Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'95), July 1995, Limassol, Cyprus, ©1995 IFIP.

Analysis of Communications and Overhead Reduction in Multithreaded Execution*

Lucas Roh Walid A. Najjar

Department of Computer Science
Colorado State University
Fort Collins, CO 80523 USA
{roh,najjar}@cs.colostate.edu

Abstract

In a multithreaded execution, each thread can be thought of as running on its own virtual processor, with several virtual processors multiplexed onto a single physical processor. At any given time, some of these virtual processors are either sending or waiting for messages. When the degree of multithreading is high, there is a high potential load on the interconnection network. It is important to understand the aggregate behavior of these messages for the design of the memory hierarchy, network structure, and understand more concretely the behavior of multithreaded execution.

In this paper, we study several issues related to communication patterns in a non-blocking, framelet-based multithreaded model. These issues include the sources of message generations and the locality of these messages.

The results indicate that roughly a third of all tokens are memory-related, another third are involved in parallelism management, and the final third are involved in intra-function and intra-loop communications (e.g., scalar data values). Based upon these results, we examine two techniques designed to reduce the amount of communications involved and the associated overheads. The results indicate that about 80% of non-memory related inter-processor messages are eliminated, which results in an overall reduction of network traffic by about 50%.

1 Introduction

Multithreading attempts to exploit instruction level locality implicit in the von Neumann model as well as the latency tolerance and fast synchronizations of the dataflow model. There currently exists a wide array of multithreading processor architecture models reflecting a large set of design parameters. These design parameters include:

- A blocking [Smi81, ACC+90, DFK+92, WG89] or non-blocking thread execution model [NPA92, SYH+89, SYH+91, HG93, HTG94, Vas94]. Note that in a non-blocking execution the synchronization points are statically determined by the compilers while in a blocking execution the hardwrae must support dynamic synchronization.
- The use of hardware contexts with separate register sets [Smi81, ACC+90].
- The degree of multithreading allowed.
- The use of dataflow-style matching for synchronization [RNSB94] or direct matching [PC90, NPA92, SYH+89].
- Hardware support for synchronization [NPA92, SYH+89] or synchronizations in the software [CSS+91].

Any of these multithreading processor architectures can serve as a building block for massively parallel processors.

Conceptually, each thread runs on its own virtual processor, and many virtual processors are multiplexed onto a single physical processor. At any given time, some of these virtual processors are either sending or waiting to receive messages. The higher the degree of multithreading (i.e., the more virtual processors per physical processor), the higher the potential load on the interconnection network. Therefore, multithreading not only increases the processor utilization but also the interconnection network utilization. In fact, the message handling overhead and the network bandwidth limitation could limit the degree of multithreading that is achievable, and hence the ability to tolerate latencies [Cul94].

The aim of this paper is to study the nature of inter-processor communications between multithreaded processors, which will be used to develop techniques that reduce the amount of inter-processor communications and the associated overhead. Our study is based upon a non-blocking thread model that use a framelet-based storage model. Non-blocking threads require all inputs to be present before the execution can start and, once started, it executes to completion. Instead of associating a unit of frame storage with an activation of a code-block such as a loop or a function containing several threads (e.g., TAM [CSS+91], *T [NPA92], EM-4 [SYH+89, SYH+91], the Iannucci's Hybrid Architecture (IHA) [Ian88]), a storage is associated with an activation of a single thread instead (a framelet). In this storage model, a frame size would typically be much smaller and requires more frequent allocation and deallocation operations; however, the potential for exploiting parallelism is greater since threads within a given code-block can run on different processors. In general, due to a small execution state associated with a non-blocking, framelet-based model, the

*This work is supported by NSF Grant MIP-9113268

communication between threads play a more important role. Three specific issues related to communications are explored in this paper:

- A classification of messages into different types according to the function performed. The number and frequency of messages in each class are also reported. This data is helpful in the design of code generators.
- A definition and quantitative evaluation of the input locality of threads. These results are useful in the design of caches, and storage hierarchies in general, and of effective scheduling policies.
- A definition and measurements of the output locality of threads. These result show to what degree output messages of therads can be *vectorized* according to their specific thread targets.

Results indicate that about a third of messages are related to remote memory accesses, and another third are related to handling parallelism. In addition, there is a high amount of both input and output localities. These results indicate that a couple of compiler techniques, called *grouping* and *bundling*, can effectively reduce the requirements on the network via reductions in the number of inter-processor messages and in overheads. As a result, the number of non-memory related messages are reduced by 70-80%. Although these techniques are not novel, their appeal lies in their simplicity of implementations and applications of these techniques demonstrate the effectiveness of quantitative analysis of communication patterns in multithreaded executions.

