1:n)

symbol table entries:

."ARRAY_FUNC_RESULT"
      ...
      Dimen:       [
        FETCH-i4#8(t-*i4#30:"@2")]
      ...
```

Figure 5.8: Significant parts of the dump after CIR2DTR for the program in Figure 5.7. Symbols beginning with `@` are compiler-generated temporaries.



where the children of the opLIST node are the CIR translations of the dotree nodes between the vaNEWSCOPE and vaENDSCOPE nodes.

Chapter 6

DATA

The DATA phase does not modify the dotree. It walks the symbol table and creates a dope structure for each data object, based on information contained in the HPF data distrubution directives. It fills in the size fields in the dope structure from the symbol table size information created by the front end. This chapter describes some of the details of this processing.

6.1 Processing HPF directives

DATA walks through all the HPF directives in a scoping unit and processes all the information in them, attaching that information to symbols in the program. Any symbol for which mapping information is not available, or is only partially available, is given default mapping information at this point as well.

The data structures filled in by this processing are mostly specified in the file `dtb/master/directive.ldo`, and are attached to the symbol table entries as specified in `cir/master/cir_tables.h`.

This is what happens for each kind of directive:

PROCESSORS The symbol table entry for the name specified in this directive has an associated `processors_dir` structure. This structure is composed of two parallel tables:

- `slen` This table is filled in with the expressions representing the extents specified in the directive. These correspond to what this report refers to as strip lengths.
- `mults` This table is left empty by DATA, but is later filled in with the multipliers corresponding to the successive dimensions of the processor array.

TEMPLATE A template is treated as a named array in that it has dope associated with its symbol table entry; the dope contains the specified bounds.

ALIGN The symbol table entry for the array being aligned has an associated `align_dir` structure. This structure has the following fields:

- `template_name` The `pSymNode` representing the template with which the array is aligned.
- `alignee_dims` A table having one entry for each dimension of the array.

`template_dims` A table having one entry for each left-over dimension of the template.

For instance, the directives

!hpf$ template $t(n, n, n)$
    real $a(n, n)$
!hpf$ align $a(l, *)$ **with** $t(l, 5, *)$

lead to an `alignee_dims` table having two entries (one for l and one for $*$) and a `template_dims` table having two entries (one for each of the two template dimensions not corresponding to an array dimension).

The `alignee_dims` table consists of `alignee_dim` entries. Each such entry has three fields:

`stride` The stride from the ALIGN directive.

`offset` The offset from the ALIGN directive.

`index` The index into the table of template dimensions. This field maps array dimensions to template (i.e., virtual processor array) dimensions. 0 in this field represents a serially aligned array dimension (as in the second dimension of a in the example above).

The default mapping for an array makes the `stride` 1 and adjusts the `offset` so that the first element of the array is at virtual processor 0 (i.e., template position 0).

The `template_dims` table consists of `template_dim` entries. Each such entry has two fields:

`index` The index into the table of template dimensions.

`scalar_expr` The expression in the template for that dimension, if the expression is other than "$*$". In the example above, this field would hold the constant 5 for the template dimension of index 2, and would hold ⟨nil⟩ for the template dimension of index 3.

DISTRIBUTE Each template has associated with its symbol table entry a `distribute_dir` structure, containing the following fields:

`onto` Holds the `pSymNode` for the processor array onto which the template is being distributed.

`kinds` A table of `dist_kind` entries. Each such entry is one of the following:

`dist_block`

`dist_cyclic`

`dist_serial`

`dist_unspec`

Each `dist_kind` has an `extent` attribute, which gives the extent of the associated template dimension.

`dist_block` and `dist_cyclic` also have a `size` attribute which specifies the optional parameter given in the source BLOCK(n) or CYCLIC(n) directive.

6.2 Creating data subspaces

1. For each PROCESSORS directive, fill in the `mults` table (see page 65) under the `processors_dir` structure recursively, by setting
$$multiplier_1 = 1$$
and then successively calculating
$$multiplier_n = strip\_length_{n-1} * multiplier_{n-1}.$$

2. If the number of processors is a compile-time constant, factor it into primes. Use the list of sorted primes to create a triangular table of striplengths for use when a template has not been distributed over a processor array. If, for instance, the number of processors is $60 = 2*2*3*5$, the triangular table will look like the table in Figure 6.1. The n^{th} row in the table specifies the default strip lengths for a template of dimension n.

60						
6	10					
4	3	5				
2	2	3	5			
2	2	3	5	1		
2	2	3	5	1	1	
2	2	3	5	1	1	1

Figure 6.1: Table of Default Strip Lengths for 60 Processors

The following algorithm can be used to construct this table. The algorithm is entered at MAIN, and it fills in the array which could be declared as

table: array[1..7, 1..7] of **integer**;

The algorithm tries to find the factors of the number of processors which are most nearly equal in each row. (Of course, there is no guarantee that this will be the optimal choice for the algorithm. The programmer should really specify this choice by a PROCESSORS directive to be sure of getting the best performance.) The only way to get the best such set of factors would be to perform an exhaustive search; this algorithm provides a somewhat greedy but reasonable approximation. In particular, if there is a solution with all the factors equal, this algorithm will find it.

The global variables used in this algorithm have the following significance:

P The number of processors.

n The number of prime factors of P. n is made to be at least as large as 7 by adding additional "factors" if necessary. Each such additional "factor" has the value $P+1$. The value $P+1$ is chosen just large enough so that the tests in the algorithm will prevent it from being included as an actual factor. These additional large phony factors, which allow us to make $n \geq 7$, trick the algorithm into allowing additional factors of 1 to be included as entries in the table in case there are not enough prime factors of P, as in the last three lines of the table in the example above.

GOAL What each *strip_length* in a given row would be if they were all equal. GOAL is a real number; all the other variables in this algorithm are integers.

A An array of booleans indicating which of the prime factors of P have already been used in processing previous entries of the current row of the table.

TOTAL The number of prime factors of P which have already been used by previous entries in this row of the table. Equals the number of TRUE elements of A.

M The current best guess for the table entry we are looking for. (See below for how M is used.)

In this algorithm, i runs over the rows of table[ ], and j runs over the elements in each row. Since we only fill in the lower diagonal elements of the table, j runs from 1 to i.

For each row of the table,

- The variable GOAL is recalculated.
- The variable TOTAL is reset to 0.
- Each entry of the array A is reset to FALSE.

For each element of each row,

- The array B of booleans is initialized to FALSE. B will be used to specify a subset of available prime factors which have not been used (i.e. which are not represented as TRUE in A). The number of primes represented in B is maintained in the local variable kk.
- The array C of booleans is initialized to FALSE, and the integer M is initialized to 1. For each subset of available primes, represented by B, the product of those primes (denoted by factor) is computed. The factor which is closest to GOAL is stored in M, and the set B which corresponds to it is stored in C.
- When this processing is done, M is inserted into the table in position $[i, j]$, and the primes represented in C are added to those represented in A.

In principle, every subset of the n primes is processed. However, the processing breaks off as soon as the product of those primes is found to be greater than or equal to GOAL. (That is, no larger subsets are processed.)

procedure SAVE(B, factor, kk)

begin
  if (TOTAL $+ kk \geq n - i + j$) **then**
    return;
  -- The purpose of this test is to insure that there is at least one prime factor available for each dimension. For instance, if there are $68 = 2 * 2 * 17$ processors, then with this test, the third row of the table becomes (2, 2, 17); without the test it becomes (4, 17, 1).
  if ($|$factor $-$ GOAL$| < |M -$ GOAL$|$) **or**
    ($M = 1$ **and** factor $\leq P$) **then begin**
      $M \leftarrow$ factor;
      $C \leftarrow B$;
    end if
  -- Without the second part of this if test, the factor 17 in the example in the comment above would never get included.
end

procedure SEARCH(B: array[1..n] of **integer**; k, kk, factor: **integer**)
-- k holds a position in the list of possible primes
-- kk holds the number of primes used so far for this (i,j) entry

begin
  if $k > n$ **then return**;
  while ($A(k)$ **and** ($k < n$)) **do**
    $k \leftarrow k + 1$;
  call SEARCH(B, $k + 1$, kk, factor);
  nextfactor $\leftarrow$ factor $* p_k$;
  if nextfactor $\geq$ GOAL **then**
    begin
    call SAVE(B, factor, kk);
    $B(k) \leftarrow$ **TRUE**;
    $kk \leftarrow kk + 1$;
    call SAVE(B, nextfactor, kk);
    end
  else
    begin
    $B(k) \leftarrow$ **TRUE**;
    $kk \leftarrow kk + 1$;
    call SEARCH(B, $k + 1$, kk, nextfactor);
    end
end

procedure PRE-ANALYZE

begin

Denote the number of processors by P. Sort the prime factors of P into an increasing sequence $p_1 p_2 \ldots p_n$. Insure that $n \geq 7$ by appending additional "prime" factors of $P + 1$ as needed.

Allocate the following global variables:

GOAL: **real**;

TOTAL, M, i, j, l: **integer**;

A, B, C: array[1..n] of **boolean**;

end

```
procedure MAIN
    begin
        call PRE-ANALYZE;
        for i = 1 to 7 do begin
            -- do row i
            GOAL ← P^(1/i);
            TOTAL ← 0;
            A[1..n] ← FALSE;
            for j = 1 to i do begin
                -- do j'th entry in row i
                B[1..n] ← FALSE;
                C[1..n] ← FALSE;
                M ← 1;
                call SEARCH(B, 1, 0, 1);
                -- Merge C into A:
                for l = 1 to n do begin
                    if C[i] then begin
                        A[i] ← TRUE;
                        TOTAL ← TOTAL + 1;
                    end if
                end for
                table[i, j] ← M;
            end for
        end for
    end
```

3. For each TEMPLATE directive, fill in the fields of the `cells` entries (see page 53) of the dope attached to the symbol table entry for the template name, as follows:

begin
  if there is a DISTRIBUTION directive for this template **then**
    Fill in the `dist` field from the DISTRIBUTION directive.
    if the DISTRIBUTION directive specifies a processors array **then**
      Fill in the `strip_len` and `multiplier` fields from the `slen` and `mults` fields of the `processors_dir`.
    else If the number of processors is a compile-time constant, fill in the `strip_len` fields from the default strip length table constructed above, and compute the multipliers in the same way as they were computed for processor arrays above. If the number of processors is not a compile-time constant, give the first dimension of the template all the processors, and give all the other template dimensions a strip length of 1.
    end if
  else Give the template the default distribution. If the number of processors is a compile-time constant, fill in the `strip_len` fields from the default strip length table, and compute the multipliers as above. If the number of processors is not a compile-time constant, give the first dimension of the template all the processors, and give all the other template dimensions a strip length of 1.
  end if
end

4. For each symbol

begin
  if there is an ALIGN directive for this symbol (i.e., if the associated `align_dir` is not empty; see page 65) **then**
    Fill in the `subspaces`, `complete`, and `replicated` tables of the dope for that symbol:
      The `subspaces` are filled in from the `alignee_dims`. The associated data allocation functions come from the `stride` and `offset` fields of those `alignee_dims`.
      The `complete` table (i.e., the completion structure) is filled in from the `template_dims` having non-nil `scalar_expr`.
      The `replicated` table is filled in from the `template_dims` having a nil `scalar_expr`.
  else
    Give the symbol the default allocation.
  end if
end

6.3 Other functions

1. By inspecting the subscript expressions for each VCarrayref node in the dotree, fill in its `uses` table (see page 51).

2. For all arrays, fill in the `num_folds` fields of each data subspace, as described in the next section. This corresponds to local dope, as described above in Section 4.3.1, page 48. (There are no iteration subspaces as yet; as they are created, in ITER, their `num_folds` fields will be filled in as well.)

3. For each array which will be allocated statically, fill in the size field in the symbol table entry for that array with the product of the `num_folds` expressions for the dimensions of the array.

6.4 Calculating the *num_folds* of a data subspace

For each array symbol, we fill in the `num_folds` of its data subspaces, as follows:

We denote the number of folds in the j^{th} dimension (which is the contribution to the memory depth of the array in the j^{th} dimension) by $\mathit{num\_folds}_j$. Of course, $\mathit{num\_folds}_j$ will be a run-time variable unless the size of the array[1] is known at compile time. In any case, its value is available to every processor (and is the same for every processor). It is computed as follows:

Let f_j denote the data allocation function for the subspace corresponding to the j^{th} dimension of A. Equivalently, if s_j is the j^{th} subscript of A, then

$$v_j = f_j(s) = \mathit{alloc\_stride}_j * s_j + \mathit{alloc\_offset}_j.$$

f_j is normalized so that it generates 0-based results. (This just means that the partial virtual addresses v_j are each 0-based.)

Let DLB_j and DUB_j refer to the declared lower and upper bounds in dimension j, respectively.

Let T_j denote the (declared) template extent in the j^{th} dimension. (T_j is only needed in the case of BLOCK distribution.)

In the case of BLOCK(n) or CYCLIC(n) distribution, let n_j denote n, the (declared) block size in the j^{th} dimension. That is, n_j is the parameter appearing after the keyword BLOCK or CYCLIC, as BLOCK(n_j) or CYCLIC(n_j).

In the following formulas, we use "normalized" lower and upper bounds, defined as follows:

$$NLB_j = \begin{cases} DLB_j & \text{if } f_j \uparrow \\ DUB_j & \text{if } f_j \downarrow \end{cases}$$

$$NUB_j = \begin{cases} DUB_j & \text{if } f_j \uparrow \\ DLB_j & \text{if } f_j \downarrow \end{cases}$$

Defined in this way, we have $f_j(NUB_j) \geq f_j(NLB_j)$, unless the array is a 0-sized array.

SERIAL distribution

$$\mathit{num\_folds}_j = \max\{0,\ f_j(NUB_j) - f_j(NLB_j) + 1\}$$

BLOCK distribution

$$\mathit{num\_folds}_j = \left\lceil \frac{T_j}{\mathit{strip\_length}_j} \right\rceil$$

BLOCK(n) distribution

$$\mathit{num\_folds}_j = n_j$$

[1] in the case of BLOCK distribution, the size of the corresponding template.

NOTE: once *num_folds* has been computed, there is no difference between BLOCK and BLOCK(n) distributions. So from here on in this report, BLOCK(n) distributions will not be considered separately.

CYCLIC distribution

$$num\_folds_j = \begin{cases} 0 & \text{if } DUB_j < DLB_j \text{ That is, if the array is 0-sized} \\ \left\lfloor \dfrac{f_j(NUB_j)}{strip\_length_j} \right\rfloor - \left\lfloor \dfrac{f_j(NLB_j)}{strip\_length_j} \right\rfloor + 1 & \text{otherwise} \end{cases}$$

CYCLIC(n) distribution

$$num\_folds_j = \begin{cases} 0 & \text{if } DUB_j < DLB_j \text{ That is, if the array is 0-sized} \\ n_j * \left(\left\lfloor \dfrac{f_j(NUB_j)}{n_j * strip\_length_j} \right\rfloor - \left\lfloor \dfrac{f_j(NLB_j)}{n_j * strip\_length_j} \right\rfloor + 1 \right) & \text{otherwise} \end{cases}$$

We formerly optimized this so that if, for instance, the programmer declared an array to be distributed CYCLIC(1000) but the array only had 3 elements, we would only allocate 3 elements. This turned out to be more trouble than it was worth, so we eliminated it.

The main problem occurred in parameter passing—the caller might only allocate 3 elements but the callee might not know the size at compile time and therefore allocate a positive multiple of 1000, the multiple being determined at run time. If the dummies are passed explicit shape with the sizes being also passed as dummy arguments, this could result in some of the dummies being located by the callee in non-existent memory.

Some of our original automatically generated regression tests had small arrays that were distributed CYCLIC(1000); this is what led us to this attempt at optimization. We now believe these tests were unrealistic, and have changed them.

Our feeling now is that a user who specifies CYCLIC(1000) has some reason for expecting at least 1000 elements per processor to be allocated. We will not try to second-guess this decision.

Note that in the case of SERIAL, CYCLIC, and CYCLIC(n) distributions, num_folds_j is based on the declared lower bound in dimension j, so that, for instance, an array declared as $A(1000 : 2000)$ will not have space reserved for any values of the subscript less than 1000.

Also note that in the case of BLOCK distribution, num_folds_j is based on the template size, not on the array size. (Of course if there is no explicit template, then the array itself is the template, and so the array size is also the template size.)

Essentially the same formulas can also be used to compute the number of folds of an iteration subspace. The following changes have to be made:

- Letting ϕ_j denote the iteration allocation function and S_j denote the stride of the iteration subspace, the formulas for *NUB* and *NLB* become somewhat more complex:

$$NUB_j = \begin{cases} UB_j & \text{if } \phi_j \uparrow \text{ and } S_j > 0, \text{ or } \phi_j \downarrow \text{ and } S_j < 0 \\ LB_j & \text{if } \phi_j \downarrow \text{ and } S_j > 0, \text{ or } \phi_j \uparrow \text{ and } S_j < 0 \end{cases}$$

$$NLB_j = \begin{cases} LB_j & \text{if } \phi_j \uparrow \text{ and } S_j > 0, \text{ or } \phi_j \downarrow \text{ and } S_j < 0 \\ UB_j & \text{if } \phi_j \downarrow \text{ and } S_j > 0, \text{ or } \phi_j \uparrow \text{ and } S_j < 0 \end{cases}$$

Thus, $\phi_j(UB_j) \geq \phi_j(LB_j)$ unless the iteration subspace is 0-sized.

- The data allocation function f_j is replaced consistently by the iteration allocation function ϕ_j.

- The guards for CYCLIC and CYCLIC(n) distributions (that check for a 0 size) are changed from

$$DUB_j < DLB_j$$

to

$$\big((S_j > 0) \wedge (UB_j < LB_j)\big) \vee \big((S_j < 0) \wedge (UB_j > LB_j)\big)$$

6.5 Adjusting *alloc_offset*

The computation for *num_folds* in the last section allocates storage, in the case of SERIAL and CYCLIC distributions, only for the array elements actually specified, and for CYCLIC(n) distributions, only for the blocks actually used. To be consistent about this and to avoid addressing non-existent memory, the *alloc_offset* part of the data allocation functions have to be adjusted. We do this as follows:

SERIAL distribution

$$alloc\_offset_j \leftarrow alloc\_offset_j - f_j(NLB_j)$$

CYCLIC distribution

$$alloc\_offset_j \leftarrow alloc\_offset_j - \left\lfloor \frac{f_j(NLB_j)}{strip\_length_j} \right\rfloor * strip\_length_j$$

CYCLIC(n) distribution

$$alloc\_offset_j \leftarrow alloc\_offset_j - \left\lfloor \frac{f_j(NLB_j)}{n_j * strip\_length_j} \right\rfloor * n_j * strip\_length_j$$

We refer to this adjustment as “squishing”.

6.6 Function return values

We continue the running example of section 5.1.3 (page 59). DATA adds the following dope information (as represented in the dump file) to the symbol table for the function return value array_func_result:

```
Dope for "ARRAY_FUNC_RESULT", Owner: "ARRAY_FUNC"

        Regular Dope:
  Irank: 1
  Address: "ARRAY_FUNC_RESULT"
  Charlen: <NIL>
  Dim 1
    low:  1
    high: FETCH(@2)
  all replicated = TRUE
  SubSpace Table:
    Dim: 1
      SubSpace
        folds: FETCH(@2)
        as:    1
        ao:    -1
          Cell
            slen: 1
            mult: 1
            vbar: 0
            Dist: serial, extent: FETCH(@2)

  Replicated cells  Cell Table:
    Dim: 1
      Cell
        slen: 4
        mult: 1
        vbar: FETCH("_HPF_MYNODE")
        Dist: block, shadow: 0 , extent: 4

  Completion


        AFuncSizeInfo <null>

        AltDopeInfo <null>
```

Chapter 7

LOWER

7.1 Further lowering of array expressions

LOWER removes ARCS and SZLCON nodes from the dotree. Then it turns each triple node into a vINDEX node. Since each vINDEX node that comes from a FORALL statement is indexed (see below) by a positive number referring to the position of the variable it represents in the FORALL header, we let the index child of these new vINDEX nodes be indexed by the numbers -1, -2, $\ldots$, where -1 represents the first vTRIPLE in an expression, -2 the second, and so on, so that conforming dimensions have the same index.

In effect, this amounts to replacing each statement containing Fortran array syntax with an equivalent FORALL statement. Although this replaces a more specialized construct by a more general one, there is no loss of efficiency in the compiler's generated code.

In this way, the expression in Figure 5.2 is changed so that it looks like Figure 7.1. As before in such pictures, a table of subscripts with more than one element is enclosed in a dotted box.

LOWER also inserts triple nodes that are implicit in the CIR, together with their associated vINDEX nodes, so Figure 5.4 becomes Figure 7.2.

7.2 Lowering of FORALL statements

For FORALL statements which can be executed in parallel, LOWER also turns FAINDEX nodes into vINDEX nodes. Since a vINDEX node carries all the information about a forall index (including its bounds), the FORALL header information does not have to be represented separately in the dotree. Therefore the FORALL node itself is replaced by a VSTORE node. Thus, Figure 5.6 becomes Figure 7.3.

If a FORALL statement cannot be executed in parallel, LOWER replaces it by the equivalent nest of DO loops. In any case, after LOWER there are no more FORALL nodes in the dotree.

Either of the following expressions in the body of a FORALL statement causes that FORALL to be lowered to a nest of sequential DO loops:

- A statement function having a FORALL index in one of its arguments.

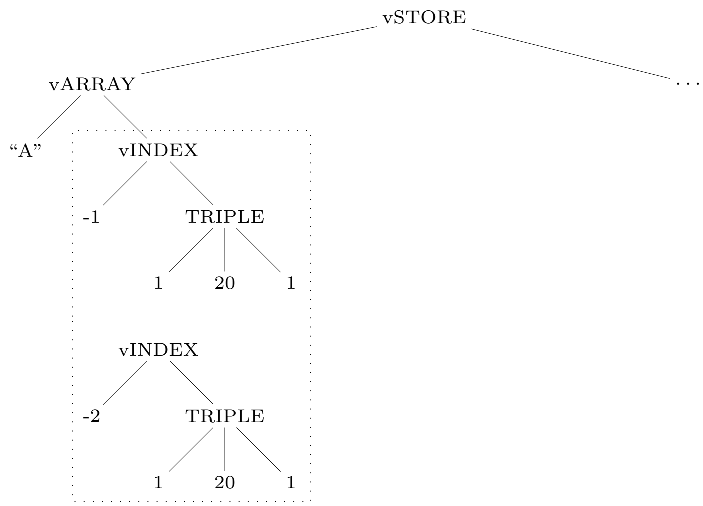


Figure 7.1: $a(:,\ :) = \ldots$ after LOWER

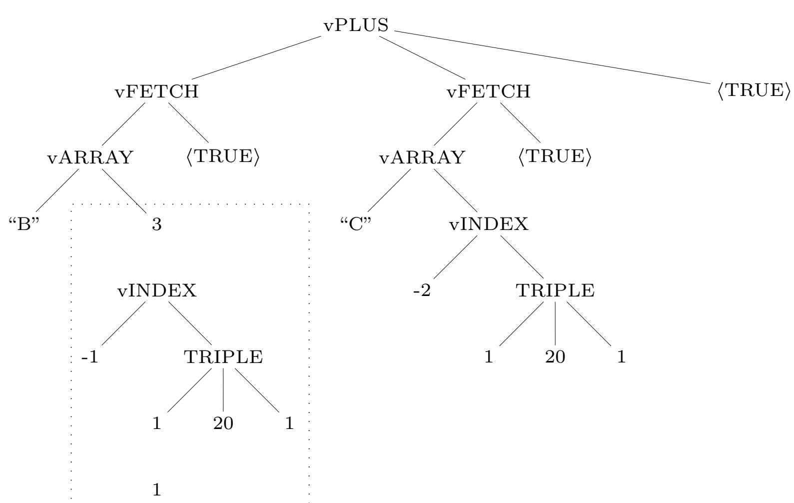


Figure 7.2: $B(3,\ :,\ 1) + C$ after LOWER

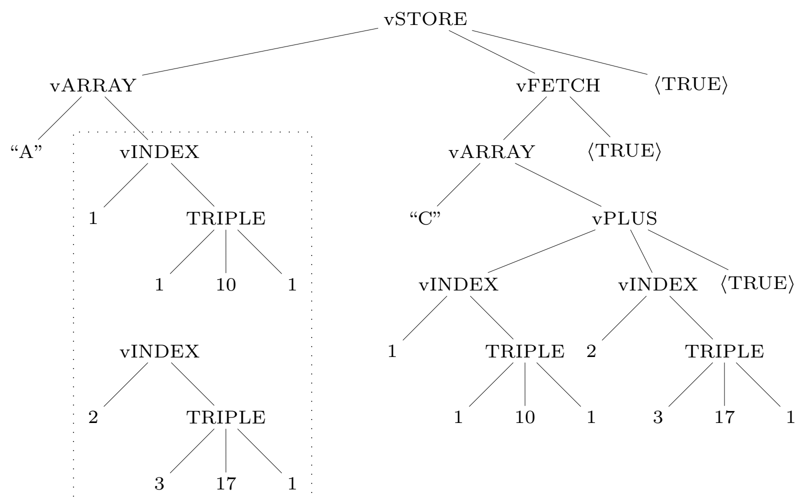


Figure 7.3: **forall** $(I = 1\!:\!10,\ J = 3\!:\!17) \quad A(I,\ J) = C(I{+}J)$ after LOWER

- An array constructor containing a FORALL index.

These are both cases that could be reconsidered in the future. They would both need additional design work.

Any form of the general HPF FORALL construct (sometimes called the “block FORALL”) that we support is conceptually replaced by a sequence of single-statement FORALL statements, after which each of these statements is lowered as above. (The bounds of the FORALL may have to be evaluated into temporaries first, to avoid evaluating them more than once.)

7.3 Array-valued functions

We continue with the running example of section 5.7 (pages 59, 63, and 64; see also page 76).

Computing the size of the returned value of an array-valued function presents a problem which is exhibited by that example:

Since the array-valued function array_func is the entire argument of the intrinsic function SUM, the size of its returned value cannot be inferred from the expression it is in (as it could be if for instance array_func were one of two children of a vector plus operation—the size might then be able to be inferred from the other child). In order to lower the function SUM, STRIP needs to have its argument in the form of a simple array expression, and so array_func needs to be assigned to a temporary. To construct this temporary, we need to know the size (at least as a run-time expression). This size is computed later, in ARG. Right now, LOWER just performs a partial lowering of the calling sequence (in the main program), generating the following representation. (In reading this, note the convention that names starting with the "@" sign represent compiler-generated temporaries.)

SEQ (a `dt_seq` node)
  NEWSCOPE
  SEQ
    `@3` = LFUNC(scalar_func, (3)) `--` LFUNC refers to a "local function", i.e., a function that is **contain**ed in the routine being compiled.
    AFUNC(`@4`, LFUNC(array_func, (a(1:5:1), `@3`)))
    s = **sum**(`@4`(1:AFTER_ARG_UB(`@4`,1))) `--` AFTER_ARG_UB(`@4`,1) is a stand-in for the upper bound in dimension 1 of the array `@4`. It will be fixed up later, in the ARG phase.
  ENDSCOPE

The AFUNC operator is used to add the return temp `@4` to array_func as a hidden argument. We do it this way so that `@4` can be allocated on the stack frame of the caller. If we did not do this, then the callee would have to allocate its return temp on the heap, and this return temp would then have to be explicitly deallocated by the caller after the call returned.

Of course in order to allocate `@4`, its size needs to be known, at least as an expression. Computing an expression for this size is the work of the ARG phase. So in the meantime, `@4` is just entered into the symbol table as a compiler-generated allocatable array, together with any directive information for it that comes from directives for array_func in its explicit interface. The allocation of `@4` is inserted later, in ARG.

The symbol table information for the temporary array `@4` looks like this:

```
Dope for "@4", Owner: "AF"

        Regular Dope:
  Irank: 1
  Address: "@4"
  Charlen: <NIL>
  Dim 1
    low:  1
    high: AFTER_ARG_UB("@4",1)
  all replicated = TRUE
  SubSpace Table:
    Dim: 1
      SubSpace
        folds: FETCH(ARRAY(ACTPOS(1),[FETCH(ACTPOS(2))]))
        as:    1
        ao:    -1
          Cell
            slen: 1
            mult: 1
            vbar: 0
            Dist: serial, extent: FETCH(ARRAY(ACTPOS(1),[FETCH(ACTPOS(2))]))

  Replicated cells  Cell Table:
    Dim: 1
```

```
      Cell
        slen: 4
        mult: 1
        vbar: FETCH("_HPF_MYNODE")
        Dist: block, shadow: 0 , extent: 4

  Completion


        AFuncSizeInfo <null>

        AltDopeInfo <null>
```

The two significant things here are:

1. The high bound in dimension 1, which is expressed using the operator `AFTER_ARG_UB`. This operator will be replaced later, in ARG.

2. The expressions containing ACTPOS. These will also be replaced by ARG.

Since `@4` is a stack temporary, it is implicitly deallocated by the ENDSCOPE statement.

The body of the function array_func is lowered slightly so it looks like this:

SEQ (a `dt_seq` node)
  ENTRY (a `dt_entry` node)
    PREAMBLE (field of the `dt_entry` node)
      `@1` = n
      `@2` = d(`@1`)
  array_func_result(1:`@2`) = d(1:n)

Finally, PURE functions occurring in FORALL constructs are handled as a special case. If a PURE function is array-valued, the transformation outlined above is still performed but not in LOWER: It is performed later in the ARG phase (see page 97).

7.4 Array constructors containing implied DO loops

In general, array constructors are just passed through TRANSFORM and then handled by the Middle End. This works provided everything in the constructor is replicated. Any mapped data in the constructor has to be extracted from the constructor and evaluated into a replicated temp which is then used in the constructor. If there is an implied DO loop in the constructor, then LOWER has to create an explicit DO loop in the dotree, because the loop variable has to be available when evaluating any mapped data in the constructor.

So (unless LOWER can determine that the array constructor is simple enough to be passed straight through to the Middle End) LOWER creates explicit DO loops in the dotree corresponding to implicit DO loops in array constructors. These loops are created before the punting processing in LOWER, and also before the handling of the SALLOCATE statement, also in LOWER.

As in the previous section, LOWER may not know the size of an expression that is pulled out of such an array constructor. For example, in the expression

`(/ a(1:i), i = n:m /)`

LOWER does not know the size of the array section `a(1:i)`, since it contains an implied do variable, which is meaningless outside the context of the implied do loop.

Therefore, a temporary to hold the value of such an expression cannot be completely specified. To solve this problem, we have introduced a new operator—DEFER—which enables us to name a temporary and indicate that later phases of TRANSFORM need to complete the job of allocating it and filling in its symbol table dope information.

The source construct

`call faa( (/ fbb(a(1:i, 1:j)), i=1:n, j=1:m /) )`

is transformed by LOWER into following dotree code:

```
ALLOCATE(prev, SIZE = 0)
prevsize = 0
do i = 1, n
    do j = 1, m
        NEWSCOPE
        DEFER(T1, (/fbb(a(1:i, 1:j))/))
        -- LOWER doesn't know the size of this; after ITER we will know it. Also note
           that T1 is a mapped array; it gets its mapping from fbb. The array construc-
           tor is needed because in effect we are reshaping a 2-dimensional object to a 1-
           dimensional one.
        newsize = prevsize + size(T1)
        SALLOCATE(new, newsize) -- new is a replicated array
        new(1:prevsize) = T1
        prevsize = newsize
        new(prevsize+1:newsize) = T2
        DEALLOCATE(prev)
        prev ==> new -- pointer assignment
        nullify(new)            ENDSCOPE
    end do
end do

call faa(T1)
```

The lowering of the DEFER operator happens later in ITER.

7.5 Other lowerings

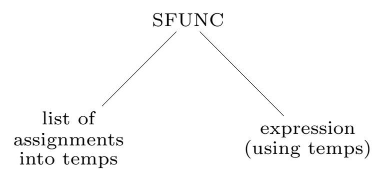
7.5.1 DATA statements

Since data objects initialized by DATA statements are generally distributed, the initialized values will differ on different processors. Therefore, the results of the initialization cannot be reflected in the executable image, which is loaded identically onto each processor. For this reason, DATA statements are turned into an equivalent sequence of assignment statements.

Since a variable initialized by a DATA statement is automatically a SAVE variable, we make sure that these assignment statements are executed only on the first invocation of the routine.

7.5.2 Statement functions

A statement function in the CIR, and in the dotree before LOWER, looks like this:



The left-hand child is a sequence of assignment statements created by the Front End which assign the actual arguments of the statement function into (compiler-generated) temporaries. The right-hand child is just the expression which defines the statement function, written in terms of those temporaries.

The assignments in the left-hand child of the statement function are inserted at the statement level prior to the statement in which the function occurs. The function is then in effect macro expanded by simply replacing the call to the function by the expression defining the function (i.e. the right-hand child shown above).

7.5.3 SPREAD

Each SPREAD node is replaced by an EXPLICIT_SPREAD node. This is a node with four children:

- The expression being spread.
- The `ncopies` parameter of the SPREAD.
- The `dim` parameter of the SPREAD.
- The index (i.e., the first child) of the vINDEX node representing the dimension being spread to.

The first three of these children are just the corresponding children of the original SPREAD node, and come from the source program. The last child, which is easily computed here, is stored in this node so that it does not have to be recomputed later.

7.5.4 PACK, RESHAPE, and UNPACK

These operators are punted, as explained above in Section 3.8, (pages 30ff).

7.5.5 MAX and MIN operators

These operators each take a list of arguments, and return the max or min of them all. LOWER replaces these operators by a nest of BINMAX or BINMIN operators, which are binary max or min operators.

7.5.6 WHERE masks

WHERE masks are attached as children to all expression nodes to which they apply. Expression nodes under the ELSEWHERE clause are given the negation of the WHERE mask. The WHERE operator itself is deleted from the dotree.

7.5.7 Special processor 0 handling

LOWER performs the array copying necessary for alternate entries and exits, and also for NAMELIST handling, as described above in Section 3.4.

7.5.8 Filling in global dope

LOWER creates global dope from symbol table dimension information for arrays that are not assumed-shape, as mentioned above in Section 4.3.1. For assumed-shape arrays, LOWER creates global dope as expressions referring to fields in the passed-in dope vector.

Chapter 8

ITER

Iter fills in the iteration space information for all the nodes in the dotree.

8.1 The leaves

Constants and Front-End temporaries (i.e. vaCONTOK and vaTMPTOK nodes) are given an iteration space which is all replicated.

The iteration space for each scalar is simply the same as that of its data space (i.e. the replicated and completion structures are the same; there are no subspaces).

The only other kind of leaf node is an arrayref. They are handled as follows:

- The replicated and completion structures are copied from the dope (i.e. from the data space).
- Data subspaces corresponding to scalar subscripts are added to the completion structure.
- Primary data subspaces can be matched with corresponding iteration subspaces by the primary cell constraints.

At this point, the total iteration virtual processor (VP) space at the arrayref consists of

- all the iteration subspaces which match up with data space structures in this manner, and
- all the iteration completion and iteration replicated structures

together with additional iteration subspaces which represent iteration coordinates not represented in primary data subspaces. Let us use $\mathcal{I}$ to denote the family of these additional iteration subspaces.

Similarly, the total data virtual processor space at the arrayref consists of

- all the data subspaces which match up with iteration space structures as above, and
- all the data completion and data replicated structures

together with additional data subspaces which have complex subspace uses. Let us use $\mathcal{D}$ to denote the family of these additional data subspaces.

To complete the specification of the map from iteration space to virtual processor space, we need to

- map each element of $\mathcal{I}$ somewhere, and
- make sure that the dimensions of the virtual processor space corresponding to the elements of $\mathcal{D}$ are accounted for.

In principle, there might be many ways in which this could be done. The method we use is to match up as much as possible, elements of $\mathcal{I}$ with elements of $\mathcal{D}$. Precisely, we use the following method:

begin
  while $\mathcal{I} \neq \varnothing$ **do begin**
    if $\mathcal{D} \neq \varnothing$ **then**
      Extract a subspace i from $\mathcal{I}$ and a subspace d from $\mathcal{D}$. Copy the information from d (cell place, distribution) into i.
    else
      Make all the subspaces in $\mathcal{I}$ serial.
    end if
  end
  if $\mathcal{D} \neq \varnothing$ **then**
    foreach $d \in \mathcal{D}$ **do**
      Create a completion subspace in $\mathcal{I}$ with the same cell place as d and value 0.
  end if
end

8.2 Intermediate nodes

First perform a bottom up tree walk over each expression.

1. If the node is a FETCH or a vFETCH, take its iteration VP space from its child.
2. Otherwise, if the node is a scalar,
   - If it has at least one child whose iteration VP space is not all replicated, use the VP space of one such child.
   - Otherwise, give the node a completely replicated iteration VP space.
3. Otherwise, ignore the node.

Next, implement the owner-computes determination of iteration VP spaces by taking the iteration VP space of the left-hand side and performing a top-down walk of right-hand side of the statement, attaching that VP space to each node, stopping only at scalar nodes and at terminal vFETCH nodes, all of which have previously been given an iteration VP space by the bottom-up walk above.

Actually, we do much better than a naive implementation of the owner-computes rule. For instance, if we are compiling a statement of the form $A(:) = B(:) + C(:)$ and B and C align with each other but not with A, then iter will indeed determine the iteration space of the root of the statement to be that of A. However, DIVIDE will later discover that B and C are in the "same region", and will compute their sum into a temporary T aligned with each of them—this is a local computation—and then generate a statement moving T to another temporary aligned with A. Thus, only half as much data is moved as in a naive implementation of the owner-computes rule.

Regardless of any other optimizations which may be made, a PURE function is assigned an iteration VP structure as follows:

- Subspaces corresponding to unnamed dimensions must have *strip_length* 1.
- If all such subspaces in the parent of the function have *strip_length* 1, then the iteration VP structure of the function can be identical with that of its parent. In any other case, motion will be necessary, and the rest of the iteration VP structure can be assigned in any convenient way.

Note that the iteration cells in the VP structure of a PURE function other than those corresponding to unnamed dimensions must be either completion or replicated. The cells corresponding to the FORALL indexes are represented as replicated cells, but the indexes themselves are actually privatized variables (see the footnote on page 152).

8.3 Iteration space bounds for temporaries

After ITER has run, each computation node in the dotree has an iteration space structure associated with it. The following two phases—ARG and DIVIDE—may introduce new temporaries. The iteration space associated with these temporaries (or with their associated FETCH or STORE parents) has to be determined with some care, because of the following problem. Suppose we have introduced a temporary T as follows:

$$\begin{array}{l}\textbf{forall}\ I = 2{:}10{:}2\\ \qquad T(I) = B(I/2)\end{array}$$

It would seem natural to say that the iteration bounds of T (i.e. of the vSTORE) are $1{:}5{:}1$. However, by the time STRIP begins, the iteration bounds of all nodes in the statement must be the same, and so at the end of DIVIDE, all such bounds are normalized so that they are equal (see Section 10.7). If we were to choose $1{:}5{:}1$ for the iteration bounds of every node in this statement, we would run into problems with the integer division in the subscript of B.[1]

We solve this problem as follows: when a temporary is created, it takes the iteration bounds for any named dimension from the corresponding bounds of some iteration space on the right-hand side of the statement. (All such bounds must be the same, since they come from the same FORALL header.) Unnamed dimensions are created with lower bound 1 and stride 1, since there is no way that an implicit division can be represented in a TRIPLE on the right-hand side.

[1] If it were not for the fact that the integer division throws away the remainder, there would be no problem, for at the same time as the iteration range at B was adjusted from $2{:}10{:}2$ to $1{:}5{:}1$, the data *alloc_stride* would also be multiplied by 2, so that the same virtual processor would be referenced.

8.4 Handling the DEFER operator

The dotree statement

DEFER($T1$, $fbb(a(1\!:\!i,\ 1\!:\!j))$)

is created by LOWER in the process of lowering array constructors containing implied DO loops (see page 82). When ITER encounters such a statement, it does three things:

1. It synthesizes the iteration space of the DEFER node from that of its second child.
2. It generates an SALLOCATE for the temporary $T1$—it can do this because it now knows the size of $T1$—and fills in the symbol table dope information for $T1$, which it will then use in generating iteration space information for later statements created by LOWER (see page 82).
3. It replaces the DEFER node with a local punted vector assignment.

Thus, the DEFER statement is replaced by the following dotree code:

SALLOCATE($T1$, sizeof(fbb))
$T1 =$ (/ $fbb(a(1\!:\!i,\ 1\!:\!j))$ /) `--` a punted statement

8.5 Intrinsic functions

The intrinsic functions

- SIZE
- SHAPE
- LBOUND
- UBOUND
- NUMBER_OF_PROCESSORS
- PROCESSOR_SHAPE

are inlined: NUMBER_OF_PROCESSORS and PROCESSOR_SHAPE become constants (if known at compile time) or references to the internal variable `numnodes`, as explained previously in Section 3.8 (pages 30ff). The other four intrinsics are turned into expressions computed from information in the iteration space.

Chapter 9

ARG

The ARG phase handles arguments to I/O calls, external procedures, and some intrinsics. In the course of doing this, it has to perform some special transformations to handle alternate exits from I/O statements correctly.

PURE functions are handled differently from other functions by ARG. Therefore, none of the processing in this chapter applies to PURE functions unless specifically stated. The PURE function processing is detailed in Sections 9.4 and 9.6.

9.1 I/O: arguments

Children of I/O nodes have had iteration spaces attached to them by ITER. ARG uses this iteration information in copying all necessary data into processor 0 temporaries, after which the I/O call is made on processor 0. The iteration information is necessary to determine the size of the temporaries.

The data copied to processor 0 includes all the arguments of the call together with all the information in the specification list. For example:

dimension $A(1000)$, $B(1000)$
!hpf$ template $T(1000)$
!hpf$ distribute $T($**block**$)$
!hpf$ align $A(i)$, $B(i)$ **with** $T(i)$
read (**unit** $= A(j)$) B

requires both $A(j)$ and $B(:)$ to be moved to processor 0. In addition, B must be copied back from processor 0 after the read, because it was modified by the I/O statement. In addition to the list of arguments of the I/O statement, some specifications (e.g. an IOSTAT parameter) may also be modified by the call, and so they must also be copied back.

Of course, if the data is already on processor 0, we do not perform an unnecessary copy.

The determination of whether a temporary is actually necessary is based on the location of the referenced data, in the following way:

if the data is not modified **then**
  if the data is completely available on processor 0 (but may also live elsewhere; e.g. if the data is replicated) **then**
    no temporary is needed
  else
    the data must be copied into a processor 0 temporary
  end if
else `--` the data is modified
  if the data lives *only* on processor 0 **then**
    no temporary is needed
  else if the data is completely replicated **then**
    we perform the optimization described on page 20: we do not create a temporary. Instead, we simply read the data into its destination on processor 0 and then update the rest of the replicated data on the other processors.
  else
    the data must be copied into and out of a processor 0 temporary
  end if
end if

As mentioned above, determining if a value is modified is more complicated than simply asking if the statement is a read or a write. For example, in the statement

print($*$, IOSTAT $= A(5)$) B, C

the array A is actually modified.

The structure of the actual code for an I/O statement without alternate exits is shown in Figure 9.1.

`dt_seq`
  BEGINSCOPE
  `dt_io`, as explained in Section 4.2.1.
    `cond:` **if** *my_processor*`()` $==0$
    `pre_stmts:` an empty `dt_seq` node
    `specs:` a `dt_seq` node containing a list of `dt_io_item` nodes. Each of these nodes is either NULL or represents a specification (e.g., UNIT number, label of FORMAT statement, IOSTAT variable, etc.).
    `items:` a `dt_seq` node containing a list of `dt_io_item` or `dt_io_loop` nodes (each `dt_io_loop` node itself containing `dt_io_item` nodes). Each `dt_io_item` node specifies a value to be read or written, together with any copy-in/copy-out code that is needed to support that operation.
    `post_stmts:` an empty `dt_seq` node
  ENDSCOPE

Figure 9.1: How I/O with no alternate exits is handled.

9.2 I/O: alternate exits

Making the copies to and from processor 0 explicit in the dotree also makes it necessary to transform the I/O call to handle alternate exits properly. For instance, if for the call

read ($*$, **err** = 200), a

we were to generate the equivalent of

```
        temp = a
        if (my_processor() == 0) then
            read (*, err = 200), temp
        end if
        a = temp
        ...
200     ...
```

then if the READ statement exited early to line 200, processor 0 would miss the copy-out code a = temp. It would go directly to line 200, and all the other processors would stall, waiting for processor 0 to send them the value being copied out.To avoid this, we perform a transformation as follows: The statement

read ($*$, **err** = 200, **end** = 300, **eor** = 400), a

results in the dotree structure under the `dt_io` node of Figure 9.1 being modified in the following way:

- Each `io_item` is modified so that the original labels (which we will call `errlabel`, `endlabel`, and `eorlabel`) are replaced with new labels, which we denote by `terrlabel`, `tendlabel`, and `teorlabel`.
- Each `io_item` node, which has the following structure,

  `dt_io_item`
    `pre_stmts:` a `dt_seq` node containing copy-in code for the item being written or read
    `item:` the actual item being written or read
    `post_stmts:` a `dt_seq` node containing copy-out code as needed

  is modified by adding the following code to the beginning of each `pre_stmts` and `post_stmts` section:

  ```
  replicate(err_occurred) -- from processor 0
  if (err_occurred) then
      goto errlabel
  end if
  ```

  Similar code is inserted for END and EOR exceptions as specified. If no error occurs, all processors will reach this code, a FALSE value of err_occurred will be propagated to all the processors, and each processor will fall through to the next statement. If on the other hand, an error does occur, it will occur on processor 0, which will therefore not get to this point—it will jump to terrlabel. The remaining processors will wait at this point to receive the value of err_occurred from processor 0. Since this value will now be TRUE, these processors will then jump to the normal error processing line (errlabel), as specified in the source program.

- It remains to show how processor 0 is managed so that it sends a FALSE value of err_occurred when necessary and jumps to errlabel. To make this happen correctly, the code in Figure 9.1 is replaced by the code in Figure 9.2. Each of the error processing segments in the `post_stmts` section is guarded by **if** (FALSE), and so can only be reached by jumping into it. This only happens on processor 0, when an error happens and a jump is executed to terrlabel, tendlabel, or teorlabel. Processor 0 then replicates the appropriate error condition and jumps to the actual error location (errlabel, endlabel, or eorlabel).

- Finally, if only IOSTAT is specified, then there is no label associated with it in Fortran. Therefore, processor 0 makes no jump. Instead, it simply falls through to the next statement, which is the one after the comment "only if IOSTAT was specified". This is where processor 0 sets the variable iostat_occurred. This variable is replicated in the following statement.

9.3 Function and procedure calls

All function and procedure calls (remember that we are excluding PURE functions here) are raised to the statement level.

For global function calls, ARG determines where copy-in/copy-out is needed, and inserts the necessary copies to and from temps to implement this. Again, the iteration space information attached to the actual arguments is used to construct any needed temporaries.

When a mapped array section is passed to a subroutine and the dummy has the INHERIT attribute, no copy-in should take place. However, because of some difficulties in the interface with the Middle End, we have found it necessary for the time being to perform a copy-in in this case. This is what is done:

This will be fixed in the future so that no copy-in will take place.

Say the array section is

$$A(LB_1 : UB_1 : S_1,\ \ldots,\ LB_n : UB_n : S_n)$$

and denote the iteration allocation function for the j^{th} iteration dimension by ϕ_j. Then this actual argument will be replaced by the whole array T, where the array T is defined as follows:

- The base address of T is the same as the base address of A.

- The declared lower bound of T in dimension $j = DLB_j^T = 1$.

- The declared upper bound of T in dimension $j = DUB_j^T$ is

$$\frac{UB_j - LB_j}{S_j} + 1$$

- The data allocation function of T is given by

```
    dt_io
       cond: if my_processor() == 0
       pre_stmts: a dt_seq node:
          err_occurred = FALSE -- only if specified
          end_occurred = FALSE -- only if specified
          eor_occurred = FALSE -- only if specified
          iostat_occurred = FALSE -- only if iostat and none of err, end, eor are specified

       specs: modified as indicated.
       items: modified as indicated.

       post_stmts:
          if (FALSE) then -- only if ERR was specified
terrlabel        err_occurred = TRUE
                 replicate(err_occurred)
                 goto errlabel
          end if

          if (FALSE) then -- only if END was specified
tendlabel        end_occurred = TRUE
                 replicate(err_occurred) -- only if ERR was specified
                 replicate(end_occurred)
                 goto endlabel
          end if

          if (FALSE) then -- only if EOR was specified:
teorlabel        eor_occurred = TRUE
                 replicate(err_occurred) -- only if ERR was specified
                 replicate(end_occurred) -- only if END was specified
                 replicate(eor_occurred)
                 goto eorlabel
          end if

          -- only if IOSTAT was specified:
          if (my_processor() == 0) then
             if (iostat_variable ≠ 0) then
                iostat_occurred = TRUE
             end if
          end if

          -- For I/O without any items (!), we must put this code here, one instance for each
             kind of error exit specified, since there are no items to put this code into. This
             code also handles the replication of the iostat_occurred variable.
          replicate(err_occurred)
          if (err_occurred) goto err_label
tiostatlabel     continue

       ... ⟨rest of program⟩
errlabel -- original line 200
       ...
endlabel -- original line 300
       ...
eorlabel -- original line 400
       ...
```

Figure 9.2: How Alternate I/O Exits are Handled.

$$
\begin{aligned}
\mathit{alloc\_stride}_j(T) &= \mathit{alloc\_stride}_j(A) * S_j \\
\mathit{alloc\_offset}_j(T) &= \mathit{alloc\_offset}_j(A) + \phi_j(LB_j) - \phi_j(DLB_j) \\
&= \mathit{alloc\_offset}_j(A) + (LB_j - DLB_j) * \mathit{alloc\_stride}_j(A)
\end{aligned}
$$

9.3.1 ACTPOS processing

A caller may need to create a temporary for an actual argument, and the size of that temporary may depend on the values of other actual arguments. Consider for instance, what is entailed in compiling a call to the function foo, where foo is declared as follows:

foo(A, N, B)
  integer N, A(:), B(:)
  !hpf$ distribute A (**block**(B(N)))

The value of N at run-time is needed to find $B(N)$, which in turn is needed to compute the local size of the array A that is being passed. Arbitrarily complicated examples of this nature can be constructed.

The fundamental idea, however, is that dependences of this nature create a partial order on the actual arguments. The compiler computes this partial order, and then walks the actual arguments in some order compatible with it in order to generate code for any temporaries that are needed.

9.3.2 Array-valued functions

We continue with the running example introduced in section 5.1.3 (pages 59, 63, and 64; also pages 76 and 79).

Array-valued function calls, which were partially processed in LOWER, are further processed in ARG to allocate the temporary into which the function is to be computed. In this processing, the operator AFTER_ARG_UB that was inserted in the code by LOWER is replaced by the expression that ARG computes for the actual upper bound, based on its ACTPOS processing. In addition, the ACTPOS expressions are replaced by temps into which the values will be computed at run-time. Thus, the dope information for the temp `@4` (see page 80 is now put in its final form:

```
Dope for "@4", Owner: "AF"

        Regular Dope:
  Irank: 1
  Address: "@4"
  Charlen: <NIL>
  Dim 1
    low:  1
    high: FETCH("@4_ub1")
  all replicated = TRUE
  SubSpace Table:
    Dim: 1
      SubSpace
```

```
        folds: FETCH("@60")
        as:    1
        ao:    -1
          Cell
            slen: 1
            mult: 1
            vbar: 0
            Dist: serial, extent: FETCH("@58")

  Replicated cells  Cell Table:
    Dim: 1
      Cell
        slen: 4
        mult: 1
        vbar: FETCH("_HPF_MYNODE")
        Dist: block, shadow: 0 , extent: 4

  Completion


        AFuncSizeInfo <null>

        AltDopeInfo <null>
```

where the temporaries `@4_ub1`, `@58`, and `@60` are assigned to by the ACTPOS processing, as shown below.

In addition, ARG uses some utilities from DIVIDE to discover that the actual and the dummy have different mappings, so that a copy-in/copy-out is needed. An array temporary `@53` is constructed for this purpose. Its dope looks like this:

```
Dope for "@53", Owner: "AF"

        Regular Dope:
  ...
  Dim 1
    low:  1
    high: 5
  ...
```

The executable code in our running example (in the caller) is then modified by ARG to look like this:

SEQ (a `dt_seq` node)
  NEWSCOPE
  @3 = LFUNC(scalar_func, (3)) -- LFUNC refers to a “local function”, i.e., a function that is **contain**ed in the routine being compiled.
  SEQ
    SALLOCATE(@53)
    @53(1:5:1) = a(1:5:1) -- The dummy variable d is mapped differently than the actual a is, so a copy-in is necessary.
    -- Now the ACTPOS processing begins:
    @54 = @3
    @55 = 1
    @56 = @53(@54 - (1 - @55))
    @4_ub1 = @56
    @57 = 1
    @58 = @53(@54 - (1 - @57))
    @59 = 1
    @60 = @53(@54 - (1 - @59))
    -- End of the ACTPOS processing.
    SALLOCATE(@4)
    AFUNC(@4, LFUNC(array_func, (@53(1:5:1), @3)))
    a(1:5:1) = @53(1:5:1) -- This is the copy-out.
  s = **sum**(@4(1:@4_ub1))
  ENDSCOPE

9.4 Handling SPREADs

A SPREAD in an assignment statement is deleted by DATA. This is possible for the following reason: a SPREAD cannot occur at the top of the left-hand side of an assignment statement. So the left-hand side of an assignment statement explicitly represents the total iteration space. Therefore, when a SPREAD is deleted by DATA, the iteration space of the resulting node is determined by the original parent of the SPREAD, which, if it is the top-level vSTORE, gets this information synthesized from the left-hand side.

If the SPREAD causes data motion, that fact is subsequently determined by DIVIDE just by looking at iteration space information—cf. the same_region() function on page 106.

For a function argument, however, the SPREAD could actually be the top-level node, and once it was deleted, there would be no way to reconstruct the iteration space. Therefore, DATA does not delete such SPREADS, but leaves them for ARG to handle.

ARG takes SPREADs in arguments to I/O or external procedure calls and brings them up to the statement level; i.e., the first child of each SPREAD is remapped into a partially replicated temporary, and the SPREAD is replaced by a fetch of that temporary.

9.5 Intrinsic functions

The Fortran intrinsic functions CSHIFT and EOSHIFT are replaced by an equivalent set of array assignments.

The XXX_SCATTER functions are handled by first introducing a new assignment statement to express the initialization of the result by the base, and then generating a second statement which expresses the scatter operation itself. The remainder of the work in lowering these functions is done by STRIP.

9.6 PURE functions in FORALL constructs

ARG performs all the lowering for PURE functions in a FORALL construct; if the function is array-valued, this includes the lowering that otherwise would have been performed by LOWER (see Section 7.3).

All intrinsic functions and functions in the HPF library are by definition PURE, so this processing in ARG applies to them as well. In particular, functions which otherwise would have been inlined by the compiler, such as CSHIFT and the reduction intrinsics, are handled by this alternate method when they appear inside FORALL constructs.

The lowering is performed as follows:

- Any data references in actual arguments are remapped if necessary before the FORALL so that all their unnamed dimensions are distributed serially, and so that their named dimensions and completion and replicated structures conform with the iteration space of the FORALL.

  Any named FORALL dimension not corresponding to a primary data subspace in the actual argument is turned into a conformant replicated cell in the iteration space of the actual.[1]

- A `dt_group_forall` node is created to contain the body of the FORALL. Under this node, the actual arguments to the PURE function are privatized over the FORALL dimensions (i.e. over the named dimensions).[2]

  This amounts to a copy-in. No copy-out is necessary, because arguments to a PURE function must all be IN parameters. (See the HPF language specification, section 4.3.1.1.)

- If the PURE function is array-valued, the temporary into which the function is computed (e.g. $t1$ in the example in Section 7.3) is handled similarly: its unnamed dimensions are all serial, and its named dimensions disappear, being privatized. In this case, a copy is then made to a "de-privatized" temporary.

  The NEWSCOPE and ENDSCOPE surround the `dt_group_forall`.

- After the `dt_group_forall`, the result is remapped to its final location. This action may be unnecessary if no initial remapping was needed.

[1] This replicated cell actually represents a privatization over the corresponding logical processor dimension – note that it is not over the corresponding virtual processor dimension (as in the next footnote), which is folded over that logical processor dimension)

[2] In this implementation, privatized VP subspaces are represented by replicated cells; see the footnote on page 152

The dt_group_forall node enables the STRIP phase to handle this sequence of statements under it as a unit. In particular, the STRIP phase is then able to create a nest of strip loops around the entire body of the dt_group_forall which reflects the FORALL indexes.

ARG attaches to the dt_group_forall node an iteration space that STRIP uses to create the loop nest. This iteration space (call it I) is constructed as follows:

- The table of replicated cells of I is initialized from the table of replicated cells of the ispace at the root of the statement.
- The completion structure of I is taken from the completion structure of the ispace at the root of the statement.
- Each vector array reference on the left-hand side of the statement is visited. For each such reference, its iteration subspaces are processed as follows:
  - Any unnamed dimension or dimension with nested vINDEX nodes (e.g., a TRIPLE whose lower bound contains a reference to a FORALL index) is transformed into a replicated cell in I. No loop will be created for this dimension—the vector construct (which is of necessity serial) will be passed through to the middle end.
  - A named dimension that is complex is made serial and inserted into I, and a replicated cell is added to I to hold the original cell place.
  - Any other named dimension (which must be primary) is copied as is into the subspace table of I.

Thus, for instance, the source construct (where F is a PURE function)

$$\textbf{forall}\ (I = LB_1 : UB_1 : S_1,\ J = LB_2 : UB_2 : S_2)$$
$$A(I,\ J,\ :,\ :) = F(B(I,\ :,\ :,\ :))$$

which appears as a vector assignment statement after LOWER, is transformed by ARG into the following three constructions:

1. remap B

$$T1(I,\ :,\ :,\ :) = B(I,\ :,\ :,\ :)$$

where the last three dimensions of $T1$ (which correspond to the unnamed dimensions of B in the FORALL construct) are distributed serially and the first dimension of $T1$ conforms with the iteration space of the FORALL index I. The FORALL primary dimension for J, which is not represented in the subscripts of B, becomes a replicated cell for $T1$. If the FORALL iteration space has any completion or replicated structure, that is also present in the data space of $T1$.

2. lower the PURE function

NEWSCOPE
dt_group_forall
$T2(:,\ :,\ :) = T1(I,\ :,\ :,\ :)$
AFUNC($R1(:,\ :)$, EFUNC(F, LIST($T2(:,\ :,\ :)$)))
$R2(I,\ J,\ :,\ :) = R1(:,\ :)$
ENDSCOPE

where all the unnamed dimensions (i.e. the dimensions indicated by :) are serial, and where $T2$ and $R1$ are privatized over the I and J dimensions. If $T2$ would be scalar, it is omitted, and $T1$ is used directly as an actual argument in the AFUNC. Similarly, if $R1$ would be scalar, it is omitted, and $R2$ is used directly as the return value in the AFUNC.

3. remap the result to A

$$A(:,\ :,\ :,\ :) = R2(:,\ :,\ :,\ :)$$

Of course the initial and final remapping are omitted if they are not needed.

As a help in reading the code in the file `arg_calls.c` where these transformations are implemented, $T1$ is produced from B and $R2$ is produced from A by the function `itr_tmp_for_pure_fn_arg()`. $T2$ is produced from $T1$ and $R1$ is produced from $R2$ by the function `itr_tmp_for_privatized_arg()`.

If the FORALL construct contains a mask, that mask is attached by ARG to the `dt_group_forall` node as a dtb attribute. STRIP will retrieve this and use it to create an IF guard.

Chapter 10

DIVIDE: Setting Up for Motion

10.1 Some formulas

The purpose of DIVIDE is to insure that corresponding nodes of an expression represent elements on the same processor, for corresponding values of the iteration space indexes. To understand the way this is done, we first explain some notation and formulas that are used for this purpose[1]:

For any quantity in either data or iteration space, the *effective* value of that quantity is the image of that quantity in virtual processor space. So for instance, if LB is the lower bound of an iteration subspace, and if

$$\phi(x) = \mathit{alloc\_stride} * x + \mathit{alloc\_offset}$$

is the iteration allocation function for that subspace, then $\phi(LB)$ is the *effective lower bound* for that iteration subspace. Similarly, if S is the source stride for that iteration subspace (as would be the case if that iteration subspace came from a triple $\langle LB\colon UB\colon S\rangle$), then $\mathit{alloc\_stride} * S$ is the *effective stride* for that iteration subspace.

If an iteration subspace has lower bound LB_j and stride S_j—for instance, if it arises from a construct such as

FORALL $(\ldots,\ I = LB_j\colon UB_j\colon Sj,\ \ldots)$

—then denoting the successive coordinate values of the iteration subspace by

$$\left\{ LB_j + S_j * i : 0 \le i \le \frac{UB_j - LB_j}{S_j} \right\}$$

we have

$$\begin{aligned} v_j &= \phi_j(LB_j + S_j * i) \\ &= \phi_j(LB_j) + \mathit{alloc\_stride}_j * S_j * i \end{aligned}$$

So the coordinate in virtual processor space of this iteration is determined by the iteration number (i) together with the effective lower bound and the effective stride of the iteration subspace.

[1] See [14] for the notation v, $\bar{v}$, and $\hat{v}$ used below

The maps from the iteration virtual processor space to the iteration logical processor space are given by the following formulas:

$$\bar{v}_j = \begin{cases} \left\lfloor \dfrac{v_j}{num\_folds_j} \right\rfloor & \text{BLOCK distribution} \\ v_j \bmod strip\_length_j & \text{CYCLIC distribution} \\ \left\lfloor \dfrac{v_j}{n_j} \right\rfloor \bmod strip\_length_j & \text{CYCLIC}(n_j) \text{ distribution} \end{cases}$$

10.2 Regions

We say that two nodes in an expression are in the same region if and only if the conformance preference between them is honored. (This is just another way of expressing the condition in the first sentence of this chapter.) Ignoring for the moment complications that arise in the presence of replicated subspaces (we will show how to handle these below), this amounts to saying (cf. [14]) that the iteration VP subspaces of the two nodes match up so that

- corresponding subspaces have the same *strip_length*; and if the *strip_length* > 1, then also
  - corresponding iteration allocation functions are equal,
  - corresponding iteration distribution functions are equal, and
  - corresponding iteration cells have the same cell place. (Remember that two cells have the same *cell place* iff they have the same *strip_length* and *multiplier*.)

An equivalent formulation is this: two computation nodes are in the same region if and only if corresponding iteration VP cells of each node

1. Both have *strip_length* 1; or
2. Both
   (a) are identically distributed (i.e. both serial, or both BLOCK, or both CYCLIC, or both CYCLIC(n) with the same value of n).
   (b) have the same value of *num_folds* if they are both BLOCK; have the same *strip_length* if they are both CYCLIC or CYCLIC(n).
   (c) have the same *multiplier*.
   (d) have the same effective stride.
   (e) have the same effective lower bound.

Because the effective strides and effective lower bounds are the same, corresponding values of v_j are equal for the two nodes; and because the *strip_length* (or *num_folds*) and *multiplier* (and the parameter n if appropriate) are the same, corresponding values of $\bar{v}$ are the same for the two nodes.

10.3 The tasks of DIVIDE

DIVIDE moves up all motion to the statement level. This happens in the following way:

- Remote fetches are assigned into a temp (aligned with the parent of the remote fetch) if necessary so that they become the complete right-hand side of an assignment statement. A fetch of the temp is then substituted for the remote fetch.
- Any other nodes which are not in the same region as their parents are similarly assigned into temps aligned with their parents, and fetches of those temps are substituted for the original nodes.

  If there are replicated cells in the iteration VP space, the same-region test outlined above must happen *after* the algorithm on page 104 is performed.

For example, a typical remote fetch looks like this, after having been raised up if necessary so that it is the complete right hand side:

$$A(1{:}N,\ 1{:}M) = B(V(1{:}N),\ 1{:}M)$$

DIVIDE then pulls out the $V(1:N)$ and spreads it into a temporary, yielding

$$\text{PBRDCST}(W(1{:}N,\ 1{:}M),\ \text{DIMS} = \{2\},\ V(1{:}N))$$
$$A(1{:}N,\ 1{:}M) = B(W(1{:}N,\ 1{:}M),\ 1{:}M)$$

Of course, the second statement is not expressible in Fortran, but this is effectively what the internal representation looks like. Also, the first statement is not really what happens. The root operator is actually a REMAP_STORE, and the information about the spread dimension is found in the iteration VP spaces of the two sides.

In the course of doing this motion processing, DIVIDE updates the `stmt_kind` attribute of each statement. This field is set whenever a statement is constructed in the dotree, and previous phases may have modified this field. DIVIDE further refines this classification. The complete list of possible values for this attribute is as follows:

`STMT_ILLEGAL` Indicates an error.

`STMT_TBD` A temporary annotation, to be refined by a later phase. Indicates that the children of the operator must be examined to determine the correct `stmt_kind`. This never occurs after DIVIDE.

`STMT_IO` An I/O statement, subsequently lowered (in ARG) to a `dt_io` construct.

`STMT_ASSIGN` An scalar or vector assignment statement, before DIVIDE. DIVIDE gives a more specific `stmt_kind` to these statements.

`STMT_IGNORE` STRIP will just pass this statement through without modifying it.

`STMT_FLOW` A flow of control statement.

`STMT_CALL` The statement is a call, or the right-hand side of the statment is a function call.

`STMT_LOCALIZE` This statement does not need strip loops made for it, but subscripts in it need to be localized.

`STMT_PRIVATIZED_ASSIGN` Used in connection with FORALL statements containing pure functions.

`STMT_PASSIGN` A pointer assignment. This tells STRIP to generate dope vector stores.

`STMT_REPLICATE` Replicate the data from processor 0.

`STMT_RANDOM` For `random_number()`.

`STMT_ALLOC` For heap allocation.

`STMT_PRIVATE_ALLOC` For heap allocation with no dope vector. This is only used in `stp_datamot.c` when allocating a 1-dimensional temporary array on the heap for the purpose of passing values to runtime routines.

`STMT_DV_DUMP` For dumping a dope vector.

`STMT_DV_COPY` For copying a dope vector.

`STMT_SCATTER` A statement derived from an XXX_SCATTER intrinsic.

The following statement kinds are assigned by ARG:

`STMT_REDUCTION_INTRINSIC` A reduction intrinsic without a DIM argument.

`STMT_DIMREDUCE_INTRINSIC` A reduction intrinsic with a DIM argument.

The following statement kinds are assigned by DIVIDE:

`STMT_LOCAL_ASSIGN` The statement is a local assignment statement. See Section 12.1, page 121. In particular, no interprocessor communication is needed in this statement.

`STMT_LIBRARY_RESHAPE` A remapping store implemented by calling a library routine rather than by generating inline code including message-passing calls.

`STMT_RESHAPE_ASSIGN` The statement is a remapping store. See Section 13.1, page 142.

`STMT_REMOTE_ASSIGN` The statement is a remote store. See Section 13.3, page 146

`STMT_RFETCH_ASSIGN` The statement is a simple store of a remote fetch. See Section 13.2, page 144.

`STMT_NEIGHBOR_ASSIGN` The statement is a nearest-neighbor assignment statement. See Chapter 15, page 169.

`STMT_NN_FETCH` A nearest-neighbor fetch. That is, a fetch that fills in the shadow edges in support of a nearest-neighbor operation.

`STMT_NUMA_REPLICATED_STORE` Strip needs to generate a loop to store data to a replicated left-hand side. See Chapter 16, page 215.

`STMT_LOCLOC_ASSIGN` Used for an assignment to a dope vector for an object with a non-trivial completion. We need to guard the assignment, because the object does not live on every processor. This is done by putting the ispace of the right-hand side (the %LOC) on the root of the store; STRIP then adds the appropriate guard.

The actual enumeration is of type `dt_stmt_kind`, and is defined in the file `udb/master/dote_udb.h`.

All of the transformational functions are raised to the statement level. The XXX_SCATTER function

RESULT = XXX_SCATTER(SRC, BASE, INDX1, . . . , INDXn, MASK)

is replaced by

RESULT = BASE
RESULT = XXX_SCATTER(SRC, INDX1, . . . , INDXn, MASK)

When the statement is a nearest-neighbor computation, DIVIDE suppresses the same-region check, since all motion will be handled later by STRIP for these statements. But it does tag each array name in the statement as being a nearest-neighbor array. This is because the addressing for these arrays has to be handled differently by STRIP (see page 173).

In addition, DIVIDE generates local remapping for subroutine parameters which are involved in nearest-neighbor computations (see page 184).

Finally, DIVIDE makes sure that the computed return value of a function is laid out as specified in the function interface (or has the default layout if not specified). An extra remapping store may be introduced at the function return to implement this.

An important optimization is performed when generating motion: if the parent iteration space has a (primary) subspace not represented in the child—this constitutes an implicit spread of the child's values into that subspace—then the temporary into which the child is moved is created from the parent's iteration space *except* that the unmatched primary subspace is replaced by a replicated cell in the temporary. By doing this, we insure that only 1 copy of the child's values are moved to each processor, rather than moving the child's values a large number of times (the number of times being the `num_folds` of the unmatched primary cell).

10.4 The handling of replicated cells

Here is a way of dealing with replicated cells in iteration VP spaces which is general enough to handle most cases without introducing unnecessary motion:

The replicated cells in the iteration VP space of a leaf node are synthesized from the replicated cells in its data VP space, using the replicated cell constraints.

The replicated cells in the iteration VP space of a non-leaf node are synthesized from those of its children by the following algorithm. This algorithm is not intended to be a sketch of an implementation—it just specifies the results we expect. The actual implementation could be quite different.

First we note that if a replicated cell R has a strip length l which can be factored, say $l = pq$, then R can be replaced by two replicated cells R_1 and R_2, where

$$\begin{aligned} &\mathit{multiplier}(R_1) = \mathit{multiplier}(R) && \mathit{strip\_length}(R_1) = p \\ &\mathit{multiplier}(R_2) = \mathit{multiplier}(R) * p && \mathit{strip\_length}(R_2) = q \end{aligned}$$

The method then consists of the following steps:

1. If a replicated cell R in one child corresponds to a product $\prod C^i$ of cells in another child, in the following sense:

$$\begin{array}{rrcl}
(\forall i) & \mathit{multiplier}(R) & \leq & \mathit{multiplier}(C^i) \\
 & & \leq & \mathit{multiplier}(C^i) * \mathit{strip_length}(C^i) \\
 & & \leq & \mathit{multiplier}(R) * \mathit{strip_length}(R) \\
 & \prod \mathit{strip_length}(C^i) & = & \mathit{strip_length}(R)
\end{array}$$

   then replace R by a family of replicated cells R^i (by successive applications of the process noted above) so that R^i has the same cell place as C^i.

   We refer to this procedure as *factoring* the replicated cell.

2. After performing step 1 until no more subdivisions of replicated cells can be done, if all the children of the parent node have the same replicated cell (by "the same" we mean that the cell places are the same), then that cell occurs also in the parent node's iteration VP space.

3. If all except one child of the parent node have the same replicated cell, and the remaining child (call it child c) has a non-replicated cell C in the same cell place which is either

   - a scalar cell; or
   - a primary cell whose iteration coordinate, if it appears at all in the iteration VP space of any child c' ($c' \neq c$) containing the replicated cell R, appears only in that child (c') as the iteration coordinate of a serial cell (i.e. a cell with $\mathit{strip\_length} = 1$)

   then that non-replicated cell is synthesized to the parent in the place of the replicated cell, and if there was a serial cell as indicated above, then that serial cell does not appear in the parent. An example of this appears in Section 17.2.3.

4. Any replicated cells which are not accounted for by steps 3 and 4 result in explicit motion being inserted. This motion is implemented as a remapping store.

After all nodes in a statement have been processed in this manner, the only question which has to be dealt with is how to handle ordinary vector stores (that is, stores at which no motion has already been indicated).

Any replicated cell on the right-hand side of an assignment statement can be ignored. The context of the assignment statement is taken from the root of the statement.

If there is a replicated cell on the left-hand side of an assignment statement which does not correspond to a replicated cell on the right-hand side, then a REMAP_STORE needs to be inserted to spread the right-hand side over the range of processors indicated by the replicated cell on the left-hand side.

10.5 An implementation of same_region()

Figure 10.1 illustrates an outline for code to implement same_region(), which takes account of most of the considerations of the preceding sections.

In reading the pseudo-code, think of same_region() as a boolean function which compares the iteration VP spaces of a node (the "parent") and each of its children in an expression tree. It returns TRUE if and only if no motion is necessary between the two nodes.

The parent's iteration VP space is passed in to the function as Π and the child's iteration VP space is passed in as Γ.

---

function same_region(iteration_VP_space Π, Γ):**boolean**
begin
  if Π and Γ are the same (i.e. if they are dagged) **then return TRUE**;
  if Γ consists entirely of replicated cells **then return TRUE**;
  for each subspace π in Π, make sure that there is either
    1. a subspace γ in Γ such that
      begin
      The iteration index of γ = the iteration index of π; and
      multiplier$[\gamma]$ = *multiplier*$[\pi]$; and
      γ and π are identically distributed (i.e. both serial, or both BLOCK, or both CYCLIC, or both CYCLIC(n) with the same value of n); and
      γ and π have the same value of *num_folds* if they are both BLOCK; have the same value of *strip_length* if they are both CYCLIC or CYCLIC(n);and
      γ and π have the same effective stride and the same effective lower bound if they are not serial.
      end
  or else that there is
    2. a replicated cell in Γ with the same cell place as π;
  and if neither of these is true, **return FALSE**;

  for each replicated cell in Π make sure that there is a replicated cell in Γ with the same *multiplier* and *strip_length*; otherwise **return FALSE**;

  for each completion in Π make sure that there is either
    1. a completion in Γ with the same *multiplier* and *strip_length* and the same $\bar{v}$;
  or else that there is
    2. a replicated cell in Γ with the same *multiplier* and *strip_length*;
  and if neither of these is true, **return FALSE**;

  return TRUE;
end

Figure 10.1: An implementation of same_region().

10.5.1 PURE functions

PURE functions require some special handling, as follows:

- Any unnamed dimension in the iteration space of an argument to a PURE function must be serial.
- All the other iteration subspaces of arguments to a PURE function must match up with corresponding iteration subspaces of the function itself.

10.6 Optimizing TRANSPOSE

In general, the TRANSPOSE operator is inlined in a completely straightforward manner.

However, when compiling a statement such as

$$A = \texttt{TRANSPOSE}(B)$$

where the transposed dimensions of B are either serial or block, we generate more efficient code. Our current implementation has some further restrictions, the main ones being:

- A and B must be whole arrays, each of rank 2.
- There must be no non-trivial alignments.
- There must be no replicated or completion cells.

In such a case, we know that each processor holds a single chunk of the array B that is contiguous in array element order. (It is also contiguous in memory because there are no non-trivial alignments). The same is true for A. Further, each such chunk of B corresponds exactly to a unique chunk of A—the only thing that has to be taken account of is the local transpose that has to take place in addition to the data motion.

We compute the data motion as follows: say A is distributed onto the processor arrangement

$$P(\mathit{strip\_length}_1^A, \mathit{strip\_length}_2^A)$$

B must then be distributed onto the processor arrangement

$$Q(\mathit{strip\_length}_1^B, \mathit{strip\_length}_2^B)$$

where

$$\begin{aligned} \mathit{strip\_length}_1^B &= \mathit{strip\_length}_2^A \\ \mathit{strip\_length}_2^B &= \mathit{strip\_length}_1^A \end{aligned}$$

A particular processor holding an element of B can be represented as $\bar{v}^B = (\bar{v}_1^B, \bar{v}_2^B)$, where (letting the multiplier corresponding to the i^{th} dimension of B be denoted by $\mathit{multiplier}_i^B$)

$$\bar{v}_i^B = \left\lfloor \frac{\bar{v}^B}{\mathit{multiplier}_i^B} \right\rfloor \bmod \mathit{strip\_length}_i^B$$

Similarly, the processor holding an element of A can be represented as $\bar{v}^A = (\bar{v}_1^A, \bar{v}_2^A)$, where (letting the multiplier corresponding the i^{th} dimension of A be denoted by $\mathit{multiplier}_i^A$)

$$\bar{v}^A = \sum_{i=1}^{2} \bar{v}_i^A * \mathit{multiplier}_i^A$$

Since we are implementing a TRANSPOSE, the correspondence between the processor $\bar{v}^B$ holding an element of B and the processor $\bar{v}^A$ holding the corresponding element of A is given by

$$(\bar{v}_1^A, \bar{v}_2^A) = (\bar{v}_2^B, \bar{v}_1^B)$$

Thus, we have

$$\bar{v}^A = \left(\left\lfloor \frac{\bar{v}^B}{multiplier_2^B} \right\rfloor \bmod strip\_length_2^B \right) * multiplier_1^A + \left(\left\lfloor \frac{\bar{v}^B}{multiplier_1^B} \right\rfloor \bmod strip\_length_1^B \right) * multiplier_2^A$$

We generate in such a case a call to a library routine that uses this formula to move the data and perform the local transpose on each processor.

10.7 Normalizing the iteration space

As far as the STRIP phase is concerned, the precise iteration bounds in a statement are not important, so long as every iteration space in that statement has the same bounds.

In our first implementation, therefore, at the end of DIVIDE (when all the statements created before STRIP exist in the dotree) the iteration space of each computational node in every statement was normalized so they have the same bounds.

We no longer do this in DIVIDE. It turned out that this created subscript expressions that were too complicated for the data motion part of strip to analyze effectively for some of its optimizations. Therefore, this normalization happens on demand in STRIP in the course of localizing the subscripts of array expressions. (See page 130, item 1.)

For historical interest, however, and to show conceptually what is at stake, we indicate our original implementation here:

The particular bounds we use to normalize the iteration spaces to are almost arbitrary. We generally use the bounds of the iteration space of the root node of the statement.[2] Section 8.3 showed how we have arranged matters so that this can be done safely.

Let us denote the iteration bounds of the root of the statement by

$$\langle LB_{\text{new}},\ UB_{\text{new}},\ S_{\text{new}} \rangle.$$

If the old (i.e. original) iteration bounds of a node in the statement are denoted by

$$\langle LB_{\text{old}},\ UB_{\text{old}},\ S_{\text{old}} \rangle$$

then a vINDEX node (call it I) which came from a TRIPLE node (i.e. which represents an unnamed dimension) must be replaced by

$$\frac{I - LB_{\text{new}}}{S_{\text{new}}} * S_{\text{old}} + LB_{\text{old}}$$

Finally, the iteration allocation function is adjusted as follows:

[2]The one exception occurs when the right-hand side of the statement is a reduction intrinsic. In this case, the iteration space of the root has smaller rank than that of the array argument of the reduction, and so in this case, we normalize to the bounds of the array argument of the reduction.

$$alloc\_stride_{\text{new}} = \frac{alloc\_stride_{\text{old}} * S_{\text{old}}}{S_{\text{new}}}$$

$$alloc\_offset_{\text{new}} = alloc\_offset_{\text{old}} + alloc\_stride_{\text{old}} * LB_{\text{old}} - alloc\_stride_{\text{new}} * LB_{\text{new}}$$

In the case of a local statement (the term "local statement" is defined on page 121), the *alloc_stride* and *alloc_offset* could just be inherited from the root node, just as the iteration bounds are; this is not true for non-local statements, however.

As explained above, this processing is no longer performed in DIVIDE. The processing that takes its place in STRIP only modifies array subscripts, since there is no longer any need to keep the iteration allocation function up to date after STRIP.

Chapter 11

DIVIDE: Independent DO Loops

11.1 High-level design

We distribute iterations of `INDEPENDENT` do loops under certain conditions: The only procedure calls we allow inside distributed loops are procedures that are within the scope of an ON directive with a RESIDENT clause with no *res-object-list*. This guarantees that such a procedure accesses only data available in its local memory, and has the further property that no remapping will be necessary between actual and dummy arguments. So we make the following definition:

A *top-level independent do loop* is an independent do loop (i.e., a `dt_do_loop` tagged as `INDEPENDENT`), having the following properties:

1. Each procedure call within the loop lies within the scope of an ON directive with a RESIDENT clause with no *res-object-list*.
2. No array reference within the loop contains a subscript with a non-affine reference to an independent loop variable, or contains a subscript with a reference to more than one independent loop variables.
3. No array reference within the loop has two subscripts containing a reference to the same independent loop variable.
4. No punted construct is found within the loop.
5. There is no independent do loop node above it having the above properties.

Only loops within a top-level independent do loop will be processed as independent, i.e., will have their iterations distributed.

Now for each top-level independent do loop d, consider the tree of do tree nodes with d as its root. The term "tree" below will always refer to this tree.

1. For each path in the tree from the root d to a terminal node, we build an iteration space, as follows:

   (a) Non-independent loops in this path do not contribute to the iteration space.

(b) Each unnamed dimension (i.e. each "colon") corresponds to a serial subspace. (This is for simplicity; we assume that in general users will not mix the two idioms of independent loops and Fortran 90 array syntax.)

(c) Each independent loop on the path contributes a subspace to the iteration space. The corresponding cell places are allocated in some fashion. We don't worry too much about getting an optimal allocation, because the programmer can specify this precisely, using an ON clause.

If the programmer does use an ON clause that specifies an array, that array is referred to in our implementation as a `home_ref`. In other cases, the compiler may pick, where possible, one of the available array references to act as the `home_ref`.

We call this iteration space the *loop iteration space*. Each terminal node in the tree thus has an associated loop iteration space.

2. Each array reference in the tree has associated with it a *reference iteration space*, derived from the subscript map and the data space of the array, in the usual fashion.

3. Thus, at each array reference, there are two iteration spaces:

(a) The reference space for that array reference.

(b) The loop iteration space at that point in the tree.

If at each array reference, these two iteration spaces are the same, then the loop nest is capable of being executed in parallel—all the data is available where it needs to be accessed. This is the typical case.

In any other case, we have some work to do. An optimal solution to this problem involves the techniques of data optimization; we discuss this briefly on page 116 below. Here we indicate a simpler approach that will probably handle the most common cases.

(a) If all the references to an array A in the tree have the same loop iteration space, and also have the same reference iteration space, but the two spaces are not the same, then we perform a "copy-in/copy-out" of A in the conventional fashion:

- Create a temporary whose data space conforms with the (unique) loop iteration space for the references to A in the loop nest, and copy A into that temporary before the tree.
- Replace each reference to A in the tree by a corresponding reference to that temporary.
- Copy the temporary back to A after the tree. (This last step can be omitted if there were no assignments to A in the tree.)

(b) Otherwise, if there is no *true* dependence involving h in the loop nest[1], and if all the references to A that are stored to (e.g., that are the left-hand sides of assignment statements) have the same loop iteration space and the same reference iteration space, then

- For each reference to A whose reference iteration space is not the loop iteration space at that point, create a temporary for the whole array A whose data space is consistent with that loop iteration space, and assign (all of) A to that temporary before the

[1] Precisely, there is no true dependence at the level of one of the independent loops in the nest. Note that any dependence at the level of an independent loop in the nest would have to be a loop independent dependence, since the loop is independent.

tree. (Note that if two such references have the same reference iteration space and the same loop iteration space, they can use the same temporary.)

- Replace that reference to A by a corresponding refence to that temporary.
- After the tree, copy out the temporary corresponding to the stored reference(s) back to A (if the temporary was used for a stored occurrence). Again, we emphasize that this can happen for at most one temporary.

(c) Otherwise, for each two references to A at which the loop iteration spaces are different, we modify the loop iteration spaces at those two occurrences by making the subspaces contributing to the differences in mappings all serial. (The cell place that was used by such a subspace can be made either a completion cell or a replicated cell.) After this step, each occurrence of a given array has exactly one loop iteration space throughout the entire tree. The same copy-in/copy-out technique as used above in item 3b can then be used (without having to check for true dependences; we know that all the loop iteration spaces are now the same).

4. A data object that does not have the full complement of subscripts (i.e., one for each iteration subspace) is then constrained to be replicated over the missing dimensions, and is copied in and out as above if necessary to satisfy this constraint.

The remainder of the processing is handled later, by the STRIP phase. We specify it here simply for clarity:

1. The iteration space is used later in STRIP to localize the loop bounds and subscripts. For this purpose, the primary subspace corresponding to the loop is stored in a loop attribute called `independent_ispace`.
2. Any unnamed dimensions of course have to be turned into (serial) loops. These loops are inspected (in STRIP) for the possible presence of strip mining obstacles, which are removed in the usual manner.
3. The remainder of the loops as written are guaranteed to be semantically correct. Therefore no checking for strip mining obstacles has to be done for these loops. However, we can still perform loop fusion and exterior loop optimization. This is handled in the usual manner by dependence analysis.

11.2 Examples

Here we give four examples. The first three correspond to the situations listed under item 3 in the previous section.

11.2.1 Example for 3a

The program fragment in Figure 11.1 shows two references to the array h, each having the same loop iteration space and the same reference iteration space, but the two spaces are different. In this case the processing indicated above in item 3a applies: h would be copied to a temporary before this program fragment, and copied out afterwards.

```
!hpf$ distribute(block, block, block) :: h
!hpf$ independent
do k = 1, 100
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            h(j, i, k) = ...
            !hpf$ end on
        end do
    end do
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            ··· = h(j, i, k)
            !hpf$ end on
        end do
    end do
end do
```

Figure 11.1: Since the loop iteration space for each set $\langle i,\ j,\ k\rangle$ is consistent with the mapping of $h(i,\ j,\ k)$, each reference to h has the same reference iteration space, and each also has the same loop iteration space, and the two spaces are not the same.

11.2.2 Example for 3b

The program fragment in Figure 11.2 shows an array h whose references have two different reference iteration spaces.

However, there is no true dependence involving h, and there is only one statement storing to h. Since the loop iteration space at all the references to h is the same, this example can be handled as in item 3b above: since the loops are mapped to align with $h(i,\ j,\ k)$, a temp is created for h, with the same mapping as h, and before the loop

```
forall (i = 1 : 100, j = 1 : 100, k = 1 : 100)
    temp(j, i, k) = h(i, j, k)
end forall
```

is inserted. In the program fragment, $h(j,\ i,\ k)$ is replaced consistently with $temp(i,\ j,\ k)$. And finally, after the program fragment, *temp* is copied back out to h.

11.2.3 Example for 3c

The program fragment illustrated in Figure 11.3 again shows an array h with two different reference iteration spaces.

In this case, however, there is a true dependence involving h, and also there there are two different

```
!hpf$ distribute(block, block, block) :: h
!hpf$ independent
do k = 1, 100
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            if g(i, j) then
                ··· = h(i, j, k)
            else
                ··· = h(j, i, k)
            end if
            !hpf$ end on
        end do
    end do
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            h(j, i, k) = ...
            !hpf$ end on
        end do
    end do
end do
```

Figure 11.2: The i loops are mapped the same and similarly for the j loops (i.e., the loop iteration spaces at all the references to the array h are identical). The references to h have different reference iteration spaces, but we can still honor the distribution of the loop iterations.

reference iteration spaces for the references to h that are stored to. For both of these reasons, item 3c applies. Therefore, we serialize the first two dimensions of h (by a copy-in/copy-out). In effect, this removes the INDEPENDENT directive from both sets of i and j loops, and also means that we are not fully honoring the ON directive.

11.2.4 One more example

The program fragment illustrated in Figure 11.4 gives rise to the tree illustrated in Figure 11.5.

This program fragment has several interesting characteristics:

1. If both j loops are given the same cell place in iteration space, and if the i and j dimensions account for all of the processors[2], then the k loop will be serialized, since there will be nothing left to distribute it over. However, the i and both j loops can then be distributed and executed in parallel.

[2]that is, in the terminology of our internal representation, if there are no completion or replicated cells in the loop iteration space of the first $\langle i, j\rangle$ path down the tree.

```
!hpf$ distribute(block, block, block) :: h
!hpf$ independent
do k = 1, 100
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            if f(i, j) then
                h(i, j, k) = ...
            else
                h(j, i, k) = ...
            end if
            !hpf$ end on
        end do
    end do
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            if g(i, j) then
                ··· = h(i, j, k)
            else
                ··· = h(j, i, k)
            end if
            !hpf$ end on
        end do
    end do
end do
```

Figure 11.3: h has two different reference iteration spaces, and must be copied to a partially serialized temporary, in effect serializing the i and j loops.

2. If the k loop is distributed in a non-trivial manner, then the two j loops can only correspond to the same cell place in their iteration spaces if the iteration space for the $\langle i, j\rangle$ path (the first, or leftmost, path) has either a completion or a replicated cell in the place of the k cell in the iteration space for the $\langle i, j, k\rangle$ path. *If it does not*, the loop iteration spaces at the two references to b be different, and the same will be true at the two references to c. This in turn will force b and c to be completely serialized. The only computation that can be carried out in parallel in this case is the computations involving f.

3. If the statement involving f is changed to

$$f(i, m, k) = f(j, m, k)$$

then the two references to f in this statement have different reference iteration spaces (assuming that i and j are distributed non-trivially). Hence f must be serialized in its first dimension.

