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

Control of Loop Parallelism in Multithreaded Code*

Bhanu Shankar, Lucas Roh, Wim Böhm, Walid Najjar

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

Abstract

*Due to the large amount of potential parallelism, resource management is a critical issue in multithreaded architectures. The challenge in code generation is to control the parallelism without reducing the machines 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 present and evaluate two methods, slicing and chunking, to control program parallelism. We present the compilation strategy and evaluate its effectiveness in terms of performance characteristics such as run time and matching store occupancy.

Keywords: multithreaded architectures, code generation, quantitative evaluation, control of parallelism.

1 Introduction

Multithreading has been proposed as an execution model for massively parallel processors. Its approach is to hide latency by switching among a set of ready threads and thus improve the processor utilization. Both inter-processor communication and remote data access latencies can be masked.

Another view of multithreading is that it attempts to exploit instruction level locality implicit in von Neumann model as well as the latency tolerance and fast synchronization of dataflow model. Many current multithreading models lie on various points along the von Neumann-dataflow design spectrum. As designs move closer to the von Neumann world, data structure locality can be better exploited. Examples of these designs include the HEP [1], Tera [2], and J-Machine [3]. As designs move closer to dataflow, latencies are better tolerated and parallelism is more easily exploited. Examples are the Monsoon [4], *T [5], and EM-4 [6]. There also exists a software abstraction of multithreading as exemplified by the TAM [7] that can be implemented on traditional multiprocessors such as the CM-5.

In dataflow derived multithreading models, abundant parallelism could overwhelm the machine resources and reduce the exploitation of locality. In this paper, we present two techniques, *slicing* and *chunking*, that attempt to address the above shortcomings. *Slicing* mainly attacks the problem of managing the resources, whereas *chunking*, like vectorization, mainly attempts to exploit locality. Simulation results that compare the effectiveness of these techniques against code with unrestrained parallelism are also presented in this paper. The results indicate that the chunking method helps reduce execution time and also show 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.

In Section 2 we describe our execution model including a basic processor model and briefly summarize our threaded code generation. Section 3 describes the implementation of the two techniques. Section 4 describes the machine independent code characteristics including the average number of instructions executed per thread. Section 5 describes the run time measurements using either or both techniques and compare them against the regular code. Related work is discussed in Section 6. Concluding remarks are given in Section 7.

2 Execution Model and Thread Generation

The multithreaded execution model used in this study is based on dynamic dataflow scheduling where each actor, or a node in dataflow graphs, 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 in threads must have a fixed execution time. 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

*This work is supported in part by NSF Grant MIP-9113268

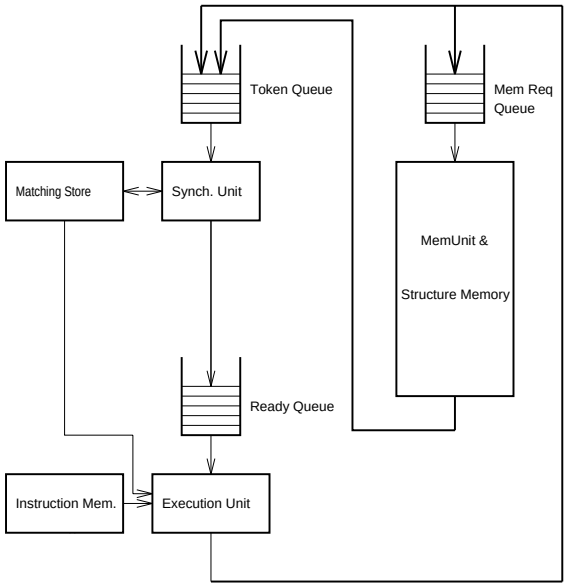

Figure 1: Abstract Model of a Processing Node.

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

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 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.

2.1 Code Generation

Programs are represented in a form of dataflow graphs called MIDC [8]. Each node of the graph represents a thread of straight line von Neumann type 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 code generation is guided by the following objectives: minimize synchronization overhead, maximize intra-thread locality, assure non-blocking (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 [9] that blind efforts to increase the thread size, even when they satisfy the non-blocking and parallelism objectives, can result in a decrease in overall performance. Larger threads tend to have larger number of inputs which can result in a larger input latency[1].

