Performance Tuning Scientific Codes for Dataflow Execution

Andrew Shaw, Arvind, R. Paul Johnson
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
{shaw|arvind|rpaul}@lcs.mit.edu

Abstract

Performance tuning programs for dataflow execution involves tradeoffs and optimizations which may be significantly different than for execution on conventional machines. We examine some tuning techniques for scientific programs with regular control but irregular geometry. We use as an example the core of an ocean modeling code developed in the implicitly parallel language Id for the Monsoon dataflow machine. Dataflow implementations of loops, in particular, require careful examination by the compiler or programmer to attain high performance because of overheads due to fine-grained synchronization, control, and operand distribution.

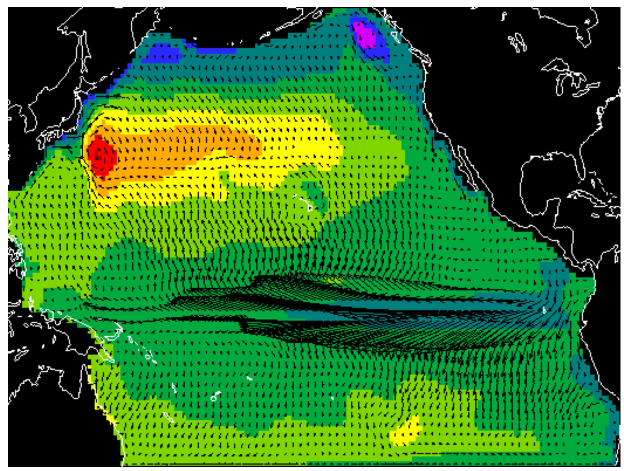

Figure 1. In an ocean modeling code, the ocean is represented by multiple three-dimensional arrays holding values for different physical characteristics of the ocean. Typically, a large percentage of the geometry represents land, which is not of interest.

1. Introduction

Dataflow machines have two basic types of overheads which do not exist in sequential machines: *synchronization* and *control* overheads. Synchronization overheads are primarily encountered within basic blocks, and control overheads are encountered between basic blocks (on conditional, loop, and procedure call boundaries). These overheads are the cost of exploiting fine-grained parallelism. In this paper we consider tuning techniques to alleviate these overheads for regular, *scientific* applications on a dataflow computer. These tuning techniques apply to both compiler-directed optimizations as well as programmer-directed optimizations.

Many scientific applications spend much of their time in loops traversing over large arrays which represent physical characteristics distributed over the geometry of the problem. One such application is the MIT ocean circulation model GCM [11] [2] whose data parallel implementation on the CM-5 has shown very good performance. While developing GCM for the CM-5, we also wrote a version of it in Id [13] for the Monsoon dataflow machine [14] [7]. Id is an implicitly parallel language with a multithreaded implementation model, and the GCM code allowed us to compare the execution overheads for the multithreaded and data parallel models on an application which was extremely well-suited for data parallel execution. A more in-depth study of the data parallel and multithreaded implementations of GCM is presented in [2], where we compare program styles, ease of programming, and algorithmic issues, while in this paper, we primarily discuss performance and tuning on the dataflow platform.

Figure 1 shows a typical problem geometry – the Pacific ocean basin – used in the circulation model. A Pacific ocean state array would typically contain $168 \times 90 \times 10$ elements, 40% of which represent land, and do not contribute to the simulation. However, unlike some physical models,

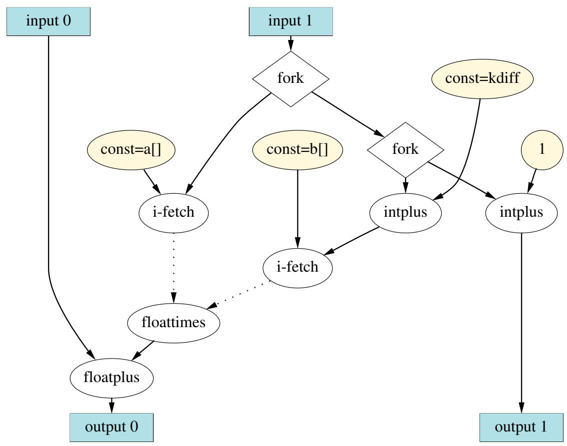

Figure 2. Optimized Monsoon dataflow graph for the body of the innermost loop of a three-dimensional inner product code. It directly represents instruction-level parallelism, but incurs fine-grained synchronization costs.

geometries used in ocean modeling are usually not irregular enough to warrant repartitioning to eliminate land elements.

The core of GCM code is a preconditioned conjugate gradient (PCG) solver which takes 80% or more of the computational time of the application. The PCG consists of simple operations over entire state arrays which can be described as triply-nested loops with relatively small loop bodies. In order to attain high performance for this application, we must execute these loops efficiently while also distributing the work among the processors. Dataflow architectures differ significantly from conventional sequential microprocessors or vector processors, so the process of tuning these loops to obtain high performance is also quite different than tuning loops written in conventional languages for conventional architectures.

1.1. Id: an implicitly parallel language

Id is an implicitly parallel language which can express parallelism at the instruction-, procedure-, and loop-level. Id programs are compiled into *dataflow graphs*, which are optimized, and then executed directly by the Monsoon architecture.

