TStreams: How to Write a Parallel Program

Kathleen Knobe, Carl D. Offner
Cambridge Research Laboratory
HP Laboratories Cambridge
HPL-2004-193
October 27, 2004*

E-mail: kath.knobe@hp.com, carl.offner@hp.com

TStreams, parallelism

TStreams is a simple yet powerful parallel programming model which separates the expression of all potential parallelism in an application both from the serial computations and also from the target-specific details such as mapping and scheduling needed to run an application on a particular architecture.

This separation leads to an efficient and modular way of developing parallel applications. TStreams programs are able to make use of all kinds of parallelism-TStreams is not prejudiced in favor of one particular kind, as parallel programming languages typically are. And the code in a TStreams application does not have to be rewritten when porting TStreams from one architecture to another.

In addition, because TStreams is more general than other parallel programming systems, it has the capability of automatically providing some facilities, such as automatic checkpoint/restart, that are not even usually thought of as issues of parallel programming.

* Internal Accession Date Only
Approved for External Publication

TStreams: How to Write a Parallel Program

Kathleen Knobe
HP Cambridge Research Lab
kath.knobe@hp.com

Carl D. Offner
HP Cambridge Research Lab
carl.offner@hp.com

Abstract

TStreams is a simple yet powerful parallel programming model which separates the expression of all potential parallelism in an application both from the serial computations and also from the target-specific details such as mapping and scheduling needed to run an application on a particular architecture.

This separation leads to an efficient and modular way of developing parallel applications. TStreams programs are able to make use of all kinds of parallelism—TStreams is not prejudiced in favor of one particular kind, as parallel programming languages typically are. And the code in a TStreams application does not have to be rewritten when porting TStreams from one architecture to another.

In addition, because TStreams is more general than other parallel programming systems, it has the capability of automatically providing some facilities, such as automatic checkpoint/restart, that are not even usually thought of as issues of parallel programming.

1 Introduction

TStreams is a parallel programming model which is simple yet powerful. As a consequence, the model directly supports a number of important capabilities. For instance,

- TStreams code can be used without modification for both serial and parallel target platforms.
- Checkpoint/restart is a command-line option and requires no code modification.
- The decisions about parallelism, distribution, and scheduling can be made either before execution begins, minimizing runtime overhead, or during execution, based on more accurate information.

The reason that TStreams has these capabilities is that TStreams is more general than other parallel programming models.

TStreams is a programming model that enables the programmer to express all the potential parallelism in an application. We mean by "potential" parallelism all types of parallelism available in the application, without regard to the target architecture.

There are three parts to a TStreams program:

parallel algorithm part: (We sometimes refer to this as the *parallel algorithm level*.) This part describes the TStreams objects (items, steps, and tags, which are all defined below) and the relations between them. It is written in the TStreams language, which we illustrate in Section 2.4. It expresses all the potential parallelism in the program. The parallel algorithm part is the topic of Section 2.

serial part: (We sometimes refer to this as the *serial level*.) This part describes the internals of the TStreams objects and expresses the non-parallel computations and data structures in the program. It is written in any conventional programming language. The serial part is almost completely standard, so aside from explaining what it is in Section 2, we do not discuss it in depth.

mapping part: This part describes how the potential parallelism in the algorithm is implemented on a particular target. This involves scheduling and distribution. The mapping part is the topic of Section 3.

Our thesis is this: all current parallel programming languages, such as MPI [3], OpenMP [7], HPF [4], and streaming languages [2], focus on the mapping part. Each of these languages was developed with a particular target architecture in mind. This causes each of these languages to naturally express certain kinds of parallelism well (for instance, data parallelism or task parallelism) and other kinds (for instance, pipelined parallelism) poorly or not at all.

By solving the general problem of expressing parallel applications in a target-independent manner, TStreams can do a better job than each such language, even on applications that are usually regarded as natural for that language. And as an added benefit of this more general viewpoint, TStreams leads to new ways of dealing with situations not normally thought of as issues of parallelism. We will describe some of these benefits of being general in Section 4.