Our nonblocking threads are generated from Sisal programs. Sisal [11] 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 [12]. 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, 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 [13], allows operations that explicitly allocate and manipulate memory in a machine independent manner through the use of buffers. A buffer comprises 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.

The top down cluster generation [14] process then transforms IF2 into MIDC. This phase breaks up the complex IF2 graphs so that threads can be generated. Initial values for reduction operators are generated in the appropriate threads. Threads terminate at control graph interfaces for loops and conditionals, and at nodes for which the execution time is not statically determinable, in order to satisfy the deterministic execution time objective. For instance, a function call does not complete in fixed time, neither does a memory access. Terminal nodes are identified and the IF2 graphs are partitioned along this seam. Overall, threads do not cross function or loop boundaries and therefore useful forms of parallelism is preserved. Although it is not strictly necessary to have threads bounded by branches at this stage, doing so, however, provides more flexibility in the later stage.

The generated MIDC code is further optimized via a bottom-up stage [15] at both the intra-thread and inter-thread levels. Intra-thread 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

[1] Input latency, in this paper, 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 start executing [10]

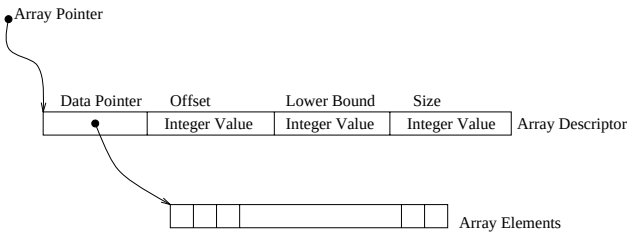

Figure 2: Layout of Arrays in MIDC.

versions of the above optimizations as well as redundant edge eliminations and merge operations that attempt to create larger threads by combining neighboring ones. The merging of threads also takes place across the branch instructions. Benchmark codes used in our experiments typically have thread sizes ranging from 10 to 30 MIDC instructions.

3 Control of Parallelism

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

As most parallel loops deal with streams or 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 is required in order to fetch a single array element.

3.1 Loop Chunking

Chunking is a method by which vector execution can be simulated on a non-vector processor. A fixed amount of work is provided to a variable number of worker processes. In order to generate chunked code, the innermost parallel loops have to be identified. For the loop to be vectorizable, or chunkable, consecutive loop bodies must refer to consecutive array elements. Other types of array references and constructs causing unknown latencies, such as function calls, cause the loop to be identified as non-chunkable. The presence of a dual level memory access to fetch an array element would cause a problem in chunking loops, as the memory access pattern would be unknown at compile time. Fortunately, when reading an array element, reading the array descriptor in the case of a vectorizable/chunkable loop is a common subexpression. Common subexpression elimination moves it out of the loop allowing the compiler to analyze the access pattern. Chunking is used when we require the work done

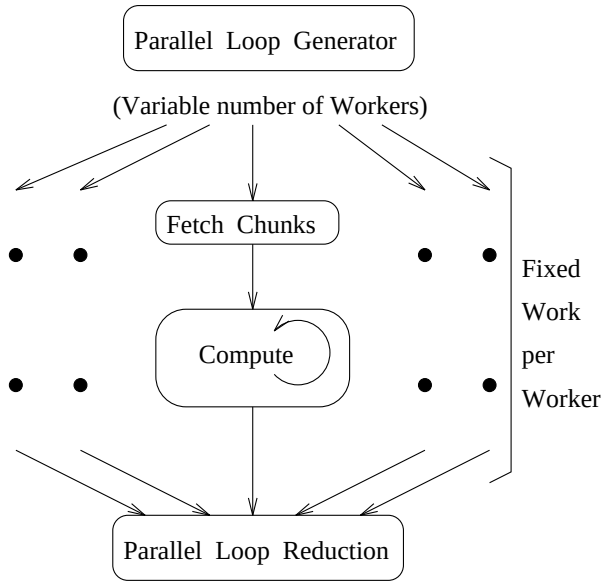

Figure 3: Chunk Control in loops.

by a worker to be fixed and we do not mind having a variable number of workers.