Instruction-level parallelism. Dataflow graphs such as in Figure 2 directly represent instruction-level parallelism – in the Monsoon architecture, such instruction-level parallelism is only exploited within a processor. Basic blocks are represented by acyclic graphs such as in Figure 2. The *fork* operator in the figure is necessary for sending values to more than two instruction at a time – because of opcode encoding limitations, most Monsoon instructions can send a value to two destination instructions, and some instructions can send a value to only one instruction.

Memory operations such as *i-fetch* are split-phase – the initiation and response are separate, and the latency of the response may be very long. Each memory operation has an implicit synchronization associated with it so that reads which are executed before the write to the same memory location wait for the write to occur before responding.

Compile-time constants (i.e. literals) and run-time constants (such as loop constants) are represented in Figure 2 in light gray. Run-time constants must be stored into local memory at run-time using the *constant-store* instruction, and literals are stored into local memory at load-time by the loader. Both types of constants can be accessed without synchronization overhead.

Control is implemented by using the *switch* operator which steers an input value to one of two locations depending upon the value of an input predicate. Switch operators can be used to build conditionals or loops, but incur high overhead because a different switch operator is needed for each loop or conditional variable.

Procedure-level parallelism. Procedure-level parallelism is expressed by implicitly forking every procedure call and joining on the return value of the procedure. Each procedure requires an *activation frame*, which is used to store intermediate results and as a synchronization namespace for instructions in the procedure. An activation frame is roughly analagous to a stack frame in a sequential processor.

Each procedure call requires a frame allocation and deallocation. A frame may be allocated on the local processor or on a remote processor to expoit inter-processor parallelism, and frame allocation and deallocation is handled by the run-time system. Care has been taken to eliminate run-time system costs, to the point where much of the run-time system is written in assembly, and supported by special "microcoded" machine instructions [5]. In the default run-time system, frame memory on each processor is divided equally among all processors, so that each processor manages part of the frame space of every other processor. Frame allocation requests are then handled locally in a round-robin fashion across all of the processors in an attempt to provide some form of load balancing.

Despite the care taken in designing the run-time system, frame management in this parallel environment is still more expensive than in a sequential environment, where a simple pointer-bump can allocate or deallocate a stack frame. Furthermore, every argument is sent as a separate message, and

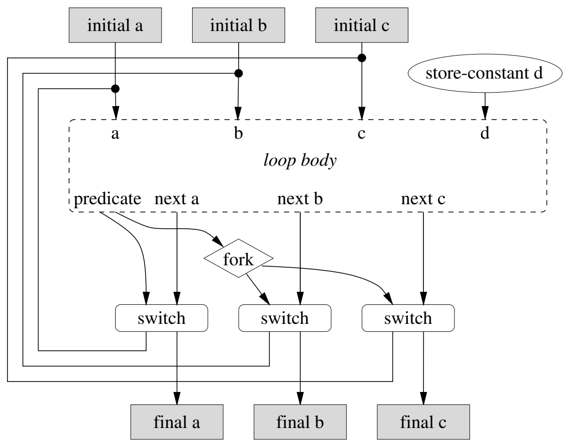

Figure 3. Schema for "sequential" dataflow loops, which incur high fine-grained synchronization costs because of the need for switch and fork operators.

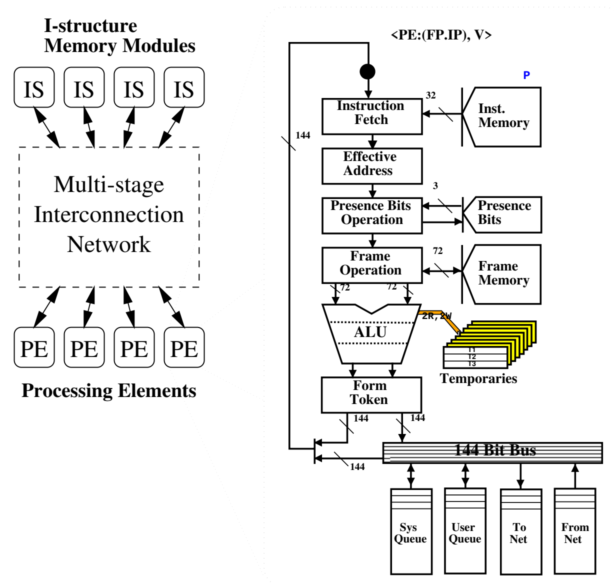

Figure 4. Monsoon consists of processing elements (PE's) connected to I-structure memory modules (IS's) through a multi-stage network. Each PE is an eight-stage pipelined dataflow processor.

although the composition, sending and receiving of these messages are well supported in hardware, argument-passing is still relatively more expensive than register-style argument passing in sequential processors. In general, the programmer should inline most non-recursive procedures to avoid these twin overheads of procedure calls.

Loop-level parallelism. Loop-level parallelism can be exploited in several ways. A direct tail-recursive implementation of loops will use too many frame resources, so most loops are either "sequential" loops or k-bounded loops [6]. Sequential loops execute on one processor, and only exploit instruction-level parallelism; k-bounded loops execute k consecutive iterations worth of parallelism either within or across processors.