2 The parallel algorithm part

2.1 Basic TStreams Objects

We introduce the fundamentals of the basic TStreams objects by a very simple program: building a binary tree, as in Figure 1. (The description of these objects is fleshed out in more detail in Section 2.2.) For simplicity, we assume that the tree has a fixed depth n where n is known at the start of execution. We will discuss a more realistic example later.

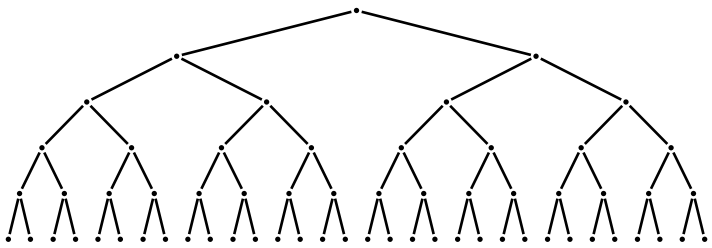

Figure 1: A binary tree, of depth 5.

There are three kinds of TStreams objects:

items encapsulate data. Each item has

- *contents*, which is the data that the item holds, and
- an associated *tag*, which serves to uniquely identify the item.

Conceptually at least, each item is immutable.

steps encapsulate computations. Each step has

- *code*, written in some standard programming language, and
- an associated *tag*, which serves to uniquely identify the step.

Conceptually at least, each step executes atomically.

tags encapsulate unique identifiers for both steps and items. Each tag has a *value*.

In this example there is an item and a step corresponding to each node in the tree. The item holds the data associated with the node. The step corresponding to a node inputs the item corresponding to the same node and produces two items, one corresponding to each of its children. If the node is a terminal node, the step does nothing. In addition, there is a tag associated with each node that simply identifies the node.

In our example, the precise form of the value of the tag is not relevant. One possibility is that the value of the tag of a node is a pair $\langle gen, num \rangle$, where *gen* is the generation of the node and *num* identifies the node within its generation. Another is that the tag is a binary string of length $\leq n$, representing a path starting from the root where each bit indicates one of the two children. Either of these is fine.

2.2 Spaces and Relations

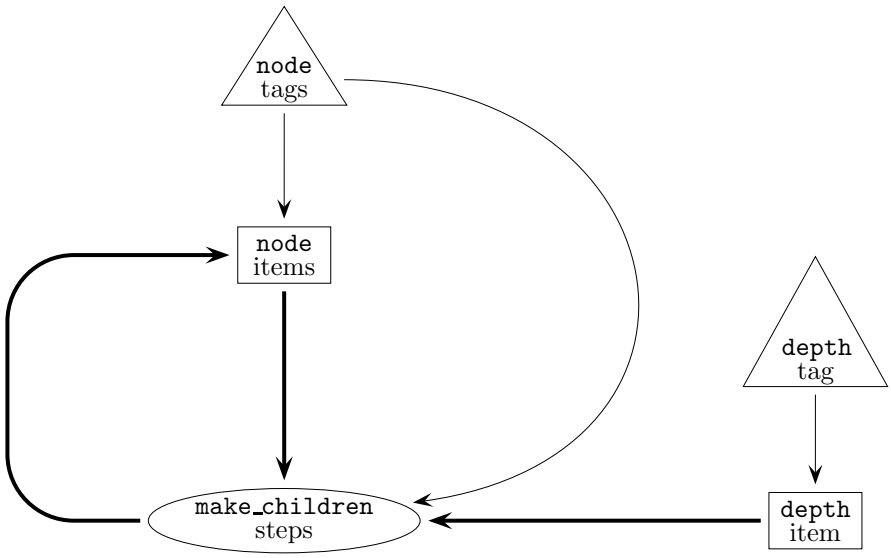

Figure 2: Representation of a TStreams program at the parallel algorithm level. The program builds a binary tree, starting from the root.

