Parallelizing a Cell Simulation: Analysis, Abstraction, and Portability

by

Stephen Steinberg

University of California at Berkeley, 1995

This report is a revised version of the author's Master's report, which was submitted to the Department of Electrical Engineering and Computer Sciences on December 14, 1994 in partial fulfillment of the requirements for the degree of Master of Science at the University of California at Berkeley. The project was supervised by Katherine A. Yelick and was supported in part by ARPA contract DABT63-92-C-0026, by DOE grant DE-FG03-94ER25206, and by NSF Infrastructure Grant CDA-8727788.

Abstract

Biologists hope to use detailed physical simulations to test theories about cell movement. Computer scientists would like to be able to design parallel programs that run efficiently on a wide range of machines. This paper describes the intersection of these two desires. We present a general design methodology for portable parallel programs, use it to parallelize a cell simulation, and then analyze the result on a CM-5, SP-1, and network of workstations.

Our methodology is to use analytical models combined with empirical measurements of important kernels in order to choose between algorithms and data layout schemes. We can the implement efficient and re-usable data sturctures to hide the complexities of distributed memory programming. Detailed measuremenets of the application and the machines are then used to determine machine-specific optimizations. Using this technique we achieved good speedups on all three machines. This paper describes the results of each stage and discusses their implications for multiprocessor designers.

Chapter 1

Introduction

Computer scientists talk about parallelizing applications, but few actually do so. Instead, physicists and chemists have written most of the existent parallel applications while computer scientists have primarily focused on analyzing and implementing a few small kernels. The result has been applications that are too difficult to modify or port to be useful for computer scientists, and new tools and techniques for parallel programming that remain largely unproved in the messy world of real applications.

For this project we parallelized a real-life, medium-sized scientific application designed to simulate blood clotting and cell aggregation. We followed a rigorous methodology during the three phases of program development. For the initial design we used analytical models, validated by measurements, to choose between alternate parallel algorithms and data-layouts. Then, to aid in implementation, we built parallel data structures that hid the distributed nature of the machine. Lastly, to ensure the programs longevity, we ported the application to three different machines, paying close attention to how different machine characteristics affected the program's design and implementation.

For program design we used a simple model of computation and communication, similar to the LogP model of [?], in conjunction with measured performance costs. The model was useful for limiting alternatives, but because it ignored memory costs and network contention, it was poor at predicting running time and fallible in its design choices for small n. Therefore, we wrote multiple implementations of a few important routines in order to validate our design decisions.

Programming complexity has always been the biggest hurdle to parallel computing's success.

But the use of efficient and re-usable data structures can considerably simplify the process. For this application we implemented two parallel data structures, a grid and a graph. These data structures significantly eased implementation by providing a physically meaningful interface, and by largely hiding the underlying data layout and communication. Only locality information necessary for efficient programming was made visible to the user.

The third stage, the pursuit of portability, really began with the application's original design, when we plugged measurements of three machines into our analytical models. Our hypothesis was that different machine characteristics – different communication/computation ratios – would affect how the program should be written. We found several cases that confirm this hypothesis and that have interesting implications for programmers and multiprocessor designers.

The rest of this project report explores these issues of modeling, data structure design, and portability in more detail. The remainder of this section gives a quick review and justification of the linchpins of our methodology. Section 2 provides an overview of the application, while section 3 gives an overview of the parallel machines that we used. Then, in section 4, we explore in detail the parallelization of the first phase of the application, the Navier-Stokes solver. In section 5, we discuss the application's second phase, cell manipulation. We conclude, in section 6, with information on total achieved performance and a summary of our results.

1.1 Overview of methodology

In order to achieve both efficiency and portability we wrote the application in Split-C. Like its sequential precursor, Split-C is a small language, available on many platforms, that provides low- level mechanisms that expose the full performance of the underlying hardware. The language extends C by providing a global-address space as well as message-passing and synchronization primitives. The notion of a global pointer, formed from a processor number and a local memory address, allows global addresses to be accessed. We chose Split-C over parallel FORTRAN variants such as HPF or CRAFT because of the irregular nature of the application. In particular, it would be difficult to partition and efficiently manipulate the distributed network data-structure with current versions of HPF.

The second notable component of our methodology is a heavy reliance on empirical measurements to justify design decisions. This is partly due to the nature of our application: memory access time is as significant as floating-point performance, making analysis particularly difficult. This is shown in section 4.1. But it is also out of a strong distaste for analytical methods on epistemological grounds. Models of MPP performance must be incredibly simplified for analysis to be tractable, and it is correspondingly difficult to determine a model's error range. This makes the use of analytical models to choose between algorithms for finite problem sizes largely an act of faith. Instead, our approach has been to build rough models to narrow the design space, and then time multiple implementations of important kernels. We demonstrate this approach in determining partitioning schemes for the application's grid and network data-structures.

Chapter 2

Application overview

Biologists do not currently understand what causes the cells of an embryo to combine and build characteristic features such as fingers and eyes. At UC Berkeley, biologists are studying this process of morphogenesis through microscopic analysis of sea urchin embryos. These scientists now believe that the geometric structure of the embryo emerges from a few simple, local rules of cell movement [?]. The goal of our application is to allow scientists to posit such rules, and then by combining them with accurate physics, simulate the resulting movement and aggregation of cells. Similar simulations have been developed but they have been designed for machines with only a few processors. This has limited scientists to either running very small, unrealistic scenarios, or fighting for time on one of the handful of Cray C90s.

An additional motivation for our work is that the simulation technique we are parallelizing is general enough that it can be used to simulate many different systems that involve bodies immersed in an incompressible liquid. The technique was first developed by Charles Peskin to model blood flow in the heart in order to aid in the design of artificial-heart valves [?]. Peskin and his team are currently developing more complex, 3-dimensional simulations, but have been constrained both by the limited availability and memory of non-parallel supercomputers. The same simulation algorithm has also been used by Aaron Fogelson to simulate sperm motility and platelet aggregation during blood clotting [?], and our application closely follows his work. This project began by converting Fogelson's sequential FORTRAN code to C, and much of the performance analysis in this report is based on running blood clotting simulations.

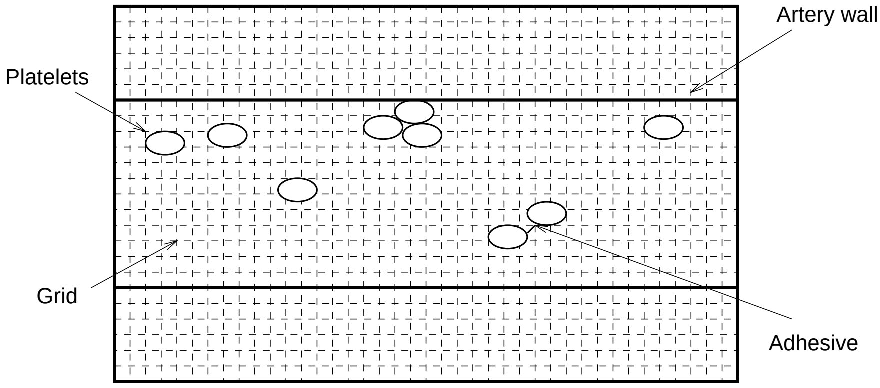

Figure 1: Artery simulation

2.1 Computational model

The simulation's key concept is to model the system, whether it be a heart or a sea urchin embryo, as a collection of elastic fibers immersed in an incompressible fluid. The forces of these fibers are spread to the fluid and the velocity of the fluid is then found with the Navier-Stokes equations. The fluid forces are then used to push the fibers to their new location.

The fluid is modeled on a dense, rectangular, periodic grid. The grid points must be much closer together than the points that make up the fibers and cells. In a periodic grid, each edge wraps around like a torus in order to avoid boundary conditions.

Cells are represented as polygons made up of fibers, where each fiber is modeled as an elastic spring. Cells may change in shape but they retain a constant area. For the platelet simulation, these cells are inserted at the left side of the grid and then flow to the right. Two cells that are nearby will form adhesives, also modeled as springs, which keep the two cells tied together. Cells may also form adhesives with the artery walls near the top and bottom of the grid. These walls are modeled as fibers, but special anchors are used to ensure that the walls remain relatively stable despite the fluid forces (figure 1).

