Characterizing Shared-Memory Applications: A Case Study of the NAS Parallel Benchmarks

Gheith A. Abandah
Computer Systems Laboratory
HPL-97-24
January, 1997

application analysis, shared-memory applications, NAS Parallel Benchmarks, trace analysis, simulation

The objective of this report is to present our characterization of a shared-memory implementation of the NAS Parallel Benchmarks (NPB). This characterization is needed to support the design decisions of future shared-memory multiprocessors. This report presents two sets of characterization data; the first set is the application characteristics that do not change from one hardware configuration to another, and the second set is the traffic characteristics of the application when run on a possible future hardware configuration. The data presented in this report include characterizations of NPB communication, sharing and cache behavior.

Contents

1 Introduction

Computer architects are increasingly relying on application characteristics for insights in designing cost-effective systems. This case study of NAS Parallel Benchmarks (NPB) is part of a project to characterize a collection of shared-memory applications to support the design of future systems. This study makes use of a collection of analysis tools that we have developed to analyze and characterize shared-memory applications. These tools are described in detail in our HP Laboratories technical report [1].

We have developed two main tools for analyzing shared-memory applications. The first tool is intended to generate abstractions that expose the inherent application characteristics. The second tool is intended to predict the application performance on a specific hardware configuration. Given a system configurations, the second tool predicts the traffic flow volume and characteristics under this configuration. It also generates traffic traces that are used to drive detailed system-level simulators for further evaluation of alternative design options.

Figure 1 shows an outline of our methodology in characterizing shared-memory applications. It shows that a shared-memory multiprocessor is used to collect traces by executing instrumented application codes. Nevertheless, other methods can be used for trace collection.

The *Shared-Memory Application Instrumentation Tool* (SMAIT) is used to instrument applications [1]. Instead of generating trace files, SMAIT can also pipe the traces to the analysis tools for *on-the-fly analysis*. On-the-fly analysis enables analyzing longer execution periods by solving the problem of huge trace files.

The two analysis tools are the *Configuration Independent Analysis Tool* (CIAT) and the *Configuration Dependent Analysis Tool* (CDAT). CIAT generates a characterization of the memory instructions, branching, synchronization, communication patterns, and data sharing. CIAT also generates a *memory usage file* that specifies the usage statistics of all accessed memory pages. CDAT reads a *configuration file* that specifies a proposed system configu-

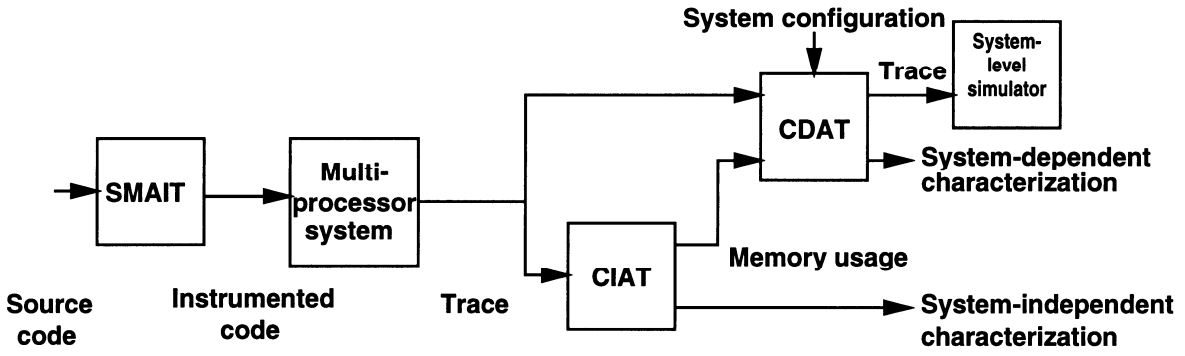

Figure 1: Application analysis methodology.

ration and simulates the execution of the traces on this configuration. CDAT produces a characterization and trace files of the traffic that would occur when the application is run on the simulated configuration. CDAT can use the memory usage file to map memory pages to the simulated memory banks.

Section 2 gives an overview of NPB and the analysis conditions. Section 3 presents NPB configuration independent characteristics. Section 4 presents NPB traffic characteristics on a possible future distributed shared memory (DSM) multiprocessor configuration. Section 5 concludes the paper by stating some of the key NPB characteristics.

2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks 1.0 [2] are 5 kernels and 3 pseudo-applications that mimic the computation and data movement characteristics of large-scale computational fluid dynamic applications. These benchmarks are specified algorithmically so that computer vendors can implement them on a wide range of parallel machines. In this report we present our analysis of the HP Convex implementation of NPB on the Convex Exemplar multiprocessor [3]. This implementation was mainly developed by Herb Rothmund of the HP Convex Technology Center. The performance of an earlier version of this implementation is reported in a NPB results report [4].

We have analyzed two of the NPB kernels (CG and MG) and the three pseudo-applications (SP, LU, and BT). Table 1 shows the two NPB problem sizes analyzed in this study. The MG problem sizes are not standard and are selected to get a reasonable trace length.

Table 1 shows two numbers for each benchmark-size combination; the first number specifies the problem data size, and the second number specifies the number of iterations in the main parallel loop. The execution time of the *Sample size* is in the order of one second, while the execution time of the *A size* is in the order of tens to hundreds of seconds.

The Convex implementation of NPB uses threads. In this implementation, all benchmarks start with a serial *initialization phase* where only thread 0 is active. SP and LU also have parallel loops in the initialization phase. After the initialization phase, p threads are spawned to run on p available processors where they cooperate in executing the main loop for the number of iterations specified in Table 1. We call this phase the *parallel phase*. The threads coordinate their work by using synchronization barriers. At the end of the parallel phase, the

Table 1: Analyzed NPB problem sizes.

Problem size name	CG	MG	SP	LU	BT
Sample	$1,400/15$	$64^3/4$	$12^3/100$	$12^3/50$	$12^3/60$
A	$14,000/15$	$128^3/4$	$64^3/400$	$64^3/250$	$64^3/200$

multiple threads join and only thread 0 remains active in the *wrap-up phase* to do validation and reporting.

Generally, most of the execution time is spent in the parallel phase which has the main time component reported for these benchmarks. For this reason, we give more attention to the parallel phase. Unless otherwise specified, the reported characteristics are for the parallel phase. We have noticed that, within the parallel phase, the characteristics do not change from one iteration to another. Hence, to save analysis time, we perform our analysis of the A size from the start to the end of the second iteration of the parallel phase. We use the characteristics of the second iteration as a representative of the whole parallel phase. However, for the Sample size, the reported characteristics are for the whole parallel phase.