Figure 2 is a graphical representation of this program at the parallel algorithm level. In such a representation, we do not see tags, items, or steps individually. What we see are *tag spaces*, *item spaces* and *step spaces*. Each item space is a collection of items that are naturally related, and similarly for step spaces and tag spaces. In a little more detail, we have the following spaces:

Item spaces: An item space has a name and represents a collection of individual items that are natural to view as an single larger entity. The internal structure of the contents of these items, however, is not visible at the parallel algorithm level.

An item space is represented in the graphical form as a rectangle. In our example:

- The space of `node` items contains one item for each node of the tree.
- The space of the `depth` item contains the single item containing the fixed tree depth.

Step spaces: A step space has a name and represents a collection of steps that share the same code. The actual step code, however, is not visible at the parallel algorithm level. The code of a step can make reference to the value of the step's tag.

A step space is represented in the graphical form as an oval. In our example:

- The space of `make_children` steps contains one such step for each node in the tree.

Tag spaces: A tag space has a name and represents a collection of individual tags. The individual tags in a tag space identify individual items in an item space, steps in a step space or both. The tags might correspond to subscripts that identify a subsection of an array or a point in the iteration space of a loop nest. Alternatively, they might correspond to pointers or database keys. They might in fact be anything that makes sense in the application. If the tag space only contains a single tag, (e.g., the `depth` tag in our example), the actual value of the tag might not be significant since it is not used to distinguish one item from another.

- The `node` tag space contains one tag for each node in the tree. Each such tag corresponds to both an item and a step.
- The `depth` tag space contains only one tag. This tag corresponds only to an item.

A TStreams program specifies certain binary relations between these item, step, and tag spaces:

Prescriptive relations: A prescriptive relation is one from a tag space to either an item space or a step space. This relation is an assertion that at the end of the program each available tag in the tag space will correspond to an available item in the item space or, respectively, to an executed step in the step space. (The term *available* is defined precisely below.) The tag space thus *prescribes* which items in the item space must ultimately become available, and also which steps in the step space must ultimately execute. The value of the tag in the tag space is the value of the tag of each step and item it prescribes. A tag space may prescribe more than one item space, and more than one step space.

The prescriptive relation is indicated by a thin arrow in the graphical form.

- The `node` tag space prescribes the `node` item space.

  When the program finishes execution the `node` item space will contain a `node` item corresponding to each tag in the `node` tag space.
- The `node` tag space prescribes the `make_children` step space.

  When the program finishes execution, one `make_children` step will have executed for each tag in the `node` tag space.
- The `depth` tag space prescribes the `depth` item space.

Producer and consumer relations: Producer and consumer relations are represented by thick arrows in the graphical form.

A **producer relation** associates an object with the step that produces it. There is one producer relation in our example:

- from the `make_children` step space to the `node` item space.

In this example, there is no step that produces a tag; we will see an example for that later.

A **consumer relation** indicates which items are consumed by which steps. There are two consumer relations:

- from the `depth` item space to the `make_children` step space, and
- from the `node` item space to the `make_children` step space.

Associated with each consumer relation is a function that converts the tag of the step to the tag of the item consumed. This function must be computable, must be a function only of the step tag (so in particular it cannot use program data), and must have no side effects. In our tree-building example, this function is simply the identity, but more complex functions are also common.

We can't always associate such a function with a producer relation on the other hand, because the tag of an item produced by the step may depend on the contents of the items consumed by that step. In other words, the execution of the step itself defines the tags of its output.

Between a step and an item either a producer or a consumer relation might exist. Between a tag and an item only a prescriptive relation can exist. Between a step and a tag either a prescriptive or a producer relation might exist. (In this example, there are no steps that produce tags, but we will show such a program below.)