There are four main operations at each time step of the simulation algorithm. First, the fiber forces are spread to the fluid. This is done using the Dirac delta function to extrapolate the force of each fiber point to the surrounding grid points. The second step, and the most computationally expensive, is to solve the Navier-Stokes equations over the fluid grid. The third step is the inverse of the first; now the fluid force is interpolated back to the cell fibers and the fibers are moved to their new position. Finally, any local interaction between cells is performed. For the platelet simulation this involves forming adhesives between nearby platelets. For the

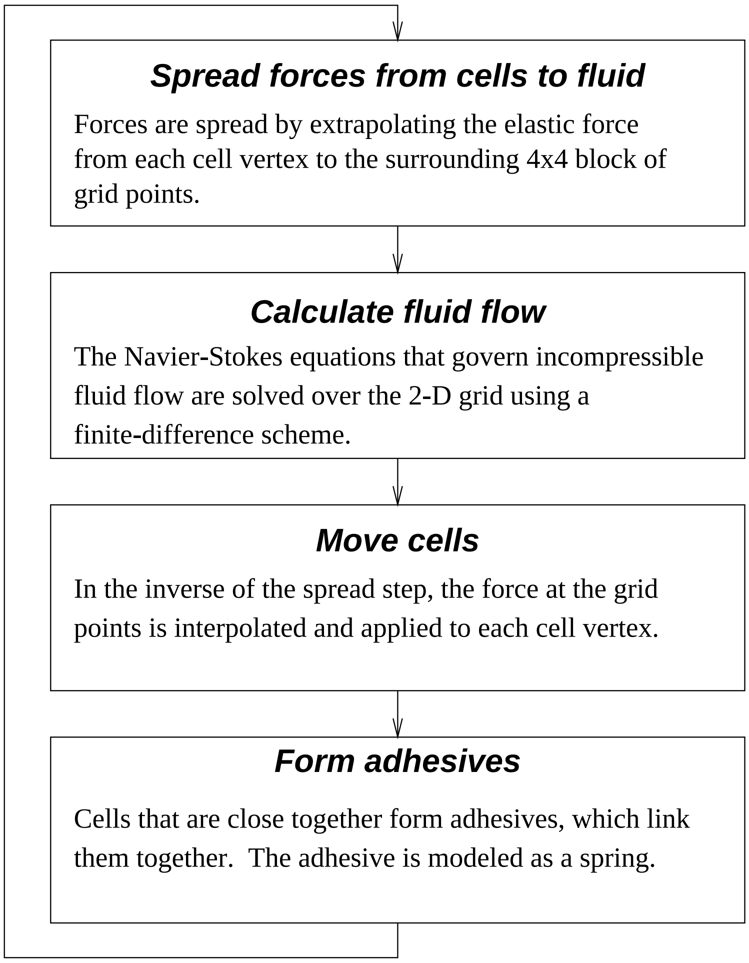

Figure 2: Operations during one simulation time-step.

sea urchin simulation there will be some additional protrusion calculations. These steps are summarized in figure 2.

Chapter 3

Machine overview

In order to examine the portability of the cell application – both the ease of getting the program to run and the efficiency of the result – we analyzed the program on three very different parallel machines. The first machine was a 32-node CM-5, a MPP released around 1991-1992 and costing approximately $1 million ($31,250/node). The second machine was an 8-node IBM SP-1, released in 1993 for $400,000 ($50,000/node). Like the CM-5, the SP-1 is a collection of commodity processors connected by a custom network, however, the processors are significantly faster and access to the network hardware is tightly restricted. The third system we examined is an example of a NOW, or Network of Workstations. It is made up of four HP 9000/35 (late-1993) workstations hooked together with a FDDI network. It is symptomatic of a NOW that it has the newest and fastest processors but a relatively high-latency network. This collection of HPs and Medusa network-interface cards lists for about $156,000 ($39,000/node).

The following sections characterize the three systems in terms of local computational speed and communication performance. This information proves critical in making sense of subsequent application timings.

3.1 Local computation

The three computer systems are powered by three generations of microprocessors: the oldest, the CM-5, is built from 33 MHz SPARC 2 processors (and optional floating-point vector accelerators), the SP-1 is built from 62 MHz RS-6000s, and the HPs from 99 MHz PA-RISC chips.

In order to evaluate the local computation performance of these processors we focused on

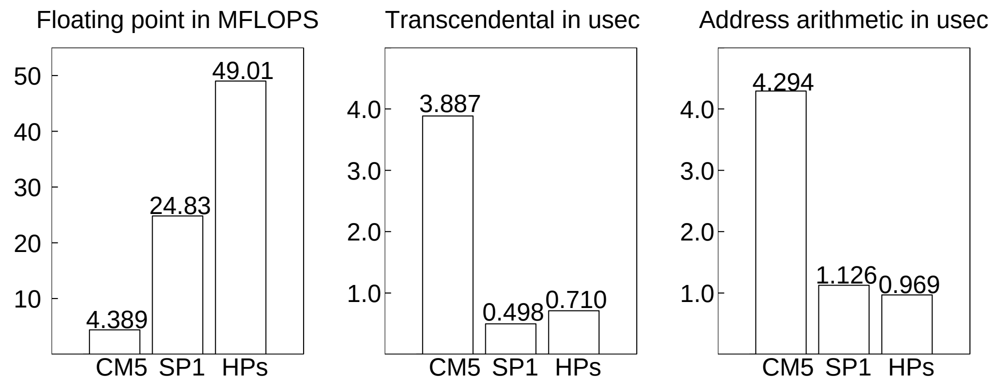

Figure 3: Local computational performance using multiply for floating point performance and sin for transcendental

what could actually be achieved from optimized C code, since published claims of MFLOPS are notoriously unrealistic. We timed long loops of simple operations written in C and compiled with `gcc -O4`. The first test was time required for multiply-adds (with doubles). The second test timed the `sin()` function, an important metric because transcendental functions are heavily used in the cell simulation. It is interesting to note that the SP processors are faster at `sin()` than the newer HPs. The third test was the time required to calculate a global grid address, which involves two divisions, two mods, and then calculating the address for an element in the spread-array `Grid[PROCS][PROCS+1]::[XLEN][YLEN][2]`. The high-cost of this operation points to the significance of computational parallel overhead, a type of overhead that is too often ignored (figure 3).

3.2 Local memory

The cost of memory operations are critical to this application because it exhibits poor locality. As later sections will show, cache-miss time is as relevant to performance as floating-point speed. To characterize the systems' memory hierarchy we used micro-benchmarks as described by Saavedra in [?]. The technique is straightforward: step through an array, varying the size of the stride, and plot the average latency per memory access. This is best described with the pseudo-code in figure 4.

Like Saavedra, we used a read-modify-write sequence for the memory operation, which shows the relative costs of cache hits and misses. All loop and address calculation overhead is subtracted out so that the time reflects only the requisite memory operation. The results of

```
for (N = 4 Kilobytes; N < 8 Megabytes; N *= 2) {
    for (stride = 1; stride < N; stride *= 2) {
        for (i = 0; i < N; i += stride) {
            memory operation on A[i];
        }
    }
}
```

Figure 4: Code used to test memory operations

the experiment for the three systems are shown in figure 5.

These graphs expose both the structure of the memory hierarchy as well as relative costs. Each line is for a different size region, and each point represents the average memory cost for a given region and stride. When a region is less than or equal to the cache size, the corresponding line remains flat. Once the region exceeds the size of the cache, the reads begin to generate misses and the line goes up. When the stride is equal to the cache's line size every access results in a miss and the memory cost flattens. Then, as we continue to increase the array size and stride, we may see another rise in average time per memory access. This is due to the TLB (translation look-aside buffer), which can not map all the pages of the array at once.

From the above graphs we can see that the CM-5 and SP-1 both have medium-sized caches (64K and 32K respectively), associative TLBs, and relative cache miss times on the order of 20 cycles. The HPs have very large caches (256K), either a direct-mapped cache or a page size too large to be exposed by the experiment, and a cache miss time around 50 cycles. Clearly, the HP memory hierarchy has been much more aggressively designed for cache hit, rather than miss, times. Unfortunately, this does not work well for the cell-simulation application.

Like most scientific code, the cell-simulation exhibits poor locality. During the application's first phase, which involves fluid calculations, the program streams through the grid data which

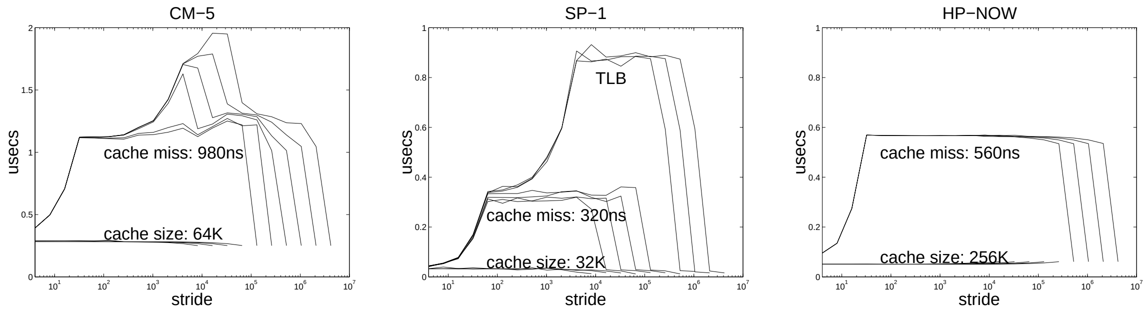

Figure 5: Local memory characteristics

is much too large to fit in the cache. In the second phase, cell interaction, the program relies on irregular, linked-list data structures that exhibit little or no locality. This points to one potential problem with NOWs. Because workstations are designed for applications that do exhibit locality they may be poorly suited to operate as processing nodes for large scientific applications. Having a large cache, or an L2 cache, will increase the memory miss time and therefore hurt performance of some large scientific applications. Note that a similar conclusion was drawn with respect to the Cray T3D and DEC Alpha microprocessor [?].