This NPB implementation was instrumented, compiled, and analyzed on a 4-node Convex SPP-1600 multiprocessor. Table 2 shows the configuration of this system.

We have analyzed these benchmarks on a varying number of processors: 1, 2, 4, 8, and 16. Analyzing these benchmarks for two problem sizes and a variety of processor numbers enabled us to understand the characteristics of these applications as a function of problem size and number of processors used.

Table 2: The SPP-1600 configuration.

Feature	SPP-1600 Data
Number of processors	32 in 4 nodes
Processor	PA 7200 @ 120 MHz
Main memory	1024 MB per node
OS version	SPP-UX 4.2
Fortran compiler	Convex FC 9.5

3 Configuration Independent Characteristics

In this section, we present the characteristics of NPB reported by CIAT. This section presents the characteristics that are inherent in the code generated by compiling this NPB implementation using the compiler specified in Table 2. These characteristics are called configuration independent characteristics because they only include characteristics that do not change from one hardware configuration to another. Here, hardware configuration refers to interconnection topology, cache type and size, coherence protocols, memory allocation, etc.

These characteristics are summarized in the following 5 subsections. Subsection 3.1 presents the overall volume characteristics. Subsection 3.2 presents the sharing characteristics. Subsection 3.3 presents the memory instructions characteristics. Subsection 3.4 presents the average communication characteristics and Subsection 3.5 presents the communication variations over time.

3.1 Overall Characteristics

Table 3 presents the overall characteristics of the 5 benchmarks. These characteristics are found by analyzing the execution from the start to the end for the Sample size of the 5 benchmarks and for the A size of the CG benchmark, and from the start to the end of the second iteration for the A size of the other 4 benchmarks. Although these characteristics were extracted from 4-processor runs, we have noticed that they do not significantly change with different number of processors.

- The *code size* is the size of the touched instructions,
- the *data size* is the size of the accessed data elements,
- the *data locality index* is found as the ratio of the sum of the sizes of all load and store instructions to the data size, and
- the *code locality index* is found as the ratio of the number of executed instructions to the code size.

Table 3 shows that while the code size of the 3 pseudo-applications is larger than the code size of the 2 kernels, the pseudo-applications access smaller data sets. This in turn results in smaller code locality and larger data locality.

While the code size remains almost constant for the two problem sizes, the data size of the A size is about an order of magnitude larger than the Sample size. Note that the locality numbers of the A size of the last four benchmarks are from a partial execution and would be larger if we had analyzed all the iterations of the benchmark.

Table 4 shows some of the overall characteristics of the parallel phase.

Table 3: Overall characteristics.

Characteristic	Size	CG	MG	SP	LU	BT
Problem size	Sample	$1,400/15$	$64^3/4$	$12^3/100$	$12^3/50$	$12^3/60$
	A	$14,000/15$	$128^3/4$	$64^3/400$	$64^3/250$	$64^3/200$
Code size (in KB)	Sample	12	21	71	59	70
	A	12	21	71	59	70
Data size (in MB)	Sample	9.0	5.3	0.21	0.22	0.25
	A	120	39	30.1	30.1	30.3
Data locality Index	Sample	50	92	2536	1583	4534
	A	86	*62*	*147*	*189*	*344*
Code locality index (in 10^3)	Sample	39	30	11	10	19
	A	915	*151*	*100*	*121*	*116*

Table 4: Parallel phase characteristics.

Characteristic	Size	CG	MG	SP	LU	BT
Instructions per iteration (in 10^6)	Sample	6.6	28.9	1.9	2.8	5.7
	A	228	221	432	555	1335
Memory instructions	Sample	64%	41%	37%	34%	46%
	A	67%	42%	37%	36%	46%
Average instructions between taken branches	Sample	13	44	62	18	117
	A	13	43	58	78	151
Average instructions between barriers (in 10^3)	Sample	34	243	118	142	354
	A	722	1,590	27,000	27,700	83,400

- The number of instructions executed in each iteration of the parallel phase by all processors,
- the percentage of the instructions that reference memory,
- the average number of instructions between taken branches, and
- the average number of instructions between synchronization barriers.

As expected, the instructions per iteration numbers show that the amount of "work" in one iteration increases as the problem size increases. CG has the highest percentage of memory instructions because it spends most of its time doing a reduction process where it loads two values followed by an arithmetic operation. The large number of instructions between taken branches is typical in some scientific applications where the application spends most of its time in loops that have significant arithmetic computations and one backward branch at the end of the loop. Table 4 also shows that this implementation uses sparse synchronization. For the A size, the average number of instructions between synchronization barriers is, at least, hundreds of thousands of instructions.

3.2 Sharing

Figure 2 shows four graphs that summarize the sharing characteristics of the five benchmarks. The left-hand graphs show the shared memory percentage as a function of the number of processors. The shared memory percentage is the percentage of memory locations that were touched by more than one processor of all touched memory locations. For example, about 90% of CG's memory is accessed by a single processor in the Sample size. In general, the shared memory percentage is approximately constant for multiple processors, or increases to an asymptotic value as $O(1 - 1/p)$, or decreases for large number of processors (as in LU).

The constant behavior occurs when there is a fixed size of memory that is shared by all the available processors. The $O(1 - 1/p)$ behavior often occurs when a fixed size of memory is

