A Comparison of Implicitly Parallel Multithreaded and Data Parallel Implementations of an Ocean Model

Andrew Shaw* Arvind* Kyoo-Chan Cho* Chris Hill† R. Paul Johnson* John Marshall†

Abstract

Two parallel implementations of a state-of-the-art ocean model are described and analyzed: one is written in the implicitly parallel language Id for the Monsoon multithreaded dataflow architecture, and the other in data parallel CM Fortran for the CM-5. The multithreaded programming model is inherently more expressive than the data parallel model. One goal of this study is to understand what, if any, are the performance penalties of multithreaded execution when implementing a program that is well suited for data parallel execution. To avoid technology and machine configuration issues, the two implementations are compared in terms of overhead cycles per *required* floating point operation. When simulating flows in complex geometries typical of ocean basins, the data parallel model only remains efficient if redundant computations are performed over land. The generality of the Id programming model allows one to easily and transparently implement a parallel code that computes only in the ocean. When simulating ocean basins with complex and irregular geometry the normalised performance on Monsoon is comparable with that of the CM-5. For more regular geometries, that map well to the computational domain, the data-parallel approach proves to be a better match. We conclude by examining the extent to which clusters of mainstream symmetric multiprocessor (SMP) systems offer a scientific computing environment which can capitalize on and combine the strengths of the two paradigms.

1 Introduction

In this paper we present a case study of a state-of-the-art ocean circulation model [32, 31] implemented in Id (a truly implicitly parallel language) [34] on Monsoon (a multithreaded dataflow machine) [38] and then critically compare it with the same algorithm developed in a data parallel Fortran dialect on the CM-5 (a massively parallel, distributed memory, Von-Neumann architecture) [25]. The ocean model contains abundant data parallelism and attains good performance on the CM-5. The process of implementing the multithreaded Id version gives us insight into the techniques necessary to attain good performance, on real scientific applications, using implicitly parallel languages, multithreaded architectures and data-driven computation.

Multithreading is the simultaneous parallel execution of multiple threads of computation: threads can spawn new threads and may synchronize with other threads. In the arbitrarily-general form of multithreading presented by Id on Monsoon the sequence of computation is entirely data-driven and controlled solely by data dependencies in the underlying algorithm. This allows all forms of parallelism in an algorithm to be exploited.

Despite the elegance of this approach, the arbitrarily-general multithreading programming model has not gained widespread usage in parallel scientific codes. Instead the single program multiple data (SPMD) approach has been widely adopted for production parallel codes. Here the inherent data parallelism common to many scientific codes is exploited and a rigid regime of synchronous computation followed by communication is followed. Although an expressively restrictive framework which only exploits certain forms of parallelism, this programming model is generally acknowledged as being highly efficient for many scientific codes in which the data structures are highly regular.

The data parallel model implementation examined here uses CM Fortran, which exploits data parallelism at the statement level. Another common approach in scientific computing uses shared-memory or message-passing multi-processing to exploit data parallelism at the subprogram level. In either case the computational model is limited to, and can only really work efficiently for, problems that are predominantly data parallel and regular. In contrast, the fully general multithreaded environment presented by Id on Monsoon can exploit all the forms of parallelism, including data parallelism, that an algorithm possesses, and so imposes less restrictions on expression. However, a lack of mainstream software and hardware support and a conseqent lack of absolute performance have precluded much serious attention being given to such gen-

*Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

†Center for Meteorology and Physical Oceanography, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

eral multithreading by the architects of present day parallel scientific applications.

Even within the discipline of computer science, there are relatively few studies of multithreaded implementations of large scientific applications carried out on multithreaded architectures. Closely related to this study, Hicks, et.al. [22] considered performance of Id on Monsoon for several applications and compared it to sequential performance in C or Fortran 77 on a conventional RISC architecture. Yeung [48] uses the preconditioned conjugate gradient algorithm to study the effects of architectural support for synchronization on the MIT Alewife architecture. Hiromoto, et.al. [27] have performed detailed performance studies of various scientific applications for the Denelcor HEP. Hammes, et.al. [21] have performed a comparison of Id and Haskell for a Monte Carlo photon transport code. Sur and Bohm [46] study different Id implementations of FFT in the context of solving partial differential equations and implementations of the Dongarra-Sorensen eigensolver [45]. Arvind and Ekanadham [8] have compared the implementation of the hydrodynamics modeling application SIMPLE in Id and in Fortran; Arvind, et.al [7] have also explored the benefits of fine-grained parallelism in scientific applications. In one of the original papers about programming for dataflow machines, Dennis, et.al. [16] describe the process of implementing a weather modeling code for a theoretical static dataflow architecture as well as the sources of parallelism within the code.

Other related work includes high-performance implementations of functional languages. Most notably, SISAL has shown very good performance on a number of "conventional" parallel architectures [12] as well as multithreaded architectures [20] [44] [36] and has been used to write a number of scientific applications [12]. Miranda has been used to code an oil reservoir simulation [37]. Haskell has been used to code a parallel finite-element problem [19].

The authors believe this study is the first detailed system-level comparison of a large, real scientific application implemented in a data parallel environment *and* a multithreaded environment. Comparison is possible, but definitive conclusions are difficult to draw because of the differences in languages, compilers, architectures, technology, and investment levels in manpower. Nonetheless, in accounting for some of these differences, we conclude that overheads for the two implementations come from very different sources and that, surprisingly, they balance out for realistic problems.

The paper is organized as follows: Section 2 begins with a brief algorithmic description of the model and an analysis of its computational requirements (readers primarily interested in the computer science aspects of this study may choose to skip Section 2 without much loss of continuity). Section 3 and Section 4 describe the programming, optimization, and best case performance of the data parallel and multithreaded versions of the ocean model, respectively. Section 5 compares the performance of the two versions when they are adjusted for overheads seen in real problem definitions. Finally in Section 6 presents the relevance of this research in the context of current trends in the design of high-performance parallel systems.

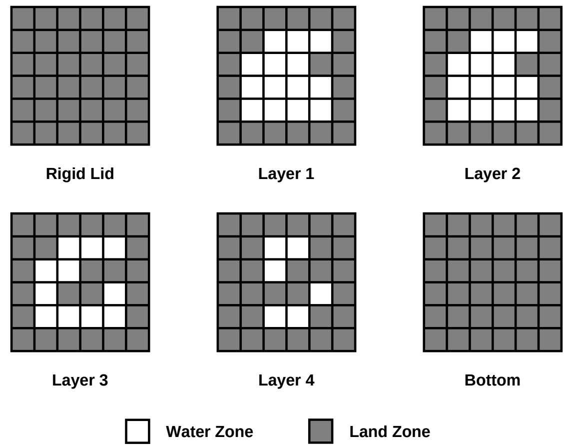

Figure 1: Ocean states are represented as three dimensional arrays. Some elements of the arrays represent land zones and some represent water zones.

2 The ocean model

The model used in this study is the MIT ocean circulation model, referred to in this paper as the "General Circulation Model" or GCM. It is being actively used in oceanographic research, and daily production runs are performed on a CM-5 and, more recently, on high-end symmetric multi-processor systems.

The physical basis of the model and its continuous and discrete forms are described briefly in the Appendix, and more detailed accounts can be found in [32], [31] and [23]. The model is based on the incompressible Navier-Stokes equations and can be used to study the ocean from convective ($\sim$100m in the horizontal) up to global scales. The numerical procedure involves alternating prognostic and diagnostic steps: prognostic to step forward in time the ocean currents and thermodynamic variables, diagnostic to find the pressure field. The "pressure correction" method is used to ensure that the evolving velocity field remains non-divergent. A Poisson equation with Neumann boundary conditions is solved to diagnose for the pressure field. This latter step is a computationally challenging one because it involves communication of information across

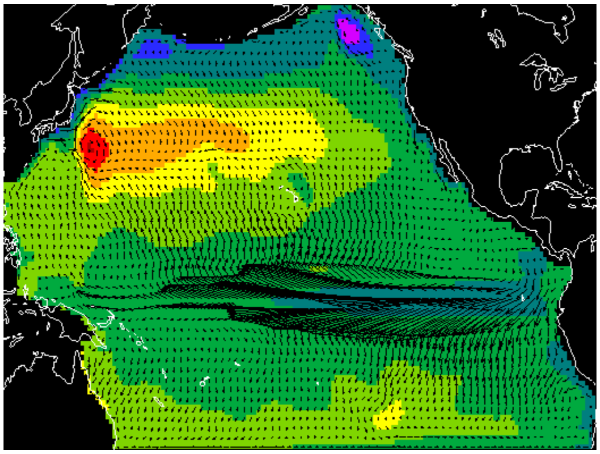

Figure 2: A significant percentage of real ocean geometries may be land. In this 171 × 93 × 4 Pacific basin geometry, 40.0% of the zones represent land. Land zone states remain constant over time.

the whole model grid to the boundary, in a geometry as complicated as that of an ocean basin.

2.1 Ocean geometry and state

The ocean is confined to a basin that can have a highly irregular geometry. A *geometry*, which defines the shape of the ocean basin, is represented as a three-dimensional array of finite volumes that we call *zones*. The faces of the zones must be chosen to coincide with coordinate surfaces, except when they abut a solid boundary (the coast or bottom). At these solid boundaries the zones may be "sculptured" to fit them to irregularities in the topography [31, 1]. Each element of the geometry array specifies whether the corresponding zone is land or water. We will assume that the ocean has a rigid lid at the upper surface. The rigid lid is a device to filter from the model rapidly propagating surface gravity waves that would severely restrict the possible length of the model time-step. For some integrations a free surface is admitted, but handled implicitly as discussed in [31]. The geometry array is constructed from the bathymetry – a two-dimensional array of depths for each column of the ocean at a chosen horizontal resolution. A highly idealized geometry for a 4 × 4 ocean with 4 layers of water is shown in Figure 1. The white zones are water; the gray zones land. A more realistic geometry is shown in Figure 2, representing the Pacific basin at the resolution of 1 degree of latitude by 1 degree of longitude; there are 171 zones along each line of parallel, 93 zones along each line of longitude and 4 zones in the vertical. In typical ocean modeling applications the number of zones stacked on top of one another is between 1 and 100, generally fewer than the number employed in the horizontal.

Each zone has six faces, and these faces categorized according to orientation:

- *longfaces*: vertical faces running along longitudes
- *parfaces*: vertical faces running along parallels
- *horfaces*: horizontal faces

The faces of the zones remain fixed during the integration over time; they are dependent only on the geometry of the ocean basin and the resolution at which it is discretized.

The state of the ocean is described by the velocity $\vec{v} = (u, v, w)$, the pressure p and the density ρ. The density is a function of the (potential) temperature T, the salinity S and pressure p. Zone quantities – p, ρ, T and S – are defined at the center of the zone and are implemented as *zone arrays*. Face quantities are defined on the faces of the zones – velocity components $\vec{v}$ – in *longface arrays* (u), *parface arrays* (v) and *horface arrays* (w). Zone and face arrays are three-dimensional arrays, and typically there are a dozen such arrays in use at any one time.

In certain applications of the model (see [32]) appropriate periodicity conditions must be employed.

2.2 The numerical algorithm

The heart of the model is a main time-stepping loop that is repeated many times. Depending upon the dynamical process being studied each time step may represent several hours of time or just a few seconds; a typical simulation entails thousands or millions of time steps. The main loop computes the changes in the values of $\vec{v} = (u, v, w)$, p, T and S by applying the laws of classical mechanics and thermodynamics to the fluid contained within the zones. Discrete forms of the continuous equations are deduced by integrating them over the zones and using Gauss's theorem. From p, T and S the density ρ is computed.

During each time step, it is also necessary to compute the forces normal to the faces of the zones to yield, according to Newton's Laws, the acceleration of fluid parcels. This is done by evaluating the pressure-gradient G, inertial, Coriolis, and applied stresses and frictional forces, as described in [32, 31].

A high-level description of GCM is shown in Figure 3. Steps 1, 3, 4, 5 and 6 only require local communication and nearest-neighbor communication to update ocean state arrays, but Step 2 necessitates solving the following linear system, a discrete form of the Poisson equation for the pressure:

$$\mathbf{A}p^{n+1} = f(\vec{G_v}, \vec{v}^n) \tag{1}$$

where $\mathbf{A}$ is a sparse $N \times N$ matrix.

GCM ()
$\vec{v}^0, p^0, T^0, S^0, \rho^0$ = *compute_initial_ocean_states*()
for each time step n
Step 1: $\vec{G_v}^*$ = *estimate_Gv*($\vec{v}^n$,p^n,ρ^n)
Step 2: p^{n+1} = *diagnose_pressure*(p^n,$\vec{v}^n$,$\vec{G_v}^*$)
Step 3: $\vec{v}^{n+1}$ = *advance_velocity*(p^{n+1},v^n,$\vec{G_v}^*$)
Step 4: T^{n+1} = *advance_temperature*(T^n,$\vec{v}^{n+1}$)
Step 5: S^{n+1} = *advance_salinity*(S^n,$\vec{v}^{n+1}$)
Step 6: ρ^{n+1} = *calculate_density*(T^{n+1},S^{n+1})
return($\vec{v}^{final}$,p^{final},T^{final},S^{final},ρ^{final})

Figure 3: High-level specification of the GCM algorithm.

Our algorithm is designed to perform efficiently across the scales of interest in the ocean, from non-hydrostatic phenomena on the smallest scales, to the highly balanced hydrostatic flows on large scales. The approach, which involves separating the pressure field in to its hydrostatic, surface and non-hydrostatic components, is set out in [32]. In the hydrostatic limit, Equation (1) is a two-dimensional inversion for the surface pressure, with $N = N_x \times N_y$. The pressure in the interior of the ocean is then obtained on intergation vertically using the hydrostatic approximation. But in non-hydrostatic applications of the model $N = N_x \times N_y \times N_z$ is the total number of zones, p is a vector of length N carrying the pressure of each zone, and f is the source function. In typical ocean modeling applications, N_x and N_y range from 100 to 1,000; and N_z from 1 to 100. Thus, $\mathbf{A}$ is potentially huge, and special techniques must be used to invert Equation (1) for p^{n+1}. Because non-hydrostatic simulations are the most demanding of computation, that limit is the focus of attention here.

2.2.1 The elliptic problem

Solving Equation (1) efficiently requires an understanding of the form taken by the matrix $\mathbf{A}$; its structure is sketched schematically in Figure 4. The column vector p in Equation (1) comprises the pressure in every zone in the ocean. Its singly subscripted elements are p_l, where first points in each vertical column are enumerated first, then points in the horizontal directions, as follows:

$$l = k + N_z(j-1) + N_z N_y (i-1)$$

where i is an index increasing eastwards, j is an index increasing southwards and k increases downwards. There are $N = N_x \times N_y \times N_z$ elements in total.

The elements of $\mathbf{A}$ comprise the discrete representation of the Laplacian operator ∇^2 (in the three space dimensions), suitably modified to take account of boundary conditions on the upper, lower and lateral boundaries; it is

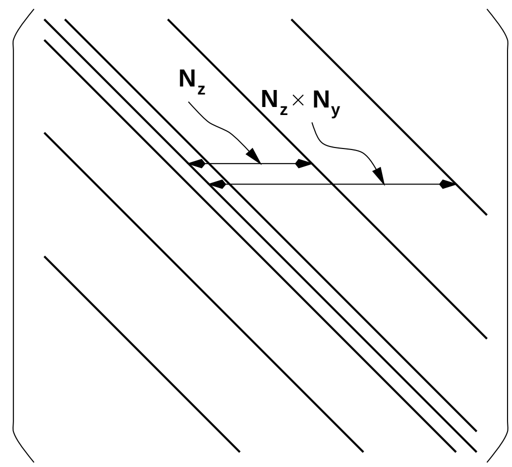

Figure 4: Matrix $\mathbf{A}$, of size $N \times N$ where $N = N_x \times N_y \times N_z$.

constant in time and depends only on the basin geometry. $\mathbf{A}$ is symmetric and, although huge, has only 7 diagonals representing the coupling in the 3 space dimensions. The three central diagonals are flanked, on both sides, by diagonals displaced a distance N_z and $N_z \times N_y$ from the central diagonal, as indicated in Figure 4.