3.3 Communication

While computational performance follows a simple trend toward increased speed over time, communication performance is more difficult to characterize. Although multiprocessors have traditionally offered very low-latency, medium-bandwidth communication, the shift toward NOWs is resulting in much higher-latency networks. The three systems examined in this study take very different positions in the network design space [?].

The CM-5 has both a low-latency data network and a dedicated control network for rapid synchronization. The networks are based on a fat-tree topology and use adaptive routing [?]. For Split-C, communication is done using libsplit-c, which in turn is built on top of CMAML, a fairly light-weight implementation of active messages [?]. An important note is that because the CM-5 network does not offer any form of flow control outside of back pressure, the user must perform careful bandwidth matching to achieve maximum performance [?]. We discuss our use of bandwidth matching in section 4.

The SP-1 offers the slowest communication of the three systems, but this is an indictment of the machine's message-passing library rather than its network. The network is based on the Vulcan switch, which is used to implement a topology similar to a Banyan network. This hardware provides a raw bandwidth of 39 MBytes/sec and a latency of under 3 μseconds [?]. However, libsplit-c for the SP-1 was built on top of IBM's communications library, MPLp, and offers significantly lower performance. Perhaps most startling is the high-cost of barriers: 195 μseconds (or 4680 FLOPS) versus the CM-5's 3.6 μseconds (16 FLOPS). This significantly impacts code portability since code written for the CM-5 or T3D often relies on barriers to speed up performance, a technique that is contradicted for the SP-1.

The third system, the HPs, are connected together with an FDDI network using Medusa

	CM-5 (CMAML)	SP1 (MPLp)	HPs (HPAM)
Packet	20 bytes	up to 255 bytes	up to 4KB
AM latency	14.5 μsec	88.6 μsec	35.2 μsec
AM send overhead	2.1 μsec	33.7 μsec	3.2 μsec
Get	5.2 μsec	57.5 μsec	16.9 μsec
Store	2.7 μsec	34.5 μsec	3.4 μsec
Store bandwidth	8 MB/sec	8 MB/sec	40-12 MB/sec
Barrier	3.6 μsec	195 μsec	50 μsec

Table 3.1: Communication performance with 4 processors [Luna94]

interface cards. Performance is surprisingly good for what is essentially a LAN: latency is little more than twice that of the CM-5, and the network provides the greatest bandwidth of the three systems studied [**?**]. Because FDDI supports variable sized packets of up to 4K it is particularly advantageous to use bulk operations when possible. Because the network interface provides buffering for up to one packet, write bandwidth for messages less than 4K is up to 40 MB/sec (depending on interarrival time). However, for messages larger than 4K, bandwidth drops to the FDDI rate of 12 MB/sec. Note that because the network does not guarantee delivery, every send results in a reply.

Chapter 4

Navier-Stokes solver

The computational core of the cell simulation is the Navier-Stokes solver. At each time-step the incompressible Navier-Stokes equations,

$$\rho \left[\frac{\partial u}{\partial t} + u \cdot \nabla u \right] = -\nabla \rho + \mu \nabla u + f(x,t) \quad (4.1)$$

$$\nabla \cdot u = 0 \quad (4.2)$$

are solved over a rectangular, periodic grid to find the fluid's velocity and pressure field. The solver is based on Chorin's projection method (a finite-difference scheme) and the Fourier-Toeplitz fast Poisson solver [?].

For the original sequential application, operating on a 128x64 grid with 16 platelets, 84% of computation time was spent inside this Navier-Stokes solver. Of that time, 61% was spent calculating two 2-dimensional Fast Fourier Transforms and 31% was spent in two tridiagonal solvers (table 4.1).

The first step in parallelizing the Navier-Stokes solver was determining how to partition the grid. Although a blocked layout seemed intuitively best, modeling and detailed measurements showed that a skewed layout results in better performance. This analysis is described

Total breakdown		**Within NS**	
Navier Stokes	84%	FFT	62%
Main	5.7%	Tri	31%
Spread	5.7%	Other	7%
Xmove	2.3%		
Attach	2.2%		

Table 4.1: Sequential breakdown (CM-5)

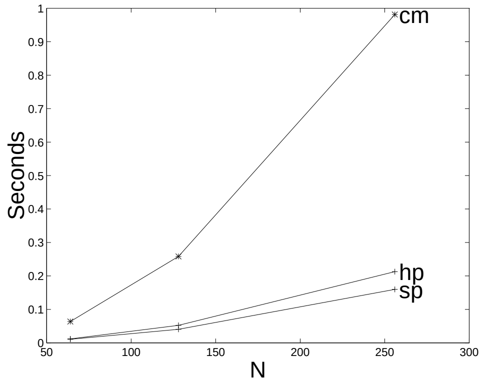

Figure 6: Time for sequential `tridgx()`

in section 4.1. Once the grid's layout had been determined we created a library of routines to operate on the grid data-structure. These routines, and measurements of the accompanying overhead, are detailed in section 4.2. The heart of the Navier-Stokes solver is the Fast Fourier Transform, and its conversion is described in section 4.3. Finally, in section 4.4, we discuss the total speedups obtained for the parallel Navier-Stokes solver.

4.1 Tridiagonal solver

Thirty-one percent of the time spent inside the sequential Navier-Stokes solver is spent within the tridiagonal solvers. These solvers operate in a series of wave-front operation, where each grid value depends on the previously calculated grid-value. There are two solvers: `tridgx()` which operates first from right-to-left and then left-to-right, and `tridgy()` which moves from top-to-bottom and then bottom-to-top. Each routine performs 30 normalized FLOPS and roughly 35 memory accesses per grid point. Unfortunately, most of these memory accesses are uncached. This can be seen in figure 6, where the SP-1 processors execute the sequential code faster than the HPs, despite the HP's superior floating- point performance.

Because of the wave-front nature of the tridiagonal solvers, the parallel version requires both synchronization and communication. Synchronization because a processor can not begin operating on its grid values until the previous grid values have been calculated, communication because the last column of calculated grid values must be sent to the adjacent processor. But how much synchronization and communication is required depends on how the grid is partitioned across processors. For example, if the grid is laid out in row blocks, so that each processor owns N/p contiguous rows of grid values, no communication would be required for

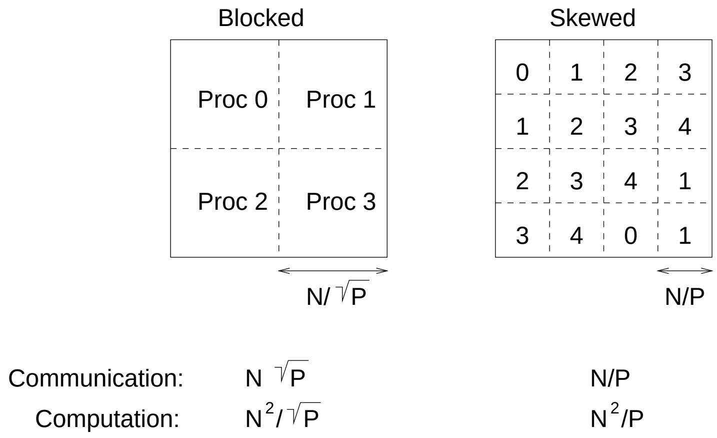

Figure 7: Blocked vs. skewed layout

`tridgx()`. However, `tridgy()` would require p steps of communication, and no more than one processor could ever be operating simultaneously. The following section examines two other, more efficient layouts, and uses a simple model to predict their performance.

4.1.1 Analytical model

The fewer the grid blocks in each dimension, the less the communication. Therefore, a blocked layout, where there are only $\sqrt{p}$ blocks in each dimension, results in the minimum communication. However, this layout also provides only $\sqrt{p}$ parallelism. To maximize parallelism (i.e., minimize synchronization) we want as many independent blocks in each dimension as possible. This implies a skewed layout, which allows for full p parallelism in both dimensions, but which requires p stages of communication (figure 7).

