Journal of Parallel and Distributed Computing **61**, 581–608 (2001)
doi:10.1006/jpdc.2001.1708, available online at http://www.idealibrary.com on IDEAL®

Resource Management in Dataflow-Based Multithreaded Execution[1]

Lucas Roh, Bhanu Shankar, Wim Böhm, and Walid Najjar

Department of Computer Science, Colorado State University, Fort Collins, Colorado 80523

Received March 1, 1997; revised September 6, 2000; accepted November 7, 2000

Due to the large amount of potential parallelism, resource management is a critical issue in multithreaded execution. The challenge in code generation is to control the parallelism without reducing the machine's ability to exploit it. Controlled parallelism reduces idle time, communication, and delay caused by synchronization. At the same time it increases the potential for exploitation of program data structure locality. In this paper, we evaluate the performance of methods to control program parallelism and resource usage in the context of the fine-grain dataflow execution model. The methods are in themselves not new, but their performance analysis is. The two methods to control parallelism here are *slicing* and *chunking*. We present the methods and their compilation strategy and evaluate their effectiveness in terms of run time and matching store occupancy. Communication is categorized in memory, loop, call, and expression communication. Input and output message locality is measured. Two techniques to reduce communication are introduced. *Grouping* allocates loop and function bodies on one processor and *bundling* combines messages with the same sender and receiver into one. Their effects on the total communication volume are quantified.

Key Words: multithreaded architectures; code generation; quantitative evaluation; control of parallelism.

1. INTRODUCTION

Fine-grain 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. It tolerates latency by rapidly switching among a set of ready threads thus improving the processor utilization. Both interprocessor communication and remote data access latencies can be masked, and therefore multithreading is especially suitable as an execution model for massively parallel processors. Current fine-grain multithreading models lie on various points along the von Neumann dataflow design spectrum. As designs move closer to the von Neumann world, thread

[1] This work is supported in part by NSF Grant MIP-9113268 and by DARPA Contract DABT63-95-0093. E-mail: roh,bohm@cs.colostate.edu, najjar@cs.colostate.edu.

0743-7315/01 $35.00

size tends to become larger and data structure locality can be better exploited. Examples of these designs include HEP [1], Tera [2], J-Machine [4], and M-Machine [5]. As designs move closer to dataflow, latencies are better tolerated and parallelism is more easily exploited. Examples are Monsoon [6], *T [7], EM-4 [8], and the EARTH project [9]. There also exist software abstractions of fine-grain multithreading as exemplified by TAM [10] that can be implemented on traditional multiprocessors such as the CM-5.

In multithreading models with strong dataflow heritage for which the code generator typically produces smaller thread grains requiring frequent synchronizations coupled with explosive parallelism, at any given time there may be many threads involved in either sending or waiting to receive messages or waiting for the execution unit to become available, all of which occupy some resources. Hence, the management of machine resources such as memory and communication network becomes more important. In many cases, the management of resources takes advantage of different forms of locality. In this paper, we quantitatively evaluate a set of compiler optimization techniques, namely *slicing*, *chunking*, *grouping*, and *bundling* that attempt to address the above issues.

Slicing mainly attacks the problem of controlling parallelism. *Chunking* mainly attempts to exploit data locality, similar to the traditional vectorization and loop unrolling methods. Simulation results are presented that compare the effectiveness of these techniques against code with unrestrained parallelism. The results indicate that the chunking method helps reduce the execution time and also shows an appreciable decrease in the utilization of the synchronization unit. The slicing method shows lower average and maximum matching store occupancies at the expense of increased execution time. By combining both techniques, it is possible to balance speedup with resource utilization.

Grouping reduces the amount of internode communication by allocating function or loop bodies on one processing node at the expense of possible load imbalance. *Bundling* combines tokens with the same sender thread and the same receiver thread into one message. For our experiments, tokens are classified into *Memory*, *Call*, *Loop*, and *Expression* tokens. The results show that grouping does not significantly reduce parallelism nor lead to poor load balancing, but eliminates most of the Expression tokens from the network traffic. On the other hand, bundling reduces the number of Call and Loop tokens. Together, the average reduction is about 80%.

In the Monsoon compiler study we use the simpler Livermore Loops [11]. In the other experiments, we use a set of larger Sisal benchmarks with sizes ranging from 500 to 2700 lines of source code.

- *AMR* is an unsplit integrator taken from an adaptive mesh refinement code at Lawrence Livermore National Laboratory.
- *BMK11A* is particle transport code developed to evaluate Cray Computer systems at Los Alamos National Laboratory.
- *FFT* is a one-dimensional 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 rest of this paper is organized as follows. In Section 2 we describe the execution model, including a basic processor model. In Section 3 we briefly summarize threaded code generation. Section 4 briefly describes the compiler transformations for the control of parallelism and reports on their effects on performance. This section contains an initial study of Monsoon code generation. Section 5 discusses communication reduction techniques and their effect on performance. Related work is discussed in Section 6. Concluding remarks are given in Section 7. Earlier versions of these results have been presented in [12] and [13].

2. EXECUTION MODEL

The multithreaded execution model used in this study is based on dynamic dataflow scheduling, where each actor represents a sequentially executing thread. A thread is a statically determined sequence of RISC-style instructions operating on registers. 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 must have a fixed execution time and cannot incur latency inside a thread. Therefore, latency incurring instructions have their consumers in other threads. Register values do not live across threads.

Inputs to a thread comprise all the data values required to execute the thread to its completion. 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. 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. 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)

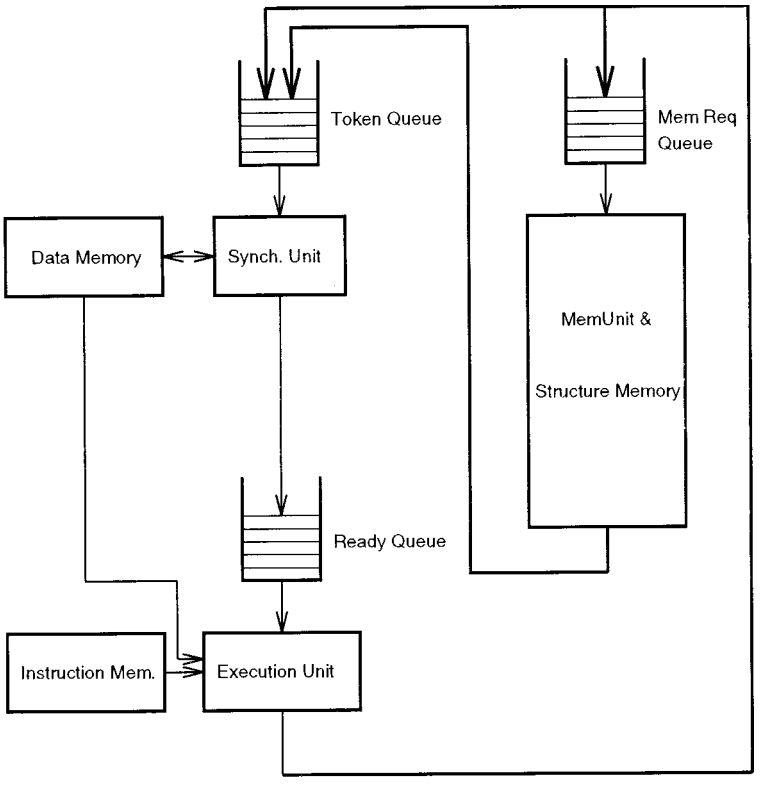

FIG. 1. Abstract model of a processing node.

scheduling and near perfect load balancing is achieved by a simple hashing of the continuation.