Figure 3 shows the schema for a sequential loop for Monsoon. A number of input variables and loop constants are initially fed to the loop body. Their values for the next iteration are either fed back into the top of the loop body, or else output to the final values. The values are steered to their appropriate destination through the *switch* dataflow instruction. Since the switch instruction can only steer one value to one destination at a time, an additional fork may be required to distribute the loop test predicate value for each additional loop variable. Loop constants are stored into the frame before loop execution and extracted after loop completion.

The k-bounded loop schema is more complicated than the sequential schema, and employs a ring of k activation frames which are allocated and linked at the beginning of the loop and deallocated at the end of the loop. Loop constants must be stored into and extracted from each frame in a k-bounded loop. Induction variables are passed from iteration to iteration like arguments are passed for procedure calls.

Standard loop optimizations such as unrolling, peeling, strip-mining, and interchange can have a larger impact on performance than for conventional architectures because of the overhead of parallel asynchronous execution of loop iterations in the dataflow model.

High loop overhead for dataflow execution is exacerbated in Id because array extents in Id, unlike Fortran, can be computed at run-time. Consequently, to perform many loop optimizations important for performance on scientific codes, one would need pragmas in Id to indicate static array extents. However, this type of overhead is due to the language, and not dataflow execution.

1.2. The Monsoon dataflow machine

Figure 4 (taken from [9]) shows a high-level view of the Monsoon multiprocessor. Monsoon [14] consists of eight 64-bit processing elements (PE's) and eight I-structure memory modules (or IS's) connected via a multistage packet switch network. Each PE is an eight-stage pipelined processor. On each processor cycle, a token enters the pipeline and its result is available eight cycles later. Keeping an eight pro-

```
def ip_3d a b =
  { s = 0.0 ;
   in
     {for i <- 1 to imax BOUND ib do
        s_jk = 0.0;
        next s = s +
         {for j <- 1 to jmax BOUND jb do
            s_k = 0.0;
            next s_jk = s_jk +
             {for k <- 1 to kmax do
                next s_k =
                 s_k + a[i,j,k]*b[i,j,k];
                finally s_k};
            finally s_jk};
      finally s} };
```

Figure 5. 3D inner product written in Id, outer two loops k-bounded, inner loop sequential. `imax`**,** `jmax`**, and** `kmax` **are compile-time constants.**

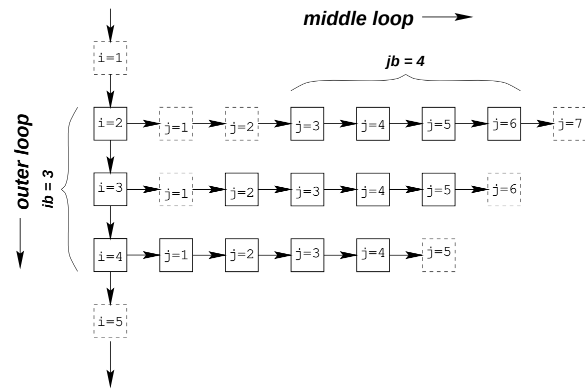

Figure 6. Pattern of frame usage for nested k-bounded loops for the 3D inner product.

cessor Monsoon busy requires at least 64-fold parallelism because each stage of the pipeline executes a different thread. Each PE can process ten million tokens per second and has 256K 64-bit words of local memory on which frames are allocated. In the memory module where heap objects reside, word-level presence bits are implemented in hardware to support fine-grain synchronizing data structures for global heap. Each IS has 4M 64-bit words of heap memory. Monsoon's network interface with bandwidth of 100M bytes per second can deliver a token to the network every cycle.

Each instruction executes in one or two cycles, including floating point, integer, memory, control, and message instructions. Any instruction which has two non-constant inputs must synchronize on the arrival of its two inputs, and requires two cycles to execute, one of which is counted as a *bubble*.

Every global memory access goes over the network, and all locations in the global memory are equidistant. The mapping of work to processors is governed by the frame allocation policy. Heap objects are interleaved by hardware across the IS's and in general, data mapping has no effect on performance. Monsoon accesses global memory in a split-phase manner, and can do useful work during the long-latency of a memory access. However, this requires additional instruction-level parallelism to keep the processor fully utilized. Each Monsoon processor has 16 counters which can keep accurate counts of various types of instructions executed.

2. Performance tuning on Monsoon

The performance of Id on Monsoon is largely determined by two factors:

- The *total number of instructions* executed, which determines the amount of computation and communication necessary.
- The *CPU utilization*, which is affected by the loop bounds and the frame allocation policy. Idle cycles are caused by lack of parallelism, load imbalance, and some hardware "hazards".

To the first order, the number of instructions executed depends only on the program and its pragmas, and not the machine configuration – i.e. the same program will execute about the same total number of instructions for a 1- or 8-processor Monsoon. CPU utilization does depend on the machine configuration and attaining high CPU utilization requires setting the loop bounds and in some cases, source code modification.

Id provides pragmas for inlining functions, unrolling loops, or declaring loops to be sequential or k-bounded, all of which can have a dramatic impact on the instruction count. Id users almost always inline (`defsubst`) small, non-recursive functions. In the following Sections, we describe how these other pragmas and program modifications can be used to tune PCG.