To write down the complete form of $\mathbf{A}$ consider the representation of $\nabla^2 p$ for one particular zone (labelled C) in terms of the six surrounding zones (labelled U, L, N, E, S and W). Suppose that the dimensions of the zones are $\triangle x$ by $\triangle y$ by $\triangle z$, where $\triangle x$ is its length in the eastward direction, $\triangle y$ is its length in the southward direction, and $\triangle z$ is the length in the vertical extent of the zone. Then $\nabla^2 p$, to an accuracy which is second order in the $\triangle x$, can be written as follows (keeping $\triangle x = \triangle y$ and $\triangle z$ constant for simplicity).[1]

$$\begin{aligned}
\nabla^2 p &= \frac{p_U + p_L - 2p_C}{(\triangle z)^2} + \frac{p_N + p_S + p_E + p_W - 4p_C}{(\triangle x)^2} \\
&= \frac{2}{(\triangle z)^2}[\underbrace{\{\frac{1}{2}p_U - (1+2\epsilon)p_C + \frac{1}{2}p_L\}}_{same\ vertical\ column} + \\
&\qquad \underbrace{\epsilon\{p_N + p_S + p_E + p_W\}}_{same\ horizontal\ plane}]
\end{aligned} \tag{2}$$

where $\epsilon = (\frac{\triangle z}{\triangle x})^2$ measures the aspect ratio of the zones.

The detail of the structure underlying $\mathbf{A}$ is now readily understandable; the three leading diagonals are the coefficients multiplying p's in the same *vertical column* of ocean (p_U, p_C and p_L); the four diagonals in the wings are the coefficients multiplying p's in zones surrounding the zone of interest in the same *horizontal plane* (p_N, p_S, p_E and p_W).

[1] In applications, $\triangle x$, $\triangle y$ and $\triangle z$ may vary and often do.

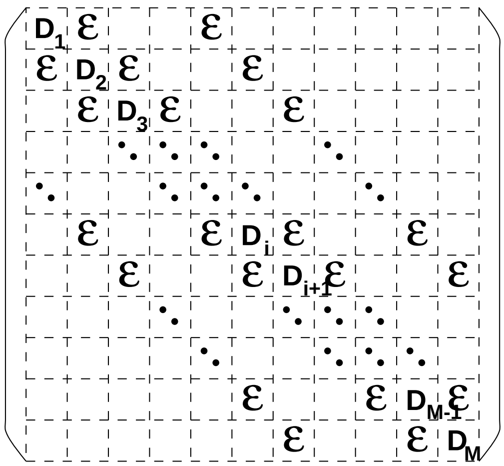

Figure 5: Blocked representation of the Matrix $\mathbf{A}$. $M = N_x \times N_y$, and each block is of size $N_z \times N_z$. The $\mathbf{D}$ and ε blocks are themselves tridiagonal and diagonal matrices, respectively.

The inner three diagonals of $\mathbf{A}$ can thus be blocked and arranged as shown in Figure 5 where each block $\mathbf{D}$ is a tri-diagonal matrix representing the communication between zones in the same vertical column of ocean; $\mathbf{D}$ is a matrix of size $N_z \times N_z$ where there are N_z zones in each column of ocean (there are $N_x N_y$ such blocks, one for each column of ocean). The ε's are $N_z \times N_z$ matrices made up of ϵ along the diagonal and zero elsewhere:

$$\varepsilon = \begin{pmatrix} \epsilon & & & \\ & \epsilon & & \\ & & \ddots & \\ & & & \epsilon \end{pmatrix}$$

For example, if the ocean were only four zones deep at a particular horizontal position then by inspection of Equation (2) the $\mathbf{D}$ would have the form:

$$\mathbf{D} = \begin{pmatrix} -(\frac{1}{2}+2\epsilon) & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & -(1+2\epsilon) & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -(1+2\epsilon) & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & -(\frac{1}{2}+2\epsilon) \end{pmatrix}$$

Note that the elements in the top left and bottom right of the matrix have been appropriately modified to take account of the boundary conditions at the top and bottom of the ocean. If at another horizontal position, the ocean were only three zones deep, then $\mathbf{D}$ would have zeros in the last row and column.

The entries of $\mathbf{A}$ must also be modified for zones that are laterally adjacent to a rigid boundary, whether it be the coast, a submerged island or an island that cuts the ocean's surface. For example, suppose the north face of zone C is the coast. Then homogeneous Neumann conditions on p must be imposed here (see [32, 31]) which, in discrete form is:

$$\frac{p_N - p_C}{\triangle x} = 0$$

and the entries of ε are appropriately modified; zeros will appear at appropriate positions along the diagonal. $\mathbf{A}$, however, will remain symmetric.

One further property of $\mathbf{A}$ must be noted. ϵ is typically very much less than unity. This is because the ocean is a thin film of fluid filling the basin that is several thousand kilometers wide but only a few kilometers deep – the $\nabla^2 p$ operator is dominated by $\frac{\partial^2}{\partial z^2}$. So $\mathbf{A}$ is dominated by the elements of the blocks $\mathbf{D}$. This is exploited by the solution method.

The size of $\mathbf{A}$, and the distance of the outer diagonals from the main diagonal make direct methods of solving the system difficult for large geometries because the inverse of $\mathbf{A}$ is dense. Operating with the inverse would involve N^2 multiplications, and this is an unrealistic task given that typically $N > 10^5$. An iterative procedure is adopted that exploits the sparseness of $\mathbf{A}$ and its diagonal dominance. The procedure involves repeated multiplication of the iterative solution by $\mathbf{A}$ and by another sparse matrix $\mathbf{K}$, an approximate inverse of $\mathbf{A}$; $\mathbf{K}$ is called the preconditioner. The iterative algorithm employed to solve the linear system is called the *preconditioned conjugate gradient algorithm* and it is described in more detail in the next section.

2.2.2 Preconditioned conjugate gradient algorithm

The preconditioned conjugate gradient (PCG) algorithm has been extensively studied and is a popular algorithm for implementation on highly parallel platforms because it is easily parallelized. Barrett, et.al. [9] give a good overview of various iterative methods for solving linear systems including conjugate gradient; Golub and Van Loan [17] is a good general reference for matrix computations, including conjugate gradient and Shewchuk [42] gives a clear and light introduction to conjugate gradient. Here, a brief derivation of the preconditioned conjugate gradient method is provided.

Pre-multiply Equation (1) by a (carefully chosen) preconditioning matrix $\mathbf{K}$, which is an approximate inverse of $\mathbf{A}$. Equation (1) can then be written (dropping superscripts) as

$$(\mathbf{I} - \mathbf{C})p = \mathbf{K}f$$

where $\mathbf{C} = \mathbf{I} - \mathbf{KA}$. If $\mathbf{C}$ is *close* to zero then the above suggests the following iterative scheme, where i is the iteration step:

$$p^{i+1} \quad = \quad \mathbf{C}p^i + \mathbf{K}f$$

$$= \quad p^i + b^i$$

where

$$b^i = \mathbf{K}r^i$$

is called the *search direction* and

$$r^i = f - \mathbf{A}p^i$$

is the *residual vector*. The r^i's and b^i's can be deduced from the previous iteration using the relations:

$$\begin{aligned} r^{i+1} &= r^i - \mathbf{A}b^i \\ b^{i+1} &= \mathbf{K}r^{i+1} \end{aligned}$$

The above iterative procedure, referred to as the Richardson iteration, can be accelerated by choosing search directions in a more optimal way. In GCM, the *conjugate gradient method* is adopted, which selects search directions as linear combinations of the previous search direction b^i and current gradient r^{i+1} modified by the preconditioner $\mathbf{K}$:

$$\begin{aligned} b^{i+1} &= \xi^{i+1} + \beta b^i \\ \text{where} \quad \xi^{i+1} &= \mathbf{K}r^{i+1} \end{aligned}$$

Conjugate gradient also selects a parameter α to minimize the magnitude of the residual vector as measured by $e \cdot \mathbf{A}e$, where (if there were no preconditioning) $e = p^i - p^{answer}$ is the error vector along the direction b^i. Choosing the optimal values of α and β results in the algorithm shown in Figure 6.

We have designed a preconditioner $\mathbf{K}$ so that (i) it can be efficiently stored, (ii) the number of operations one has to perform when multiplying by it is as small as possible, (iii) it is a good approximation to $\mathbf{A}^{-1}$, allowing the iterative procedure converge more rapidly and (iv) it requires as little communication as possible in the data parallel implementation. After considerable experimentation, we have chosen a *block-diagonal* preconditioner (a matrix $\mathbf{K}$ whose diagonal is made up of the inverse of the tri-diagonal matrices $\mathbf{D}$ defined above):

$$\mathbf{K} = \begin{pmatrix} \mathbf{D}_{1,1}^{-1} & & & & \\ & \mathbf{D}_{1,2}^{-1} & & & \\ & & \mathbf{D}_{1,3}^{-1} & & \\ & & & \ddots & \\ & & & & \mathbf{D}_{N_x,N_y}^{-1} \end{pmatrix}$$

Evaluation of the inner products to compute α and β in the above involves forming the vector ξ, where

$$\xi = \mathbf{K}r$$

PCG($\mathbf{A}$, f)
$r^0 = f - \mathbf{A}p^0$;
$b^0 = \mathbf{K}r^0$
$\xi^0 = b^0$
while $\sqrt{r^i \cdot r^i} > \tau$ **do**
$\gamma = (r^i \cdot \xi^i)$
$\alpha = \gamma \;/\; (b^i \cdot \mathbf{A}b^i)$
$p^{i+1} = p^i + \alpha b^i$
$r^{i+1} = r^i - \alpha \mathbf{A}b^i$
$\xi^{i+1} = \mathbf{K}r^{i+1}$
$\beta = (r^{i+1} \cdot \xi^{i+1}) \;/\; \gamma$
$b^{i+1} = \xi^{i+1} + \beta b^i$
$i = i + 1$
return(p^{final})

Figure 6: The preconditioned conjugate gradient algorithm, which solves the linear system $\mathbf{A}p = f$ in an iterative fashion. The variable τ is the tolerance and p^0 is an initial estimate of p^{final}.

Multiplication by the preconditioner K is therefore $N_x \times N_y$ independent ($N_z \times N_z$) matrix multiplications, each one corresponding to a column of water at i, j:

$$\xi_{ij} = \mathbf{D}_{i,j}^{-1} r_{i,j}$$

Dispensing with the subscripts,

$$\xi = \mathbf{D}^{-1} r$$

If the ocean model has many levels, then $\mathbf{D}^{-1}$ will also be dense; consequently, storing and multiplying by $\mathbf{D}^{-1}$ is also demanding of resources. So, GCM exploits the fact that $\mathbf{D}$ is tri-diagonal, and use LU decomposition to solve the preconditioning equations for ξ in the form:

$$\mathbf{D}\xi = r$$

We write $\mathbf{D} = \mathbf{LU}$ where $\mathbf{L}$ is a lower-triangular matrix and $\mathbf{U}$ an upper triangular matrix. Solving for ξ is then equivalent to solving the two sets of equations:

$$\begin{aligned} \mathbf{L}\xi' &= r \\ \mathbf{U}\xi &= \xi' \end{aligned}$$

first for ξ' and then for ξ. This is straightforward because of the triangular structure of $\mathbf{L}$ and $\mathbf{U}$. (See [15] for more details of the tri-diagonal LU decomposition.)

Ignoring the operations for LU decomposition, these forward and backward substitution steps involve $O(N_z)$ operations as compared to $O(N_z^2)$ if we had used the inverse directly. However, because the geometry (and consequently $\mathbf{A}$) are fixed, LU decomposition of $\mathbf{D}$, the block

Floating Point Ops per Time Step per Zone	
Step 1: *estimate_Gv*	467
Step 2: *diagnose_pressure*	32 × PCG iter. + 32
Step 3: *advance_velocity*	20
Step 4: *advance_temperature*	122
Step 5: *advance_salinity*	118
Step 6: *calculate_density*	6
Total	32 × PCG iter. + 765

Figure 7: Step 2 (*diagnose_pressure*) of the time stepping loop contains the preconditioned conjugate gradient algorithm. In the situations we focus on in this study, this step may iterate hundreds of times until convergence, making it the dominant step in the algorithm.

Floating Point Ops per PCG iteration per Zone			
PCG component	**occurrences**	**FLOPs**	**Total**
3D inner product	4	2	8
3D daxpy	3	2	6
7-point stencil	1	13	13
preconditioner	1	5	5
PCG Total			32

Figure 8: PCG is composed of four simple components. Each component performs a set number of floating point operations per zone, and some components occur more than once in PCG.

diagonal elements of $\mathbf{A}$, is performed only once. Notice that the multiplication by the preconditioner is actually implemented as $N_x \times N_y$ independent tridiagonal linear equation solvers of size $N_z \times N_z$.

The resulting preconditioner was found to be a good compromise: it significantly reduces the number of iterations required to find a solution to Equation (1), it is an acceptable approximation to the inverse of $\mathbf{A}$, it is sparse and its application requires no communication across the network in the data parallel implementation of the algorithm.

2.3 Computational characteristics

To effectively program GCM for a parallel platform, we first determine how much computation must be performed in each step of the algorithm and then consider what sources of parallelism are available.

2.3.1 Distribution of overall computation time

Figure 7 shows how many floating point operations are required in the GCM's main time-stepping loop. There are also initialization and termination phases that are not considered in this table. These only occur once per model run, and will not be significant factor in long simulations.

From Figure 7 we can see that a fully general implementation of GCM requires that the PCG algorithm be implemented efficiently. For example if the number of PCG iterations exceeds 300 per time step, the *diagnose_pressure* stage will dominate the total count of floating point operations, containing more than 90% of the operations. This scenario is typical of extremely high-resolution ($\sim$ 100m resolution laterally and vertically) non-hydrostatic ocean simulations, the focus of this study.

Time is also spent in communications, non-floating point computation, and idling due to load imbalances; however, for GCM, the style of computation in all of the steps is similar enough that floating point operation counts give a fair view of the distribution of time. To optimize GCM, we must concentrate on the time spent in non-essential floating point and other overhead operations, because given a particular model run, the number of essential floating point operations is fixed.

Although ocean modeling applications do perform substantial amounts of I/O we do not analyze I/O costs here. Rather, we focus on the comparison of two different computational models. The difference in performance of I/O susbsystems is not predominantly a function of the computational model. Thus, to expose the impact of the computational models, we have removed I/O costs from both implementations.

2.3.2 Distribution of computation in PCG

The PCG algorithm itself can be further decomposed into four simple components: 3D inner product, 3D daxpy, a 7-point stencil and the preconditioner. These components can be seen in the algorithmic outline in Figure 6; the actual number of floating point operations contributed by each component is shown in Figure 8.

Each of the components operates on three-dimensional state arrays. The 7-point stencil component implements $\mathbf{A}b$ (the multiplication of the Laplacian operator) and the preconditioner component performs the multiplication of the preconditioner $\mathbf{K}$, which is actually implemented as a tridiagonal linear equation solvers (see Section 2.2.2).

To efficiently implement the PCG algorithm, each of the components must also be implemented efficiently. In Sections 3 and 4, the implementations of these components in both the data parallel and multithreaded models are described.

2.3.3 Available parallelism

Most of the potential parallelism in GCM comes from the *data parallelism* in updating each of the large ocean state arrays. This parallelism is typically sufficient to keep a large parallel computer busy and, because this computation is so regular, it lends itself well to vector computation.

There is also *procedure-level parallelism* available: for example, much of Step 4 can be done in parallel with Steps 1, 2 and 3, and within individual steps some blocks can be executed in parallel. In Step 2, for instance, at least three separate state arrays can be computed in parallel. This source of parallelism can be exploited by the multithreaded implementation, but not by the data parallel implementation.

Producer-consumer parallelism is also plentiful in GCM, because the computation of new state arrays always depends upon old state arrays. Given appropriate linguistic and architectural support, the computation of parts of new state arrays can begin as soon as the parts of the old state arrays they depend upon are computed.