The logical structure of the processor model is presented in Fig. 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 Matching Store; when all inputs have arrived, the corresponding thread is enabled. The *Ready Queue* contains the continuations representing 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 dancehall configuration. The *MemUnit* handles the structure memory requests.

The following machine configuration is simulated using a cycle-level, discrete event machine. The machine has 10 processing nodes, each with a 4-way issue super-scalar CPU with the instruction latencies of the Motorola 88110 and 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 the first input. Problem sizes in our benchmarks are chosen to give reasonable simulation times and realistic processor utilization. All internode communications take 50 CPU cycles in network transit time. Every structure memory read takes the minimum of two network transits (one to send the request and another to send the reply). 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. The size of matching store is unlimited, and therefore can handle any amount of parallelism.

3. CODE GENERATION

Programs are represented in a dataflow graph form called MIDC [14]. Each node of the graph represents a thread of straight line von Neumann type instructions. Edges represent data paths along 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 may be helpful to postprocessors and program loaders.

Code generation is guided by the following objectives: minimize synchronization overhead, maximize intrathread locality, assure nonblocking (and deadlock-free) threads, and preserve functional and loop parallelism in programs. The first two objectives call for very large threads that maximize the locality within a thread and decrease the synchronization overhead. The thread size, however, is limited by the last two objectives. In fact, it was reported in [15] that blind efforts to increase the thread size, even when they satisfy the nonblocking and parallelism objectives, can result in a decrease in overall performance. Larger threads tend to have larger numbers of inputs and can result in a larger *input latency*, defined as 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 start executing [16].

Our nonblocking threads are generated from Sisal programs. Sisal [17] is a pure, first-order, functional programming language with loops and arrays. Sisal programs are initially compiled into a functional, block-structured, acyclic, data dependence graph form IF1 [18]. The functional semantics of IF1 prohibits the expression of copy-avoiding optimizations. This causes new data structures to be defined and the elements copied even when a single data element is modified, and this leads to a large amount of code just to copy data elements from one physical location to another even when it is unnecessary to do so.

An extension of IF1, called IF2 [19], allows operations that explicitly allocate and manipulate memory in a machine-independent manner through the use of buffers. A buffer is comprised of a buffer pointer into a contiguous block of memory and an element descriptor that defines the constituent type. All scalar values are operated by value and therefore copied to wherever they are needed. On the other hand, all of the fanout edges of a structured type are assumed to reference the same buffer; that is, each edge is not assumed to represent a distinct copy of the data. IF2 edges are decorated with pragmas to indicate when an operation such as "*update-in-place*" can be done safely, which dramatically improves the run-time performance of the system.

A top-down cluster generation process transforms IF2 into MIDC [20]. This phase breaks up the complex block-structured IF2 graphs so that threads can be

generated and wires the threads together. Initial reduction values are generated in the appropriate threads. Threads terminate at control graph interfaces for loops and conditionals, and at nodes, such as memory accesses, for which the execution time is not statically determinable, in order to satisfy the deterministic execution time objective. Threads do not cross function or loop boundaries and therefore useful forms of parallelism are preserved. Although it is not strictly necessary to have threads bounded by branches at this stage, doing so provides more flexibility in later stages.

The generated MIDC code is further optimized via a bottom-up stage [21] at both the intrathread and interthread levels. Intrathread optimizations consist of traditional optimizations including dead code elimination, constant folding/copy propagation, redundant instruction eliminations, and instruction scheduling to exploit the instruction level parallelism. Global optimizations include global versions of the above optimizations as well as redundant edge eliminations and merge operations that attempt to create larger threads by combining neighboring threads. The merging of threads also takes place across the branch instructions. The benchmark codes used in our experiments have thread sizes ranging from 10 to 30 MIDC instructions.

There are two types of loops in SISAL. *Iterative loops* have loop-carried dependencies and termination tests. *Parallel loops* have data independent loop bodies and known loop counts. Only the parallel loops are considered for parallelization and vectorization.

Since most parallel loops deal with arrays, it is instructive to know the layout of these data structures in memory. Figure 2 shows the layout of an array data structure containing an array descriptor and the data elements. SISAL arrays can start at any lower bound and can be of a variable size, and this information is encoded in the array descriptor. In order to reduce copy operations (e.g., when concatenating arrays), additional memory may be allocated on either side of the array data elements and several arrays can therefore be "*built-in-place*." This requires an "offset" value to specify where the logical array starts. Thus, the start of the array is given by adding the values of the `data pointer` to the `offset value`. All elements of that array are indexed off this resultant start address. With this layout, two memory latencies are required in order to fetch a single array element.

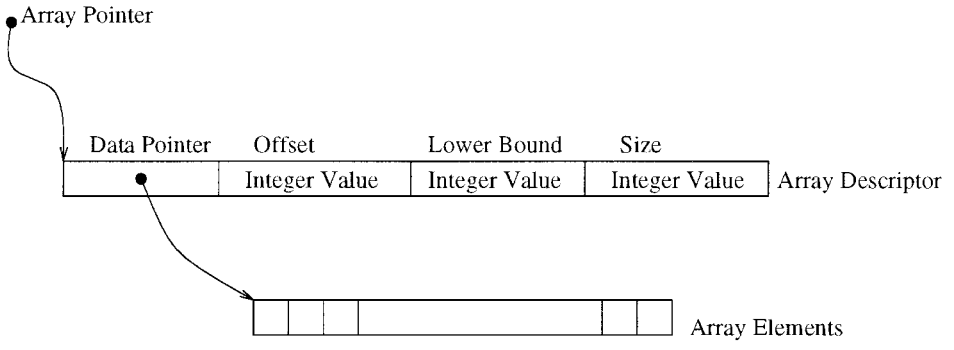

FIG. 2. Layout of arrays in MIDC.

4. CONTROL OF PARALLELISM

4.1. Loop Chunking

Loop Chunking stripmines a loop into a doubly nested loop, where the inner loop consists of fixed sized, consecutive chunks of loop bodies. This optimization is much like vectorization or loop unrolling (see Fig. 3). For the loop to be chunkable, the loop bodies must access consecutive array elements. For a loop of iteration space `n` and a machine chunk size of `c`, the number of workers is $\lfloor \frac{n}{c} \rfloor + 1$ if $n \bmod c \neq 0$ or $\frac{n}{c}$ otherwise. The code dealing with the irregularly sized chunk is executed only if such a chunk exists.

A split phase `FetchChunk` operator is used to fetch a chunk of data from structure memory. The semantics of this operation is defined as follows: memory is reserved in the target processor's data memory to hold the chunk. SISAL is a strict language, hence, all the data elements of an array will be available when the fetch occurs.