Which layout is superior depends both on the machine's computation-to-communication ratio, as well as the values of N and p. In order to determine which layout to use for the three different machines (using the lowest-common-denominator of 4 processors), we built a simple model based on the equations given above as well as the machine characteristics from section 3. Following the LogP model, we calculated communication costs using message-overhead and per-byte times, and ignored the cost of memory accesses. The results are shown in figure 8.

The model shows that a skewed layout is superior to blocked for all N on the CM-5. But for the SP-1 and HPs, with their slower communication, skewed becomes superior only once N is larger than 64. Note that if no communication were required for the solver, skewed would run in $1/\sqrt{p}$ the time of blocked. The SP-1 comes closer to this ideal ratio ($1/\sqrt{p}$ =.5 when p=4) than the HPs, despite the HP's better communication performance, because of the two

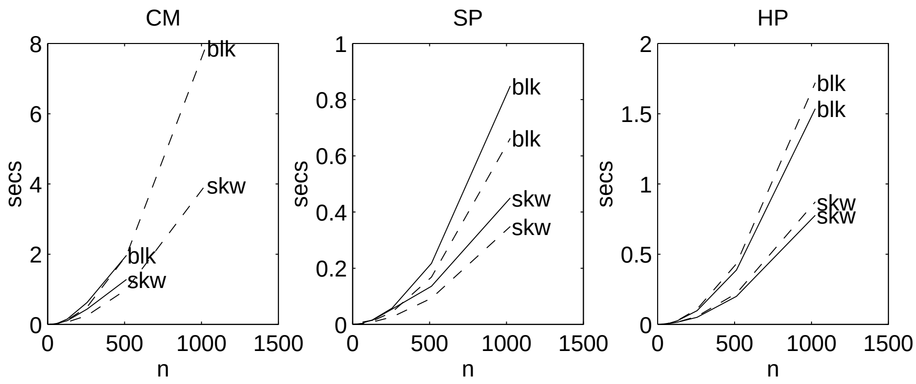

Figure 8: Analytical model for `tridgx()` with $p = 4$

machines' MFLOPS to bandwidth ratios.

This model ignores many important machine characteristics and simplifies others. The most important omission is memory-access time. As was shown in figure 6, the SP-1 is actually faster than the HPs at the local tridiagonal code because of the SP-1's faster uncached reads. This extra local computation cost could significantly add to the advantage of a skewed layout. On the other hand, the model also ignores communication contention and receive overhead. While for a blocked layout the sending and receiving processors are mutually exclusive, for a skewed layout every processor is simultaneously sending and receiving. This could dramatically increase the time required for skewed communication, especially if the message-passing layer uses interrupts rather than polling. A final example of the model's simplification is that the HP's send bandwidth is represented as a single number, when it actually depends on the size of the message (see section 3.3).

Of course, the question raised by noting these flaws with the model is, do the simplifications matter enough to affect the final answer? The following section, in which the two versions of the tridiagonal solver are implemented and then timed, addresses this question.

4.1.2 Empirical analysis of solver

Figure 9 shows the total time required for one of the tridiagonal solvers to run on the various 4-processor systems. (Note that because the solver requires many temporary NxN grids, memory limitations restricted N to 512 on the CM-5.)

The clearest lesson of these measurements is that the model badly underestimated running time, especially in the case of the CM-5 and the HPs. But this should come as no surprise since

Figure 9: Actual performance of `tridgx()` with $p = 4$

Figure 10: Communication time for `tridgx()` with $p = 4$

the model ignores memory access time. Note that, contrary to the model but similar to the sequential case, the SP-1 is faster than the HPs.

A second lesson is that the crossover points where skewed becomes superior to blocked are different than those predicted by the model for the CM-5 and SP-1. For the CM-5, blocked is faster than skewed through $N = 64$. And for the SP-1, blocked is faster all the way through $N = 256$.

This second lesson may seem to contradict the first, since more expensive local computation should make skewed even more advantageous. Part of the explanation lies in the communication contention and receive overhead. Because a skewed layout requires $p-1$ rounds of communication while a blocked layout only requires $\sqrt{p}-1$, our model for $p = 4$ predicts that the skewed communication time will be exactly three times that of blocked. However, figure 10 shows that it is actually significantly worse.

This discrepancy is unsurprising since, as discussed earlier, skewed communication requires all processors to simultaneously send and receive. The discrepancy is largest for the SP-1 because of its high receive overhead; the affect is compounded on the HPs because the network is a shared bus and therefore handles contention poorly.

4.1.3 Resulting tridiagonal efficiency

The original sequential simulation operated on a 128 x 64 grid, but this size was determined largely by memory constraints. Because there are many temporary grids, a simulation using a 128 x 64 fluid grid requires over three megabytes of memory, while a 256 x 128 grid requires more than twelve megabytes. The biologists at UC Berkeley are interested in operating on much larger grids. Therefore the best metric for this application's parallel performance is scaled efficiency, or *scaled speedup*/p, where

$$\textit{scaled speedup} = \frac{t_1(pM)}{t_p(pM)}, M = \text{total memory required} = N^2 \text{ for } \texttt{tridgx}() \tag{4.3}$$

Because the Navier-Stokes solver is designed only for grids whose dimensions are powers of 2, scaled efficiency figures can only be found for (in the case of our CM-5) 1, 4, 16, and 64 processors. Figure 11 shows these results for `tridgx()`. The efficiency of the skewed implementation decreases linearly with p because their are $p-1$ communication phases and because the block size (but not the total number of local grid points) decreases with p. Blocked efficiency decreases with the ratio $\sqrt{p}$ to p, so 16 processors results in an efficiency of .25.

Because of the small size of the other two systems we can only look at the case of 4 processors, but this is small enough for meaningful, un-scaled, speedup calculations. Doing so shows that the CM-5 provides a speedup of 3.5 on a 256 x 128 grid, the SP-1 provides a parallel speedup of 3.3 on a 512 x 256 grid, and the HPs reach 3.88 for a 256 x 128 grid.

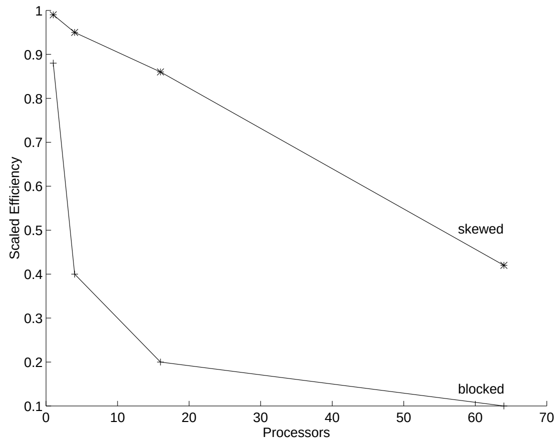

Figure 11: Scaled efficiency of `tridgx()` starting at 128 x 64

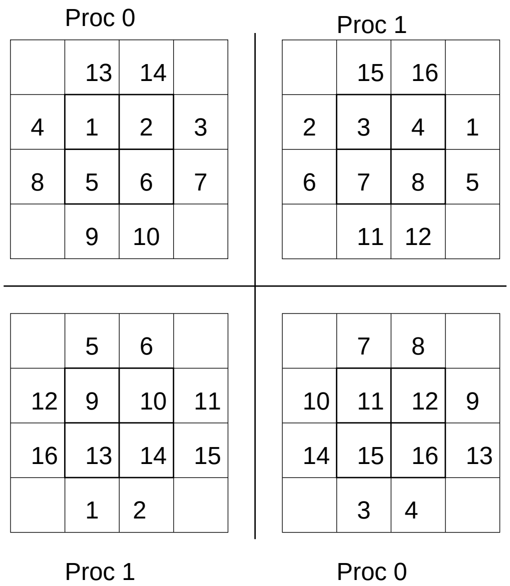

Figure 12: Ghost cells for 4x4 grid on 2 processors

4.2 Local manipulations

Most of the code within the Navier-Stokes solver involves local manipulation, where only data local to the processor is modified. This would seem to be embarrassingly parallel, but there are two elements of overhead that prevent linear speedups. First, each "block" of the grid is surrounded by *ghost values*, array elements that contain the value of neighboring, remote grid points (figure 12). These ghost values must be updated at some point after a local manipulation operation is done, and before any operation that requires the ghost values. This update requires each processor to communicate with its neighbors and is fairly expensive.