Notice that it makes no sense to say that there is a consumer relation between a step and a tag. A consumer relation represents the action of a step receiving information external to itself. A step consumes an item by computing the value of the tag of that item and then receiving the contents of that item. But a tag itself contains nothing but its own value, so if a step were to compute that value, there would be nothing more to receive.

All a step can do is use its tag, consume items, perform computations on all this data, and produce other items and/or tags.

In addition to the producer and consumer relations described above, some items and tags are available at program start-up from the environment. These are called *initial* items and tags. Some items and tags are made available to the environment upon termination. These are called *terminal* items and tags. We consider the relationship between these items (or tags) and the environment as a producer or consumer relations. In this example, the depth tag, depth item, the node tags and the single node item corresponding to the root of the tree are initial and are considered to be "produced" by the environment. All the node items are terminal and are considered "consumed" by the environment.

As a matter of etymology, the name TStreams stands for "tagged streams". The word "stream" is used to describe a collection of data objects produced by one computation and used by another. In particular, the term "stream" has sometimes been used to describe a model implementing strict FIFO ordering, such as in some versions of media processing [8] or as in stream constructs in some functional languages [1, 6]. In other cases the term "stream" is used to describe a model in which the items can be processed in an arbitrary order [2]. TStreams describes a third model. In TStreams, there may be constraints on the ordering; they are not arbitrary; and they reflect the producer and consumer relations in the program itself which are expressed in terms of tags and may become apparent only as the program unfolds.

2.3 Generating Tags

In the static tree building program presented above, the tags for the nodes (i.e., the tags in the `node` tag space) are initial tags. We know the entire shape of the tree to be constructed before we begin executing. But in most tree-based applications this is not true. In a general dynamically constructed tree, part of the computation at a node must determine if this node is a leaf or needs to be further subdivided.

This is not a major change from the previous example as far as the representation at the parallel algorithm level is concerned. Figure 3 shows that representation for a dynamic tree builder. If the step in fact actually does make children, the `node` tags of the children are produced by the step itself. Notice that in the static version the `node` tag space was an initial tag space. In this new version, however, the tag space that prescribes the nodes starts with one tag (corresponding to the root node) and grows as the program executes. We can only determine if a certain step will execute after the step associated with its parent has completed.

The previous example only contained producer relations between steps and items. In the dynamic tree builder we have in addition a producer relation between steps and tags.

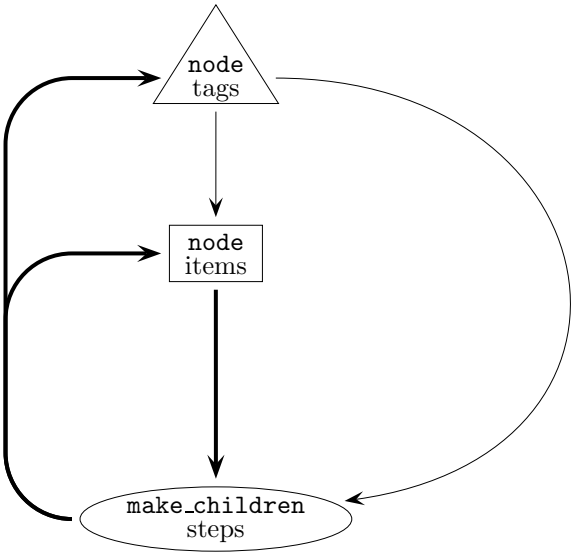

Figure 3: Representation of a TStreams program at the parallel algorithm level. The program builds a tree whose full extent is known only at run-time.

2.4 Syntax

Here we briefly present a lexical (rather than graphical) representation of the parallel algorithm part of a TStreams program. Figure 4 shows how the spaces and relations in the TStreams version of the static tree builder are represented.

A symbol in angle brackets, like ⟨`node`⟩, represents a tag space. A symbol in square brackets, like `[depth]`, represents an item space. A symbol in parentheses, like `(make_children)`, represents a step space.