4.2. Loop Slicing

Loop slicing distributes a parallel loop over a fixed number of worker processors (see Fig. 4). Slicing reduces the resource load on the system in terms of the number of colors or activations required, with each slice taking one color rather than with each iteration. All workers perform at least $\lfloor \frac{n}{k} \rfloor$ work (n is the size of the iteration space, k is the number of workers), and $n \bmod k$ workers perform an additional iteration of work. It should be noted that any parallel loop can be sliced. Due to

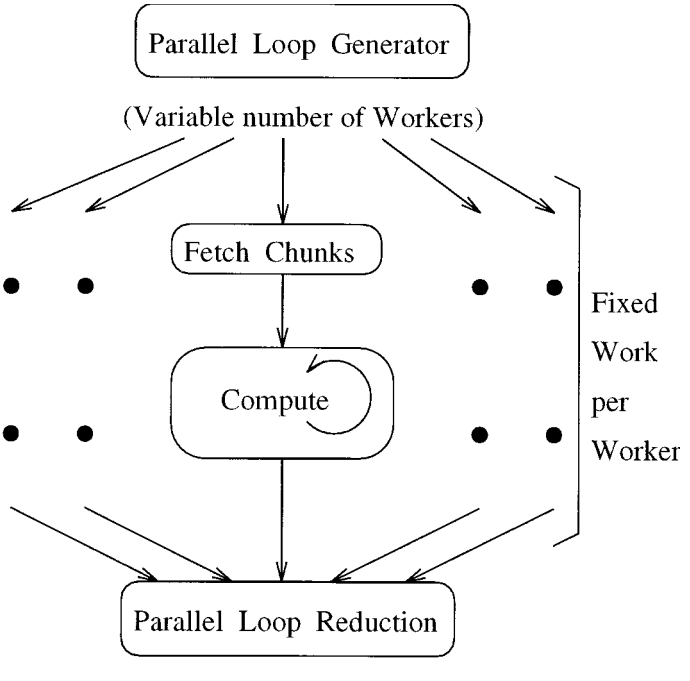

FIG. 3. Chunk control in loops.

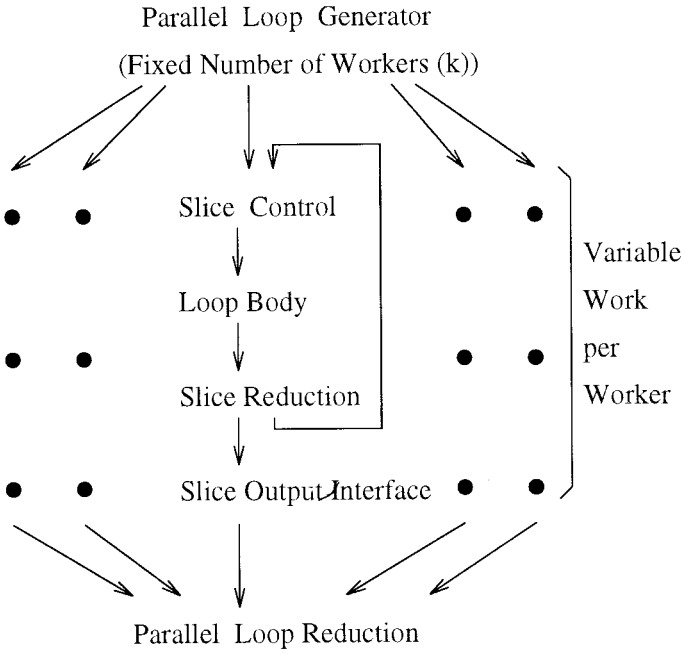

FIG. 4. Slice control in loops.

the semantics of SISAL there is no potential deadlock in the sliced code. This loop execution scheme is similar to K-bounding in Id [22]. The difference is that in slicing, the loops are block distributed, whereas in K-bounding the loops are cyclically distributed. The cyclic distribution is necessary for Id, because the loops can have loop-carried dependencies via nextified variables.

Each portion of the iteration space assigned to workers is executed in a sequential fashion. Inputs to all iterations of the iteration space are equal except for the index value. The index value port is identified and is updated at each execution of the loop body. The reduction required in the parent loop is also divided over the iteration spaces, reducing the amount of serial reduction. Reduction operators in SISAL (`sum`, `product`, `max`, and `min`) are commutative and associative. Thus, they can be reduced in any order.

4.3. Performance Evaluation of Slicing and Chunking

We evaluate the dynamic properties of our code before and after applying various combinations of slicing and chunking. The characteristics of the benchmark programs we use in this section, in terms of the number of parallel loops and chunkable loops are given, in Table 1. We note that SDD has the lowest percentage of chunkable parallel loops of around 29% and FFT has the highest percentage with 79%.

We compare four combinations of slicing and chunking: *R*, unconstrained parallel threaded code, used as the base case for all comparisons; *C*, loops chunked; *S*, loops sliced; and *CS*, loops chunked and other loops sliced. The following measures are used to evaluate performance:

TABLE 1

Program Characteristics of the Benchmarks

Program	Problem size [Time steps]	Parallel loops	Chunkable loops
AMR	$4 \times 80 \times 40$ [4]	87	34
FFT	2^{15}	13	11
HILBERT	100×100	65	30
SDD	2^6	80	23
SIMPLE	100×100 [5]	78	42
WEATHER	840 km [5]	81	29

- *Time*: the number of cycles taken by the program to execute.
- Ω_{avg}: the average occupancy of the matching store in terms of the number of threads that are waiting for inputs.
- $\Omega_{\max}$: the maximum occupancy of the matching store in terms of the number of threads that are waiting for inputs at any given time.
- U_P%: the processor utilization, i.e., the percentage of time the processors are busy.
- U_S%: the utilization of the synchronization unit, i.e., the percentage of time the synchronization unit is busy.

In evaluating the comparative performance between the different sets of parallelism control methods, the following measures will be used.

- Imp%: the percentage improvement of execution time (e) over the execution time of the unconstrained case (u), i.e., $((u/e)-1) * 100$.
- $R(\Omega_{\text{avg}})$% and $R(\Omega_{\max})$%: the ratios of space utilization over the space utilization of the unconstrained case.

The baseline performance of the unconstrained model is given in Table 2. The table shows that the processor utilizations range from a low of 15.3% for HILBERT up to 93.5% for AMR. The low utilization for HILBERT is due to the fact that a typical parallel loop body is relatively small with only one or two threads, and a

TABLE 2

Performance with Unconstrained Parallelism (R)

Bench	Time	U_P%	U_S%	Ω_{avg}	$\Omega_{\max}$
AMR	2294940	93.5	45.7	3774	22148
FFT	1154821	70.7	36.1	778	7199
HILBERT	2474720	15.3	13.1	259	3730
SDD	2907122	57.2	36.2	1265	11716
SIMPLE	7040310	54.5	38.1	28485	101758
WEATHER	1427385	67.8	54.1	4102	20775

significant fraction of the time is spent in the serial reductions of those parallel loops. Also, in general, the synchronization unit utilization is in the same order, albeit smaller, as the processor utilization.

4.3.1. Performance of chunking. Chunking exploits data locality and hence should decrease the execution time in addition to restraining parallelism. The improvement of the various benchmarks is given in the Table 3. The best performing chunk size for each benchmark is also presented. The experiment was conducted with chunk sizes of 8, 16, and 32. HILBERT showed the lowest improvement of 0.3%. FFT showed the best improvement of 34.7%. The chunkable loops in HILBERT have much smaller loop bodies than the nonchunkable loops and therefore the impact of chunking is minimal. The data access pattern of FFT is very regular and hence highly chunkable. Since parallelism is controlled only in those loops that can be vectorized/chunked, the occupancy of the matching store memory does not show any significant decrease. However, the table shows a noticeable reduction in the synchronization unit utilization. Since the matching is done a chunk at a time rather than a single value at a time, the utilization of the synchronization unit utilization should be reduced.

4.3.2. Performance of slicing. Slicing is used to control matching store memory occupancy. All parallel loops, including chunkable loops, in the benchmarks have been sliced in this experiment. The experiment has been conducted with the loops sliced with 10 or 20 workers each. In this experiment, space refers to snatching store memory.