In Section 2 we describe our execution model including a basic processor model and briefly summarize our threaded code generation. Section 3 describes the analysis of communications. In Section 4, the *bundling* and *grouping* techniques are presented along with their results. Related work is discussed in Section 5. Concluding remarks are given in Section 6.

2 Multithreaded Execution Model and Code Generation

In this section we briefly describe the Pebbles execution model [RNSB94] and its code generation. The Pebbles model is based on dynamic dataflow scheduling where each actor, or a node in dataflow graphs, represents a thread. A thread is a statically determined sequence of RISC-style instructions operating on registers. Similar to dataflow, threads are dynamically scheduled to execute based upon the availability of data. Once a thread starts executing, it runs to completion without blocking and with a bounded execution time. The bounded execution time implies that each instruction in threads must have a fixed execution time.

Inputs to a thread comprise of all data values required to execute the thread to its completion. Hence, register values do not live across threads. A thread is enabled to execute only when *all* the inputs to the thread are available. Multiple instances of a thread can be enabled at the same time and are distinguished from each other by a unique "color." The thread enabling condition is detected by the matching/synchronization mechanism which matches inputs to a particular instance of a thread. Data values are carried by *tokens*. Each token consists of a continuation, an input port number to the thread and one or more data values. A continuation uniquely identifies an activation of a single thread and consists of a color and a pointer to the start of thread.

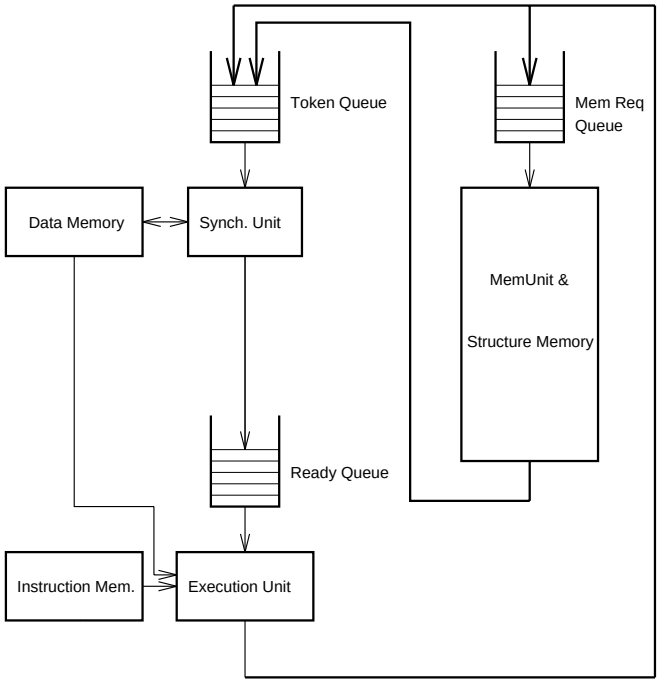

Figure 1: Abstract Model of a Processing Node.

A unique color is generated for each activation of a code-block such as a function or a loop. Data structures, such as arrays and records, are stored in a logically shared structure store. The results of thread execution are either written to the structure store or directly sent to their destination thread(s). A given thread activation can be executed on any processor. Since each thread is relatively small (10 to 30 instructions), global (dynamic) scheduling and near perfect load balancing is achieved by a simple hashing of the tag. We do not take into account the locality of communicating threads in the hashing of threads to processors. Part of this paper deals with exploring the amount of inter-thread locality that can be exploited.

The abstract logical structure of the processor model is presented in Figure 1. The local memory of each node consists of an *Instruction Memory* which is read by the *Execution Unit* and a *Data Memory* which is accessed by the *Synchronization Unit* and the Execution Unit. Inputs to a thread are stored in the Data Memory; when all the inputs have arrived, the corresponding thread is enabled. The data memory devoted to storing the inputs are also referred to as the matching store. The *Ready Queue* contains the *continuations* representing the enabled threads. There may be different contexts of the same thread that may be enabled at any given time either on the same node or on different nodes. The *Structure Memory* may be either distributed among the nodes, or among dedicated memory modules arranged in a dance-hall configuration. The *MemUnit* handles the structure memory requests.

For each instance of a thread, a small storage area (called a *framelet*) is allocated in the Data Memory to hold the incoming inputs to that thread. When the first input of a thread activation (i.e. instance) arrives to a node, the Synchronization Unit will first allocate a framelet and set the count of the total number of inputs in the framelet.