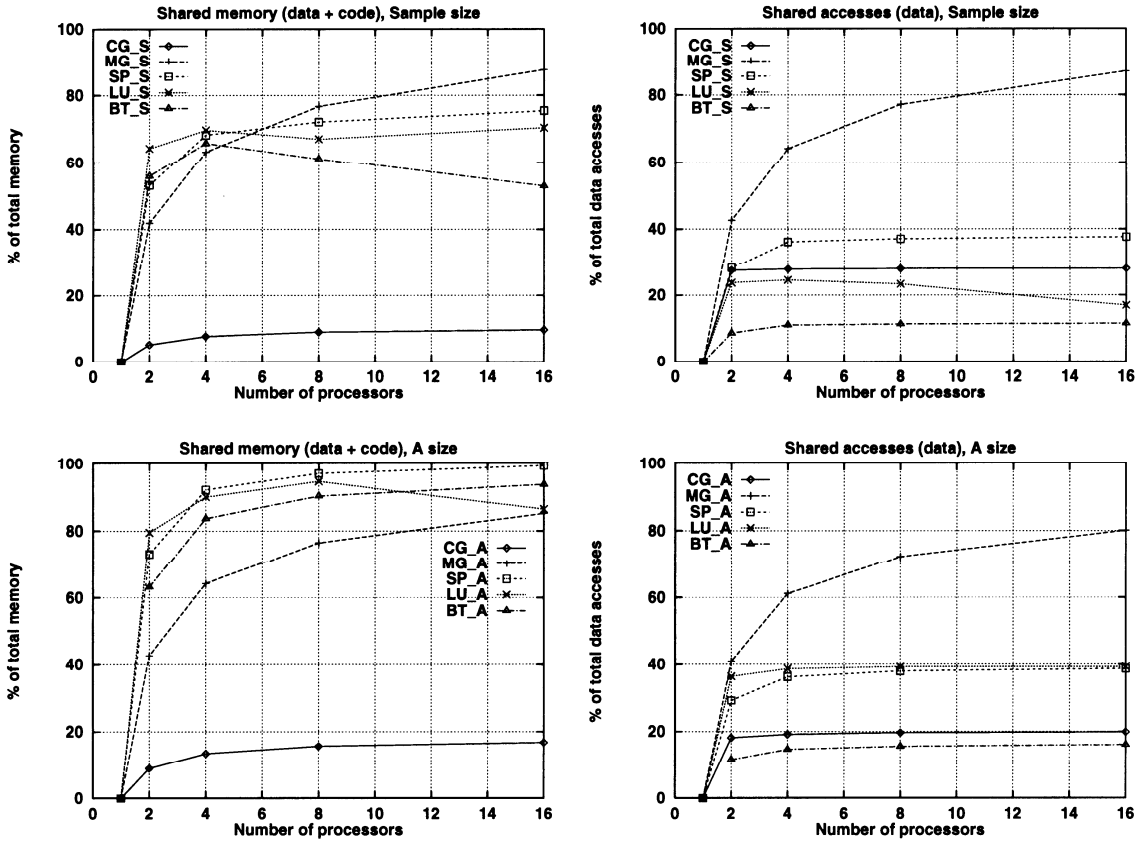

Figure 2: Sharing characteristics.

initialized by one processor and is then partitioned linearly to the available processors. The decrease in the shared memory percentage is due to the increase in the private memory as the number of processors increases.

The shared memory percentage is larger for the larger problem size because most of the private data structures like the stack and constant data do not increase for larger problem sizes.

The right-hand graphs show the shared data access percentage as a function of the number of processors. The shared data access percentage is the percentage of memory instructions accessing shared memory locations of all memory instructions. Notice that for the 3 pseudo-applications in the Sample size, while the shared memory is about 70%, the shared accesses are only about 30%. This indicates that private memory is more intensely accessed than shared memory.

3.3 Memory Instructions

Table 5 shows the percentages of the various memory instruction types when using 4 processors. These percentages do not change much for a different number of processors.

Table 5 shows that about 70%–96% of memory instructions are loads. The vast majority of memory instructions access word and double elements. Double accesses are 68%–85% of the total. There is a very small percentage of byte and halfword accesses if any. Moreover, the percentage of double accesses generally increases as the problem size increases.

Table 5: Distribution of memory instructions (in percent).

Type	Size	CG	MG	SP	LU	BT
Load byte	Sample	0	0	0	0	0
	A	0	0	0	0	0
Load halfword	Sample	0	0	0.01	0	0
	A	0	0	0	0	0
Load word	Sample	30.10	1.70	3.35	8.59	10.64
	A	32.09	0.85	1.72	3.06	11.33
Load float	Sample	0	0.15	0.02	0	0
	A	0	0.07	0	0	0
Load double	Sample	64.24	85.81	69.50	70.14	57.25
	A	66.25	87.90	73.23	77.56	58.88
Store byte	Sample	0	0	0	0	0
	A	0	0	0	0	0
Store halfword	Sample	0	0	0	0	0
	A	0	0	0	0	0
Store word	Sample	1.06	0.21	1.49	2.52	4.29
	A	0	0.08	0.21	0.30	3.71
Store float	Sample	0	0.08	0.01	0	0
	A	0	0.03	0	0	0
Store double	Sample	4.60	12.06	25.62	18.75	27.80
	A	1.65	11.06	24.84	19.08	26.08

3.4 Communication

In a shared-memory application, processors communicate by accessing shared memory. CIAT is used to find the amount of communication and it is classified into the following communication patterns:

1. Number of *read-after-write* accesses (RAW): A RAW access occurs when one or more processors load a memory location that was stored by a processor. This pattern does

not include the case where only one processor loads a memory location that was stored by this same processor. This is a common communication pattern; it occurs in a producer-consumer situation where one processor produces data and one or more processors consumes it.

2. *Sharing degree* for RAW. This is a vector **S**, where **S**[k] is the number of times that a memory location was read by k processors after being written.

3. Number of *write-after-read* accesses (WAR): A WAR access occurs when a processor stores a memory location that was loaded by one or more processors. This pattern does not include the case where a processor stores to a memory location that was only loaded by itself. This is also a common pattern; it occurs when a processor updates data that was read by other processors.

4. *Invalidation degree* for WAR. This is a vector **I**, where **I**[k] is the number of times that a memory location was written after being previously read by k processors.

5. Number of *write-after-write* accesses (WAW): A WAW access occurs when a processor stores to a memory location that was stored by another processor. This is a less common pattern; it occurs when multiple processors write without reading, or when processors take turns on a memory location where in each turn a processor writes and reads.

6. Number of *read-after-read accesses* (RAR): A RAR access occurs when a processor loads a memory location that was loaded by another processor and the first visible access to this location is a load. This is an uncommon pattern; it occurs in bad programs that read uninitialized data. Nevertheless, CIAT sometimes encounters this pattern when the data is initialized in untraced routines. These accesses can be added to the RAW accesses.

Figure 3 shows two graphs for the percentage of RAW access of all accesses as a function of the number of processors. The left-hand graph is for the Sample size and the right-hand side is for the A size.

It is clear that the percentage of RAW access in the Sample size is larger than the A size. This is because larger problem sizes result in less communication per instruction due to the decrease in the ratio of processor domain boundary to the domain size. CG and MG have the worst communication scalability because their RAW access increases linearly with the number of processors. The pseudo-applications' RAW access follows $O(1 - 1/p)$.