We will illustrate how to tune nested loops for performance using the three-dimensional inner product function `ip_3d` shown in Figure 5. The outer and middle loops are annotated to be k-bounded, and k-bounds are `ib` and `jb`, respectively. Figure 6 shows the pattern of frame usage for the k-bounded loops for this code when `ib` is 3 and `jb` is 4.

2.1. Choosing loop schemas

Consider three different cases for implementing the innermost loop: k-bounded, sequential, and completely unrolled.

Cycles/iteration of `3d_ip` inner loop			
Opcode	completely unrolled	sequential	k-bounded
fork	2	3	6
intplus	2	2	2
i-fetch	2	2	3
floattimes	2 (1)	2 (1)	2 (1)
floatplus	1 (.5)	1 (.5)	2 (1)
switch		4 (2)	1
intle		1	1
message			4 (2)
i-take			1
i-put			2 (1)
sync			3 (1)
Total	9 (1.5)	15 (3.5)	27 (6)

Figure 7. Completely unrolled loops require the minimal cycles per iteration. Sequential and k-bounded implementations of the same loop require more cycles. In parentheses are the number of bubbles incurred by each operation per iteration.

Given the same loop body, these three implementations of the same loop will require a significantly different number of instructions to execute per iteration.

Loop schema cycle counts. A completely unrolled loop is most efficient implementation of a dataflow loop. Complete unrolling is done by inserting the pragma `@unfold completely` inside the innermost loop. This is possible only if the number of iterations a loop executes is fixed and known at compile time.

Figure 7 shows the differences in the cycle counts for one iteration of three implementations of the inner loop of `ip_3d`. Figure 2 is the loop body for the sequential version of the inner loop of `ip_3d`. There are two circulating or induction variables in the loop: the running sum (called "input 0" in Figure 2), which is a floating point value and the index variable (called "input 1" in Figure 2), which is either `k` or a linear offset from `k`. The i-fetch instruction is not associative with addition, so the base of the array and an integer offset are necessary to perform a memory reference.

The completely unrolled version uses the minimum number of instructions to implement one iteration of the inner loop – the two fork instructions are used to distribute the index `k` to each intplus, which feeds the i-fetch instructions. The intplus instructions cannot be eliminated because the i-fetch is not associative with addition. The values returned from the i-fetch are multiplied together, also incurring a bubble. The result of the multiplication is added to the running sum, and only half of the cost of the addition is incurred by each additional iteration, including half of the bubble. There is no loop test in the completely unrolled version.

The sequential implementation of the loop uses two switches to steer the two induction variables around the loop, and each switch incurs a bubble. An additional loop test (*intle*) is also performed, and an additional fork is necessary to feed the loop test.

The k-bounded implementation of the loop uses two message instructions to send the induction variables to the next iteration and associated frame. The i-take and i-put instructions are used as semaphores to synchronize between loop iterations to assure when a frame may be reused for a new iteration. The sync instructions gather signals from work being performed in the current iteration to insure that all work is completed in the current iteration. Additional fork instructions are used to distribute values to the new instructions.

Some traditional loop optimizations for conventional processors introduce new induction variables. For dataflow loops, such optimizations may actually decrease performance even if fewer instructions are executed in the loop body, because of additional switch and fork overheads, or from additional overheads from message-passing for k-bounded loops.

Loop startup and shutdown costs. In addition to the per iteration costs, loops have startup and shutdown costs which may be significant, depending upon the loop schema. Completely unrolled loops have the minimum amount of startup cost – the storing of loop constants into the frame will be lifted to the next outer loop level or else transformed into forks. For sequential loops, constants must be stored into the frame before the loop executes, and then extracted from the frame after they are executed. k-bounded loops incur the highest startup and shutdown costs. At startup for a k-bounded loop, k frames are allocated and then pointers are set in each frame to point to the previous and next frame. Loop constants are stored into each frame, and at shutdown, the loop constants must be extracted and the frames deallocated. Allocating and deallocating a frame takes about 60-70 cycles. The startup/shutdown cost of a k-bounded loop can be from about 100 to several hundred cycles per k, depending on the number of loop constants.

Nesting of loops can exacerbate the number of loop constants which must be stored and extracted. A loop constant in an inner loop becomes a loop constant for every loop nesting out to the loop which may change the value of the loop constant. Therefore, that constant must be stored and extracted even though its value may not change between stores and extractions if the middle loop does not change the value of the loop constant.

Loop startup overhead can have a significant effect on performance when the number of loop iterations is few. In those cases, complete unrolling should be considered when possible, even if slight algorithmic restructuring is necessary.

2.2. Loop striding

In vector and cache-based machines, the loop stride over array elements can have a significant effect on performance because of memory interleaving, cache associativity or line size. For slightly different reasons, loop stride makes a difference in Id also. For example, in Id there is a significant advantage in having an inner loop which has a constant stride known at compile-time. Because array extents are not known at compile time, the innermost loop should iterate over the stride-1 dimension of the array so that the compiler knows the stride of the loop at compile time. If the compiler doesn't know the stride at compile-time, the stride becomes a loop constant, which is more expensive than a literal.

For the completely unrolled case of `ip_3d`, if the array is not aligned with the loop along the stride-1 dimension, an additional 6 cycles per iteration are necessary because of two additional multiplications and two additional loop constant-store instructions to save the loop constants into the frame. If we didn't know the array extents at compile-time, but did know that the arrays were of the same extents, then only one additional multiplication and constant-store would be necessary per iteration.