Each input token is stored in an appropriate slot within the framelet and the counter is decremented. When the count reaches zero, the thread is enabled to execute by making an entry in the Ready Queue. After the thread executes, the framelet is deallocated.

Pebble programs are represented in a form of dataflow graphs called MIDC. Each node of the graph represents a thread of machine independent instructions. Edges represent data paths through which tokens travel. In addition to the nodes and edges, there are pragmas and other specifiers to encode information (e.g., program-level constructs) that maybe helpful to the post-processors and program loaders. The use of low-level intermediate code in MIDC stems from the desire to have a degree of architectural freedom but at the same time expose various instruction latencies to the compiler. The objective is to have a reasonable platform on which to evaluate the code generation and Pebble architecture.

The MIDC code is generated from Sisal programs. Sisal [MSA+85] is a strict functional programming language with arrays and streams. The code generation is guided by the following objectives: (1) Minimize synchronization overhead, (2) Maximize intra-thread locality, (3) Assure non-blocking (and deadlock-free) threads, and (4) Preserve functional and loop parallelism in programs [RNB93, NRB94]. The resulting MIDC code can exploit many forms of parallelism including parallel loops and functions. Threads terminate at control graph interfaces for loops and conditionals, and at nodes for which the execution time is not statically determinable, such as function calls and memory (arrays and structures) accesses. More accurately, it terminates at the *use* of the values returned from the function calls and memory reads. Therefore, multiple calls and memory reads may be initiated in a thread. Structure store reads are turned into split-phase reads, with the initiator and consumer residing in different threads. These threads are then further optimized [RNSB94] at a global level that includes merging of threads to create larger threads. Merging of threads can take place across branches. Since the instruction execution order *within* a thread is constrained only by *true data dependencies*, the compiler orders the instructions by a simple topological ordering that maximize the potential exploitation of ILP by a super-scalar or VLIW processor. On the average, each thread has 10 to 30 instructions.

3 Communications Analysis

In this section, we describe the metrics used to analyze communication patterns and give simulation results based on a set of benchmarks. The metrics used are the token characteristics, input locality, and output locality.

3.1 Message Characteristics

For the purpose of characterizing tokens, we classify tokens into two major categories: *memory* tokens represent structure store reads and writes, and *regular* tokens represent non-memory related control and data tokens. The *regular* tokens are broken down into three types: *loop*, *call*, and *other*. *Loop* tokens represent tokens (data and control) used in distributing parallel work and in gathering their results including simple reductions. *Loop* tokens could represent a significant fraction of regular tokens, especially for programs with a large number of parallel `forall` loops and small loop bodies with large loop counts. *Call* tokens represent all tokens (data and control) involved in calling a function and returning its results. *Other* tokens represent all other tokens.

	Threads	Tokens	MPI
AMR	207,178	3,207,898	0.50
FFT	225,706	4,148,429	0.41
HILBERT	744,047	3,286,375	0.56
PSA	1,670,678	7,098,854	0.44
SDD	1,716,892	10,520,838	0.53
SGA	1,409,905	8,713,308	0.54
SIMPLE	1,333,209	9,269,106	0.56
WEATHER	882,508	8,045,883	0.56

Table 1: Benchmark Characteristics

Experimental results discussed in this paper are based on the following set of eight non-trivial Sisal programs:

- *AMR* is an unsplit integrator taken from an adaptive mesh refinement code at Lawrence Livermore National Laboratory.
- *FFT* is a one dimensional complex Fast Fourier Transform code.
- *HILBERT* computes the condition number for Hilbert matrix coefficients. It uses Linpack routines.
- *PSA* is a parallel scheduler code using a variation of simulated annealing to solve the problem.
- *SDD* solves an elliptic partial differential equation using the Symmetric Domain Decomposition method.
- *SGA* is a genetic algorithm program finding a local minima of a bowl-shaped function developed at Colorado State University.
- *SIMPLE* is a Lagrangian 2-D hydrodynamics code that simulates the behavior of fluid in a sphere developed at Lawrence Livermore National Laboratory.
- *WEATHER* is a one level barotropic weather prediction code and was originally developed at the Royal Melbourne Institute of Technology.

The experiments are conducted with our Bedrock simulator[1]. The characteristics of each benchmark are given in Table 1. The first and second columns give the number of threads executed and tokens generated, respectively per run. The third column gives the average number of matches performed per MIDC instruction executed (MPI). The average figures are fairly consistent of around 0.5 matches per instruction.