```
!hpf$ independent
do i = 1, n
   a(i) = ...
   !hpf$ independent
   do j = 3, s
      b(i, j) = ...
      c(j) = ...
   end do
   !hpf$ independent
   do j = 3, s
      b(i, j) = ...
      do m = 4, p
         d(i, j, m) = ...
         do n = 5, r
            e(i, j, m, n) = e(i, j, n, m)
            !hpf$ independent
            do k = 6, s
               f(i, m, k) = f(i, n, k)
            end do
         end do
      end do
      c(j) = ...
   end do
end do
```

Figure 11.4: An example with independent and non-independent loops.

11.3 Data optimization

The example in Figure 11.3 can actually be handled without serializing any of the dimensions of h, at the cost of introducing addition data motion. In effect, the compiler turns it into the code in Figure 11.6.

In this figure, no copy-out is necessary. The analysis the compiler would have to use to generate this code is the same as that used in data optimization: at each point in the program at which h is live, a "home" for h would be established; this just means a temporary array holding the values of h. The mappings of those homes would be determined by finding the graph of preferences between them and breaking those preferences that minimize a cost function. In this case, a home was established between the two i and j loop nests, and motion was created at that site.

Of course, the programmer could have written this all out explicitly. And in this particular example, an even simpler way to write the program would be as in Figure 11.7.

This last rewriting is only possible, however, because the example is so simple. If there were other statements and flow of control in the loops, this kind of rewriting might be impossible.

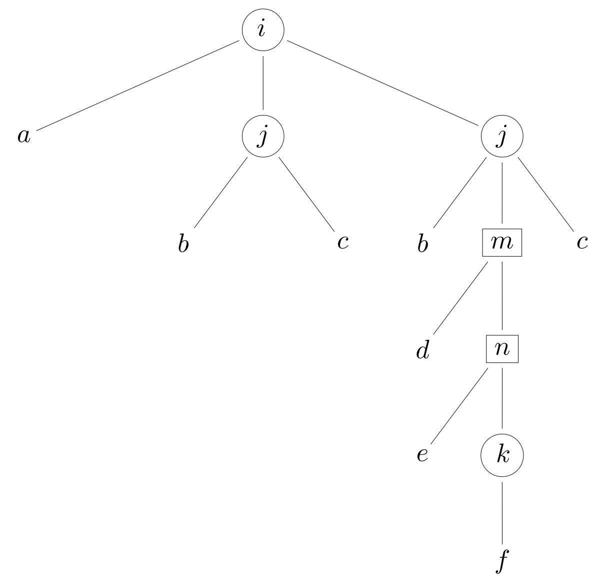


Figure 11.5: The tree corresponding to the program fragment of Figure 11.4. Nodes in circles represent independent do loops. Nodes in rectangles represent non-independent do loops. The remainder of the nodes represent statements.

11.4 An optimization for independent loops with indirect addressing

Consider the loop in Figure 11.8. Here the arrays x, y, and r are assumed to be replicated. The first two statements in the loop body set up the indirect addressing which is used in the remaining statements of the body.

Although the compiler cannot determine how to distribute iterations at compile-time, the iterations can certainly be performed independently, and without incurring any data motion. We do it roughly like this (the precise details are described subsequently):

- Take off the INDEPENDENT attribute of the DO loop.
- Execute the first two statements in parallel. That is, all processors execute the identical code. This is possible because the everything in these statements is replicated.
- Guard each of the last two statements so that only processors owning elements of the left-hand side of each statement execute that statement.

These guards are evaluated at run-time. Thus, we have actually distributed iterations of the loop (or at least those parts of the iterations that deal with mapped data), but the details of the distribution are not known until run-time.

Precisely, we first perform the following checks:

```
!hpf$ distribute(block, block, block) :: h
!hpf$ align temp with h
forall (i = 1 : 100, j = 1 : 100, k = 1 : 100)
temp(i, j, k) = h(j, i, k)
end forall
!hpf$ independent
do k = 1, 100
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            if f(i, j) then
                h(i, j, k) = ...
            else
                temp(i, j, k) = ...
            !hpf$ end on
            end if
        end do
    end do
    forall (i = 1 : 100, j = 1 : 100, f(i, j))
        temp(j, i, k) = h(i, j, k)
    end forall
    forall (i = 1 : 100, j = 1 : 100, .not.f(i, j))
        h(j, i, k) = temp(i, j, k)
    end forall
    !hpf$ independent
    do i = 1, 100
        !hpf$ independent
        do j = 1, 100
            !hpf$ on home(h(i, j, k)) begin
            if g(i, j) then
                ··· = h(i, j, k)
            else
                ··· = temp(i, j, k)
            end if
            !hpf$ end on
        end do
    end do
end do
```

Figure 11.6: An alternative way for the compiler to handle the example shown in Figure 11.3. This requires data optimization techniques.

```
!hpf$ distribute(block, block, block) :: h
!hpf$ independent
do k = 1, 100
    forall (i = 1 : 100, j = 1 : 100)
        where f(i, j)
            h(i, j, k) = ...
        else where
            h(j, i, k) = ...
        end where
    end forall

    forall i = 1 : 100, j = 1 : 100
        where g(i, j)
            ··· = h(i, j, k)
        else where
            ··· = h(j, i, k)
        end where
    end forall
end do
```

Figure 11.7: Yet another way the example in Figure 11.3 could have been written. In general, this sort of rewriting would not be possible, however.

```
!hpf$ distribute(block, block) :: a
!hpf$ distribute(block, cyclic) :: b
!hpf$ align s(i, j) with b(i, j)
!hpf$ independent, new(j, k)
do i = 1, 100
    j = x(i)
    k = y(i)
    a(j, k) = r(j, k) + 2
    b(j, k) = s(j, k) * r(j, k)
end do
```

Figure 11.8: An INDEPENDENT loop with indirect addressing.

1. We are in a set of tightly nested INDEPENDENT DO loops. (Perhaps we could relax this restriction at some point; we have not found the need to do so yet.)

2. At least one non-NEW array reference lvalue (i.e., location being stored to) is explicitly mapped. (If this were not true, there would be no reason to execute anything in parallel.)

3. Each non-NEW array reference rvalue (i.e., location being fetched) is available where it is used. In particular, it is either unmapped, replicated, or aligns with the left-hand side of the assignment in which it is used.

4. Each subscript of each mapped array reference is a fetch of a NEW scalar. (Assignments to non-NEW scalars are not allowed inside an INDEPENDENT DO loop in any case.)

5. All the assignments to these NEW scalars (i.e., that are involved in subscripts of mapped array references) occur in statements at the top of the loop body.

If these conditions are all met, we perform the following transformation:

- Take off the INDEPENDENT attribute of the DO loops.
- Divide the statements in the loop body into two consecutive sequences, each covered by a `dt_seq` node.
- The first `dt_seq` covers the assignments to the scalar NEW variables used as subscripts.
- The second `dt_seq` covers the rest of the original body. This `dt_seq` is tagged with the attribute `needs_privatized_guard`.
- STRIP will later compute a guard for this second `dt_seq` composed of the OR of the `cc_guards` of each statement under the `dt_seq` node. STRIP then places this `dt_seq` node under a `dt_simple_if` node guarded by this computed expression.

Of course each statement under the second `dt_seq` is still guarded by its own `cc_guard`, just as it normally would be in non-parallel code.

(We don't know at this point whether or not the guard over the `dt_seq` really helps. It does provide an early exit where appropriate. However, deleting it would not alter the semantics of the program.)

Note that the transformation we perform in this way does indeed preserve the semantics of the original program:

- Since the loops were declared INDEPENDENT, we are assured that there are no loop-carried dependences among the loop iterations.
- Normally, each iteration of an INDEPENDENT loop is executed in its entirety by one processor. This ensures that any loop-independent dependences within the iteration are honored.

  Now with this optimization, it is possible that different statements within a single iteration may execute on different processors. This cannot cause a problem, however, because the only variables that are defined and then used within the loop body are either NEW scalar variables, or are mapped array references that are used by the same processor that assigned to them.

There is a special case of this optimization that occurs when in addition to the 5 restrictions stated above,

6. *All* non-NEW array references are aligned with each other.

This additional restriction is not satisfied in the example in Figure 11.8, but it would be if the mappings of the arrays a and b were identical.

In such a case, *only* the guard for the second `dt_seq` needs to be generated—the individual guards for the statements under that `dt_seq` would all be identical to this guard, and are therefore suppressed by STRIP. In this case, the `dt_seq` is classified by DIVIDE as `single_guard`, rather than as `needs_privatized_guard`.

Chapter 12

STRIP: Local Statements

For vector statements not requiring interprocessor communication, STRIP first generates strip loops and modifies the subscripts in the array references so that the code becomes explicit SPMD; i.e., the code is parametrized by *my_processor*() and is therefore different on each processor. (This corresponds to what we call "naive strip mining" on a uni-processor; the term "strip mining" comes from an early implementation for a vector uniprocessor machine.) Then STRIP performs whatever transformations are necessary to correct the semantics of the transformed code, and to further optimize the resulting loop structure.

Vector statements involving interprocessor communication are handled differently, as described in Chapters 13–15 below. This separate handling is designed to reduce the number of interprocessor messages as much as possible.

Sections 12.8 and 12.9 describe additional tasks that STRIP performs, in addition to its handling of local statements.

12.1 Local statements

Assignment statements involving arrays are of two kinds: *local* and *remote*. A local statement is one in which no interprocessor motion takes place. Local statements can be characterized as follows:

- All nodes in the statement are in the same region, and
- Each complex cell (including complex scalar cells) in the data VP space is allocated serially; i.e. has $strip\_length = 1$. Of course, in typical cases, there will be no complex cells in the data VP space.

The second condition insures that the address of each array reference is locally computable, provided that the value of each of its subscripts is locally available. And the first condition insures that the value of each subscript is in fact locally available.

So for instance, any assignment statement where all the array references are in the same region and where the subscripts are all triples or local scalars is a local array reference. But also a statement such as

$$A(V(:),\ :) = B(:,\ :)$$

is a local statement provided A and B are are aligned, are both distributed serially in the first dimension, and that V is replicated over all the processors, so in any case the values of both array references are computable locally.

Local statements can be thought of as falling into two categories. (The compiler does not actually make this distinction—it is put here only to help with the explanation.)

- Statements in which all data VP cells are primary, local scalar, or replicated. These statements can be called *completely local*.
- Statements in which there is at least one complex data VP cell. (Of course, such a cell would have to have *strip_length* = 1.) Such a statement can be called *local with local copy*.

The assignment statement above exhibits a local copy. Here is another example:

!hpf$ template $t(n)$
  real $result(n)$, $data(n, n)$
  integer $index(n)$
!hpf$ distribute t(**block**)
!hpf$ align $result(i)$, $index(i)$, $data(i, *)$ **with** $t(i)$
  forall $(i = 1{:}n)$
    $result(i) = data(i, index(i))$

And here is a more complicated example exhibiting a local copy:

!hpf$ template $t(n, n, n)$
  real $A(n, n, n)$, $B(n, n, n, n)$
  integer $V(n, n, n, n)$, $W(n, n, n)$
!hpf$ distribute t(**block**, **block**, **block**)
!hpf$ align $A(l, m, n)$, $V(l, m, n, *)$, $W(l, m, n)$, $B(l, m, n, *)$ **with** $t(l, m, n)$
  forall $(i = 1{:}n,\ j = 1{:}n,\ k = 1{:}n)$
    $A(i, j, k) = B(i, j, k, V(i, j, k, W(i, j, k)+j)+i)$

12.2 Generating strip loops

Three tasks are necessary in order to strip mine a local statement:

- Construct a nest of DO loops around the statement, based on the statement's iteration space. The loop bounds in general differ on each processor.
- Localize the subscripts of the array references in the statement. This amounts to expressing these subscripts in terms of the loop variables for the loop nest just introduced.
- Remove any obstacles to strip mining (i.e. loop-carried dependences which have been introduced by the first two steps).

In this section we show how to construct the nest of DO loops around a local statement.

First, enumerate the primary cells in the iteration space. For each such primary cell, we construct a DO loop

do $I_j = LB_j^{loc}, UB_j^{loc}, S_j^{loc}$

In general, the loop bounds and step differ for different processors. In the remainder of this section, we show how to compute these parameters and how to adjust the corresponding subscript expressions of the array references in the body of the loop.

The iteration cell in question either comes from a FORALL index

forall $(I_j = LB_j,\ UB_j,\ S_j)$

or a triple

$\ldots A(LB_j\!:\!UB_j\!:\!S_j)\ldots$

We will refer to LB_j, UB_j, and S_j as the *iteration bounds*.

We will deal with each iteration cell separately. What we do depends on the distribution of that dimension:

SERIAL distribution. This includes all cases in which the j^{th} iteration cell has $\mathit{strip\_length} = 1$—all such cells have been made SERIAL by the time the processing gets to STRIP. (We are using j simply as a tag here; the order of the cells in the two data structures might well be different.) The strip loop must be the same on each processor. (In all other cases, the strip loop may have different bounds on different processors.) The local loop bounds are constructed immediately from the iteration bounds:

$$\begin{aligned} LB_j^{loc} &= LB_j \\ UB_j^{loc} &= UB_j \\ S_j^{loc} &= S_j \end{aligned}$$

If on the other hand the j^{th} iteration cell has $\mathit{strip\_length} > 1$, then since the statement is local, it must correspond to exactly one data cell of the array reference on the left-hand side of the assignment statement. For simplicity, let us refer to this also as the j^{th} data cell (although the numbering might really be different; the two numbers would be related through the `correspond` field (page 51). f_j will denote the corresponding data allocation function, and ϕ_j will denote the iteration allocation function. The partial subscript map between the iteration cell and the data cell will be denoted by σ_j. σ_j is an affine map, and we have

$$\phi_j = f_j \circ \sigma_j.$$

DLB_j will denote the declared lower bound of the array reference on the left-hand side in the dimension corresponding to the j^{th} data cell, and DUB_j will denote the corresponding declared upper bound.

BLOCK distribution. The values of the j^{th} iteration coordinate are

$$\left\{ S_j * i + LB_j : 0 \le i \le \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor \right\}.$$

To find the addresses in the j^{th} cell (i.e. the partial values of $\bar{v}$) corresponding to these subscripts, we reason as follows: The j^{th} component of the virtual processor address (which we call v_j) is just $f_j(\sigma_j(S_j * i + LB_j))$, or, equivalently, $\phi_j(S_j * i + LB_j)$. The partial value $\bar{v}_j$ is then given by

$$\left\lfloor \frac{\phi_j(S_j * i + LB_j)}{\mathit{num\_folds}_j} \right\rfloor.$$

Now each processor knows its own value of $\bar{v}_j$, as outlined above on page 50. This value is computed once during the DATA phase and stored as the `v_bar` attribute of each cell. The first and last elements of the array in a particular processor's memory correspond to the least and greatest value of i in the range

$$0 \leq i \leq \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor$$

satisfying

$$\left\lfloor \frac{\phi_j(S_j * i + LB_j)}{\mathit{num\_folds}_j} \right\rfloor = \bar{v}_j$$

Solving for i, we see that the least and greatest values of i are computed as follows:
If $\phi_j \uparrow$ and $S_j > 0$, or $\phi_j \downarrow$ and $S_j < 0$ then

$$i^{loc}_{low} = \max\left\{0, \left\lceil \frac{\phi_j^{-1}(\bar{v}_j * \mathit{num\_folds}_j) - LB_j}{S_j} \right\rceil\right\}$$

and

$$i^{loc}_{high} = \min\left\{\left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor, \left\lceil \frac{\phi_j^{-1}((\bar{v}_j + 1) * \mathit{num\_folds}_j) - LB_j}{S_j} \right\rceil - 1\right\}$$

On the other hand, if $\phi_j \uparrow$ and $S_j < 0$, or $\phi_j \downarrow$ and $S_j > 0$ then

$$i^{loc}_{low} = \max\left\{0, \left\lfloor \frac{\phi_j^{-1}((\bar{v}_j + 1) * \mathit{num\_folds}_j) - LB_j}{S_j} \right\rfloor + 1\right\}$$

and

$$i^{loc}_{high} = \min\left\{\left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor, \left\lfloor \frac{\phi_j^{-1}(\bar{v}_j * \mathit{num\_folds}_j) - LB_j}{S_j} \right\rfloor\right\}$$

If $i^{loc}_{low} \leq i^{loc}_{high}$, there are elements of A in the local processor's memory. In this case, the local (0-based) j^{th} component of the address of the element of A corresponding to i is $\hat{v}_j = \phi_j(S_j * i + LB_j) \bmod \mathit{num\_folds}_j$. Hence the local lower and upper loop bounds and the local loop stride are given by

$$\begin{aligned}
LB^{loc}_j &= S_j * i^{loc}_{low} + LB_j \\
UB^{loc}_j &= S_j * i^{loc}_{high} + LB_j \\
S^{loc}_j &= S_j
\end{aligned}$$

An example showing a BLOCK distribution is presented on page 133.

If the effective stride and $alloc\_stride_j$ are both known at compile time to be 1 (and hence also the source stride $S_j = 1$), these calculations can be greatly simplified[1]:

The possible values of v_j in the current block run from $\bar{v}_j * n_j$ to $\bar{v}_j * n_j + n_j - 1$. Thus, we have

$$
\begin{aligned}
v_{j,low}^{loc} &= \max\left\{\phi_j(LB_j), \bar{v}_j * num\_folds_j\right\} \\
v_{j,high}^{loc} &= \min\left\{\phi_j(UB_j), (\bar{v}_j + 1) * num\_folds_j - 1\right\}
\end{aligned}
$$

Now we apply ϕ_j^{-1} to each term. We can do this because since $\phi_j \uparrow$, applying ϕ_j^{-1} preserves the inequalities. (In fact, applying ϕ_j^{-1} in this special case just amounts to subtracting $alloc\_offset_j$.) This gives us the following formulas:

$$
\begin{aligned}
LB_j^{loc} &= \max\left\{LB_j, \phi_j^{-1}(\bar{v}_j * num\_folds_j)\right\} \\
UB_j^{loc} &= \min\left\{UB_j, \phi_j^{-1}\big((\bar{v}_j + 1) * num\_folds_j - 1\big)\right\}
\end{aligned}
$$

Note in particular that if $\bar{v}_j$ is not in the interval

$$\phi_j(LB_j) \operatorname{div} num\_folds_j \le \bar{v}_j \le \phi_j(UB_j) \operatorname{div} num\_folds_j$$

then these formulas yield a local upper bound that is less than the local lower bound—i.e., the local loop is empty, as it should be.

CYCLIC distribution. The idea behind the calculations is similar, but a closed-form solution does not seem possible. The calculations to find the local bounds and stride become a short algorithm, illustrated in Figure 12.1.

The values of v_j corresponding to these local bounds are then given by the triple $v_{j,low} : v_{j,high} : v_{j,str}$, where

$$
\begin{aligned}
v_{j,low} &= \phi_j(LB_j^{loc}) \\
v_{j,high} &= \phi_j(UB_j^{loc}) \\
v_{j,str} &= \phi_j(LB_j^{loc} + S_j^{loc}) - v_{j,low}
\end{aligned}
$$

[1]A significant part of this simplification is due to Bill Noyce.

function compute_cyclic_bounds

begin

-- First initialize the return values to create an empty loop. If either (and hence both) of the **for** loops in this function cannot find an element, this is what will be returned.

$LB_j^{loc} \leftarrow \mathit{strip\_length}_j$;
$UB_j^{loc} \leftarrow 0$;
$S_j^{loc} \leftarrow \mathit{strip\_length}_j$;

-- Search forward to find the local lower bound and local stride:

```
element ← 0;
for i = 0 to ⌊(UB_j − LB_j)/S_j⌋ do
    if φ_j(S_j * i + LB_j) mod strip_length_j = v̄_j then begin
        element ← element + 1;
        if element = 1 then begin
            LB_j^loc ← S_j * i + LB_j;
            continue;
        end if
        if element = 2 then begin
            UB_j^loc ← S_j * i + LB_j;
            S_j^loc ← UB_j^loc − LB_j^loc;
            break;
        end if
    end if
end for
```

-- Search backward to find the local upper bound:

```
for i = ⌊(UB_j − LB_j)/S_j⌋ down to 0 do
    if φ_j(S_j * i + LB_j) mod strip_length_j = v̄_j then begin
        UB_j^loc ← S_j * i + LB_j;
        break;
    end if
end for
```

end

Figure 12.1: Definition of the library function compute_cyclic_bounds(). By having two loops, one searching forward and the other backward, we avoid the problem of visiting every element of a very large iteration space and performing an expensive computation (including a mod) at each step.

As in the case of BLOCK distribution, if we know at compile time that the effective stride is 1, these calculations can be simplified, and an actual expression (rather than a function call or inlined code for the algorithm) can be generated, as shown in Figure 12.2.

If $LB_j^{loc} > UB_j^{loc}$, then the local processor does not address any elements of the arrays and so the DO loop is empty.

$S_j^{loc} \leftarrow strip\_length_j;$
if $\phi_j(LB_j) \bmod strip\_length_j > \bar{v}_j$ **then**
  $LB_j^{loc} \leftarrow LB_j + (\bar{v}_j - \phi_j(LB_j) \bmod strip\_length_j) + strip\_length_j;$
else
  $LB_j^{loc} \leftarrow LB_j + (\bar{v}_j - \phi_j(LB_j) \bmod strip\_length_j);$
end if
if $\phi_j(UB_j) \bmod strip\_length_j \geq \bar{v}_j$ **then**
  $UB_j^{loc} \leftarrow UB_j + (\bar{v}_j - \phi_j(UB_j) \bmod strip\_length_j);$
else
  $UB_j^{loc} \leftarrow UB_j + (\bar{v}_j - \phi_j(UB_j) \bmod strip\_length_j) - strip\_length_j;$
end if

Figure 12.2: Computation of local loop bounds and stride for CYCLIC distribution when the effective stride is known to be 1.

The corresponding values of v_j are

$$v_{j,low} : v_{j,high} : strip\_length_j$$

where

$$v_{j,low} = \phi_j(LB_j^{loc})$$
$$v_{j,high} = \phi_j(UB_j^{loc})$$

CYCLIC(n) distribution Before generating strip loops corresponding to a primary cell which has been distributed CYCLIC(n), STRIP first creates a loop around the statement, each iteration of which corresponds to one "block" of the array. The loop is of the form

do $K_j = K_j^{init}, K_j^{trm}, K_j^{incr}$
...
end do

where

$$K_j^{init} = \left\lfloor \frac{\phi_j(LB_j)}{n_j * strip\_length_j} \right\rfloor$$

$$K_j^{trm} = \left\lfloor \frac{\phi_j(UB_j)}{n_j * strip\_length_j} \right\rfloor$$

$$K_j^{incr} = \begin{cases} 1 & \text{if} \quad K_j^{init} \leq K_j^{trm} \\ -1 & \text{otherwise} \end{cases}$$

We further set

$$K_j^{low} = \begin{cases} K_j^{init} & \text{if } \phi_j \uparrow \text{ and } S_j > 0, \text{ or } \phi_j \downarrow \text{ and } S_j < 0 \\ K_j^{trm} & \text{if } \phi_j \uparrow \text{ and } S_j < 0, \text{ or } \phi_j \downarrow \text{ and } S_j > 0 \end{cases}$$

$$K_j^{high} = \begin{cases} K_j^{trm} & \text{if } \phi_j \uparrow \text{ and } S_j > 0, \text{ or } \phi_j \downarrow \text{ and } S_j < 0 \\ K_j^{init} & \text{if } \phi_j \uparrow \text{ and } S_j < 0, \text{ or } \phi_j \downarrow \text{ and } S_j > 0 \end{cases}$$

Note that we always have

$$0 \leq K_j^{low} \leq K_j^{high}.$$

Now we continue in a similar manner to the discussion of BLOCK distribution. The j^{th} iteration coordinate takes on the values $\{S_j * i + LB_j\}$, where

$$K_j * n_j * \mathit{strip\_length}_j \leq \phi_j(S_j * i + LB_j) < (K_j + 1) * n_j * \mathit{strip\_length}_j.$$

The addresses in the j^{th} cell (i.e., the partial values of $\bar{v}$) corresponding to these subscripts are the values

$$\left\{ \left\lfloor \frac{\phi_j(S_j * i + LB_j)}{n_j} \right\rfloor - K_j * \mathit{strip\_length}_j \right\}$$

where again

$$K_j * n_j * \mathit{strip\_length}_j \leq \phi_j(S_j * i + LB_j) < (K_j + 1) * n_j * \mathit{strip\_length}_j.$$

Note that this last inequality could just as well be written as

$$0 \leq \left\lfloor \frac{\phi_j(S_j * i + LB_j)}{n_j} \right\rfloor - K_j * \mathit{strip\_length}_j < \mathit{strip\_length}_j$$

So we see that the first and last elements of the array in a particular processor's memory correspond to the least and greatest value of i such that

$$\left\lfloor \frac{\phi_j(S_j * i + LB_j)}{n_j} \right\rfloor = \bar{v}_j + K_j * \mathit{strip\_length}$$

Setting $K_j * \mathit{strip\_length}_j = W_j$ and solving for i, we see that the least and greatest values of i are computed as follows:

If $\phi_j \uparrow$ and $S_j > 0$, or $\phi_j \downarrow$ and $S_j < 0$ then

$$i_{low}^{loc} = \max\left\{ 0, \left\lceil \frac{\phi_j^{-1}((\bar{v}_j + W_j) * n_j) - LB_j}{S_j} \right\rceil \right\}$$

and

$$i_{high}^{loc} = \min\left\{ \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor, \left\lceil \frac{\phi_j^{-1}((\bar{v}_j + W_j + 1) * n_j) - LB_j}{S_j} \right\rceil - 1 \right\}$$

If on the other hand $\phi_j \uparrow$ and $S_j < 0$, or $\phi_j \downarrow$ and $S_j > 0$ then

$$i_{low}^{loc} = \max\left\{ 0, \left\lfloor \frac{\phi_j^{-1}((\bar{v}_j + W_j + 1) * n_j) - LB_j}{S_j} \right\rfloor + 1 \right\}$$

and

$$i_{high}^{loc} = \min \left\{ \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor, \left\lfloor \frac{\phi_j^{-1}(\bar{v}_j + W_j) * n_j) - LB_j}{S_j} \right\rfloor \right\}$$

The local lower and upper loop bounds and the local loop stride are then given (just as in the case of BLOCK distribution) by

$$\begin{aligned} LB_j^{loc} &= S_j * i_{low}^{loc} + LB_j \\ UB_j^{loc} &= S_j * i_{high}^{loc} + LB_j \\ S_j^{loc} &= S_j \end{aligned}$$

An example showing a CYCLIC(n) distribution is presented on page 134.

Once again, if the effective stride and $alloc\_stride_j$ are both known at compile time to be 1 (and hence also the source stride $S_j = 1$), simpler formulas can be generated in the following manner:

The current block number is given by

$$my\_block = K_j * strip\_length_j + \bar{v}_j$$

The possible values of v_j in this block run from

$$my\_block * n_j = K_j * strip\_length_j * n_j + \bar{v}_j * n_j$$

to

$$my\_block * n_j + n_j - 1 = K_j * strip\_length_j * n_j + (\bar{v} + 1) * n_j - 1$$

We thus have

$$\begin{aligned} v_{j,low}^{loc} &= \max \left\{ \phi_j(LB_j), K_j * strip\_length_j * n_j + \bar{v}_j * n_j \right\} \\ v_{j,high}^{loc} &= \min \left\{ \phi_j(UB_j), K_j * strip\_length_j * n_j + (\bar{v} + 1) * n_j - 1 \right\} \end{aligned}$$

and so, just as in the similar computation for BLOCK distributions, we have

$$\begin{aligned} LB_j^{loc} &= \max \left\{ LB_j, \phi_j^{-1}(K_j * strip\_length_j * n_j + \bar{v}_j * n_j) \right\} \\ UB_j^{loc} &= \min \left\{ UB_j, \phi_j^{-1}\big(K_j * strip\_length_j * n_j + (\bar{v} + 1) * n_j - 1\big) \right\} \end{aligned}$$

12.3 Handling the completion structure

The iteration space of a local statement may have a non-trivial completion structure. Such a structure corresponds to a restriction of the set of processors performing the local statement. This restriction can be implemented in the form of a boolean expression computed locally on each processor. This boolean expression can then be used to construct a test outside of the outermost generated strip loop. However, we do not want to expose this test until the end the strip mine phase, because it will inhibit loop fusion. Therefore, we compute the boolean expression and store it as an attribute of the outermost strip loop. This attribute is preserved by the transformations the strip miner makes (including loop interchange). Then a final pass over the dotree at the end of the strip miner inserts the IF tests.

The boolean expressions constructed from completion subspaces are constructed by ANDing together boolean expressions formed for each such subspace. These elementary boolean expressions are each of the form

$$\text{IF } \bar{v}_j = \left\lfloor \frac{\mathit{my\_processor}\texttt{()}}{\mathit{multiplier}_j} \right\rfloor \bmod \mathit{strip\_length}_j$$

where $\bar{v}_j$ is derived from the completion subspace `value` s_j, and is calculated as follows:

BLOCK distribution.

$$\bar{v}_j = \left\lfloor \frac{\phi_j(s_j)}{\mathit{num\_folds}_j} \right\rfloor .$$

CYCLIC distribution.

$$\bar{v}_j = \phi_j(s_j) \bmod \mathit{strip\_length}_j .$$

CYCLIC(n) distribution.

$$\bar{v}_j = \left\lfloor \frac{\phi_j(s_j)}{n_j} \right\rfloor \bmod \mathit{strip\_length}_j .$$

Such a boolean expression is known in the implementation as a `cc_guard` (“completion cell guard”)

12.4 Localizing subscripts

After building a loop nest for each local statement (and in the process assigning a loop variable to each iteration subspace), the array references in the statement are processed. Two things happen:

1. Each vINDEX node (which may come from a FORALL index or from a triple) is replaced by its corresponding loop index. It then is normalized (see the discussion on page 108) so that in effect all the iteration spaces in the statement have the same bounds. This is done simply by using the loop bounds as the normalized bounds. Thus, LB, and S come from the loop. If we denote the corresponding values from the vINDEX node by LB_{old} and S_{old}, then the loop variable I is replaced by the expression

$$\frac{I - LB}{S} * S_{\text{old}} + LB_{\text{old}}$$

2. The subscript is then localized. That is, in our notation, the subscript s_j is replaced by $\hat{v}_j$.

The first of these tasks is straightforward. Here we show how the second one is performed. In general, $\hat{v}_j$ is computed as follows (note that here we are walking through the subscripts and not the iteration space): First of all, v_j is simply computed as $f_j(s_j)$. Then, depending on the distribution of the data subspace, we have

SERIAL distribution

$$\hat{v}_j = f_j(s_j)$$

BLOCK distribution.

$$\hat{v}_j = f_j(s_j) \bmod \mathit{num\_folds}_j.$$

CYCLIC distribution.

$$\hat{v}_j = \left\lfloor \frac{f_j(s_j)}{\mathit{strip\_length}_j} \right\rfloor$$

CYCLIC(n) distribution

$$\hat{v}_j = n_j * \left\lfloor \frac{f_j(s_j)}{n_j * \mathit{strip\_length}_j} \right\rfloor + f_j(s_j) \bmod n_j$$

The subtracted term in the SERIAL, CYCLIC, and CYCLIC(n) cases takes account of the optimization in the computation of $\mathit{num\_folds}_j$ on page 74—we only allocate storage starting at the lower bound for these cases.

In the BLOCK case, the expression $f_j(s_j) \bmod \mathit{num\_folds}_j$ involves a mod which has to be evaluated each time the array reference is encountered in any loop iteration. This expression can be rewritten as

$$f_j(s_j) - \left\lfloor \frac{f_j(s_j)}{\mathit{num\_folds}_j} \right\rfloor * \mathit{num\_folds}_j$$

which is not in itself an improvement. However, when the the subscript is a primary subscript, this is in turn equivalent to the expression

$$f_j(s_j) - \left\lfloor \frac{\phi_j(LB_j^{loc})}{\mathit{num\_folds}_j} \right\rfloor * \mathit{num\_folds}_j$$

and in this formulation, the term containing the division operator is a loop invariant, and so can be moved out of the loop.

Some care is necessary here to realize that this really works: the expression $\phi_j(LB_j^{loc})$ is computed in terms of the iteration space of the loop, which is not the same as the iteration space of the array reference. However, for BLOCK distributions, corresponding elements live on the same virtual processor, so this has to yield the same result as if both ϕ_j and LB_j^{loc} were computed from the array reference itself. This reasoning does not work, however for CYCLIC(n) dimensions, as we see next:

For a CYCLIC(n) primary subscript, we would like in a similar way to replace the formula for $\hat{v}_j$ above by

$$n_j * K_j + f_j(s_j) - \left\lfloor \frac{\phi_j(LB_j^{loc})}{n_j} \right\rfloor * n_j$$

This would be correct if ϕ_j, LB_j^{loc} and the bounds for K_j were derived from the iteration space of the array reference whose subscripts are being localized. However, in general, they are not—they are derived from some particular iteration space (which in a local statement is typically that of the array reference on the left-hand side).

This did not cause a problem when handling BLOCK distributions, but CYCLIC(n) distributions are slightly more complex: Immediately below, we will let $\overline{\phi}$ denote the iteration allocation function of the iteration space of the array reference being localized, as opposed to ϕ, which denotes the iteration allocation function fo the iteration space used to construct the loop nest. Similarly, LB_j^{loc} denotes the localized lower bound of that iteration space. It may then be the case that $\phi_j(LB_j^{loc})$ can differ from $\overline{\phi}(\overline{LB_j^{loc}})$ by a multiple of $v_j * \mathit{strip\_length}_j$. That is, the corresponding values of $\bar{v}_j$ are equal, and the offsets within the individual blocks are equal, but the block numbers may differ. We therefore have to adjust the first and third terms as follows:

- Let us use the following notation:
  - $\overline{\phi}$ denotes the iteration allocation function of the iteration space of the array reference being localized (as opposed to ϕ, which denotes the iteration allocation function of the iteration space used to construct the loop nest).
  - $\overline{LB}_j$ denotes the lower bound of that iteration space.
  - $\overline{LB_j^{loc}}$ denotes the localized lower bound—i.e., the local lower bound that we would compute if we were using this iteration space to generate the local loop bounds.

  The third term can then be adjusted by adding to it the expression

  $$\left\lfloor \frac{\phi_j(LB_j^{loc})}{n_j} \right\rfloor * n_j - \left\lfloor \frac{\overline{\phi}_j(\overline{LB_j^{loc}})}{n_j} \right\rfloor * n_j$$

  Now this is much too complicated to compute. However, we do know that in any case, we must have

  $$\phi_j(LB_j^{loc}) \equiv \overline{\phi}_j(\overline{LB_j^{loc}}) \pmod{n_j * \mathit{strip\_length}_j}$$

  Therefore the above expression simplifies to

  $$\phi_j(LB_j^{loc}) - \overline{\phi}_j(\overline{LB_j^{loc}})$$

  Further, we also know that

  $$\phi_j(LB_j) - \phi_j(LB_j^{loc}) = \overline{\phi}_j(\overline{LB}_j) - \overline{\phi}_j(\overline{LB_j^{loc}})$$

  so the expression simplifies further to

  $$\phi_j(LB_j) - \overline{\phi}_j(\overline{LB}_j)$$

- Similarly, the first term can be adjusted by subtracting from it the expression

  $$\frac{\phi_j(LB_j) - \overline{\phi}_j(\overline{LB}_j)}{\mathit{strip\_length}_j}$$

Thus, for BLOCK and CYCLIC(n) primary subscripts, the formulas for $\hat{v}_j$ above are optimized to

BLOCK distribution.

$$\hat{v}_j = f_j(s_j) - \left\lfloor \frac{\phi_j(LB_j^{loc})}{\mathit{num\_folds}_j} \right\rfloor * \mathit{num\_folds}_j.$$

CYCLIC(n) distribution.

$$\hat{v}_j = n_j * K_j + f_j(s_j) - \left\lfloor \frac{\phi_j(LB_j^{loc})}{n_j} \right\rfloor * n_j - \frac{\phi_j(LB_j) - \overline{\phi}_j(\overline{LB}_j)}{strip\_length_j} + \phi_j(LB_j) - \overline{\phi}_j(\overline{LB}_j)$$

For an example involving a BLOCK distribution, suppose we have the following situation:

> **!hpf\$ template** $t(50)$
> **!hpf\$ distribute** t(**block**)
> **real** $A(46)$
> **!hpf\$ align** $A(i)$ **with** $t(i{+}3)$
> $A(1\!:\!46\!:\!3) = \ldots$

and suppose we know that $strip\_length = 5$. Then num_folds is forced to be 10, and $alloc\_stride = 1$, and $alloc\_offset = 2$. In this example, $\phi_j = f_j$, and the array is distributed as follows across 5 processors (where each of the 5 columns in this picture represents the linear memory associated with one of the 5 processors):

			$A(28)$	
		$A(19)$		
	$A(10)$			$A(40)$
$A(1)$			$A(31)$	
		$A(22)$		
	$A(13)$			$A(43)$
$A(4)$			$A(34)$	
		$A(25)$		
	$A(16)$			$A(46)$
$A(7)$			$A(37)$	

.

The computations in processor 2 (numbering the processors from 0) then generate the DO loop

> **do** $I = 19,\ 25,\ 3$
> $A(I - 18) = \ldots$

which assigns to the array section $A(1 : 7 : 3)$, and in processor 3 they generate the DO loop

> **do** $I = 28,\ 37,\ 3$
> $A(I - 28) = \ldots$

which assigns to the array section $A(0:9:3)$. Note that the subscripts now refer to local addresses within each processor's memory.

For an example involving a CYCLIC(n) distribution, we can consider the same local statement

$$A(1:46:3) = \ldots$$

as before, where again $strip\_length = 5$, $alloc\_stride = 1$, and $alloc\_offset = 2$, but this time allocated CYCLIC(5). Here K $(= K_1)$ runs from 0 to 1, and the corresponding picture looks like this:

			$A(13)$	
	$A(4)$			$A(19)$
		$A(10)$		
$A(1)$			$A(16)$	
	$A(7)$			$A(22)$
	$A(28)$			$A(43)$
		$A(34)$		
$A(25)$			$A(40)$	
	$A(31)$			$A(46)$
		$A(37)$		

.

12.5 "Simplifying subscripts"

In many cases, we can perform a transformation on the generated loop bounds and corresponding subscripts that has the effect of making the subscripts a lot simpler (at the cost of making the loop bounds more complicated). This can make it easier for the scalar optimizer in GEM to do its work, and sometimes results in more efficient code.

The transformation leads to a simplification in the cases when the subscripts are primary and the dimension is non-serial (i.e., the data subspace has $strip\_length > 1$). We consider these cases first:

12.5.1 Primary subscript with $strip\_length > 1$

The key fact that we lean on in these cases is that since the statement is local, if two array references (L and R in the code fragments below) contain such subscripts in corresponding dimensions, then corresponding elements of L and R in those dimensions have the same virtual processor coordinate.

(In the case when L and R are 1-dimensional, this is easier to state: in such a case, corresponding elements of L and R live on the same virtual processor.)

BLOCK distribution

Suppose we have a statement equivalent to

!hpf$ distribute(**block**) :: L,R
  forall $(i = LB{:}UB{:}S)$
    $L(a * i + b) = \ldots R(c * i + d) \ldots$

Our normal STRIP processing, as explained above, would transform this into a statement equivalent to

!hpf$ distribute(**block**) :: L,R
  do $i = LB^{loc},\ UB^{loc},\ S^{loc}$
    $L(\lambda_L(a * i + b)) = \ldots R(\lambda_R(c * i + d)) \ldots$

where λ_L is the composition of the data allocation function and the data distribution function for (the first dimension of) the array L, and similarly for λ_R. That is, the input to λ_L is the subscript, and the output is $\hat{v}$. Section 12.4 showed how λ_L is computed.

The transformation we make in this situation replaces the code above by

!hpf$ distribute(**block**) :: L,R
  do $i = \lambda_L(a * LB^{loc} + b),\ \lambda_L(a * UB^{loc} + b),\ das_L * a * S^{loc}$
    $L(i) = \ldots R(i) \ldots$

where das_L is the data allocation stride for (the first dimension of) L.

The reason this works is that, as explained above, since the statement is local, corresponding elements of L and R live on the same virtual processor. In particular,

$$\lambda_L(a * LB^{loc} + b) = \lambda_R(c * LB^{loc} + d)$$
$$das_L * a = das_R * c$$

The computation of λ_R for a BLOCK distributed subscript has to be slightly modified when we are simplifying subscripts: the value of LB_j^{loc} used in the computation refers to the *original* value (i.e., the value of the localized loop bound if we had not performed the transformation on the loop bounds indicated above). So this value has to be computed and saved when the loop is created, for use later in transforming the loop bounds.

CYCLIC distribution

The idea is the same. The only differences are the following:

- The new loop increment has to be
$$\frac{das_L * a * S^{loc}}{strip\_length_L}$$

- When localizing the subscripts, we have to allow for the fact that corresponding elements of arrays L and R, while they must lie on the same physical processor, may not lie on the same virtual processor. We account for this by actually generating

  !hpf$ distribute(cyclic) :: L,R

  $$\textbf{do } i = \lambda_L(a * LB^{loc} + b),\ \lambda_L(a * UB^{loc} + b),\ \frac{das_L * a * S^{loc}}{strip\_length_L}$$
  $$L(i + d_L) = \ldots R(i + d_R) \ldots$$

  where for any array A, d_A represents the following expression:

  $$d_A = \left\lfloor \frac{\phi_A(LB_A) - \phi_L(LB_L)}{strip\_length} \right\rfloor$$

  In particular, this makes $d_L = 0$. In the typical case, d_R will be 0 also. Actually, we want to compute d'_A, defined in terms of LB_B^{loc} or UB_B^{loc}. But this is in general much more difficult, and, as on page 132, we know that we must have

  $$\phi_A(LB_A^{loc}) - \phi_L(LB_L^{loc}) = \phi_A(LB_A) - \phi_L(LB_L)$$

  so $d'_A = d_a$.

CYCLIC(n) distribution

In the case of a CYCLIC(n) distribution, the following modifications are made:

- The transformation of loop bounds is carried out only on the inner of the two loops that are created. The outer loop is unchanged.
- The new loop increment (in the inner loop) remains $das_A * a * S^{loc}$, as in the case of a BLOCK distribution. (This is because the inner loop is really a BLOCK loop.)
- As in the case of a BLOCK distribution, the computation of λ_R uses the value of LB_j^{loc} that would have been used if the loop bounds had not been transformed. So in this case also, this "original" value is computed when the loop is created, and saved for the subsequent application of the function λ_R.
- As in the case of a CYCLIC distribution, we have to allow for the fact corresponding elements may lie on different virtual processors. So we generate the following code (we don't bother to show the outer loop here, since it is the same whether or not we are simplifying subscripts):

  !hpf$ distribute(cyclic(n)) :: L,R

  $$\textbf{do } i = \lambda_L(a * LB^{loc} + b),\ \lambda_L(a * UB^{loc} + b),\ das_L * a * S^{loc}$$
  $$L(i + d_L) = \ldots R(i + d_R) \ldots$$

  where for any array A, d_A represents the following expression:

  $$d_A = n_A * \left\lfloor \frac{\phi_A(LB_A) - \phi_L(LB_L)}{n_A * strip\_length_A} \right\rfloor$$

  where n_A is the declared block size of A in the dimension being considered. (We know that $n_A = n_L$ and $strip\_length_A = strip\_length_L$.) As in the case of CYCLIC data above, this computation is really a substitute for the same expression written in terms of LB_A^{loc} and LB_L^{loc}, which we know must have the same value.

12.5.2 Non-primary subscript, or *strip_length* = 1

When the *strip_length* is 1 in a primary subscript in the array L, we still perform the transformation on the loop bounds: the loop becomes

> **!hpf$ distribute**(**block**) :: L,R
>
> **do** $i = \lambda_L(a * LB^{loc} + b),\ \lambda_L(a * UB^{loc} + b),\ das_L * a * S^{loc}$

However, we cannot transform the subscripts as simply as we did above, because corresponding elements are not guaranteed to live on the same virtual processor. The same problem occurs in a subscript that does not come from a primary subspace. (For instance, we may have a reference $A(i, V(i))$. The second subscript is not primary, but contains a reference to i, whose local iteration bounds have been transformed as above. We actually inhibit the simplifying subscripts transformation in this case; see below.)

In these cases, we have to bite the bullet and create a subscript expression that is the equivalent of the expression that would be generated without the simplifying subscripts transformation. We do this in two steps:

- We adjust each loop variable so that it runs over the non-simplified loop range. We do this by replacing each reference to a loop variable I by the expression

  $$\frac{I - LB^{loc}_{new}}{S^{loc}_{new}} * S^{loc}_{orig} + LB^{loc}_{orig}$$

  where the "orig" values are just the values LB^{loc} and S^{loc} computed without the simplifying subscripts changes indicated above, and the "new" values are computed with with these changes. That is,

  $$LB^{loc}_{new} = \lambda_L(a * LB^{loc} + b)$$
  $$S^{loc}_{new} = das_L * a * S^{loc}$$

  Since serial subspaces are quite common, and since this transformation has the potential for introducing division operators in addressing expressions, we currently inhibit the simplifying subscripts transformation for a serial iteration subspace unless the corresponding subscript (on the left-hand side array L) has

  - $das_L = 1$
  - $a = 1$

  This is, of course, the typical case, so this is not really much of a restriction. An example of code, however, that would *not* be transformed with this restriction is the following:

  > **!hpf$ template** t(100)
  > **!hpf$ distribute**(*****) :: t
  > **!hpf$ align** $L(i)$ **with** $t(2 * i)$
  > **!hpf$ align** $R(i)$ **with** $t(i)$
  > $L = R$

  since it would lead to generated code that looked like this::

> **do** $i = 2,\ 100,\ 2$
> $\quad L(i) = R(i/2)$

- We then localize subscripts as usual, as explained in Section 12.4 (page 130). This also takes care of local scalar and local complex subscripts (i.e., the subscripts that do not correspond to any loop index).

- As a further simplification, we also inhibit the simplifying subscripts transformation completely for any statement that contains a complex subscript. (Our code would work correctly, but the addressing expressions might again become somewhat more complex, and we want to avoid this.)

12.6 Inserting IF guards for WHERE constructions

We have shown how a strip loop is generated for each primary data cell, and how the subscript in that cell is localized. If the resulting local subscript is non-existent, the generated strip loop will never be executed—the zero-trip test will branch around the loop. Thus, "context" information is automatically inserted into the dotree by the strip miner in this case.

Scalar subscripts, on the other hand, contribute to the completion cell of the iteration VP space, and are handled by the mechanism described above in section 12.3.

If the statement being strip mined is guarded by a non-trivial WHERE mask, an IF guard has to be generated for the stripped mask expression. This IF guard is constructed directly over the stripped statement, inside the generated do loop nest.

12.7 Removing strip mining obstacles

At this point, STRIP has generated strip loops for all local statements, and has localized the subscripts of all array references in those statements.

These generated loops run locally on each processor and the loop bounds are functions of the processor. However, the introduction of the strip mining loops may have introduced artificial (and incorrect) loop-carried dependences. For instance, if an array assignment

$$A(2\!:\!1000) = A(1\!:\!999)$$

(where the dimension is laid out serially) is strip mined to become

> **do** $I = 1,\ 999$
> $\quad A(I{+}1) = A(I)$
> **end do**

then a loop-carried dependence has been introduced (e.g. $A(2)$ is assigned to on the first iteration of the loop, and this new value is subsequently used on the second iteration of the loop). This dependence is an artifact of naive strip mining, and it is incorrect—it violates the semantics of the original statement, which is that the value of $A(2)$ which is used should be the original value. Such introduced dependences—loop-carried dependences in strip loops—are referred to as strip mining *obstacles*.

The dependence analyzer is used to identify these strip mining obstacles, and loop reversal and loop interchange are used where possible to either eliminate them or to minimize their effects.

In the worst case, a temporary must be introduced to eliminate a strip mining obstacle. Such a temporary is always allocated on the stack—i.e., a local temporary is allocated on the stack of each processor. The size of this local temporary must be large enough to accomodate the maximum size local temporary that each processor needs. This maximum size is computed for each dimension corresponding to a loop which is being copied (i.e. for all loops inside the loop carrying the strip mining obstacle). The contribution of the size for a loop corresponding to a particular dimension can be computed as follows:

BLOCK distribution.

$$\left\lceil \frac{num\_folds_j}{|S_j|} \right\rceil$$

CYCLIC distribution.

$$\left| \left\lfloor \frac{\phi_j(UB_j)}{strip\_length_j} \right\rfloor - \left\lfloor \frac{\phi_j(LB_j)}{strip\_length_j} \right\rfloor \right| + 1$$

CYCLIC(n) distribution. If $K_j^{high} = K_j^{low}$ (so the entire array section fits in the first block), we use

$$\min \left\{ |\phi_j(UB_j) - \phi_j(LB_j)| + 1, \left\lceil \frac{n_j}{|S_j|} \right\rceil * (K_j^{high} - K_j^{low} + 1) \right\}$$

provided this expression is a compile-time constant; otherwise we simply use

$$\left\lceil \frac{n_j}{|S_j|} \right\rceil * (K_j^{high} - K_j^{low} + 1)$$

The reason for the first expression is that it allows for the possibility that the array section might actually be a small part of the block size—in this case, we only allocate a temp large enough to hold the array, rather than allocating a full block size of memory.

SERIAL ($*$) distribution.

$$\frac{UB_j - LB_j}{S_j} + 1$$

After the removal of strip mining obstacles, further transformations may be used to improve the quality of generated code. Such transformations include loop fusion and exterior loop optimization.

The techniques referred to in this section are described in more detail in [3].

12.8 Actual arguments to function calls

All function calls appear in the dotree as SCALAR statements (even when the function returns an array; the call is buried under an AFUNC operator which is classified as SCALAR)

STRIP localizes the subscripts of all array references in SCALAR statements as follows: First, all vINDEX nodes are replaced by their contained TRIPLE nodes. vINDEX nodes are no longer necessary, because they were only needed for ITER to create iteration space information. We don't want vINDEX nodes to appear after Transform. Then if the array reference is part of an actual argument, the following processing takes place, depending on the kind of function:

Arguments to I/O Such array references must live on processor 0. STRIP localizes each scalar subscript, and localizes the bounds of each TRIPLE. Such localization amounts to no more than introducing a possible affine function, since these are serial dimensions.

Arguments to non-I/O Global Functions STRIP replaces each TRIPLE by [0:`num_folds` $-$ 1:1]. (Note that this makes each array be 0-based; the convention is that after Transform, all arrays are indexed as if they were 0-based, and the DTR2CIR phase at the end of Transform resets the bounds of each dimension in the array's entry in the symbol table to be $[0 \mathinner{\ldotp\ldotp} \text{extent} - 1]$.)

It may actually be the case that fewer than `num_folds` $- 1$ array elements are passed on each processor. Nevertheless, there is no harm in transforming the subscripts in this way. The only thing the subscripts are used for is by a later phase of the compiler in generating a temporary. The actual addressing is determined by the HPF directives and/or dope vector information, and this is always guaranteed to represent the actual array section which is passed.

Arguments to HPF_LOCAL Functions STRIP localizes each scalar subscript, and replaces each TRIPLE by its localized version (that is, by the TRIPLE representing what the local loop bounds would be if the original TRIPLE were used to create a loop).

12.9 Processing PURE functions in FORALL constructs

A FORALL construct containing PURE function calls appears after ARG as a `dt_group_forall` node with a list of statements below it, as shown in Section 9.6 (page 97).

STRIP performs the following transformation on such constructions:

- The `dt_group_forall` node is deleted. (Its sole purpose is to group the statements under it together so that STRIP can identify them as a unit for processing.)

- A nest of do loops (with localized loop bounds) is created for the FORALL dimensions. (The iteration space corresponding to these dimensions is created by ARG and stored as an attribute of the `dt_group_forall` node.)

- All the statements under the `dt_group_forall` node are placed in the body of that nest of do loops, and the corresponding subscripts are localized. The techniques used to perform these last two steps are just the same as those detailed earlier in this chapter for generating strip loops for local statements.

  The only statements needing localization are vanilla vector assignments. This is because the only other statements under the `dt_group_forall` are AFUNC nodes, and all the arrays under these nodes have been privatized, so no vINDEX nodes corresponding to FORALL indexes appear in their subscripts.

  Any vINDEX node that is not changed into a loop index by this process must then be changed back into its corresponding TRIPLE, as is the case in the handling of global functions (Section 12.8).

- Any unnamed vector dimensions in these statements are left alone. They are all guaranteed to be serial dimensions, and so they can be handled as normal Fortran constructs by the Middle End.
- If the `dt_group_forall` had a non-null mask attribute (coming from the original FORALL construct), then this mask expression is also localized and used to construct a guard (a `dt_simple_if`) over the list of children of the inner strip loop.

Thus, the PURE function lowering in the example on page 98 becomes

NEWSCOPE
do $I^{loc} = LB_1^{loc}, UB_1^{loc}, S_1^{loc}$
  do $J^{loc} = LB_2^{loc}, UB_2^{loc}, S_2^{loc}$
    $T2(:, :, :) = T1(I^{loc}, J^{loc}, :, :, :)$
    AFUNC($R1(:, :)$, EFUNC(F, LIST($T2(:, :, :)$)))
    $R2(I^{loc}, J^{loc}, :, :) = R1(:, :)$
  end do
end do
ENDSCOPE

Chapter 13

STRIP: Handling Motion Code

Conceptually, all interprocessor communication is handled by means of run-time library routines, with the following parameters:

SEND

- Address of the buffer to be sent.
- Length of the buffer to be sent.
- Target processor number.
- Message-type: a compiler-generated identifying message number.

RECEIVE

- Address of buffer which will hold the received message.
- Length of the message to be received.
- Message-type: must match the message type in the corresponding SEND. If the receiving processor needs to specify the processor from which the message is being sent, that information can be encoded in the message-type.

There is also a routine SEND_BROADCAST, which implements a partial broadcast, or multicast. Its parameters are identical with those of SEND, except that instead of the target processor number, there is a list of target processor numbers.

In the following, $pr(x)$ denotes the processor holding the value x, $mem(x)$ denotes the memory address of x in that processor, and p denotes the number of processors. In addition, we denote iteration allocation functions by ϕ. (Recall that data allocation functions are denoted by f.)

13.1 Remapping stores and partial broadcasts

A remapping store is an assignment statement which would be a local assignment statement (see page 121) except that the right-hand and left-hand sides are in different regions.

The most common example is

$$A(:,:) = B(:,:)$$

B may actually be an expression; but each node in that expression will be in the same region. What characterizes this as a remapping store is that A is in a different region. Thus the sections of the arrays A and B on each processor are in general not conformant.

Another example would be

$$A(V(:),:) = B(:,:)$$

where the first dimension of A is distributed serially, V is replicated over each processor, and both dimensions of B are distributed. If in this second example A and B are aligned, this is a local statement with local copy; if they are not, it is a remapping store.

By allowing the left and right hand sides of the assignment statement to contain replicated cells, we can include the kind of statements we have been referring to as partial broadcasts. It turns out that one method handles all these cases at once.

The strip miner generates the following code, where A really means "the left-hand side", and B really means "the right-hand side" of the assignment statement.

begin

Allocate an array of empty buffers $\{OUT[j] : 1 \leq j \leq p\}$ and an integer array $\{\mathbf{in}[j] : 1 \leq j \leq p\}$ initialized to 0;

Create a nest of strip mine loops based on the iteration space of B, and localize the subscripts of B.

If the local section of B has any elements in this processor, then for each such element, calculate the corresponding processors and memory addresses which hold the values of A it is being spread to, as follows: If the iteration VP space for B contains a replicated cell which is not in the same place as a replicated cell for A, and if $\bar{v}_B$ for that cell is not 0, do nothing—this processor will not generate any SENDs of B. Otherwise,

1. For each primary iteration cell of B, retrieve $v_{B,j}$ and then calculate

   $$v_{A,j} = \phi_{A,j} \circ \phi_{B,j}^{-1}(v_{B_j})$$

   where $v_{A,j}$ is the virtual cell contents of the corresponding primary cell of A. Then use the cell place information of A to derive $\bar{v}_{A,j}$ and $\hat{v}_{A,j}$ from $v_{A,j}$.
2. For each replicated iteration cell of A, set $\hat{v}_{A,j} = 0$. If the cell is in the same place as a replicated cell of B, set $\bar{v}_{A,j} = \bar{v}_{A,j}$ of the current processor. Otherwise, let $\bar{v}_{A,j}$ run over all possible values for that cell.
3. For each primary iteration cell of A which does not correspond to an iteration cell of B, let $v_{A,j}$ run over all possible values of v in that cell, and construct the corresponding values of $\bar{v}_{A,j}$ and $\hat{v}_{A,j}$.
4. For each scalar iteration cell of A, compute

   $v_{A,j} = \phi_{A,j}(\langle\text{scalar value}\rangle)$,

   and then derive $\bar{v}_{A,j}$ and $\hat{v}_{A,j}$ as before.
5. Using the values of $\bar{v}_{A,j}$ and $\hat{v}_{A,j}$ thus obtained, compute $\bar{v}_A$ and $\hat{v}_A$, from which a sequence $\langle pr(A), mem(A)\rangle$ is derived. For each element in this sequence, generate a call

   setup_remap_send($pr(A), mem(A), B$);

   -- This run-time call appends the ordered pair $\langle mem(A), B\rangle$ to the buffer $OUT[pr(A)]$.

For each nonempty buffer $OUT[j]$, generate a SEND of it to processor j.

Create a nest of strip mine loops based on the iteration space of A, and localize the subscripts of A.

If the local section of A has any elements in this processor, then for each such element, calculate the corresponding processor which is sending an element of B here, as follows:

1. For each primary iteration cell of A which corresponds to a primary cell of B, retrieve v_{A_j} and then calculate

   $v_{B,j} = \phi_{B,j} \circ \phi_{A,j}^{-1}(v_{A,j})$

   where as above $v_{B,j}$ is the virtual cell contents of the corresponding primary cell of B. Then use the cell place information of B to derive $\bar{v}_{B,j}$.

   -- Nothing further needs to be done for primary iteration cells of A which do not correspond to primary cells of B—these cells have already been represented in the loop nest created from the iteration space of A
2. For each replicated iteration cell of B, if the cell is in the same place as a replicated cell of A, set $\bar{v}_{B,j} = \bar{v}_{B,j}$ of the current processor. Otherwise, set $\bar{v}_{B,j} = 0$.
3. For each scalar iteration cell of B, compute $v_{B,j} = \phi_{B,j}(\langle\text{scalar value}\rangle)$, and then derive $\bar{v}_{B,j}$ as before.
4. Use the values of $\bar{v}_{B,j}$ to compute $\bar{v}_B = pr(B)$.
5. Generate the call

   setup_remap_recv($pr(B)$);

   -- This run-time call increments $IN[pr(B)]$.

For each $IN[j] \neq 0$, generate a RECEIVE of a message from processor j of length $IN[j]$. For each ordered pair $\langle m, v\rangle$ in that received message, assign the value v to the local memory address m.

Deallocate OUT and IN.

end

Actually, in our implementation, the receiver rather than the sender computes the memory address—this saves us from having to send the memory address to the target, and so enables us to generate smaller messages. We can do this because we make sure that the loop nests for the sending computations and receiving computations cause corresponding elements to be processed in the same order.

In a simple case where the assignment statement involves only scalar array references and there are no replicated cells, STRIP merely generates a SEND/RECEIVE pair – this is actually just a special case of the above code, but no auxiliary buffers need to be allocated. An example of this occurs on page 231.

13.2 A remote fetch

As previously noted, ITER transforms the remote fetch

$$A(:,\ :) = B(V(1\!:\!N),\ 1\!:\!M)$$

so it looks like this:

$$\text{PBRDCST}(W(1\!:\!N,\ 1\!:\!M),\ \text{DIMS} = \{2\},\ V(1\!:\!N))$$
$$A(:,\ :) = B(W(1\!:\!N,\ 1\!:\!M),\ 1\!:\!M)$$

(Again, the first statement is really a REMAP_STORE.) We consider the second statement. STRIP generates the following code:

```
Allocate a buffer BUF;
do I = 1, N
    do J = 1, M
        if W(I, J) is local then
            package(pr(B(W(I, J), J)),
                    mem(B(W(I, J), J)),
                    mem(A(I, J)),
                    BUF);
    end do
end do
do I = 1, p
    send_right(BUF);
    receive_left(BUF);
    process_fetch_request(BUF);
end do
store_fetched_values(BUF);
Deallocate BUF;
```

where *package*() appends the triple consisting of its first three arguments to a list being assembled in *BUF*. The IF test is just the test

$$\textbf{if } pr(W(I,J)) = \text{mynode}()$$

mentioned in section 12.3 above. Note that $W(I,J)$ is local if and only if $A(I,J)$ is local, because W has been created to align with A.

The size of the buffer *BUF* on each processor can be taken to be the number of elements of W (equivalently, of A) which live on that processor. In practice, the size of the buffer is taken to be an upper bound for this number, which is the same on all processors.

The fields in the triples in *BUF* have the significance

$$\langle \text{source\_processor}, \text{source\_address}, \text{target\_address} \rangle.$$

The procedure *send_right*() generates a SEND of its argument to the processor "on its right" in the ring of processors. To implement this, we could use an operating system call which tells us, for each processor, which is the processor on its right.

The procedure *receive_left*() generates a RECEIVE into its argument. The RECEIVE will be from the processor "on the left", but this does not have to be specified.

The procedure *process_fetch_request*() inspects each triple in *BUF*. For each triple in which the source_processor = *my_processor*(), the appropriate value of $B(W(I,J),J)$ is fetched (it is local, and the memory address is given by the source_address field in the triple), and this value is placed in the triple—it can overwrite any fields but the target_address field.

At the end of the loop, each buffer is back in the original processor that created it, and all the requested information has been filled in. The procedure *store_fetched_values*() then extracts this information from each triple and stores it at its corresponding address (which is the target_address field in the triple).

There is really no need to include the target_address in the message, since it will just be used by the processor that computed it when the message comes back to where it started. So actually we just compute it and save it locally, to be used when needed.

A remote fetch which involves only a single scalar array reference, such as a fetch of $A(V(I))$, say, is handled by simply broadcasting the request to all processors. Each processor executes a RECEIVE of the broadcast, and the processor holding $A(V(I))$ then executes a SEND of its value to the requesting processor, which in turn executes a RECEIVE for this value.

13.3 A remote store

Remote stores are handled similarly, except that they are a little simpler: A typical remote store is

$$A(V(1:N), 1:M) = B(:,:)$$

where the right-hand side of course could be a more general expression. After ITER, it looks like this:

$$\text{PBRDCST}(W(1{:}N,\ 1{:}M),\ \text{DIMS} = \{2\},\ V(1{:}N))$$
$$A(W(1{:}N,\ 1{:}M),\ 1{:}M) = B(:,\ :)$$

where the first statement is really a REMAP_STORE. The replacement code for the second statement is:

Allocate a buffer BUF;
do $I = 1, N$
  do $J = 1, M$
    if $W(I, J)$ is local **then**
      package($pr(A(W(I, J), J))$,
        $mem(A(W(I, J), J))$,
        $B(I, J)$,
        BUF);
  end do
end do
do $I = 1, p - 1$
  send_right(BUF);
  receive_left(BUF);
  process_store_data(BUF);
end do
Deallocate BUF;

The fields in the triples now have the significance

$$\langle \text{target\_processor}, \text{target\_address}, \text{value} \rangle.$$

The procedures *send_right*() and *receive_left*() are the same as are used above in processing remote fetches.

The procedure *process_store_data*() inspects each triple in BUF. For each triple in which target_processor = *my_processor*(), the contents of the value field is stored at the target_address.

Note that in this case the last loop runs only from 1 to $p-1$ instead of p.

A remote store which involves only a single scalar array reference, such as a store into $A(V(I))$, is handled by simply broadcasting the corresponding triple to all processors. Every processor executes a RECEIVE for that triple, and the appropriate one stores the data.

In implementing all these pseudo-code descriptions, we avoid generating messages from a processor to itself.

13.4 Dealing with reusable irregular data accesses

Irregular data accesses which occur more than once in a scoping unit are preprocessed at run-time so that (after the initial cost of preprocessing has been paid) they in effect become remapping stores. This is done as follows:

Before processing any motion code, STRIP makes a pass through the executable statements. It ignores all statements except those classified by DIVIDE as REMOTE ASSIGN or REMOTE FETCH ASSIGN. For each such statement, it identifies the remote array access, and for each array access, it identifies the vector-valued subscripts (if any) in that array access. A table is created, and each such array reference is used to create an entry in that table. The entries in the table each contain three fields:

array_reference This is the array reference itself.

vv_sub_list The list of vector-valued subscripts in that array reference. This list may be empty.

do_loop This field is initially empty.

Next, for each array reference in this table, we walk up the dotree from that reference to find the highest `dt_do_loop` such that within that loop there is no dependence involving the vector-valued subscripts in that reference. Equivalently, this is the outermost `dt_do_loop` within which the vector-valued subscripts in that reference are not changed. That `dt_do_loop` is then entered in the do_loop field of the table.

Now if there are any entries in the table with an empty do_loop field, delete those entries from the table. These entries represent irregular data accesses which will not profit from additional run-time preprocessing, and so we omit them from further consideration.

Now tag each statement containing an array reference in the table with the corresponding table entry.

(We could go farther and try to find entries in the table which correspond to array references which are formally the same but occur in different parts of the program, and see if they can be processed as a unit—this could happen if the corresponding vector-valued subscripts were not assigned to between the two occurrences. We will try to see if this will be a useful thing to do.)

Once this table has been built by the compiler, the run-time code is generated as follows:

Remote Fetches We use the same notation as in Section 13.2, and in particular, we show how to generate code for the statement which after ITER looks like this:

$$A(:,\ :) = B(W(1{:}N,\ 1{:}M),\ 1{:}M)$$

For each such statement, at the entry to the do_loop found in the corresponding entry, code is generated to perform the following actions:

Each processor allocates two arrays:

```
send_info: array [1..p] of list of record
        target_addr
        source_addr
    end

receive_info: array [1..p] of list of record
        source_addr
        target_addr
    end
```

Then each processor does the following:

```
Allocate a buffer BUF;
do I = 1, N
    do J = 1, M
        if W(I, J) is local then
            insert ⟨mem(B(W(I, J), J)), mem(A(I, J))⟩
                in receive_info[pr(B(W(I, J), J))]
            insert into BUF the record
                ⟨pr(B(W(I, J), J)), my_processor(),
                 mem(B(W(I, J), J)), mem(A(I, J))⟩
        end if
    end do
end do

-- Thus, the entries in BUF are 4-tuples having the significance
-- ⟨source_proc, target_proc, source_addr, target_addr⟩.

do K = 1, p
    send_right(BUF)
    receive_left(BUF)
    foreach 4-tuple in BUF for which source_proc = my_processor(),
        insert ⟨target_addr, source_addr⟩ in send_info[target_proc]
end do
Deallocate BUF
```

That finishes the code generated at the head of the loop which does the pre-processing. At the end of the loop, code is generated to deallocate the arrays send_info and receive_info.

Now at each occurrence of the remote fetch in the loop, the following code is generated:

do $K = 1, p$
  if the send_info[K] is non-empty, then using the entries in that array, create a message consisting of the pairs
    ⟨target_addr, my-value at source_addr⟩
  and send this message to processor K.
end do

do $K = 1, p$
  if the receive_info[K] array is non-empty, receive a message from processor K. That message will consist of pairs
    ⟨target_addr, value⟩.
  For each such pair in that message, assign value to the location target_addr in local memory.
end do

Remark The array receive_info really could be just an array of booleans, initialized to **FALSE**, and the line inserting a record into receive_info could be replaced by

receive_info[$pr(B(W(I, J),\ J))$] $\leftarrow$ **TRUE**

Remote Stores Remote stores are handled similarly. We use the same notation as in Section 13.3, and we show how to generate code for the statement which after ITER looks like this:

$$A(W(1{:}N,\ 1{:}M),\ 1{:}M) = B(:,\ :)$$

Arrays send_info and receive_info are allocated as above at the entry to the containing do_loop. Then each processor does the following:

Allocate a buffer BUF;
do $I = 1, N$
  do $J = 1, M$
    if $W(I, J)$ is local **then**
      insert ⟨$mem(A(W(I, J),\ J)),\ mem(B(I, J))$⟩
        in send_info[$pr(A(W(I, J),\ J))$]
      insert into BUF the record
        ⟨$my\_processor$`()`, $pr(A(W(I, J),\ J))$,
         $mem(B(I, J)),\ mem(A(W(I, J),\ J))$⟩
    end if
  end do
end do

-- Thus, the entries in BUF are 4-tuples having the significance
-- ⟨source_proc, target_proc, source_addr, target_addr⟩.

do $K = 1, p$
  send_right(BUF)
  receive_left(BUF)
  foreach 4-tuple in BUF for which target_proc = *my_processor*`()`,
    insert ⟨source_addr, target_addr⟩ in receive_info[source_proc]
end do
Deallocate BUF

That finishes the code generated at the head of the loop which does the pre-processing. At the end of the loop, code is generated to deallocate the arrays send_info and receive_info.

Now at each occurrence of the remote fetch in the loop, the following code is generated:

do $K = 1, p$
    if the send_info[K] is non-empty, then using the entries in that array, create a message consisting of the pairs
        ⟨target_addr, my-value at source_addr⟩
    and send this message to processor K.
end do

do $K = 1, p$
    if the receive_info[K] array is non-empty, receive a message from processor K. That message will consist of pairs
        ⟨target_addr, value⟩.
    For each such pair in that message, assign value to the location target_addr in local memory.
end do

Remark In this case, the array send_info really could be just an array of booleans, initialized to **FALSE**, and the line inserting a record into send_info could be replaced by

$$\text{send\_info}[pr(A(W(I,J),\ J))] \leftarrow \textbf{TRUE}$$

Chapter 14

STRIP: Inlined Library Functions

Some HPF and Fortran intrinsics and library functions are inlined by the compiler in STRIP. This chapter describes how the inlining is done.

14.1 Reduction functions

14.1.1 An example

We start with an example. Consider the reduction

$$B = \mathbf{sum}(A,\ \mathbf{dim} = d,\ \mathbf{mask} = M)$$

where none of the primary dimensions of A is distributed CYCLIC(m), and where A and M have dimension n and B has dimension $n-1$. We lower this as follows:

- M is first remapped if necessary so that it aligns with A.
- If the following two conditions are not satisfied, then B is replaced by a temporary T for which they are satisfied. T is subsequently remapped into B.
  - With the exception of the cell corresponding to dimension d, the iteration space for B is in the same region as that of A. That is, B is available where it is needed by A.
  - The cell of B which corresponds to the dimension d cell of A is a scalar cell. We will use $\bar{c}_d$ to denote the value in dimension d of the logical processor space represented by that cell. That is, $\bar{c}_d$ is the value of $\bar{v}_d$ represented by the constant value in that cell.

    NOTE: If we knew that $strip\_length_d$ (i.e. the strip length of the dimension d cell of A) was a power of 2, then we could just as easily create B so it was replicated over that cell. But without that knowledge, it is easier to make that cell a scalar cell.
- A temporary array S of dimension n is used to accumulate values locally. With the exception of the cell corresponding to dimension d, the cells of S and the associated data allocation maps and distribution maps are identical to those of A.

- The cell of S which corresponds to the dimension d cell of A has the same cell place as that of the A cell, and is a replicated cell.[1]

  The way that this array is used (see Figure 14.2 on page 154), it can really be implemented as a privatized scalar (again privatized through the d dimension), since only one element of S on each processor is accumulated at a time, and then is used, before the next element is accumulated. This is entirely similar to the privatized scalar S introduced below in Section 14.1.4 (page 155).

Thus, the data cell structures for A, B, and S differ only in cell d, as shown in Figure 14.1.

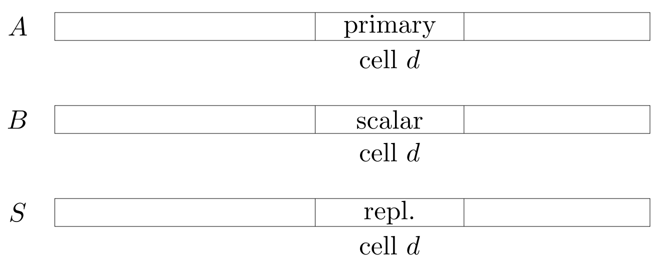


Figure 14.1: Cell d is the only cell which differs in the data space structures for A, B, and S.

The reduction happens in four steps:

1. S is initialized to 0.

2. S is used locally to accumulate the values of A.

3. Processors differing only in their values of $\bar{v}_d$ (i.e., processors having the same values of $\bar{v}_i$ for all $i \neq d$) will have parallel sections of S which need to be accumulated. A partial global reduction is used to accumulate these sections into a similar section living on the same processor as the corresponding elements of B live (i.e. the processor having the same values of $\bar{v}_i$ for $i \neq d$ and for which $\bar{v}_d = \bar{c}_d$).

   This partial global reduction is performed in a "logarithmic" fashion, so that the minimal number of messages are generated.

4. That accumulated section is then copied (locally) into the corresponding section of B.

In effect, code as in Figure 14.2 is generated.

A notational convention used in the pseudocode is that a parenthesized subscript means that the subscript is omitted—this is used in writing an $n-1$-dimensional array which is missing one of the subscripts of a similar n-dimensional array.

[1] The real intent here is that this array S is privatized over dimension d. This is an overloading of the notion of a replicated cell. The strip miner handles this by pretending that S is replicated through that dimension of the virtual processor space, but not enforcing the replication.

The operator mod is used in the sense of a non-negative value. That is,

$$0 \le x \bmod strip\_length_d \le strip\_length_d - 1.$$

The buffers INBUF and OUTBUF (allocated on each processor) each have size calculated as a product of contributions from the various dimensions, as follows: for any but CYCLIC(n) dimensions, the contribution to the size is

$$\prod_{\substack{j=1\\ j\neq d}}^{n} \left(\frac{UB_j^{loc} - LB_j^{loc}}{S_j^{loc}} + 1 \right)$$

For CYCLIC(n) dimensions, the size is found from the corresponding formula in Section 12.7. (Actually, those formulas could be used for all distributions, but the formula given here is slightly tighter, because it allows the size to be different on different processors for the non-CYCLIC(n) dimensions.)

This value is the same on processors with equal values of $\bar{v}_i$ for all $i \neq d$, so that the elementwise accumulation of INBUF into OUTBUF in the code in Figure 14.2 makes sense. In the last assignment statement in the pseudocode, OUTBUF should be thought of as a local (linear) array of shape

$$\left(\frac{UB_1^{loc} - LB_1^{loc}}{S_1^{loc}} + 1, \ldots, \frac{UB_n^{loc} - LB_n^{loc}}{S_n^{loc}} + 1 \right)$$

(where the expression corresponding to dimension d is omitted).

14.1.2 The general case

Except for MAXLOC and MINLOC, all other reductions with a **dim** argument are handled in a similar way, with the following changes:

- If a primary dimension of A is allocated CYCLIC(m), then the corresponding loop becomes two loops (in the usual fashion, as outlined in Chapter 12). The buffers INBUF and OUTBUF have in such a dimension the same extent as A (and S); thus, these buffers are not compacted in such dimensions.
- The fact that we were lowering the SUM intrinsic (as opposed to any other) was reflected in the operator "+=" and in the initialization of S. Other intrinsics correspond to other operators and other initializations, as follows:

Reduction Intrinsic	Scalar Operator	Initial Value
SUM	+	0
COUNT	+	0
PRODUCT	*	1
ALL	.AND.	.TRUE.
ANY	.OR.	.FALSE.
MAXVAL	MAX	$-\infty$
MINVAL	MIN	∞
IALL	IAND	-1
IANY	IOR	0
IPARITY	IEOR	0
PARITY	.NEQV.	.FALSE.

Allocate buffers INBUF and OUTBUF.
$S \leftarrow 0$

do $I_1^{loc} = LB_1^{loc}, UB_1^{loc}, S_1^{loc}$
    $\cdots$ (no I_d^{loc})
        do $I_n^{loc} = LB_n^{loc}, UB_n^{loc}, S_n^{loc}$
            do $I_d^{loc} = LB_d^{loc}, UB_d^{loc}, S_d^{loc}$
                if (**mask**$(\hat{v}_1, \ldots, \hat{v}_n)$)
                    $S(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n)$ += $A(\hat{v}_1, \ldots, \hat{v}_n)$
            end do
            insert $S(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n)$ into OUTBUF
        end do
    $\ldots$
end do

$I \leftarrow 1$
$\tilde{v}_d \leftarrow (\bar{v}_d - \bar{c}_d) \bmod \mathit{strip\_length}_d$
$\mathit{node\_tag} \leftarrow \tilde{v}_d$
while $I < \mathit{strip\_length}_d$ **do**
    begin
    if odd($\mathit{node\_tag}$)
        begin
        send OUTBUF to processor
            $\langle \bar{v}_1, \bar{v}_2, \ldots, (\tilde{v}_d - I + \bar{c}_d) \bmod \mathit{strip\_length}_d, \ldots, \bar{v}_n \rangle$
        break
        end
    else
        begin
        if $\tilde{v}_d + I < \mathit{strip\_length}_d$
            begin
            receive INBUF from processor
                $\langle \bar{v}_1, \bar{v}_2, \ldots, (\tilde{v}_d + I + \bar{c}_d) \bmod \mathit{strip\_length}_d, \ldots, \bar{v}_n \rangle$
            OUTBUF += INBUF (elementwise)
            end
        $\mathit{node\_tag} \leftarrow \mathit{node\_tag}/2$
        end
    $I \leftarrow 2 * I$
    end

if $\tilde{v}_d = 0$
    do $I_1^{loc} = LB_1^{loc}, UB_1^{loc}, S_1^{loc}$
        $\cdots$ (no I_d^{loc})
            do $I_n^{loc} = LB_n^{loc}, UB_n^{loc}, S_n^{loc}$
                $B(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n) \leftarrow$ OUTBUF$(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n)$
            end do
        $\ldots$
    end do

Deallocate buffers INBUF and OUTBUF.

Figure 14.2: An implementation of the SUM reduction.

14.1.3 MAXLOC and MINLOC

MAXLOC and MINLOC need special handling, because both a value and a location need to be kept track of during the computation. We show the changes needed for MAXLOC with **dim** $= d$:

- The temporary array S becomes an array of records with two fields: val and loc. The type of the val field is the base type of A, and the type of the loc field is INTEGER. The buffers OUTBUF and INBUF similarly become buffers containing records of that same form.
- The val field is initialized as usual to $-\infty$, and the loc field is initialized to any legal value; the value 1 would be fine.
- The first "+=" assignment (to S) is replaced by the following:

  if $S(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{val} < A(\hat{v}_1, \ldots, \hat{v}_n)$ **then**
  begin
  $S(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{val} \leftarrow A(\hat{v}_1, \ldots, \hat{v}_n)$
  $S(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{loc} \leftarrow \dfrac{I_d^{loc} - LB_d}{S_d} + 1$
  end

  (Note that LB_d and S_d are *not* the localized bounds LB_d^{loc} and S_d^{loc}, but are the global iteration bounds.)
- The second "+=" assignment (the elementwise assignment to OUTBUF) is replaced by the following:

do $I_1^{loc} = LB_1^{loc}, UB_1^{loc}, S_1^{loc}$
$\cdots$ (no I_d^{loc})
do $I_n^{loc} = LB_n^{loc}, UB_n^{loc}, S_n^{loc}$
if $\mathrm{OUTBUF}(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{val} < \mathrm{INBUF}(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{val}$ **then**
$\mathrm{OUTBUF}(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{loc} \leftarrow \mathrm{INBUF}(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{loc}$
end do
$\ldots$
end do

- The assignment to B at the end is replaced by

  $B(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n) = \mathrm{OUTBUF}(\hat{v}_1, \ldots, (\hat{v}_d), \ldots, \hat{v}_n).\mathrm{loc}$

14.1.4 Reductions with no dim argument

The reductions with no dim argument are performed with a somewhat simpler algorithm, but somewhat more complicated data structures.

The target B is a scalar. B will not be replicated, however: its data space structure is determined as follows:

- Cells corresponding to (i.e. having the same cell place as) primary cells of A are completion cells with value $\bar{c}_j$, where $\bar{c}_j$ is the value in that cell of some element of the iteration space.

- Cells corresponding to replicated cells of A are completion cells with value 0.
- Cells corresponding to completion cells of A are completion cells with the same value as the cells of A.

S, OUTBUF, and INBUF are local (i.e. privatized) scalars with the following data space structure:

- Cells corresponding to primary cells of A are replicated.[2]
- Cells corresponding to replicated cells of A are completion cells with value 0.
- Cells corresponding to completion cells of A are completion cells with the same value as the cells of A.

(In the case of MAXLOC and MINLOC, these are not scalars but records with two fields as above.)

We show how to generate code for

$B = \textbf{sum}(A,\ \textbf{mask} = M)$

where none of the primary dimensions of A is allocated CYCLIC(m), and where A and M have dimension n. This code, which is shown in Figure 14.3, uses some auxiliary functions, which are needed to deal with the cells of A which are replicated. We create these auxiliary functions as follows:

Say the cell places of A are denoted by

$$\{\langle m_j, s_j\rangle : 1 \le j \le n\}$$

where m_j is the *multiplier* and s_j is the *strip_length*. Let us assume that we have numbered the cells so that the first ν cells are primary cells. (So $\nu \le n$. This numbering is purely for convenience in presenting this algorithm, and is not intended to imply anything about the actual implementation.)

Further, if cell j is a completion cell, we let $\bar{c}_j =$ the value it represents in dimension j of the logical processor space. Thus, $\bar{c}_j$ is analogous to the quantity $\bar{c}_d$ used before for reductions with a **dim** argument.

We create a new set of cell places, in effect by deleting the replicated and completion cells of A and compressing the rest of the cells so there are no "holes" in the cell structure. This new set of cell places is denoted by

$$\{\langle \mu_k, \sigma_k\rangle : 1 \le k \le \nu\}$$

We create this new set of cell places as follows: Say the j^{th} "new cell" corresponds to the j^{th} "original cell". (Remember that the original first ν cells are primary). We then set

$$\sigma_j \leftarrow s_j \qquad (1 \le j \le \nu)$$

and then perform the following algorithm:

$\mu_1 \leftarrow 1$
for $k = 2$ **to** ν **do**
  $\mu_k \leftarrow \mu_{k-1} * \sigma_{k-1}$

[2]really privatized; see the footnote on page 152

At this point in the compilation process, the "squashed" cell places $\langle \mu_j, \sigma_j \rangle$ have been assigned correctly and ν has its correct value.

To translate back and forth from a real processor number to a "squashed" processor number, we create two auxiliary run-time functions:

function SQUASH(x: **integer**): **integer**
  begin
  $\mathit{val} \leftarrow 0$
  for $j = 1$ **to** ν **do**
    $$\mathit{val} \leftarrow \mathit{val} + \left(\left(\left\lfloor \frac{x}{m_j} \right\rfloor - \bar{c}_j \right) \bmod s_j \right) * \mu_j$$
  return (val)
  end

function UNSQUASH(x: **integer**): **integer**
  begin
  $\mathit{val} \leftarrow 0$
  for $j = 1$ **to** ν **do**
    $$\mathit{val} \leftarrow \mathit{val} + \left(\left(\left\lfloor \frac{x}{\mu_j} \right\rfloor + \bar{c}_j \right) \bmod \sigma_j \right) * m_j$$
  foreach completion cell with cell place $\langle m_j, s_j \rangle$
    $\mathit{val} \leftarrow \mathit{val} + \bar{c}_j * m_j$
  return (val)
  end

We also need an auxiliary run-time function to tell if a processor number has a 0 component in all the replicated cells of A and the correct value in all the completion cells of A:

function IS_RESTRICTED(x: **integer**): **boolean**
  begin
  foreach replicated cell with cell place $\langle m_j, s_j \rangle$
    if $\left\lfloor \frac{x}{m_j} \right\rfloor \bmod s_j \neq 0$ **then**
      return (**FALSE**)
  foreach completion cell with cell place $\langle m_j, s_j \rangle$
    if $\left\lfloor \frac{x}{m_j} \right\rfloor \bmod s_j \neq \bar{c}_j$ **then**
      return (**FALSE**)
  return (**TRUE**)
  end

With these functions, the code in Figure 14.3 is generated. If in fact A has no completion or replicated cells, and if is known at compile time that all $\bar{c}_j$ are 0, then $\nu = n$, no IS_RESTRICTED guard is generated, and, since the functions SQUASH and UNSQUASH would each be equivalent to the identity, no calls to them are made.

if IS_RESTRICTED(*my_processor*()) **then**
    begin
    Allocate scalars INBUF and OUTBUF.
    $S \leftarrow 0$

    do $I_1^{loc} = LB_1^{loc}, UB_1^{loc}, S_1^{loc}$
        $\ldots$
            do $I_n^{loc} = LB_n^{loc}, UB_n^{loc}, S_n^{loc}$
                if (**mask**($\hat{v}_1, \ldots, \hat{v}_n$))
                    $S \mathrel{+}= A(\hat{v}_1, \ldots, \hat{v}_n)$
            end do
        $\ldots$
    end do
    copy S into OUTBUF

    $I \leftarrow 1$
    node_tag $\leftarrow$ SQUASH(*my_processor*())
    while $I < \mu_\nu * \sigma_\nu$ **do**
        begin
        if odd(*node_tag*)
            begin
            send OUTBUF to processor
                UNSQUASH(SQUASH(*my_processor*()) $- I$)
            break
            end
        else
            begin
            if SQUASH(*my_processor*()) $+ I < \mu_\nu * \sigma_\nu$
                begin
                receive INBUF from processor
                    UNSQUASH(SQUASH(*my_processor*()) $+ I$)
                OUTBUF += INBUF
                end
            node_tag $\leftarrow$ *node_tag*/2
            end
        $I \leftarrow 2 * I$
        end

    Deallocate buffers INBUF and OUTBUF.
    end

Figure 14.3: An implementation of the SUM reduction with no **dim** argument.

14.2 Combining SCATTER functions

After DIVIDE has finished its preliminary lowering of the XXX_SCATTER functions, what STRIP has to deal with is a statement of the form

RESULT = XXX_SCATTER(SRC, INDX1, ..., INDXn, MASK)

STRIP then handles XXX_SCATTER in the same manner as a remote store, except that at the target location, instead of doing a store of the source values, the corresponding scalar operator is used to combine the source and target values.

The corresponding scalar operator is the same as that indicated previously for the reductions, except for the function COPY_SCATTER (COPY is not a reduction). For this function, the corresponding scalar operator is just the assignment of the source to the target. Thus, COPY_SCATTER provides a non-deterministic version of remote store, where there is no requirement that distinct source data elements are assigned to distinct target locations.

14.3 DOT_PRODUCT

DOT_PRODUCT is handled by expressing it in terms of other reductions, as outlined in the Fortran Language Specification:

- If the arguments are of real or integer type, then

  $\mathbf{dot\_product}(A,\ B) = \mathbf{sum}(A * B)$

  where $*$ denotes the elemental vector multipication operator.

- If the arguments are of complex type, then

  $\mathbf{dot\_product}(A,\ B) = \mathbf{sum}(\mathrm{CONJG}(A) * B)$

  where CONJG is the elemental Fortran vector complex conjugate operator.

- If the arguments are of logical type, then

  $\mathbf{dot\_product}(A,\ B) = \mathbf{any}(A\ .\mathrm{AND.}\ B)$

  where .AND. is the elemental Fortran vector logical AND operator.

14.4 Parallel random number generation

Here is how we implement the call

$$\texttt{RANDOM\_NUMBER}(A(LB_1\!:UB_1\!:S_1, \ldots, LB_n\!:UB_n\!:S_n))$$

where A is an array of rank n.

We will use the notation e_j to denote the (iteration space) extent in dimension j. To be precise, we set

$$e_j = \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor + 1$$

The effect of the call to `RANDOM_NUMBER` is to fill in the actual argument with random numbers *in array element order*. What this means is that we linearize the array and fill in the i^{th} element of the array with the i^{th} random number that our generator produces. In the case at hand, the value of i for the array reference $A(i_1, i_2, \ldots, i_n)$ is given by

$$\begin{aligned} i = & \frac{i_1 - LB_1}{S_1} \\ & + \frac{i_2 - LB_2}{S_2} * e_1 \\ & + \frac{i_3 - LB_3}{S_3} * e_1 * e_2 \\ & + \ldots \\ & + \frac{i_n - LB_n}{S_n} * e_1 * e_2 * \cdots * e_{n-1} \\ & + 1 \end{aligned}$$

or, more simply,

$$i = 1 + \sum_{k=1}^{n} \frac{i_k - LB_k}{S_k} \prod_{p=1}^{k-1} e_p$$

We start out by assuming that no distribution is CYCLIC(n)—that is, that the distributions are limited to serial, BLOCK, and CYCLIC. We will explain below in Section 14.4.5 what changes have to be made in order to support CYCLIC(n) distributions.

14.4.1 Privatized variables

Create the following privatized variables:

- Integers i and *last_i*. On each processor, *last_i* is the index (in array element order) of the last array element assigned to, and i is the index of the current array element being assigned to.

  Thus, if the statement we are compiling is

  `RANDOM_NUMBER(`$A(3\!:\!5, 0\!:\!8)$`)`

  Then i will have values between 1 and 27. Each processor will deal with a subset of these values, and these subsets will constitute a (disjoint) partition of the entire range (1:27).

- A 2-element real array *seed*[2]. The two elements of this array, *seed*[0] and *seed*[1], hold 32-bit integers that are combined to produce a 32-bit random number. The length of the period of this generator is about 2^{58}. This array is a privatized version of the (replicated) array *seed*[2] used in the current Fortran intrinsic implementation.

14.4.2 An initial implementation

A straightforward implementation is shown in Figure 14.4.