Loop interchange is the standard optimization to achieve loop/array stride alignment [20]. This optimization is not implemented in the Id compiler: for this study, we performed this optimization by hand by making certain that the innermost loop iterated over the 1-stride dimension of the arrays.

2.3. Choosing k for k-bounded loops

In a triply nested loop such as `ip_3d`, suppose that the k-bound for the outermost bounded loop is ib and for the middle bounded loop is jb. The amount of interprocessor parallelism exposed is then on the order of $ib \times jb$. Given a certain machine configuration, we are interested in setting the product $ib \times jb$ such that it will keep the entire machine busy. What are the optimal values of ib and jb, such that the number of instructions executed by the loops are minimized?

If there are $imax$ iterations of the outer loop executed, then the number of frame allocations and initializations (requiring hundreds of cycles apiece) is then: $ib + imax \times jb$. For a set amount of parallelism $ib \times jb$ we wish to exploit in a nested loop, in general it is better to make ib larger and jb smaller to minimize loop initialization costs. A possible optimization is to lift middle loop frame allocations so that it occurs ib times as opposed to $imax$ times. The optimization has been implemented, but has not worked reliably.

Large values of k may not be useful if there are dependencies of any kind (data, control, producer/consumer) between loop iterations, or if the number of iterations is not significantly larger than k. Choosing loop bounds can make a tremendous difference in performance, as shown by Culler [6], but to date, no automatic compiler-directed policy has been implemented which achieves high performance.

2.4. Other loop optimizations

Partial loop unrolling (as opposed to complete unrolling) is effective for loops which execute many iterations – the loop body is increased while the number of iterations is decreased, therefore amortizing loop overheads. However, for loops which may execute a few iterations, unrolling may decrease performance because of additional conditionals which are used to determine if a loop has enough iterations left to execute a unrolled version or a standard version.

Strip mining is a useful optimization for high-overhead k-bounded loops. To strip mine a loop, the iterations of a k-bounded loops are divided in a block or interleaved fashion into pieces of work which are performed by more efficient sequential loops.

Because dataflow execution allows for dynamic instruction-level parallelism, dataflow loops may incur additional synchronization costs to determine termination of an iteration. In some cases, a certain loop iteration must terminate before another iteration is initiated – for sequential loops, the two consecutive iterations may be sharing the same synchronization space (i.e. the frame) or for k-bounded loops, an iteration may not start before the frame k iterations before is complete. Because the parallelism is so fine-grained, additional sync instructions may be inserted to determine when an iteration is complete. Sometimes, the inherent structure of the loop body performs much or all of the synchronization, in which case redundant sync instructions may be removed [1].

Finally, Ang [1] has described a sequential loop schema for Monsoon which is frame-based rather than switch-based, and which performs better for a variety of applications.

3. Preconditioned Conjugate Gradient

The preconditioned conjugate gradient algorithm [4] is the core of many computational fluid dynamics applications and its regular structure is similar to many other scientific applications. It solves linear systems using an indirect (i.e. iterative) method. The preconditioning step of PCG is used to speed convergence, and varies depending upon the characteristics of the linear systems being solved. The algorithm itself is composed of only four basic operations; in the case

```
{while (ip_3d r r) > epsilon do
   gamma   = ip_3d r xi;
   Ab      = seven_pt_stencil b;
   alpha   = gamma / (ip_3d b Ab);
   next p  = daxpy_3d p alpha b;
   next r  = daxpy_3d r (-alpha) Ab;
   next xi = fb_preconditioner (next r);
   beta    = (ip_3d (next r) (next xi)) /
                  gamma;
   next b  = daxpy_3d (next xi) beta b;
  finally p}
```

Figure 8. Inner loop of preconditioned conjugate gradient in Id.

of the GCM ocean modelling code, the operations are implemented as follows:

- **3D inner product** – this is the computation described in the previous Section.
- **3D daxpy** – $aX + Y$, where X and Y are 3D vectors and a is a scalar constant. 3D daxpy requires a floating point add and multiply per element.
- **7-point stencil** – the stencil takes a weighted average of the local element and the six surrounding elements. 7-point stencil requires six floating point additions and seven floating point multiplications per element.
- **forward-backward substitution** – the preconditioner we chose for our PCG is a forward-backward substitution of a tridiagonal linear system performed in each column of the ocean. The forward-backward substitution requires about five floating point operations per element.

Each of these operations can be implemented as a triply-nested loop. The loops are simple, structured, and have no dependencies between iterations. Results from some of these operations feed into others, and because of Monsoon's fine-grained word-level synchronization on memory, operations may proceed before their input arrays are completely defined.

The structure of the state arrays directly reflects the geometry of the ocean being modeled. Certain elements of the arrays represent land state, and certain elements represent water state. We are not concerned with elements on land because they do not change with time. We can structure most of the computation so that all computation is uniform over all elements of the array, or just attempt to do computation on water. In some cases, land may take up to 60-70% of the geometry.

We consider only computation performed on water to be required, and for a given geometry, the number of *required* *floating point* operations is constant for this algorithm. Computations performed on land are extraneous and overhead. In the next Section, we consider optimization by computing only on water elements.