The chunk size is fixed at the compile and is limited by the underlying memory architecture (e.g. the size of a cache line); hence, the loop is strip-mined to allow for variable loop sizes. The chunk generation and control schema are shown in Figure 3.

At run-time the number of workers required have to be computed. For a loop of iteration space `n` and a machine chunk size of `c`, the number of workers, `w`, are computed as: $\lfloor \frac{n}{c} \rfloor + 1$ if $n \ mod \ c \neq 0$ or $\frac{n}{c}$ otherwise. The code dealing with the irregularly sized chunk would be executed if and 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. The MemUnit sends the chunk to the target thread if and only if all the chunk elements are present, i.e., the presence bits for the entire chunk is set. Since SISAL is a strict language, all the data elements of an array will be written sooner or later with no hole left in the array. Once the chunk is received and stored into the target processor's local memory, the handle to the memory is passed onto the thread that consumes this data chunk.

While the thread executes, the `LOAD` instruction is used to move the chunk element from local memory into a register for the computation to proceed. A simple sequential loop structure is used for the computation of successive elements.

The code necessary to compute the number of workers, the size of the odd chunk and the code to spawn off the odd chunk gives rise to overheads in spawning chunked loops. In addition, each chunk element has to be loaded into a register before computation can proceed, although the load latency can be masked via software pipelining.

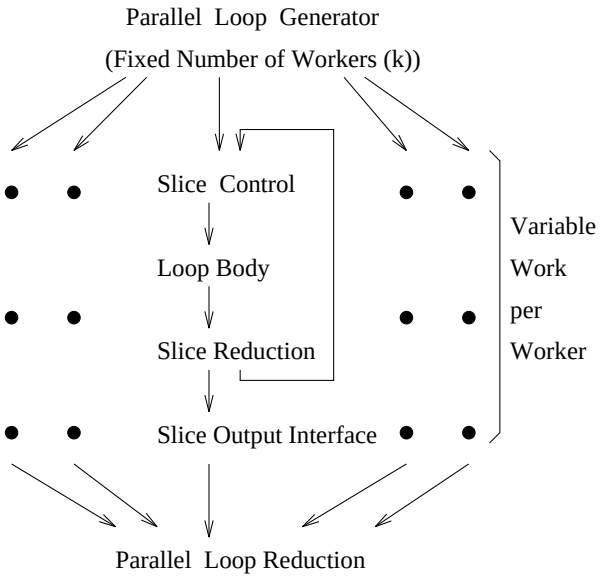

Figure 4: Slice Control in loops.

3.2 Loop Slicing

Slicing is a method by which a parallel loop is split up among a fixed number of worker processors. This enables the control of parallelism. Slicing allows for multiple serial latencies in the loop body. The iteration space is split between a fixed number of workers. All workers will perform at least $\lfloor \frac{n}{k} \rfloor$ work (n is the size of the iteration space, k is the number of workers). $n \ mod \ k$ workers will perform an additional iteration of work. Slice control is shown in Figure 4. It should be noted that any parallel loop can be sliced. Due to the semantics of SISAL there is no potential deadlock in the sliced code.

Each portion of the iteration space assigned to workers are 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[2] required in the parent loop is also divided over the iteration spaces, reducing the amount of serial reduction required, i.e., the bottleneck caused by the strict semantics of SISAL is widened to some extent.

Three additional threads are introduced to control the execution of the run time slices as shown in Figure 4. These are the `Slice Control`, the `Slice Reduction` and the `Slice Output Interface` threads. The `Slice Control` is provided with the loop count and the iteration space. A simple sequential counter is used to determine whether further work is required to be performed. The `Slice Reduction` takes care of all synchronization and partial reduction of results. The `Slice Output Interface` is used to color the outputs of each slice to that of the parent, and furnishes partial results to complete the final reduction of results.