In most of these experiments, we are trading time for space. The more the parallelism is throttled, the less space it uses and the more time it takes to complete the execution in general. This is evidenced in Table 4 where the slice sizes that favor execution time over the space are chosen, and in Table 5 where the space is favored over the time. In the tables, slice sizes are specified in terms of the number of worker processes that are used to execute the parallel loops. For instance, slice size 10,20 indicates that the innermost parallel loop would be split up between 10 worker processes and the outer, second level parallel loop is split up between 20 worker processes.

Table 4 shows that the processor utilizations are approximately equal to that of the unconstrained case. Good processor utilization implies that the parallelism has

TABLE 3

Performance with Inner Loops Chunked (*C*)

Bench	Chunk size	U_P%	U_S%	Imp%
AMR	32	95.2	29.4	17.5
FFT	32	76.8	15.8	34.7
HILBERT	8	15.0	12.8	0.3
SDD	32	55.1	31.9	3.3
SIMPLE	16	51.9	32.1	7.3
WEATHER	16	63.8	46.3	3.5

TABLE 4

Performance with Loops Sliced, with Best Execution Times

Bench	Slice size	U_P%	U_S%	Imp%	$R(\Omega_{avg})$%	$R(\Omega_{max})$%
AMR	10, 20	93.8	58.5	−8.2	28.8	15.1
FFT	20, 20	70.4	52.0	−12.5	21.5	10.9
HILBERT	20, 20	16.8	18.2	−18.1	60.2	77.4
SDD	20, 20	64.2	49.2	−5.4	73.2	33.5
SIMPLE	20, 20	51.5	41.1	−12.5	5.8	8.7
WEATHER	20, 20	74.9	64.4	−8.8	78.0	86.7

not been throttled to the extent where the processor is sitting idle when it should not. The improvement ranges from −5.4% in SDD to −18.1% in HILBERT. The average space used drops dramatically. The best saving is shown by SIMPLE, which utilizes on average 5.8% of what is required by the unconstrained (R) case. The worst is WEATHER, which requires 78.0% of the space occupied in the unconstrained case. When maximal occupancy is considered, the best saving is still shown by SIMPLE, utilizing just 8.7% of the maximum occupied in the unconstrained case. The worst saving is shown by WEATHER, saving just 13.3% of the space.

Table 5 shows the results for those cases where processor utilization remains fairly high but the space utilized is the lowest. In this case, the improvements drop even more with the range of −9.5% in AMR, to −24.9% in HILBERT. In the average space case, the best saving is shown by SIMPLE, requiring just 2.9% of the space. The worst is shown by SDD, requiring 51.6% of the regular space. In the maximum space case scenario, the best savings is again shown by SIMPLE with 4.2% ratio and the worst savings is shown by WEATHER, with 44.6% ratio.

4.3.3. Performance of combined chunking and slicing. The next logical step is to combine chunking and slicing and try for good execution times with low space utilization.

Table 6 shows the effect of combining chunking, with sizes as in Table 3, and the slicing with parameters as in Table 4. These settings favor better execution time over space saving and provide a good trade-off. In the case of WEATHER, the

TABLE 5

Performance with Loops Sliced, with Best Occupancy

Bench	Slice size	U_P%	U_S%	Imp%	$R(\Omega_{avg})$%	$R(\Omega_{max})$%
AMR	10, 10	92.3	57.6	−9.5	14.9	8.6
FFT	20, 10	69.5	51.3	−12.5	15.5	6.4
HILBERT	20, 10	15.4	16.6	−24.9	48.2	39.9
SDD	20, 10	60.0	46.0	−11.6	51.6	26.2
SIMPLE	20, 10	48.8	39.0	−16.9	2.9	4.2
WEATHER	10, 10	60.4	51.7	−19.2	29.3	44.6

TABLE 6

Performance with Inner Loops Chunked and Loops Sliced, with Best Processor Utilization

Bench	Chunk size	Slice size	Imp%	$R(\Omega_{max})$%
AMR	32	10, 20	+8.9	15.2
FFT	32	20, 20	+30.5	5.4
HILBERT	16	20, 20	−16.8	77.4
SDD	32	20, 20	−0.6	43.4
SIMPLE	16	20, 20	−1.6	5.9
WEATHER	16	20, 20	−0.6	86.6

space saving is on the order of 13% over the unbounded case for a time slow down of 0.6%.

Table 7 shows the combination of the chunking parameters from Table 3 and the slice parameters from Table 5, that favors space saving over execution time. This table shows greater savings in the matching store space utilization. However, as expected, the time taken to solve the problem increases. One extreme case is WEATHER, where space usage is about half of that shown in Table 4, but the execution took almost 20% longer. In the case of FFT, the improvement is 27.8% while only using 5.3% of the space used by the unconstrained case.

An important result is that the upper bound on the space usage is determined by the slicing scheme and is no longer dependent on the problem size.

4.3.4. Determining throttle values. Here we describe the method used to arrive at the best throttle parameters. For this we have chosen the benchmark `FFT`. Table 8 shows the results of various experiments executed for the `FFT` benchmark.

The first experiment is for the unconstrained parallelism case. Results indicate that processor utilization is about 71% and the maximum occupancy is 7199 thread slots in the matching memory.

The next set of experiments is run for the three different chunk sizes of 8, 16, and 32. As expected, the matching store occupancy does not change much in all the cases. In this particular case, the table indicates that all three chunk sizes yield similar performance with the chunk size 32 being the best at 34.7% improvement. Processor utilization remains fairly uniformly high in all the cases. The synchronization unit utilization drops by half.

TABLE 7

Performance with Inner Loops Chunked and Loops Sliced, with Best Occupancy

Bench	Chunk size	Slice size	Imp%	$R(\Omega_{max})$%
AMR	32	10, 10	+7.7	8.1
FFT	32	20, 10	+27.8	5.3
HILBERT	16	20, 10	−23.8	39.9
SDD	32	20, 10	−7.8	32.1
SIMPLE	16	20, 10	−6.2	3.5
WEATHER	16	10, 10	−18.7	44.7

TABLE 8

FFT: Arriving at the Best Throttle Value

Exp.	Chunk	Slice	Imp%	U_P%	U_S%	Ω_{avg}	Ω_{max}
R	—	—	1	70.7	36.1	778	7199
C	8	—	32.5	78.2	18.7	411	7199
	16	—	34.3	77.5	16.8	386	7203
	32	—	34.7	76.8	15.8	371	7203
S	—	10	−43.0	45.8	33.5	228	4380
	—	20	−11.9	70.9	52.2	428	5600
	—	20, 10	−13.7	69.5	51.3	121	461
	—	20, 20	−12.5	70.4	52.0	167	783
CS	32	20, 20	30.5	77.3	23.1	79	391
	32	20, 10	27.8	75.7	22.6	71	387

In the third experiment, the best slicing levels are determined. The first runs sliced only the innermost parallel loops. As stated earlier, the slice size is expressed in the number of workers that are chosen for parallel loops at the nesting level specified. The sizes chosen were 10 and 20. When the number of worker processors is 10, the throttle is too strong and processor utilization drops to 45.8%. The processor utilization with 20 workers remains fairly high at 70.9% and hence is better. The next step is to slice the parallel loops at the second level also, using the inner loop slice of 20. The experiments were run for 20,10 and 20,20. Processor utilization for both runs remain fairly high. Slicing with 20,20 is faster but slicing with 20,10 has a greater space saving.