3 Programming data parallel GCM

Given the algorithmic specification in Section 2, the high-level view of both the data parallel and multithreaded implementations are very similar. Both implementations define the same high-level procedures, and both have the same structure in the time-stepping loop.

We begin with the data parallel version because we feel that the reader is likely to be more familiar with it, and many studies have been done on the performance of the CM-5 and other data parallel platforms.

Although programming models are separate from the hardware they are implemented on, there is a very close relationship between what the programmer writes and what is executed on the hardware. Without a clear understanding of how the hardware works and how a high-level programming language construct is mapped to the hardware, the programmer cannot develop an efficient implementation of his application. Therefore, we first describe the hardware at a level of detail that is relevant to the user or the reader of this paper, then the programming and compiling of components of GCM, and finally evaluate the overall performance of the data parallel implementation of GCM.

3.1 The CM-5 hardware

Figure 9 shows a high-level view of the CM-5 architecture. Scalar computation and control are performed on the front-end workstation, which is typically a Sparc 10, while parallel computation is performed on the nodes. Each node consists of a 32 Mhz Sparc 2 processor with four 16 MHz vector units (VU's), with each VU capable of 32 Mflop/s 64-bit floating point peak performance. Memory is local to each node, and each VU has direct access to

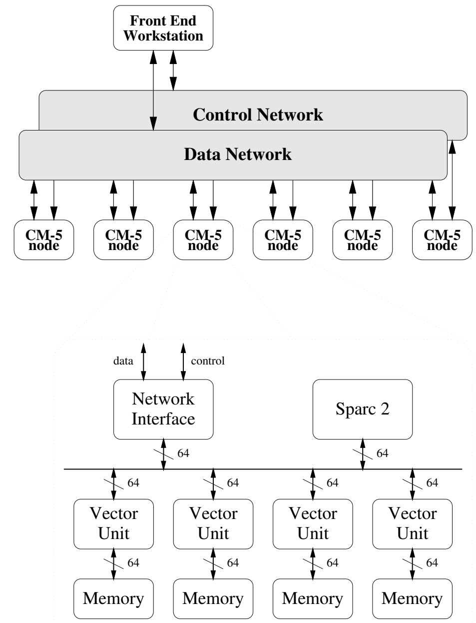

Figure 9: The CM-5 consists of nodes connected through a point-to-point data network and a global control network. A front end workstation is used for scalar computation, while nodes are used for parallel computation. Each node consists of a Sparc 2 microprocessor and four vector units, which provide most of the computing power.

its own bank of memory. Every node is connected to two separate networks: a data network and a control network, which perform different kinds of data parallel communications. The data network is capable of a maximum of 20 MB per second per node, and is optimized to handle short messages. The control network is used for global synchronization, broadcast and reduction operations.

The CM-5 nodes are controlled by the Sparc 2's, which have poor floating point performance. The Sparc 2's should be considered sequencers or controllers for the VU's, which are fairly powerful – a VU can simultaneously generate memory addresses, load data and execute pipelined floating point operations if properly scheduled by the CM Fortran compiler. Figure 10 shows a simplified version of VU assembly code emitted by the CM Fortran compiler for the inner loop of the $aX + Y$ daxpy computation. An estimated cycle count and exact floating point operation count are annotated by the compiler. (Note that Step S1 seems to belong outside of the loop; in this case, the CM Fortran compiler may not have been able to determine that it was a loop constant.)

The VU's can be very powerful and efficient when executing highly regular vector code, as seen in step S3,

	Operation	Cycles	FP ops
S1:	load scalar *a* into *S*	4	0
S2:	load 8 dwords of *X* into *V1*	8	0
S3:	load 8 dwords of *Y* into *V2*	8	16
	and $V2 = S \times V1 + V2$		
S4:	store *V2* into *Z*	8	0
	and bump offset		
	and compare offset to end		
	and goto S1 if not done		
Total		**28**	**16**

Figure 10: The sequence of steps executed by each vector unit for the daxpy computation $Z = aX + Y$ as generated by the CM Fortran compiler. *S* is a scalar register and *V1* and *V2* are vector registers that hold eight doublewords. The cycles counted are 16 MHz VU cycles and are only approximate counts used by the compiler.

where the VU performs a vector load simultaneously with a multiply-add operation. The Sparc 2 can concurrently execute with the VU, as is shown in step S4, where the loop compare-and-branch is handled by the Sparc 2, while the vector register store is handled by the VU.

If we only consider this daxpy inner loop, a floating point operation is executed every $28/16 = 1.75$ VU cycles. This ratio is considerably smaller than on most conventional RISC processors for this computation, especially for large problem sizes where the RISC processors would go out of cache. Because the VU is also a memory controller, the VU does not stall for memory operations that are vectorizable.

We should note, however, this inner loop of the daxpy computation is about as efficient as the CM-5 gets. Furthermore, there are other overheads even for this computation such as garbage padding, and the cycle count estimate generated by the compiler is overly optimistic. As we will see in Section 3.3, the actual speed of the daxpy code is not as fast as this inner loop would indicate.

The CM Fortran compiler can also generate code to run without VU's (i.e. directly on the Sparc processors), but the performance is exceptionally poor because the code generated for execution without VU's is naive, and the Sparc processors themselves have very poor floating point performance. Therefore, in this study we have chosen to show performance numbers only for the CM-5 with VU's.

All performance measurements on the CM-5 were taken with the CM-5 hardware timers, which can measure at microsecond resolution time spent by the front-end, the vector units, control communications and data communications. Front end scalar computation can execute at the same time as parallel computation on the nodes, so in the best case, time spent in scalar computation is completely masked by parallel computation, where we expect most of the work to be done.

3.2 Data parallel programming in CM Fortran

CM Fortran (CMF) [47] is a data parallel extension to Fortran 77 for the CM-5. Many of the features of CM Fortran are present in Fortran 90 and High Performance Fortran (HPF) [28]. CM Fortran includes syntax for array operations such as elementwise operations, permutations, and extraction of array subsections. The CM Fortran compiler is back compatible to Fortran 77, but will not automatically parallelize Fortran 77 code: the programmer must explicitly specify parallel array operations that the compiler distributes among the nodes.

Execution of a CM Fortran program on the CM-5 comprises alternating computation and communication phases with each phase separated by a global synchronization barrier. During the computation phases, the front end is reponsible for scalar computation such as the addition of two scalar values, as well as the overall control of the program, such as looping, procedure calls, and conditionals. The nodes are responsible for parallel computation, and all nodes perform identical operations on different data. Although the nodes may be closely synchronized, each node computes results and modifies data independently, and consequently modified data values are exclusive to individual nodes. At the end of a computation phase, communication between nodes and between the front-end and the nodes is orchestrated by the front-end.

Data parallel communication occurs primarily in two highly stylized manners:

- *Control* communications, which include barriers, reductions and broadcasts. These operations are mapped to the control network on the CM-5.
- *Data* communications, which cause the rearrangement of elements of an array, are mapped to the data network. They include nearest neighbor communications which are used in the GCM code, and more general permutations of data that are significantly more expensive.

In the next sections, we describe how the data parallel version of GCM is written, and some of the decisions we had to make to optimize its performance.

3.2.1 Array operations and layout directives

CM Fortran allows the expression of element-wise array operations to be specified very concisely. For instance, a three-dimensional daxpy computation $aX + Y$ can be

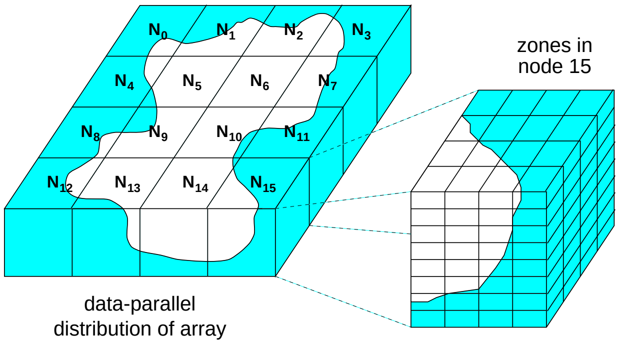

Figure 11: Array distribution on distributed-memory parallel computer. The water zones are in white, and land zones are in gray; distribution of water on nodes differs widely.

expressed as follows:

```
Z = a*X+Y
```

where X, Y and Z are declared statically as array variables, and a is a scalar variable.

Parallelism for the above expression is exploited by different nodes performing operations on different parts of the X, Y, and Z arrays. How the work is divided up is determined by the layout of the arrays across the nodes – each node only works on portions of the arrays that are in its local memory.

Array extents and layouts must be explicitly declared at compile-time. For example, a state array for GCM of size $171 \times 93 \times 5$ may be declared as follows:

```
      real X(5,171,93)
cmf$layout X(:serial,:NEWS,:NEWS)
```

Given this declaration, we have decided to map the up-down (k) dimension to the leftmost index, the east-west (i) dimension to the second index, and the north-south (j) dimension to the rightmost index.

The :serial layout declaration for the k axis places elements within the same column (i.e. with the same i and j coordinates) on the same processor. The :NEWS declaration spreads the elements along that axis across the machine. The resulting array is laid out along the nodes as shown schematically in Figure 11. All of the ocean state arrays in the GCM code have the same extents and layouts as described above.

For given array extents, array layout and machine size, the CM Fortran compiler cannot always distribute the array across the machine evenly, for the simple reason that array extents do not always divide evenly across the machine. Furthermore, the unit of division is not nodes, but VU's, so for a 32 node CM-5, the two :NEWS dimensions of the array must be divided across 128 VU's. The array X as declared above is therefore padded out to size (5,172,96) to fit evenly on the CM-5 – the additional padding elements are "garbage" elements that are added to the sides of the arrays.[2] These garbage elements are ignored by the VU hardware but add overhead in the sense that a problem of size $171 \times 93 \times 5$ requires exactly the same amount of time to execute as a problem of size $172 \times 96 \times 5$.

Because the array layout declarations affect padding, we must choose the layouts carefully not just to minimize inter-processor communications, which is the primary concern in selecting a layout, but also to make sure that there no more padding than is necessary. For example, if we decide to make only one of the axes :NEWS, then more garbage padding is necessary to make the array distribution even over the machine. Because there are 128 VU's, the :NEWS axis must be padded out to a multiple of 128.

If we have the option of choosing our problem sizes, it is usually easy to choose a problem size which does not require too much padding, if any. In the case of GCM, the problem size is often determined by external factors (such as the data collection intervals of satellites), so a seemingly odd problem size like 171×93 is actually used despite the additional, though generally small, overhead it may incur.

3.2.2 Array reduction

The vector inner product of two arrays $X \cdot Y$ can be expressed as follows:

```
b = sum(X*Y)
```

where sum is a CM Fortran function that sums the elements of the array passed by its argument. The control network is used in operations that require *reduction* of values across the whole machine. The sum function is implemented by sequentially summing elements local to each processor and then globally summing the results of each processor using the control network.

The CM Fortran compiler tends to do less analysis beyond a single array operation than other Fortran compilers, which try to group array operations and execute them together to reduce memory accesses. This sort of optimization is especially crucial for cache-based RISC machines. Instead, the CM Fortran handles each array

[2] The array is actually padded out to $192 \times 96 \times 5$ unless the compiler flag -nopadding is used – the larger amount of padding is a historical artifact due to the desire to make iterations over array dimensions even multiples of the VU vector length of 8. The flag -nopadding refers to vector padding, not garbage padding as described above.

operation individually, using temporaries rather than attempting to gain performance through loop fusion and chaining vector registers. On the one hand, this simplifies the job of programming because we can estimate the speed of each individual line of code and determine the overall performance of a procedure by simply summing the times required by each line of code. On the other hand, even for an operation as simple as this inner product, the CM-5 performs the operation in two steps: by first multiplying X and Y and storing the result into a temporary array and then calling the sum library function with the temporary array as an argument. The time breakout for the reduction communication shown later therefore includes time spent summing the local parts of the array.

The scalar value b that is calculated by the reduction network is sent to the front end processor, which uses it for scalar computations. If a scalar value is needed for a parallel computation by the nodes, the front end must broadcast it to them using the control network. In general, the movement of scalar values between the front end and nodes are not expensive because the control network has fairly low latency.

3.2.3 Shifting array elements

Some computations in GCM require each element of an array to be calculated using values from neighboring elements of other arrays. In CM Fortran, this operation is implemented by *shifting* entire arrays by one element and then performing element-wise operations on the shifted arrays.

Shifting an array X to the north one direction can be expressed as follows:

```
Y = CSHIFT(X, +1, 3)
```

where +1 indicates shifting in the positive direction, and 3 indicates shifting of the third dimension of the array X. (Array dimensions are named $1, 2, 3$ respectively). Each value at location $(k, i, j + 1)$ in array X is now residing at location (k, i, j) in array Y. The values residing at the border wrap around to the other side in the obvious manner. The wrap-around behavior is correct for representing circular geometries such as the globe, but in cases where the geometry is not circular, a “ghost zone” of land is added around the entire array on each side. These ghost zones are part of the application code and may cause additional garbage padding to be introduced by the compiler, as discussed earlier.

The core of the 7-point stencil used for the matrix-vector multiplication $\mathbf{A}b$ can therefore be expressed as follows:

```
 Ab =
&     b               * AC
&   + CSHIFT(b,+1,1)  * AL
&   + CSHIFT(b,-1,1)  * AU
&   + CSHIFT(b,+1,2)  * AE
&   + CSHIFT(b,-1,2)  * AW
&   + CSHIFT(b,+1,3)  * AN
&   + CSHIFT(b,-1,3)  * AS
```

The “vector” b is actually a three-dimensional array, and $\mathbf{A}$ is represented only by its diagonals in the three-dimensional arrays AC, AL, AU, etc. Although this is a very non-intuitive way of performing matrix-vector multiplication, this stencil actually implements the multiplication using a data representation that fits the model, and using operations that map well to the CM-5. (The “&” character is used to continue the statement to the next textual line in the program.)

Note that AL, AE and AN represent the symmetric diagonals of AU, AW, and AS in the $\mathbf{A}$ matrix and are therefore identical arrays, shifted by one element. In the Id version, we use one array to represent both diagonals because we assume a shared memory model where accessing an element and its neighbor cost the same amount. For CM Fortran, if the machine configuration has enough memory, we keep separate versions of the diagonals to avoid additional CSHIFTs, because CSHIFTs can cause a large amount of additional communication.

Although the stencil computation does not require creation of full temporaries to hold shifted arrays, the CM Fortran compiler does not do analysis to determine that most of the elements required by a processor are already local to the machine, and accessible with some address calculation – rather, temporary shifted arrays are usually created because the vector units pay little penalty for the memory accesses.

3.2.4 Computational masks

Some calculations require different behavior at zones that have a land boundary than at zones that are completely surrounded by water. Whenever possible, we handle these cases by setting land or border zones to special “benign” values that make the array computation uniform for land zones and water zones. For some computations, this is not possible, and we must handle the borders differently and explicitly as shown by the following example.

The $\mathbf{A}$ matrix (consisting of the arrays AC, AN, etc.) used in the stencil computation is calculated once at the beginning of the program from data on the size of each zone and a few other physical constants. Zones for the different components of the $\mathbf{A}$ matrix that are on the border of water and land must be calculated in a slightly different manner than water zones that are completely surrounded by water. For example, to compute AC and AN, the central and the north diagonal of the **A** array respectively, we first compute these quantities as if all zones were water zones and were surrounded by water on all sides. Then, we fix the zones that are adjacent to land zones. In the following code, we make the correction when a water zone has a land zone as its northern neighbor. (WLMASK is an array that contains 1's on land zones, and 0's on water zones.)

```
  where ((CSHIFT(WLMASK,+1,2)-WLMASK)
&        .eq.-1)
    AC = AC + AN
    AN = 0.d0
  endwhere
```

The statements within the `where` block are evaluated only on the points where the predicate is true; on other points, values of the variables remain unchanged. The expression

```
(CSHIFT(WLMASK,+1,2)-WLMASK).eq.-1
```

describes those water zones that have a land zone as a northern neighbor.