Where timed measurements are reported, the following architectural parameters are specified to the simulator: a 4-way issue super-scalar CPU execution unit per node with the instruction latencies of the Motorola 88110, an output network bandwidth of one token per node, the synchronization latencies of the EM-4: a pipelined synchronization unit with a throughput of one synchronization per cycle and a latency of three cycles on a mismatch (first reference to a framelet). We have assumed that inter-node communications take 50

[1] Bedrock is a cycle-level, discrete event Pebbles machine simulator with configurable parameters.

CPU cycles in network transit time for all messages unless they are specifically tagged as local. Remote memory reads take two network transit. We assume that *all* structure store reads and writes go through the interconnection network. Obviously, some of these messages can be made local by a judicious allocation of data structures. This, however, requires extensive static analysis in the compiler which is beyond the scope of this paper. These architectural parameter values are chosen to give a reasonable approximation of a real machine rather than exact measurements.

Table 2 shows the breakdown of tokens according to the different types. It shows that memory tokens represent a significant fraction of the total traffic, amounting to roughly a third on the average. This implies that techniques that reduce memory tokens are essential in achieving a significant reduction in the amount of network communications.

Among *regular* tokens it can be observed that *loop* tokens comprise 30-97% of the total. For AMR, FFT, and HILBERT the loop tokens represent a vast majority of regular tokens implying a significant amount of traffic devoted to handling parallelism. These results are not surprising considering that threads, in our model, are constrained to be no larger than a loop body or function body and are terminated at structure store accesses[2]. The three programs above are highly parallel with each parallel loop containing only few (1-3) threads.

Because our code generation and optimization in-lines all small functions, *call* tokens represent a relatively small proportion of all tokens. Therefore, except in the case of PSA, most regular tokens are either *loop* and *other* tokens.

3.2 Input Locality of Messages

We define input locality of a thread as the inverse of input latency. Input latency refers to the time delay between the arrival of the *first* token to a thread instance and that of the *last* token, at which time the thread can be enabled. Figure 2 shows the cumulative distributions of input locality in terms of processor cycles. The trend shows that for most programs, about 30% of threads have inputs that arrive within 10 cycles of each other; and in 50-60% of threads, inputs arrive within 100 cycles of each other. This is significant considering that each network transit takes 50 cycles. On the other hand, between 5% to 35% of threads have input latency greater than 900 cycles. These results are most useful in the design of a token storage hierarchy where short input latency threads are allocated in the cache and long input latency ones are migrated to the slower, longer term main memory. A cache architecture that exploits these properties is described and evaluated in [RN95].

3.3 Output Locality of Messages

One of the metrics related to message generations is the *output locality* which attempts to quantify the amount of locality in the output messages (tokens) that are generated by threads. It is measured by the ratio of the total number of regular tokens generated by a thread (M) and the number of distinct threads to which these tokens are sent (T). This measure attempts to capture the locality in the output destination of messages. Note that in any type of implementations, a thread destination will eventually be translated into a frame or framelet address. The output locality therefore captures the locality of writes to the frame or framelet store. The values of M, T and their ratio are shown in Table 3 for our benchmarks.

[2]Several optimizations at the global level attempt to make threads as large as possible [RNSB94].

	M	T	M/T
AMR	11.9	2.7	4.40
FFT	15.2	3.4	4.47
HILBERT	3.2	1.14	2.80
PSA	2.7	1.13	2.39
SDD	3.8	1.28	2.96
SGA	5.3	1.82	2.92
SIMPLE	4.8	1.50	3.21
WEATHER	7.2	1.94	3.71

Table 3: Output Locality

The table indicates that there exist a significant amount of output locality in most programs. It shows that, on the average, about 3-4 tokens are sent to each target thread. Except for AMR and FFT, the average number of targets are less than two. For both AMR and FFT, each thread generates a much larger number of regular tokens sent to a larger number of threads, compared with other benchmarks; however, the ratio M/T is only slightly larger than some of the other benchmarks. AMR and FFT both have large average thread sizes and each loop body has a smaller number of threads, as revealed in the next section. These results indicate that this output locality can be exploited by *bundling* tokens destined for a same thread into one message in order to reduce the overhead in message set-up and reception.

4 Overhead Reduction

In this section, given the results in the previous section, we present two techniques designed to reduce the amount of regular token traffic or the overhead. Again, we do not consider ways to reduce the remote memory accesses. For example, static data partitioning can reduce the amount of remote accesses. This, however, requires extensive compile-time static analysis which is beyond the scope of our present work. Instead, we assume that all structure store reads and writes still go through the network.