Prescription relations, as in the first three lines in Figure 4, are represented by double colons. The first such relation is between a tag space and a step space; the remaining two are between tag spaces and item spaces. Producer and consumer relations, as in the remaining three lines, are represented by arrow. The first two of these lines denote consumer relations and the last denotes a producer relation.

While ⟨`node`⟩ represents a tag space, we can represent an element in that space by giving its explicit value, like this: ⟨`node:0`⟩; or by giving the value a symbolic name, like this: ⟨`node:root`⟩. An item can be specified by giving its associated tag value: [node]⟨`0`⟩, or `[node]`⟨`root`⟩.

$$\langle \texttt{node} \rangle :: \texttt{(make\_children)}$$
$$\langle \texttt{node} \rangle :: \texttt{[node]}$$
$$\langle \texttt{depth} \rangle :: \texttt{[depth]}$$

$$\texttt{[depth]} \rightarrow \texttt{(make\_children)}$$
$$\texttt{[node]} \rightarrow \texttt{(make\_children)}$$
$$\texttt{(make\_children)} \rightarrow \texttt{[node]}$$

Figure 4: Static tree-builder: spaces and relations. The notation is intended to remind one of the graphical representation: Angle brackets (like triangles) identify tag spaces; square brackets (like rectangles) identify item spaces; ordinary parentheses (like ovals) identify step spaces.

Without fleshing out this language in more detail, it is evident that the parallel algorithm part of a TStreams program can be represented compactly and clearly in this manner.

2.5 Characteristics of TStreams

Here are some characteristics of TStreams that taken together give it a unique flavor different from other parallel programming models.

- Steps are (conceptually) atomic. In addition, steps are functional—the items and/or tags produced by the step depend only on the value of the tag of the step and the contents of the input items consumed by the step.
- Items are immutable: The contents of an item can never be changed. Therefore, one never has to be concerned about using an item before it is overwritten; it will never be overwritten. This allows the programmer, compiler, and run-time system a lot of freedom in reordering computations.

  As a consequence of these two properties of steps and items, a step whose execution has been interrupted, say by a machine failure, can simply be restarted. Since items are immutable the step's inputs will not have changed, and since steps are functional the restarted step will necessarily produce the same outputs.

  As we will note below in Section 4, this property makes it possible to implement checkpoint/restart in a TStreams program without any instrumentation of the program code.

- Tags are used to identify both steps and items.

  - In their role as prescribers of steps, they help to determine control flow—they take the place of iteration spaces (e.g. loop nests) and conditional execution in conventional programs.

    Although the tags which prescribe steps are used in determining control flow, tags themselves simply indicate what steps will get executed. They do not specify in what order execution of those steps must occur. The only ordering constraints on step execution are those inferred from the producer and consumer relations.
  - In their role as identifiers of items, they serve to describe the shape of an item space. For instance, a 3-dimensional array might be described as a space of 1-element items whose associated tags have 3-dimensional values. In analogy with the term *iteration space*, this is what is sometimes referred to as *data space*.

  In most other languages data and iteration spaces are handled by separate mechanisms. In TStreams, tags and tag spaces handle these in a unified manner.

- The multi-part structure of a TStreams program that isolates the parallel algorithm level from the underlying serial level decouples steps from each other and decouples items from each other.

  Steps do not use mechanisms in the underlying serial language to refer to other steps. The serial code within a step may invoke procedures within the same step. However, a step may not invoke another step.

  Items do not use mechanisms in the underlying serial language to refer to other items. An item may contain a pointer to data within the same item, but not to another item or part of another item.

- TStreams programs are declarative. For instance, permuting the lines in Figure 4 does not change the meaning of the program.

- Many programs that we think of as very different are identical at the parallel algorithm level, and in fact have the same potential parallelism. For example, any top-down tree walk looks the same at the parallel algorithm level, regardless of the nature of the computations in the steps and the data structures in the items. The fact that all these programs look the same at the parallel algorithm level means precisely that they all have the same potential parallelism.