```
-- Initialize the privatized variables:

last_i ← 0
seed[0] ← the global version of seed[0]
seed[1] ← the global version of seed[1]

-- Create a nest of INDEPENDENT DO loops:

do i_n = LB_n, UB_n, S_n
    do i_{n-1} = LB_{n-1}, UB_{n-1}, S_{n-1}
        ...
        do i_1 = LB_1, UB_1, S_1

            -- Compute i and jump:
            i ← 1 + Σ_{k=1}^{n} ((i_k − LB_k)/S_k) Π_{p=1}^{k−1} e_p
            jump ← i − last_i

            -- Compute the next random number and update seed[ ]:
            A(i_1, i_2, ..., i_n) ← compute(jump, seed)

            -- Finally, update last_i:
            last_i ← i
        end do
            ...
    end do
end do

-- Update the seed:

After the do loops have finished, the global values of seed[0] and seed[1] have to be updated
    from the last iteration of the loop. (That is, the array seed is treated as if it were both
    FIRSTPRIVATE and LASTPRIVATE.)
```

where $i \leftarrow 1 + \sum_{k=1}^{n} \frac{i_k - LB_k}{S_k} \prod_{p=1}^{k-1} e_p$

Figure 14.4: Initial implementation of the parallel random number generator

The function **compute** is a simplification of the already existing C function `my_for_ran_` used in the serial Fortran implementation of the algorithm in L'Ecuyer's paper [11]. (The computation in the original version of `my_for_ran_` is unnecessarily complicated because it was designed for a machine that could only support 32-bit integer arithmetic. Since we have a 64-bit machine, our computation is completely straightforward.) The computation is shown in Figure 14.5.

In order to update the seeds at the end of this computation, we have to find the processor that has executed the (logically) last iteration. If the data is partially or wholly replicated, there may be more than one such processor. We pick one of them arbitrarily. We find the processor in question as follows: we create a Boolean expression that will be TRUE only for that processor. The expression is formed by taking the logical AND of the following four expressions:

- The first expression is the boolean constraint computed from the completion cells (if there are

function compute($jump, seed$)

> **parameter**($m_0 = 2147483563, a_0 = 40014, a\_inv_0 = 2082061899$)
> **parameter**($m_1 = 2147483399, a_1 = 40692, a\_inv_1 = 1481316021$)
> **parameter**($inv231 = 4.65661305739176\text{e-}10$)
>
> $atj_0 \leftarrow a_0^{jump} \bmod m_0$
> $seed[0] \leftarrow atj_0 * seed[0] \bmod m_0$
>
> $atj_1 \leftarrow a_1^{jump} \bmod m_1$
> $seed[1] \leftarrow atj_1 * seed[1] \bmod m_1$
>
> $z \leftarrow seed[0] - seed[1]$
> **if** $(z < 1)$ **then**
> $\quad z \leftarrow z + m_0 - 1$
> **end if**
> **return** (double)$(z * inv231)$

Figure 14.5: The function **compute**

any) by taking the logical AND of the following expression for each completion cell:

$$\bar{v}_j = \left\lfloor \frac{my\_processor()}{multiplier_j} \right\rfloor \bmod strip\_length_j \tag{14.1}$$

where $\bar{v}_j$ is computed from the completion subspace value. This is the same computation as specified in Section 12.3.

- The second expression enforces the constraint that there was at least one iteration on this particular processor:

$$\bigwedge_{j=1}^{n} \left[\left((S_j^{loc} > 0) \wedge (LB_j^{loc} \leq UB_j^{loc}) \right) \vee \left((S_j^{loc} < 0) \wedge (LB_j^{loc} \geq UB_j^{loc}) \right) \right] \tag{14.2}$$

- The third expresssion enforces the constraint that this particular processor performed the last iteration:

$$\bigwedge_{j=1}^{n} \left(LB_j^{loc} + \left\lfloor \frac{UB_j^{loc} - LB_j^{loc}}{S_j^{loc}} \right\rfloor * S_j^{loc} = LB_j + \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor * S_j \right) \tag{14.3}$$

- The fourth expression, which occurs only if there are replicated cells in the iteration space, serves to choose the processor with value 0 in each such cell. That is, for each replicated cell in the subspace, with multiplier $multiplier_j$ and strip length $strip\_length_j$, we must have

$$\left\lfloor \frac{my\_processor()}{multiplier_j} \right\rfloor \bmod strip\_length_j = 0 \tag{14.4}$$

The unique processor that satisfies all these constraints then propagates its values of *seed*[0] and *seed*[1] to the other processors, which then place these values in the replicated array *seed*[ ].

It is not immediately obvious that the second of these four expressions is really necessary. The reason it is needed is that, at least in our implementation, it is possible for the third expression to evaluate to TRUE on a processor in which the loop is empty. It doesn't seem like a good idea to change the way we handle empty loops to make sure this can't happen, since this might change again in the future. Keeping the second expression is inexpensive and makes the implementation robust.

14.4.3 Computing the possible jumps

This computation as it has been presented would really not speed things up much, if at all, because each processor would compute the same number of multiplications by a_0 and a_1 as if it performed all the iterations—the sum of the jumps is the total number of iterations.

However, we can greatly optimize this computation by realizing that there are in practice only a small number of jumps, and that we can precompute them all. We can then make a table having one entry for each precomputed jump. Each entry can contain precomputed values for the quantities atj_0 and atj_1. Once we have done this, a simple table lookup takes the place of most of the computation in the function **compute**.

So here is how the jumps are computed: the do loop nest that is created above is localized by STRIP. Figure 14.6 shows the resulting nest.

do $i_n = LB_n^{loc}, UB_n^{loc}, S_n^{loc}$
  do $i_{n-1} = LB_{n-1}^{loc}, UB_{n-1}^{loc}, S_{n-1}^{loc}$
    ...
    do $i_1 = LB_1^{loc}, UB_1^{loc}, S_1^{loc}$
      $i \leftarrow 1 + \sum_{k=1}^{n} \dfrac{i_k - LB_k}{S_k} \prod_{p=1}^{k-1} e_p$
      ...
    end do
    ...
  end do
end do

Figure 14.6: The localized loop nest.

If we let j_k denote the jump due to the variable i_k increasing by S_k^{loc} in the k^{th} loop, then we have

$$
\begin{aligned}
j_1 &= \frac{S_1^{loc}}{S_1}\\
j_2 &= \frac{S_2^{loc}}{S_2}e_1 - \frac{UB_1^{loc} - LB_1^{loc}}{S_1}\\
j_3 &= \frac{S_3^{loc}}{S_3}e_2e_1 - \frac{UB_2^{loc} - LB_2^{loc}}{S_2}e_1 - \frac{UB_1^{loc} - LB_1^{loc}}{S_1}\\
j_4 &= \frac{S_4^{loc}}{S_4}e_3e_2e_1 - \frac{UB_3^{loc} - LB_3^{loc}}{S_3}e_2e_1 - \frac{UB_2^{loc} - LB_2^{loc}}{S_2}e_1 - \frac{UB_1^{loc} - LB_1^{loc}}{S_1}
\end{aligned}
$$

and so on. That is, if we use λ_k to denote

$$\lambda_k = UB_k^{loc} - LB_k^{loc}$$

then we have

$$j_k = \frac{S_k^{loc}}{S_k}\prod_{q=1}^{k-1} e_q - \sum_{p=1}^{k-1}\frac{\lambda_p}{S_p}\prod_{q=1}^{p-1} e_q$$

To compute atj_0 and atj_1, we use the $O(\log j)$ algorithm shown in Figure 14.7.

14.4.4 Optimizing the actual jump computations

The design as just presented computes the actual jump on each iteration by first computing i and then computing $jump = i - last\_i$. This is needlessly complicated. The computations of the last section indicate a better way. We can generate the set of loops shown in Figure 14.8.

(Note that in this optimized version *jump* is initialized to j_{init} before the loop nest.) The point of this is that each of the expressions assigned to *jump* are all loop invariants. Therefore, they can all be precomputed into temporary variables before the loop starts, so the assignments are very cheap.

In fact, what we actually do is precompute the jumps and the values atj_0 and atj_1 into a table. The assignments to *jump* in Firgure 14.8 are replaced by assignments to a variable that is then used to index into this table and retrieve these three quantities.

14.4.5 Handling CYCLIC(n) distributions

Computing the jumps

Suppose now that the k^{th} loop has a CYCLIC(n) distribution. Then it turns into two loops, which we will refer to as "inner" and "outer" loops. We will consider two cases:

Case I. The $alloc\_stride_k$ and the source stride S_k are both 1. This must be true for *all* CYCLIC(n) dimensions for the computations in this case to apply.

1. Denote the multiplicative inverse of a_i mod m_i by $\overline{a_i}$ $(i = 1, 2)$. As already noted in Figure 14.5, $\overline{a_1} = 2082061899$ and $\overline{a_2} = 1481316021$.

2. Precompute the four arrays

$$P_{i,j}[k] = \begin{cases} a_i^{2^k} \bmod m_i, i = 1,2 \text{ and } k = 1,32 & \text{j = 1} \\ \overline{a_i}^{2^k} \bmod m_i, i = 1,2 \text{ and } k = 1,32 & \text{j = -1} \end{cases}$$

as follows:

function fill_in_array_P(P, a, m)

$P[1] \leftarrow a \bmod m$
do $k = 2, 32$
  $P[k] \leftarrow P[k-1] * P[k-1] \bmod m$
end do

3. Compute a^j mod m as follows: set

$$P = \begin{cases} P_{i,1} & \text{if } j > 0 \\ P_{i,-1} & \text{otherwise} \end{cases}$$

and then invoke the following function:

function a_to_the_j_mod_m(P, j, m)

$r \leftarrow 1$
$k \leftarrow 1$
while $(j > 0)$ **do**
  if $(j\&1)$ **then**
    $r \leftarrow r * P[k] \bmod m$
  end if
  $j \leftarrow j >> 1$
  $k \leftarrow k + 1$
end while
return r

Figure 14.7: An $O(\log j)$ algorithm for computing a^j

In this case, the "inner" and "outer" jumps are computed as

$$j_{inner} = \prod_{q=1}^{k-1} e_q - \sum_{p=1}^{k-1} \frac{\lambda_p}{S_p} \prod_{q=1}^{p-1} e_q$$

$$j_{outer} = \{n_k * (strip\_length_k - 1) + 1\} * \prod_{q=1}^{k-1} e_q - \sum_{p=1}^{k-1} \frac{\lambda_p}{S_p} \prod_{q=1}^{p-1} e_q$$

λ_k has to be computed as follows. (Here and in the next few lines, we suppress the subscript

$jump \leftarrow j_{init}$
do $i_n = LB_n, UB_n, S_n$
  do $i_{n-1} = LB_{n-1}, UB_{n-1}, S_{n-1}$
    ...
    do $i_2 = LB_2, UB_2, S_2$
      do $i_1 = LB_1, UB_1, S_1$
        -- Compute the next random number and update *seed*[]:
        $A(i_1, i_2, \ldots, i_n) \leftarrow$ **compute**$(jump, seed)$

        $jump \leftarrow \dfrac{S_1^{loc}}{S_1}$
      end do
      $jump \leftarrow \dfrac{S_2^{loc}}{S_2} e_1 - \dfrac{\lambda_1}{S_1}$
    end do
    $jump \leftarrow \dfrac{S_3^{loc}}{S_3} e_2 e_1 - \dfrac{\lambda_2}{S_2} e_1 - \dfrac{\lambda_1}{S_1}$
    ...
  end do
  $jump \leftarrow \dfrac{S_n^{loc}}{S_n} \prod_{q=1}^{n-1} e_q - \sum_{p=1}^{n-1} \dfrac{\lambda_p}{S_p} \prod_{q=1}^{p-1} e_q$
end do

Figure 14.8: An optimized version

k in many of the variables.)

$$\lambda_k = \overline{i_{high}^{loc}} - \overline{i_{low}^{loc}}$$

where, in the notation used in chapter 12,

$$\overline{i_{low}^{loc}} = \begin{cases} i_{low}^{loc}\Big|_{K=K^{init}} & \text{if } i_{low}^{loc}\Big|_{K=K^{init}} \leq i_{high}^{loc}\Big|_{K=K^{init}} \\ i_{low}^{loc}\Big|_{K=K^{init}+1} & \text{otherwise} \end{cases}$$

$$\overline{i_{high}^{loc}} = \begin{cases} i_{high}^{loc}\Big|_{K=K^{trm}} & \text{if } i_{low}^{loc}\Big|_{K=K^{trm}} \leq i_{high}^{loc}\Big|_{K=K^{trm}} \\ i_{high}^{loc}\Big|_{K=K^{trm}-1} & \text{otherwise} \end{cases}$$

These terms are evaluated as follows: First, we have

$$K^{init} = \left\lfloor \frac{LB_k + alloc\_offset_k}{n_k * strip\_length_k} \right\rfloor$$

$$K^{trm} = \left\lfloor \frac{UB_k + alloc\_offset_k}{n_k * strip\_length_k} \right\rfloor$$

Then just substitute the correct value of K in the following expressions:

$$i^{loc}_{low} = \max\{0,\ (\bar{v}_k + K * \mathit{strip\_length}_k) * n_k - \mathit{alloc\_offset}_k - LB_k\}$$
$$i^{loc}_{high} = \min\{\ UB_k - LB_k,\ (\bar{v}_k + K * \mathit{strip\_length}_k + 1) * n_k - \mathit{alloc\_offset}_k - LB_k - 1\}$$

Finally, we have to modify the computation of j_{init}. We also have to allow for the fact that on one or more processors, the inner loop may be empty for the first iteration of the outer loop. This happens precisely when

$$i^{loc}_{low}\Big|_{K=K^{init}} > i^{loc}_{high}\Big|_{K=K^{init}}$$

So first, in the computation for j_{init},

$$j_{init} = 1 + \sum_{k=1}^{n} \frac{LB^{loc}_k - LB_k}{S_k} \prod_{p=1}^{k-1} e_p$$

the term LB^{loc}_k must be evaluated as $LB^{loc}_k\Big|_{K=K^{init}}$ or $LB^{loc}_k\Big|_{K=K^{init}+1}$ (the second expression being used for each CYCLIC(n) dimension in which the inner loop is empty for $K = K^{init}$ on that processor). Thus, again using the notation of chapter 12, for any such dimension the factor

$$\frac{LB^{loc}_k - LB_k}{S_k}$$

must be replaced by $\overline{i^{loc}_{low}}$.

Finally, if on some processor there is at least one CYCLIC(n) dimension whose inner loop is empty for $K = K^{init}$, then letting j_{outer} denote the jump in the outer loop for the *innermost* such dimension on that processor, we replace the local initial seed on that processor by

$$\mathit{seed}[i] \leftarrow \mathit{seed}[i] * a_i^{(j_{init} - j_{outer})} \bmod m_i \qquad (i = 1, 2)$$

We do this because in fact the original j_{init} will never be used in this case. The first computation will happen with $\mathit{seed}[i]$ multiplied by $a^{j_{outer}}$ where j_{outer} is the jump in the innermost loop described above. This modification adjusts $\mathit{seed}[i]$ so that after this multiplication, we arrive at the value of j_{init}.

Case II. Any other case. We don't feel it is worth the trouble to try to precompute jumps for this case at compile time. (And we also feel that this case will seldom if ever occur.) So in this case, we revert to the initial implementation. The jumps and their associated auxiliary variables can be memoized in a run-time array as they are computed to save recomputation. But this case will certainly not run as efficiently as the other cases.

Resetting the seed

In the case of a CYCLIC(n) distribution, we have to allow for the fact that the expression (14.2) in section 14.4.2 (page 162) implicitly contains expressions for LB^{loc}_j and UB^{loc}_j that depend on the outer loop variable. (Let us call that variable $outer_j$.) At the time the loops have exited, this variable does not hold the last used value in the loop iterations.

We solve this problem as follows:

For each CYCLIC(n) loop, we create a privatized variable $saved\_outer_j$. This variable is initialized before entry to the loop nest as follows:

$$saved\_outer_j \leftarrow K_j^{init} - K_j^{incr}$$

(where in the notation used in Chapter 12, K_j^{init} and K_j^{incr} represent the initial bound and increment of the outer loop).

In each inner CYCLIC(n) loop, we insert the assignment

$$saved\_outer_j \leftarrow outer_j$$

In this way, the privatized variable $saved\_outer_j$ contains the last used value of the outer loop variable $outer_j$.

Then after the loop nest has exited, but before the evaluation of the expression (14.2) referred to above, we insert an assignment

$$outer_j \leftarrow saved\_outer_j$$

This will cause LB_j^{loc} and UB_j^{loc} to be evaluated correctly in expression (14.2).

14.4.6 The RANDOM_SEED intrinsic

Our implementation depends on the values of the random seed being the same on every processor. For this reason, the intrinsic function RANDOM_SEED is punted. What happens is this:

- If the SEED array passed to RANDOM_SEED is explicitly mapped, it is copied to a replicated temporary. This temp is then passed through to the RANDOM_SEED intrinsic routine on each processor.

  (It would seem counterproductive to distribute the seed for a random-number computation, but we have actually seen one benchmark test that did exactly this. We have no idea why. It certainly can't speed things up.)

- When RANDOM_SEED is called without any arguments, the meaning is that a seed should be picked from the clock time of the current processor. Of course this gives in general different seeds on different processors. So after the call returns, we pick the seed from processor 0, and propagate that to the other processors.

In this way, we guarantee that we have a consistent random seed on each processor.

Chapter 15

STRIP: Nearest Neighbor Computations

We define a *nearest-neighbor computation* to be a statement with the following properties:

- The statement is a vector assignment. Such an array assignment could have appeared in the source expressed using Fortran array syntax; or it could have come from a FORALL construct, provided the rest of the restrictions below are satisfied. Let us say the statement is of the form

  $$A(:,:,:) = \mathrm{expr}(:,:,:)$$

  where the expression on the right-hand side contains array references to any number of arrays, possibly including A.
- There are no vector-valued subscripts in the statement. Equivalently, there are no complex data subspace uses in any array reference in the statement.
- The statement is not completely computable locally; that is, motion is necessary at the root of the statement.
- Each array referred to in the statement is distributed either BLOCK or serially in each dimension. (Different dimensions can be distributed differently, however.)
- There are no completion or replicated cells anywhere in the statement. (This constraint could be weakened, but we don't think it is worth the trouble at this point.)
- For each array reference on the right-hand side (call it C), corresponding iteration subspaces of C and A satisfy all the conditions for being in the same region, with the exception that the effective lower bounds may differ. That is, the corresponding iteration subspaces
  1. are identically distributed (e.g. if A's first dimension is allocated BLOCK, then the same must be true for C's first dimension).
  2. have the same *strip_length*.
  3. have the same *multiplier*.
  4. have the same effective stride.

Furthermore, the effective lower bounds differ by at most some pre-set bound β. A typical value for β might be 3. We provide a command-line switch `-nearest_neighbor [width]`, whose optional parameter sets the value of β, and there is a default value in case no parameter is specified. In any case, the arrays in the statement each have to satisfy the following conditions: for each distributed dimension j,

$$\beta \leq \mathit{num\_folds}_j$$

This condition insures that corresponding elements of array references in the statement will occur on processors at most 1 away from the processor holding the element on the left-hand side, which is why we call this a nearest-neighbor computation.

In case the compiler cannot verify this (i.e., at compile-time), this constraint is regarded as not having been fulfilled, and the nearest-neighbor optimization is suppressed. This typically happens when the extents of an array are not known at compile-time; this is a common occurrence.

The compiler switch `-assume bigarrays` (which is also implied by the `-fast` switch) asserts that this condition is satisfied, so that the compiler does not make this check. If a small array is mistakenly used in a program compiled with either of these two switches and the compiler generates nearest-neighbor code for this array, the program will generally fail.

- Finally, the iteration allocation functions in the statement are all increasing, and the iteration strides are also all positive. (It is enough to check that the data allocation functions of all the arrays in the statement are all increasing—which simply means that the *alloc_stride* values are all positive—and that the source strides in the statement are all positive.)

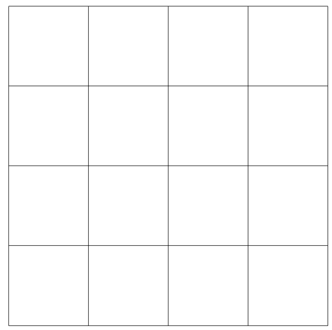
  This restriction is imposed simply to make the formulas below simpler; we think that it is unlikely to be violated in any real case. It could be eliminated however, at the cost of implementing somewhat more complicated addressing computations.

For example, the ordinary "4-point average" computation

$$\begin{aligned} A(2\!:\!N{-}1,\ 2\!:\!N{-}1) \quad = \quad & 0.25 * (A(1\!:\!N{-}2,\ 2\!:\!N{-}1) \\ & + A(3\!:\!N,\ 2\!:\!N{-}1) \\ & + A(2\!:\!N{-}1,\ 1\!:\!N{-}2) \\ & + A(2\!:\!N{-}1,\ 3\!:\!N)) \end{aligned}$$

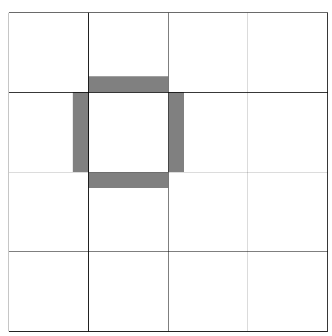
is a nearest-neighbor computation, so long as A is distributed either BLOCK with effective stride 1 or serially in each dimension: $\beta = 1$ would suffice.

If the statement is a nearest-neighbor computation, then we have each processor send "shadow edges" (sometimes also referred to as a "guard wrapper", or as "ghost cells") to each other processor that needs them before executing the computation. For instance, in the example above, if the array A is allocated (BLOCK,BLOCK) on a machine with 16 processors, then the array can be thought of as laid out as follows:



where that part of the array in each small square is allocated to the memory of one of the 16 processors.

The storage for A in each processor can be thought of as having a shadow area surrounding it which holds storage for those array elements in the neighboring processors which are needed by the computations in that processor. The shadow in the memory of one of the processors in this example looks like this:



Counting corners, the shadow region of an n-dimensional array can include components in up to $3^n - 1$ neighboring processors.

We actually allocate space in the memory of each processor to hold shadow edges of arrays that need them. This changes the addressing computations for the arrays in the statement. It also means that if an array is used in a nearest-neighbor computation and is a passed parameter or in common, then the array must be remapped on entry to a subroutine. This remapping does not involve interprocessor communication, however.

Each array that needs shadow edges (either as determined by the compiler or because of an explicit SHADOW directive) is classified as a nearest-neighbor array by means of the NEAR_NEIGHBOR handy bit.

15.1 Original version

When STRIP encounters a vSTORE operator which has been tagged by DIVIDE as a nearest-neighbor computation, it first creates a data structure to hold information about the shadow edges:

1. Let $n = \operatorname{rank} A$. ($n$ is of course known at compile time.)
2. Create an array $\mathit{shadow}[-1\!:\!1, -1\!:\!1, \ldots, -1\!:\!1]$ of rank n. Each element of this array is itself a 1-dimensional integer array of extent n (giving the dimensions of the corresponding shadow edge). Initialize this array to 0.
3. Let L denote the occurrence of A on the left-hand side of the statement. For $i = 1$ to n, let LB_j^{loc} denote the localized lower iteration bound of L in the j^{th} dimension. Let ϕ_j^L denote the iteration allocation function for L.

   For each occurrence R of A on the right-hand side, let ϕ_j^R denote the iteration allocation function for R.

   (a) For $i = 1$ to n, let $\delta_j = \phi_j^R(LB_j^{loc}) - \phi_j^L(LB_j^{loc})$. Let $\alpha_j = \operatorname{sign}(\delta_j)$.

   (b) Invoke process_shadow_bounds(1, $\alpha_1, \alpha_2, \ldots, \alpha_n$).

   where the recursive procedure process_shadow_bounds() is defined in Figure 15.1. This procedure enumerates all subsets of $\{1 \ldots n\}$, and for each subset, fills in information concerning the corresponding shadow edge for the given array occurrence.