Figure 4 shows the percentage of the 16 possible sharing degrees of the RAW access when using 16 processors for the A size $((S[k] \times 100/\sum_{i=1}^{16} S[i]); k = 1, \ldots, 16)$. The sharing degree for the Sample size is similar. It shows that the majority of RAW accesses for SP and BT have a sharing degree of 1; this is a one producer, one consumer situation. For MG and LU,

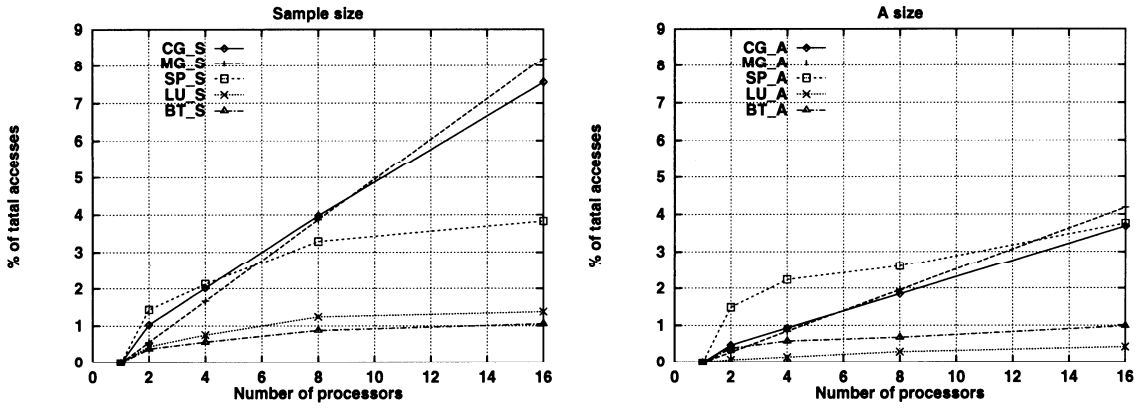

Figure 3: Read-after-write communication.

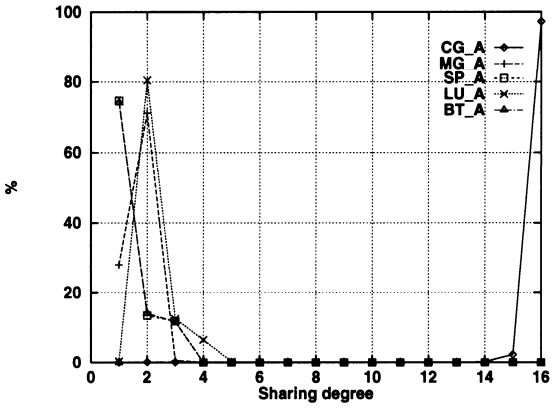

Figure 4: Sharing degree in RAW for 16 processors.

the majority of RAW accesses have a sharing degree of 2. CG has a wide sharing degree; the majority of RAW accesses have a sharing degree that equals the number of available processors.

Figure 5 shows two graphs for the percentage of WAR access of all accesses as a function of the number of processors. The left-hand graph is for the Sample size and the right-hand side is for the A size.

Similar to RAW, the percentage of WAR access in the Sample size is larger than the A size. Although MG still has a linear increase in WAR as a function of processors, CG has a constant percentage of WAR for multiple processors. The pseudo-applications' WAR access also follows $O(1 - 1/p)$. Notice that RAW accesses are more than WAR accesses because in these iterative applications one WAR access can follow multiple RAW accesses.

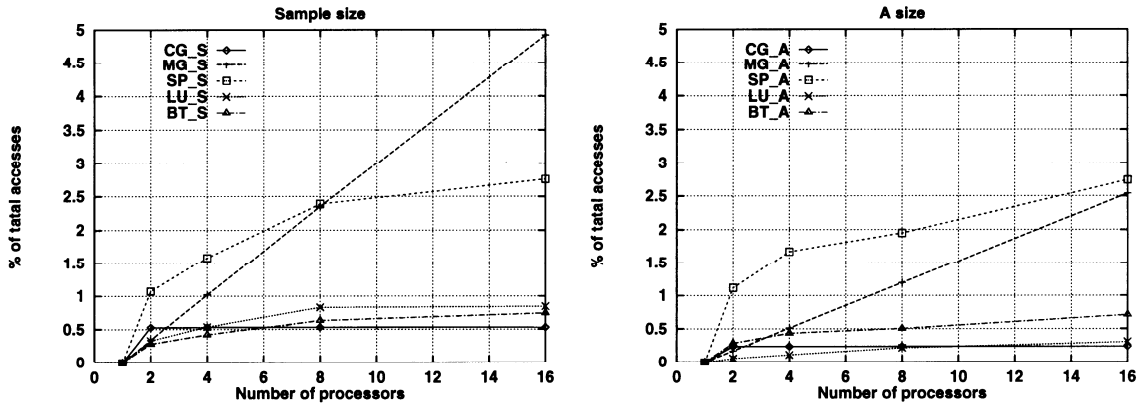

Figure 5: Write-after-read communication.

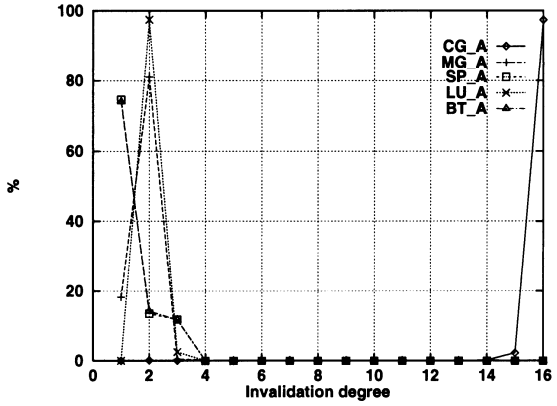

Figure 6: Invalidation degree in WAR for 16 processors.

Figure 6 shows the percentage of the 16 possible invalidation degrees of the WAR access when using 16 processors for the A size $((I[k] \times 100/\sum_{i=1}^{16} I[i]); k = 1, \ldots, 16)$. The invalidation degree for the Sample size is similar. For these iterative applications, the invalidation degree is similar to the sharing degree.

These applications have negligible RAR and WAW accesses.

Table 6 summarizes the communication characteristics of the 5 benchmarks. The percentage of RAW accesses ranges from 0.4% to 8% for 16 processors which is larger than the percentage of WAR accesses that ranges from 0.2% to 5%.