4.1 Grouping Technique and Measurements

As a first step toward reducing the regular token traffic, we first observe that *loop* and *call* tokens are the primary means by which parallelism is exploited. *Other* tokens represent intra-loop and intra-function thread communications that could be localized to a processor. In essence, by making sure that threads within a loop or a function body are allocated to the same processor, the communications between these threads remain local to that processor. This *allocation* strategy should also result in a higher locality and better exploits the memory hierarchy. In doing the grouping there is potential for reducing the program parallelism, care must be taken therefore to avoid such a situation.

Although the grouping idea is rather obvious, our objective is to quantify its possible benefit against the cost it might entail. The allocation of a group, either a loop or function body, to a processor is done by hashing the *color* portion

Types	Regular				Memory		
	Loops	Calls	Others	Total	Reads	Writes	Total
AMR	65.0	1.1	5.50	71.6	21.7	6.7	28.4
FFT	68.9	0.0	2.0	70.9	14.9	14.3	29.2
HILBERT	59.6	0.1	7.6	67.3	26.3	6.4	32.7
PSA	17.9	14.4	23.0	55.3	31.8	12.9	44.7
SDD	44.5	0.1	13.2	57.8	36.1	6.2	42.3
SGA	24.4	4.3	49.1	77.8	13.2	9.0	21.2
SIMPLE	28.0	5.0	32.7	65.7	29.0	5.2	34.2
WEATHER	38.6	0.8	35.5	74.9	19.8	5.3	25.1
Average	43.4	3.2	21.1	67.7	24.1	8.2	32.3

Table 2: Breakdown of All Messages in Percentage

of a token tag, rather than the entire tag. In addition, the compiler also generates code so that threads directly write to the data memory, rather than generating tokens that gets short-circuited, for intra-group communications.

Since different colors are generated for each `forall` loop iteration and for each call, each activation of a group would never cross the boundaries of loops or functions. We emphasize that grouping applies only to the processor allocation and does not entail any scheduling policy. The order of execution of threads within a group is still determined by the availability of data. However, a policy which schedules all enabled threads from the same group together should better exploit memory hierarchy. Also, the grouping of threads is performed whenever possible even when it might not be such a good idea (e.g., a large function body).

In summary, the execution of thread groups is characterized by

- A thread group is a statically determined set of threads that are allocated on a single node (processor).
- Different activations of the same group could be allocated on different nodes, their allocation is determined dynamically at run time (by the hashing on the color field).
- The order of thread execution within a groups is purely determined by the availability of data.

The characteristics of resulting thread groups are shown in Table 4. S_1^G represents the average number of threads in a group, and Π^G represents the average parallelism of a group in terms of threads. The table shows that each group consists of a relatively small number of threads. The table also shows the average number of instructions in a thread and the average parallelism within a thread, represented by S_1^T and Π^T respectively. FFT in particular has very large threads and large internal parallelism. Note that the average instruction level parallelism within a thread, Π^T, is very close to four instructions per cycle.

The second column indicates that intra-group parallelism is nearly one for all the benchmarks. In other words, even when there are infinite number of processors that can exploit all inter-thread parallelism, threads within a group execute nearly sequentially. Therefore, no parallelism is lost by grouping threads and allocating them on a single processor, which is an important objective of our code generation.

The result of thread grouping on reducing the amount of inter-processor communication is shown in Table 5. The

	S_1^G	Π^G	S_1^T	Π^T
AMR	2.47	1.00	30.79	5.29
FFT	1.65	1.00	44.35	4.77
HILBERT	3.91	1.00	7.95	2.41
PSA	4.72	1.04	9.65	2.69
SDD	3.77	1.01	11.61	3.05
SGA	5.43	1.01	11.45	3.39
SIMPLE	5.05	1.02	12.42	3.40
WEATHER	3.65	1.00	16.35	4.22
Average	3.83	1.01	18.1	3.65

Table 4: Thread and Group Characteristics

first two columns show the number of the *regular* tokens before and after the grouping. The last column show the percentage reduction of the *regular* tokens. The table shows that there are significant reductions in the amount of regular tokens generated (nearly 30%). Grouping only affects the *other* tokens, in fact eliminating most of these tokens. A main exception arises from sequential `forinit` loops passing data across iterations. As expected, some benchmarks, such as AMR and FFT, do not gain much from grouping due to the small number of *other* tokens.