```
procedure process_shadow_bounds(index, λ1, λ2, ..., λn)
begin
    if index = n + 1 then begin
        for i = 1 to n do begin
            if λi = 0 then continue;
            if λi = 1 then
                shadow[λ1, ..., λn][i] ← max{shadow[λ1, ..., λn][i], δi};
            else if λi = −1 then
                shadow[λ1, ..., λn][i] ← min{shadow[λ1, ..., λn][i], δi};
        end for
    else
        begin
            process_shadow_bounds(index + 1, λ1, λ2, ..., λn);
            if λindex ≠ 0 then begin
                λindex ← 0;
                process_shadow_bounds(index + 1, λ1, λ2, ..., λn)
            end if
        end
    end if
end
```

Figure 15.1: Definition of the recursive procedure process_shadow_bounds().

When this has been done, we know that the absolute value of each entry of the *shadow* array is $\leq \beta$. There is no communication in a given dimension i if and only if

$$\mathit{shadow}[\alpha_1, \alpha_2, \ldots, \alpha_n][i] = 0$$

for all choices of the numbers $\{\alpha_j : 1 \leq j \leq n\}$.

Note also that the sign of $\mathit{shadow}[\alpha_1, \alpha_2, \ldots, \alpha_n][i]$ is the same as the sign of α_i.

Each shadow edge now corresponds to an element

$$\mathit{shadow}[\alpha_1, \ldots, \alpha_n][1 : n]$$

which is not $\vec{0}$. (By "element" here we mean a choice of the n-tuple $[\alpha_1, \ldots, \alpha_n]$.)

A similar computation is carried out for each array name B occurring on the right-hand side of the statement. Thus, there is a separate *shadow* data structure constructed for each distinct array used in the statement. In constructing the *shadow* array for B, l_i is always determined by the occurrence of A on the left-hand side, while r_i is determined by a particular occurrence of B on the right-hand side.

For each array B used in the nearest-neighbor computation, space will have to be allocated in each processor to hold not only that part of B which lives on the processor, but also to hold space for the shadow edges. We do this as follows. The dimensions of B which are allocated serially, or which are distributed BLOCK but whose corresponding data cell has a *strip_length* of 1, do not need any additional space, and are addressed normally. Each remaining dimension of B (which must be distributed BLOCK with $\mathit{strip\_length} > 1$), instead of contributing a factor of $\mathit{num\_folds}_j$ to the storage reserved locally for B, contributes a factor of

$$\mathit{num\_folds}_j + 2 * \beta.$$

That is, we reserve a space of width β "on each side" of the normal space reserved for B in each of these dimensions.

The addressing in each of these dimensions then has to be offset by

$$\mathit{off}_j = \beta.$$

In particular, in any statement which is not a nearest-neighbor computation, the local addressing computation for an occurrence of B is modified by adding off_j in the j^{th} dimension.

After having enumerated the shadow edges in this manner, the strip miner performs the following actions:

- it generates loops to load the shadow edges, and then
- it performs regular strip mining on the statement itself.

First, we show how to perform regular strip mining on the statement itself:

The strip loops are generated in the usual fashion. To localize subscripts, we use a slight variant of the ordinary formula for BLOCK dimensions on page 132

$$\hat{v}_j = f_j(s_j) - \left\lfloor \frac{\phi_j(LB_j^{loc})}{\mathit{num\_folds}_j} \right\rfloor * \mathit{num\_folds}_j.$$

as follows:

For each triple on the array reference on the left-hand side, we define

$$L_j = num\_folds_j * \left\lfloor \frac{\phi_j(LB_j^{loc})}{num\_folds_j} \right\rfloor .$$

(Note that for a serial dimension this value is 0.)

Now we could just write $\hat{v}_j = \mathit{off}_j + f_j(s_j) - L_j$, except that we are assuming we are starting with a TRIPLE, not a scalar subscript. Therefore, we use the fact that $f_j(s_j) = \phi_j(I_j)$. We have to be careful to note, however, that here I_j denotes the normalized value of the loop index, as computed on page 130. In addition, ϕ_j differs for different references to the same array—this reflects the fact that the differenct references do not strictly align, but may differ by a value bounded by the shadow width. This allows us to generate addressing that references elements in the shadow edges.

Thus for each corresponding triple $LB_j : UB_j : S_j$ in any array reference in the statement, the localized address becomes

$$\hat{v}_j = \mathit{off}_j + \phi_j(I_j) - L_j$$

where, as just explained, this ϕ_j differs for different array references.

It is interesting to note that this expression, which originates as an optimization of the regular BLOCK addressing computation involving a MOD operator, is actually necessary for nearest-neighbor computations, because it enables us to address the shadow edges. The MOD computation would actually be incorrect in this case.

Next, to load the shadow edges, we have to

- calculate the processor that the shadow edge comes from,
- calculate the addresses in that processor holding the shadow edge, and
- calculate the addresses in the current processor into which the shadow edge will be loaded.

In calculating these addresses, we use the ϕ_j of the left-hand side, offsetting the result by the computed shadow widths.

We let Δ_j denote the *data* allocation stride in the j^{th} dimension (i.e. the $alloc\_stride_j$ obtained from data space). Δ_j is the increment in virtual processor space of successive elements in the j^{th} dimension. As a consequence, $shadow[\alpha_1, \ldots, \alpha_n][j]$ is automatically a multiple of Δ_j.

Now each shadow edge corresponds to a choice of $[\alpha_1, \ldots, \alpha_n]$. If as usual $\bar{v} = [\bar{v}_1, \ldots, \bar{v}_n]$ designates the current processor, the processor holding the shadow edge is given by

$$[\bar{v}_1 + \alpha_1, \ldots, \bar{v}_n + \alpha_n].$$

Using the computations indicated for localizing subscripts, we calculate LB_j^{loc} and UB_j^{loc} for the left-hand side array reference for each dimension j. The quantity L_j is calculated as indicated above. If j is a BLOCK dimension, the addresses in the source processor of the shadow edge are given by the triple

$$\hat{v}_{low}^{loc} : \hat{v}_{high}^{loc} : \hat{v}_{str}^{loc}$$

(we suppress including j as a subscript here)

where if $\alpha_j > 0$

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \mathit{off}_j + (\phi_j(LB^{loc}_j) - L_j) \bmod \Delta_j \\
\hat{v}^{loc}_{high} &= \hat{v}^{loc}_{low} + \mathit{shadow}[\alpha_1, \ldots, \alpha_n][j] - \Delta_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $\alpha_j < 0$

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \hat{v}^{loc}_{high} + \mathit{shadow}[\alpha_1, \ldots, \alpha_n][j] + \Delta_j \\
\hat{v}^{loc}_{high} &= \mathit{off}_j + \phi_j(UB^{loc}_j) - L_j + \left\lfloor \frac{\mathit{num_folds}_j - (\phi_j(UB^{loc}_j) - L_j)}{\Delta_j} \right\rfloor * \Delta_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $\alpha_j = 0$

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \mathit{off}_j + \phi_j(LB^{loc}_j) - L_j \\
\hat{v}^{loc}_{high} &= \mathit{off}_j + \phi_j(UB^{loc}_j) - L_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

There is an additional point to be noted here: In computing the localized values LB^{loc}_j and UB^{loc}_j in these formulas, the values LB_j and UB_j have to be adjusted as follows:

$$LB_j \leftarrow LB_j - \mathit{shadow}[\alpha_1, \ldots, \alpha_n][j]$$
$$UB_j \leftarrow UB_j - \mathit{shadow}[\alpha_1, \ldots, \alpha_n][j]$$

The reason for this is that we may need to receive values from processors that are beyond the end of the array on the left-hand side; we need to have the localized bounds on those processors set meaningfully. This greatly complicates the computations; this is one reason why we do not in fact use this method, but use a simplified procedure to be discussed in the next section.

If j is a BLOCK dimension, the addresses in the target processor into which the shadow edge will be loaded are given by the triple

$$\hat{v}^{loc}_{low} : \hat{v}^{loc}_{high} : \hat{v}^{loc}_{str}$$

where if $\alpha_j > 0$

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \mathit{off}_j + \phi_j(UB^{loc}_j) - L_j + \left\lfloor \frac{\mathit{num_folds}_j - (\phi_j(UB^{loc}_j) - L_j)}{\Delta_j} \right\rfloor * \Delta_j + \Delta_j \\
\hat{v}^{loc}_{high} &= \hat{v}^{loc}_{low} + \mathit{shadow}[\alpha_1, \ldots, \alpha_n][j] - \Delta_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $\alpha_j < 0$

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \hat{v}^{loc}_{high} + \mathit{shadow}[\alpha_1, \ldots, \alpha_n][j] + \Delta_j \\
\hat{v}^{loc}_{high} &= \mathit{off}_j + (\phi_j(LB^{loc}_j) - L_j) \bmod \Delta_j - \Delta_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $\alpha_j = 0$

$$
\begin{aligned}
\hat{v}^{loc}_{low} &= \mathit{off}_j + \phi_j(LB^{loc}_j) - L_j \\
\hat{v}^{loc}_{high} &= \mathit{off}_j + \phi_j(UB^{loc}_j) - L_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}
$$

For each shadow edge, the strip miner packages up all the elements in that edge (in the source processor) into a buffer and sends that buffer to the target processor. Each target processor generates a receive for that information and assigns the data in that message *in the same order* to the shadow edge in its memory.

Actually, this is not quite true: processors holding elements on the edge of the array do not generate sends or receives to or from processors which would be nonexistent or holding elements not in the array. Therefore, a test has to be performed in each processor for each shadow edge, as follows:

For each dimension of an array reference containing a triple, and distributed BLOCK with $strip\_length_j > 1$, we have

$$
\begin{aligned}
\bar{v}_{j,low} &= \left\lfloor \frac{\phi_j(LB_j)}{num\_folds_j} \right\rfloor \\
\bar{v}_{j,high} &= \left\lfloor \frac{\phi_j\left(LB_j + S_j * \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor\right)}{num\_folds_j} \right\rfloor
\end{aligned}
$$

For each shadow edge given by $[\alpha_1, \ldots, \alpha_n]$,

- For each $\alpha_j = -1$,

  The SEND of that edge is suppressed unless

  $$\bar{v}_{j,low} \le \bar{v}_j < \bar{v}_{j,high}$$

  (This corresponds to motion in the positive direction along the j axis.)

- For each $\alpha_j = 1$,

  The SEND of that edge is suppressed unless

  $$\bar{v}_{j,low} < \bar{v}_j \le \bar{v}_{j,high}$$

  (This corresponds to motion in the negative direction along the j axis.)

- For each $\alpha_j = -1$,

  The RECEIVE of that edge is suppressed unless

  $$\bar{v}_{j,low} < \bar{v}_j \le \bar{v}_{j,high}$$

  (This corresponds to motion in the positive direction along the j axis.)

- For each $\alpha_j = 1$,

  The RECEIVE of that edge is suppressed unless

  $$\bar{v}_{j,low} \leq \bar{v}_j < \bar{v}_{j,high}$$

  (This corresponds to motion in the negative direction along the j axis.)

Regular strip mining is then performed on the statement itself, using the addressing indicated above.

In addition, STRIP has to change the addressing computations for these arrays when they are used in non-nearest-neighbor computations. For these statements, STRIP adds off_j to each value of $\hat{v}_j$.

15.2 Simplified version

We have actually implemented a modified version of these nearest-neighbor algorithms. The modified version in in some respects simplified, although in accomodating different actual sizes of shadow edges it is slightly more general. It may well be that this modified version is better than the original design in general. It has the advantage that in general, fewer sends and receives are generated (although in the most typical cases, there will be no difference), and it has the disadvantage that in general more data is moved.

The simplifications we have made are as follows:

- The storage allocated to each shadow edge has effective width β_j in both directions (positive and negative) in the j^{th} coordinate of the iteration space. This is somewhat more general than the original design, in which all the β_j are equal. (Note that not all of this storage may be moved in a particular computation, however. Only that part of the shadow edge that is actually needed is moved.)
- Instead of $3^n - 1$ moves being needed, we generate at most 2^n moves, by moving larger strips. Each strip which is sent to a neighboring processor consists of the elements of the array which will become part of that neighbor's shadow edge, plus all other elements from that array's shadow edges which have images in the neighbor's shadow edges and which have already received values from other processors. This way, each processor only has to send and receive from its two "orthogonal" neighbors in each dimensional direction—there is no "diagonal" motion.

As before, in calculating these addresses, we use the ϕ_j of the left-hand side, offsetting the result by the computed shadow widths.

The recursive procedure to compute the *shadow* array is no longer needed, because the *shadow* array itself is no longer needed—we already know the size of the shadow in every direction.

The rest of this section describes how the formulas for computing and moving shadow edges are modified.

An index j (or k, etc.) in these formulas is interpreted as follows:

- If the field being fetched is from the data space, j is the index of a subspace in the data space.

- If the field being fetched is from the iteration space, j refers to the index of the subspace of the iteration space that corresponds to the j subspace of the data space.

In general (and unless specifically stated otherwise below), all data and iteration spaces are those of the left-hand side.

As before, $\bar{v} = [\bar{v}_1, \ldots, \bar{v}_n]$ designates the current processor. For each k from 1 to n, this processor will send shadow edge data to the processor $[\bar{v}_1, \ldots, \bar{v}_k + 1, \ldots, \bar{v}_n]$ (we call this a *positive* send) and also to the processor $[\bar{v}_1, \ldots, \bar{v}_k - 1, \ldots, \bar{v}_n]$ (we call this a *negative* send).

Let us use Δ_j to stand for the *data* allocation stride in the j^{th} dimension (i.e. $alloc\_stride_j$ obtained from data space).

Now to compute the values of $\hat{v}_j$ being sent and received, we use data space, rather than iteration space. We derive the formulas in the same way as the formulas for BLOCK distribution were derived in Chapter 12 for localizing iteration space bounds, but this time we apply them to data space bounds.

The formulas become simpler in this case, because there is no equivalent of iteration stride. Each coordinate of an array varies over a contiguous set of values. Therefore, instead of considering $\phi_j(S_j * i + LB_j)$ where $0 \le S_j * i + LB_j \le \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor$, we simply consider $f_j(i)$, where $DLB_j \le i \le DUB_j$.

In this way, for each dimension distributed BLOCK, we get local lower and upper bounds of the array on the left hand side in each dimension j. Let us call these bounds dLB_j and dUB_j. The formulas we derive for these values are

$$dLB_j = \begin{cases} \max\left\{ DLB_j, \left\lceil \dfrac{num\_folds_j * \bar{v}_j - alloc\_offset_j}{alloc\_stride_j} \right\rceil \right\} & \text{if } f_j \uparrow \\ \max\left\{ DLB_j, \left\lfloor \dfrac{num\_folds_j * (\bar{v}_j + 1) - alloc\_offset_j}{alloc\_stride_j} \right\rfloor + 1 \right\} & \text{if } f_j \downarrow \end{cases}$$

$$dUB_j = \begin{cases} \min\left\{ DUB_j, \left\lceil \dfrac{num\_folds_j * (\bar{v}_j + 1) - alloc\_offset_j}{alloc\_stride_j} \right\rceil - 1 \right\} & \text{if } f_j \uparrow \\ \min\left\{ DUB_j, \left\lfloor \dfrac{num\_folds_j * \bar{v}_j - alloc\_offset_j}{alloc\_stride_j} \right\rfloor \right\} & \text{if } f_j \downarrow \end{cases}$$

Note that the quantities $alloc\_stride_j$ and $alloc\_offset_j$ in these formulas refer to the data allocation function f_j of the array on the left-hand side.

Now while these formulas indicate the least and greatest array elements on a given processor, what we are going to use them for is to specify a range of memory addresses. We want this range to go from lower to higher. Therefore, in the case when $f \downarrow$, we must interchange dLB_j and dUB_j. That is, we define "normalized versions" nLB_j and nUB_j as follows:

$$nLB_j = \begin{cases} dLB_j = \max\left\{DLB_j, \left\lceil \dfrac{num\_folds_j * \bar{v}_j - alloc\_offset_j}{alloc\_stride_j} \right\rceil\right\} & \text{if } f_j \uparrow \\ dUB_j = \min\left\{DUB_j, \left\lfloor \dfrac{num\_folds_j * \bar{v}_j - alloc\_offset_j}{alloc\_stride_j} \right\rfloor\right\} & \text{if } f_j \downarrow \end{cases}$$

$$nUB_j = \begin{cases} dUB_j = \min\left\{DUB_j, \left\lceil \dfrac{num\_folds_j * (\bar{v}_j + 1) - alloc\_offset_j}{alloc\_stride_j} \right\rceil - 1\right\} & \text{if } f_j \uparrow \\ dLB_j = \max\left\{DLB_j, \left\lfloor \dfrac{num\_folds_j * (\bar{v}_j + 1) - alloc\_offset_j}{alloc\_stride_j} \right\rfloor + 1\right\} & \text{if } f_j \downarrow \end{cases}$$

Defined in this way, we have $f_j(nLB_j) \leq f_j(nUB_j)$, unless the array is a 0-sized array.

When the strip length is 1, these formulas become much simpler. In fact, they describe what happens in a serial dimension, of the array. We pull this out as a special case:

For serial dimensions, we make use (as on page 73) of the "normalized" lower and upper declared bounds of the array on the right-hand side (i.e., the array whose shadow edges are being filled in):

$$NLB_j = \begin{cases} DLB_j & \text{if } f_j \uparrow \\ DUB_j & \text{if } f_j \downarrow \end{cases}$$

$$NUB_j = \begin{cases} DUB_j & \text{if } f_j \uparrow \\ DLB_j & \text{if } f_j \downarrow \end{cases}$$

Defined in this way, we have $f_j(NLB_j) \leq f_j(NUB_j)$, unless the array is a 0-sized array.

The formulas make use of two additional quantities that the compiler computes: The quantity β_k^+ is the greatest effective positive send for a given array in the statement (or group of statements) being processed, and β_k^- is the corresponding greatest effective negative send. (Both β_k^+ and β_k^- are taken to be non-negative quantities.) For instance, in the statement

$$A(i,j) = B(i-1, j-2) + B(i-3, j+1) + A(i+1, j-2)$$

where B is aligned with A and the iteration allocation strides are all 1, we have

- For the array A:

$$\beta_1^+ = 0 \qquad \beta_2^+ = 2$$
$$\beta_1^- = 1 \qquad \beta_2^- = 0$$

- For the array B:

$$\beta_1^+ = 3 \qquad \beta_2^+ = 2$$
$$\beta_1^- = 0 \qquad \beta_2^- = 1$$

Now we are ready to compute the bounds of the shadow edges that are actually sent and received.

For each array on the right-hand side, and for each BLOCK dimension k, we compute the values of $\hat{v}_j$ for the shadow edge being sent in the k direction as follows:

The values of $\hat{v}_j$ in the source processor for the shadow information being sent are given by the triple

$$\hat{v}^{loc}_{low} : \hat{v}^{loc}_{high} : \hat{v}^{loc}_{str}$$

The values of the elements in this triple are given by the following formulas. Everything in these formulas *except* $f_j(nLB_j)$, $f_j(nUB_j)$, $f_j(NLB_j)$ and $f_j(NUB_j)$ refers to values from the data space of the array on the right-hand side whose shadow edges are being filled in.

If $k = j$ and the send is negative,

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \mathit{off}_j + f_j(nLB_j) \bmod \mathit{num_folds}_j \\
\hat{v}^{loc}_{high} &= \hat{v}^{loc}_{low} + \left(\left\lfloor \frac{\beta^-_j}{\Delta_j} \right\rfloor - 1 \right) * \Delta_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $k = j$ and the send is positive,

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \hat{v}^{loc}_{high} - \left(\left\lfloor \frac{\beta^+_j}{\Delta_j} \right\rfloor - 1 \right) * \Delta_j \\
\hat{v}^{loc}_{high} &= \mathit{off}_j + f_j(nUB_j) \bmod \mathit{num_folds}_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $k \neq j$,

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \begin{cases} \mathit{off}_j + f_j(nLB_j) \bmod \mathit{num_folds}_j - h(k, j, \beta^+_j) & \text{if } j \text{ is a BLOCK dimension} \\ f_j(NLB_j) & \text{if } j \text{ is a SERIAL dimension} \end{cases} \\
\hat{v}^{loc}_{high} &= \begin{cases} \mathit{off}_j + f_j(nUB_j) \bmod \mathit{num_folds}_j + h(k, j, \beta^-_j) & \text{if } j \text{ is a BLOCK dimension} \\ f_j(NUB_j) & \text{if } j \text{ is a SERIAL dimension} \end{cases} \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

where

$$h(k, j, \beta^\pm_j) = \begin{cases} \left\lfloor \frac{\beta^\pm_j}{\Delta_j} \right\rfloor * \Delta_j & \text{if } j < k \\ 0 & \text{otherwise (i.e., if } j > k) \end{cases}$$

The use of the term $h(k, j, \beta^\pm_j)$ implements a small optimization: a "corner" of a shadow edge only has to be sent after something has been received into its preimage. $h(k, j)$ is defined to make this happen: successively larger shadow edges are sent as k increases.

(Note that while the quantity *shadow* (in the previous section of this chapter) can be positive or negative, the numbers $\beta^\pm_j$ are always positive.)

The reason the terms off_j and $h(k, j, \beta^\pm_j)$ do not appear in the case that j is a serial dimension is that such a dimension has no shadow edges.

Next, for each BLOCK dimension k of an array on the right-hand side, we compute the values of $\hat{v}_j$ for the shadow edge being received in the k direction as follows:

The values of $\hat{v}_j$ in the target processor for the shadow information being sent are given by the triple

$$\hat{v}^{loc}_{low} : \hat{v}^{loc}_{high} : \hat{v}^{loc}_{str}$$

The values of the elements in this triple are given by the following formulas. As before, everything in these formulas *except* $f_j(nLB_j)$, $f_j(nUB_j)$, $f_j(NLB_j)$ and $f_j(NUB_j)$ refers to values from the data space of the array on the right-hand side whose shadow edges are being filled in.

If $k = j$ and we're receiving a negative send,

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \mathit{off}_j + f_j(nUB_j) \bmod \mathit{num_folds}_j + \Delta_j \\
\hat{v}^{loc}_{high} &= \hat{v}^{loc}_{low} + \left(\left\lfloor \frac{\beta^-_j}{\Delta_j} \right\rfloor - 1 \right) * \Delta_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $k = j$ and we're receiving a positive send,

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \hat{v}^{loc}_{high} - \left(\left\lfloor \frac{\beta^+_j}{\Delta_j} \right\rfloor - 1 \right) * \Delta_j \\
\hat{v}^{loc}_{high} &= \mathit{off}_j + f_j(nLB_j) \bmod \mathit{num_folds}_j - \Delta_j \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

and if $k \neq j$,

$$\begin{aligned}
\hat{v}^{loc}_{low} &= \begin{cases} \mathit{off}_j + f_j(nLB_j) \bmod \mathit{num_folds}_j - h(k, j, \beta^+_j) & \text{if } j \text{ is a BLOCK dimension} \\ f_j(NLB_j) & \text{if } j \text{ is a SERIAL dimension} \end{cases} \\
\hat{v}^{loc}_{high} &= \begin{cases} \mathit{off}_j + f_j(nUB_j) \bmod \mathit{num_folds}_j + h(k, j, \beta^-_j) & \text{if } j \text{ is a BLOCK dimension} \\ f_j(NUB_j) & \text{if } j \text{ is a SERIAL dimension} \end{cases} \\
\hat{v}^{loc}_{str} &= \Delta_j
\end{aligned}$$

where $h(k, j, \beta^{\pm}_j)$ is defined as above.

Next, we compute Boolean expressions to determine which physical processors are actually involved in sending and receiving. We compute these expressions mainly in terms of the array on the left-hand side. So in the following formulas, all values are those of that array, with the exception of $\beta^{\pm}_j$. The Boolean expressions are computed as follows:

For any component j other than the component k corresponding to the direction of the motion, the j^{th} component of the virtual processors sending or receiving any shadow edge will lie in the interval

$$\phi_j(LB_j) : \phi_j \left(LB_j + S_j * \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor \right)$$

and therefore, the actual component of the virtual processors sending or receiving any shadow edge lies in the interval $\bar{v}_{j,low} : \bar{v}_{j,high}$, where the bounds are given by the same formulas as on page 176:

$$\bar{v}_{j,low} = \left\lfloor \frac{\phi_j(LB_j)}{num\_folds_j} \right\rfloor$$

$$\bar{v}_{j,high} = \left\lfloor \frac{\phi_j\left(LB_j + S_j * \left\lfloor \frac{UB_j - LB_j}{S_j} \right\rfloor\right)}{num\_folds_j} \right\rfloor$$

For the components in the direction of the motion (i.e., for the k components), we have a different set of formulas, which look at first rather strange. These formulas compute four send bounds and four receive bounds:

actual processor component bound	direction	send	receive
low	right	$\bar{v}^{s+}_{k,low}$	$\bar{v}^{r+}_{k,low}$
high	right	$\bar{v}^{s+}_{k,high}$	$\bar{v}^{r+}_{k,high}$
low	left	$\bar{v}^{s-}_{k,low}$	$\bar{v}^{r-}_{k,low}$
high	left	$\bar{v}^{s-}_{k,high}$	$\bar{v}^{s-}_{k,high}$

The send bounds can then be computed as follows:

$\bar{v}^{s+}_{k,low} = \left\lfloor \frac{\phi_k(LB_k) - \beta^+_k}{num\_folds_k} \right\rfloor$	$\bar{v}^{s+}_{k,high} = \left\lfloor \frac{\phi_k\left(LB_k + S_k * \left\lfloor \frac{UB_k - LB_k}{S_k} \right\rfloor\right)}{num\_folds_k} \right\rfloor - 1$
$\bar{v}^{s-}_{k,low} = \left\lfloor \frac{\phi_k(LB_k)}{num\_folds_k} \right\rfloor + 1$	$\bar{v}^{s-}_{k,high} = \left\lfloor \frac{\phi_k\left(LB_k + S_k * \left\lfloor \frac{UB_k - LB_k}{S_k} \right\rfloor\right) + \beta^-_k}{num\_folds_k} \right\rfloor$

This can be understood as follows:

- When sending to the right, the lower bound in the k^{th} coordinate of the set of *virtual* sending processors is $\phi_k(LB_k) - \beta^+_k$.
- On the other hand, the upper bound in the k^{th} coordinate of the set of *physical* sending processors is just 1 less than the upper bound of the receiving processors.
- When sending to the left, the reasoning is similar, but the role of upper and lower bounds is interchanged.

Now the receive bounds can be computed as follows: we must have

$$\bar{v}^{r+}_* = \bar{v}^{s+}_* + 1$$
$$\bar{v}^{r-}_* = \bar{v}^{s-}_* - 1$$

and so

$\bar{v}^{r+}_{k,low} = \left\lfloor \dfrac{\phi_k(LB_k) - \beta^+_k}{num\_folds_k} \right\rfloor + 1$	$\bar{v}^{r+}_{k,high} = \left\lfloor \dfrac{\phi_k\left(LB_k + S_k * \left\lfloor \frac{UB_k - LB_k}{S_k} \right\rfloor\right)}{num\_folds_k} \right\rfloor$
$\bar{v}^{r-}_{k,low} = \left\lfloor \dfrac{\phi_k(LB_k)}{num\_folds_k} \right\rfloor$	$\bar{v}^{r-}_{k,high} = \left\lfloor \dfrac{\phi_k\left(LB_k + S_k * \left\lfloor \frac{UB_k - LB_k}{S_k} \right\rfloor\right) + \beta^-_k}{num\_folds_k} \right\rfloor - 1$

We note as a check that if $\bar{v}^{s+}_{k,low} = \bar{v}^{r+}_{k,high}$, then $\bar{v}^{s+}_{k,low} > \bar{v}^{s+}_{k,high}$, so no positive send will in fact be generated—this will be enforced by the Boolean expressions computed immediately below. For the same reason, in such a case no positive receive will be generated, and similar considerations apply to negative sends and receives.

Finally, we can define the promised Boolean expressions, which enable us to determine which processors actually send and which receive. First define

$$A_k = \bigwedge_{i=1}^{k-1} (\bar{v}_{i,low} \le \bar{v}_i \le \bar{v}_{i,high})$$

$$B_k = \bigwedge_{i=k+1}^{n} (\bar{v}_{i,low} \le \bar{v}_i \le \bar{v}_{i,high})$$

(By convention, $A_1 = B^n = \mathbf{TRUE}$.)

Then for $k = 1$ to n (in that order), SENDs and RECEIVEs are generated, subject to the following restrictions:

- A positive SEND is generated if and only if

$$A_k \wedge (\bar{v}^{s+}_{k,low} \le \bar{v}_k \le \bar{v}^{s+}_{k,high}) \wedge B_k$$

- A negative SEND is generated if and only if

$$A_k \wedge (\bar{v}^{s-}_{k,low} \le \bar{v}_k \le \bar{v}^{s-}_{k,high}) \wedge B_k$$

- A RECEIVE of a positive SEND is generated if and only if

$$A_k \wedge (\bar{v}^{r+}_{k,low} \le \bar{v}_k \le \bar{v}^{r+}_{k,high}) \wedge B_k$$

- A RECEIVE of a negative SEND is generated if and only if

$$A_k \wedge (\bar{v}^{r-}_{k,low} \le \bar{v}_k \le \bar{v}^{r-}_{k,high}) \wedge B_k$$

15.3 COMMON arrays and parameter passing

The changes in addressing outlined in this chapter apply most straightforwardly to arrays which are declared locally to a routine.

Non-local arrays that are used in nearest-neighbor contexts in a subroutine need to be remapped on entry (and possibly on exit) from that subroutine unless the compiler can determine that they are already equipped with appropriate shadow edges. The SHADOW directive is used for this purpose. The rest of the discussion below describes how we handle the case when the compiler has no such information, or when the compiler can determine that a remapping is needed in any case.

COMMON arrays without shadow edges that are used in nearest-neighbor computations must be remapped to provide for shadow edges at the beginning of the routine and remapped back on exit from the routine.

We now show what has to be done to handle parameters without shadow edges. All the motion which is necessary to support these parameters is introduced by the DIVIDE phase.

First we consider data which is passed in to a called subroutine:

If an array is a passed parameter, or is in COMMON, does not have shadow edges, and is used in a nearest-neighbor computation, that array must be remapped on entry (i.e. at the beginning of the routine). This remapping takes the form of a local statement—it does not involve interprocessor communication. Before the subroutine exits, the array must be restored to its original position.

Now we consider data which is to be passed to a called subroutine:

Before calling a subroutine, every array which is used in a nearest-neighbor computation and is an actual parameter to the call, and also every array in COMMON which is used in a nearest-neighbor computation in the calling routine, is remapped to eliminate the shadow edges before calling the routine; and it is restored to its original mapping when the subroutine returns. (Of course this does not happen if there is an explicit interface that indicates that the corresponding dummy has equivalent shadow edges.) This remapping and restoring is done by the calling routine, so that the subroutine can be ignorant of this consideration. Of course, if the array is used within the subroutine in nearest-neighbor computations, then the subroutine itself will again remap the array on entry to provide the shadow edges, and will restore the array on exit, as described earlier.

It would certainly be better if an array which was used in a routine and in a called routine in nearest-neighbor computations could just be passed with no remapping, even without an explicit interface. However, this would require interprocedural information, which we do not have. In any case, all the required remapping discussed in this section is a purely local computation involving no interprocessor communication.

The local remapping described above is implemented as follows: To remap a nearest-neighbor array A so that it can be passed to a subroutine (i.e. to get rid of the shadow edges), a nest of local DO loops is created on each processor around the statement

$$A = A.$$

On the right-hand side, each $\hat{v}_j$ is given by

$$\hat{v}_j = \mathit{off}_j + \phi_j(I_j) - L_j.$$

On the left-hand side, each $\hat{v}_j$ is given by

$$\hat{v}_j = \phi_j(I_j) - L_j.$$

These loops have no strip mining obstacles in them.

The restoring remap is carried out similarly except that the right and left hand sides of the equation are interchanged, and all the loops are run backwards to avoid introducing strip mining obstacles.

While the remapping statements are set up by the DIVIDE phase, these addressing changes are implemented in STRIP.

15.4 Post processing

At the end of the STRIP phase, the `num_folds` attribute of every data cell that is distributed BLOCK and used in an array tagged as a nearest-neighbor array is replaced by its original value plus $2 * \mathit{off}_j$. This ensures that enough space will be allocated for the array plus the shadow edges when storage allocation is performed, and also ensures that the addressing of the array will be correct when the subscript expressions are linearized.

15.5 Nearest-neighbor computations on NUMA machines

This section could properly belong in Chapter 16, but we place it here because nearest-neighbor processing is so complex that it makes sense to keep all the variations of it in one place. We assume without additional comment all the discussion in Chapter 16, however. For this section, we assume that we are compiling a program for a NUMA machine.

Since nearest-neighbor handling, involving the introduction of shadow edges, is based on the distributed-memory model, we only classify an array as nearest-neighbor (via the NEAR_NEIGHBOR handy bit) if the array is specified as an element granularity array.

For each array A having shadow edges, we know that the array is distributed over separate memories, just as in a distributed-memory machine. We create modified versions of $\mathit{num\_folds}_j$ and *num_folds* for A as follows:

$$\mathit{num\_folds}^*_j = \mathit{num\_folds}_j + 2\beta_j$$
$$\mathit{num\_folds}^* = \prod_{j=1}^{n} \mathit{num\_folds}^*_j$$

where β_j is as usual the shadow width in dimension j, which may be 0.

DTR2CIR then describes the array as having bounds

$$0 : \mathit{num\_folds}^*_j - 1$$

in dimension j.

Now we show how to deal with subscripts of references to A. First, if A is not a local_stack array, we modify slightly the computation of *nf_round*, by defining it (by either Method 3 or Method 1 of Chapter 16, as appropriate) in terms of $\mathit{num\_folds}^*$, rather than *num_folds*.

If A is a local_stack array, the increment added to *_OtsNumaCurrentTos* and specified in item 7c on page 230 is similarly changed by replacing *num_folds* by $\mathit{num\_folds}^*$.

Now we discuss how to localize the subscripts of references to A. Each array reference has a Boolean `numa_local` attribute which is set as described on page 215 below. To handle nearest-neighbor arrays, we make one change to that processing:

Array references inside NUMA parallel loops that refer either to local array elements or to array elements in shadow edges are classified as `numa_local`. (That is, they might technically be remote references, but because the only remote elements referred to are actually in shadow edges, we classify them as `numa_local`. Shadow edges for these arrays are filled in in the usual manner before the entry to the loop.

Using this classification we can localize subscripts for nearest-neighbor arrays:

1. If the array reference is not `numa_local`, then the normal non-`numa_local` processing takes place—that is, the subscripts are localized using the data space of A. Since A is classified as a nearest-neighbor array (via the NEAR_NEIGHBOR handy bit), the appropriate offset is also added to each subscript.

2. If the array reference is `numa_local`, then we treat A almost exactly as we do in an HPF program. The subscripts are lowered exactly as described previously for HPF arrays. After this lowering, the numa adjustment term, which is either

   - $\bar{v} * \textit{nf\_round}$, as in expression 16.1 (page 194), or
   - $\bar{v} * \textit{@A\_extent} + \textit{@A\_base}$, as in 16.4 (page 228), if A is a local_stack variable,

   is added to the first subscript. (We don't have to use $k^p + k^r * P$ instead of $\bar{v}$ in these formulas, because we are assuming that there are no replication or completion cells.)

The message passing needed to fill in the shadow edges is replaced by loop nests containing ordinary assignment statements. This is done as follows:

Each SEND/RECEIVE pair is replaced by an assignment statement of the form

$$\text{received section of } A = \text{sent section of } A$$

We generate each such assignment as a "pull", rather than a "push". That is, the processor owning the received section of A is the processor that executes the statement. The sections are then computed as follows:

- The received section of A is computed exactly as specified by the equations for received data on page 181.
- The sent section of A is computed exactly as specified by the equations for sent data on page 180, with the following changes, which are needed in order that the expressions will evaluate to the correct values when evaluated by the receiving processor. (The original HPF version has them evaluated by the sending processor, for which the value of $\bar{v}$ is different.)
  - If the send is positive, then replace $\bar{v}_j$ in nLB_j and nUB_j by $\bar{v}_j - 1$.
  - If the send is negative, then replace $\bar{v}_j$ in nLB_j and nUB_j by $\bar{v}_j + 1$.

Following this, a parallel loop nest is generated to implement the assignment, and the numa adjustment term is added to the first subscript of each array reference. This term is just $\bar{v} * \textit{nf\_round}$ or $\bar{v} * \textit{@A\_extent} + \textit{@A\_base}$ as above, and is modified for sends (i.e., for the right-hand array references) in the same way as above; namely:

- If the send is positive, then replace $\bar{v}_j$ in $\bar{v}$ by $\bar{v}_j - 1$.
- If the send is negative, then replace $\bar{v}_j$ in $\bar{v}$ by $\bar{v}_j + 1$.

Chapter 16

Support for Shared-Memory NUMA Machines

16.1 Introduction

TRANSFORM has been adapted to support extensions to OpenMP Fortran [4] that allow data mapping when a NUMA machine is targeted. This design is based on the following assumptions:

- The machine consists of a number of networked smaller machines. Each smaller machine, which is referred to in this design as an "hpf processor", contains one memory and a number of physical processors. These are examples of what the operating system calls a *rad*, or "Resource Allocation Domain". We generally do not use the term *rad* in this discussion (except in some of the pseudocode below), however, because a rad can be a more general collection of processors and/or memories, not necessarily containing even one of each. (On the Wildfire machine, the hpf processors are referred to as "quads", since each hpf processor contains four physical processors.)

  An hpf processor is also called a *memory*.

- Each available physical processor generally corresponds to an OMP thread. The number of threads per hpf processor is denoted by *threads_per_memory*.

- The machine as a whole has a single address space. However, access by a processor to local memory (i.e., memory on the same hpf processor) is faster than to non-local memory. For this reason, mapping large arrays over the hpf processors can be important in generating efficient code. This data mapping is expressed in the source program by HPF mapping directives.

- The TRANSFORM phases of the compiler generate code that is in principle parametrized by calls to *my_processor*`()`, where *my_processor*`()` returns the number of the hpf processor on which it is called.

- The reason for the phrase "in principle" in the previous assumption is that we assume that there is a well-defined map from thread number to hpf processor number. In fact, we will assume that the threads are assigned to the hpf processors in a block fashion. For instance, if

$threads\_per\_memory = 4$, then threads 0-3 are assigned to hpf processor 0, threads 4-7 to hpf processor 1, and so on. In general,

$$my\_processor() = \left\lfloor \frac{my\_thread()}{threads\_per\_memory} \right\rfloor$$

16.2 Language issues

The language we support is OpenMP Fortran with the HPF mapping directives added. For convenience, in this chapter, we will call this language simply "NUMA". The semantics of the HPF mapping directives is, unless specified otherwise, that of the HPF language.

There are, however, a few instances in which the mapping directives have a somewhat different semantics:

- SEQUENCE is compatible with mapped objects in NUMA. SEQUENCE causes the compiler to interpret the mapping of the objects with "page granularity" (see below).
- In HPF, SEQUENCE and NOSEQUENCE may be applied only to data objects and common blocks. In NUMA, these directives may also be applied to fields of derived types.
- In HPF, mapped arrays in common must be declared identically (except for trivial name changes) in each place the common block is declared. This continues to be the case for "element granularity" common blocks (see below).

  However for page granularity common blocks (these are common blocks declared with the default SEQUENCE directive), this restriction is relaxed. Of course, if two inconsistent mappings are specified for an array in such a common block, the compiler will simply pick one of them and ignore the other. This will not lead to incorrect code, although in general it will lead to inefficient code.
- In NUMA, mapping may be applied to an assumed-size object, provided

  1. it has the SEQUENCE attribute, and
  2. the last dimension is mapped serially.

  However, such an object may not be replicated or partially replicated.

  The reason for this is simply that the compiler does not know the size of an assumed-size array. (It does not know the size of an assumed-shape array, either, but it can create an expression for this size from dope vector information.) In our implementation, the replicated copies of the array are stored consecutively in memory. Without knowing the size of the array, the compiler has no way of knowing where each successive replicated copy of the array starts in memory.

  Note that since the last dimension of a mapped assumed-size object is mapped serially, the compiler can use the mapping information to infer an iteration space when it needs to.
- In HPF there is no effective distinction between descriptive and prescriptive mappings on dummy arguments. In both HPF and NUMA, the callee may assume that a descriptively mapped dummy is mapped as the directive specifies. In HPF it is the responsibility of the caller to ensure that this is so, remapping the actual if necessary. However, in NUMA, the

caller is under no obligation to enforce this. In other words, in NUMA, a descriptive directive is an assertion by the programmer that the actual already conforms to the indicated mapping. If it does not, the program will still be semantically correct (since the machine is a shared-memory machine), but performance will no doubt be degraded.

As a consequence, since STRIP cannot be sure (in NUMA) that a descriptively mapped dummy actually is mapped as specified, it cannot know for certain when an array or array element is local (really `numa_local`; see page 215 below). Therefore, iteration space cannot be used in localizing subscripts for such an array reference—only data space may be used.

(As a matter of historical interest, this was the original intended interpretation of descriptive mappings in HPF. Of course, in a distributed-memory system, a violation of the assertion would in general cause the program to be non-conforming.)

16.3 Interpretation of the mapping directives

The HPF mapping directives are also often referred to in the NUMA context as "data layout directives".

There are two ways the compiler can treat these directives:

- Preserve the HPF interpretation of the mapping directives. This breaks sequence and storage association, but gives a mapping of array elements to hpf processors that exactly honors the mapping directives specified by the programmer.

  We call this method *element granularity* .

- Preserve sequence and storage association. This does not exactly honor the HPF mapping specified by the programmer, but it does make it possible to compile code whose correctness depends on sequence association.

  We call this method *page granularity*.

The default mapping for the NUMA compiler is serial, restricted to hpf processor 0. This is in contrast to the default mapping for HPF, which is also serial but is replicated over all hpf processors.

The default granularity for the NUMA compiler is page granularity. For ease of explanation, however, we discuss element granularity first.

The analysis that TRANSFORM performs in the HPF compiler is used in three ways in generating code for such a NUMA machine:

1. We determine, based on the HPF mapping directives, how the data is to be allocated to virtual pages in memory, and how those pages are allocated to the physical memory of specific hpf processors.

2. In some cases, we adjust the subscripts of mapped arrays in the program to support the allocation in item 1.

3. We distribute iterations of parallel loops over hpf processors for a NUMA machine exactly as we distribute them over processors for distributed memory. We do this by parametrizing the loop bounds and increment by calls to *my_processor*`()` so they are different on each hpf processor. If the programmer has done a good job of inserting mapping directives, the iterations that execute

on an hpf processor will mainly or entirely address data whose pages have been allocated to that hpf processor.

We start out by describing data layout support. Later, in Section 16.7, we describe how loop iterations are distributed.

We use the standard notation used everywhere else in this report. To make this chapter somewhat self-contained, we repeat some of the main notation and concepts used:

Say we are given an n-dimensional array reference $A(s_1, s_2, \ldots, s_n)$ (s stands for "subscript") that is mapped onto a processors arrangement $P(sl_1, sl_2, \ldots, sl_n)$ of the same rank. (sl stands for "strip length", an archaic term in this context.) Further, suppose that each dimension of the array A is laid out along the corresponding dimension of P.

- num_folds_i denotes the local extent of the i^{th} dimension of the array on each hpf processor. Thus, the total local extent of the array on each hpf processor is given by

$$num\_folds = \prod_{i=1}^{n} num\_folds_i$$

- $\hat{v}_i$ denotes the contribution to the local address due to the i^{th} subscript s_i. The entire local address (i.e., the offset from the base address of the array) is given by

$$\hat{v} = \sum_{i=1}^{n} \left(\hat{v}_i * \prod_{j=1}^{i-1} num\_folds_j \right)$$

- $\bar{v}_i$ denotes the i^{th} coordinate of the abstract processor holding this array reference. Thus, the abstract processor is

$$\langle \bar{v}_1, \bar{v}_2, \ldots, \bar{v}_n \rangle$$

  and the actual hpf processor number will typically be given by

$$\bar{v} = \sum_{i=1}^{n} \left(\bar{v}_i * \prod_{j=1}^{i-1} sl_j \right)$$

Previous chapters in this report contain formulas for computing these quantities.

We will use a running example: for simplicity, we assume that our machine consists of

- 4 hpf processors, each with its own linear memory.
- 4 threads per hpf processor; i.e., $threads\_per\_memory = 4$.
- a page size that holds 4 integer values.

We will consider how to lay out the array A, declared as follows:

integer $A(5,\ 5)$
!hpf$ distribute A(**block**, **block**)

We will consider this array as being distributed over an abstract 2×2 processor arrangement.

16.4 Element granularity: mappings with the HPF interpretation

Figure 16.1 shows how array A would be distributed over the four hpf processors, using the HPF interpretation.

proc 0	proc 1	proc 2	proc 3
(1,1)	(4,1)	(1,4)	(4,4)
(2,1)	(5,1)	(2,4)	(5,4)
(3,1)	—	(3,4)	—
(1,2)	(4,2)	(1,5)	(4,5)
(2,2)	(5,2)	(2,5)	(5,5)
(3,2)	—	(3,5)	—
(1,3)	(4,3)	—	—
(2,3)	(5,3)	—	—
(3,3)	—	—	—

Figure 16.1: The array $A(5,\ 5)$ distributed (**block**, **block**) over 4 hpf processors, using the HPF interpretation

For this example, we have

$$sl_1 = 2 \qquad num\_folds_1 = 3$$
$$sl_2 = 2 \qquad num\_folds_2 = 3$$

Here are the values of $\hat{v}_i$ and $\bar{v}_i$, together with the values of $\hat{v}$ and $\bar{v}$ as computed by the formulas above, for some representative array elements:

	$\hat{v}_1$	$\hat{v}_2$	$\bar{v}_1$	$\bar{v}_2$	$\hat{v}$	$\bar{v}$
$A(1,\ 2)$	0	1	0	0	3	0
$A(3,\ 5)$	2	1	0	1	5	2
$A(5,\ 4)$	1	0	1	1	1	3

If we make the pages in each hpf processor's memory visible, we get the picture in Figure 16.2.

The next three subsections describe three methods of achieving the HPF interpretation of the mapping directives. We actually generally use Method 1; we use Method 3 when we do not know the element size of the array at compile time. Method 2 is presented to show an alternative approach that was previously considered, and to aid in understanding the issues involved.

proc 0	proc 1	proc 2	proc 3
(1,1)	(4,1)	(1,4)	(4,4)
(2,1)	(5,1)	(2,4)	(5,4)
(3,1)	—	(3,4)	—
(1,2)	(4,2)	(1,5)	(4,5)
(2,2)	(5,2)	(2,5)	(5,5)
(3,2)	—	(3,5)	—
(1,3)	(4,3)	—	—
(2,3)	(5,3)	—	—
(3,3)	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—

Figure 16.2: The array $A(5,\ 5)$ distributed (**block**, **block**) over 4 hpf processors, using the HPF interpretation, with page boundaries indicated.

In reading this, note that Method 1 gives us exactly the HPF memory layout shown above, while Method 2 gives us a memory layout that is not quite the same, but in which each element is still on the same hpf processor, as the HPF interpretation specifies. The data layout in Method 3 is similar to that in Method 1, differing only in what might be thought of as a different initial offset on the memory of each hpf processor.

16.4.1 Method 1: fill out *num_folds*

We turn the pair $\langle \hat{v}, \bar{v} \rangle$ into a linear address. We do this by padding the physical memory allocated to each local section of the array up to a common multiple of the page size and the array element size. That is, letting *esize* denote the size in bytes of an element, and letting *page_size* denote the size in bytes of a physical memory page, we need to increase *num_folds* if necessary to a quantity *nf_round* so that $nf\_round * esize$ is a multiple of *page_size*. We do this as follows: first define

$$aux\_page\_size = \frac{page\_size}{\mathrm{gcf}(page\_size,\, esize)}$$

Then in terms of this quantity, define

$$\begin{aligned} nf\_round &= \left\lceil \frac{num\_folds}{aux\_page\_size} \right\rceil * aux\_page\_size \\ NF\_round &= nf\_round * esize \end{aligned}$$

so *nf_round* is the size (in units of array elements) just referred to, and *NF_round* is the physical size in bytes that this corresponds to. (*NF_round* is thus a multiple of *page_size*.) Now we manage to have the array reference $A(s_1, s_2, \ldots, s_n)$ generate the address (that is, the offset from the base address of A) given by

$$\hat{v} + \bar{v} * \mathit{nf\_round}$$

(in units of the array element size, as usual). We can do this by replacing $A(s_1, s_2, \ldots, s_n)$ by

$$A(\hat{v}_1 + \bar{v} * \mathit{nf\_round}, \hat{v}_2, \ldots, \hat{v}_n) \tag{16.1}$$

The reason this works is that when the subscripts of A are linearized, the first subscript is left alone. Thus, in effect we allocate the pages corresponding to the physical offsets $0 : \mathit{NF\_round} - 1$ to processor 0, the pages corresponding to the physical offsets $\mathit{NF\_round} : 2 * \mathit{NF\_round} - 1$ to processor 1, and so on.

We refer to the term $\bar{v} * \mathit{nf\_round}$ as the *numa adjustment* term. (In the code in `stp_loc.c` it is the variable `numa_adjust`.)

16.4.2 Method 2: fill out the first dimension

There is a second way of achieving the HPF interpretation for the data mapping. This way has the advantage that the subscripts are modified less drastically than in method 1, but has the disadvantage that additional gaps are introduced in the sequence of array elements on each hpf processor.

We round up the first local dimension to a multiple of the page size. Figure 16.3 shows how this works, with the pages in the memories of these four hpf processors made visible.

To do this, we change the declaration of the array A so that it is $A(8, 5)$. The new extent 8 of the first dimension is produced as follows: $\mathit{num\_folds}_1$ is 3. We round this up to a multiple of the page size—in this case, to 4—and then multiply by sl_1 (which is 2), giving 8. In general, all the dimensions except the right-most one have to be adjusted in this manner.

Then we adjust the addressing so that in this case, with a (**block**, **block**) distribution onto a 2×2 processor array, $A(i, j)$ is replaced by $A(i + \lfloor i/3 \rfloor), j)$. If the statement is known to be local (so that each element is accessed only by the hpf processor that owns it), then this even becomes simpler: $A(i, j)$ is replaced by $A(i + (\text{myprocessor mod } 2), j)$, and the added term $(\text{myprocessor mod } 2)$ is an invariant of any loops containing the array A.

Precisely, it works like this:

- With the same notation as in Method 1, redefine $\mathit{num\_folds}_1$ by

$$\mathit{num\_folds}_1^* = \left\lceil \frac{\mathit{num\_folds}_1}{\mathit{aux\_page\_size}} \right\rceil * \mathit{aux\_page\_size} * \mathit{esize}$$

  This is the smallest number greater than or equal to $\mathit{num\_folds}_1$ such that $\mathit{num\_folds}_1^* * \mathit{esize}$ is a multiple of *page_size*.

- Then alter the extent of A in dimension 1 to be consistent with this. In general, how this is done depends on the alignment of A. In the common case when the alignment is trivial, the new extent of A in dimension 1 will be $\mathit{num\_folds}_1^* * sl_1$.

proc 0	proc 1	proc 2	proc 3
(1,1)	(4,1)	(1,4)	(4,4)
(2,1)	(5,1)	(2,4)	(5,4)
(3,1)	—	(3,4)	—
—	—	—	—
(1,2)	(4,2)	(1,5)	(4,5)
(2,2)	(5,2)	(2,5)	(5,5)
(3,2)	—	(3,5)	—
—	—	—	—
(1,3)	(4,3)	—	—
(2,3)	(5,3)	—	—
(3,3)	—	—	—
—	—	—	—

Figure 16.3: (Method 2) The array A(5, 5) distributed (**block**, **block**) over 4 hpf processors, using the HPF interpretation, with the first array dimension filled out and with the pages on each hpf processor shown

- Then define the *excess* added in dimension 1 by

  $$e_1 = num\_folds_1^* - num\_folds_1$$

  In the example we have been using, $e_1 = 1$.

- The array reference $A(s_1,\ s_2,\ \ldots,\ s_n)$ is then replaced by

  $$A(s_1 + \bar{v}_1 e_1,\ s_2,\ \ldots,\ s_n)$$

  Note that only the first subscript is changed. In the example we have been using, e_1 is 1, and $A(4,\ 1)$ and $A(5,\ 1)$ get replaced by $A(5,\ 1)$ and $A(6,\ 1)$, respectively.

16.4.3 Comparison of these two methods, and a third method (Method 3)

Either of these two methods of mapping data to conform with the HPF interpretation has the disadvantage that we have to adjust the array bounds and subscripts. Enlarging the array in either case might cause problems with memory allocation. In addition to this, each method has its own peculiarities:

Method 1 has the disadvantage that comparatively more complex subscript expressions are generated than in Method 2. On the other hand, these expressions are no more complicated than the compiler generates for distributed-memory code, and this has not been a problem in the efficiency of such code.

Method 2 has the disadvantage that it allocates more unused memory than Method 1 does. In addition, it does this in such a way that there may be many physical pages with no array elements allocated to their higher addresses. This could well cause cache problems.

Method 2 also has the disadvantage that the assignment of physical pages to hpf processors is much harder to describe. The assignment in Method 1 is quite simple (see section 16.6 on page placement).

When we are sure that the array element size *esize* divides the page size, we use Method 1, for the above reasons.

When *esize* does not divide the page size, however, the formulas for both Method 1 and Method 2 can lead to a gross overallocation of wasted space. In such a case, $\mathrm{gcf}(page\_size, esize)$ can be 1, and NF_round will be a multiple of $page\_size * esize$, which in general is much too big (e.g., by a factor of *esize*). That is, this method can in the worst case waste $num\_folds * page\_size * (esize - 1)$ bytes on each hpf processor; equivalently, it can waste $num\_folds * (esize - 1)$ pages on each hpf processor. The same problem occurs in Method 2. For this reason, we cannot use either Method 1 or Method 2 when we cannot prove that *esize* divides *page_size*.

Because we are addressing shared memory, however, we can modify Method 1 to give acceptable behavior, as follows: in both Method 1 and Method 2 we implicitly assumed that each local section started on a page boundary. This is actually not necessary[1]. We can modify Method 1 by defining

$$nf\_round = num\_folds + \left\lceil \frac{page\_size}{esize} \right\rceil$$

This ensures that each local section starts on a page that is different from the page on which the previous local section ended. The starting position, however, may not be at the beginning of the page.

This method, which we call Method 3, wastes at most 2 pages on each hpf processor. This is the method we use when are not certain that the array element size divides the page size.

Note that when we use Method 1 (so *esize* divides *page_size*), the formula for *nf_round* becomes

$$nf\_round = \left\lceil \frac{num\_folds}{\frac{page\_size}{esize}} \right\rceil * \frac{page\_size}{esize}$$

Since

$$\frac{num\_folds}{page\_size/esize} \le \left\lceil \frac{num\_folds}{page\_size/esize} \right\rceil < \frac{num\_folds}{page\_size/esize} + 1$$

we see that the value of *nf_round* computed when we use Method 1 is always less than

$$num\_folds + \frac{page\_size}{esize}$$

which would be the value if we used Method 3. Therefore it is an advantage to use Method 1 when we can.

[1] We owe this observation to Bill Noyce.

16.4.4 Completion and replicated cells

We now indicate how to extend Method 3 to handle completion cells (this pertains to data that is not mapped to all of the hpf processors) and also partially replicated data.

Each cell in the data space of an array is classified as either primary, completion, or replicated. Let us say there are π primary cells, κ completion cells, and ρ replicated cells for a particular array A. The cell places corresponding to these three kinds of cells will be denoted as follows:

$$\begin{aligned}
\text{primary:} &\quad \left\{\langle \mathit{strip_length}^p_j, \mathit{multiplier}^p_j\rangle : 1 \le j \le \pi\right\} \\
\text{completion:} &\quad \left\{\langle \mathit{strip_length}^c_j, \mathit{multiplier}^c_j\rangle : 1 \le j \le \kappa\right\} \\
\text{replicated:} &\quad \left\{\langle \mathit{strip_length}^r_j, \mathit{multiplier}^r_j\rangle : 1 \le j \le \rho\right\}
\end{aligned}$$

Note that $\mathit{multiplier}^p_j$ does *not* generally equal $\prod_{i=1}^{j-1} \mathit{strip\_length}^p_j$ (and similarly for the completion and replicated parameters).

For convenience, however, let us define the quantities

$$\begin{aligned}
\mu^p_j &= \prod_{i=1}^{j-1} \mathit{strip_length}^p_j \qquad (1 \le j \le \pi) \\
\mu^c_j &= \prod_{i=1}^{j-1} \mathit{strip_length}^c_j \qquad (1 \le j \le \kappa) \\
\mu^r_j &= \prod_{i=1}^{j-1} \mathit{strip_length}^r_j \qquad (1 \le j \le \rho)
\end{aligned}$$

Note that $\mu^p_1 = \mu^c_1 = \mu^r_1 = 1$.

Now each hpf processor number $\bar{v}$ has a representation $\langle \bar{v}_1, \bar{v}_2, \ldots, \bar{v}_n\rangle$, where each component $\bar{v}_j$ is a component in either a primary, completion, or replicated cell. Let us denote these components by

$$\begin{aligned}
&\bar{v}^p_i \qquad (1 \le i \le \pi) \\
&\bar{v}^c_i \qquad (1 \le i \le \kappa) \\
&\bar{v}^r_i \qquad (1 \le i \le \rho)
\end{aligned}$$

respectively—each $\bar{v}_j$ is one of these. We can imagine the entire vector $\bar{v}$ being projected onto the three subspaces spanned by the primary cells, the completion cells, and the replicated cells. These three projections have coordinates

$$\begin{aligned}
&\langle \bar{v}^p_1, \bar{v}^p_2, \ldots, \bar{v}^p_\pi\rangle \\
&\langle \bar{v}^c_1, \bar{v}^c_2, \ldots, \bar{v}^c_\kappa\rangle \\
&\langle \bar{v}^r_1, \bar{v}^r_2, \ldots, \bar{v}^r_\rho\rangle
\end{aligned}$$

In terms of these coordinates, we can define three auxiliary vectors

$$\bar{v}^p = \sum_{j=1}^{\pi} \bar{v}^p_j * \mathit{multiplier}^p_j$$

$$\bar{v}^c = \sum_{j=1}^{\kappa} \bar{v}^c_j * \mathit{multiplier}^c_j$$

$$\bar{v}^r = \sum_{j=1}^{\rho} \bar{v}^r_j * \mathit{multiplier}^r_j$$

We then have $\bar{v} = \bar{v}^p + \bar{v}^c + \bar{v}^r$. Note that $\bar{v}^c$ is constant, since each $\bar{v}^c_j$ is itself constant.

In particular, there are $\prod_{j=1}^{\pi} \mathit{strip\_length}^p_j$ values for the π-tuple $\langle \bar{v}^p_1, \bar{v}^p_2, \dots, \bar{v}^p_\pi \rangle$ of primary coordinates. Let us denote the value of this product by P. We call P the *primary multiplicity* There are thus P values taken on by $\bar{v}^p$. However, these values are scattered—they do not in general form a contiguous interval of integers starting at 1.

We would like to index these possible values as such a contiguous range of integers. We do this as follows: The π-tuple

$$\langle \bar{v}^p_1, \bar{v}^p_2, \dots, \bar{v}^p_\pi \rangle$$

is assigned the index

$$k^p = \sum_{j=1}^{\pi} \bar{v}^p_j * \mu^p_j$$

k^p then takes on all integer values between 0 and $P - 1$. Further, the π-tuple can be retrieved from k^p by the formulas

$$\bar{v}^p_j = \left\lfloor \frac{k^p}{\mu^p_j} \right\rfloor \bmod \mathit{strip\_length}^p_j$$

The factor $\bar{v}$ in formula (16.1) then is replaced by k^p. Note that

- k^p can always be computed from the set $\{s_i\}$ of subscripts.
- If the array reference is known to be local, k^p can also be computed from $\bar{v} = \mathit{my\_processor}()$, which is generally a simpler computation.

No special handling is needed for completion cells—the completion cell guard that is ordinarily generated continues to work in the same way.

As far as replicated cells go, we can create k^r and R in the same way as we created k^p and P for the primary cells (although of course k^r cannot be computed from the set of subscripts).

A partially replicated data object can be thought of as R separate copies of that data object. (For this reason, we call R the *replication multiplicity* of the array A.) Therefore, we will allocate R times the number of pages that would be allocated in the absence of replication. This is calculated as follows:

Without accounting for replication, the memory (in units of array element size) allocated to the array A is just $\mathit{nf\_round} * P$. Thus, allowing for replication, we allocate R times this much memory, or $\mathit{nf\_round} * P * R$.

Now when addressing an element of a partially replicated object, we modify the subscripts as follows: Let $\bar{v}$ denote as usual the processor storing to or fetching from its local element. The projection of $\bar{v}$ on the replicated subspace of the data space is just the ρ-tuple $\langle \bar{v}_1^r, \bar{v}_2^r, \ldots, \bar{v}_\rho^r \rangle$, as mentioned above. We compute k^r from this ρ-tuple as before, and add $k^r * \mathit{nf\_round} * P$ to the address. (This is all measured in units of the size of 1 array element). The way we do this is simply to add that quantity to the first subscript. Thus, formula (16.1) is finally transformed as follows: the array reference $A(s_1, s_2, \ldots, s_n)$ is rewritten as

$$A\big(\hat{v}_1 + (k^p + k^r * P) * \mathit{nf\_round}, \hat{v}_2, \ldots, \hat{v}_n\big) \tag{16.2}$$

where k^p and k^r are both computed as described above. Here the numa adjustment term is $(k^p + k^r * P) * \mathit{nf\_round}$.

If the reference to A is being fetched, there are two possibilities:

1. The reference to A is local. In this case, the same analysis applies; each thread will fetch from elements that it owns as above.
2. The reference to A is not local. In this case, k^p has to be computed from the set $\{s_i\}$ of subscripts, as mentioned above. k^r could continue to be computed from $\bar{v} = \mathit{my\_processor}()$, but since the reference is a remote one in any case, any k^r would work just as well. (That is, any k^r yields a correct address; the only question is to find the best one. In a non-local reference there is most likely not a best one.) Therefore, for non-local references, we will simply set $k^r = 0$ in the above formula. This at least makes the addressing as simple as possible.

To recapitulate: if we are generating code for a fetch of an array reference, there are two possibilities:

- The reference is local. This is always true for references that are being stored to. In this case, k^p and k^r are both computed from $\bar{v}$.
- The reference is remote. In this case, k^p is computed from the set $\{s_i\}$ of subscripts, and k^r is set to 0.

Stores to array references are somewhat more complex. The complete story is given below in Section 16.8.

16.4.5 Storage allocated to element granularity arrays

Based on the formulas given above, the storage allocated for an element granularity array is

$$\mathit{nf\_round} * P * R$$

where as above P is the primary multiplicity and R is the replication multiplicity. *nf_round* is computed as described above in Section 16.4.3.

16.5 Page granularity: for sequence and storage association

If we want to honor sequence and storage association, then

- The data allocation function of the array in each dimension must have `alloc_offset` 1.
- We cannot use the data (or iteration) distribution function to compute subscripts.

The effect of this is that when the subscripts of the array are localized in STRIP, each subscript is simply fed through the data allocation function, which at most offsets that subscript by some constant value. We can, in fact, arrange matters so that this makes the first element of the array map to virtual processor 0; this is certainly the default in any case.

Note that it is still true for page granularity arrays that the iteration space must conform with the data space—this is because strip loops are always computed from iteration space, while data space is used for addressing, so clearly we can't change one independently of the other.

To handle replication of page granularity arrays, we allocate R copies of each page of the array, where R is the replication multiplicity, as defined above, and we add $k^r * nf\_round$ to the first subscript of each array reference. (Note that this means that the first elements of the replicated copies of the array may well not start on page boundaries. This is something we just live with.)

Beyond that, the only thing we are allowed to do is assign pages to hpf processors, and we want to do that in a manner that gives a result that is reasonably close to what we would have got with the HPF mapping directive interpretation. We do this in the following way:

1. Conceptually distribute the array as in Figure 16.1. This assigns each array element to a hpf processor.
2. List the array elements in array element order and break them up into virtual pages in that order. This is the way the array will actually be assigned to pages in memory. Figure 16.4 shows how this is done for the array A in our example.
3. Assign each page from step 2 to the hpf processor determined by step 1 to own the first array element on that page.

The result of this is the array layout shown in Figure 16.5. (Of course, this picture does not reflect any completion or replicated cells.)

Mapping data to honor sequence and storage association has the advantage that modifications to the array bounds and array subscripts are minimal. It has the disadvantage, however, that individual array elements may not be mapped to the expected hpf processor. For instance, in the case we have considered here, 12 out of the 25 array elements are mapped to an hpf processor that differs from what the programmer would have expected based on a strict adherence to a (**block**, **block**) distribution (as in Figure 16.1). The problem of elements not being mapped to the expected hpf processor will likely lead to inferior run-time performance.

In this case, the compiler could have been more clever in assigning pages to hpf processors. Perhaps a better method for assigning virtual pages to hpf processors would be to use a middle element of that page to decide which hpf processor that page will be allocated to. For instance, if either the second or third array element on a page in this example were used to determine the mapping of pages to hpf processors, 9 array elements (rather than 12) would be mapped to the wrong page. This is an improvement, but a significant number of array elements are still on the wrong hpf processors. (If we used the fourth element on each page, 14 array elements would be mapped incorrectly.)

Another possibility is to map each page to the hpf processor that wants to own the majority of its array elements. In this example, this would still leave 9 array elements mapped incorrectly. Also,

(1,1)
(2,1)
(3,1)
(4,1)
(5,1)
(1,2)
(2,2)
(3,2)
(4,2)
(5,2)
(1,3)
(2,3)
(3,3)
(4,3)
(5,3)
(1,4)
(2,4)
(3,4)
(4,4)
(5,4)
(1,5)
(2,5)
(3,5)
(4,5)
(5,5)
—
—
—

Figure 16.4: The array $A(5,\ 5)$, in array element order, showing how array elements get allocated to pages when sequence and storage association are honored.

proc 0	proc 1	proc 2	proc 3
(1,1)	(5,1)	(2,4)	(5,5)
(2,1)	**(1,2)**	(3,4)	—
(3,1)	**(2,2)**	**(4,4)**	—
(4,1)	**(3,2)**	**(5,4)**	—
(3,3)	(4,2)	(1,5)	—
(4,3)	(5,2)	(2,5)	—
(5,3)	**(1,3)**	(3,5)	—
(1,4)	**(2,3)**	**(4,5)**	—

Figure 16.5: The array $A(5, 5)$ distributed (**block**, **block**) over 4 hpf processors, but now with sequence and storage association honored, showing how pages are allocated to hpf processors. The 12 shaded array elements are assigned to hpf processors that are not those specified by the HPF directive.

in general it is not known at compile time exactly which array elements will be assigned to each hpf processor by the HPF mapping directives, and therefore there would be a run-time cost associated with this. This might be acceptable, however.

This problem does become less significant as the array size increases. What we are really seeing here is an edge effect. The fraction of mis-mapped elements decreases as the array becomes larger, so for very large arrays, the layout would be a reasonable approximation to the true (**block**, **block**) mapping. On the other hand, the page size is already quite large, and if an array has 2 or three distributed dimensions, it would have to be truly huge to make these edge effects negligible.

16.5.1 Storage allocated to page granularity arrays

The amount of storage that has to be allocated to a page granularity array is

$$nf\_round * R$$

where as before, R is the replication multiplicity of the array, and now nf_round is computed as follows:

$$nf\_round = \prod_{j=1}^{n} extent_j + \left\lceil \frac{page\_size}{esize} \right\rceil$$

where $extent_j$ is just the storage needed in dimension j, i.e.,

$$extent_j = f_j(DUB_j) - f_j(DLB_j) + 1$$

where as usual DUB_j and DLB_j are the declared bounds in dimension j and f_j is the data allocation function for dimension j.

In the usual case when $R = 1$, the last term on the right-hand side in the expression for *nf_round* can be eliminated.

Further, when $R > 1$ and we know that *esize* divides *page_size*, we can reduce the storage allocated somewhat, just as we did above in Section 16.4.3 for element granularity arrays. In such a case we have

$$nf\_round = \left\lceil \frac{\prod_{j=1}^{n} extent_j}{\frac{page\_size}{esize}} \right\rceil * \frac{page\_size}{esize}$$

16.6 Page placement

The previous discussion has shown how we map array elements to pages and pages to hpf processors. The compiler must ensure that a page really does wind up in the memory of its associated hpf processor. Let us use *number_of_processors* to refer to the number of hpf processors in the machine. We have to tell the operating system where to put each page. We do this by specifying one offset from the base address of each array for each page allocated to that array, and for each such offset, we specify the associated hpf processor that the page in question must be assigned to.

16.6.1 Element granularity

The algorithm for page placement in the case of element granularity (remember that we are using Method 1) can be specified succinctly. In the following formula R is the multiplicity of replication, as defined above, and k is derived from n

for $n = 0$ **to** $number\_of\_processors - 1$ **do**
  Compute k^p and k^r from n (which is $\bar{v}$).
  if $\bar{v}$ satisfies the completion constraints **then**
    Addresses in the range
      low: $base\_address + (k^p + k^r * P) * NF\_round$
      high: low $+ (num\_folds - 1) * esize$
      stride: $page\_size$
    go on hpf processor n.
  end if
end for

Actually, we implement this in a manner that is only slightly different:

Compute $\bar{v}^c$.
for $k^p = 0$ **to** $P - 1$ **do**
  for $k^r = 0$ **to** $R - 1$ **do**
    Compute $\bar{v}^p$ from k^p.
    Compute $\bar{v}^r$ from k^r.
    Compute $\mathit{rad\_number} = (\bar{v}^p + \bar{v}^r + \bar{v}^c) \bmod \mathit{number\_of\_rads}$
    Addresses in the range
        low: $\mathit{base\_address} + (k^p + k^r * P) * \mathit{NF\_round}$
        high: $\text{low} + (\mathit{num\_folds} - 1) * \mathit{esize}$
        stride: $\mathit{page\_size}$
      go on hpf processor $\mathit{rad\_number}$.
  end for
end for

16.6.2 Page granularity

Say our array is declared as having declared lower and upper bounds DLB_j and DUB_j in dimension j. Let us set

$$e_j = DUB_j - DLB_j + 1$$

so e_j is the extent in dimension j.

The array element $A(s_1, \ldots, s_n)$ then has index (or "position")

$$p = \sum_{i=1}^{n} (s_i - DLB_i) \prod_{j=1}^{i-1} e_j$$

in (0-based) array element order. That is, $p = 0$ corresponds to the first element of A, $p = 1$ to the second, and so on. We can solve this to get each of the subscripts s_i in terms of p as follows:

$$s_i = DLB_i + \left\lfloor \frac{p \bmod \prod_{j=1}^{i} e_j}{\prod_{j=1}^{i-1} e_j} \right\rfloor \tag{16.3}$$

We create a table of pairs

$$\langle \text{offset}, \text{HPF processor number} \rangle$$

that contains exactly one offset from the base address of the array for each physical page allocated to the array. Since with page granularity we are not allowed to adjust the size of the array, the first element of the array must occur at the base address of the array. We further make the assumption that the base address of each array is at a page boundary.

The table has to be big enough to hold

$$R * \left\lceil \frac{\mathit{esize} * |\mathit{alloc\_stride}_1| * \prod_{j=1}^{n} e_j}{\mathit{page\_size}} \right\rceil$$

pairs as above.

All the following computations assume that $esize * alloc\_stride_1 \leq page\_size$. Equivalently, we are assuming that listing the addresses of the elements of the array in array element order does not cause us to skip any physical pages.

In the following, we use the symbol ω to denote a physical offset from the base address.

We give two versions of the table:

1. In the first version, we simply pick one element on each page, without being concerned about the position of the element on that page. We give two algorithms for this method, the second one being a simplified form of the first that can be used in most cases.

2. In the second version, we pick our elements so that they are all in the same relative position on the page. This results in a more complicated algorithm. Again, we give two algorithms, the second being a simplified form that can be used in most cases.

We use the following convention in the algorithms below: Assignment statements in which the assignment operator is written as "$\leftarrow$" denote assignments in the generated code. Assignment statements in which the assignment operator is written as "=" denote the computation of expressions in the generated code that are used subsequently but are not actually assigned to variables.

First version: relatively simple algorithms

In this version, we pick one element of the array on each page. The elements chosen are roughly "one page apart", but the positions on the pages shift as we pass from one element to the next:

Compute $\bar{v}^c$. -- This is a constant computed from the completion cells.
Compute R. -- This is a constant, computed from the replicated cells.
for $p = 0$ **to** $\prod_{i=1}^{n} e_i - 1$ **step**/ $\left\lfloor \dfrac{page\_size}{esize * |alloc\_stride_1|} \right\rfloor$
    $\omega = p * esize * alloc\_stride_1$
    $abs\_\omega \leftarrow |\omega|$
    if $\left\lfloor \dfrac{abs\_\omega}{page\_size} \right\rfloor$ changes **then**
        $address = base\_address + \omega$
        Compute $\{s_i\}$ from p by (16.3)
        Compute $\bar{v}^p$ from $\{s_i\}$
        for $k^r = 0$ **to** $R - 1$
            Compute $\bar{v}^r$ from k^r
            Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
            Enter $\langle address + k^r * nf\_round * esize, rad\_number\rangle$ in the table.
        end for
    end if
end for
-- Enter the last element of the array, if necessary:
$\omega = \left(\prod_{i=1}^{n} e_i - 1\right) * esize * alloc\_stride_1$
$abs\_\omega \leftarrow |\omega|$
if $\left\lfloor \dfrac{abs\_\omega}{page\_size} \right\rfloor$ changes **then**
    $address = base\_address + \omega$
    Compute $\bar{v}^p$ from $\{DUB_i\}$
    for $k^r = 0$ **to** $R - 1$
        Compute $\bar{v}^r$ from k^r
        Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
        Enter $\langle address + k^r * nf\_round * esize, rad\_number\rangle$ in the table.
    end for
end if

In the case that $esize * alloc\_stride_1$ divides $page\_size$, which we expect to happen frequently, this algorithm can be simplified:

Compute $\bar{v}^c$. -- This is a constant computed from the completion cells.
Compute R. -- This is a constant, computed from the replicated cells.
for $p = 0$ **to** $\prod_{i=1}^{n} e_i - 1$ **step**/ $\dfrac{page\_size}{esize * |alloc\_stride_1|}$
    $\omega = p * esize * alloc\_stride_1$
    $abs\_\omega \leftarrow |\omega|$
    $address = base\_address + \omega$
    Compute $\{s_i\}$ from p by (16.3)
    Compute $\bar{v}^p$ from $\{s_i\}$
    for $k^r = 0$ **to** $R - 1$
        Compute $\bar{v}^r$ from k^r
        Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
        Enter $\langle address + k^r * nf\_round * esize, rad\_number\rangle$ in the table.
    end for
end for
-- Enter the last element of the array, if necessary:
$\omega \leftarrow \left(\prod_{i=1}^{n} e_i - 1\right) * esize * alloc\_stride_1$
$abs\_\omega \leftarrow |\omega|$
if $\left\lfloor \dfrac{abs\_\omega}{page\_size} \right\rfloor$ changes **then**
    $address = base\_address + \omega$
    Compute $\bar{v}^p$ from $\{DUB_i\}$
    for $k^r = 0$ **to** $R - 1$
        Compute $\bar{v}^r$ from k^r
        Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
        Enter $\langle address + k^r * nf\_round * esize, rad\_number\rangle$ in the table.
    end for
end if

Second version: fancier algorithms

We now give methods to pick the elements in (roughly) the same position on each page. We need some more notation:

We will let α be a real number in the interval $[0, 1]$ that specifies the position in the physical page that we use for determining the HPF processor owning that page. That is, $\alpha = 0$ denotes the first possible position in the page, $\alpha = 1$ denotes the last possible position, and $\alpha = 1/2$ denotes a position about halfway into the page. For convenience in the following pseudo-code, we let β denote the index of the element in the first page that is at the position determined by α. So

$$\beta = \left\lfloor \alpha * \left\lfloor \frac{page\_size - 1}{esize * |alloc\_stride_1|} \right\rfloor \right\rfloor$$

Our algorithm then looks like this (the lines marked (*) and (**) are modified when we are generating "touch loops", as explained below):

Compute $\bar{v}^c$. -- This is a constant computed from the completion cells.
Compute R. -- This is a constant, computed from the replicated cells.
$p \leftarrow \beta$
while $p \leq \prod_{i=1}^{n} e_i - 1$ **do**
  $\omega = p * esize * alloc\_stride_1$
  $abs\_\omega \leftarrow |\omega|$
  if $\left\lfloor \frac{abs\_\omega}{page\_size} \right\rfloor$ changes **then**
    $p \leftarrow p - \left\lfloor \frac{abs\_\omega \bmod page\_size}{esize*|alloc\_stride_1|} \right\rfloor + \beta$
    if $p > \prod_{i=1}^{n} e_i - 1$ **then**
      Exit the loop.
    end if
    (*)$\omega = p * esize * alloc\_stride_1$
    (*)$abs\_\omega \leftarrow |\omega|$
    (*)$address = base\_address + \omega$
    Compute $\{s_i\}$ from p by (16.3)
    Compute $\bar{v}^p$ from $\{s_i\}$
    (**)**for** $k^r = 0$ **to** $R - 1$
      (**)Compute $\bar{v}^r$ from k^r
      (**)Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
      (**)Enter $\langle address + k^r * nf\_round * esize, rad\_number \rangle$ in the table.
    (**)**end for**
  end if
  $p \leftarrow p + \left\lfloor \frac{page\_size}{esize * |alloc\_stride_1|} \right\rfloor$
end while
-- Enter the last element of the array, if necessary:
$\omega = \left(\prod_{i=1}^{n} e_i - 1 \right) * esize * alloc\_stride_1$
$abs\_\omega \leftarrow |\omega|$
if $\left\lfloor \frac{abs\_\omega}{page\_size} \right\rfloor$ changes **then**
  (*)$address = base\_address + \omega$
  Compute $\bar{v}^p$ from $\{DUB_i\}$
  (**)**for** $k^r = 0$ **to** $R - 1$
    (**)Compute $\bar{v}^r$ from k^r
    (**)Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
    (**)Enter $\langle address + k^r * nf\_round * esize, rad\_number \rangle$ in the table.
  (**)**end for**
end if

We may implement this in such a way that each hpf processor computes the pages it needs (by this method) and generates a separate call to place those pages. That is, each hpf processor executes the above code in parallel and builds its own table, consisting only of those pages that it owns.

Another option is to have each hpf processor execute the code above in parallel and generate code that touches one element on each page that it owns. We refer to this as generating "touch loops". When we do this, the lines marked (*) are deleted, and the first **for** loop marked (**) is replaced by the following code:

> **if** $((\bar{v}^p = (\bar{v})_p)$**and**$(\bar{v}^c = (\bar{v})_c))$ **then**
> Touch (and mark as local) $A(s_1, s_2, \ldots, s_n)$
> **end if**

Here $(\bar{v})_p$ and $(\bar{v})_c$ denote the accumulated components of *my_processor*`()` in the primary and completion cells. That is,

$$(\bar{v})_p = \sum_{j=1}^{\pi} \left(\left\lfloor \frac{my\_processor\texttt{()}}{multiplier_j^p} \right\rfloor \bmod strip\_length_j^p \right) * multiplier_j^p$$

$$(\bar{v})_c = \sum_{j=1}^{\kappa} \left(\left\lfloor \frac{my\_processor\texttt{()}}{multiplier_j^c} \right\rfloor \bmod strip\_length_j^c \right) * multiplier_j^c$$

The expression $(\bar{v}^c = (\bar{v})_c)$ is also known as the `cc_guard` ("completion cell guard") in the implementation.

Note that since the array reference is marked as local, the term $k^r * nf\_round$ is automatically added to the first subscript s_1 when the array reference is localized; therefore, we don't have to do it here.

The second **for** loop marked (**) is replaced similarly by

> **if** $((\bar{v}^p = (\bar{v})_p)$**and**$(\bar{v}^c = (\bar{v})_c))$ **then**
> Touch (and mark as local) $A(DUB_1, DUB_2, \ldots, DUB_n)$
> **end if**

There is a problem with this algorithm in the case of a replicated or partially replicated array: unless $\alpha = 0$, it may be that the first page of one or more of the replicated versions of the array is left unmapped by this algorithm. We are not going to worry about this initially.

As before, in the case that $esize * alloc\_stride_1$ divides *page_size*, which we expect to happen frequently, this algorithm can be simplified (we omit the changes for touch generating touch loops):

Compute $\bar{v}^c$. -- This is a constant computed from the completion cells.
Compute R. -- This is a constant, computed from the replicated cells.
for $p = \beta$ **to** $\prod_{i=1}^{n} e_i - 1$ **step/** $\dfrac{page\_size}{esize * |alloc\_stride_1|}$
    $\omega = p * esize * alloc\_stride_1$
    $abs\_\omega \leftarrow |\omega|$
    $address = base\_address + \omega$
    Compute $\{s_i\}$ from p by (16.3)
    Compute $\bar{v}^p$ from $\{s_i\}$
    for $k^r = 0$ **to** $R - 1$
        Compute $\bar{v}^r$ from k^r
        Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
        Enter $\langle address + k^r * nf\_round * esize, rad\_number\rangle$ in the table.
    end for
end for
-- Enter the last element of the array, if necessary:
$\omega = \left(\prod_{i=1}^{n} e_i - 1\right) * esize * alloc\_stride_1$
$abs\_\omega \leftarrow |\omega|$
if $\left\lfloor \dfrac{abs\_\omega}{page\_size} \right\rfloor$ changes **then**
    $address = base\_address + \omega$
    Compute $\bar{v}^p$ from $\{DUB_i\}$
    for $k^r = 0$ **to** $R - 1$
        Compute $\bar{v}^r$ from k^r
        Compute $rad\_number = (\bar{v}^p + \bar{v}^c + \bar{v}^r) \bmod number\_of\_rads$
        Enter $\langle address + k^r * nf\_round * esize, rad\_number\rangle$ in the table.
    end for
end if

16.7 Distributing loop iterations on NUMA machines: examples

Now we describe how loop iterations are distributed in a NUMA machine.

The work that Transform does in lowering OMP parallel loops has conceptually two stages.

1. Distribute iterations over hpf processors.

2. Distribute the iterations in each hpf processor over the threads allocated to that hpf processor. There is a function *my_thread*`()` that returns the number of the thread that is executing.

We can think of these two stages as occurring one after the other. In the actual implementation however, they occur simultaneously.

16.7.1 Stage 1: distributing iterations over hpf processors

In the first stage, the loop bounds are localized so that they are parametrized by *my_processor*(). This is done in exactly the same way as if we were compiling for a distributed-memory machine. In addition, the subscripts of the arrays are altered in some appropriate manner.

To take a simple example, suppose we have a 3-dimensional array A, and suppose we start with the parallel loop nest in Figure 16.6.

!$omp parallel do numa
do $i_3 = LB_3, UB_3, S_3$
  !$omp parallel do firstprivate**(T), **numa
  do $i_2 = LB_2, UB_2, S_2$
    !$omp parallel do numa
    do $i_1 = LB_1, UB_1, S_1$
      $A(i, j, k) = A(i, j, k) + T$
    end do
  end do
end do

Figure 16.6: Example of nested parallel loops.

Then after stage 1, this loop nest will appear as in Figure 16.7.

!$omp fe_workshare
do $i_3 = LB_3^{loc}$, UB_3^{loc}, S_3^{loc}
  **!$omp fe_workshare firstprivate**(T)
  do $i_2 = LB_2^{loc}$, UB_2^{loc}, S_2^{loc}
    !$omp fe_workshare
    do $i_1 = LB_1^{loc}$, UB_1^{loc}, S_1^{loc}
      $A(\lambda_1, \lambda_2, \lambda_3) = A(\lambda_1, \lambda_2, \lambda_3) + T$
    end do
  end do
end do

Figure 16.7: Parallel loop nest after stage 1.

Here the localized bounds (LB_3^{loc} and so on) depend on the mapping (that is, the distribution and alignment) of the loop nest iteration space. The modified subscripts (λ_j) depend on the mapping of the array A and on whether we are using element granularity or page granularity. For instance, suppose we have a Wildfire machine consisting of 30 hpf processors (i.e., 30 "quads"), and suppose the array A is declared as follows:

real, **dimension**(0:199, 0:239, 0:299) :: A
!$omp processors $P(2,\ 3,\ 5)$
!$omp distribute(**block**,**block**,**block**) **onto** P :: A

We have made the subscripts in the array 0-based for convenience, but obviously this is not necessary.

In any case, the extents of the dimensions of this array are 200, 240, and 300. If we assume that the iteration space of this loop nest is inferred from the distribution of the array A (as would certainly be the case in this example), then, the local iteration bounds are computed as follows:

$$
\begin{array}{lll}
LB_1^{loc} = (\mathit{my\_processor}\texttt{()} \bmod 2) * 100 & UB_1^{loc} = (\mathit{my\_processor}\texttt{()} \bmod 2 + 1) * 100 - 1 & S_1^{loc} = 1 \\
LB_2^{loc} = \left(\dfrac{\mathit{my\_processor}\texttt{()}}{2} \bmod 3\right) * 80 & UB_2^{loc} = \left(\dfrac{\mathit{my\_processor}\texttt{()}}{2} \bmod 3 + 1\right) * 80 - 1 & S_2^{loc} = 1 \\
LB_3^{loc} = \dfrac{\mathit{my\_processor}\texttt{()}}{6} * 60 & UB_3^{loc} = \left(\dfrac{\mathit{my\_processor}\texttt{()}}{6} + 1\right) * 60 - 1 & S_3^{loc} = 1
\end{array}
$$

The modified subscripts, assuming element granularity, are

$$
\begin{aligned}
\lambda_1 &= i_1 - \bar{v}_1 * \mathit{num\_folds}_1 + \bar{v} * \mathit{nf\_round} \\
\lambda_2 &= i_2 - \bar{v}_2 * \mathit{num\_folds}_2 \\
\lambda_3 &= i_3 - \bar{v}_3 * \mathit{num\_folds}_3
\end{aligned}
$$

With page granularity, the subscripts are merely pushed through the data allocation function, which typically (for 1-based arrays) will simply subtract 1 from each subscript:

$$
\begin{aligned}
\lambda_1 &= i_1 - 1 \\
\lambda_2 &= i_2 - 1 \\
\lambda_3 &= i_3 - 1
\end{aligned}
$$

Of course, all these formulas depend on the particular mapping. For instance, if the mapping of A were (**block**, **cyclic**, **cyclic**(10)), or if A were given a non-trivial alignment, the formulas would look quite different. In addition, in order to manage **cyclic**(n) distributions, an additional loop would be generated for each dimension distributed **cyclic**(n). But there is nothing new in this—it is exactly what is done for distributed memory, with slight modifications only in how subscripts are treated.

16.7.2 Stage 2: Distributing iterations in an hpf processor over threads

Let us denote the number of threads per hpf processor by *threads_per_memory*. (In a typical case, $\mathit{threads\_per\_memory} = 4$.) We will assume that the hpf processor number—the number returned by *my_processor*`()`) is given by

$$
\mathit{my\_processor}\texttt{()} = \left\lfloor \frac{\mathit{my\_thread}\texttt{()}}{\mathit{threads\_per\_memory}} \right\rfloor
$$

For instance, if $\mathit{threads\_per\_memory} = 4$, then threads 0-3 are assigned to hpf processor 0, threads 4-7 to hpf processor 1, and so on.

In our implementation all the threads in an hpf processor are distributed at one loop level (rather than having the threads conceived as a multi-dimensional space whose dimensions are parceled out over various loop levels). Let us call this process “threadizing”.

!$omp fe_workshare
do $i_3 = LB_3^{loc},\ UB_3^{loc},\ S_3^{loc}$
  **!$omp fe_workshare firstprivate**(T)
  do $i_2 = LB_2^{loc},\ UB_2^{loc},\ S_2^{loc}$
    !$omp fe_workshare
    do $i_1 = \overline{LB}_1^{loc},\ \overline{UB}_1^{loc},\ \overline{S}_1^{loc}$
      $A(\lambda_1, \lambda_2, \lambda_3) = A(\lambda_1, \lambda_2, \lambda_3) + T$
    end do
  end do
end do

Figure 16.8: Parallel loops after stage 2: inner loop threadized.

Figure 16.8 shows the generated code in our running example when the inner loop is threadized.

The new quantities $\overline{LB}_1^{loc}$, $\overline{UB}_1^{loc}$, and $\overline{S}_1^{loc}$ can be computed in two different ways: (We will use *tpm* in the following formulas to denote *threads_per_memory*.)

The first way is to distribute the iterations on an hpf processor in a block fashion over the threads assigned to that processor:

$$\overline{LB}_1^{loc} = LB_1^{loc} + (\mathit{my\_thread}() \bmod \mathit{tpm}) * \left\lceil \frac{\left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{S_1^{loc}} \right\rfloor + 1}{\mathit{tpm}} \right\rceil * S_1^{loc}$$

$$\overline{UB}_1^{loc} = \min \begin{cases} UB_1^{loc} \\ LB_1^{loc} + \left((\mathit{my\_thread}() \bmod \mathit{tpm} + 1) * \left\lceil \frac{\left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{S_1^{loc}} \right\rfloor + 1}{\mathit{tpm}} \right\rceil - 1 \right) * S_1^{loc} \end{cases}$$

$$\overline{S}_1^{loc} = S_1^{loc}$$

The second way is to distribute the iterations on an hpf processor in a cyclic fashion over the threads assigned to that processor:

$$\overline{LB}_1^{loc} = LB_1^{loc} + S_1^{loc} * (\mathit{my\_thread}() \bmod \mathit{tpm})$$

$$\overline{UB}_1^{loc} = \begin{cases} \overline{LB}_1^{loc} + \left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{\mathit{tpm} * S_1^{loc}} \right\rfloor * \mathit{tpm} * S_1^{loc} & \text{if } \mathit{my\_thread}() \bmod \mathit{tpm} \leq \left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{S_1^{loc}} \right\rfloor \bmod \mathit{tpm} \\ \overline{LB}_1^{loc} + \left(\left\lfloor \frac{UB_1^{loc} - LB_1^{loc}}{\mathit{tpm} * S_1^{loc}} \right\rfloor - 1 \right) * \mathit{tpm} * S_1^{loc} & \text{otherwise} \end{cases}$$

$$\overline{S}_1^{loc} = \mathit{tpm} * S_1^{loc}$$

The code in the case that the outer loop is threadized is entirely similar. The only difference is that the bounds of the outer loop are modified, rather than those of the inner loop.

No modification of this needs to be made to handle non-tight loop nests. The design handles such loop nests as is.

We decide which loop to threadize as follows: In general, the outermost NUMA loop is threadized. However, the programmer can specify that a loop has the "shortloop" attribute. Shortloops will not be threadized unless there is no alternative, and the compiler searches for the outermost non-shortloop to threadize. Since the NUMA loops form a forest, each tree has to be handled separately, and since shortloop declarations can be scattered throughout this tree, some care has to be taken in the compiler to handle this correctly. The actual algorithm used is as follows:

1. Perform a depth-first walk of each numa loop tree. Some of the loops are marked S ("shortloop").

   On each path down from the root, mark the first non-S loop as M ("go multiple"), and don't continue the walk further down the tree at that node.

   On the way back up the tree, mark each ancestor of an M node as A ("above M").

2. Perform a second depth-first walk of each numa loop tree. For each node N,

   a) if N is an M node, stop the recursion at this point.

   b) if N is an A node (and so in particular not an M node), recurse.

   c) if N is an S node but not an A node, mark it as M and stop the recursion at this point.

There are no other possibilities. For the root of the tree must be either an S node (and possibly also an A node) or an M node. If any other node N is unmarked, then it must not have been reached in the first walk, which means that there is an M loop above it, and this in turn means that it is not reached in the second walk.

16.8 Generating NUMA code for array and scalar assignments

We consider assignments, including FORALL and WHERE constructs. Except when specifically stated, the term "subspace" or "cell" refers to an iteration subspace or cell. We distinguish between two kinds of iteration subspaces:

NUMA subspaces: These are subspaces that correspond to OMP PARALLEL DO NUMA loops.

Non-NUMA subspaces: all other subspaces. Such subspaces may correspond to named or unnamed dimensions in the source.

STRIP generates loops based on iteration space. STRIP generates addressing based on data space, and optionally may use iteration space as well to implement some optimizations. Generating loops and generating addressing are taken up in the next two subsections.

16.8.1 Generating loops

Assignments inside a NUMA region: default processing

If any of the non-NUMA subspaces are not already serial, DIVIDE reclassifies this region as not NUMA.

These non-NUMA serial subspaces in NUMA regions are then automatically turned into serial loops by STRIP.

The iteration space for the NUMA loop is composed solely of the single corresponding primary subspace of the iteration space of its `home_ref`. This iteration space is saved in the `independent_ispace` attribute of the loop.

If the target of the assignment contains a replicated cell in its iteration space, the statement is classified by DIVIDE as a STMT_NUMA_REPLICATED_STORE. This causes STRIP to generate a serial do loop for each such replicated cell.

If the target of the assignment contains one or more completion cells, those completion cells are not included in the iteration space of the NUMA loop.

Array references within NUMA loops have a Boolean attribute `numa_local`; see Section 16.8.2 for details.

Assignments inside a NUMA region: optimized processing

The default processing described above can lead to inefficient code in some cases. For example, the code in Figure 16.9 might be a reasonable thing to write in HPF:

```
real, dimension(n, n) :: b
real, dimension(n) :: a
!hpf$ distribute(block,block) :: b
!hpf$ align a(i) with b(i, *)

!hpf$ independent
do i = 1, n
    a(i) = ...
    !hpf$ independent
    do j = 1, n
        b(i, j) = ... a(i) ...
    end do
end do
```

Figure 16.9: HPF code with array a partially replicated over a dimension of array b.

Figure 16.10 shows the corresponding code in OMP. There is an important difference in how code is generated for these two examples: In HPF, all the processors are active at all levels of the loop nest. Therefore, the assignment to all the replicated instances of $a(i)$ happens locally and in parallel. In OMP, however, since b is of necessity chosen to be the `home_ref`, only one processor holding each $a(i)$ has a thread that executes the assignment to $a(i)$. Therefore, this assignment has to be enclosed

```
real, dimension(n, n) :: b
real, dimension(n) :: a
!dec$ distribute(block,block) :: b
!dec$ align a(i) with b(i, *)

!$omp parallel do numa
do i = 1, n
    a(i) = ...
    !$omp parallel do numa
    do j = 1, n
        b(i, j) = ... a(i) ...
    end do
end do
```

Figure 16.10: OMP code with array a partially replicated over a dimension of array b.

in a serial do loop by STRIP. (This happens, as described above, because the assignment to $a(i)$ is classified as a STMT_NUMA_REPLICATED_STORE.) Thus the assignments happen serially, and most of the references are non-local. It would most likely have been better in this case for the OMP programmer to have written something like the code in Figure 16.11. In that figure, the assignment to $a(i)$ could in fact be done in parallel.

```
real, dimension(n, n) :: b
real, dimension(n) :: a
!dec$ distribute(block,block) :: b
!dec$ align a(i) with b(i, *)

!$omp parallel do numa
do i = 1, n
    a(i) = ...
end do

!$omp parallel do numa
do i = 1, n
    !$omp parallel do numa
    do j = 1, n
        b(i, j) = ... a(i) ...
    end do
end do
```

Figure 16.11: OMP code with array a partially replicated over a dimension of array b; improved version.

As an optimization, our compiler generates code in the "HPF manner" where it is safe to do so. What this means is that our compiler may create a team at a given loop level containing all the threads for that loop and one or more loops below it, provided only that the compiler can prove that this leads to no visible change in the output of the program. So if for instance the programmer had included a call to `omp_get_num_threads()`, or to a non-intrinsic function in this code somewhere, the compiler would not perform this optimization. In addition, this optimization is suppressed when the program is compiled `-O0` (i.e., with optimization turned off).

In the example in Figure 16.10, introducing this optimization would mean that the compiler generates a team consisting of all the available threads at the outermost loop level (and introduces no new threads for the inner loop).

Figure 16.12 shows another example in which our compiler allocates an entire team of threads, even though in principle it should not need to. In this figure, a slice of b is being initialized. A

```
real, dimension(n, n) :: b
!dec$ distribute(block,block) :: b

!$omp parallel do numa
do i = 1, n
    b(i, n) = 1
end do
```

Figure 16.12: OMP code initializing a slice of the array b.

1-dimensional team of threads could in principle be constructed to perform this initialization. The problem is that the set of threads that have data local to them (let us call this the "natural" set of threads for this loop) may not contain the initial (or "master") thread that is executing when this construct is reached, because the master thread may be bound to a processor on a memory not containing any data in this slice.

This natural set of OpenMP threads therefore could not by itself constitute an OpenMP *team* of threads because it does not include the master thread. So in this case, our compiler creates a team consisting of all the threads, and guards the assignment statement so that only those threads that live on hpf processors owning elements of the slice $b(:, n)$ participate in the assignment statement.

To make things simple in our initial implementation of this optimization, we only generate this HPF-style code when the compiler can prove that it is safe to (as specified above) *and also* when the following conditions are satisfied:

Let H denote the `home_ref` of the loop nest. For each left-hand side array reference L in the loop nest,

1. Each subspace of L has the same cell place and bounds as a primary subspace of H.
2. Any subspace of H not accounted for in item 1 has the same cell place as a replicated cell of L.
3. Any replicated cell of L not accounted for in item 2 has the same cell place as a replicated cell of H.
4. There are no replicated cells of H not accounted for in item 3.
5. There must be a 1-1 correspondence between the cell places of the completion cells in H and the those of the completion cells of L. (The corresponding values do not have to be equal, however.)

Further, the following constraint must also be satisfied:

6. The HPF_NUMA region (i.e., the region immediately containing the HPF_NUMA loop must not contain any non-NUMA loop whose loop index subscripts a mapped dimension of the `home_ref`.

   The reason for this last condition is that such a dimension in the `home_ref` would result in a completion cell in the iteration space of the `home_ref` (which would then get propagated, as explained below) to the `independent_ispace` of the loop. The value in this completion cell would not be an invariant of the HPF_NUMA loop; in fact it would be a function of the loop variable of the inner (non-NUMA) loop. Thus, we would end up generating a `cc_guard` around the HPF_NUMA loop that depended on the value of an inner loop; this is obviously a semantic error.

If these conditions are not all met, the HPF-style coding optimization is not performed.

Each array reference has a Boolean attribute `numa_local`. If the completion of L is not identical with the completion of H, or if any of the iteration subspaces of L has no corresponding primary data subspace (e.g., there is a complex data subspace), then the `numa_local` attribute of L is set to FALSE. (See Section 16.8.2 below.)

If any completion cell in the iteration space of the `home_ref` has a value different from 0 (or actually, has a value corresponding to a processor coordinate different from 0), then the set of virtual processors corresponding to the iteration space of the `home_ref` will not include processor 0, which is the processor assumed to hold the master thread on entry to the NUMA loop nest. For this reason, we will turn such completion cells into replicated cells for the purpose of allocating threads, and then guard the HPF_NUMA loop with the set of values of those completion cells.

Thus, when the compiler can prove that it is safe to generate HPF-style code, the following processing is performed:

1. The outermost NUMA loop is tagged as HPF_NUMA. This tells strip to allocate all the threads at that loop level. The iteration space that describes the totality of threads to be allocated is that attached to the `home_ref` of the HPF_NUMA loop.

2. Any completion cell in the iteration space of the `home_ref` that corresponds to a processor coordinate different from 0 is turned into a replicated cell.

3. The `independent_ispace` attribute of the HPF_NUMA loop is given a completion structure composed precisely of those completion cells specified in the previous item. This will cause STRIP to generate a `cc_guard` over the HPF_NUMA loop restricting execution to those processors on which the `home_ref` actually lives.

In this way we fix the `home_ref` and the `independent_ispace` so that

- The `home_ref` is used to generate the `numa_config descriptor` that describes the team of threads that is started up on entry to the HPF_NUMA loop.

- The `independent_ispace` is used to determine the loop bounds and also to build a `cc_guard` around the loop if needed to disable some of the threads started up as just described.

```
real, dimension(n) :: a
!dec$ template T(n, n)
!dec$ distribute(block,block) :: T
!dec$ align a(i) with T(i, 1)

!$omp parallel do numa
do i = 1, n
   a(i) = a(i) + 1
end do
```

Figure 16.13: Ordinary OMP code; only one column of threads is started up.

Here are some examples that illustrate the constraints that are needed to perform this optimization:

First, Figure 16.13 shows a simple loop. There is no need for special treatment of this loop. Nevertheless, it is worth noting that in this example, not all threads will be active during the execution of this loop—only those threads corresponding to the first column of the template T will be started up on entry to this parallel loop.

Figure 16.14 is a slight variation of this in which the HPF-style optimization is useful. Here the iteration space of the `home_ref` has a completion cell corresponding to a non-zero processor coordinate.

```
real, dimension(n) :: a
!dec$ template T(n, n)
!dec$ distribute(block,block) :: T
!dec$ align a(i) with T(i, n)

!$omp parallel do numa
do i = 1, n
   a(i) = a(i) + 1
end do
```

Figure 16.14: OMP code needing threads starting up on a column different from the first column.

Therefore, this completion cell is turned into a replicated cell in the iteration space of the `home_ref` and the original completion cell is placed in the `independent_ispace` attribute of the loop. What we generate is similar to what might have been written somewhat like the code in Figure 16.15.

Figure 16.16 shows a somewhat more complicated example. In this loop, the `home_ref` is $c(i, j)$. Since the reference $a(i)$ has a completion cell not represented in the iteration space of the `home_ref`, this loop is not suitable for HPF-style optimization. In fact, one can see that if HPF-style optimization were attempted, so that all the threads were started up on entry to the outer loop, the following would happen:

1. If the assignment to $a(i)$ were guarded, then a thread not participating in that assignment could enter the inner loop and use a value of $a(i)$ that had not yet been updated.

2. If on the other hand the assignment to $a(i)$ were not guarded, then the assignment would be ex-

real, **dimension**(n) :: a
!dec$ template $T(n, n)$
!dec$ distribute(**block**,**block**) :: T
!dec$ align $a(i)$ **with** $T(i, *)$

!hpf$ on home$(T(:, n))$ **begin**
  !$omp parallel do numa
  do $i = 1, n$
    $a(i) = a(i) + 1$
  end do
!hpf$ end on

Figure 16.15: What TRANSFORM in effect generates for the code in Figure 16.14.

real, **dimension**(n) :: a
real, **dimension**(n, n) :: c
!dec$ template $T(n, n)$
!dec$ distribute(**block**,**block**) :: T
!dec$ align $a(i)$ **with** $T(i, 1)$
!dec$ align $c(i, j)$ **with** $T(i, j)$

!$omp parallel do numa
do $i = 1, n$
  $a(i) = a(i) + 1$
  !$omp parallel do numa
  do $j = 1, n$
    $c(i, j) = a(i)$
  end do
end do

Figure 16.16: OMP code not admitting HPF-style optimized code generation. If a is partially replicated over the second dimension of T, however, this optimization can be performed.

ecuted redundantly by more than one thread, which is undoubtedly not what the programmer intends.

Suppose, however, that the code in Figure 16.16 is altered so that the array a is replicated over the second dimension of T:

!dec$ align $a(i)$ **with** $T(i, *)$

In this case, the entire construct can be performed with HPF-style optimization. Now all the threads are active at each point. While more than one thread participates in the assignment to $a(3)$, for instance, it is updating (and reading from) a separate instance of the partially replicated array, so that each instance of $a(3)$ is updated exactly once. No thread enters the inner loop before the value of $a(i)$ that it will use in that loop has its new value.

Figure 16.17 contains a still more complicated example. The HPF-style optimization can be used

```
real, dimension(n) :: a, b
real, dimension(n, n) :: c
!dec$ template T(n, n, n)
!dec$ distribute(block,block,block) :: T
!dec$ align a(i) with T(i, *, 1)
!dec$ align b(i) with T(i, *, 2)
!dec$ align c(i, j) with T(i, j, 2)

!$omp parallel do numa
do i = 1, n
   a(i) = a(i) + 1
   b(i) = a(i)
   !$omp parallel do numa
   do j = 1, n
      c(i, j) = b(i)
   end do
end do
```

Figure 16.17: OMP code susceptible to HPF-style optimized code generation.

for this code. Note that again in this code, each assignment to each of the replicated versions of $a(i)$ is performed exactly once (i.e., not redundantly by more than one thread), just as one would expect. (However, note also that the `numa_local` attributes of the array references $a(i)$ are all set to FALSE.) Further, the assignment to each instance of $b(i)$ happens on each participating thread only after the value $a(i)$ on the right-hand side has been updated. And finally, each thread enters the inner parallel loop only after its values of $a(i)$ and $b(i)$ have been computed.

If the compiler can prove that the completion value in the iteration space of the `home_ref` corresponding to the constant 2 in the alignment of $c(i, j)$ specifies a processor coordinate of 0, then there is nothing special to be done in this case. If it cannot prove this, however, that completion cell will have to be turned into a replicated cell in the iteration space of the `home_ref`, and the original completion cell will be put as the only element of the `independent_ispace` attribute of the outermost loop.

Finally, in Figure 16.18, we add a new innermost loop. Since the `home_ref` is now $d(i, j, k)$, the constraint on completion cells inhibits HPF-style code generation from being performed.

Assignments inside a NUMA region: a further optimization

In the loop in Figure 16.19, the processing indicated in the previous section would (assuming $a(i, 1)$ is the `home_ref`) create a team of threads living on the hpf processors owning elements of $a(:, 1)$. The references to $b(i, n)$ would all be remote references.

However, there is really no need for this to happen: in this case, we could certainly start up *all* the threads, and simply guard each statement so that only the threads appropriate for each statement executed it.

```
real, dimension(n) :: a, b
real, dimension(n,n) :: c
real, dimension(n,n,n) :: d
!dec$ template T(n,n,n)
!dec$ distribute(block,block,block) :: T
!dec$ align a(i) with T(i,*,1)
!dec$ align b(i) with T(i,*,2)
!dec$ align c(i,j) with T(i,j,2)
!dec$ align d(i,j,k) with T(i,j,k)

!$omp parallel do numa
do i = 1,n
    a(i) = a(i) + 1
    b(i) = a(i)
    !$omp parallel do numa
    do j = 1,n
        c(i,j) = b(i)
        !$omp parallel do numa
        do k = 1,n
            d(i,j,k) = c(i,j) + k
        end do
    end do
end do
```

Figure 16.18: The new innermost loop inhibits HPF-style code generation.