Table 6 also summarizes the RAW and WAR access as a function of the number of processors p and shows the weighted average of the sharing and invalidation degrees.

Table 6: Summary of the communication characteristics.

Type	Size	CG	MG	SP	LU	BT
RAW for p=16	Sample	7.6%	8.2%	3.8%	1.4%	1.1%
	A	3.7%	4.2%	3.8%	0.4%	1.0%
WAR for p=16	Sample	0.5%	4.9%	2.8%	0.8%	0.7%
	A	0.2%	2.5%	2.7%	0.3%	0.7%
RAW(p)	Both	O(p)	O(p)	O(1-1/p)	O(1-1/p)	O(1-1/p)
RAW(p)	Both	O(1)	O(p)	O(1-1/p)	O(1-1/p)	O(1-1/p)
RAW sharing degree for p=16	Sample	15.1	1.8	1.4	2.8	1.4
	A	16.0	1.7	1.4	2.3	1.4
WAR Invalidation degree for p=16	Sample	15.1	1.9	1.4	2.2	1.4
	A	16.0	1.8	1.4	2.0	1.4

3.5 Communication Time Analysis

In this subsection we present our characterization of the communication variation over time. We only present our analysis of two representative benchmarks. Our characterization method is summarized by the following steps:

1. CIAT is used to analyze the benchmarks and to generate a trace of the four communication events described in Subsection 3.4. Each event is tagged with the instruction number that caused that event.
2. The execution period is divided into 1000-instruction intervals.
3. The number of communication events in each interval is counted.
4. The communication rate in each interval is calculated as the number of communication events divided by the product of the interval width and the number of processors.
5. Using standard statistical methods, the average, minimum, and maximum communication rates are calculated.
6. The communication rate density function is calculated (not including rate=0). The rate zero is excluded to minimize the effect of the serial initialization phase which does not have any communication.
7. The density function is integrated to find the distribution function.

Figure 7 shows the number of communication events over time for the kernel CG and the pseudo-application SP using 16 processors for the Sample size. The two graphs show a

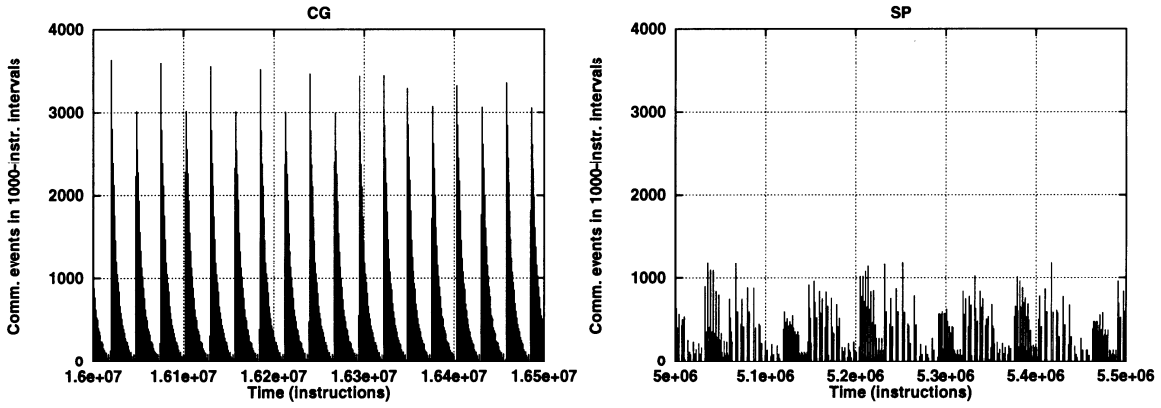

Figure 7: Number of communication events over time using 16 processors for the Sample size.

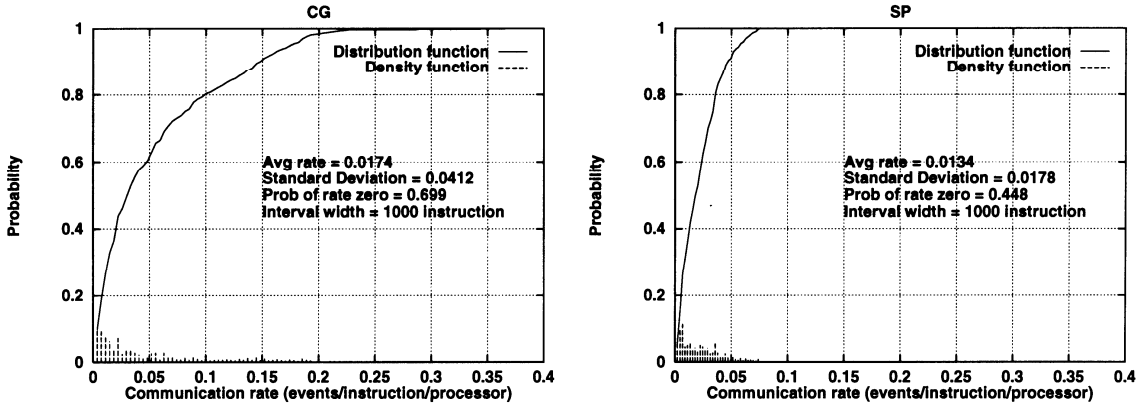

Figure 8: Communication rate using 16 processors for the Sample size.

500,000-instruction period from the parallel phase. Both applications show a bursting behavior; however, CG has a higher rate of communication events.

From Figure 7, the periodic behavior of CG is obvious; its graph has 18 cycles with a cycle time of about 28,000 instructions. SP's periodic behavior is less clear with larger cycles; its graph has 3 cycles with a cycle time of about 170,000 instructions. For larger problem sizes, these cycles get longer. The periodic behavior of these applications is mainly because of their good load balance and barrier synchronization.

Figure 8 shows the density and distribution functions of CG and SP using 16 processors for the Sample size. The two graphs show that the communication rate probability decreases for larger rates and SP's rate decreases faster than CG's rate which has a longer tail.

4 Configuration Dependent Characteristics

In this section, we present the characteristics of NPB reported by CDAT. This section presents the traffic characteristics of this NPB implementation when run on a cc-NUMA multiprocessor. The configuration of this multiprocessor is selected to match our projections of systems that may be available a few years from now. Additionally, this section presents results for evaluating some of the design options.