4.2 Message Bundling Technique and Measurements

Output locality results suggest that it may be profitable to combine separate tokens to the same thread into one larger token. This *message vectorization* should reduce the number of network packets generated at the expense of a larger message size. It is worth noting that the *bundling* does not introduce any additional latency since the target thread cannot be enabled until all the inputs arrive anyway. Although *grouping* is only effective for the *other* tokens, *bundling* should be effective for both the *loop* and *call* tokens. Therefore, they are complementary techniques.

Table 6 shows the result of combined bundling and grouping. It shows the number of messages generated before and after applying the techniques. In the bundling technique, we limited the maximum token size to consist of no more than five data values. The reduction is most significant for the *loop* tokens. This is due to the fact that activating each loop iteration requires sending a lot of information to the same node. Almost all the *other* tokens are eliminated mostly due

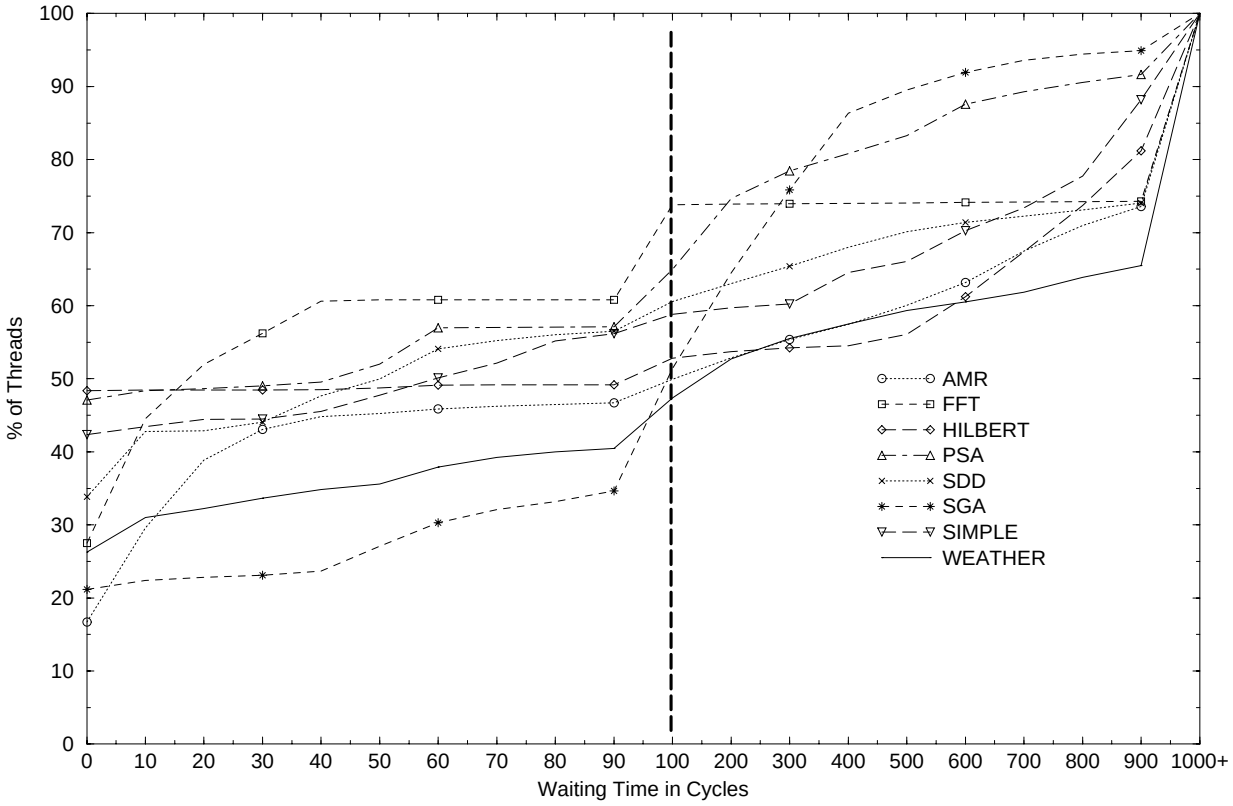

Figure 2: Input Locality in Cycles (Cumulative Distributions). For each benchmark, the indicated line plots the percentage of threads whose waiting time is within a given time. In the left half of the figure, each horizontal tick mark represents 10 cycles, and in the right half, each tick mark represents 100 cycles.

to the grouping.