- A step may represent an encapsulated computation that can be used in different TStreams programs. In this way we could regard steps as primitive operators in a domain-specific language developed for reuse in related programs.

2.6 The abstract operational model

Here we present an abstract operational model that describes the execution of a TStreams program. This model is consistent with the TStreams semantics, which is formally specified in a separate document [5].

As a TStreams program executes, each object acquires attributes. (We will specify these attributes immediately below; but note first that we do not regard the contents of an item, or the value of a tag, as attributes. Items and tags are immutable; their attributes can change.)

- Tags have one attribute:

  available A tag is *available* if it is either an initial tag or it has been produced by the execution of a step.

- Items have two attributes:

  prescribed Suppose `[I]` is an item space which is prescribed by a tag space ⟨`T`⟩. An item in `[I]` whose tag has value `v` is *prescribed* if there is an *available* tag in ⟨`T`⟩ with the same value.

  available An item is *available* if it is either an initial item or it has been produced by the execution of a step.

- Steps have the following attributes:

  prescribed Suppose `(S)` is a step space which is prescribed by a tag space ⟨`T`⟩. A step in `(S)` whose tag has value `v` is *prescribed* if there is an *available* tag ⟨`T`⟩ with the same value.

  inputs-available A step is *inputs-available* if all its input items (i.e., all the items that the step needs to consume, as determined by the consumer relation function) are *available*.

  executed The step has been executed.

We say that a step having the attributes *prescribed* and *inputs-available* is *enabled*.

TStreams objects acquire attributes as the program executes. There is only one restriction on the order in which these attributes may be acquired: A step may not be *executed* until it is *enabled*.

Now here is how these attributes evolve during the execution of our static tree-building program (Figure 2): At program start-up,

- The single ⟨`depth`⟩ tag is *available*.
- The single `[depth]` item (whose tag is the ⟨`depth`⟩ tag) is *available*.

- All the ⟨`node`⟩ tags are *available*.
- The root `[node]` item (whose tag is the root ⟨`node`⟩ tag) is *available*.

Given this initial state, without executing any steps, we know that

- The single `[depth]` item (whose tag is the ⟨`depth`⟩ tag) is *prescribed* because its tag is *available*.
- All the `[node]` items and all the `(make_children)` steps are *prescribed* because all the ⟨`node`⟩ tags are *available*.
- The `(make_children)` step whose tag is the root ⟨`node`⟩ tag is *enabled*, because
  - it is *prescribed*, and
  - it is *inputs-available* because its inputs (the root `[node]` item and the single `[depth]` item) are *available*.

This *enabled* step can therefore be executed. As a result of this step execution, the following things happen.

- The step becomes *executed*.
- The two new `[node]` items produced by the step become *available*

For the same reasons as above, the two steps that will input the two new `[node]` items are now *enabled*.

When the program reaches a point at which no steps are enabled, it *terminates*. A program is *valid* if upon termination

- every prescribed item is available, and
- every prescribed step has been executed.

It turns out that this property of being valid is a property of the program at the parallel algorithm level, and not of any decisions that are made at the mapping level, such as scheduling enabled steps. In particular, in a valid program the order of execution is irrelevant in the sense that at any time during the execution of the program if more than one step is enabled any one of those steps may be chosen to execute next without altering the final outcome. In fact several of them, or even all of them, may execute in parallel. Regardless of the ordering, by the end of the program, the same steps will have executed, and the same items and tags will have been produced. In our example, for instance, we might execute the steps in depth-first or breadth-first order, or in some other order. The parallel algorithm part of the TStreams program does not itself specify

a particular ordering. The producer and consumer relations, however, do put constraints on the order. Thus, in our tree builder, the tree must be built top-down. A bottom-up tree builder would constitute a fundamentally different algorithm.

The class of programs that are valid is the class of *well-formed* programs, as defined in [5].

3 The mapping part

In this section we discuss the decisions that need to be made at the mapping level.