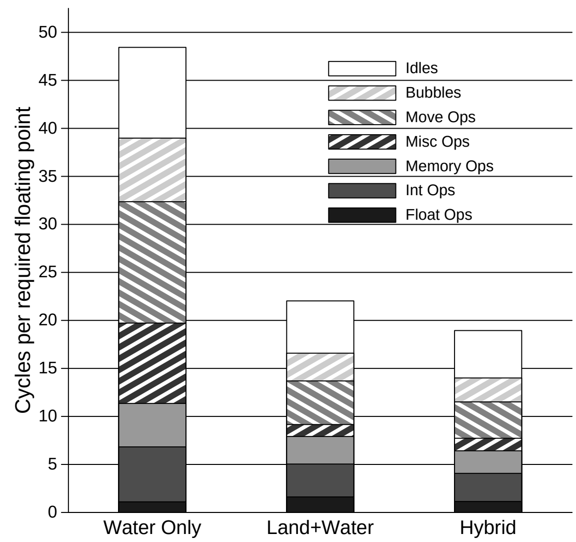

Figure 9. Comparison of the three versions of PCG for the $168 \times 90 \times 4$ Pacific ocean, 60% water. ($ib = 5$, $jb = 3$, 2 processor Monsoon)

3.1. Optimizing out computation on land

By eliminating computation on land, the maximum speedup we could expect over an implementation that computes on both land and water is the ratio of land and water elements over water elements. For most real geometries, this is a maximum speedup of three or four, and for the Pacific ocean geometry, the maximum speedup is less than a factor of two.

We also incur overhead when we attempt to compute only on water, because we must use sequential loops in the innermost loop (which computes over columns in the ocean) as compared to completely unrolled loops in the case where we compute on both land and water. Figure 9 shows that the version which computes on land and water actually is faster than the version which computes only on water, even though fewer floating point operations are performed in the water-only version. For this type of application on this type of architecture, reducing floating point operations is less important than simplifying the structure of the code to reduce loop overheads.

In an attempt to take some advantage of both simplified loop structure and eliminating computation on land elements, we also developed a *hybrid* version of the PCG which

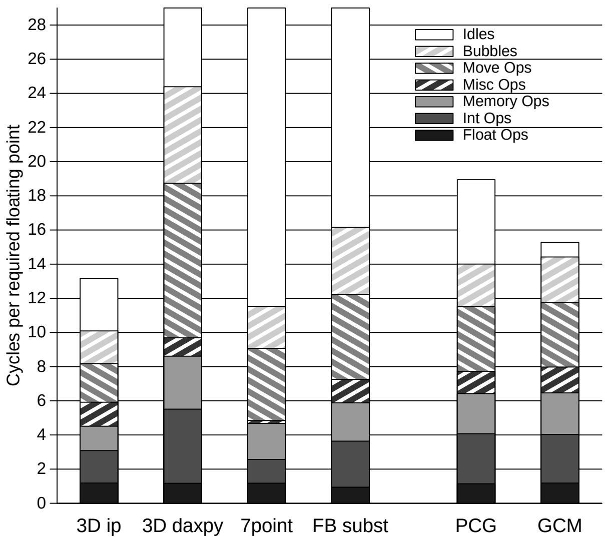

Figure 10. The components of the hybrid PCG, the PCG itself, and the whole GCM ocean modeling code for the $168 \times 90 \times 4$ Pacific ocean. ($ib = 5$, $jb = 3$, 2 processor Monsoon)

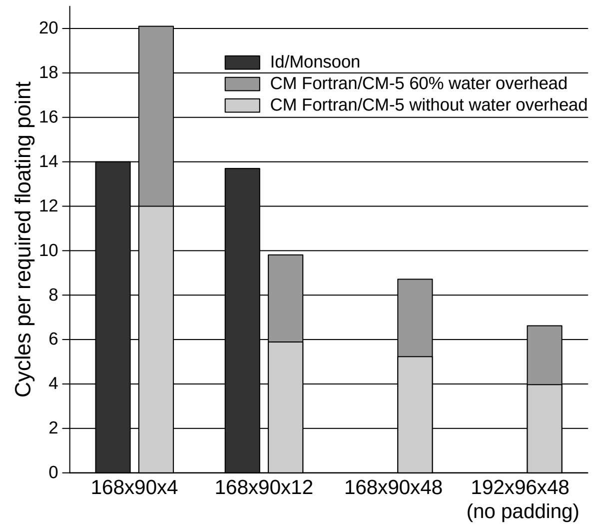

Figure 11. PCG efficiency on the CM-5 versus Monsoon. (Pacific ocean, $ib = 5$, $jb = 3$, 2 processor Monsoon, 32 processor CM-5)

only computes on the bounding hull of the water on the surface of the geometry, while computing on entire columns of water within the bounding hull. In this way, we eliminate much of the computation on land, while still allowing complete unrolling of the innermost loop. Figure 9 shows that this version is slightly faster than the version which computes on both land and water.

The number of idle cycles in the graph of Figure 9 is misleading because PCG is only one part (albeit the most computationally intense part) of the entire GCM code. When the PCG is incorporated into the larger code, parallelism from the rest of the code can mask idle cycles in PCG.