This manner of handling conditional execution in the data parallel programming style is inefficient because only a tiny fraction of the elements of an array are on the border, but the entire array must be tested with the mask to determine which elements should be handled. Furthermore, each of the north, south, east, west, up and down borders must be handled separately. The CSHIFT is also expensive but, because the geometry of the ocean does not change significantly, the masks can be computed once and reused if there is sufficient memory to hold arrays determining the borders. For the GCM code, however, this sort of computational masking is used only during initialization, and is not a serious overhead.

3.2.5 Computing on array subsections

As discussed in Section 2, the preconditioning step of the PCG actually performs a forward-backward substitution using precalculated **L** and **U** arrays for each column of the ocean independently. Note that for our data layout, the preconditioning step does not require any communication between nodes because each column lies in the same node. The preconditioner and array layout were chosen explicitly for this reason; although we could use a different and more efficient preconditioner for the multithreaded version, we use the same preconditioner chosen for the data parallel version to give a better comparison between the two.

To exploit data parallelism in CM Fortran, the expression of the preconditioning step requires that all of the linear solvers for each column be executed in lock-step. That is, rather than expressing the computation in the most natural fashion, as $N_x \times N_y$ independent forward-backward substitutions, we perform the first step of the forward substitution on every element of the top layer of the ocean then continue with each layer of the ocean down to the bottom of the ocean.

The backward substitution happens in the reverse manner, starting from the bottom layer, and then substituting up for each layer to the top. In the following code, UD, MD and LD represent, respectively, the upper, middle and lower diagonals of the **L** and **U** factors of the **D** matrices, and `q(K,:,:)` refers to the horizontal slice of the array q at depth K of the geometry. (The **D** matrices represent the diagonal block elements of the **A** matrix.)

```
C Forward Substitution
  q(1,:,:) = MD(1,:,:)*r(1,:,:)
  DO K = 2, NZ
    q(K,:,:) = MD(K,:,:)*
&     (r(K,:,:)-LD(K,:,:)*q(K-1,:,:))
  ENDDO
C Backward Substitution
  DO K = NZ-1, 1, -1
    q(K,:,:) = q(K,:,:) -
&      UD(K,:,:)*q(K+1,:,:)
  ENDDO
```

Because the ocean has different depths at different places, the linear systems for each column of water may be of different sizes, but the data parallel programming model forces us to treat each column as if it were the same depth as the deepest column in the ocean, padding land zones representations with null equations.

3.2.6 Putting it together

With the pieces we have described so far, we can implement the preconditioned conjugate gradient solver, which has four basic operations: inner product, daxpy, 7-point stencil and the preconditioner. Figure 12 shows the PCG loop expressed in CM Fortran. For clarity, we have omitted array declarations and layout directives – all of the arrays in this loop have identical extents and layouts. Also, in the actual code, the procedure calls to `multiply_by_A` and `precondition` are written in-line, rather than implemented as separate procedures, in order to eliminate procedure call overhead.

Because PCG is the key inner loop for the entire application, we have taken care to eliminate expensive operations; there are no mask operations in this loop, and there is no use of the data network, except for the CSHIFT's in `multiply_by_A`. The `sum` reduction operations use the control network, which is relatively inexpensive.

As our analysis in Section 2.3 indicated, for extremely high-resolution problems, the vast majority of the time in GCM is spent in the PCG kernel, so the overall GCM

```
  DO WHILE (loopflag)
    gamma  = sum(r*xi)
    CALL multiply_by_A(b,Ab)
    alpha  = gamma / sum(b*Ab)
    p      = p+alpha*b
    r      = r-alpha*Ab
    CALL precondition(r,xi)
    beta   = sum(r*xi) / gamma
    b      = xi+beta*b
    if (sum(r*r).le.epsilon)
&     loopflag=.FALSE.
  ENDDO
```

Figure 12: The preconditioned conjugate gradient loop written in CM Fortran is very concise because of the array syntax. Note that the results of the two subroutine calls to `multiply_by_A` and `precondition` are returned through arrays passed to those routines, namely `Ab` and `xi`, respectively.

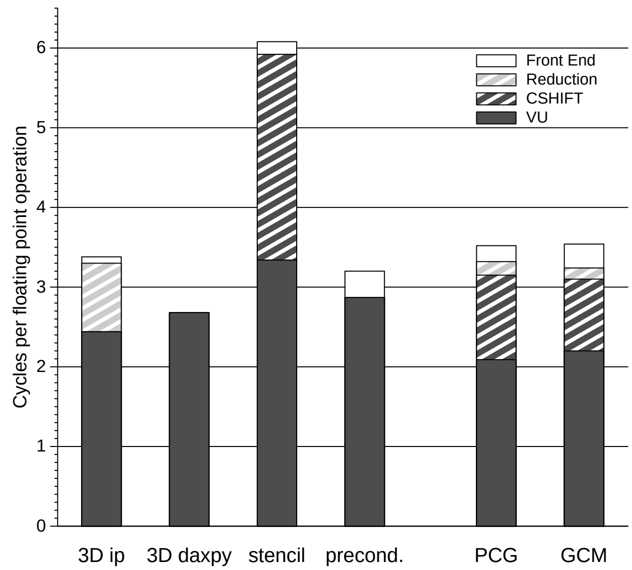

Figure 13: VU cycles spent per floating point operations on the CM-5, best case. Times are for a 32 node machine, running a problem size of $256 \times 256 \times 32$.

performance is almost completely dependent upon PCG. We should note, however, that the issues important for executing PCG efficiently are the same ones that come up in optimizing the rest of GCM.

3.3 Data parallel GCM performance

Figure 13 shows the cycles spent for each floating point operation for the constituent parts of the preconditioned conjugate gradient. This measure of performance allows one to factor out machine size, and we have chosen a problem size ($256 \times 256 \times 32$) that does not produce any garbage padding and is also large enough to factor out idling due to insufficient parallelism and vector startup.

These times were obtained from the most highly optimized versions of the PCG components. Note that in our calculation of overhead cycles for floating point operations here, it is assumed that every floating point operation executed by the CM-5 for these operations is necessary. In fact, in the GCM code, many of the operations counted could be occur over land and, therfore, are redundant.

Regardless of these caveats, the performance shown by the CM-5 and CM Fortran compiler is impressive; if is is assumed that each vector unit has a peak performance of 32 MFlop/s (assuming a multiply-add on every cycle) then GCM achieves approximately 1/6 of the peak performance. A more realistic assumption of 16 MFlop/s as peak would give GCM 1/3 of peak performance.

Although the compiler estimates for the inner loop from Figure 10 seemed to show that the daxpy computation would require only 1.75 cycles per floating point operation, there is additional overhead due to looping code surrounding the inner loop shown in Figure 10, which brings up the cycle count to a little more than 2.5 cycles. This same overhead is seen in each of the four operations.

The reduction operation in inner product takes a bit more than 0.5 cycles, which should be the minimum, and the CSHIFT operation in the 7-point stencil is fairly expensive, as expected. For the most part, the code executed by the front end was insubstantial or overlapped by parallel computation, except in the forward-backward substitution, where the looping control overhead was higher to handle the forward and backward substitutions in the preconditioner.

The distribution of time in the PCG is a weighted average of the distribution of time for the components of the PCG. The stencil is most heavily weighted because it accounts for 13 of the 32 total floating point operations in the PCG. An additional 8 floating point operations per zone come from the inner product, 6 from the daxpy, and 5 from the preconditioner.

We measured the performance of the entire GCM code by running the program for several hundred time steps to eliminate the effects of initialization of the system. For the cases we focus on here, the overall distribution of time in GCM reflects PCG, where most of the time in GCM is spent, and this is clear from the almost identical profiles for PCG and GCM.

Note that these timings are the absolute best case for the CM-5 because there is no garbage padding, the ghost zones overhead is negligible, and the problem size is sufficiently large to allow the VU's to work at full speed. Section 5, shows how to compare these best-case numbers against the performance statistics for the Id and Monsoon version.

4 Programming multithreaded GCM

The data parallel model of computation is ideally suited for an application like GCM, that uses large, regular data structures, and where most of the work is in element-wise operations over these data structures. For other, less regular applications, the data parallel model is too restrictive in the way that parallelism can be expressed. Multithreading is a more general model of parallel computation that allows different processors to work on different tasks at the same time in a less synchronous fashion. Consequently, multithreading has some overheads that result from more dynamic scheduling and synchronization. In the case of GCM and other highly structured codes, the multithreading overheads may be unnecessary because the applications may not require the expressiveness multithreading offers. One goal of this study was to quantify and characterize multithreading overhead and contrast it with some of the overheads of data parallel execution.

Figure 14 gives a high-level view of the multithreaded execution model. Each procedure may be associated with an activation frame, which is local to a processor; a procedure may fork off child procedures that may execute in parallel with the original procedure, either on the same processor or a different one. Similarly, the iterations of a loop may also execute in parallel. These procedures and loops may access data in a global heap of shared objects. Multithreading can be expressed in various languages and can be executed on different kinds of machines. The semantics of threads and styles of communication and synchronization between threads vary significantly, as can the typical length of threads, depending upon the language and architecture.

The programming language used to exploit multithreading may be explicitly parallel such as Cilk [10] or Mul-T [29] (allowing the programmer to annotate procedures or loops that should be executed in parallel), or implicitly parallel such as Id [34] (allowing the compiler to decide how to parallelize the code). Furthermore, the memory models for the languages can differ considerably. For example, most data structures in Id are based on I-structures, which are implicitly synchronizing; a fetch of an I-structure element will not return until the store to that element is executed.

The degree of hardware support for multithreading may also vary, as Cilk has been implemented on the CM-5, networks of workstations, and SMPs, none of which have special support for multithreading. Mul-T has been implemented on Alewife, which has support for scheduling and synchronization primarily at the basic block level, [2] and Id has been implemented on Monsoon [38], which has support for multithreading down to the instruction level. Monsoon also has direct support for I-structures. Other architectures that support instruction-level multithreading and I-structure-like memory include HEP [43], Sigma-1 [26], EM-4 [40], and Tera [3].

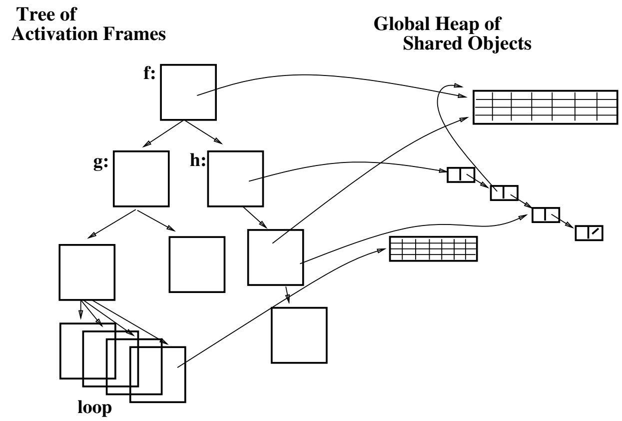

Figure 14: A high-level view of multithreaded execution; activation frames are associated with procedure invocations and parallel loop iterations while data structures reside in a shared global heap.

We have implemented GCM in Id on Monsoon, which is one extreme of multithreaded architectures. In doing so, we try to exploit parallelism at the instruction-level, procedure-level, and loop-level, incurring various execution overheads despite extensive hardware support for multithreading in Monsoon. We begin our discussion of the multithreaded implementation of GCM with a description of the Monsoon hardware and its execution model that is based on dataflow graphs. We then discuss how GCM was coded in Id. Next, we describe some practical problems in running GCM for realistic data sets on Monsoon. Finally, we describe optimizations that are performed by the compiler and programmer to obtain good performance on Monsoon and give some performance figures for the multithreaded implementation.

4.1 The Monsoon dataflow machine

Figure 15 shows a high-level view of the Monsoon multiprocessor. Monsoon [38] consists of eight 64-bit processing elements (PE's) and eight I-structure memory modules (or IS's) connected via a multi-stage packet switched network. Each PE is an eight-stage pipelined processor that handles dataflow *tokens*. On each processor cycle, a token enters the pipeline and its result is available eight cycles

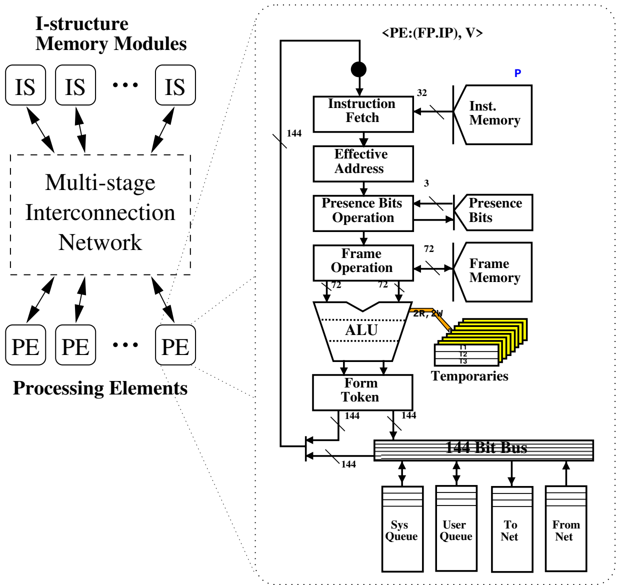

Figure 15: Monsoon consists of processing elements (PE's) connected to I-structure memory modules (IS's) through a multi-stage network. Each PE is an eight-stage pipelined dataflow processor.

later. Each token carries with it (1) a value, (2) a pointer to an instruction to execute, and (3) a pointer to a context, or activation frame, in which to execute the instruction. The execution of an instruction may cause the creation of up to two more tokens, which may be circulated locally or sent out on the network to other PE's. Each PE can process 10 million tokens per second and has 256K 64-bit words of local memory where activation frames are allocated.

Globally shared heap objects reside in the IS's, each of which contains 4M 64-bit words of memory. Both frame and heap memory have presence tags associated with each word to support fine-grain synchronization. Each IS memory access goes over the network. Monsoon's network has a bandwidth of 100 MB per second, and the network interface can accept or deliver a message every cycle. Messages are sent from PE's to IS's for heap references, or from PE to PE to communicate data between procedures or loops. All heap locations are equidistant to all the PE's. Heap objects are interleaved by hardware across the IS's and in general, data mapping has no effect on performance.

Every Monsoon instruction executes in one or two cycles. A binary instruction synchronizes on the arrival of its two inputs: the data from the first input token is stored in the activation frame and a *bubble* is passed in the pipeline, while the arrival of the second token causes the instruction to execute, producing more tokens. Unary instructions execute in one cycle without a bubble, as do binary

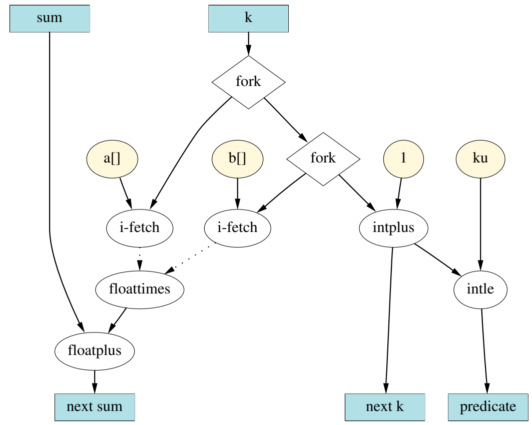

Figure 16: Monsoon dataflow graph for the body of the innermost loop of a three-dimensional inner product code.

instructions with one constant input. Global memory accesses are handled in a split-phase manner; a message is sent out to the network to the IS memory, and a result or acknowledgement comes back from the network at an arbitrary time later. During this time, Monsoon can process other tokens, using parallelism to mask the latency of the memory operation. Keeping an eight processor Monsoon busy requires at least 64-fold parallelism because each stage of the pipeline executes a different thread, and some additional parallelism to mask the latency of heap memory references.

To measure performance, each Monsoon processor has 16 cycle counters that can keep exact instruction counts (these counters were used for all measurements in this paper). We also have a cycle-level simulator, called MINT (Monsoon Interpreter), which is capable of running small programs. MINT has played a crucial role in microbenchmarking and enhancing our understanding of various factors that affect performance.