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. Essentially, by slicing, the demand for system resources are "smoothed out" over time; for example, a sudden high demand caused by a parallel loop generation is spread out over time. Therefore, the peak load on the synchronization unit is also reduced. The overhead of loop slicing is mainly due to the `Slice Control` thread that control the iterations. Both `Slice Reduction` and `Slice Output Interface` threads do the work (i.e. reduction and recoloring) that need to done anyway in the non-sliced execution and represent only slightly higher overhead.

[2]Reduction operators in SISAL(`sum, product, max` and `min`) are commutative and associative. Thus, they can be reduced in any order.

Program	Problem Size [Time Steps]	Parallel Loops	Vectorizable Loops
AMR	4 x 80 x 40 [4]	87	34
FFT	2^{15}	14	11
HILBERT	100 x 100	65	30
SDD	2^6	80	23
SIMPLE	100 x 100 [5]	78	42
WEATHER	840km [5]	81	29

Table 1: Program Characteristics of the Benchmarks.

4 Code Characteristics

In this section, we evaluate the dynamic properties of our code before and after applying various combinations of slicing and chunking. We have obtained these results by running codes on a multithreaded machine simulator.

4.1 Benchmarks and Machine Model

For all our experiments, we used a set of nontrivial benchmarks. *FFT* is a one dimensional complex Fast Fourier Transform code and has 528 lines of source code. *HILBERT* computes the condition number for Hilbert matrices. It uses Linpack routines. Hilbert has 564 lines of source code. *BMK11A* is particle transport code developed to evaluate Cray Computer systems at Los Alamos National Laboratory and is composed of 1003 lines of source code. *SDD* solves an elliptic partial differential equation using the Symmetric Domain Decomposition method, 1006 lines of source code. *SIMPLE* is a Lagrangian 2-D hydrodynamics code that simulates the behavior of fluid in a sphere, 1527 of source code. *AMR* is an unsplit integrator taken from an adaptive mesh refinement code at Lawrence Livermore National Laboratory, 1937 lines of source code. *WEATHER* is a one level barotropic weather prediction code and was developed at the Royal Melbourne Institute of Technology. 840km grid size is used with 5 time steps, 2711 lines of source code. The characteristics of the benchmark programs in terms of the number of parallel loops and vectorizable or chunkable loops are given in Table 1. We note that the SDD has the lowest percentage of vectorizable/chunkable parallel loops of around 29% and FFT has the highest percentage with 79%.

The following machine configuration is simulated to derive time-related figures: the machine has ten nodes and each node uses a 4-way issue super-scalar CPU with the instruction latencies of the Motorola 88110 and has an output network bandwidth of two tokens per cycle. The synchronization latencies are those 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. The number of nodes and problem sizes are chosen to give reasonable simulation times and realistic processor utilizations.

Bench	Time	U_P	U_S	Ω_{avg}	Ω_{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

Table 2: Performance with unconstrained parallelism (R).

Bench	Chunk Size	U_P%	U_S%	Speedup%
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 3: Performance with inner loops chunked (C).

All inter-node 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). Also, the size of matching store is unlimited, and therefore can handle any amount of parallelism.

4.2 Measurements

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

- *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.
- Ω_{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.

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

5 Performance Evaluation

In this section, we evaluate the performance of the various parallelism control models against the unconstrained model. The baseline performance given by the unconstrained model is given in Table 2. The table shows that the processor utilizations range from a low of 15.3% for HILBERT and 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 significant fraction of the time is spent in the serial reductions of those parallel loops. Also, in general, the synchonization unit utilization is in the same order, albeit smaller, as the processor utilization.

5.1 Chunked Code.

Chunking exploits data locality and hence should decrease the execution time in addition to restraining parallelism. The speedup of the various benchmarks is shown 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 speedup of 0.3%. *FFT* showed the best speedup of 34.7%. The vectorizable/chunkable loops in HILBERT have much smaller loop bodies than the nonvectorizable loops and therefore the impact of vectorization/chunking is minimal. The data access pattern of FFT is very regular and hence highly vectorizable/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.