The overall percentage reductions in the number of regular tokens is given in the last column of the table. The results are fairly uniform in that all programs result in significant reductions ranging from 68.2-86.2% with the average of nearly 80%. These result demonstrate that bundling and grouping are complementary techniques; when one is ineffective, the other makes up for it. For example, AMR and FFT have a small number of intra-group tokens to be eliminated, but bundling reduces the number of their *loop* and *call* tokens by more than a factor of four, resulting in overall reductions comparable to other benchmarks.

In the first column of Table 7, the overall percentage reduction in the number of *all* tokens generated is given. We see that more than half of all tokens are eliminated. The second column of the table gives the resulting MPI figures and shows a large drop in matches per instructions. This ranges from about 0.14 to 0.29 with the average of 0.21.

The larger average message size should result in slightly higher overhead per message processing. However, the network transit time should not be significantly affected if we assume a cut-through type of routing. This message vectorization should therefore be effective in reducing the overall communication overhead.

	% Reduction	MPI
AMR	56.6	0.19
FFT	55.4	0.14
HILBERT	45.9	0.29
PSA	41.0	0.23
SDD	42.7	0.29
SGA	68.1	0.14
SIMPLE	53.9	0.24
WEATHER	63.6	0.18
Average	53.4	0.21

Table 7: Overall Percentage Reduction of All Tokens and MPI

	Tokens (Millions) Before Grouping	Tokens (Millions) After Grouping	% Reduction
AMR	2.30	2.12	7.5
FFT	2.94	2.86	2.8
HILBERT	2.21	1.96	11.3
PSA	3.93	2.43	38.0
SDD	6.08	4.67	23.2
SGA	6.78	2.71	60.0
SIMPLE	6.09	3.54	41.9
WEATHER	6.03	3.37	44.1
Total	36.36	23.66	35.0

Table 5: The Effect of Grouping on Regular Tokens

	Loop Tokens (x10^6)		Call Tokens (x10^6)		Other Tokens (x10^6)		%
	Before	After	Before	After	Before	After	Reduction
AMR	2.09	0.47	0.04	0.01	0.18	0.00	79
FFT	2.86	0.64	0.00	0.00	0.01	0.00	78
HILBERT	1.96	0.70	0.00	0.00	0.25	0.00	68
PSA	1.27	0.41	1.02	0.53	1.63	0.07	74
SDD	4.68	1.58	0.01	0.00	1.39	0.00	74
SGA	2.13	0.65	0.37	0.21	4.28	0.07	86
SIMPLE	2.60	0.71	0.46	0.12	3.03	0.22	82
WEATHER	3.11	0.84	0.06	0.02	2.86	0.05	85
Total	20.70	6.00	1.96	0.89	13.63	0.41	78

Table 6: The Reduction in Tokens After Grouping and Bundling

5 Related Work

In essence, all multithreaded models that use code-block based frames [CSS+91, NPA92, SYH+89, Ian88] automatically group threads within a code-block. For them, since a frame do not span across processors, threads associated with the frame are bound to it and the processor. Although the technique presented in this paper uses a code-block based grouping, our model do not preclude other methods of grouping. Another method is to group according to the amount of communication between threads (e.g., two threads are grouped together if they communicate at least, say, three values). This method might be better in the presence of a large loop or function body, and reduce potential load-balancing problems due to grouping.

The use of *quanta* as a way to exploit locality at the inter-thread level has been first studied by Culler *et al.* for TAM [SCvE91, CSvE93]. It uses a software scheduling policy that favors threads within the currently executing quantum, and allows sharing of state between these threads (such as registers). Such a scheduling policy could be fruitfully applied to our model also.

By exploiting the *communication locality*, the cost of communication for large-scale multiprocessor can be minimized. In [JA92], an analytical model is developed to study the impact of exploiting the *communication locality*. The potential performance gain is limited by the degree of multithreading and the network limits. With the high degree of multithreading as embodied in the Pebbles model, the network represents a major limitation. Without expensive hardware solution to increase the capacity limits, the communication locality must be exploited to reduce the effective load on the network. Another approach which also exploits communication locality is through *chunking* method described in [SRBN95] where data locality is exploited by bundling several reads into one chunk read.

6 Conclusion

We have looked at several communication issues in a non-blocking, framelet-based multithreaded model in detail. We first measured the sources of inter-processor communications and determined their utilities. Then we introduced and studied the input locality to observe how inputs are related in each thread. This data may help guide compiler perform scheduling at the thread level and in the hardware design. Finally, we also introduced and measured the output locality of threads.