4.1.1 Dataflow graphs for parallel execution

Unlike conventional RISC processors, the machine language of Monsoon is *dataflow graphs*. Dataflow graphs specify only a partial order on instructions and thus implicitly represent instruction-level parallelism; in contrast, conventional superscalar RISC processors dynamically detect opportunities for instruction-level parallelism within a linear instruction stream. To illustrate several features of the Monsoon execution model, consider the innermost loop of the 3D inner product:

$$\sum_{\texttt{k=kl}}^{\texttt{ku}} \texttt{a[k,j,i]*b[k,j,i]}$$

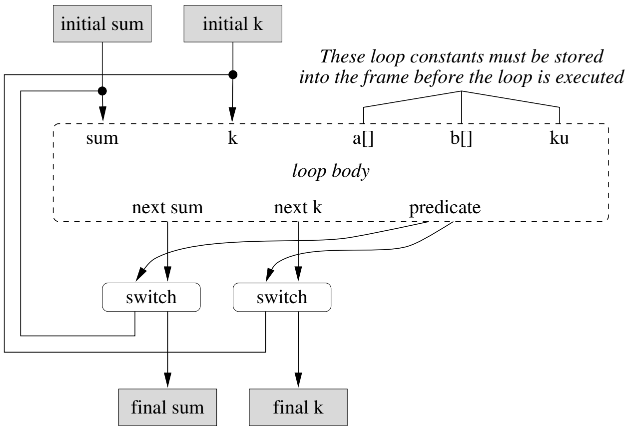

Figure 17: Sequential loop implementation of the inner loop of 3D inner product. The sequential loop executes on one PE, and only exploits instruction-level parallelism.

Assuming the arrays are stored in row major order, Figure 16 shows the Monsoon dataflow graph for the body of the loop, including the loop predicate.

The nodes of a dataflow graph represent instructions, and dataflow tokens flow along the arcs to pass values generated by parent instructions to their children. Because Monsoon instructions can send a value to at most two destination instructions, the *fork* operator is required when a value values is needed by more than two destination instructions. Because of instruction encoding limitations, some instructions, can send a value to only one destination instruction. The arcs leading from the two *i-fetch* instructions to the *floattimes* instruction are drawn as dotted arcs to emphasize that I-structure references are split-phase.

Literals and loop constants are represented in light gray in Figure 16. Literals are stored into local memory at load-time by the loader, while loop constants must be stored into local frame memory at run-time using the *constant-store* instruction. Both types of constants can be accessed at run-time without incurring a bubble in the pipeline.

Control in Monsoon is implemented by using the *switch* operator that steers an input value to one of two locations, depending upon the value of an input predicate. Figure 17 shows the "sequential" loop schema for the loop body shown in Figure 16. Before the first iteration of the loop begins, loop constants are stored into the activation frame. The initial values for the input variables are fed to the top of the loop body, and then the loop body in Figure 16 is executed. The resulting values are either steered back to the top of the loop, or else output as the final values. The values are steered using *switch* instructions, with one *switch* for each induction variable. If there are more than two induction variables, forks are needed to distribute the predicate value to the switch's. Because the switch instruction can steer a value to only one destination instruction, additional forks are often required after a switch.

Sequential loops only execute on one processor, and generally use the activation frame of the surrounding procedure. In the next section, we show how inter-processor parallelism is exploited in Monsoon for both procedure-level parallelism and loop-level parallelism.

4.1.2 Run-time system and inter-processor parallelism

In Monsoon, dataflow-style instruction-level parallelism is exploited only within a PE, to keep the eight pipeline stages busy. Inter-processor parallelism is exploited by allocating activation frames on remote PE's. Activation frames are allocated when a procedure is called or a parallel loop is initiated, and the frame may be allocated on an arbitrary Monsoon PE, although frames do not migrate once they are allocated. Thus, the mapping of work to processors is governed by the frame allocation policy, which is a part of the Monsoon run-time system (RTS).

In the Monsoon RTS [13], 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 to provide load balancing. Since the RTS is invoked on every procedure and parallel loop invocation, care has been taken to minimize RTS costs, to the point where much of the RTS is written in assembly and supported by special "microcoded" machine instructions. Despite all this care, allocating and deallocating a frame on Monsoon typically takes about 40 to 50 cycles. This shows that 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.

A procedure call begins with allocating an activation frame. Once the frame is returned by the RTS, the arguments to the procedure and the return continuation are sent as messages to the frame, which is usually on a remote processor. Although the composition, sending and receiving of these messages are well supported in Monsoon hardware, argument-passing is still relatively more expensive than register-style argument passing in sequential processors. In general, the programmer should define most small leaf procedures as inlinable to avoid procedure call overhead.

Loops can trivially exploit inter-processor parallelism if they are implemented as recursive procedure calls. However, such an implementation usually uses too many frames and consequently has a very high overhead both in terms of storage and RTS calls. Id on Monsoon exploits an alternative compiling scheme known as k-bounded loops [14].

Cycles/iteration of the 3D inner product inner loop			
Opcode	completely unrolled	sequential	k-bounded
fork	2	3	6
intplus	1	1	1
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	8 (1.5)	14 (3.5)	26 (6)

Figure 18: Comparison of cycles per iteration for unrolled, sequential and k-bounded loop implementations. The number of bubbles incurred by each operation per iteration are set off in parentheses.

The k-bounded loop schema employs a ring of k activation frames that are allocated and linked at the beginning of the loop execution and deallocated at the end of the loop execution. Loop constants must be stored into each frame in a k-bounded loop, and induction variables are passed from iteration to iteration like arguments are passed for procedure calls.

Exploitation of inter-processor parallelism can be expensive even on Monsoon. In Figure 18, the cycle counts for k-bounded, sequential, and completely unrolled implementations of the innermost loop of the 3D inner product are shown. Recall that the sequential loop schema only exploits intra-processor parallelism and a loop can be completely unrolled only if the loop bounds are known at compile time. It takes three times as many cycles per iteration for the k-bounded loop as the completely unrolled loop!

We explain these cycle counts further to satisfy the reader's curiosity. The completely unrolled implementation reflects the instructions in Figure 16 excluding the instructions needed to evaluate the loop predicate. The count for the sequential implementation can be derived by adding the predicate and the two switches shown in Figure 17. The k-bounded implementation uses two message instructions to send the induction variables to the next iteration. The i-take and i-put instructions are used as semaphores to synchronize between loop iterations to determine when a frame may be reused for the next iteration. The sync instructions gather signals from work being performed in the current iteration to ensure that all work is completed in the current iteration. Additional fork instructions are used to distribute values to the new instructions.

In addition to the per iteration costs shown in Figure 18, loops have startup and shutdown costs that may be significant, depending upon the loop schema. Completely unrolled loops have no startup cost, whereas sequential loops must store loop constants into the frame before beginning loop iterations. The startup and shutdown costs of k-bounded loops can be from about 100 to several hundred cycles per k, to allocate and deallocate frames, to set up loop constants and to wire up the frames in a cycle.

Given the widely varying costs of loop execution, even for the same source program, it is clear that the programmer or compiler must make decisions to expose loop parallelism while minimizing instruction counts. We will discuss these issues further in Section 4.2.5.

4.2 Programming in Id

Id is a layered language whose core consists of a higher-order, statically typed, polymorphic, non-strict functional language. Id also has mutable and synchronizing data structures called I-structures and M-structures. The GCM code is written using only the functional subset of Id, but the Id compiler transforms functional data structures into I-structures for execution on Monsoon. I-structures are implicitly synchronizing, so that a fetch of an I-structure element will not return until the store to that element is executed. I-structure elements can be written only once (although they can be read any number of times) to ensure that programs are deterministic.

4.2.1 Array Abstractions

Id has only three built-in operations on arrays; fetch, store and fetch array extents. However, it is possible to define powerful array abstractions in Id and use them repeatedly. Before defining `make_3D_array`, an abstraction to create a three dimensional array, we show how it is used:

```
x = make_3D_array extents f
```

where `extents` defines the extents of the array and `f` is a *filling* function. The net effect is that `x[index] = f index`, for each index `(k,j,i)` that lies within the `extents`. By giving a suitable definition for `f` we can define each GCM state variable. Note that `x` is *immutable* in the sense that once it has been defined it cannot be modified. Consequently, at each time step a new array for each state variable is created and the old is implicitly discarded.

An (inlinable) function definition of 3D daxpy can be expressed as:

```
Zones =
    ((min_W_longitude, max_E_longitude),
     (min_N_parallel , max_S_parallel ),
     (surface        , max_depth      ));

defsubst aX_plus_Y a X Y zone =
  a * X[zone] + Y[zone];

defsubst daxpy_3d a X Y =
  make_3D_array Zones (aX_plus_Y a X Y);
```

where `zone` represents an index and `Zones` represents the dimensions of the box containing the ocean. One can think of `aX_plus_Y` as a inlinable function that takes four arguments and produces a number. When only three arguments are passed to it, as shown above, a new function is returned that takes one argument `zone` (an index) and returns a number.

The `make_3D_array` function is not a built-in operator in Id: it is defined as a library function using *array comprehension* syntax as follows:

```
defsubst make_3D_array extents f =
  { ((kl,ku),(jl,ju),(il,iu)) = extents;
    in
   {3D_array extents of
      | [index] = f index
                  || i <- il to iu;
                     j <- jl to ju;
                     k <- kl to ku;
                     index = (k,j,i) }};
```

The initial binding destructures the `extents` argument into its component parts. The phrase between "|" and "||", that is "`[index] = f index`", specifies an index and an expression to be filled in the indexed slot of the array. The phrases such as "`i <- il to iu`" that occur after "||" are called *generators* and define a way to enumerate the indices.

All of the abstraction involved in the array comprehension, destructuring and higher-order function call could be extremely expensive, but if the programmer is careful to declare the filling function and array constructor as inlinable, all of the overhead is compiled down into an I-structure array allocation and a triply-nested loop that fills the I-structure using the inlined filling function.

This level of abstraction may seem to be overkill for a simple operation like daxpy, but it is also used at every level of the code, simplifying the specification of the algorithm greatly and also allowing for easier modification to the computation by changing of the underlying abstraction.

Notice that, in the Id array constructor abstraction, we ordered of the axes as we did in the CM Fortran code, although for a quite different reason. In Id, we want the innermost loop to iterate over the vertical axis of the state arrays so that we can avoid computation on land, and also because we want to unroll the loop completely. Given that the innermost loop iterates over the vertical axis, we also want that axis to be stride-1, so that we can avoid overheads from additional support instructions to calculate memory addresses.

4.2.2 Avoiding unnecessary computation

In the elliptic solver, we compute four inner products involving state variables. Because the land zones do not contribute to the inner product, we could optimize this computation by only iterating over water zones as shown below:

```
defsubst ip_3d a b =
  { s = 0.0 ;
   in
    {for i <- min_W_longitude to
               max_E_longitude do
      s_jk = 0.0;
      next s = s +
       {for j <- northmost_parallel[i] to
                  southmost_parallel[i] do
         s_k = 0.0;
         next s_jk = s_jk +
          {for k <- surface to
                     bottom[i,j] do
             next s_k =
              s_k + a[k,j,i]*b[k,j,i];
             finally s_k};
         finally s_jk};
      finally s} };
```

where `bottom[i,j]` is the depth of the ocean at `(i,j)`. The variable `surface` has been defined as a constant instead of a 2D array without loss of generality. `northmost_parallel[i]` and `southmost_parallel[i]` represent, respectively, the northern most and southern most parallel at longitude `i` where a water zone may be found. A note on syntax – the `next` qualifier is used to express recurrences in `for` loops – "`next s`" specifies the value the variable `s` will assume in the next iteration. The `finally` qualifier is used to describe the result of a `for` loop.

We can further reduce the computation by defining some of the state variables only on water zones, if their values are not needed on land zones. The `make_water_zone_array` abstraction defined below enumerates only water zones:

```
defsubst make_water_zone_array f =
 {3D_array Zones of
    | [zone] = f zone
      || i <- min_W_longitude to
               max_E_longitude;
         j <- northmost_parallel[i] to
               southmost_parallel[i];
         k <- surface to bottom[i,j];
         zone = (k,j,i) }
```

Consequently the following daxpy would do fewer floating point operations than the one defined in Section 4.2.1.

```
defsubst daxpy_w_3d a X Y =
  make_water_zone_array (aX_plus_Y a X Y);
```

4.2.3 Avoiding deadlocks due to undefined elements

Sometimes an attempt to avoid unnecessary computations can lead to deadlocks in the Id code. Consider the stencil computation in the PCG (as discussed in the data parallel section, the diagonals of matrix A are symmetric; so only four arrays need to be stored instead of seven):

```
defsubst seven_pt_stencil b =
  make_water_zone_array (multiply_by_A b);

defsubst multiply_by_A b zone =
  { C = A_C[zone] * b[zone];
    W = A_WE[Wfz zone] * b[Wzz zone];
    E = A_WE[Efz zone] * b[Ezz zone];
    N = A_NS[Nfz zone] * b[Nzz zone];
    S = A_NS[Sfz zone] * b[Szz zone];
    U = A_UL[Ufz zone] * b[Uzz zone];
    L = A_UL[Lfz zone] * b[Lzz zone];
   in
     C + W + E + N + S + U + L }};
```

where the function `Nzz` computes the zone to the North of a zone, while `Nfz` computes the north face of a zone:

```
defsubst Wzz (k,j,i) = k,j,i-1 ;
defsubst Ezz (k,j,i) = k,j,i+1 ;
...
defsubst Wfz (k,j,i) = k,j,i-1 ;
defsubst Efz (k,j,i) = k,j,i ;
...
```

The `multiply_by_A` filling function will attempt to read elements of the array `b` that may correspond to land zones, and thus may have undefined values. An attempt to read an undefined array element in Id will cause a deadlock because the I-structure fetch will never return. In the CM Fortran version, the ocean is padded with a layer of land to handle this boundary condition around the edges, and the other land zones are filled with benign values. The same strategy can be used in the Id version, however, the land padding cannot simply be initialized once and re-used at every PCG iteration. Rather, the padding must be filled every time because a new array is defined for the variable on every iteration of the solver.

Note that the land padding is not necessary for the non-stencil operations in the PCG, so we only pad array `b`, which is needed by the stencil computation. To accomplish this, we define a different array constructor that fills zeros in all the land zones and use it to redefine `daxpy_3d` of Section 4.2.1. In the following code, `OG` represents the ocean geometry:

```
def make_zone_array' f =
  {3D_array Zones of
     | [zone] = {case OG[zone] of
                  | water = f zone
                  | land  = 0.0}
       || i <- min_W_longitude to
               max_E_longitude;
          j <- min_N_parallel  to
               max_S_parallel;
          k <- surface to max_depth;
          zone = (k,j,i) };

defsubst daxpy_3d' a X Y =
  make_zone_array' (aX_plus_Y a X Y);
```

4.2.4 Putting it all together

An important step in the elliptic solver is the multiplication by the preconditioner $\mathbf{K}$, which is actually implemented as $N_x \times N_y$ independent tridiagonal linear equation solvers of size $N_z \times N_z$ as described in Section 2.2.2. As in the CM Fortran code, each of the $N_x \times N_y$ independent linear systems is initially factored into $\mathbf{L}$ and $\mathbf{U}$ matrices corresponding to the decomposition of the block diagonal elements of $\mathbf{A}$, and these matrices are stored into a constant matrix, which are referenced in the call to `solve_tridiag_LU`.

```
defsubst precondition r =
 {3D_array Zones of
   | [zone] = X[k]
     || i <- min_W_longitude to
             max_E_longitude;
        j <- northmost_parallel[i] to
             southmost_parallel[i];
        X = solve_tridiag_LU r (i,j);
        k <- surface to bottom[i,j];
        zone = (k,j,i) };
```

Given all these functions, it is straightforward to write PCG in Id as shown in Figure 19. There are no other issues that arise in coding the whole GCM code; each filling function is either some sort of a stencil, or involves purely local computation.