The second type of parallel overhead is the computational burden of operating on a more complex data structure. For example, operating on the entire grid in the sequential code looks like:

```
for x=1 to NX
  for y=1 to NY
    grid[x][y]++;
```

while the equivalent pseudo-code for the parallel version requires an extra loop and a coercion operation:

```
for blocks=1 to PROCS
  mygridblock = tolocal(grid[blocks])
    for x=1 to NX
      for y=1 to NY
        mygridblock[x][y]++;
```

Figure 13: Effects of cache on parallel speedup (CM-5)

Yet counterbalancing both types of parallel overhead is the advantage gained from operating on 1/pth of the total grid. Because less memory is operated on, cache and TLB misses will be significantly lower for some routines. Figure 13 illustrates the effects of cache size on a routine that repeatedly iterates over the grid, incrementing the value at each position.

The next two sections will examine how all three of these factors operate together to influence the resulting speedup of local-manipulation code.

4.2.1 Macros and computational overhead

In order to obtain a skewed layout for a grid that is NX by NY, with ghost cells around each block, we use the Split-C declaration:

```
double grid [PROCS][PROCS+1]::[((NX/PROCS)+2)][((NY/PROCS)+2)];
```

The `[PROCS+1]` dimension produces an extra column of blocks that "skews" the processor grid but is otherwise ignored by our program. Operating on this 4-dimensional array is awkward because it bears little relation to how we think of a 2-dimensional grid. So to ease this problem, we provide an extensive suite of macros and library routines. These are used to convert from global to local coordinates, iterate over the blocks in a particular column or owned by a particular processor, and to provide simple global operations. Besides simplifying the programmers job, our hope was that these routines would completely hide the grid's layout from the user. This would make it possible to switch data-layouts by simply changing the grid library (from skewed to block, say) while leaving the user's code unchanged. But this was not achieved. Some of the program's routines take advantage of the fact that each processor owns exactly one block in each processor column, an assumption that would not be true if the grid is blocked. The only way to avoid this would be to add an extra level of indirection, which would mean more overhead.

By avoiding any unnecessary overhead, the additional computation resulting from the parallel data layout has been kept very low. The cost of the extra for-loop and pointer coercion, which is all that is required for most of the parallel local-manipulation, is too small to measure. However, if we are accessing only particular, specified grid points instead of iterating over the entire grid, the time required to translate from global coordinates to a local address results in substantial overhead (as was shown in section 3.1).

One further prosaic but important note: a skewed grid declaration uses PROCS, a pseudo-constant that can be set at run-time, in the dimensions to the right of the spreader. So even when a tolocal() is done the cost of accessing a local grid point will be more expensive than in the sequential version because every address calculation in the parallel version will require a load of the PROCS variable and an additional add. This can slow down a routine that iterates over a grid by as much as 40%. So for this project all timings were done with PROCS "hard-coded."

4.2.2 Ghost updates

A number of functions are provided to update the ghost values. Some procedures, such as the tridiagonal functions, may only want to update one column of ghost elements, while other procedures may need all the ghosts to be updated. For a NxN skewed grid there are a total of $4(N/P)P = 4N$ ghosts. To update these ghosts each processor sends a message of $2N * (sizeof(\texttt{double}))$ bytes to each of its two neighbors. Note that this differs from the blocked case, where there are a total of only $4N/\sqrt{p}$ ghosts and each processor must send four messages. But while the blocked case is clearly faster, because ghosts are only updated four times per time-step, this advantage is outweighed by blocked layout's disadvantages.

Nonetheless, updating ghosts is a fairly heavy-weight operation. Because Split-C does not provide strided bulk operations (doing so would be possible but non-trivial) a significant percentage of the time spent updating ghosts comes from packing grid columns into a buffer. Figure 14 shows that although the CM-5 has the fastest communication, it is weighted down by its slow memory hierarchy. This is strong evidence in favor of adding strided bulk operations to the Split-C library. We can also see that the HP's large cache becomes advantageous, since after packing the buffer the `bulk_store()` can read directly from the cache.

Figure 14: Time to update ghosts (256x128, procs = 4)

4.3 Fast Fourier Transform

The dominating cost of the Navier-Stokes solver, and hence of the entire simulation, is the time required for the two 2-dimensional FFTs in the fast Poisson solver. To implement the 2-dimensional FFT we chose the row-column technique that uses 2N 1-D FFTs [**?**]. This allows us to use one of the many sequential, highly-optimized 1-D FFT routines that are available. For example, on the CM-5 it seemed clear that the vector units should be used to attack the FFT, since the SPARC nodes are rated at 10 MFLOPS while the vector units are rated at 128 MFLOPS. However, the VUs are notoriously difficult to program. Our solution was to build a library that allowed the nodal CM Scientific Subroutine Library to be called from Split-C. Appendix A provides further information on how this was done.

To provide data locality during the 1-D FFTs we must convert the skewed grid to the correct layout for each step. First the grid is converted to a row-layout, then local 1-D FFTs are performed on each row, then the grid is converted to a column-layout and local FFTs are performed on each column, and finally the column layout is converted back to skewed (figure 15). Each conversion step requires N2 communication. The next section examines this communication in more detail.

4.3.1 Communication

As with ghost updates, a blocked layout would result in slightly faster communication, essentially because a greater number of grid values would already be correctly positioned. For example, to convert from a blocked layout to a row layout would require:

$$\text{Block to row: } \sqrt{p}-1 \text{ messages, } \frac{N^2}{P} * \frac{\sqrt{p}-1}{\sqrt{p}} \text{ data}$$

```
SkewToRow( grid )
for row = 1 to N
  FFT( row )
RowToCol( grid )
for col = 1 to N
  FFT( col )
... local manipulation ...
for col = 1 to N
  FFT( col )
ColToRow( grid )
for row = 1 to N
  FFT( row )
RowToSkew( grid )
```

Figure 15: Fast poisson solver algorithm

While the skewed case requires:

Skew to row: $p-1$ messages, $\frac{N^2}{P^2} * (p-1)$ data

But, again, the time saved by a blocked layout is much too small to make up for time that would be lost during the tridiagonal routines.

Barriers were used to keep communication permutations separated during each of the layout transposes. But while this led to a 50% improvement in communication performance on the CM-5, it resulted in 60% lower performance on the SP-1 and 52% lower on the HPs due to the high cost of barriers on those machines. Therefore, we made the transpose code machine-dependent.

Figure 16 shows the total computation and communication time required for the two 2-D FFTs. The costs for the three machines are in accordance with their known characteristics, but using the CM-5 vector units dramatically improves performance.

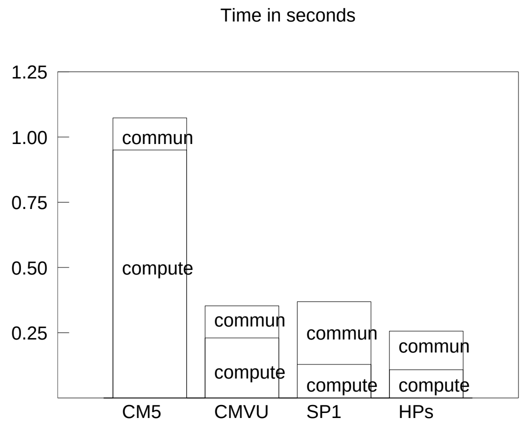

Figure 16: Total time for two 2-D FFT's (p = 4)

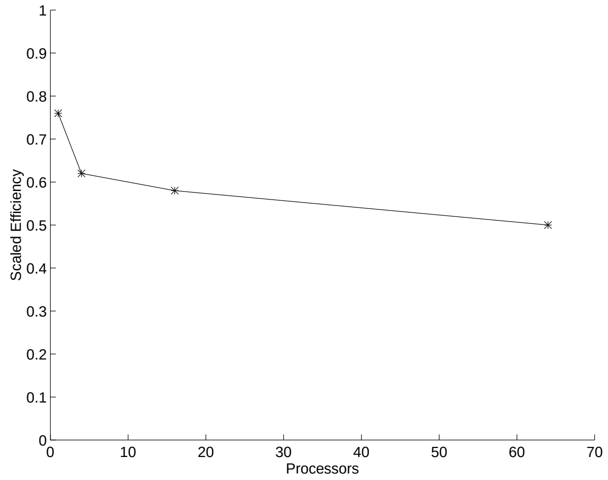

Figure 17: Scaled efficiency of 2-D FFT (CM-5, 128x64 per processor)

4.3.2 Resulting efficiency

As was done for the tridiagonal solvers, we found the scaled efficiency for the two 2-dimensional FFTs on the CM-5 without vector-units (figure 17). The 25% drop in efficiency for one processor is due to the layout conversion overhead. Although the conversions are not necessary in the one processor case, and do not require any communication, they are still expensive because of memory costs.