3.2. Breakdown by operation

Figure 10 shows the cycles per floating point operations for the four operations of the hybrid version of PCG, the PCG itself and the overall GCM code. The constituent parts were timed by extracting the code fragment for the component from the elliptic solver and running it in isolation. Note that the idle cycles for some of the components are off the scale – however, when the four components are integrated together into the PCG, the idle cycles are masked by work from other components. Also, when the PCG itself is integrated into the the overall GCM code, the instruction counts heavily reflect the ratios of the PCG, but idle cycles decrease because of other work being performed which masks those idle cycles when the PCG is run in isolation. This masking of idle cycles is one of the primary benefits of the multithreaded execution model.

When we set k-bounds for loops, we must do it in the context of an entire application, because as Figure 10 shows, processor utilization for codes with loop bounds set in isolation do not reflect utilization for integrated codes.

4. Comparison to CM-5 implementation

To put the performance of the PCG on Monsoon into perspective, we compare it to a version of PCG being used in daily production runs of the ocean code. The GCM code has been implemented in CM Fortran [19] for the CM-5 [10]. CM Fortran is a data parallel dialect of Fortran, which is similar to HPF. We use the same measure of efficiency to compare the data parallel implementation to the dataflow implementation: cycles per required floating point operation. Note that the data parallel model can also be implemented efficiently on dataflow machines [17], but our Id implementation uses a multithreaded computational model. For both the Id and CM Fortran implementations, the number of required floating point operations is identical. The cycles we are counting in the CM Fortran case are CM-5 vector unit cycles, because the CM-5 vector units perform almost all of the work in this application and most data parallel scientific applications.

Figure 11 shows the performance of PCG on a variety of ocean geometries. For a shallow, 4-layer Pacific ocean, the

Id version executes fewer overhead instructions per required floating point operation. As we increase the depth (and therefore problem size) of the ocean geometry, the efficiency of the CM Fortran version also increases, due to longer vector lengths. We were unable to run a larger than 12-layer ocean on Monsoon because of heap memory limitations.

Most of the efficiency on the CM Fortran version gained by adding ocean layers is reached at 12 layers – the 48-layer ocean has about the same efficiency as the 8-layer ocean.

Further increases in the floating-point efficiency can be extracted when the geometry is fitted to the machine such that there is no additional array padding to even out array distribution across the processors. If the ocean geometry is $192 \times 96 \times 48$, the data parallel code executes in exactly the same amount of time as the $168 \times 90 \times 48$ version of the code, but executes 22% more floating point operations. [1]

The "raw" vector performance of the CM-5 on PCG (i.e. without the 60% water overhead or array padding) is one floating point operation every four cycles. Thus, each 16MHz vector unit can produce 4 MFlop/s, which is $1/4$ of the peak performance of the CM-5. There are few realistic applications which achieve this level of performance. Additional overheads from padding, computing on land, and short vector lengths add up quickly to degrade performance by a factor of 2 to 5. For most real geometries, we expect that performance of GCM will be closer to 10-20 cycles per required floating point operation than the idealized peak performance of 4 cycles per floating point operation, which is comparable to the performance we achieved on Monsoon.

5 Conclusion

Tuning for dataflow execution (either by the programmer or compiler) requires techniques which are different from tuning on conventional architectures. In this paper, we have characterized some of the overheads of dataflow execution on Monsoon, focusing on loops and using an example of a preconditioned conjugate gradient algorithm (PCG). Our performance measured by *cycles executed per required floating point operation* is comparable to a production version of the algorithm running on the CM-5.

Much of the overhead comes from the dataflow switch and fork instructions, as well as bubbles which result from fine-grained synchronization on Monsoon. Part of this overhead could be eliminated by architectural changes – for instance, the EM-4 [16] architecture uses essentially the same dataflow-style synchronization mechanism as Monsoon, but does not incur a bubble because the synchronization and execution units are decoupled. In addition, due to some hardware limitations, some idle cycles on Monsoon are the result of network access contention, rather than being fundamentally associated with dataflow architectures. The fork overhead is eliminated in the Epsilon dataflow architecture [8] with a *copy* instruction which can send values to multiple instructions – however, the *copy* instruction reduces instruction count, but does not reduce cycle count.

Some overhead is the result of the k-bounded loop scheme, which has proven to be difficult to use for the programmer as well as incurring significant execution overhead. Setting k for the k-bounded loop is a non-trivial task, and does not lend itself easily to compiler analysis. Whereas k-bounding assumes an eager model of parallelism exposure, run-time, demand-based approaches such as lazy task creation [12] or guided self-scheduling [15] may prove more effective, as well as easier to program – however, in general, these techniques are difficult to implement on Monsoon because it is almost impossible to determine when a Monsoon processor is idle because of the deeply-pipelined, heavily-interleaved model of execution. Again, this is not an inherent feature of dataflow architectures, but an oversight in the hardware design of Monsoon.

Language definition and compiler optimization also affect performance. Id's policy of array extents being defined only at run-time inhibits some standard loop optimizations, and may be changed by either changing the language or adding pragmas. Some compiler loop optimizations for conventional architectures may not be effective for Monsoon because the addition of induction variables is very expensive for the dataflow loop schemas. However, some other loop optimizations may be more effective for Monsoon than conventional architectures because of the high overhead of dataflow loop execution.