However the decisions are made, we always assume serial execution of the code within the steps. We are not concerned here with how steps are executed.

Here are the possibilities available to us:

- Scheduling of steps within each processing nodes can be static or dynamic.
- Mapping of items to memory can be static or dynamic.
- Distributing steps across processing nodes can be static or dynamic.

When we say that a decision is statically determined, we mean that the decision is made before the program starts execution. That is, it is made by the programmer or by static analysis techniques,or by some combination of these two.

(Static analysis techniques require access to the consumer relation functions. However, such techniques do not require access to the code associated with the steps.)

In general, before execution begins we don't know all the steps to be executed or all the items that will be available. So a static decision simply means that we have a statically known function that implements that decision.

A static decision typically incurs less run-time overhead. Statically computed decisions are ideal when the both the computation and the system are predictable. The computation is predictable when the amount of computation and the communication needed in our program does not depend on the data. The system is predictable when the number of available processors in the target machine is constant, the system is dedicated and the system is not faulty.

A dynamic decision typically incurs some overhead cost at runtime. But if the situation is not predictable, dynamic decisions have the advantage of operating when more knowledge about the computation or the system is available. Dynamic decisions are justified when this knowledge can be used effectively.

Scheduling steps If the steps are scheduled dynamically, a runtime system maintains the state for each object as described in Section 2.6. It chooses among enabled steps for the next one to execute. If on the other hand the steps are scheduled statically, the static schedule must ensure that the steps are enabled when they execute.

Mapping of items to memory Although conceptually TStreams items are immutable, any efficient implementation will reuse memory by storing distinct items in the same location. Let us say that an item is *dead* if

- the item is not a terminal item, and
- all steps that consume that item have executed.

Whether the mapping of items to memory is determined statically (i.e., in the program itself) or dynamically by a run-time system, we must ensure that an item is dead before its memory is reused.

Distributing steps The distribution decision will assign each step to a processor and each item to possibly multiple memories. The goal of distribution is to optimize the load balance, minimize the communication and hide latency.

In this paper we have assumed that the grain of computation and data (i.e., of steps and items) is fixed. We have not discussed a hierarchical version of TStreams, capable of accommodating varying grain sizes. We are currently investigating this. In such a hierarchical variant, we again have the possibility of static or dynamic grain choice.

We currently have two prototypes. One operates with all decisions made statically. The other has a static distribution of objects across processors but uses dynamic scheduling and dynamic memory mapping. Both prototypes are written in C++. Their underlying run-time systems use MPI for interprocessor communication.

4 The benefits of being general

With appropriate mappings, a TStreams program can execute efficiently on a wide range of architectures: SIMD, MIMD, vector, shared and distributed memory, and streaming architectures, without the restrictions that typically are associated with them, such as those embodied in SPMD and strictly hierarchical fork-join constructs.

Rather than describing this in more detail, we turn to some unexpected benefits we get from the fact that TStreams represents parallelism in a more general manner than do other languages. These benefits are all ultimately due to the approach we have taken of isolating the parallel, serial, and mapping parts of each application. Here are some examples:

- We have a large application composed of several parts, some of which are regular, analyzable, and appropriate for static mapping, while other parts are data dependent or simply harder to analyze. We can execute such a program in a hybrid manner: we execute the analyzable parts with a highly tuned static mapping, and the unanalyzable parts dynamically, delaying decisions until more information is available.

- We have a low priority TStreams job sharing resources with high priority jobs. We can make use of idle processing power as follows: When a processor becomes idle, we speculate that the processor has time to complete an enabled step of the low priority job. If it completes, we have advanced the state of the low priority job. If higher priority work arrives before the low priority step completes, we can simply drop the low priority step. Because of the functional nature of the step, no harm is done; the step can be rerun at a later time, and the high priority work is not delayed.