For four processors, and a grid of 256x128, the CM-5 obtains a speedup of 2.8 over the sequential code, the SP-1 obtains 2.7, and the HP reaches 3.41. The HP's speedup is superior because its larger cache pays off during both the communication and computation phases of the FFT. Using the vector units on the CM-5 results in an additional speedup of 2.8, for a total speedup of 8.4 over the sequential/non- vector version.

4.4 Total NS Speedups

Putting it all together, figure 18 shows the total time required for the parallel Navier-Stokes solver on 4 processors and itemizes where the time is spent. The performance ratio for the three machines is about 3.9:1.3:1.

Because of the FFT's dominance, the total scaled efficiency and speedups obtained for the Navier- Stokes solver are about the same as those achieved for the FFTs. The scaled efficiency for the CM-5 is shown in figure 19. For four processors and a grid of 256 x 128, the CM-5 provides a total speedup of 2.83, the SP-1 obtains a speedup of 2.8, and the HPs reach a

Figure 18: Profile of parallel Navier-Stokes solvers

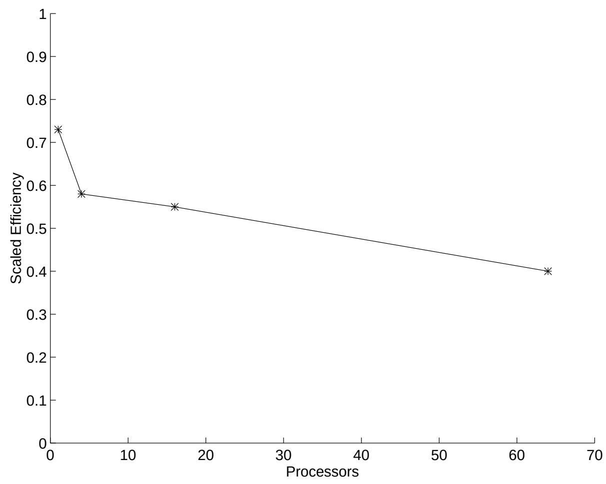

Figure 19: Scaled efficiency of NS solver (CM-5, 128x64 per processor)

speedup of 3.4

4.4.1 Lessons learned

To summarize the key lessons we have learned so far:

- We chose to use a skewed grid layout due to the wave-front nature of the tridiagonal solvers. This decision was motivated first by a simple model, and then by empirical measurements of two versions of the tridiagonal kernel. The model was helpful in choosing the layout, but poor at predicting execution time because it ignored memory access costs. This is problematic since the simplification's impact will vary across machines, and can therefore lead to the wrong layout choice for small problem sizes.

- The SP-1 was shown to be faster than the HPs for the tridiagonal routines, despite the HPs superior communication and computation speed. This was because of differences in the two machine's memory hierarchies, and points toward an important lesson for NOWs: Because workstations are designed for applications that exhibit locality, their memory system may be unsuited for some scientific applications.

- Providing a parallel grid data-structure substantially simplified conversion of the sequential code. However, in order to allow optimal performance, the grid's layout was not entirely hidden from the user.

- The high-cost of ghost updates demonstrated that communication speed is of little importance if we must first perform extensive data copying. This is strong support in favor of adding strided bulk operations to the Split-C library.

- The code for the Navier-Stokes solver was extremely portable, requiring only two significant changes when moved between machines. The first change was required due to inconsistent interfaces to the Split-C `atomic()` routine, while the second change, eliminating barriers, was strictly for performance.

Chapter 5

Cell operations

Lying on top of the grid are the elastic structures whose motion we are simulating. The exact nature of these structures will vary depending on what is being simulated, from simple polymers to cells. In the cases of the blood-platelet and epithelial cell simulations the relevant structures are cells, adhesives, and walls, all of which are built out of segments (figure 20). Cells are represented as polygons, while adhesives are segments used to simulate the binding that can occur between nearby cells. In the blood- platelet simulation the walls model artery walls, while for the epithelial cell simulation they simulate the mechanics of microscopy.

The cells, linked by adhesives, form a network that is split across p processors. Each processor owns a mutually-exclusive, linked-list of cells, where each cell is represented as a linked list of segments. The segment data-structure stores coordinate and elastic force information and points to an attached adhesive if one exists. An adhesive links together two cells by forming a connection between the closest segments. There is a distributed, linked-list of these adhesives, where the processor that owns an adhesive also owns at least one of the cells to which it points. The Split-C notion of global pointers allows adhesives to point to cells on remote processors

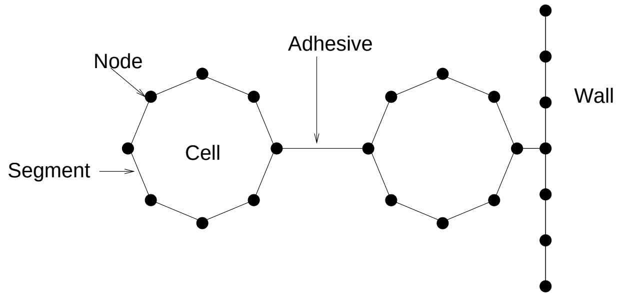

Figure 20: Terms for cell data structure

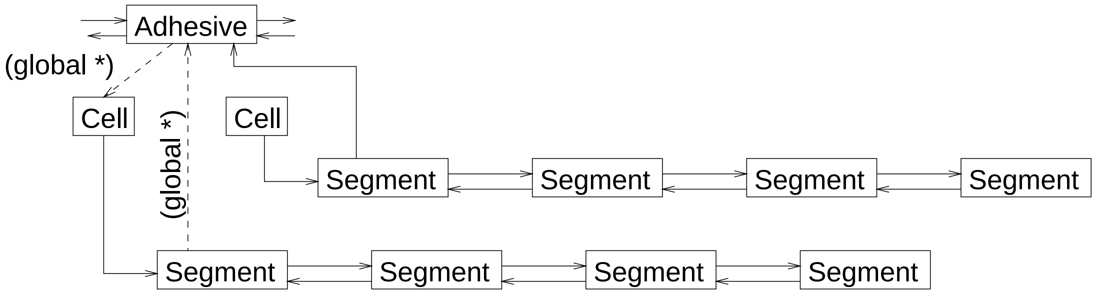

Figure 21: Terms for cell data structure

and allows the remote segments to point back to their adhesives (figure 21).

5.0.2 Overview of cell operations

This data structure can be complex to manipulate, so we provide a library of useful functions, from simple creation and deletion, to calculating area and mass, to partitioning. Because it is a parallel data structure, these functions must be careful to maintain global consistency. For example, two cells that are geographically close but on different processors may simultaneously attempt to form an adhesive. Without some form of locking, the result would be non-deterministic: only one adhesive would actually exist (whichever was linked-in last), but each processor would think it had created one. But outside of the RPCs required to implement these locks, only three operations on the cell data-structure require significant amounts of communication:

- `findNearbyWalls( )` - returns pointers to the first and last wall segment that are within dist of a cell's center of mass.
- `for_all_remote_pairs( )` - acts as an iterator, finding all pairs of cells within dist of each other, where exactly one cell is local.
- `partitionAllCells( )` - transfers each cell to the processor that owns the physical region the cell is currently in.

Because these three functions differ the most from the sequential code, and affect how cells should be distributed across processors, we measure and analyze their associated costs in sections 5.1.1 through 5.1.3.

There are two other cell operations that require communication, but because they involve both the cells and the grid we discuss them separately. In section 5.2.1 we describe Spread(), which extrapolates the force from each cell segment to surrounding grid points, and then in section 5.2.2 we discuss Move(), which performs the inverse operation. For both functions we describe a series of optimizations that were done, show how their efficacy was affected by different machines characteristics, and then discuss the resulting speedups.

It is important to keep in mind throughout the following sections that the frequency, and therefore total cost, of cell operations depends entirely on the particular problem. We have used the blood flow parameters, where there are only a few tens of cells and they are irregularly positioned, because that was the only real scenario available. Because of this, the costs of the cell-only operations in the sequential code were barely measurable, and even the costs of `Spread()` and `Move()` made up only 7% of application's total running time. These results may be very different for the epithelial simulations.

5.0.3 Methodology for determining layout

Determining how to partition the cells is difficult because we are faced with three, sometimes conflicting objectives:

1. To minimize scatter/gather communication, cells should be owned by the same processor that owns nearby grid points.

2. To minimize all_nearby_pair communication, cell partitions should be as large as possible.

3. To achieve good load balance, each processor should own roughly the same number of cells.

Although the first objective is best met with a skewed cell layout (so that the cells are aligned with the grid), the second objective seems to call for a blocked layout (figure 22). And while the smaller partitions of a skewed layout may seem to offer superior load balance, a blocked layout could have dynamic boundaries that adapt to cell clumping. Analytical methods are of little help in resolving these issues because cell movement is irregular and unpredictable.