```
real, dimension(n,n) :: a, b
!dec$ distribute(block,block) :: a, b

!$omp parallel do numa
do i = 1,n
    a(i,1) = 1
    b(i,n) = 1
end do
```

Figure 16.19: OMP code needing threads starting up on a column different from the first column.

The only thing that could go wrong with this is if there were two statements in a single loop iteration which were executed by different threads, and in which the second statment depended on the results of the first. So if we can guard against this possibility, we can perform this further variant of HPF style optimization.

We do this by looking for the following conditions:

- A loop nest is suitable for HPF_NUMA optimization as described in the previous section.
- There are at least two statements in the loop whose iteration spaces contain completion cells

with different values.

- The compiler can determine that there are no loop-independent dependences within the loop nest.

In such a case, the optimization described in the previous section can be modified in the following way:

- The `independent_ispace` attribute of the HPF_NUMA loop is not given any completion cells.
- Each statement in the loop is guarded by a guard representing the completion cells for that statement.

Thus in the loop in Figure 16.19, we can start up all the threads, and guard each statement in the loop so that only those threads that own elements of $a(i, 1)$ or $b(i, n)$, respectively, execute those statements. With this further optimization, both these array references can be marked `numa_local`.

Assignments outside a NUMA region

DIVIDE ensures that the completion structure of any iteration space outside a NUMA region must be empty. (DIVIDE can do this simply by wiping out any non-empty completion structure.) This is needed because only one thread is executing, and the compiler has no control over thread allocation.

DIVIDE also changes the distribution of each subspace if necessary so that it is serial. STRIP will then generate serial loops for each subspace, removing obstacles if necessary. (If all the arrays in the statement having primary cells in the assignment are page-granularity arrays, then STRIP could avoid generating loops, and just "localize" each triple. However, it is easier to just let STRIP treat both cases in a consistent manner.)

As a result of these actions by DIVIDE, the normal correspondence between iteration and data space mappings may be lost. However, STRIP will generate the loops in any case according to the iteration space.

If the target of an assignment is totally or partially replicated, STRIP generates serial do loops to handle the replicated store. To set up for this, such a statement is classified as before by DIVIDE as a STMT_NUMA_REPLICATED_STORE.

16.8.2 Generating addressing

All assignments are classified by DIVIDE as STMT_LOCAL_ASSIGN except for those classified as STMT_NUMA_REPLICATED_STORE.

In either case, each array reference in the statement has a BOOLEAN `numa_local` attribute, with the following semantics:

`numa_local` is TRUE iff the array reference would actually be a local reference in a distributed memory implementation, and it is FALSE iff the array reference would actually be a remote reference in distributed memory. To be precise, the DIVIDE routine same_region() compares the iteration space of the array reference to

- the iteration space of the root of the statement if the statement is not in a NUMA PARALLEL DO construct;

- a copy of the iteration space of the `home_ref`, modified so that the completion values are set equal to 0 if the statement is in a NUMA PARALLEL DO but not in an HPF_NUMA loop.

  (Note that in this case the iteration space of the `home_ref` may be multi-dimensional. We can't use the iteration space of the loop (i.e. the `independent_ispace`) because in this case it will be 1-dimensional, while the array reference itself will in general be multi-dimensional.)

- the `independent_ispace` attribute of the DO loop if the statement is in a NUMA PARALLEL DO tagged as HPF_NUMA.

  (Note that in this case the `independent_ispace` will be multidimensional, and will also have completion cell information, so we can use it for this purpose. The iteration space of the `home_ref`, on the other hand, has had its completion cells turned into replicated cells, so we cannot use it for this purpose.)

Then `numa_local` is TRUE iff same_region() returns TRUE.

Thus, if `numa_local` is TRUE, STRIP may make use of iteration space to optimize the addressing computations. If on the other hand `numa_local` is FALSE, then STRIP uses only data space in generating the addressing computations.

The `numa_local` attribute of the left-hand side of a STMT_NUMA_REPLICATED_STORE statement is always FALSE, since a single thread must assign to all the replications of each element of the array, and these replicated elements of necessity lie on different hpf processors.

As indicated above on page 199 (see particularly expression 16.2), the normal code generated for a array reference that is not `numa_local` will be of the form

$$A\big(\hat{v}_1 + (k^p + k^r * P) * \mathit{nf\_round}, \hat{v}_2, \ldots, \hat{v}_n\big)$$

(this is just expression 16.2), where now actually $k^r = 0$, so the last term in the first subscript is missing. If the reference is the left-hand side of a statement classified as STMT_NUMA_REPLICATED_STORE, STRIP builds a loop around this statement. The loop index is k^r; it runs from 0 to $R-1$, and the term $k^r * \mathit{nf\_round} * P$ then appears in the first subscript of the array reference. In this way, all the replicated instances of each array element are assigned to.

Array references outside a NUMA loop always have their `numa_local` attribute set to FALSE.

Some of the preceding discussion can be summarized in the following table:

	HPF_NUMA loop	NUMA but not HPF_NUMA	outside NUMA construct
`numa_config` descriptor computed from	`home_ref` with completions → replicated (multi-dimensional)	`independent_ispace` (has no completion cells)	—
`cc_guard` computed from	`independent_ispace` (has completion cells)	`independent_ispace` (has no completion cells; no `cc_guard`)	—
`numa_local` computed from	`independent_ispace` (multi-dimensional; has completion cells)	`home_ref` with completion values set to 0 (multi-dimensional)	root of the statement

16.8.3 Generating NUMA code for intrinsic functions and subroutines

Any procedure with one or more array arguments that falls into one of the following categories:

- a transformational intrinsic function not inlined by TRANSFORM, or
- a non-elemental intrinsic subroutine not inlined by TRANSFORM

is punted, as follows:

Such procedures are raised to the statement level. Each mapped argument that is not either serial or page granularity is pulled out and copied into a page granularity temporary. This temporary is passed to the procedure, after which the temporary is copied back to the original argument, if appropriate.

The resulting procedure is then classified as STMT_IGNORE and simply passed through to the middle end. Since all its arguments are now page granularity, the rest of the compiler can generate whatever addressing is needed in lowering the procedure.

> *This punting is performed simply because no part of the compiler after TRANSFORM has knowledge of element-granularity data layout. In the future, we plan to do a better job with these intrinsics.*

16.8.4 Actual arguments to user defined functions and subroutines

Here are the rules we follow in deciding whether to create a temporary for an actual at a call site:

No explicit interface:

Without an explicit interface, any actual argument containing a mapped object will cause the Front End to generate a diagnostic, unless the actual argument is a whole array. Thus, if the Front End does not already generate a diagnostic, there are only two possibilities:

- The actual argument contains no mapped objects. In this case, the Middle End can handle any needed generation of temporaries.
- The actual argument is a mapped whole array. In this case, the ordinary HPF rule states that (since there is no explicit interface), no temporary is required.

So in either case, TRANSFORM does not generate a temporary in this case.

Explicit interface:

- If the actual is a "whole scalar", i.e., a TMPTOK, CONTOK, or scalar SYMTOK, then do nothing.
- Else if the dummy is a scalar SYMTOK (note that a dummy can never be a TMPTOK or a CONTOK), then do nothing.
- Else if the actual is an expression or a section (i.e., anything that is not a whole array), we make a temporary.

We do this not to enforce language rules, but for an implementation reason: we know the dummy must be an array. If the Middle End decides to make a temporary, it would execute a call-back to ask us how to fill in the dope vector. But the current implementation for the call-back routines only handles symbols, not expressions.

Further, at this point, it is legal for us to create a temporary—since there is an explicit interface, we know that the shapes of the actual and the dummy agree. (Without an explicit interface, the programmer could pass a single element of an array, like $A(1)$, with a dummy that is an array.)

- Else (now we know that the actual and the dummy are both whole arrays) if both the actual and the dummy are page granularity, do nothing.
- Else (the actual and the dummy are both whole arrays and the mapping is element granularity) compare the mappings.
  - If the mappings are compatible, do nothing.
  - If the mappings are not compatible,
    - If the mapping on the dummy is prescriptive, remap the actual.
    - If the mapping on the dummy is descriptive, do nothing.

There is an optimization that we can make, based on the fact that we are generating code for a shared-memory machine. In general, if a proper section of an array is passed to a subroutine and there is an explicit interface in which the corresponding dummy has anything other than an INHERIT mapping, we remap the entire actual argument into a temporary and pass a section of that temporary; this just follows HPF rules for parameter passing. However, in certain circumstances, we do not need to make a temporary. Suppose all the following conditions are satisfied:

- The actual argument is a scalar array reference (i.e., if it contains no sections).
- The actual is mapped serially in the first dimension.
- There is an explicit interface.
- The dummy is explicit shape.
- The size of the dummy is less than or equal to the extent of the first dimension of the actual.
- The dummy is effectively serial in every dimension (this will be true in our NUMA compiler for unmapped arrays).

In such a case, no temporary is needed—we can legally just pass the address of the first element of the passed section. This is because the actual section being passed is guaranteed to be contiguous, as is the dummy being received.

These conditions could be relaxed if needed. We will do that if there is demand for it.

Now the actual argument is modified as follows by STRIP:

page-granularity arrays

- Each scalar subscript is just pushed through the data alignment function. This is normal localization for page-granularity scalar subscripts.

- Each triple has its bounds pushed through the data alignment function. This just keeps the vector argument looking like a whole array to the middle end.

element-granularity arrays

- Each scalar subscript is localized as usual for element-granularity scalar subscripts.
- Each triple is replaced by $\langle 0 : num\_folds_j : 1\rangle$. (Since DTR2CIR specifies that these are the bounds of the array, this keeps the array looking like a whole array, and thus the Middle End does not create a temporary for it.)

16.8.5 Actual arguments to I/O routines

ARG always needs to make a temporary for the actual argument to an I/O routine if the original actual argument is a vector with element granularity. In any other case, no special processing by ARG is needed.

STRIP then treats such actuals exactly the same as actuals to user-defined functions, as above.

16.9 Allocation of element-granularity mapped local arrays

16.9.1 Overview

Let us call an array declared in a subprogram that is not a dummy argument a *local* array. Thus, all automatic arrays are local arrays, but a local array might have constant bounds and therefore not be automatic. Unfortunately, this definition of "local" conflicts with the standard Fortran definition, but we use it only here.

Mapped local arrays have to be allocated on entry to the subprogram in which they are defined, and deallocated on exit from that subprogram. Such arrays are typically allocated on the stack. Thus, on entry to the subprogram, code has to be generated to map the the virtual pages of the stack to the appropriate hpf processors based on the mapping of the local variables. Furthermore, if two different subprograms contain local variables with different mappings, the virtual stack pages will have to be remapped on each entry. This is a time-consuming process. There is, however, a way that we can overcome this problem, at least in the case of element-granularity local arrays:

At the start of the program (or actually, at the start of any user thread that will create an outermost NUMA parallel region), some storage is allocated on each hpf processor. We can think of this as a collection of "local" stacks that together constitute a separate TRANSFORM-managed stack. Note that these separate "local" stacks are in global memory, not in thread-local storage; an array allocated over these stacks needs to be globally visible to all threads in the subprogram. For convenience in addressing, this collection of local stacks can be allocated as a continuous range of global addresses.

On entry to each subprogram, each element-granularity mapped local array will have storage reserved for it in each of these local stacks. Only as much storage will be reserved in each local stack as is necessary to hold the local section of the array. (Well, actually, we reserve *num_folds* on each hpf processor.) Since the array is assumed to be element-granularity, we do not have to be concerned about the fact that the pages allocated to an array are not in array element order, or in fact that an array may not use up all of a page before skipping to a page on another hpf processor.

Mapping directives are not honored for arrays in a scope that is entered from within a parallel region. This constraint means that each such stack will only be manipulated by one thread (i.e., the single thread that initially starts executing the program, or any explicitly user-created thread), and as a result, there will be no problem maintaining the integrity of these stacks.

The main advantage of this technique is that the pages never have to be remapped. Once the original allocation has been done, the operating system is never involved in this stack management. Since we are managing this memory as a stack (or a set of stacks), allocation and deallocation are quite efficient.

A secondary advantage of this technique is that we do not waste any memory. These arrays are allocated even more efficiently than element-granularity mapped arrays that could not be placed on the stack.

16.9.2 The details

1. Each symbol designating a mapped local array is given the `NUMA_STACK` handy bit. For each such array A, symbols *@A_base* and *@A_extent* are created by DAT and installed in the `base` and `extent` fields of the dope for A. (Thus, assignments to *@A_base* and *@A_extent* assign to those fields in the dope.)

   The stack described in the previous section is called the *numa stack*. *@A_base*, which will be given its value at run-time, represents the base offset of A in the numa stack. *@A_extent*, which also gets its value at run-time, holds offset in the stack from one hpf processor's memory to the next. Thus, *@A_extent* conceptually represents the memory in the numa stack allocated to one processor. However, as shown below, both *@A_base* and *@A_extent* have to be calculated with some adjustments, due to the fact that the element size of the array may not be a factor of the local numa stack size.

2. STRIP modifies the representation of references to elements of A so that

   (a) the value *@A_extent* is used for *nf_round*, and in addition,

   (b) the value *@A_base* is added to the first subscript of the array.

   The base address of the array is still represented as A after STRIP, however.

   Thus, the reference to A looks like this:

   $$A\big(\hat{v}_1 + (k^p + k^r * P) * @A\_extent + @A\_base, \hat{v}_2, \ldots, \hat{v}_n\big) \tag{16.4}$$

   where k^p and k^r are derived from the hpf processor number corresponding to the array element being referenced, which is not necessarily the number of the processor performing the reference. Therefore, k^p and k^r are computed exactly as for arrays not allocated on the numa stack. In particular, if the reference is not `numa_local` then $k^r = 0$.

   Note that, because the first subscript is multiplied by the element size of the array by the Middle End, the values of *@A_base* and *@A_extent* have to represent the "base" and "extent" values divided by *esize*. Since *esize* may not be a factor of either of these values, they first have to be rounded up to a multiple of *esize* (and of course, this has to be done on an array-by-array basis, since *esize* differs for each array. This is why *@A_extent*, which in principle has nothing to do with A, is actually a property of the symbol A.

For arrays which are passed through—these are whole array actual arguments—STRIP replaces the array by the scalar array reference

$$A\big(\,@A\_base + DLB_1, DLB_2, \ldots, DLB_n\big)$$

3. If the array reference is the left-hand side of a STMT_NUMA_REPLICATED_STORE statement, then as described above on page 224, STRIP builds a loop around this statement. Again, the reference ends up looking like this:

$$A\big(\hat{v}_1 + (k^p + k^r * P) * @A\_extent + @A\_base, \hat{v}_2, \ldots, \hat{v}_n\big)$$

   but here k^p is actually a loop index running from 0 to $R - 1$.

   Here the numa adjustment term is $(k^p + k^r * P) * @A\_extent + @A\_base$.

4. The Middle End, when processing an array reference whose base symbol (A, say) has the `NUMA_STACK` handy bit set, replaces the base address by *_OtsNumaStackBase*, and tags the array reference as having the (original) base symbol A. This creates the correct addressing, and tells GEM to regard the array reference as a reference to an element of A, even though it doesn't look that way.

   When the array reference is a whole array, this has the effect that the address passed is

$$\_OtsNumaStackBase + @A\_base * esize$$

   In addition, the `NUMA_STACK` handy bit is a signal to the Middle End to treat the symbol A in such a way that no memory is allocated to it. (Since TRANSFORM is handling allocation and deallocation explicitly, the ordinary memory allocation and deallocation that would otherwise be performed for local arrays has to be inhibited.)

   It thus remains to be shown how the stack is manipulated, and how the values *@A_base* and *@A_extent* are determined, at run-time.

5. An environment variable *OMP_NUMA_STACKSIZE* is used to determine the total amount of memory (in bytes) in the numa stack that will be allocated to each hpf processor. This amount is rounded up (if necessary) to a multiple of $number\_of\_processors * page\_size$ and stored in the variable *_OtsNumaStackSize*. Then the amount of the stack allocated to each hpf processor is stored in the variable

$$\_OtsNumaStackLocalSize \leftarrow \frac{\_OtsNumaStackSize}{number\_of\_processors}$$

6. At user thread startup time (the main program is considered a user thread in this context), the thread allocates on the heap *_OtsNumaStackSize* bytes of contiguous memory. The pages of this memory are mapped in a block fashion to the *number_of_processors* hpf processors that will be started up under this user thread. The base address of the memory thus allocated is stored in the variable *_OtsNumaStackBase*.

   In addition, there is an integer variable *_OtsNumaCurrentTos* that holds the position of the top of the local stack (i.e., the position of the top of the stack on processor 0). This variable is initialized to 0 at the start of the program.

7. On entry to a subprogram, the executing thread first saves the value of *_OtsNumaCurrentTos* in a variable *@LocalStackTop* on the (global) stack of the subprogram. That thread then allocates the element-granularity mapped local arrays for that subprogram. For each such array (call it A, say), the thread does the following:

   (a) The current value of *_OtsNumaCurrentTos* is first rounded up to the next highest multiple of the element size *esize* of the array. Its value divided by *esize* is then assigned to the variable *@A_base*.

$$\begin{aligned} \mathit{@A\_base} &\leftarrow \left\lceil \frac{\mathit{\_OtsNumaCurrentTos}}{\mathit{esize}} \right\rceil \\ \mathit{\_OtsNumaCurrentTos} &\leftarrow \mathit{@A\_base} * \mathit{esize} \end{aligned}$$

   (b) In a similar fashion, the variable *@A_extent* is assigned a value as follows:

$$\mathit{@A\_extent} \leftarrow \left\lceil \frac{\mathit{\_OtsNumaStackLocalSize}}{\mathit{esize}} \right\rceil$$

   (c) Then *_OtsNumaCurrentTos* is incremented by

$$\mathit{num\_folds} * \mathit{esize} + (\mathit{number\_of\_processors} - 1) * (\mathit{@A\_extent} * \mathit{esize} - \mathit{\_OtsNumaStackLocalSize})$$

8. On exit from the subprogram, the value of the variable *_OtsNumaCurrentTos* is restored to the saved value that it had on entry to the subprogram.

This design makes no provision for exception handling; we assume that all exceptions are unrecoverable.

Chapter 17

Examples

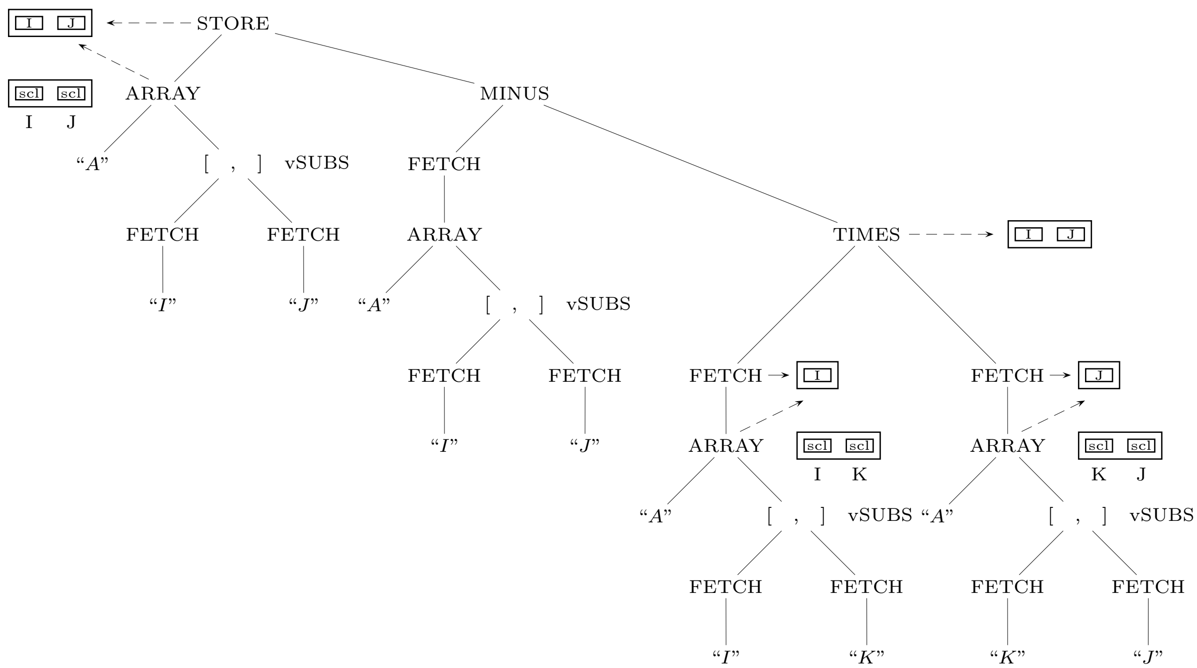
17.1 A Fortran DO loop nest

We consider code for an LU decomposition, without pivoting. Actually, only the outermost loop needs to be serial—the other loops should be characterized as INDEPENDENT, or written as FORALL constructs. So in the example here, where we do not indicate such parallelism in the source code, the generated code will be very poor. Nevertheless, it is a useful example in showing how our data structures work in practice.

do $K = 1, N$
    do $I = K + 1, N$
        $A(I, K) = A(I, K)/A(K, K)$
        do $J = K + 1, N$
            $A(I, J) = A(I, J) - A(I, K) * A(K, J)$
        end do
    end do
end do

17.1.1 CIR2DTR

The dotree for the innermost statement looks like this:



where arrows point to iteration space structures, and each data space structure is next to its corresponding ARRAY node. All the data subspaces are local scalar subspaces, and are labeled scl (scalar). All the iteration subspaces are part of the completion structure.

17.1.2 ITER

If the array A is laid out serially in the first variable, then communication is only necessary to bring $A(I,K)$ in the second statement over to the processor holding the J column of A, where it is needed for the rest of the computation.

This is reflected in the following processing: When DIVIDE reaches the TIMES node, it looks at its two children. The right-hand child, a FETCH node, is being spread into a dimension which is allocated serially, and hence no motion is needed. Motion is needed at the left-hand child, however, and so DIVIDE computes it into a temp. Since the spread is into the J dimension, DIVIDE can allocate the temp immediately before the J loop and deallocate it immediately after.

Thus, after DIVIDE, the loop nest looks (in a source representation) like this:

do $K = 1, N$
  do $I = K + 1, N$
    $A(I, K) = A(I, K)/A(K, K)$
    Allocate $T(K + 1 : N)$
    do $J = K + 1, N$
      $T(J) = A(I, K)$
      $A(I, J) = A(I, J) - T(J) * A(K, J)$
    end do
    Deallocate $T(K + 1 : N)$
  end do
end do

where now the only motion happens at the assignment to $T(J)$. The allocation and deallocation of T actually happen in the prolog and epilog blocks of the J loop.

17.1.3 STRIP

STRIP now localizes the subscripts and exposes the motion:

do $K = 1, N$
  do $I = K + 1, N$
    if $A(I, K)$ is local then
      $A(I, K) = A(I, K)/A(K, K)$
    Allocate $T(K + 1 : N)$
    do $J = K + 1, N$
      if $A(I, K)$ is local then
        SEND $A(I, K)$ to $pr(T(J))$
      if $T(J)$ is local then
        RECEIVE $T(J)$
      if $A(I, J)$ is local then
        $A(I, J) = A(I, J) - T(J) * A(K, J)$
    end do
    Deallocate $T(K + 1 : N)$
  end do
end do

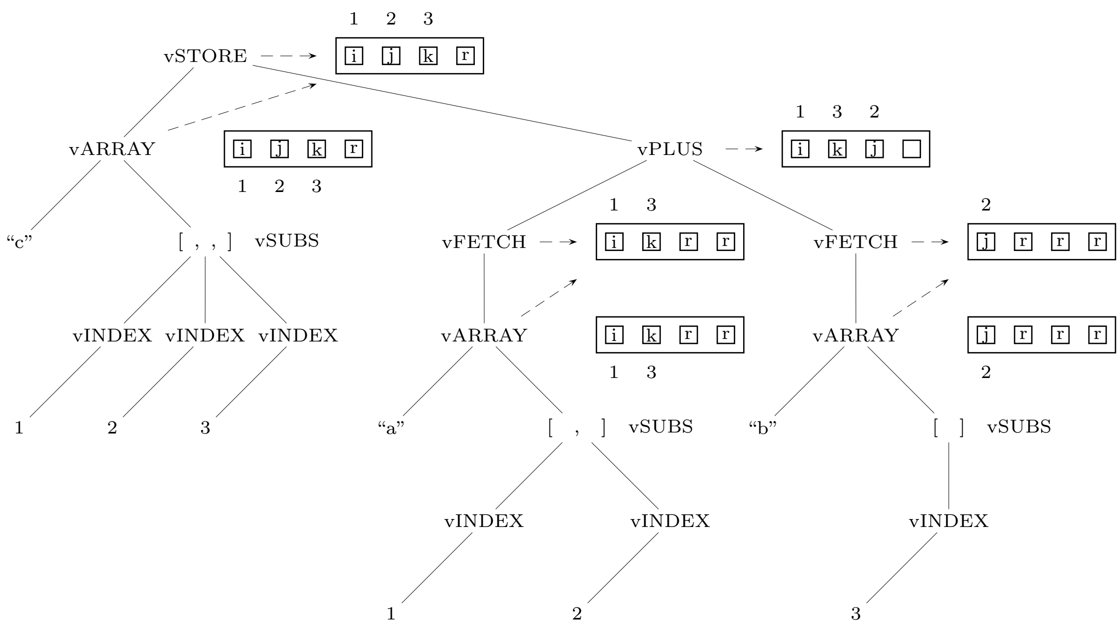
17.2 Examples with replicated data

17.2.1 Orthogonal replicated cells

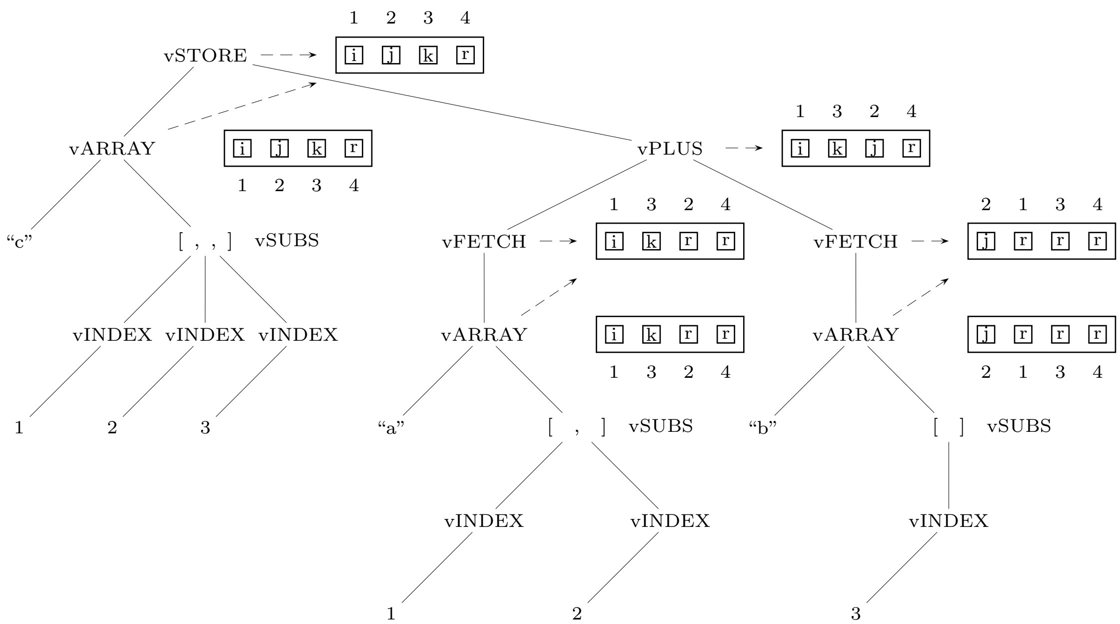
Here is an example which demonstrates the technique we use to handle replicated data:

```
!hpf$ template t(n, n, n, n)
!hpf$ distribute t(block, block, block, block)
      real a(n, n), b(n), c(n, n, n)
!hpf$ align a(l, m) with t(l, *, m, *)
!hpf$ align b(n) with t(*, n, *, *)
!hpf$ align c(l, m, n) with t(l, m, n, *)
      forall (i = 1:n, j = 1:n, k = 1:n)
         c(i, j, k) = a(i, k) + b(j)
```

The dotree looks like this:



where replicated cells are denoted by the letter r. The numbers next to the cells are the template indexes associated with primary subspace uses. The iteration subspace structure is determined by primary and replicated cell constraints. Now the DIVIDE phase can in effect synthesize the matching up of the rest of the cells (using the rules in Section 10.4), as follows:



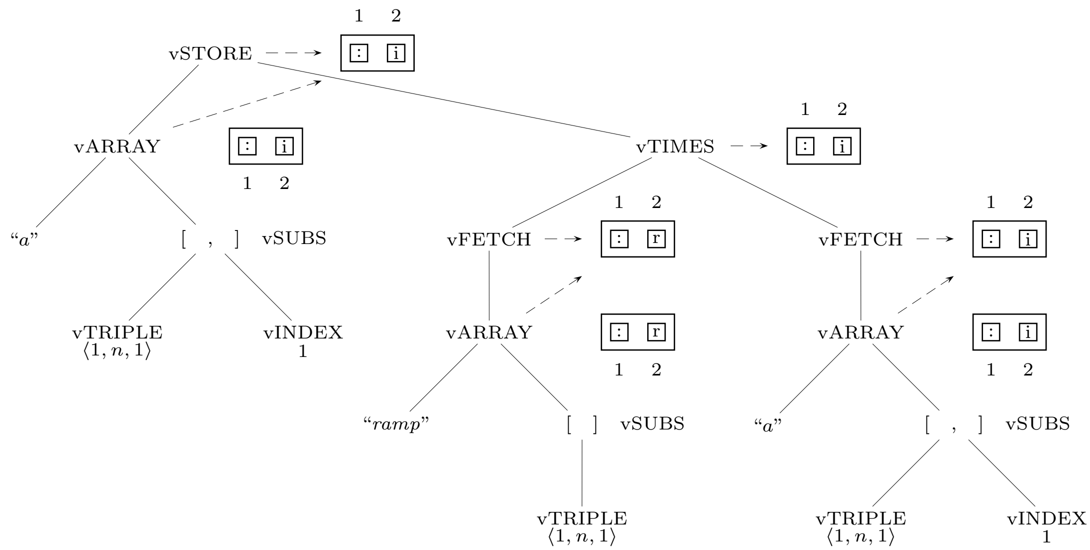
We see that in this statement no motion is necessary.

17.2.2 Globally replicated data

Here is a second code fragment:

```
!hpf$ template t(n, n)
      real a(n, n), ramp(n)
!hpf$ distribute t(*, block)
!hpf$ align a(l, m) with t(l, m)
!hpf$ align ramp(l) with t(l, *)
      do i = 1, n
         ramp(i) = float(i − 1)
      end do
      forall (i = 1:n)
         a(:, i) = ramp(:) * a(:, i)
```

In this code, the array *ramp* is replicated across all the processors. The dotree looks like this:



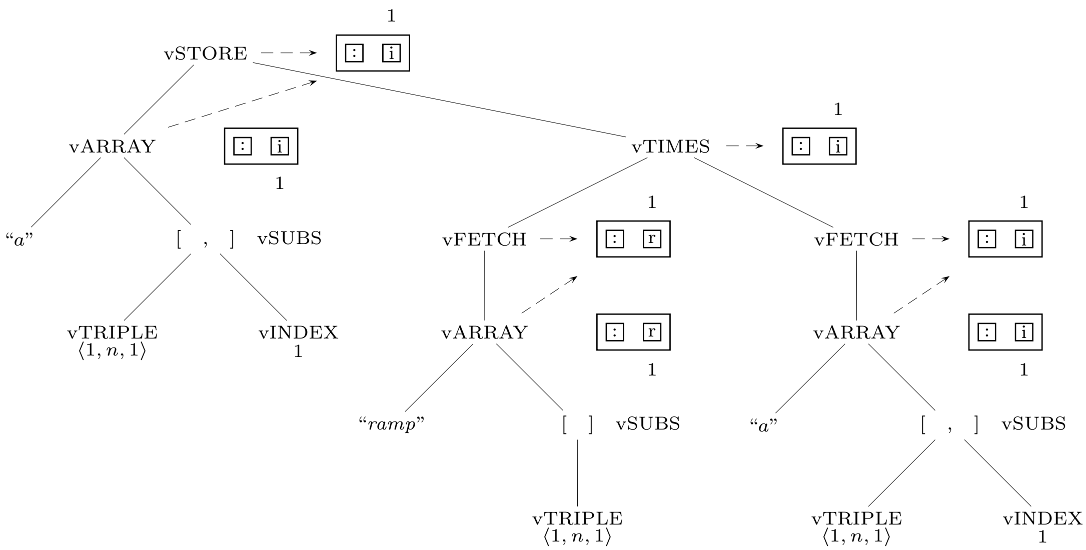
Again, the two children of the vTIMES node mesh perfectly, and there is no data motion.

If there were no alignment directive for the array *ramp*, then it would be distributed by default in the same manner anyway. The cell structure for the data and iteration spaces of the *ramp* array reference would look the same, except that the template name would not be *t*—it would be some other (default, compiler-generated) template. Nevertheless, the compiler would know that cell 2 (the replicated cell) represented all the processors, and that cell 1 had $strip\_length = 1$. Hence the algorithm again will generate the same iteration structure for the vTIMES node, and no data motion will be inserted.

The same code fragment could also be written as follows:

!hpf$ template $t(n)$
    real $a(n,\ n),\ ramp(n)$
!hpf$ distribute $t($**block**$)$
!hpf$ align $a(*,\ i)$ **with** $t(i)$
!hpf$ align $ramp(*)$ **with** $t(*)$
    do $i = 1,\ n$
        $ramp(i) = float(i-1)$
    end do
    forall $(i = 1{:}n)$
        $a(:,\ i) = ramp(:) * a(:,\ i)$

Now the dotree looks like this:



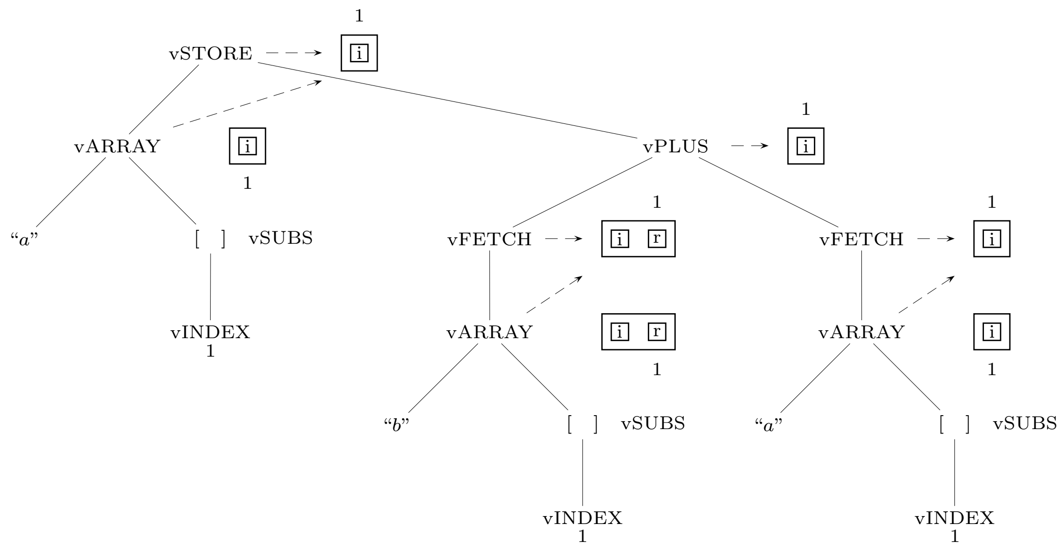
We see that no motion is necessary using this representation. Again, if no alignment directive had been given for the array *ramp*, the same considerations as outlined above would give the same iteration structure for the vTIMES node, again with no data motion inserted.

17.2.3 A disappearing primary cell

Here is a fragment which illustrates how a cell can disappear when iteration information is synthesized from children to parents:

!hpf$ template $t(n)$
  real $a(n)$, $b(n)$
!hpf$ distribute t(**block**)
!hpf$ align $a(i)$ **with** $t(i)$
!hpf$ align $b(*)$ **with** $t(*)$
  forall $(i = 1{:}n)$
    $a(i) = b(i) + a(i)$

The dotree looks like this:



Again, no motion is necessary.

17.2.4 Factoring a replicated cell

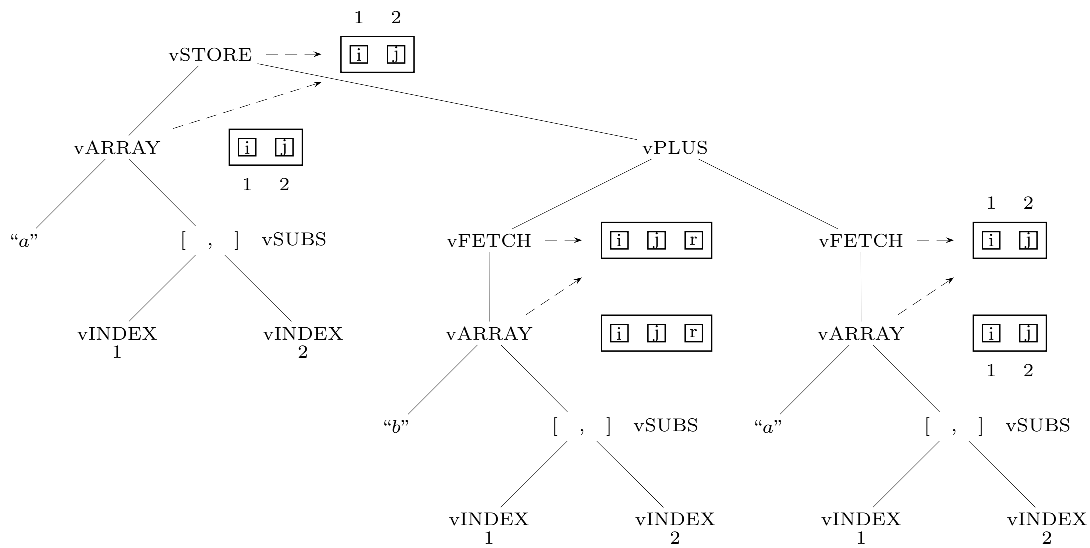
In this example, which is a variant of the previous one, the array b is implicitly replicated over all the processors, because of our definition of default data layout.

!hpf$ template $t(n,\ n)$
  real $a(n,\ n),\ b(n,\ n)$
!hpf$ distribute $t($**block**, **block**$)$
!hpf$ align $a(l,\ m)$ **with** $t(l,\ m)$
  forall $(i = 1{:}n,\ j = 1{:}n)$
    $a(i,\ j) = b(i,\ j) + a(i,\ j)$

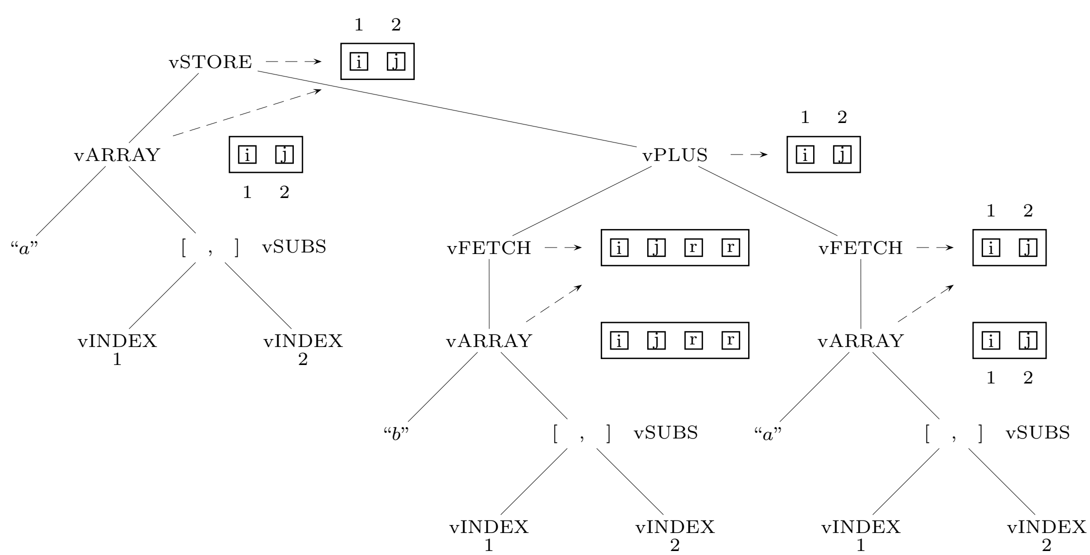
Note that even if we had a different definition of default data layout, we could achieve the same result by adding the following directives:

!hpf$ template $e(n,\ n,\ n)$
!hpf$ distribute $e(*,\ *,$ **block**$)$
!hpf$ align $b(l,\ m)$ **with** $e(l,\ m,\ *)$

The dotree looks initially like this:



Now the replicated cell of b does not have the same place as either of the cells of a. However, it can be factored into two cells which have those places (i.e. it can be thought of as the union of those two cells—cf. the algorithm on page 104). Hence we factor it into two replicated cells having those cell places, and the rest of the processing is similar to what happens in the previous example; the two primary cells of b with *strip_length* 1 go away. Again, no motion is necessary, and the dotree looks like this:



Chapter 18

Some Low Level Notes

18.1 Special operators

The strip mining formulas in Chapter 12 contain some expressions with if tests in them. There is no reason to expose this flow of control in the IR at this point; in fact it could only complicate the processing that the Transform phase has to do. For this reason, these expressions are abstracted into special operators which are then lowered separately in the Middle End. These operators are specified here:

```
vaUCOND_EXP:                     VC3    scalar    DefIntType

        child1:  bool_exp              scalar    dtLOG4
        child2:  true_exp              scalar    DefIntType
        child3:  false_exp             scalar    DefIntType

  if (bool_exp)
    return(true_exp);
  else
    return(false_exp);
```

This operator is used in constructing the loop bounds for a loop corresponding to an iteration dimension distributed BLOCK or CYCLIC(n). Such loop bounds are expressions whose value depends on whether the effective iteration stride is positive or negative. That test is expressed in `bool_exp`. This operator corresponds to the GEM operator SELU. That is, both child1 and child2 may be evaluated.

```
vaCOND_EXP:                      VC3    scalar    DefIntType

        child1:  bool_exp               scalar    dtLOG4
        child2:  true_exp               scalar    DefIntType
        child3:  false_exp              scalar    DefIntType

  if (bool_exp)
    return(true_exp);
  else
    return(false_exp);
```

This operator is the same as `vaUCOND_EXP`, except that it corresponds to the GEM operator SELC. That is, child1 must be evaluated first, and then based on its value, either child2 or child3 is evaluated. This operator is used to protect against division by 0. It is only used in Transform in guarding expressions that might involve zero-sized arrays or array sections. Such expressions can be suppressed by the `-assume nozsize` switch, which is also implied by the `-fast`, when the programmer knows that no array sections in the code have size 0; this greatly increases the efficiency of the generated code.

```
vaFLOOR_DIV:                    VC2     scalar    DefIntType

        child1:  numerator              scalar    DefIntType
        child2:  denominator            scalar    DefIntType
```

This expression evaluates to the floor of the fraction

```
            numerator/denominator
```

This is different from the operator vaOVER, which implements Fortran semantics: the fraction is always rounded towards 0. `vaFLOOR_DIV`, in contrast, always rounds down.

This operator is used throughout in implementing the strip mining formulas, wherever a floor is indicated in this report. It is also used in computing the value L_j used in nearest-neighbor computations, in Chapter 15.

```
vaCEIL_DIV:                 VC2     scalar    DefIntType

        child1:  numerator          scalar    DefIntType
        child2:  denominator        scalar    DefIntType
```

This expression evaluates to the ceiling of the fraction

```
numerator/denominator
```

This is similar to the operator `vaFLOOR_DIV`, except that it always rounds up.

This operator is used throughout in implementing the strip mining formulas, wherever a ceiling is indicated in this report.

The identity

$$\left\lceil \frac{a}{b} \right\rceil = \begin{cases} \left\lfloor \frac{a-1}{b} \right\rfloor + 1 & \text{if } b > 0 \\ \left\lfloor \frac{a+1}{b} \right\rfloor + 1 & \text{if } b < 0 \end{cases}$$

which is valid when a and b are integers, relates `vaFLOOR_DIV` and `vaCEIL_DIV`.

18.2 CSHIFT and EOSHIFT

Here we give the details of how CSHIFT and EOSHIFT are inlined in ARG. These functions are first brought up to the statement level (i.e. they are made the complete right-hand side of an assignment statement, if they are not already). Then the iteration space of that statement is normalized, as outlined in Section 10.7 (page 108). (If it is normalized again by DIVIDE, no harm is done, since normalization is idempotent.)

Then the statements are inlined as outlined below. In the following, all bounds refer to the normalized iteration bounds.

18.2.1 CSHIFT

The expression we have to lower is

$$RESULT = \mathbf{cshift}(\mathrm{ARRAY}, \mathrm{SHIFT}, \mathrm{DIM})$$

or, for simplicity in the following pseudocode,

$$RESULT = \mathbf{cshift}(A, \Delta, d)$$

We first perform the following preprocessing:

- If the DIM argument d is missing, it is taken to be 1.

We distinguish two cases. Case I is transformed and ultimately handled in terms of Case II.

Case I $N = \text{rank}(A) > 1$ *and* Δ is an array. We generate the following code. The notation $\langle I_d \rangle$ indicates a missing subscript (that is, in the particular array reference there are only $N-1$ subscripts, as in fact the array Δ has rank $N-1$):

do $I_1 = LB_1,\ UB_1$
  do $I_2 = LB_2,\ UB_2$
    ...   -- omitting I_d
    do $I_N = LB_N,\ UB_N$
      $RESULT(I_1,\ \ldots,\ LB_d : UB_d : S_d,\ \ldots,\ I_N)$
        $=$ **cshift**$(A(I_1,\ \ldots,\ LB_d : UB_d : S_d,\ \ldots,\ I_N),$
          $\Delta(I_1,\ \ldots,\ \langle I_d \rangle,\ \ldots,\ I_N),$
          $1)$
    end do
    ...
  end do
end do

Case II Δ is a scalar. $N = \text{rank}(A)$ may still be > 1, however. We generate the following code:

size $\leftarrow \dfrac{UB_d - LB_d}{S_d} + 1$
if (size = 0) **then**
  $T \leftarrow 0$
else
  $T \leftarrow$ **modulo** (Δ, size)
end if
-- We know $T \geq 0$
if $T = 0$ **then**
  $RESULT = A$
else -- $T > 0$
  begin
  $RESULT(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d : UB_d - T * S_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
    $\leftarrow A(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d + T * S_d : UB_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
  $RESULT(LB_1 : UB_1 : S_1,\ \ldots,\ UB_d - (T-1) * S_d : UB_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
    $\leftarrow A(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d : LB_d + (T-1) * S_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
  end
end if

The initial computation of T is simplified in the usual case when the compiler is told on the command line to assume that all array sections in the program have size > 0. In this case, it just becomes

$$\text{size} \leftarrow \frac{UB_d - LB_d}{S_d} + 1$$
$$T \leftarrow \textbf{modulo}\,(\Delta, \text{size})$$

18.2.2 EOSHIFT

Here the expression is

$$RESULT = \textbf{eoshift}(\text{ARRAY}, \text{SHIFT}, \text{BOUNDARY}, \text{DIM})$$

or, for simplicity in the following pseudocode,

$$RESULT = \textbf{eoshift}(A, \Delta, B, d)$$

We first perform the following preprocessing (virtually the same as that for CSHIFT):

- If the DIM argument d is missing, it is taken to be 1.
- If the rank of A (call it N) is > 1, then the BOUNDARY argument B can either be a scalar or an array of rank $N - 1$. If it is a scalar, it is treated as if it were an array of rank $N - 1$ each of whose elements has value B.
- If the BOUNDARY argument B is missing, then its shape is defined as above, and its value is taken to be some version of 0, as specified in the Fortran Language Standard.

Again, we distinguish two cases. Case I is transformed and ultimately handled in terms of Case II.

Case I $N = \text{rank}(A) > 1$ *and* Δ is an array. We generate the following code:

$$
\begin{array}{l}
\textbf{do } I_1 = LB_1,\ UB_1 \\
\quad \textbf{do } I_2 = LB_2,\ UB_2 \\
\qquad \ldots \qquad\qquad \texttt{--}\ \text{omitting } I_d \\
\qquad \textbf{do } I_N = LB_N,\ UB_N \\
\qquad\quad RESULT(I_1,\ \ldots,\ LB_d\!:UB_d\!:S_d,\ \ldots,\ I_N) \\
\qquad\qquad = \textbf{eoshift}(A(I_1,\ \ldots,\ LB_d\!:UB_d\!:S_d,\ \ldots,\ I_N), \\
\qquad\qquad\qquad\qquad \Delta(I_1,\ \ldots,\ \langle I_d\rangle,\ \ldots,\ I_N), \\
\qquad\qquad\qquad\qquad B(I_1,\ \ldots,\ \langle I_d\rangle,\ \ldots,\ I_N), \\
\qquad\qquad\qquad\qquad 1) \\
\qquad \textbf{end do} \\
\qquad \ldots \\
\quad \textbf{end do} \\
\textbf{end do}
\end{array}
$$

Case II Δ is a scalar. $N = \text{rank}(A)$ may still be > 1, however, and B may be an array. In line with the "preprocessing" outlined above, we will assume that B is in fact an array, for the purposes of this pseudocode. We generate the following code:

size $\leftarrow \dfrac{UB_d - LB_d}{S_d} + 1$
if ($\Delta = 0$ **or** size $= 0$) **then**
    $RESULT = A$
else if $|\Delta * S_d| > |UB_d - LB_d|$ **then**
    $RESULT(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d : UB_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
        $\leftarrow$ **spread**$(B(LB_1 : UB_1 : S_1,\ \ldots,\ \langle LB_d : UB_d : S_d\rangle,\ LB_N : UB_N : S_N),$
                dim $= D$, **ncopies** $= |UB_d - LB_d| + 1)$
else if $\Delta > 0$ **then**
    begin
    $RESULT(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d : UB_d - \Delta * S_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
        $\leftarrow A(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d + \Delta * S_d : UB_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
    $RESULT(LB_1 : UB_1 : S_1,\ \ldots,\ UB_d - (\Delta - 1) * S_d : UB_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
        $\leftarrow$ **spread**$(B(LB_1 : UB_1 : S_1,\ \ldots,\ \langle LB_d : UB_d : S_d\rangle,\ \ldots LB_N : UB_N : S_N),$
                dim $= D$, **ncopies** $= \Delta)$
    end
else -- $\Delta < 0$
    begin
    $RESULT(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d - \Delta * S_d : UB_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
        $\leftarrow A(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d : UB_d + \Delta * S_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
    $RESULT(LB_1 : UB_1 : S_1,\ \ldots,\ LB_d : LB_d - (\Delta + 1) * S_d : S_d,\ \ldots,\ LB_N : UB_N : S_N)$
        $\leftarrow$ **spread**$(B(LB_1 : UB_1 : S_1,\ \ldots,\ \langle LB_d : UB_d : S_d\rangle,\ \ldots LB_N : UB_N : S_N),$
                dim $= D$, **ncopies** $= -\Delta)$
    end
end if

Bibliography

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L Wagener. *Fortran 90 Handbook*. McGraw-Hill Book Company, New York, 1992.

[2] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*. ACM SIGPLAN, July 1988.

[3] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, 1992.

[4] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.1, November 1999. Available at `http://www.openmp.org/specs`.

[5] Digital Equipment Corporation. *DEC Fortran 90 Language Reference Manual*. Maynard, Massachusetts, 1994.

[6] International Organization for Standardization and International Electrotechnical Commission. *International Standard Programming Language Fortran*. December 1997. ISO/IEC 1539-1:1997.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 2.0, January 1997. Available from the Center for Research on Parallel Computation, Rice University, Houston, TX, and at `http://www.crpc.rice.edu/HPFF`.

[8] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Massively Parallel Data Optimization. In *Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation*, George Mason University, October 1988. IEEE.

[9] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[10] Kathleen Knobe and Venkataraman K. Natarajan. Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD Machines. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, University of Maryland, October 1990. IEEE.

[11] Pierre L'Ecuyer. Efficient and portable random number generators. *Communications of the ACM*, 31(6):742–749, 774, 1988.

[12] Diane Meirowitz. DTB-II Reference Manual. Report CAID-9010-0902, Massachusetts Computer Associates, Inc., Wakefield, MA, October 1990.

[13] C. Robert Morgan and Paul St. Pierre. LDO Language Specification. Report CAID-8608-0401, Massachusetts Computer Associates, Inc., Wakefield, MA, August 1986.

[14] Carl D. Offner. A Data Structure for Managing Parallel Operations. In *Proceedings of the 27th Hawaii International Conference on System Sciences. Volume II: Software Technology*, pages 33–42. IEEE Computer Society Press, January 1994. Available at `http://www.cs.umb.edu/~offner`.

[15] Carl D. Offner. Modern Dependence Testing. DEC Internal Report, 1996.

[16] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

Index