The programmer plays a large part in attaining good performance. Loop schemas and k-bounds should be chosen to expose just enough parallelism to keep the machine busy. Changes to the algorithm in order to take advantage of complete loop unfolding, such as the hybrid version of the PCG, can also improve performance in non-intuitive ways.

In related work, Arvind, et.al. [3] gives a more detailed analysis of the overheads of dataflow execution. Hicks, et.al. [9] describes performance of several smaller benchmarks on Monsoon. Yeung and Agarwal [21] discuss the importance of language support for the expression of fine-grained parallelism using preconditioned conjugate gradient (primarily the preconditioning step, which is significantly different from ours) on Alewife. Sur and Bohm [18] have performed many performance analyses of scientific codes in Id on Monsoon, primarily focusing on language features and their impact on performance.

Although it is unlikely that dataflow architectures (as they exist today) will be viable commercially in the near future, multithreading techniques and parallelism are playing an increasingly important role in the design of com-

[1] Note that the seemingly unusual 168×90 problem size is a real size used in production runs, and was not chosen to make the Monsoon performance seem relatively better than the CM-5.

mercial systems. The synchronization and control overheads we encounter in dataflow computing are inherent to fine-grained parallel multithreaded execution, and these overheads change the relative importance of program optimizations, motivating increases in sizes of basic blocks through inlining, loop unrolling, and program structuring. The higher overhead of control instructions also motivates decreases in the sizes of basic block interfaces, even when the size of the basic block may be decreased by adding a basic block interface.

Acknowledgements

This research was partially funded by ARPA under ONR contract N00014-92-J-1310, and grew from a joint effort with John Marshall and Chris Hill in developing a large ocean circulation model with the MIT Center for Meteorology and Physical Oceanography. Kyoo-Chan Cho performed the original port of the ocean model to Id.

References

[1] B. S. Ang. Efficient implementation of sequential loops in dataflow computation. In *Proceedings of the Functional Programming Computer Architecture*, June 1993. ftp://csg-ftp.lcs.mit.edu/pub/papers/misc/ seq-loop.ps.Z.

[2] Arvind, K.-C. Cho, C. Hill, R. P. Johnson, J. Marshall, and A. Shaw. A Comparison of Implicitly Parallel Multithreaded and Data Parallel Implementations of an Ocean Model based on the Navier-Stokes Equations. Technical Report CSG Memo 364, Laboratory for Computer Science, MIT, 1991. http://www-csg.lcs.mit.edu/GCM/.

[3] Arvind, D. E. Culler, and K. Ekanadham. The price of asynchronous parallelism: an analysis of dataflow architectures. In *Proceedings of CONPAR 88*, pages 541–555, Sept. 1988.

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994. http://www.netlib.org/linalg/html_templates/ Templates.html.

[5] D. Chiou. Activation Frame Memory Management for the Monsoon Processor. Master's thesis, MIT EECS, September 1992. http://www-csg.lcs.mit.edu/∼derek/MS-Thesis.ps.

[6] D. E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs*. PhD thesis, MIT EECS, 545 Technology Square, Cambridge, MA 02139, 1990. LCS TR-446.

[7] D. E. Culler and G. M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, [12] 1990.

[8] V. Grafe, G. Davidson, J. Hoch, and V. Holmes. The Epsilon Dataflow Processor. In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 36–45, May 29-31 1989.

[9] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance Studies of the Monsoon Dataflow Processor. *Journal of Parallel and Distributed Computing*, 18(3):273–300, July 1993. http://www-csg.lcs.mit.edu/∼derek/Monsoon-Performance.ps.

[10] W. D. Hillis and L. W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11):31–40, Nov. 1993.

[11] H. Jones and J. Marshall. Convection with rotation in a neutral ocean: a study of open deep convection. *Journal of Physical Oceanography*, 23(6):1009–1039, 1993.

[12] E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Computing*, 2(3):264–80, July 1991. ftp://cag.lcs.mit.edu/pub/papers/futures.ps.Z.

[13] R. S. Nikhil. Id Language Reference Manual, Version 90.1. Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT, July 1991.

[14] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monograph in Parallel and Distributed Computing. MIT Press, 1992.

[15] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–39, December 1987.

[16] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of ISCA*, pages 46–53, 1989.

[17] A. Shaw, Y. Kodama, M. Sato, S. Sakai, and Y. Yamaguchi. Performance of Data-Parallel Primitives on the EM-4 Dataflow Parallel Supercomputer. In *Proceedings of Frontiers '92*, pages 302–309, McLean, VA, October 1992.

[18] S. Sur and W. Bohm. Functional, I-structure, and M-structure implementations of NAS benchmark FT. In *Parallel Architectures and Compilation Techniques*, pages 47–56, August 1994. http://www.colostate.edu/ ∼dataflow/papers/pact94b.ps.

[19] Thinking Machine Corporation. *CM Fortran Language Reference Manual, V2.1*, January 1994.

[20] M. Wolfe. *High Performance Compiler for Parallel Computing*. Addison Wesley, 1995.

[21] D. Yeung and A. Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *Proceedings of the Fourth Symposium on Principles and Practices of Parallel Programming*, pages 187–197, May 1993. ftp://cag.lcs.mit.edu/pub/papers/fine-grain.ps.Z.