The result indicated that, roughly, there are about one-third of tokens that reads and writes to structure stores, another third involved in parallelism management, and the final third involved in control and data communications within a function or loop body. However, there was some significant differences among the benchmarks. It was found that there is a significant amount of locality present among thread inputs; in about 30% of cases, all inputs arrive to a thread arrive within 10 cycles of each other. This indicates that a cache can be efficiently utilized. Finally, it was found that a thread typically sends 3 or 4 tokens to each target thread, implying relatively high amount of output locality.

Based upon these results, we examined two simple tech-

niques, called the *grouping* and *bundling*, designed to reduce the amount of communications involved and the associated overheads by exploiting the communication locality. The *grouping* technique eliminates inter-processor communications between threads within a function or loop body by assigning these threads to the same processor. The *bundling* techniques vectorizes messages sent to the same processor in order to reduce mainly the overhead costs. Overall, the results indicate that we can eliminate the number of non-memory related inter-processor messages by about 80%, and that more than 50% of all inter-processor messages are eliminated.

As a future direction, techniques that reduce the amount of memory-related messages are essential in achieving a significant reduction in the network communications. This requires further static analysis at the language level to determine proper data partitioning and/or global caching.

References

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portfield, and B. Smith. The Tera computer system. In *Int. Conf. on Supercomputing*, pages 1–6. ACM Press, 1990.

[CSS+91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[CSvE93] D. E. Culler, K. E. Schauser, and T. von Eicken. Two fundamental limits on dataflow multiprocessing. In Cosnard, Ebcioglu, and Gaudiot, editors, *Proc. IFIP WG 10.3 Conf. on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism*, Orlando, FL, 1993. North-Holland.

[Cul94] D. E. Culler. Multithreading: Fundamental limits, potential gains, and alternatives. In R. Iannucci, editor, *Multithreading: A Summary of the State of the Art*, pages 97–138. Kluwer, 1994.

[DFK+92] W. J. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[HG93] H. H. Hum and G. R. Gao. Supporting a dynamic SPMD model in a multithreaded architecture. In *Proc. of Compcon Spring'93*, 1993.

[HTG94] H. H. Hum, K. B. Theobald, and G. R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proc. Int. Parallel Processing Symp.* IEEE CS Press, 1994.

[Ian88] R. A. Iannucci. Toward A Dataflow/Von Neumann Hybrid Architecture. In *Proc.* 15^{th} *Ann. Int. Symp. on Computer Architecture*, pages 131–140, 1988.

[JA92] K. Johnson and A. Agarwal. The impact of communication locality on large-scale multiprocessor performance. Technical Report LCS/TM-463, Massachusetts Institute of Technology, 1992.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc.* 19^{th} *Ann. Int. Symp. on Computer Architecture*, pages 156–167, 1992.

[NRB94] W. Najjar, L. Roh, and W. Böhm. An evaluation of medium-grain dataflow code. *Int. J. of Parallel Programming*, 22(3):209–242, June 1994.

[PC90] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proc.* 17^{th} *Ann. Int. Symp. on Computer Architecture*, pages 82–91, June 1990.

[RN95] L. Roh and W. A. Najjar. Storage hierarchy design in multithreaded architectures. Technical Report TR 95-102, Colorado State University – Computer Science Department, 1995.

[RNB93] L. Roh, W. A. Najjar, and W. Böhm. Generation and Quantitative Evaluation of Dataflow Clusters. In *Proc. Symposium on Functional Programming Languages and Computer Architecture*, pages 159–168, Copenhagen, Denmark, 1993.

[RNSB94] L. Roh, W. A. Najjar, B. Shankar, and A. P. W. Böhm. An evaluation of optimized threaded code generation. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'94)*, Montreal, Canada, 1994. North-Holland.

[SCvE91] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In J. Hughes, editor, *Proc. Symposium on Functional Programming Languages and Computer Architecture*, 1991.

[Smi81] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE (Real Time Signal Processing)*, 298:241–248, 1981.

[SRBN95] B. Shankar, L. Roh, W. Böhm, and W. A. Najjar. Control of loop parallelism in multithreaded code. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, 1995.

[SYH+89] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a data-flow single chip processor. In *Proc.* 16^{th} *Ann. Int. Symp. on Computer Architecture*, pages 46–53, May 1989.

[SYH+91] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. Pipeline optimization of a dataflow machine. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 225–246. Prentice Hall, 1991.

[Vas94] J. Vasell. A Fine-Grain Threaded Abstract Machine. In *Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'94)*, pages 15–24. North-Holland, 1994.

[WG89] W. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proc. 16^{th} Ann. Int. Symp. on Computer Architecture*, pages 273–280, May 1989.