After compilation, if all the functions have been declared as substitutable, the resulting code turns out to be nothing but a nest of nested loops. Most of these loops and the individual bindings in them can proceed in parallel, as soon as the data dependencies for the binding are satisfied. Note that because of the element-wise synchronization of I-structures, parts of state arrays can be filled as soon as the parts of the arrays they depend on are filled.

```
{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_w_3d alpha p b;
   next r  = daxpy_w_3d (-alpha) r Ab;
   next xi = precondition (next r);
   beta    = (ip_3d (next r) (next xi)) /
                gamma;
   next b  = daxpy_3d' beta (next xi) b;
 finally p}
```

Figure 19: The PCG in Id requires the definition of new state arrays on every iteration. The order of execution is not necessarily in the textual order – all of the bindings proceed implicitly in parallel and must only wait for data dependencies to be satisfied.

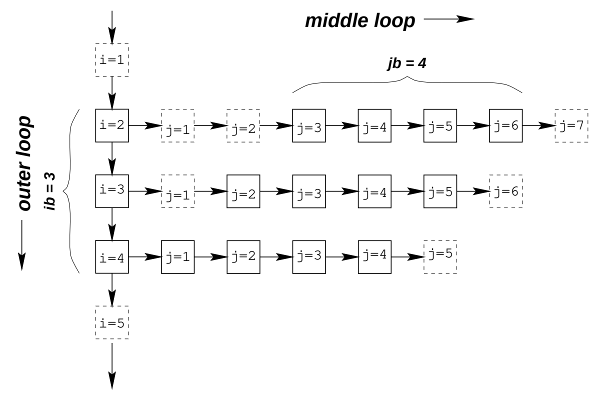

Figure 20: Pattern of frame usage for nested k-bounded loops for the 3D inner product.

4.2.5 Programmer annotations and restructuring

Once the functionality of the code has been implemented, the programmer may have to annotate his code to get good performance. Standard loop optimizations such as unrolling, peeling, strip-mining, and interchange can have a larger impact on Monsoon performance than for conventional architectures because of the overhead of parallel asynchronous execution of loop iterations in the dataflow model. The Id compiler automatically performs some of these optimizations, and the others are left to the programmer to be indicated as pragmas. We discuss the role of loop pragmas here.

Unless otherwise annotated by the programmer, all loops are assumed by the compiler to be sequential loops. Constructs such as array comprehensions desugar into loops, and in general, annotations that are used for loops may also be used for array comprehensions. It is a good rule of thumb to annotate *outermost* loops with parallelism to be k-bounded loops. (It makes no sense to declare the time-step loop in GCM to be k bounded because there is not much parallelism between time-steps.) In this way, large chunks of work are forked to remote processors, and the overheads of k-bounded loop iterations are paid less often. Inner loops should be annotated to be unrolled to do n iterations at a time to reduce sequential loop overhead. If the number of iterations is a constant and small, then the pragma to unroll the loop completely should be used.

Suppose that the programmer has annotated the outer and middle loops of a triply-nested loop to be k-bounded, with the k-bounds being *ib* and *jb*, respectively. In general k-bounds can be arbitrary expressions that return an integer; in practice, however, they are usually literals. Figure 20 shows the pattern of frame usage for the two loops when *ib* is 3 and *jb* is 4.

The amount of interprocessor parallelism exposed for such a nested loop 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. This optimization has been implemented in the Id compiler, but has not worked reliably.

Large values of k may not be useful in exposing parallelism 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 [14], but to date, no automatic compiler-directed policy has been implemented that achieves high performance.

Some optimizations cannot be expressed simply with pragmas and require restructuring of code. For example, the precondition routine in Section 4.2.4 requires allocating a temporary array `X` for each column `(i,j)` of the ocean for each iteration of PCG. The only purpose of this array is to hold the values returned by the tridiagonal solver before it is copied into the state array. This array is eliminated by making the tridiagonal solver store directly into array `xi`. Such a program can be expressed in Id by writing the preconditioner using loops and I-structures rather than array comprehensions. A subtle issue is if the ocean has a known uniform depth, then it is also possible to eliminate another array temporary in the tridiagonal solver.

4.3 Running GCM on Monsoon hardware

Monsoon was designed at MIT and built by Motorola under a research collaboration agreement. We received the first Monsoon with one PE and one I S in November 1990. Two machines, each with 8 PE's and 8 IS's, were delivered to MIT and Los Alamos National Labs (LANL) in the latter half of 1991. Several more machines with 2 PE's and 2 IS's were built and placed at various research institutions across the country. Detailed performance studies of Monsoon were conducted in 1991-92 (see, for example, [22, 5, 46]). During this period, both the Id compiler and run-time system were being tuned continually for Monsoon.

By 1993, when the GCM study was begun, the team that had built Monsoon at Motorola had disbanded; the machine at MIT was maintained by cannibalizing parts from other Monsoons. At this point, an initial unoptimized version of a fully functioning multithreaded GCM ran on an 8 PE 8 IS Monsoon approximately 100 times slower than the data parallel version on a 32-node (128 VU) CM-5. Given the speed difference, experiments took many hours to run on Monsoon, and checkpointing was implemented because the Monsoon hardware had intermittent failures. We started doing most of the experiments on PCG and 2 PE 2 IS systems to avoid some of these problems. Our experience with other applications on Monsoon [22] have shown that 2 PE 2 IS performance translates directly into 8 PE 8 IS performance.

As is often the case in any such study, both implementations were continually being tuned based on observed performance. However, algorithmic innovations were always guided by the concerns for greater efficiency on the CM-5, as this was a production code. It was a difficult task to keep the two implementations consistent for this study, especially because gathering statistics on a 4 to 24 hour run on Monsoon was tedious. A major milestone in this project was reached in early 1994 when we were able to produce the same numerical results on both implementations for several large ocean geometries.

The hardware maintenance issue became more serious in 1995 because key personnel at MIT were concentrating on building new machines, and the condition of Monsoon had deteriorated. Most of the experiments from 1995 onwards had to be performed 2 PE 2 IS machine. No new experiments have been conducted on Monsoon since mid-1996. Monsoon was decommissioned and donated to the Computer Museum in February 1997, and the CM-5 was decommissioned in July 1997.

Although we also have a robust cycle-level Monsoon simulator, even a 2 PE 2 IS Monsoon is many thousand times faster. The Monsoon simulator was used extensively for micro-benchmarking, and can run several iterations of PCG with small data sets in a reasonable amount of time. However, running realistic data sets with the entire GCM code would be impossible on the simulator, whereas it is feasible on even a 2 PE 2 IS Monsoon.

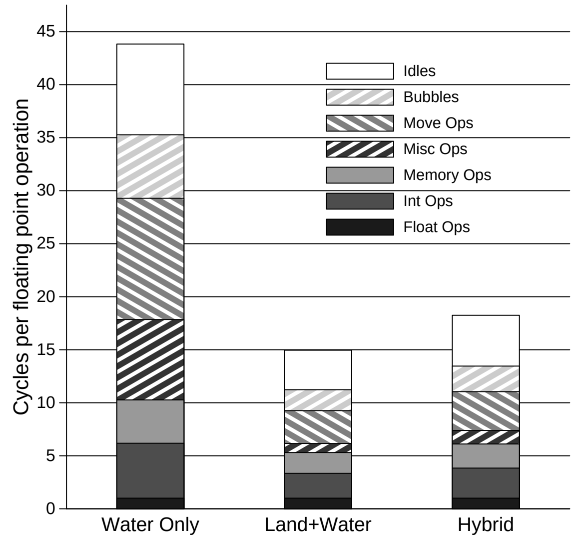

Figure 21: Comparison of the three versions of PCG for a $171 \times 93 \times 5$ geometry, 60% water. ($ib = 5$, $jb = 3$, 2 PE, 2 IS Monsoon)

4.4 Multithreaded GCM performance

To factor out machine size and clock rates, as in the data parallel case, we measure the efficiency of the multithreaded implementation of GCM by determining the number of cycles spent for each floating point operation. The "Water Only" bar in Figure 21 shows the breakdown of the overhead operations executed in the PCG code for a particular geometry, a $171 \times 93 \times 5$ ocean with 60% water.

It is clear from this graph that the vast majority of the time is spent in overhead cycles. "Move" instructions include forks and syncs, "Misc" instructions include switches and message-passing instructions, "Memory" instructions include i-fetches and i-stores, and "Int" instructions include ordinary arithmetic and logical instructions. Actual floating point operations consist of a tiny fraction of overall time.

4.4.1 Decreasing overhead cycles

To reduce overhead cycles, we attempted to simplify the computation by making it identical to the data parallel implementation. In doing so, we were able to completely unroll the inner loops of each of the PCG components. Complete loop unrolling is beneficial to 3D daxpy and 3D inner product because the loop overheads are high relative to the small loop bodies for these PCG components. It does not help the 7-point stencil computation as much because of the larger loop body of the stencil. The preconditioner is significantly simplified because of the elimination of a temporary array.

I-fetches for the north-south and bottom bounds of the ocean were eliminated, along with the calculation of the addresses for those i-fetches. By making the ocean bounds constant, some loop constants can be eliminated by making them literals. These simplifications reduce the number of cycles per floating point operation from about 35 in the "Water Only" case to about 12 in the "Land+Water" case in Figure 21.

Note that the "Idle" cycle count is misleading in that the k-bounds for the loops are set in the context of the entire application, not for the PCG executing in isolation. Those idle cycles will be overlapped by other work when PCG executes within the entire GCM code.

The reduction in overhead cycles per floating point operation comes from fewer switches and messages for simpler loops, and from fewer bubbles and forks incurred by eliminated switches and messages. Address calculation is also simpler, as evidenced by fewer "Int" instructions, and some fewer memory operations are executed because of the constant ocean bounds.

It is surprising to discover the degree to which the "Land+Water" version was more efficient than the "Water Only" version. Part of the explanation is in the very shallow geometry used; because the innermost loop executes at most four times in the "Water Only" case, it is difficult to take advantage of regular loop unrolling (as opposed to complete loop unrolling) to reduce loop overheads. For geometries with more layers, the difference will not be so marked, although the completely unrolled preconditioner would remain an advantage of the "Water+Land" version.

4.4.2 Hybrid PCG

Although the "Land+Water" version is much more efficient per floating point operation executed than the "Water Only" version, it must execute more floating point operations because of land zones. Despite this, the overheads from land are more than made up for by the efficiency of the "Land+Water" version for most geometries; however, we would still like to take advantage of some of the geometry irregularity that result from land zones, while also using a simplified loop structure.

To do this, a *hybrid* version of the PCG was developed which 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, much of the computation on land is eliminated, while still allowing complete unrolling of the innermost loop. Figure 21 shows that this version has more overhead per floating point operation than the "Land+Water" version, but less than the "Water Only" version.

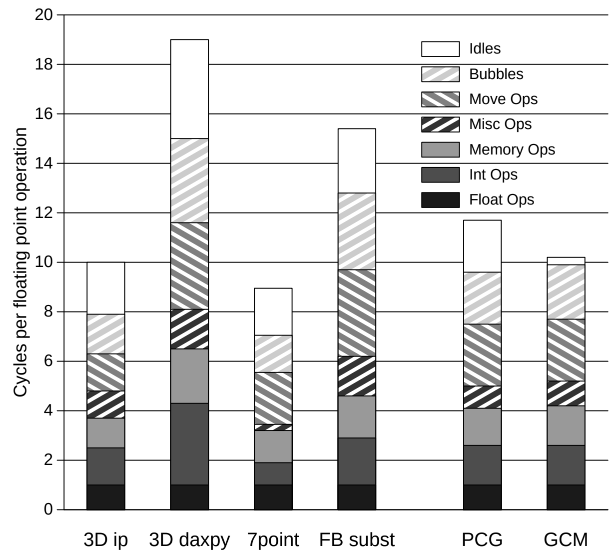

Figure 22: The best case performance for the components of the hybrid PCG, the PCG itself, and the whole GCM ocean modeling code for a $32 \times 32 \times 32$ ocean with 100% water. ($ib = 5$, $jb = 3$, 2 PE, 2 IS Monsoon).

The hybrid approach is only superior to the Land+Water approach in situations where it can take advantage of not having to compute on land. Typically, geometries that we are interested in simulating have between 30% to 70% of their surface covered with water, so the hybrid approach actually does pay off most of the time.

4.4.3 PCG component performance and GCM performance

Figure 22 shows the best case performance in terms of 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. In this case, the same criteria as the data parallel case is used; it is assumed that all floating point operations executed are required operations, without taking into account land.

The problem size we use is proportionately smaller than the CM-5 version because the Monsoon machine configuration is smaller. Note that the Id program is the same for a 2 PE Monsoon as for an 8 PE Monsoon, except for

k-bounds for loops.

We optimized the k-bounds shown in Figure 21 and Figure 22 for the entire GCM computation. By doing so, when we extract portions of the code such as the PCG computation or components of the PCG, additional idle cycles appear because there is less work that can be overlapped to eliminate idle cycles. These idle cycles are misleading, because they could be eliminated by increasing k-bounds, but doing so could distort other numbers in the overhead accounting.

It is interesting to compare the performance of the 3D inner product in Figure 22 with the detailed cycle count for the completely unrolled inner loop in Figure 18. In Figure 18, eight cycles are spent for two floating point operations, yielding a ratio of four cycles/floating point operation. In Figure 22, the full triply-nested loop shows a ratio of about eight cycles per floating point operation, indicating that much of the overhead arises from the outer two loops of the inner product. This is similar to the CM-5, where the detailed cycle count for the innermost loop was much better than the actual measured time of the entire 3D inner product.

Compared to the best case numbers for the CM-5, the Monsoon numbers show that approximately three times as many cycles are necessary for every floating point operation. However, our choice of performance measure showcases the best aspects of data parallel computing. In the next section, we attempt to adjust these raw "best case" numbers for situations that are more realistic.

5 Performance comparison

Both GCM implementations use the same numerical algorithm, except for minor differences such as the order of reductions. Despite this, comparing the two implementations requires care because of differences between the languages, compilers, programming models, architectures, implementation technologies, machine configurations, and manpower expended on software and hardware. In this section, we try to account for some of these differences and determine the primary characteristics and overheads of the two implementations.

5.1 Performance metric

Because both implementations have used the same mathematical algorithm, the numerical results and the number of iterations for the conjugate gradient to converge in each time step are identical. Consequently, both codes execute the same number of required floating point operations, though the number and type of overhead instructions executed differ greatly. We consider floating point operations performed on land zones or land-land surfaces to be non-essential.

We attempt to factor out most of the differences between the implementations using the simple metric of cycles per *required* floating point operation. Up to this point, we have been measuring cycles per floating point operation, which includes any extra floating point operations which are used for computations on land.

The cycles per required floating point metric factors out many of the differences between the implementations; it does not, however, quantify all the dimensions that affect performance. For instance, one of the architectures may be easier to implement at a high clock speed than the other or more cheaply than the other; this metric also does not take into account the much greater manpower spent in optimizing the CM-5 performance. It would be misleading for us to try and rigorously quantify these factors, and we leave it to the judgement of the reader to adjust our measurements as they feel suitable. Nevertheless, as explained below, using this metric (while understanding its limitations) does yield significant insight into the efficiency with which GCM can be mapped to the two contrasting computational environments.

The cycles per required floating point operation metric is clearly better than the cycles per every floating operation measure we have been using to this point for both the CM-5 and Monsoon implementations. For example, in Figure 21, the "Land+Water" version is more efficient per floating point operation, but many of the floating point operations that are considered are actually overhead operations because they are executed on land zones. When taken into consideration, the ratio of cycles per required floating point for the "Water Only" versus the "Land+Water" versus the "Hybrid" implementations of multithreaded PCG is actually about 1.0 : .44 : .35, showing that the hybrid version is the fastest.

5.2 Quantifying overheads

Monsoon and CM-5 can be viewed as radically different attempts at feeding relatively standard FPU's, though this was certainly not a design goal of Monsoon. Taken in this context, the "best case" numbers clearly indicate that the CM-5 is much better when the situation is ideal – i.e., when there is no garbage padding, ghost zones are not taken into account and the problem size is large enough to overcome vector startup costs. The rest of this section quantifies further what the effects of the various overheads of each model are.