Subsection 4.1 starts with an overview of the multiprocessor configuration used in this study. Subsection 4.2 starts with the big picture of the traffic by presenting the miss ratio as a function of the number of processors and the problem size. Subsection 4.3 investigates the effect of the cache line size on the miss ratio. Subsections 4.4 and 4.5 present the traffic characteristics for single-node and multiple-node configurations respectively. Finally, Subsection 4.6 investigates the performance of four memory allocation strategies.

4.1 Multiprocessor Configuration

The simulated configuration is a cc-NUMA multiprocessor that consists of nodes interconnected by a crossbar. Each node has 4 processors connected by a cache coherent bus and has a 4-bank memory with full directory. The system uses a cache coherence protocol similar to the one used in the DASH [5] project.

Each processor has a combined level 2 cache that is 4 MB in size and 4 way set-associative. We used a number of cache line sizes ranging from 32 to 256 bytes. Unless otherwise specified, the cache line size is 64 bytes. The memory page size is 4 KB.

4.2 Miss Ratio

Figure 9 shows five graphs for the data miss ratio of the five benchmarks. The data miss ratio is the ratio of load and store misses to the total loads and stores. Each graph has two curves; one curve for the data miss ratio of the Sample size as a function of the number of processors, and another for the A size. Note that the first two graphs have a different scale than the rest.

It is clear that the data miss ratio depends on the number of processors and the problem size. Table 7 summarizes the effects of these two factors on the data miss ratio and the cause of these effects.

Capacity misses occur when the working set is larger than the cache size. When the problem size decreases, capacity misses decrease until they become zero when the working set fits in the cache. In NPB, as the number of processors is increased the data structures are partitioned into smaller working sets, thus decreasing capacity misses. This behavior is clear for all the benchmarks with the A size, particularly for CG, which has a large working set

that results in large data miss ratio for 1, 2, and 4 processors. When the number of processors increases to 8, the working set suddenly fits in the 8 caches and there are no capacity misses.

Coherence misses are due to inter-processor communication and the overheads of the cache coherence protocols. The communication characterized in Subsection 3.4 can be used to explain the coherence misses in these benchmarks. As we have noticed above, communication percentage increases for smaller problem sizes and larger number of processors. This trend is most clear in the Sample size where the data miss ratio increases as p increases.

Communication efficiency refers to the ratio of used bytes to transferred bytes. The communication efficiency increases when more elements in a cache line are used. Communication efficiency and false sharing explain why the data miss ratio of the Sample size increase faster than the data miss ratio of the A size.

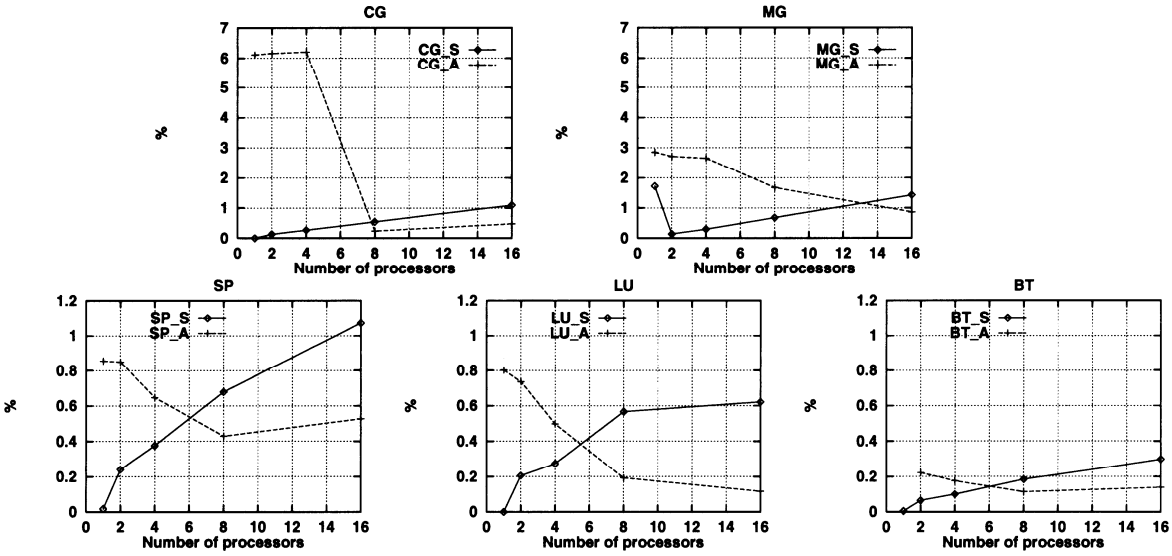

Figure 9: Data miss ratio.

Table 7: Factors affecting the data miss ratio.

	Sample size or Large p	A size or Small p
Capacity misses	less	more
Coherence misses	more	less
Communication efficiency	worse	better
False sharing	more	less

4.3 Effect of Cache Line Size

Figure 10 shows the effect of the cache line size on the data miss ratio for the Sample size.

CG and MG have data miss ratios that decrease by about 1/2 when the cache line size is doubled. The three pseudo-applications, using small problem size to processors ratio, have data miss ratios that increase as the line size is increased due to false sharing.

Figure 11 shows the effect of the cache line size on the data miss ratio for the A size when using 4 processors. All the five benchmarks have data miss ratios that decrease by about 1/2 when the cache line size is doubled. This indicates that the effect of false sharing is negligible at this problem size to processors ratio.

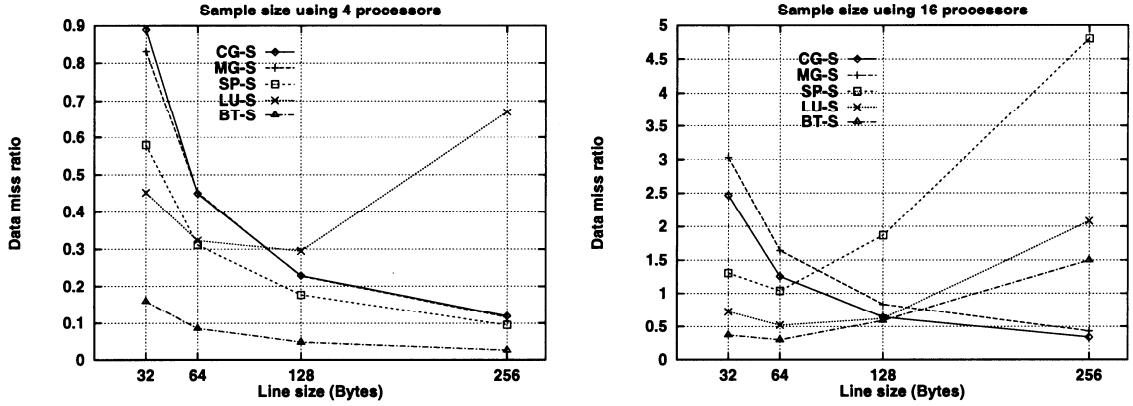

Figure 10: Effect of cache line size (Sample size).