5.2 Sliced Codes.

Slicing is used to control matching store memory occupancy. All parallel loops, including vectorizable/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 matching 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 are chosen that favor execution time over the space, 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 is approximately equal to that of the unconstrained case. Good processor utilization implies that the parallelism has not been throttled to the extent where the processor is sitting idle when it should not. The speedup ranges from -5.4% in SDD to -18.1% in HILBERT. The average space used drops dramatically for the experiment. The best saving, in the average case, is shown by SIMPLE which utilizes just 5.8% of that 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 con-

Bench	Slice Size	$U_P\%$	$U_S\%$	Speedup%	$R(\Omega_{avg})\%$	$R(\Omega_{max})\%$
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

Table 4: Performance with loops sliced, best execution times.

Bench	Slice Size	$U_P\%$	$U_S\%$	Speedup%	$R(\Omega_{avg})\%$	$R(\Omega_{max})\%$
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 5: Performance with loops sliced, best occupancy.

sidered, 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 speedups 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.

5.3 Sliced and Chunked Codes.

Since the best speedups and the best space savings have been extracted, the next logical step is to combine the two and try for good execution times with low space utilization.

Table 6 shows the combination of the chunking size from Table 3 and the slice parameters from Table 4 that favors the execution time over the space saving. For the problem sizes used in this paper, the space saving seems to give a good trade off with respect to the time taken to solve the problem. In the case of WEATHER the space saving is of the order of 13% over the unbounded case for a time slow down of 0.6%.

Table 7 shows the combination of the chunking size from Table 3 and the slice parameters from Table 5, that favors the space saving over the execution time. This table shows greater savings in the matching store space utilization. However, as expected, the time taken to solve the problem is greater. One extreme case is WEATHER where space usage is about half of that shown in Table 6, but the execution took almost 20% longer. In the case of FFT, the speedup is 27.8% while only using 5.3% of the space used by the uncostrained case.

Overall, when chunking is used along with slicing there is an improvement in the space utilization, and in some cases, an improvement in execution time as well. If chunking is effective, then the combination of chunking and slicing offers the best of the both world. This is evidenced for AMR and FFT. The choice of throttling parameters depends on the system size. However, an important point to be made is that the upper bound on the space utilized is determined by the base slicing scheme used and should no longer be dependent on the problem size.

5.4 Determining Throttle Values

In this section, 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 to be 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% speedup. Processor utilization remains fairly uniformly high in all the cases. The synchronization unit utilization drops by half, representing a significant reduction.

In the third experiment, the best slicing levels are determined. The first runs only sliced 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

Bench	Chunk Size	Slice Size	Speedup%	$R(\Omega_{max})$ %
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

Table 6: Performance with inner loops chunked and loops sliced, best processor utilization.

Bench	Chunk Size	Slice Size	Speedup%	$R(\Omega_{max})$%
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 7: Performance with inner loops chunked and loops sliced, best occupancy.

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. The choice may not always be this clear cut.

6 Related Work

Parallelism control was first proposed in fine-grain dataflow architectures due to the large amounts of parallelism that was being generated. The parallelism generated was 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 high degree of 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 [16, 17] and *K-bounding* [18]. In addition Egan et al [19] have proposed methods of slicing the iteration space. Teo and Böhm [20], have proposed a method of chunking on fine grain machines by the use of iterative instructions.

Throttling is a pure 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 have to 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. Throttling required preservation of state in the MIDC model or makes threads smaller due to the additional latency in creating tasks.

K-bounding is another method proposed to control fine-grain parallelism in loops. The compiler analyses the code and determines the maximum resource usage for a loop cycle. At run-time the hardware 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 at 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 has to be performed to reestablish the order of the resultscoming from the loop body to that of the context enclosing the loop invocation.

The chunking proposed by Teo and Böhm is for fine grained machines by the utilization of iterative instructions. Iterative instructions are used to reduce the number of tokens that are generated in a dataflow machine. The iterative instructions is used to generate a set number of tokens with incremental indicies in the tag. When the fixed number of tokens are generated, the inputs to the iterative iteration are recycled back to itself. Due to the load on the matching store, there would be a finite time between the two invocations of the set of iterative instructions. In the case of small loops this time would be sufficient to complete executing the last n loop bodies before a new batch of tokens/loop bodies are generated. However, for larger loops, these instructions provide a delay and thus a reduction in the load of the matching store in effect a throttle.