5.2.1 Overheads due to padding and ghost zones

The overhead from garbage padding on the CM Fortran verion is usually not more than 10%; the compiler can usu-

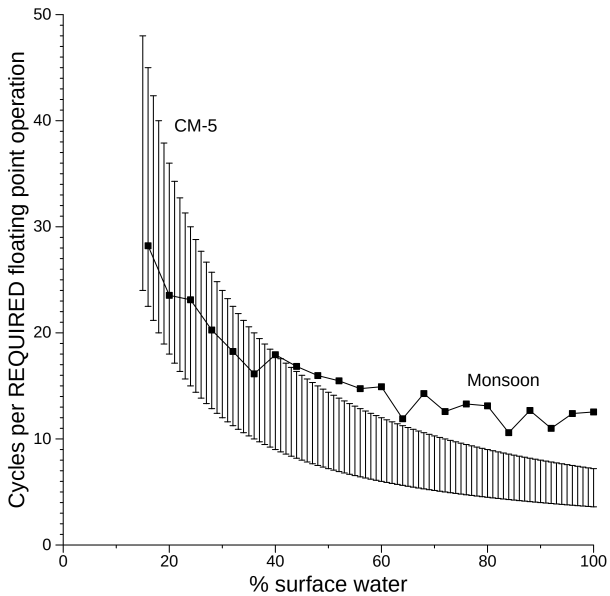

Figure 23: The CM-5 version of PCG is more sensitive to variations in the percent of water than the hybrid Monsoon version.

ally find a layout which is fairly close to the one requested by the programmer. Because of the hardware support for ignoring garbage padding, the CM-5 does not pay any additional cost in terms of execution for garbage padding, above and beyond the direct overhead due to unused zones.

Ghost zones are a more serious concern for geometries that are shallow or narrow. The Id version does not have to work on the faces of the geometry, except to explicitly insert ghost zones on the step before the stencil computation, and even the stencil computation itself does not need to compute on ghost zones. Although the Id version also must pay for the memory, it does not have to pay for the execution time for filling ghost zones.

It is clear that ghost zones are not a serious overhead for wide, deep geometries. However, we often work with geometries that are of depth 5 to 10, in which case overhead for ghost zones can range from 20% to 40% or more.

5.2.2 Overheads due to water

Figure 23 shows the effect of varying the percentage of water zones on the surface for the Id and CM Fortran codes. Water zones on the surface are counted, because the hybrid version of the Id code can only take advantage of eliminating an entire column of zones at a time.

The number of "required" floating point operations was calculated as all floating point operations which are performed in columns of the geometry which have a water surface zone. This is not the actual number of required floating point operations because floating point operations on land zones are still being executed in the hybrid version of the Id code. However, given the number of surface water zones, the number of actual required floating point operations may vary widely, and the purpose of this graph is to compare the Id and CM Fortran versions against each other, not against an ideal. For these purposes, this measurement is fair, because any difference between the required floating point operations in this graph and the actual number of required floating point operations would affect the performance of both versions proportionately.

The performance for the CM-5 version was calculated by assuming an ideal performance of 3.5 cycles per floating point operation when the geometry is 100% water, and then extrapolating backwards by assuming that the execution time will be identical regardless of the percentage of water zones. We give a zone of performance for the CM-5, because the ideal performance may vary widely depending upon the padding overhead and the overhead due to executing a smaller problem size. The upper end of the zone assumes performance at 100% water of seven cycles per floating point operation; this number is not necessarily the upper bound of the performance of the CM-5 version, and may in fact be worse for some problems as seen in the next section.

The performance of the Id version was measured on a large geometry, varying the percentage of water zones on the surface. The geometry was of size $50 \times 50 \times 10$, running on a two processor system, and is a large enough problem size to keep the machine busy. We varied the percentage of surface water zones and measured the total number of cycles as a ratio of required floating point operations.

The performance of both versions deteriorates as the percentage of water decreases, but as exepected the CM-5 version shows more severe degradation, because it carries out the same amount of computation regardless of the geometry, whereas the Id version does less work for the geometries with less water. Because many real geometries of interest consist of 30% to 70% water, this overhead in conjunction with others may make the performance of the CM Fortran version more comparable with that of the Id version.

5.2.3 Overheads due to small problem size

Both machines will have worse performance for smaller problem sizes than for larger problem sizes because of less parallelism to keep the machines busy. Figure 24 shows the effect of varying the problem size, measured in elements of the geometry per processor. To obtain this graph, the time needed to perform PCG for various problem sizes is measured on both Monsoon and the CM-5; the hybrid Monsoon version is employed, only geometries having

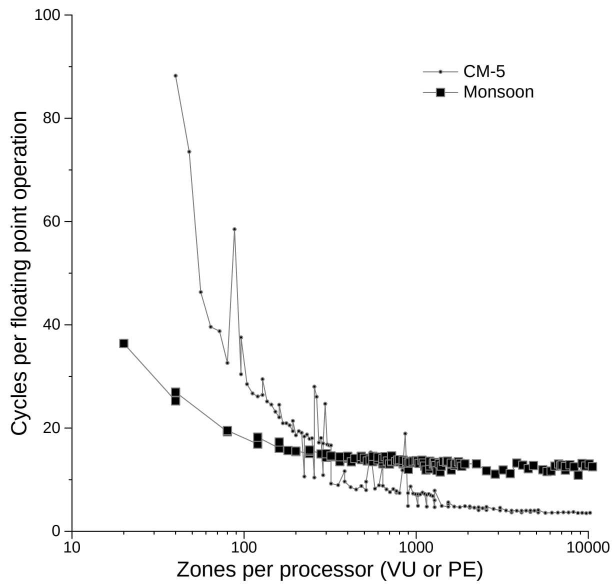

Figure 24: Efficiency versus problem size for the PCG computation; the CM-5 is more sensitive to problem size than Monsoon. This difference is primarily due to the added parallelism exploited by the multithreaded model compared to the data parallel model.

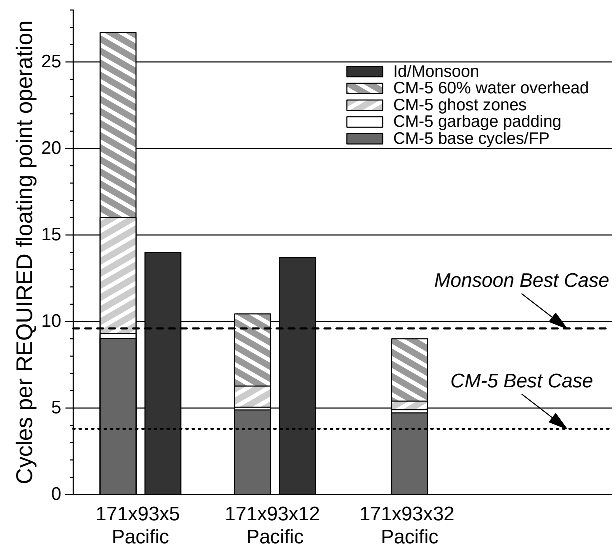

Figure 25: The multiplicative effects of overhead for the PCG, varying the number of layers.

100% water were considered.

The variable $N_{1/2}$ is typically used for vector processor to measure the vector length necessary to reach one half of the performance of the machine when it has a problem with an infinite vector length. In a parallel setting, $N_{1/2}$ will vary according to the size of the machine, and the problem. For GCM, we see that on the CM-5, $N_{1/2} \sim 1000$ per VU. To obtain the number for the entire machine, we must multiply by the number of VU's in the machine. For Monsoon, the same value is $N_{1/2} \sim 35$ per PE.

Because the CM-5 starts out at about 3.5 cycles per floating point operation versus Monsoon at about 11 cycles per floating point operation, the CM-5 is still better in absolute terms than Monsoon over much of the range shown in the graph, despite the fact that its performance deteriorates faster as the problem size decreases. The Monsoon version reaches peak performance at a smaller problem size because Monsoon exploits much more parallelism than the CM-5 does. Whereas the CM-5 generally takes advantage of data parallelism for each operation, Monsoon can take advantage of data parallelism for multiple array operations at a time, as well as procedural and instruction-level parallelism.

5.2.4 Multiplying overheads

The three primary overheads (padding, water, and size) described to this point are multiplicative in nature. Figure 25 shows the effects of these overheads on some example geometries. The ocean geometry chosen has a relatively high percentage of water (60% overall and 66% on the surface) compared to some real geometries. The high percentage of water in this particular geometry favors the data parallel approach; however, extrapolating the performance of geometries with less water is straightforward.

The first geometry on the left in Figure 25 is the 5-layer domain. Here, there is a significant amount of overhead from ghost zones and water. Also, the base cycles per (every) floating point operation is high because of poor vector unit utilization due to the relatively small problem size ($\sim$ 600 zones/VU). Garbage padding is a minimal overhead. For Monsoon, we see a ratio that is fairly close to the "best case" because Monsoon is less sensitive to problem size and percentage of water.

When we go to a 12-layer geometry, the CM-5 version improves significantly, primarily because of the better VU utilization due to a larger problem size – this directly effects the absolute water overhead. The ghost zone padding overhead is also reduced because of the deeper geometry. The Monsoon version remains about the same because it is not as sensitive to problem size.

For the final geometry, a 32-layer ocean, the CM-5 has a base cycles per floating point operation that is still noticeably worse than its best case. The padding and ghost zone overheads are almost negligible, but the water overhead boosts the ratio to about 9 cycles per required floating point operation. We were not able to run a problem of this size on Monsoon, but we predict it would be not much different than the 12-layer case, because Monsoon is not as sensitive to problem size as the CM-5.

The three problem sizes shown in Figure 25 are typical of simulations that we perform on a daily basis on the CM-5. If we had considered a geometry with less water, both versions would have had higher overheads, but the CM-5 performance would degrade more than the Monsoon version, because the CM-5 is more sensitive to variations in the percentage of water.

5.3 Conclusion

The best case for CM Fortran on the CM-5 is for very large, all-water geometries, and the efficiency of the CM-5 for these problems is about 3 times better than for Id on Monsoon (3.5 versus 10.1 cycles per floating point operation). However, in more realistic cases, the efficiency of the data parallel implementation varied from 0.5 to 2 times that of the multithreaded version. The multithreaded version was more flexible in handling problem irregularity, and it reached peak performance at a smaller problem size than the data parallel version.

For both versions, significant programmer tuning was required to obtain good efficiency, and the type of tuning for each version was very different.

Although multithreading is a far more general model of parallel computing than data-parallel, our results indicate that even for an application that is ideally suited for the data-parallel model, a multithreading implementation can be comparable in efficiency to a data-parallel implementation. Additional minor changes to the Monsoon microarchitecture (optimizing fanout, bubbles and switches) could reduce multithreading overhead by about 30-40%.

6 Implications for the Future

Both the CM-5 and Monsoon are custom, integrated parallel designs – the programming languages, compilers, processors and networks were all designed specifically for a parallel computing model. In both cases, scant attention was paid to compatibility with existing sequential or parallel programs. One lesson of the Nineties is that, to take advantage of the continuous improvements in technology, both the hardware and software for parallel computers must be derived from the widespread personal computer and associated server technology. Although the CM-5 used Sparc microprocessors for its nodes, most of its computational power came from its custom-built vector units. A modern symmetric multiprocessor (SMP) with four microprocessors is a much more general-purpose building block than the CM-5 node. In the future, Monsoon style multithreading may be incorporated in commercial microprocessors, but the external interface is more likely to be a sequential machine language than a dataflow graph.

Regardless, parallel computing has truly entered the mainstream in the form of small-scale 2 to 8-processor SMP's. These SMP's are based on the same commodity microprocessors used in desktop workstations and PC's, and being brought to market simultaneously with workstations and PC's by mainstream computer manufacturers. They are typically used in the business world either to handle multiple users running sequential applications, or as large database and World Wide Web servers. Almost incidently, they can also be used for scientific computing applications such as GCM. For higher performance levels, clusters of SMP's can be networked using either custom or commodity networks. Under the US Department of Energy ASCI program [30], two 4-Teraflop machines are being constructed as clusters of high-end SMP's.

However, so far SMP's have generally not been used for parallel computing in the traditional sense, and therefore research into the two programming models discussed in this study are still relevant. Both the data parallel and multithreading programming models can be implemented on SMP's and clusters of SMP's, and represent the two most promising models for exploiting parallelism on these architectures. In the remainder of this section, we discuss the implementation of data parallel and multithreading programming models for SMP's and clusters of SMP's.

6.1 Data parallel and multithreading on SMP's and SMP clusters

Most parallel computing performed on SMP's today can be characterized generally as data parallel or multithreaded, though SMP's have no special hardware support for either model. At the lowest level, the basic mechanism for exploiting parallelism within an SMP is OS threads or processes, and communication and synchronization occur through shared memory. Between SMP's, communication and synchronization are implemented through explicit message passing.

These low-level mechanisms can be exploited through C or Fortran programs via calls to parallel "libraries", that can be either multithreaded or data parallel. Multithreaded-style lightweight threads libraries are implemented on top of OS threads or processes, and provide primitives for creation and synchronization of lightweight threads. Message-passing libraries, such as PVM and MPI, provide communications and synchronization routines for programs that are usually structured in a data parallel manner (that is, they usually have a single logical thread of control, interleaving stages of communications and computation). Thread libraries can be implemented for clusters of SMP's, although the lack of a shared memory across the cluster becomes problematic. Data parallel message-passing libraries are more straightforward to implement both within SMP's and on clusters because they implicitly have a distributed memory model.

At a higher level, several compilers for High Performance Fortran (HPF) exist for SMPs – HPF is a successor to CM Fortran that has become the standard data parallel Fortran. Using HPF on high-end SMP's the GCM algorithm that has been under scrutiny here, can deliver a level of performance approaching that of the CM-5 [24]. Several vendors also provide HPF compilers for clusters of SMP's, and these compilers usually attempt to structure computation even within an SMP as phases of computation and message-passing style communication.

The state-of-the-art in high-level programming environments for multithreaded computation is several years behind data parallel, and no language or system has been as widely accepted as HPF has for data parallel computation. We are pursuing our research into multithreading by compiling Id and a related language pH for SMP's [41, 6]. Whereas Id and pH have implicit parallelism and synchronization, other languages require the user to expose the parallelization and synchronization. Some of these languages include Mul-T [29], Cilk [10], Concert [39], Cid [35] and Java [18].

6.2 Future research

It is important to discuss the problems that need to be resolved in order to provide the user with a single effective programming model which will work on a cluster and scale down gracefully to a single SMP or a workstation. We discuss both data parallel and multithreading models from this point of view.

Most of the technology for compiling HPF for SMP's and clusters of SMP's exists today, although it needs to be improved. For example, for efficient message passing, the compiler needs to coalesce messages to increase message granularity, and for a more effective use of an SMP, the parallel code fragments within an SMP need to maximize the use of threads and shared memory. It remains to be seen if the market for HPF is large enough for commercial vendors to make the required improvements. A way to broaden the use of data parallel compiler technology is to apply it to standard Fortran and C programs. This approach has been taken by the SUIF compiler [4], and research issues in that direction include detection of parallelism, mapping of data and work, restructuring of loops to exploit hierarchical memories, and load balancing in the presence of multiple users and processes.

Although data parallel is well-suited for GCM and other regular scientific applications, it is ill-matched to the vast majority of commercial applications that run on high-end computers today. Yet, the research in multithreaded computing on SMP's is not as advanced as for data parallel. The main compiler issues are the ability to optimize in the presence of hierarchical memories, the implementation of a shared memory abstraction in a distributed memory environment, and load balancing coupled with temporal and spatial locality concerns and multi-user environments. The compiler also needs to be integrated with the run-time system for efficient creation, synchronization and scheduling of threads. Compilation of fine-grain parallel languages like Id also requires partitioning of work into coarser grain threads for better efficiency. Further research for incorporating multithreading into a mainstream general-purpose language is also needed. It seems parallelizing a sequential language for multithreaded execution would have the biggest impact.