The last set of experiments combines the chunking and slicing techniques, which will give us an acceptable execution speed at a low resource utilization. Two experiments are run for chunk size 32 and loop slicing 20,20 and 20,10, respectively. In this case, a chunking factor of 32 and a loop slicing factor of 20,20 would probably provide the best balance between the execution time and the space utilization.

4.4. Case Study: A Monsoon Implementation of SISAL–MIDC

This section presents a study of threaded code generation and resource management strategies for MIDC as ported to a one-node Monsoon Machine [23]. The compiler processes the MIDC code and generates a Monsoon Assembly (.masm) code file and Id code to interface with the type system. These in turn are processed by the Monsoon Assembler and Id Compiler to generate the required Monsoon Object Code. The Id World Interface to the machine is utilized to load the programs and provide simple Input and Output.

We use the simpler Livermore loops in this study [11], because the current compiler does not control the size of the generated threads, nor the number of activation frame slots in a thread. This has the effect that for large functions more frame slots are allocated than are available. There are methods to limit the number of frame slots, such as reusing the frame slots and splitting overly large threads. Optimized reuse of frame slots is implemented in the Id compiler, using linear

programming techniques [22]. Splitting (and merging) threads is already done in our compiler at the higher MIDC level. It just needs to be reemployed at this lower level. Before these transformations are implemented, it is important to study the effect of the naive algorithm.

The Monsoon is a distributed memory multithreaded multiprocessor architecture with a shared address space. Each processing unit is called a *node*. Each node has a portion of the address space of the machine under its control. A node contains a Processing Element (PE) and an I-Structure Board (IS). Within each PE there can be many threads of control, eight executing simultaneously in the pipeline and up to 32,000 awaiting execution. The state of each thread is contained in a *computation descriptor*, or CD, with five registers: a continuation register C, a value register V, and three temporary registers T1, T2, and T3. The continuation register defines the context in which the thread executes. It contains a pointer to instruction memory that indicates the next instruction to be executed and a *frame pointer*, which is used as the base address of an activation frame for a procedure invocation.

Data words have three *presence bits* associated with them defining the state a memory location is in. They affect the behavior of instructions that read and write the word. Presence bits are used in the implementation of synchronization protocols. The *Join Protocol* is used to synchronize two or more threads, implementing a dataflow-matching mechanism. The *Imperative Protocol* is used for imperative loads and stores. Any number of stores and loads may be performed on the location in any order. The *I-structure Protocol* synchronizes multiple consumers and a single producer. Initially, the location is *empty*. An I-Store operation stores a value in the location and sets the presence bits to *present*. Subsequently any number of I-fetch operations may be performed on the location. When a location is empty, I-fetch operations are deferred until an I-store operation takes place, after which the deferred fetches receive the stored value. The *M-structure Protocol* implements a mutual exclusion protocol. Initially, a location is empty. A Put operation stores a value and sets the presence bits to present. A subsequent Take operation reads the stored value and sets the presence bits back to the empty state. Multiple Take operations on an empty location are deferred, causing the location to enter into the lock-deferred state. A Put operation will satisfy exactly one of the deferred Take operations. Multiple Put operations result in an error.

Functions are implemented by code blocks. Each active code block has access to a portion of local memory called the *frame*. The amount of frame storage used by the code block is fixed at compile time. Frame slots are required for storing temporary values and for performing synchronizations. When calling a function, a frame is allocated, the addresses of the function and the frame are combined to produce a *context C*, and argments and a return address are sent to C. This Sisal calling convention is simpler than the Id version as Sisal is a strict language and does not support curried and higher order functions.

In determining the multithreaded computation model for Monsoon, three options were tested. In the *dataflow model* threads are synchonized using the Join Protocol.

It is also possible to use a *barrier* to test for the availability of all thread inputs, as threads are strict and thus cannot start until all of their inputs are available.

Instead of values being copied, as in the dataflow model, they are accessed in the activation frame where they are stored. There are two options for global memory operations. The *nonstrict barrier model* uses the split-phase I-structure Protocol, where the initiator and consumer of a transaction live in separate threads. The second, the *strict barrier model*, relies on the strict semantics of Sisal, guaranteeing that when a barrier is passed, all inputs to the threads triggered by the barrier are available; also, all array data to be read by the threads is completely defined, and thus the Imperative Protocol can be used.

The three models described above are compared using Livermore loops [11]. The dataflow model executes 30% more instructions than the nonstrict barrier model, as too many values are copied. The nonstrict barrier model in turn executes 9% more instructions than the strict barrier model, which is selected for further optimization. General purpose MIDC threads initiating parallel loops are merged and optimized for Monsoon. Copy propogation and redundant code elimination are implemented. Contexts, and thus frames, are merged, avoiding many relatively expensive context creations. Merging contexts agressively is very effective but causes the problem that frames get bigger than the allowed maximum. This is why some of the Livermore loops are not used at this time. The problem can be solved by reusing frameslots for variables that do not live at the same time. The analysis for this is similar to register allocation and is based on graph coloring. Table 9 shows

TABLE 9

Strict Barrier Model before and after Optimization in Machine Cycles

		Execution time		
Program	Size	Unoptimized	With basic optimizations	Imp%
Loop 1	990	2,181,000	1,779,000	18.43
Loop 2	101	1,648,000	1,318,000	20.02
Loop 3	1001	1,405,000	1,109,000	21.06
Loop 4	35	1,585,000	1,253,000	20.84
Loop 5	1001	1,876,000	1,468,000	21.74
Loop 6	64	2,440,000	2,014,000	17.45
Loop 7	995	2,042,000	1,638,000	19.78
Loop 8	20	623,016	369,558	40.68
Loop 9	101	335,990	290,582	13.51
Loop 10	101	530,553	487,703	8.07
Loop 11	1001	1,824,000	1,441,000	20.99
Loop 12	1000	2,072,000	1,686,000	18.62
Loop 16	75	1,241,000	1,111,000	10.47
Loop 17	101	532,146	461,974	13.18
Loop 19	101	346,670	275,619	20.49
Loop 20	100	629,216	541,945	13.86
Loop 21	10	383,461	333,682	12.98
Loop 22	101	360,993	314,201	7.56
Loop 24	1001	1,238,000	1,046,000	15.50

the significant improvements made due to these optimizations. The measurements include the cycles required to trap to the run-time system.

4.4.1. Chunking and slicing on monsoon. A total of three chunking and slicing experiments are performed. The first deals with the effect of chunking, the second with the effect of slicing, and the third with their combined effect.

Chunking on Monsoon stripmines an inner loop into a nested loop with a constant sized, tight, highly optimizable, inner loop. The MIDC `FetchChunk` operator computes the start address of a chunk. Chunking creates fewer and bigger threads and allows loop inputs to be shared. The effect of chunking on the Livermore Loops with varying chunk sizes is given in Table 10 and is compared to the unbounded case. The results show that chunking has an enormous effect on performance.

Loops 2, 4, and 5 are essentially iterative loops, but still benefit from chunking. In the cases of loop 2 and 5 this is because the initial array can be constructed in a chunkable parallel loop and there is a chunkable reduction. Loop 4 is an iterative outer loop with very small parallel inner loops giving rise to one optimizable chunk.