Therefore, we initially implemented blocked cell-partitioning since it was the more straightforward of the two, but encapsulated the partitioning-specific code so that it could be easily changed. Sections 5.1 and 5.2 analyze those functions – `for_all_remote_pairs()`, `Spread()`, etc. – that are influenced by the partitioning scheme. This data, combined with rewriting key portions of the partitioning code to support a skewed layout, allow us to make some estimates in section 5.3 about the advantages of a skewed layout.

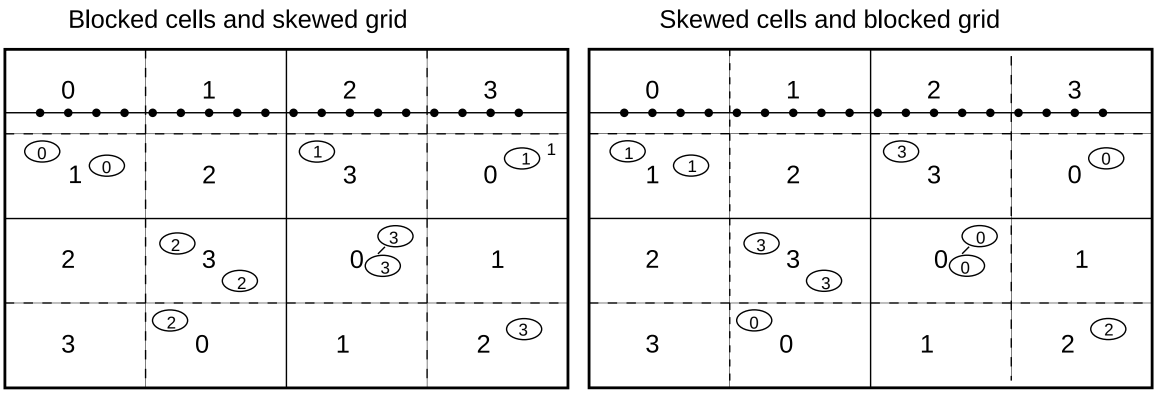

Figure 22: Two cell partitioning methods

5.1 Cell operations

As with any data-structure, our goal was to hide implementation details by exporting a general and efficient interface. However, while the parallel-cell data structure hides many of the messy details that accompany distributed-memory programming, the high cost of certain operations makes their distributed nature visible. The next three sections discuss operations that are expensive in the parallel version, while virtually free – or in the case of partitioning, non-existent – in the sequential version.

5.1.1 Operations with walls

In order to check if a cell overlaps a wall, or if an adhesive should be formed between a cell and wall segment, the wall segments nearby each cell must be checked. But to do so in the parallel simulation requires communication if some or all of the nearby wall segments are owned by a different processor than the cell. This can occur if the cell is at the edge of a processor boundary, or if the walls are partitioned differently than the cells. In order to allow the user to treat these cases uniformly, a library function, `findNearbyWalls( cell, dist, *global start_wall, *global end_wall)`, is used to obtain the global pointers to the first and last segment that are within dist of cell. These pointers can then be used normally, albeit at a much higher cost if they point to a remote location.

So parallelism results in two additional costs: the cost of `findNearbyWalls()`, and the cost of remote wall dereferences. `findNearbyWalls()` determines which processors own the relevant wall segments and then obtains the corresponding pointers using RPCs. The cost of this process, when both end-points are remote, is shown in figure 23. Note that the CM-5 is slower than the

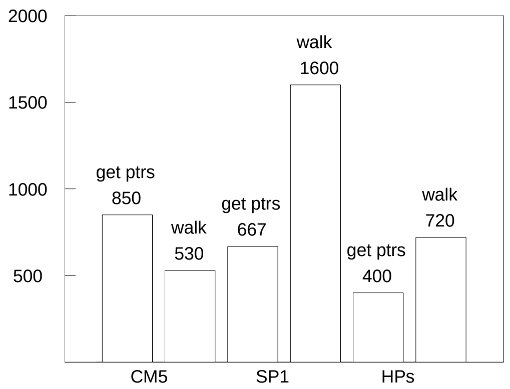

Figure 23: Operations between cell and remote wall

other machines because computation time (calculating wall coordinates, finding the pointer to return) is as significant as communication latency.

This, however, is not the case for walking through the returned linked list of ten wall segments. The cost for this operation is also shown in figure 23 and clearly exposes the different machines' latencies.

These numbers suggest that, at least in the case of the SP-1, it would make sense to have the wall segment information sent in bulk rather than piece-by-piece. But we have so far avoided doing so for two reasons. First, the current interface is the most general; it allows the receiver to easily and dynamically decide exactly which segment structures, and which pieces of the structure, should be read. Second, wall segments turned out to usually be owned by the same processor as nearby cells so the optimization promised little practical advantage.

5.1.2 Operations with cell-pairs

Just as it checks cells nearby walls, the application also checks all pairs of nearby cells to see if they overlap or if adhesives should be created. In the sequential code this is done with a naive $O(n^2)$ algorithm, where n is the number of cells and presumed to be small. In the parallel code, cell partitioning effectively sorts cells by location, reducing the number of pairs that need to be checked. However, checking cell-pairs that cross boundaries now requires communication.

The parallel version first checks all local cell-pairs, and then checks pairs where one cell is local and the other is in a partition above or to the right. This convention ensures that the set of pairs checked by each processor is mutually exclusive. Figure 24 shows a possible cell configuration and labels some of the cell-pairs with the processor responsible. (Note that for

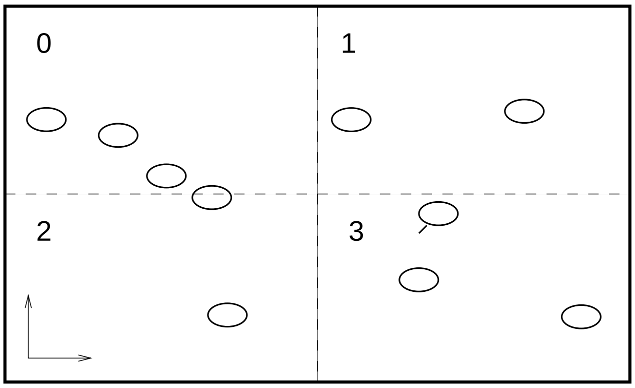

Figure 24: Checking remote cell-pairs

the purposes of attachments the grid is not considered periodic, so links can not wrap around grid boundaries.)

The cell data-structure provides the function `for_all_remote_pairs( pCell local_cell, pCell *global remote_cell, double dist )` that iterates through all pairs within dist. There are two pieces hidden within this iterator. First, it must determine which processors own partitions above, or to the right of, the local partition. This is done by checking every processor's region coordinates from a globally accessible array. While these remote-reads are expensive, they allow for dynamic boundaries. Then, the center of mass for each remote cell is read, and its distance from each local cell is calculated.

The user can create their own iterators using the provided pieces. For example, for the platelet simulation we created two variants of the remote pair iterator in order to provide better performance and slightly altered functionality. The first iterator only checks cells that meet a certain criteria ("unactivated"), while the second defines *dist* as `local_cell->Radius + remote_cell->Radius` rather than as a user parameter.

The cost of these iterators is primarily the cost of the remote reads used to obtain partition boundaries and remote cell's coordinates. We measured the time required for four processors to (successfully) check one remote cell against one local cell (figure 25). Each additional remote cell adds the cost of two remote reads.

5.1.3 Partitioning cells

The previous section showed that cell partitioning not only provides load-balance, but also geographic locality. Because our code relies on this property, the function `transferAllCells()`,

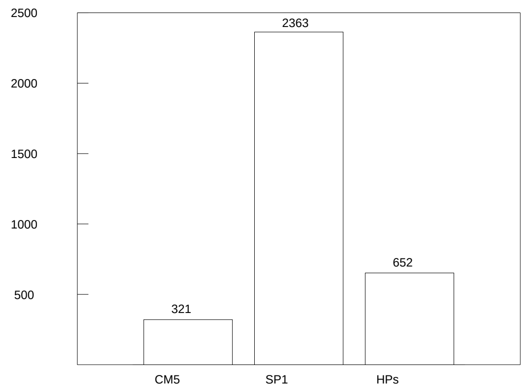

Figure 25: Time for one remote cell-pair

which transfers cells to the appropriate processor, acts as a synchronization point. This routine must be called at every time- step after cell positions have changed, and before any remote cell-pair operations occur.