In comparing the suitability of these two models for SMP's and clusters of SMP's, the same issues that we have addressed in this paper arise. Overheads from the data parallel model come from poor handling of irregular problems, and overheads from multithreading come from dynamic parallelism and synchronization. Our study indicates that in some realistic circumstances overheads for multithreading can be comparable to a data parallel implementation even for regular scientific applications. The generality of the multithreading model may attract greater resources from industry, and may have much greater impact in the future than the data parallel model.

Credits and Acknowledgements

This research is a joint effort of the MIT Center for Meteorology and Physical Oceanography and the MIT Laboratory for Computer Science. The research of Prof. John Marshall and Chris Hill of the MIT Center for Meteorology and Physical Oceanography is partially funded by ONR contract N00014-95-1-0967, and University of California, Scripps Institute of Oceanography grant PO#10037358. The research of Prof. Arvind, Kyoo-Chan Cho, R. Paul Johnson, and Andrew Shaw of the MIT Laboratory for Computer Science is partially funded by ARPA under ONR contract N00014-92-J-1310.

The ocean model presented here was developed by a team working under John Marshall at MIT. Arvind and John Marshall began the Id implementation. Kyoo-Chan Cho completed the Id implementation with help from Chris Hill. Kyoo-Chan Cho also performed the initial benchmarking of the Id code on Monsoon with the help of R. Paul Johnson. R. Paul Johnson suggested the hybrid ocean model and has been responsible for gathering most of the statistics on Monsoon reported here. Chris Hill initially suggested the metric of cycles per required float-

ing point operations to compare the two models. Andrew Shaw performed the detailed benchmarking of the Id and Fortran elliptic solvers and developed the methodology for performance comparisons of the Id and Fortran codes. Andrew Shaw and Arvind analyzed the performance of the Fortran and Id codes.

We would like to thank Alan Edelman for his help in the explanation of the preconditioned conjugate gradient algorithm, and Wim Böhm and Larry Rudolph for a careful reading of earlier drafts.

A Mathematical formulation of ocean model

The ocean model solves the full three-dimensional Navier-Stokes equations for an incompressible Boussinesq fluid in a highly irregular domain such as that of an ocean basin. Development of such a model is of general interest because:

- Navier-Stokes models can be applied to a vast range of fluid flow problems in engineering and myriad phenomenon in the atmosphere and ocean. In particular, when viewed isomorphically, the incompressible Navier-Stokes equations can be used to study a compressible atmosphere – they are the basis of the pressure-coordinate quasi-hydrostatic atmospheric convection models of Miller and Pearce [33] – see [11]. Thus incompressible Navier-Stokes algorithms can be used to study motion in the atmosphere by exploiting a mathematical isomorphism.

- In oceanographic applications, the model is appropriate for the study of convective scales, where the hydrostatic relation breaks down, right up to planetary-scale motions. Thus if the resolution of the model were to be continually increased the model equations would be capable of representing motions down to the scale of meters.

- Numerical algorithms based on the Navier-Stokes equations need not be any more complicated (and are sometimes simpler) than those based on approximated forms and may offer advantages. For example, a common problem in hydrostatic ocean models is the noise in the vertical velocity field on the grid-scale of the model, particularly in the presence of steep topography. This noise may be inherent in the numerical algorithm; the equation that expresses the condition of incompressibility – the continuity equation – is used to compute the vertical velocity, leading to an accumulation of errors as the vertical integration proceeds upwards from the bottom.

A.1 The continuous equations

The physical variables of the ocean model are the fluid velocity $\vec{v}$, the density ρ, the scaled pressure p defined as the physical pressure divided by a constant average density, the temperature T, and the salinity S. The equations of motion are:

Newton's Law

$$\frac{\partial \vec{v}}{\partial t} = \vec{G_v} - \vec{\nabla} p \tag{A1}$$

where

$$\vec{G_v} = -\vec{v} \cdot \vec{\nabla} \vec{v} + \vec{f} + \vec{F} - \vec{D} - g\hat{e}_z \tag{A2}$$

and $\vec{f}$ is the Coriolis force, $\vec{F}$ are forcing and dissipation terms and $g\hat{e}_z$ is the gravitational force.

The heat equation is

$$\frac{\partial T}{\partial t} = G_T \tag{A3}$$

where

$$G_T = -\vec{v} \cdot \vec{\nabla} T + F_T + D_T \tag{A4}$$

and the salinity equation has the analogous form

$$\frac{\partial S}{\partial t} = G_S \tag{A5}$$

$$G_S = -\vec{v} \cdot \vec{\nabla} S + F_S + D_S \tag{A6}$$

In Equations A1 $-$ A6, $\vec{F}$, F_T and F_S are specified force terms and the D's are anisotropic dissipation terms of the form

$$\left\{ \begin{array}{c} \vec{D} \\ D_T \\ D_S \end{array} \right\} = -K_H \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) - K_V \frac{\partial^2}{\partial z^2} \left\{ \begin{array}{c} \vec{v} \\ T \\ S \end{array} \right\} \tag{A7}$$

The velocity field satisfies the continuity equation

$$\vec{\nabla} \cdot \vec{v} = 0 \tag{A8}$$

and the density is given by a ten-term polynomial approximation to the local equation of state $\rho(T, S)$

A.2 Method of solution

Given the variables $\vec{v}^n$, p^n, T^n and S^n, a predictor step of size $\triangle t$ is first taken to compute an approximate velocity at step $n + 1$

$$\vec{v}^* = \vec{v}^n + \triangle t(\vec{G_v}(\vec{v}^n, \rho(T^n, S^n)) - \vec{\nabla} p^n) \tag{A9}$$

from which the Euler-backward driving term G at step $n + 1$ is determined

$$\vec{G_v}^* = \vec{G_v}(\vec{v}^*, \rho(T^n, S^n)) \tag{A10}$$

To enforce $\vec{\nabla} \cdot \vec{v}^{n+1} = 0$, the pressure at step $n+1$ is calculated from the Poisson equation

$$\nabla^2 p^{n+1} = \vec{\nabla} \cdot \vec{G_v}^* - \frac{1}{\triangle t} \vec{\nabla} \cdot \vec{v}^n \qquad \text{(A11)}$$

where the last term is retained to control rounding errors and the velocity is then evolved to step $n+1$

$$\vec{v}^{n+1} = \vec{v}^n + \triangle t (\vec{G_v}^* - \vec{\nabla} p^{n+1}) \qquad \text{(A12)}$$

The temperature and salinity are updated analogously, with predictor step

$$\begin{aligned} T^* &= T^n + \triangle t G_T(\vec{v}^{n+1}, T) && \text{(A13)} \\ S^* &= S^n + \triangle t G_S(\vec{v}^{n+1}, S) && \text{(A14)} \end{aligned}$$

followed by the Euler-backward step

$$\begin{aligned} T^{n+1} &= T^n + \triangle t G_T(\vec{v}^{n+1}, T^*) && \text{(A15)} \\ S^{n+1} &= S^n + \triangle t G_S(\vec{v}^{n+1}, S^*) && \text{(A16)} \end{aligned}$$

All derivatives in Equations A9 $-$ A16 are approximated by second-order differences with velocities being defined on the centers of the mesh cell faces and P, S, and T defined at the cell centers. The Poisson equation is solved by conjugate gradient iteration.

References

[1] ADCROFT, A., HILL, C., AND MARSHALL, J. Representation of Topography by Shaved Cells in a Height Coordinate Ocean Model. *Monthly Weather Review* (in press 1997).

[2] AGARWAL, A., BIANCHINI, R., CHAIKEN, D., JOHNSON, K., KRANZ, D., KUBIATOWICZ, J., LIM, B.-H., MACKENZIE, K., AND YEUNG, D. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of ISCA* (1995). ftp://cag.lcs.mit.edu/pub/papers/isca95.ps.Z.

[3] ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. The Tera computer system. In *1990 International Conference on Supercomputing* (1990), pp. 1–6. ftp://www.net-serve.com/tera/arch.ps.gz.

[4] AMARASINGHE, S.P. AND ANDERSON, J.M. AND LAM, M.S. AND TSENG, C.W. The SUIF Compiler for Scalable Parallel Machines. In *Proceedings of the Seventh SAM Conference on Parallel Processing for Scientific Computing* (1995).

[5] ANG, B. S. Efficient implementation of sequential loops in dataflow computation. In *Proceedings of the Functional Programming Computer Architecture* (June 1993). ftp://csg-ftp.lcs.mit.edu:8001/pub/papers/misc/seq-loop.ps.Z.

[6] ARVIND, CARO, A., AND MAESSEN, J.-W. A Multithreaded Substrate and Compilation Model for the Implicitly Parallel Language pH. In *Workshop on Languages and Compilers for Parallel Computing* (Santa Clara, August 1996). ftp://csg-ftp.lcs.mit.edu:8001/pub/papers/csgmemo/memo-382.ps.Z.

[7] ARVIND, CULLER, D. E., AND MAA, G. K. Assessing the benefits of fine-grain parallelism in dataflow programs. *International Journal of Supercomputer Applications 2*, 3 (Fall 1988).

[8] ARVIND, AND EKANADHAM, K. Future scientific programming on parallel machines. In *Proceedings. Supercomputing '88* (1988), pp. 639–86.

[9] BARRETT, R., BERRY, M., CHAN, T. F., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND VAN DER VORST, H. *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.

[10] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of PPoPP '95* (1995). http://theory.lcs.mit.edu/ cilk/.

[11] BRUGGE, R., JONES, H. L., AND MARSHALL, J. C. Non-hydrostatic ocean modelling for studies of open-ocean deep convection. *Deep Convection and Deep Water Formation in the Oceans* (1991), 325–340.

[12] CANN, D. Retire Fortran? A debate rekindled. *Communications of the ACM 35*, 8 (August 1992), 81–89.

[13] CHIOU, D. Activation Frame Memory Management for the Monsoon Processor. Master's thesis, MIT EECS, September 1992. http://www-csg.lcs.mit.edu:8001/ derek/MS-Thesis.ps.

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

[15] DAHLQUIST, G., AND ET. AL. *Numerical Methods*. Prentice Hall, 1974.

[16] DENNIS, J. B., GAO, G. R., AND TODD, K. W. Modeling the weather with a data flow supercomputer. *IEEE Transactions on Computers C-33*, 7 (July 1984), 592–603.

[17] GOLUB, G. H., AND LOAN, C. F. V. *Matrix Computations*, second ed. North Oxford Academic, 1989.

[18] GOSLING, J., JOY, B., AND STEELE, G. *The Java^{TM} Language Specification*. Addison-Wesley, 1996.

[19] GRANT, P., SHARP, J., WEBSTER, M., AND ZHANG, X. Experiences of parallelising finite-element problems in a functional style. *Software – Practice and Experience 25*, 9 (September 1995), 947–74.

[20] GURD, J. R., KIRKHAM, C. C., AND WATSON, I. The Manchester prototype dataflow computer. *Communications of the ACM 28*, 1 (Jan. 1985), 34–52.

[21] HAMMES, J., LUBECK, O., AND BOHM, W. Comparing Id and Haskell in a Monte Carlo photon transport code. *Journal of Functional Programming 5*, 3 (July 1995), 283–316.

[22] HICKS, J., CHIOU, D., ANG, B. S., AND ARVIND. Performance Studies of the Monsoon Dataflow Processor. *Journal of Parallel and Distributed Computing 18*, 3 (July 1993), 273–300. http://www-csg.lcs.mit.edu:8001/Users/derek/Monsoon-Performance.ps.

[23] HILL, C., AND MARSHALL, J. Application of a parallel Navier-Stokes model to ocean circulation. In *Proceedings of Parallel CFD* (Caltech, 1995).

[24] HILL, C. N., AND SHAW, A. Transitioning from MPP to SMP: Experiences with a Navier-Stokes Solver. In *Seventh European Centre for Medium-Range Weather Forecasting (ECMWF) Workshop on the Use of Parallel Processors in Meteorology,* (December 1996).

[25] HILLIS, W. D., AND TUCKER, L. W. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM 36*, 11 (Nov. 1993), 31–40.

[26] HIRAKI, K., NISHIDA, K., SEKIGUCHI, S., SHIMADA, T., AND YUBA, T. The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems. *Journal of Information Processing 10*, 4 (1987), 219–226.

[27] HIROMOTO, R., WIENKE, B. R., AND BRICKNER, R. G. Experiences with the Denelcor HEP. *Parallel Computing 1*, 3-4 (December 1984), 197–206.

[28] KOELBEL, C. H., LOVEMAN, D. B., SCHREIBER, R. S., STEELE, G. L., AND ZOSEL, M. E. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[29] KRANZ, D., HALSTEAD, R. H., AND MOHR, E. Mul-T: A High-Performance Parallel Lisp. In *Parallel Lisp: Languages and Systems* (June 1989), T. Ito and R. Halstead, Eds., Springer-Verlag, pp. 306–311.

[30] LOS ALAMOS NATIONAL LABORATORY. ASCI Executive Summary. Web site. http://www-c8.lanl.gov/asci/summary.html.

[31] MARSHALL, J., ADCROFT, A., HILL, C., AND PERELMAN, L. A finite-volume, incompressible Navier-Stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research 102*, C3 (1997), 5753–5766.

[32] MARSHALL, J., HILL, C., PERELMAN, L., AND ADCROFT, A. Hydrostatic, quasi-hydrostatic and non-hydrostatic ocean modeling. *Journal of Geophysical Research 102*, C3 (1997), 5733–5752.

[33] MILLER, M. J., AND PEARCE, R. P. A three-dimensional primitive equation model of cumulonimbus convection. *Quart. J. R. Meteorol. Soc.* (1974), 133–154.

[34] NIKHIL, R. S. Id Language Reference Manual, Version 90.1. Tech. Rep. CSG Memo 284-2, Laboratory for Computer Science, MIT, July 1991.

[35] NIKHIL, R. S. Cid: A Parallel "Shared-memory" C for Distributed Memory Machines. In *Proceedings of the 17th Annual Workshop on Languages and Compilers for Parallel Computing* (Ithaca, NY, August 1994), Springer-Verlag. http://www.research.digital.com/CRL/ personal/nikhil/cid/home.html.

[36] OLDEHOEFT, R., CANN, D., AND ALLAN, S. SISAL: initial MIMD performance results. In *Proceedings of the Workshop on Future Directions in Computer Architecture and Software* (May 1986), pp. 139–45.

[37] PAGE, R. L., AND MOE, B. Experience with a large scientific application in a functional language. In *Proceedings of FPCA '93* (June 1993), ACM, pp. 3–11.

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

[39] PLEVYAK, J. B. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois, 1988.

[40] SAKAI, S., YAMAGUCHI, Y., HIRAKI, K., KODAMA, Y., AND YUBA, T. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of ISCA* (1989), pp. 46–53.

[41] SHAW, A. *Compiling an Implicitly Parallel Language for Multithreaded Execution on Symmetric Multiprocessors*. PhD thesis, MIT, 1997. *In preparation*.

[42] SHEWCHUK, J. R. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU//CS-94-125, Carnegie Mellon University, School of Computer Science, Mar. 1994. ftp://reports.adm.cs.cmu.edu/1994/CMU-CS-94-125.ps.

[43] SMITH, B. J. Architecture and applications of the HEP multiprocessor computer. In *Real-Time signal processing IV* (1981), vol. 298, pp. 241–248.

[44] SOHN, A., KONG, L., AND SATO, M. Progress report on porting Sisal to the EM-4 multiprocessor. In *Parallel Architectures and Compilation Techniques* (August 1994), pp. 351–4.

[45] SUR, S., AND BOHM, A. Analysis of Non-Strict Functional Implementations of the Dongarra-Sorensen Eigensolver. In *Proceedings of the International Conference on Supercomputing* (1994), pp. 412–418.

[46] SUR, S.; BOHM, W. Functional, I-structure, and M-structure implementations of NAS benchmark FT. In *Parallel Architectures and Compilation Techniques* (August 1994), pp. 47–56.

[47] THINKING MACHINE CORPORATION. *CM Fortran Language Reference Manual, V2.1*, January 1994.

[48] YEUNG, D., AND AGARWAL, A. 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* (May 1993), pp. 187–197. ftp://cag.lcs.mit.edu/pub/papers/fine-grain.ps.Z.

Contents