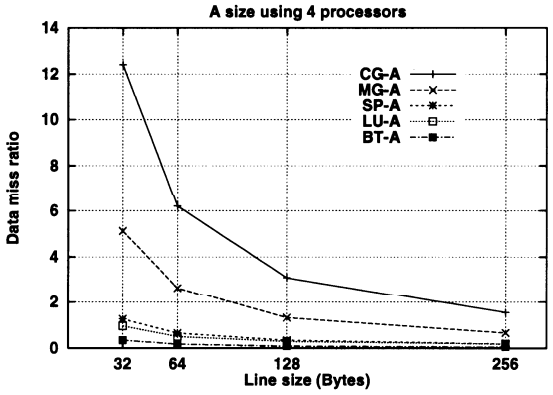

Figure 11: Effect of cache line size (A size).

4.4 Single Node Traffic

Figure 12 shows graphs for single node traffic of CG and SP using 4 processors for both problem sizes. These graphs show the number of bus signals generated in response to cache misses and replacements. When there is an instruction miss, the cache generates a `read_shar` signal, it generates a `read` signal for load miss, a `read_priv` for store miss, a `req_inv` for store hit on a shared line, an `update_data` on snoop hit where the cache writes the dirty line to the bus, a write-back (`w/b`) when replacing a dirty line, and a zero-length write-back (`0_w/b`) when replacing an exclusive line.

Note that the instruction miss number is relatively negligible. Most of the store misses of CG Sample size are due to initialization in the parallel phase start. When cold misses are ignored, load misses are extremely more than store misses. Since `req_inv` number is larger than `read_priv`, this means that the two benchmarks usually read shared values before updates. Coherence misses are responsible of most of the Sample size traffic and capacity misses are responsible of most of the A size traffic. Snoop hit rates increase with higher communication and smaller data sets. Write-back rates increase with larger data sets and zero-length write-back traffic can be intense.

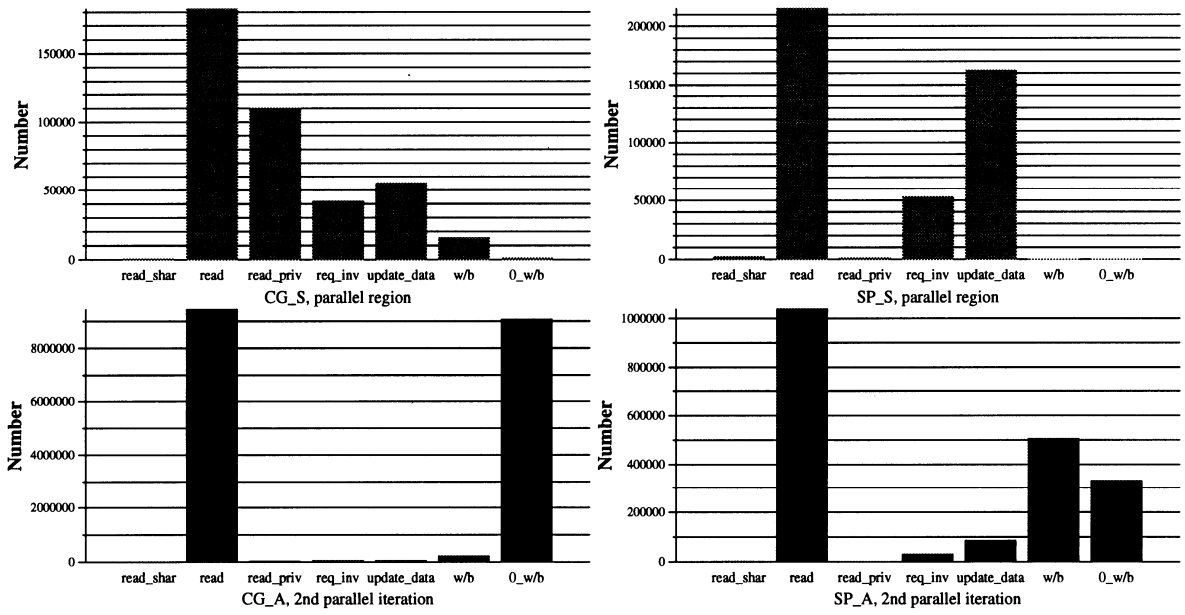

Figure 12: Single node traffic.

4.5 Multiple Node Traffic

Figure 13 illustrates one important aspect of the multiple node traffic of CG and SP using 16 processors in 4 nodes for both problem sizes. These graphs show the status of the missed