For chunking to be most effective, the size of the chunk should be approximately one-eighth of the size of the iteration space of the innermost loop. This creates eight

TABLE 10

Effect of Chunking and Slicing on Livermore Loops

	Chunking						Slicing	
Loop	Chunk size	Imp%	Chunk size	Imp%	Chunk size	Imp%	Number of slices	Imp%
Loop 1	64	687	128	727	256	595	8	472
Loop 2	64	546	128	569	256	510	8	413
Loop 3	64	1057	128	1114	256	922	8	596
Loop 4	64	712	128	752	256	649	8	491
Loop 5	64	302	128	305	256	294	8	258
Loop 6	8	108	16	108	32	108	8	280
Loop 7	64	703	128	753	256	556	8	464
Loop 8	8	146	16	147	32	144	8	130
Loop 9	8	207	16	225	32	227	8	252
Loop 10	8	135	16	135	32	143	8	211
Loop 11	64	309	128	312	256	300	8	263
Loop 12	64	859	128	904	256	729	8	531
Loop 16	8	280	16	320	32	290	8	289
							8,8	315
Loop 17	8	173	16	185	32	186	8	172
Loop 19	8	163	16	170	32	170	8	171
Loop 20	8	163	16	174	32	173	8	177
Loop 21	8	178	16	186	32	17	8	156
							8,8	157
Loop 22	8	167	16	179	32	178	8	216
Loop 24	64	213	128	216	256	201	8	199

large threads for a particular loop and optimally utilizes the pipeline of the machine.

Slicing on Monsoon allocates a fixed number of contexts to execute a loop. Each context executes a slice of loop bodies sequentially. In establishing the effect of slicing, the lesson learned from the chunking experiment is kept in mind: the number of threads of execution should be around eight. Table 10 compares the results of slicing to the unbounded case.

In two of the benchmarks, Loop 16 and Loop 21, improvements are shown when they are sliced at the second level. In 13 of the benchmarks chunking performs better than slicing. In these benchmarks, the inner parallel loops are chunkable, and control for chunking is very simple. In the loops where slicing performs better, the innermost loops are not chunkable. Slicing helps in reducing the execution time of all loops, both chunkable and nonchunkable.

Chunking and slicing are not mutually exclusive and can be combined. The codes where slicing performs better than chunking are run with both slicing and chunking turned on. Table 11 shows that in four cases the combination performs better than any one of the two, and in two cases (loop 6 and 10) the overhead of chunking is larger than its benefit and pure slicing is the best approach.

4.4.2. Further optimizations. The Sisal compiler presented by no means generates the best possible code. There are short-comings that need to be addressed to generate much more efficient code. As mentioned earlier, reuse of frame slots will reduce frame sizes. As a simple example, frame slots can be reused for different threads in the same frame slot that are guaranteed not to run concurrently.

The current implementation of parallel reduction uses an M-structure array to store the intermediate results that are reduced in parallel in a separate thread, which requires its own control steps. Eliminating this array and using single M-structure elements in a parallel reduction step would help reduce the load on the heap and could result in the ability to execute larger problems. This optimization would also reduce the number of instructions executed.

The sequential loop schema utilized in the implementation is not very efficient as evidenced by the difference between the execution times of Livermore Loop 2. Any scheme to tighten the control of these loops will considerably speed up the code.

TABLE 11

Chunking and Slicing Combined

Program	Size	Imp %
Loop 6	64	273
Loop 9	101	283
Loop 10	101	209
Loop 19	101	180
Loop 20	100	186
Loop 22	101	228

5. COMMUNICATION OVERHEAD: CHARACTERISTICS AND REDUCTION

5.1. Token Characteristics

We classify tokens into two major categories: *Memory* tokens represent structure store reads and writes, and *Regular* tokens represent nonmemory related control and data tokens. The *Regular* tokens are broken down into three types: *Loop*, *Call*, and *Expression*. *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. *Expression* tokens represent all other tokens.

Relevant characteristics of the benchmarks used in this section are given in Table 12. The first and second columns give the number of threads executed and tokens generated per run. The third column gives the average number of matches performed per MIDC instruction executed (*MPI*), which are around 0.5 matches per instruction.

Table 13 shows the breakdown of tokens according to their types. 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 communication.

Among *Regular* tokens it can be observed that *Loop* tokens comprise 30 to 97% of the total. For AMR, FFT, and HILBERT, the loop tokens represent the vast majority of the 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. Several optimizations at the global level attempt to make threads as large as possible [21]. These three programs are highly parallel with each parallel loop containing only a few (1–3) threads.

TABLE 12

Benchmark Characteristics

	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 13

Breakdown of All Messages in Percentages

Types	Regular				Memory		
	Loops	Calls	Expression	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

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* or *Expression* tokens.

5.2. *Input Locality*

We define the *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.

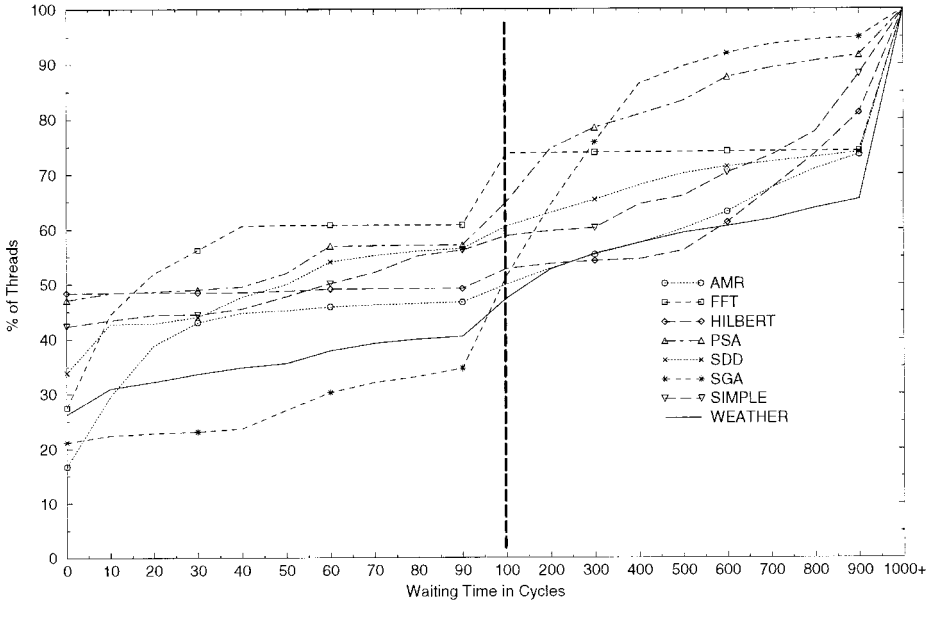

FIG. 5. Input latency in cycles (cumulative distributions). For each benchmark, the indicated line plots the percentage of threads whose waiting time is within a given time. In the left half of the figure, each horizontal tick mark represents 10 cycles, and in the right half, each tick mark represents 100 cycles.

Figure 5 shows the cumulative distribution of input latencies 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 for 50–60% of the 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 and 35% of the threads have input latencies 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 threads are migrated to the slower, longer term main memory. A cache architecture that exploits these properties is described and evaluated in [24].

5.3. Output Locality

Output locality quantifies the amount of locality in the destinations of the tokens generated by threads. It is defined as the ratio of the total number of tokens generated by a thread (M) and the number of distinct destination threads to which these tokens are sent (T). The values of M, T and their ratio are shown in Table 14 for our benchmarks.

The table indicates that there exist a significant amount of output locality in most programs. It shows that, on the average, about three to four tokens are sent to each target thread. Except for AMR and FFT, the average number of targets is 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 when 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.