- We have a TStreams program on a faulty parallel system. As each item or tag is produced, it is saved in a checkpoint store. When all the consumers of an item have executed and their outputs have been saved in the store, that item can be deleted from the store. Thus, the checkpoint store holds a frontier of the computation. To restart an application after a fault, simply inject each stored object as if it were just produced. This enables us to execute a TStreams program in a way that provides all the facilities of checkpoint/restart without any change to the program itself.

  This checkpoint/restart capability is implemented in our dynamically-scheduled prototype. In particular, this checkpointing capability does not depend on a deterministic frontier of computation.

- We have a VLIW target. The compilation of the code for each step space can of course be compiled to use instruction-level parallelism within each step. For some programs however we might be able to use instruction-level parallelism between steps, even for unanalyzable programs. Assume we have two step spaces, `(s1)` and `(s2)`, each of which has many instances. Assume we do not know statically which instances can be executed concurrently. We create a routine named `s1-s2` that executes an instance of `(s1)` and an instance of `(s2)`. We compile this routine for the VLIW target assuming that there are no ordering restrictions between the two step instances—that is, we are free to mix instructions from `(s1)` and instructions from `(s2)` within a single wide-word instruction. Now during program execution, if step `(s1)`$\langle 4 \rangle$ and step `(s2)`$\langle 25 \rangle$ are both enabled, we know that there are no dependences among the instructions of these two steps. Therefore the runtime system can execute the routine `s1-s2` with access to the tags and the input items of both steps. The state of the running program is maintained just as if the two separate steps were executed.

5 Conclusion

We have introduced, TStreams, a new model of parallel computation. Its goals are

- to support all types of parallelism,
- to support a wide variety of parallel architectures,
- to support a wide variety of execution models, and
- to isolate the distinct aspects of parallel programming.

A TStreams program is divided into three parts:

parallel algorithm part: This represents the abstract parallel algorithm.

serial part: This contains the (serial) code and data structures in the steps and items.

mapping part: This contains the details of the distribution, mapping and scheduling of steps and items.

Isolating the program into three parts in this way allows for separate development of

- a collection of abstract parallel algorithms.
- a collection of high level primitive operations (steps), and data structures (items) for some application domain, implemented in a serial language.
- a collection of mappings and runtime systems for specific environments.

The development of a new application then involves starting with these collections, adding to them and choosing among the available possibilities. Tuning involves adjusting any of these choices.

This way of developing parallel programs is simple and intuitive.

It should result in improvements in three areas:

The programs will execute more efficiently because they are not arbitrarily constrained by a target architecture. The programs will be easier to port because they are not target-specific. And the program development process itself will be more efficient because it isolates and modularizes the three parts of parallel programming.

Acknowledgement

We thank Alex Nelson, who has contributed to this project at all levels, from helping us clarify our ideas to implementing the two TStreams prototypes.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press (Cambridge, MA) and McGraw-Hill (New York), second edition, 1996. [QA76.6.A255 1996].

[2] Ian Buck. Brook Specification v0.2. Technical report, Computer Science Department, Stanford University, Palo Alto, California, 2003. Available at `http://hci.stanford.edu/~winograd/cstr/abstracts/2003-04.html`.

[3] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. Available at `http://www.mpi-forum.org/docs/docs.html`.

[4] 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`.

[5] Kathleen Knobe and Carl D. Offner. TStreams: A Model of Parallel Computation (Preliminary Report). Technical report, HP Labs Technical Report HPL-2004-78, 2004. Available at `http://www.hpl.hp.com/techreports/2004/HPL-2004-78.html`.

[6] P. J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[7] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 2.0, November 2000. Available at `http://www.openmp.org/specs`.

[8] Umakishore Ramachandran, Rishiyur Nikhil, James Matthew Rehg, Yavor Angelov, Arnab Paul, Sameer Adhikari, Kenneth Mackenzie, Nissim Harel, and Kathleen Knobe. Stampede: A Cluster Programming Middleware for Interactive Stream-oriented Applications. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1140–1154, November 2003.