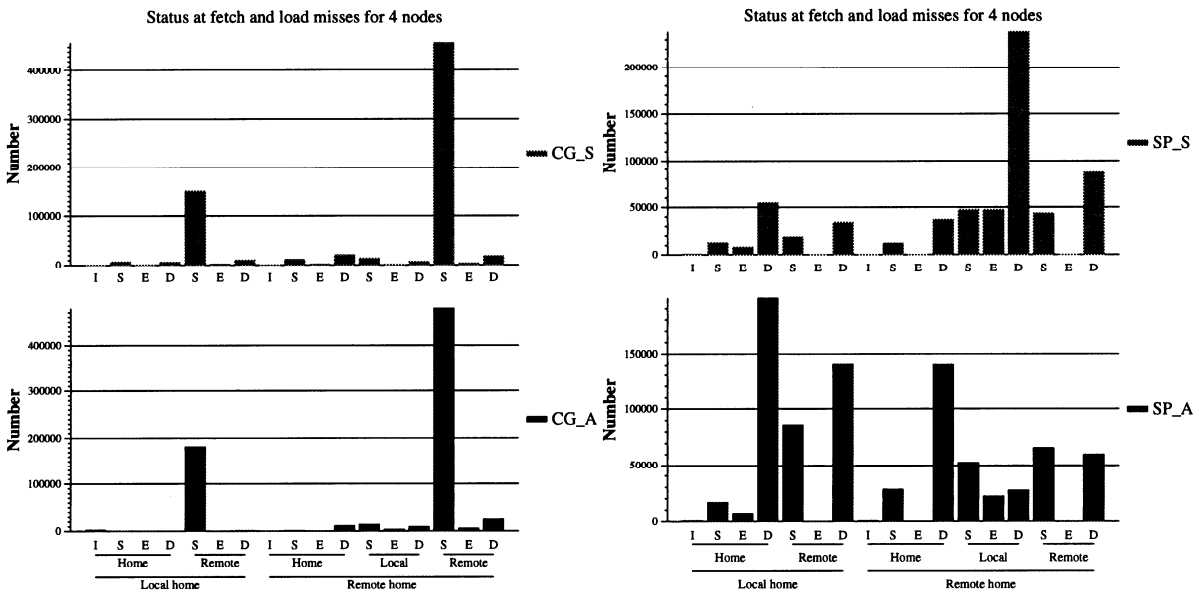

Figure 13: Multiple node traffic.

lines at instruction and load misses. The line status depends on its home (local or remote), where it is cached (home, local, and/or remote), and its caching status (I: idle not cached, S: shared, E: exclusive, D: dirty).

Because of CG's high sharing degree, a miss most likely finds the line in the shared state. Because of SP's low sharing degree, a load miss most likely finds the line in the dirty state.

4.6 Memory Allocation Strategies

In this subsection we present our results comparing 4 memory allocation policies, i.e. policies for allocating home nodes for memory pages. We have used the following 4 policies:

1. RR1: Round Robin interleaving 1. All pages are interleaved in a round robin scheme across the available nodes according to their virtual addresses.

2. RR2: Round Robin interleaving 2. Similar to RR1, but code pages are replicated in every node.

3. Oracle: Code pages are replicated, private pages are allocated locally, and shared pages are interleaved in a round robin scheme across nodes.

4. 1Touch: First Touch allocation. A page is allocated in the node where it was first accessed.

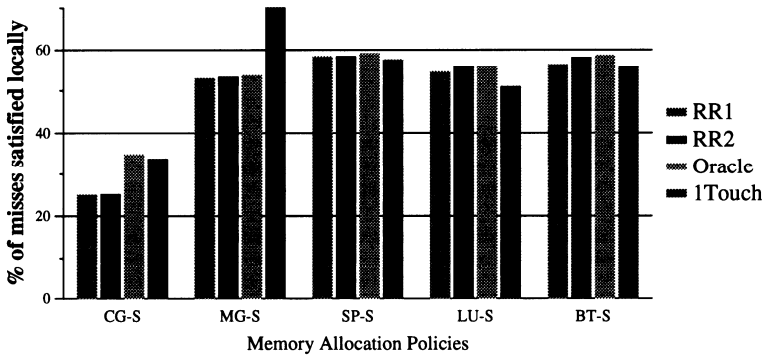

Figure 14: Memory allocation policies.

Figure 14 shows some of the performance effects of using these four memory allocation policies. The data shown is the percentage of misses satisfied locally when using 16 processors for the Sample size. Better policies result in more local misses since remote misses normally take longer latency.

Oracle is always better than round robin allocation because it allocates private data locally. For CG and MG, 1Touch has good performance because most of the accesses are to private memory, thus it pays to allocate them in the node where they are first touched. Nevertheless, 1Touch has relatively bad performance because most of the data is shared and is initialized by processor 0, thus it will all be allocated in node 0.

5 Conclusions

In this report we characterized 5 benchmarks of the NAS Parallel Benchmarks. We have analyzed these benchmarks using two problem sizes and a varying number of processors to get an understanding of their characteristics as a function of the problem size and the number of processors. We used configuration independent analysis to capture their inherent characteristics and configuration dependent analysis to get useful information about their performance on future configurations.

These benchmarks have relatively small code size and the 2 kernels have data sizes larger than the pseudo-applications. As the problem size increases, the work in the parallel phase increases. While about 2/3 of CG's instructions are memory instructions, only about 1/3 of the instructions from other benchmarks are memory instructions. These benchmarks have generally large basic blocks promoting instruction level parallelism and also have sparse synchronization.

While less than 20% of CG's memory is shared, the other benchmarks have a large fraction of shared memory. This is not reflected in the percentage of shared accesses; e.g. the pseudo-applications have less than 40% of their accesses shared, indicating that private data

is accessed more intensely. The majority of these accesses access single word and double elements.

Most of the communication is in the form of an iterative producer-consumer(s) pattern. While communication grows linearly with the number of processors for the two kernels, it grows to an asymptotic value in the 3 pseudo-applications. Although CG has wide sharing degree, the other benchmarks have the majority of their shared accesses with degree of one or two. This communication has a bursting and periodic nature.

The data miss ratio is affected by two main factors: it decreases when the number of processors is increased and the problem size is decreased due to fewer capacity misses, and it increases when the number of processors is increased and the problem size is decreased due to more coherence misses. The data miss ratio generally benefits from larger cache line sizes.

We have also presented some data for single-node and multiple-node traffic. Most of the traffic is due to load misses. When the sharing degree is high, the status of the missed lines is likely to be shared. While when the sharing degree is low, the status of the missed lines is more likely to be dirty.

Acknowledgment

I would like to thank Isom Crawford and Herb Rothmund of the HP Convex Technology Center for providing the Convex SPP NPB implementation. I would also like to thank Josep Ferrandiz, Tom Rokicki, Milon Mackey, Lucy Cherkasova, and Sekhar Sarukkai for their valuable comments in preparing and reviewing this report.

6 References

[1] G. Abandah, "Tools for characterizing distributed shared memory applications," Tech. Rep. HPL-96-157, HP Laboratories, Dec. 1996.

[2] D. Bailey *et al.*, "The NAS parallel benchmarks," Technical Report RNR-94-07, NASA Ames Research Center, Mar. 1994.

[3] T. Brewer, "A highly scalable system utilizing up to 128 PA-RISC processors," in *Digest of papers, COMPCON'95*, pp. 133–140, Mar. 1995.

[4] S. Saini and D. H. Bailey, "NAS parallel benchmark results 12–95," Tech. Rep. NAS–95–021, NASA Ames Research Center, Dec. 1995.

[5] D. Lenoski *et al.*, "The Stanford DASH Multiprocessor," *Computer*, vol. 25, pp. 63–79, Mar. 1992.