The results in Table 14 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 setup and reception. We present two techniques to reduce the amount of regular token traffic: *grouping* and *bundling*. Note that we do not consider ways to reduce the remote memory accesses. For example, static data partitioning can reduce the amount of remote accesses. This requires extensive compile-time analysis

TABLE 14

Output Locality

	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

and goes beyond the scope of this paper. Instead, we assume that all structure store reads and writes go through the network.

5.4. Grouping

As a first step toward reducing the regular token traffic, we observe that *Loop* and *Call* tokens are the primary means by which parallelism is exploited. *Expression* tokens represent intraloop and intrafunction thread communication 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 communication between these threads is localized. This *allocation* strategy should also result in a higher locality and should better exploit the memory hierarchy.

Although the grouping idea is rather obvious, our objective is to quantify its possible benefit against the cost it might entail. Particularly, there is a potential for reducing exploitable parallelism, and we need to check whether this reduction is acceptable. The allocation of a thread group, comprising either a loop or function body, to a processor is done by hashing only the *color* portion of a token tag, rather than the entire tag. In addition, the compiler generates code for the threads to directly write to the data memory, rather than form and send tokens. The order of execution of threads within a group is still determined by the availability of data. A policy which schedules all enabled threads from the same group together could better exploit the memory hierarchy. This needs further investigation. Also, the grouping of threads is currently performed independent of the size and internal parallelism of the function or loop body. In summary, the execution of thread groups is characterized as follows.

- 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 group is purely determined by the availability of data.

The characteristics of the resulting thread groups are shown in Table 15. 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 intragroup parallelism is nearly one for all the benchmarks. In other words, even when there is an infinite number of processors that can exploit all interthread parallelism, threads within a group execute nearly sequentially. Therefore, no significant parallelism is lost by grouping threads on a single processor.

TABLE 15

Thread and Group Characteristics

	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

The effect of thread grouping on interprocessor communication is shown in Table 16. The first two columns show the number of the *Regular* tokens before and after thread grouping. The last column shows 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 *Expression* tokens, in fact eliminating most of these. 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 their small number of *Expression* tokens.

5.5. *Bundling*

The strong output locality suggests that it may be profitable to combine separate tokens, destined to the same thread, into one large message. This *message vectorization* should reduce the number of network packets generated at the expense of a larger message size. It is worth noting that *bundling* does not introduce any additional latency since the target thread cannot be enabled until all the inputs arrive anyway.

TABLE 16

The Effect of Grouping on Regular Tokens

	Tokens (Millions) before grouping	Tokens (Millions) after grouping	Reduction (in %)
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 17

The Reduction in Tokens after Grouping and Bundling

	Loop tokens ($\times 10^6$)		Call tokens ($\times 10^6$)		Expression tokens ($\times 10^6$)		
Bench	Before	After	Before	After	Before	After	Reduction (in %)
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

Grouping is effective for *Expression* tokens and optimizes messages away for them. Therefore *bundling* could applied to *Loop* and *Call* tokens.

Table 17 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 message size to contain no more than five data values. The reduction is most significant for *Loop* tokens. This is due to the fact that activating a loop iteration requires sending a lot of information to the same node. Almost all the *Expression* tokens are eliminated due to grouping.

The percentage reduction in the number of regular messages 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 to 86.2% with the average of nearly 80%. These results 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 intragroup 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.

TABLE 18

Overall Percentage Reduction of All Tokens and *MPI*

	Reduction (in %)	*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

In the first column of Table 18, the 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. However, the network transit time should not be significantly affected if we assume a cut-through type of routing. Message bundling should therefore be effective in reducing the overall communication overhead.

6. RELATED WORK

Parallelism control was first proposed in fine-grain dataflow architectures, due to the large amount of parallelism that was being generated. This amount was even large enough to cause resource deadlocks, which may be partly the cause of the eventual abandonment of fine-grain architectures in favor of coarser grain multithreaded architectures. However, the problem of matching program and machine parallelism has resurfaced in multithreaded architectures, as problem sizes have grown larger. The two main parallelism control mechanisms proposed for fine-grain architectures are *Throttling* of tasks [25–27] and *K-bounding* [28]. In addition, Egan *et al*. [29] have proposed methods of slicing the iteration space.

Throttling is a run-time method of controlling parallelism. New activation requests may be suspended if the run-time mechanism deems that there is too much parallelism, based on availability of resources. The suspended process is reactivated some time later. When an active process finishes, its activation name can be used by another process. When sufficient resources become available, suspended processes can be unsuspended. There is no notion of suspending an active process. Processes must be fairly large, otherwise they lead to too much throttle overhead. Processes that are too large would have large internal parallelism, which would cause resource deadlocks. The key to throttle control is to find the right balance between the exploitation of parallelism and the use of resources.

K-bounding is another method proposed to control fine-grain parallelism in loops. The compiler analyzes the code and determines the maximum resource usage for a loop cycle. The run-time system decides the number of loop cycles that can be allowed to execute in parallel, based on the activity level of the machine and the static information of maximal resource usage. Machine resources are recycled and reused.

The slicing proposed by Egan *et al*. splits the iteration space between k workers. Each of the workers executed iterations in steps of k. This involves the recycling of colors. However, some additional work must be performed to reestablish the order of the results coming from the loop body to that of the context enclosing the loop invocation.

Teo and Böhm [30] have proposed a method of chunking iterative instructions by letting them generate a fixed number of tokens with incremental indices in the tag. After that, inputs to the iterative instruction are recycled, which stops them from swamping the machine with an unbounded amount of tokens and allows them to be controlled.

Most multithreaded models that use code-block-based frames [7, 8, 10, 31] group threads within a code-block, and bind code-blocks to processors. Although the grouping technique presented in this paper uses the same code-block-based approach, our model does not preclude other methods, such as grouping according to the amount of communication between threads. This method might be useful in the presence of a large loop or function body and might reduce potential load balancing problems.

The use of *quanta* as a way to exploit locality at the interthread level has been first studied by Culler *et al.* for TAM [32, 33]. 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.

By exploiting *communication locality*, the cost of communication for large-scale multiprocessors can be minimized. In [34], 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 in our model, the network represents a major limitation. Without an expensive hardware solution to increase the capacity limits, communication locality must be exploited to reduce the effective load on the network. Another approach which also exploits communication locality is to bundle several reads into one chunk read [12].

The *filaments project* [3], which addresses issues in fine-grain multithreading from a von Neumann point of view, has some interesting approaches to controlling parallelism. Filaments are clustered ("implicitly coarsened") statically by inlining. Also, dynamic patterns of use of filaments are recognized and optimized. Filaments also has an adaptive run-time approach to data placement. Data placement strategies can be dynamically adapted, and used by the same filaments that are moved to the same node.

7. CONCLUSION

There are valid reasons to control parallelism and to take advantage of data locality in the existing and proposed multithreaded models. In this paper, we examined chunking and slicing techniques that address the above issues. Our experimental results indicate that slicing schemes remove the upper bound of space utilization from the realm of problem size to the size of the throttle that is used. Chunking improves the execution time and reduces the load on the synchronization unit, but it does not provide any space savings. The combination of the two techniques, each having different goals, helps to balance the execution time with the resource usage. Finding the right balance is not an easy task. In adopting the right throttle, it is important to examine the processor utilization, as well as the synchronization unit utilization.