Slicing was chosen over K-bounding as the method of parallelism control for the following reasons.

- *Increased Token traffic:* As the model of computation uses non blocking threads, token traffic is increased due to the recirculation of the various input values from the generator back to itself even when there are no more iterations to execute, this will happen at least k times.
- *Increased Hot Spot activity:* In the case of k-bounded loops, there would exist two serial hot spots, the first in the generator thread and the second in the reduction and synchronization step. It is not possible to merge these two threads as there is a phase difference of k between the execution of the generator and the reduction steps. The structure can be viewed in terms of *master*

Exp.	Chunk	Slice	Speedup%	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

Table 8: FFT: Arriving at the Best Throttle Value.

and `slave` processes. The thread that generates the various instances of the loop bodies is the master process and the loop bodies themselves are slave processes. In the case of loop slicing, the generator(master process) would create a number of sub-master processes which would control the different iteration spaces in parallel and alleviate the hot spot in the generation of tasks. Partial reductions can be done in parallel using a similar mechanism. Thus, the generation and reduction bottleneck is widened to some extent. It is our claim that the bottleneck is removed completely in the slice generator and widened enough at the final reduction stage removing the hotspot.

- *Reuse of resources:* The MIDC model of execution does not allow for the easy reuse of resources as the threads are non-blocking. The best way of implementing k-bounded loops is via the use of circular buffers. Its implementation in the MIDC model is difficult at best.

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 reduce the load on the synchronization unit, but 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 (U_P), as well as the synchronization unit utilization (U_S). Lower utilization rates with respect to the unbounded case implies too small a throttle for the problem in question. The best balanced system will not compromise utilization for space efficiency.

References

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

[2] 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.

[3] 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.

[4] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Int. Symp. on Computer Architecture*, pages 82–91, June 1990.

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

[6] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a data-flow single chip processor. In *Int. Symp. on Computer Architecture*, pages 46–53, May 1989.

[7] 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 *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[8] A. P. W. Böhm, W. A. Najjar, B. Shankar, and L. Roh. An evaluation of bottom-up and top-down thread generation techniques. In *Int. Symp. on Microarchitecture (MICRO-26)*, 1993.

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

[10] W. A. Najjar, W. M. Miller, and A. P. W. Böhm. An Analysis of Loop Latency in Dataflow Execution. In *Int. Symp. on Computer Architecture*, pages 352–361, Gold Coast, Australia, 1992.

[11] 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.

[12] S. K. Skedzielewski and John Glauert. IF1: An intermediate form for applicative languages reference manual, version 1. 0. Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.

[13] 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, University of California - Lawrence Livermore Laboratory, December 1986.

[14] B. Shankar, A. P. W. Böhm, and W. A. Najjar. Top-Down thread generation for SISAL. In *Sisal '93*, 1993.

[15] L. Roh, W. A. Najjar, B. Shankar, and A. P. W. Böhm. An Evaluation of Optimized Threaded Code Generation. In *Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'94)*, pages 37–46, Montreal, Canada, 1994. North-Holland.

[16] C. A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Computer. In *Lecture Notes in Computer Science no. 274*, pages 1–15, 1987.

[17] David F. Snelling. *The Design and Analysis of a Stateless Data-Flow Architecture*. PhD thesis, Computer Science Department, University of Manchester, Manchester, UK., 1993.

[18] D.E. Culler. Resource management for the tagged token data flow architecture. Technical Report TR-332, Laboratory for Computer Science, MIT, January 1985.

[19] G. K. Egan, N. J. Webb, and A. P. W. Böhm. Some Architectural Features of the CSIRAC II Data-Flow Computer. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-flow Computing*, pages 143–174. Prentice Hall, 1991.

[20] Y. Teo and A. P. W. Böhm. Resource Managament and Iterative Instructions. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-flow Computing*, pages 481–500. Prentice Hall, 1991.