We have defined and quantified the input and output message locality of our programs. The high degrees of input locality suggest that a cached implementation of the synchronization unit would be beneficial. Locality can be increased by grouping threads together on one processor (i.e., those threads belonging to the same loop or

function body). The high degree of output locality can be exploited by bundling messages. The combination of the two techniques reduces the message traffic by 80% in our benchmarks.

REFERENCES

1. B. J. Smith, Architecture and applications of the HEP multiprocessor computer system, *SPIE* (*Real Time Signal Process.*) **298** (1981), 241–248.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portfield, and B. Smith, The Tera computer system, *in* "Int. Conf. on Supercomputing," pp. 1–6, Assoc. Comput. Mach., New York, 1990.
3. V. W. Freeh, D. K. Lowenthal, and G. R. Andrews, Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations, *in* "First Symposium on Operating Design and Implementation," pp. 201–212, 1994.
4. 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) (April 1992), 23–39.
5. M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, The m-machine multicomputer, *in* "Proc. Int. Symp. on Microarchitecture," November 1995.
6. G. M. Papadopoulos and D. E. Culler, Monsoon: An explicit token-store architecture, *in* "Proc. 17th Int. Symp. on Computer Architecture," pp. 82–91, June 1990.
7. R. S. Nikhil, G. M. Papadopoulos, and Arvind, *T: A multithreaded massively parallel architecture, *in* "Proc. 19th Int. Symp. on Computer Architecture," pp. 156–167, May 1992.
8. S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, An architecture of a data-flow single chip processor, *in* "Proc. 16th Int. Symp. on Computer Architecture," pp. 46–53, May 1989.
9. H. Hum, O. Macquelin, K. Theobald, X. Tian, G. Gao, P. Cupryk, N. Elmassri, L. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu, A design study of the EARTH multiprocessor, *in* "Proc. Int. Conf. on Parallel Architectures and Compilation Techniques," 1995.
10. 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," pp. 164–175, 1991.
11. J. T. Feo, "The Livermore Loops in Sisal," Technical Report, UCID-21159 Computing Research Group, Lawrence Livermore National Laboratory, Livermore, CA 94550, August 1987.
12. 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.
13. L. Roh and W. A. Najjar, Analysis of communication and overhead reduction in multithreaded execution, *in* "Proc. Int. Conf. on Parallel Architectures and Compilation Techniques," 1995.
14. W. Böhm, W. A. Najjar, B. Shankar, and L. Roh, An evaluation of bottom-up and top-down thread generation techniques, *in* "Proc. Int. Symp. on Microarchitecture," 1993.
15. 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," pp. 159–168, Copenhagen, Denmark, 1993.
16. W. A. Najjar, W. M. Miller, and W. Böhm, An analysis of loop latency in dataflow execution, *in* "Proc. 19th Int. Symp. on Computer Architecture," pp. 352–361, Gold Coast, Australia, 1992.
17. 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.

18. S. K. Skedzielewski and J. Glauert, "IF1: An Intermediate Form for Applicative Languages," reference manual, Version 1.0, Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.
19. M. Welcome, S. Skedzielewski, R. K. Yates, and J. Ranelleti, "IF2: An Applicative Language Intermediate Form with Explicit Memory Management," Technical Report TR M-195, Lawrence Livermore Laboratory, University of California, December 1986.
20. B. Shankar, W. Böhm, and W. A. Najjar, Top-down thread generation for SISAL, *in* "Sisal '93," 1993.
21. 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," Montreal, Canada, 1994.
22. D. E. Culler, "Managing Parallelism and Resources in Scientific Dataflow Program," Ph.D. thesis, MIT, June 1989.
23. J. Hicks, D. Chiou, B. Seong Ang, and Arvind, Performance studies of Id on the Monsoon dataflow system, **18** (1993), 273–300.
24. L. Roh and W. Najjar, Design of storage hierarchy in multithreaded architectures, *in* "Proc. Int. Symp. on Microarchitecture," pp. 271–278, November 1995.
25. C. A. Ruggiero and J. Sargeant, Control of parallelism in the Manchester dataflow computer, *in* "Lecture Notes in Computer Science," No. 274, pp. 1–15, 1987.
26. D. F. Snelling, "The Design and Analysis of a Stateless Data-Flow Architecture," Ph.D. thesis, Computer Science Department, University of Manchester, Manchester, UK, 1993.
27. J. Gurd and D. Snelling, Self-regulating workload in the manchester data-flow computer, *in* "Proc. Int. Symp. on Microarchitecture," November 1995.
28. D. E. Culler, "Resource Management for the Tagged Token Data Flow Architecture," Technical Report, TR-332 Laboratory for Computer Science, MIT, January 1985.
29. G. K. Egan, N. J. Webb, and A. P. W. Böhm, Some architectural features of the CSIRAC II data-flow computer, *in* "Advanced Topics in Data-Flow Computing" (J.-L. Gaudiot and L. Bic, Eds.), pp. 143–174, Prentice–Hall, New York, 1991.
30. Y. Teo and W. Böhm, Resource management and iterative instructions, *in* "Advanced Topics in Data-Flow Computing" (J.-L. Gaudiot and L. Bic, Eds.), pp. 481–500, Prentice–Hall, New York, 1991.
31. R. A. Iannucci, Toward a dataflow/Von Neumann hybrid architecture, *in* "Proc. 15th Int. Symp. on Computer Architecture," pp. 131–140, 1988.
32. K. E. Schauser, D. E. Culler, and T. von Eicken, Compiler-controlled multithreading for lenient parallel languages, *in* "Proc. Symposium on Functional Programming Languages and Computer Architecture" (J. Hughes, Ed.), 1991.
33. D. E. Culler, K. E. Schauser, and T. von Eicken, Two fundamental limits on dataflow multiprocessing, *in* "Proc. IFIP WG 10.3 Conf. on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism, Orlando, FL, 1993" (Cosnard, Ebcioglu, and Gaudiot, Eds.), North-Holland, Amsterdam, 1993.
34. K. Johnson and A. Agarwal, "The Impact of Communication Locality on Large-Scale Multiprocessor Performance," Technical Report LCS/TM-463, MIT, 1992.

LUCAS ROH is currently the President and CEO of Hostway Corporation. Before he co-founded Hostway, he was a staff scientist at the Mathematics and Computer Science Division of Argonne National Laboratory conducting research in automatic differentiation and compilers. He received his B.A. in physics from the University of Chicago and Ph.D. in computer science from Colorado State University.

BHANU SHANKAR received his Bachelor of Computer Science and Engineering in 1989 from the R.V. College of Engineering, Bangalore, India, and his Ph.D. from Colorado State University in 1995.

He worked at Microtec Research Incorporated and is now with Intel Corporation. His interests are in static binary translation, code generation, and optimization. He is currently the technical lead for developing and maintaining a high-performance Fortran 95 front end.

A. P. WILLEM BÖHM is a professor at Colorado State University. He received his Ph.D. at the University of Utrecht. He worked at the CWI Amsterdam on Algol 68 and at Manchester University on Sisal and Dataflow. His research interests are algorithm design, programming languages, and compilation for parallel machines. He is currently involved in the design and implementation of a high-level, algorithmic language targetting Reconfigurable Systems based on Field Programmable Gate Arrays.

WALID A. NAJJAR is an associate professor in the Department of Computer science and Engineering at the University of California Riverside. He received his M.S. and Ph.D. in Computer Engineering from the University of Southern California in 1985 and 1988, respectively, and a B.E. in Electrical Engineering from the American University of Beirut in 1979. His research interests are reconfigurable computing and multithreaded and parallel computer architecture.