The routine works by checking each cell's coordinates against the processor's local boundaries. If the cell is outside the region a temporary structure containing the cell's critical information is created and sent to the geographically appropriate processor. This tCell structure contains information on the cell's attributes as well as global pointers to variable-length lists of the cell's lines and adhesives. Once all the cells have been sent, the processors examine any tcells they received. For each tcell, two `bulk_reads()` are issued to obtain the corresponding lines and adhesives, and then a new cell is constructed from this information (figure 26).

Writing this routine illustrated how different machine characteristics can affect parallel code. Cells were originally sent with `bulk_store()`, followed by an `all_store_sync()` when all of the cells had been sent. While we assumed this would lead to improved performance by allowing communication and computation to be overlapped, the cost of the `all_store_sync()` on the

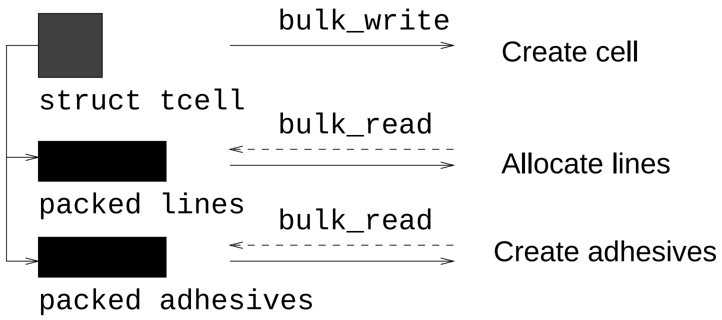

Figure 26: Transfer process

Figure 27: Time to send a cell

Figure 28: Time to receive a cell

SP-1 (750 μseconds!) proved overwhelming. Changing the `bulk_store()` to a `bulk_write()` resulted in dramatically faster communication on the SP-1 without affecting performance on the other machines (figure 27).

It takes much longer to receive a cell than to send one (figure 28). This is primarily due to expensive `malloc()` calls, as well as the searching and pointer manipulation required to insert the cell into the data structure's web of linked lists. Using custom memory allocation rather than `malloc()` could significantly improve performance. To some extent, these costs are a result of inheriting the local data structure from the sequential code, where insertions and deletions were assumed to be rare.

It is important to note that while we usually think of parallel overhead as only being communication, the entire transfer process is unique to the parallel version.

5.2 Interaction between cells and grid

The cells and grid interact during only two routines: `Spread()` and `Move()`. `Spread()` extrapolates the elastic forces from each segment to the surrounding sixteen grid points, while `Move()` performs the inverse operation, applying the force of the surrounding sixteen grid points to move each segment. Although these routines take up only a fraction of the running time required for the sequential and parallel blood-flow simulation, according to Charlie Peskin they make up one-third of the time required for a 3D heart simulation on a CRAY C90 due to the large number of heart fibers [?].

5.2.1 Spread

If the grid and cell partitions are not aligned – i.e., a processor can own a cell but not the surrounding grid points – communication will be required to spread the cell's forces. Even if the partitions are aligned, communication will be necessary for segments that lie within four grid-points of processor boundaries.

The initial parallel version of `Spread()` treated these two cases separately. It first went through the local cells and appended force information from those cells whose center of mass was on a remote grid point to a buffer. These buffers were then written to the appropriate processors with a `bulk_write()`. `Spread()` could then operate on both the cells that remained and the cells received by using atomics to increment remote points along the border. The costs for this are shown in figure 29a.

The problem with this method was that border points proved to be common, especially around the walls that stretched across the entire region. And atomics were inefficient, not only because of the send overhead, but also the receive overhead, which transforms every increment operation into dozens of instructions. Therefore, we modified the code to send cell information to any processor that owned a partition within four grid-points of the cell. Now each processor only modifies those grid points that are local and ignores the others (figure 29b).

Despite the amount of computation required for `Spread()` (almost 60 FLOPS per segment), the high cost of communication made parallel speedups difficult to obtain. With four processors, the CM-5 only achieved a speedup of .67, while the SP-1 reached .72. Only the HPs, which with their greater `bulk_store()` bandwidths were able to obtain a speedup of 1.35, showed an improvement over the sequential code. The scaled efficiency for `Spread()` on the CM-5 hovers

Figure 29: Time for `Spread()`

at around 65%; this result is approximate because the unpredictable motion of the cells makes it difficult to linearly increase the problem size. There is a clear drop-off in efficiency even for $p = 1$ due to the 4 μseconds required for every grid address calculation (see section 3.1.1). So in the case of the CM-5 at least, it may make sense to cache these computed addresses.

5.2.2 Move

`Move()` is the functional inverse of `Spread()` but must be parallelized slightly differently. Unlike `Spread()`, where cells can be pushed to the correct processor, `Move()` requires grid values to be *pulled* to the local processor. The initial version simply used gets to obtain remote grid point values and then issued a sync before adding the results to the cell's position. Unsurprisingly, this version performed very poorly on the high-latency SP-1 (Figure 30a). However, many of remote gets were redundant, so by checking a simple software cache before every remote get we were able to sharply reduce the communication costs (Figure 30b).

We then tried, as a third optimization, switching entirely to bulk communication. This was done by creating lists of remote grid coordinates, sending these lists to the appropriate processors, and then receiving back the corresponding lists of values. This process requires significantly more computation but allows latency to be amortized. Figure 30c shows that while this tradeoff makes sense for the SP-1 and the HPs, it leads to lower overall performance on the CM-5. This result is evidence that code that has been optimized for a low-latency MPP may not be best for a NOW, and vice-versa. However, this result only holds true for simulations where the number of remote grid values is small. For large enough n, using bulk operations may also be superior on the CM-5.

There is a subtle but important difference between the parallel versions of `Spread()` and

Figure 30: Time for `Move()`

`Move()`. `Spread()` pushes the data away if it lies on remote grid points, thereby distributing the computational load by grid partition. `Move()`, however, pulls in the data necessary for local cells, thereby distributing the load by cell partition.

So with the current static blocked cell partitioning `Move()` is profoundly unbalanced. This, combined with the lower computation requirements of `Move()` (in the sequential code `Move()` takes half the time of `Spread()`), results in poor parallel speedups. The four-processor CM-5 achieved a speedup of .65, the SP-1 only reached .34, and the HPs obtained .86.

5.3 Affect of cell partitioning

How cells are partitioned affects how often cells need to be transferred, how many remote cell pairs need to be checked, and how much communication is required for `Spread()` and `Move()`. For most scenarios, where cells move very slowly across the grid, it is the third affect that is the most important.

To measure the affect of skewed cell partitioning on `Spread()` and `Move()` we changed the Create() routine so that cells were created on the processor that owned the grid point at the center of the cell. By aligning the cell and grid partitions the number of remote grid points accessed in `Spread()` and `Move()` was reduced by about 95%. This significantly reduced the time required for the two functions. Figure 31 shows the combined time saved by using a skewed, rather than blocked, partitioning scheme.

The magnitude of these times, compared with the measured times of the cell operations from section 5.1, indicate that skewed cell partitioning is probably best for the blood-platelet simulation. We plan to rewrite the remote cell-pair iterators in the near future so that this

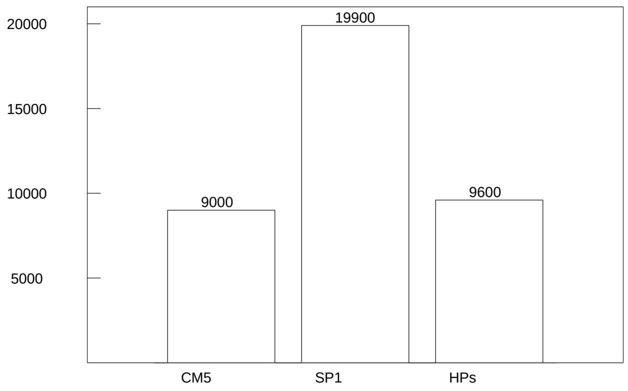

Figure 31: Time save with skewed cells

hypothesis can be confirmed.

5.4 Summary

The cell phase of the application is more difficult to parallelize efficiently than the Navier-Stokes phase because there is less computation, and communication is less regular. Both of these factors make communication latency – the parameter that varied most over the three machines – critical.

We showed with the `Spread()` and `Move()` routines that it is possible to amortize latency by combining sends and receives into a bulk-synchronous protocol at the cost of extra computation. When this trade- off makes sense depends on a machine's computation/communication ratio. We also showed that synchronization latency, which can not be amortized, affected how our program was written.

The implication of these results is that a program that has been optimized for a low-latency MPP may not be optimized for a NOW – and vice-versa. But by understanding where and when these situations arise it is possible to write code that works well for both types of computers.

Chapter 6

Conclusion

6.1 Total times and speedups

The time required for each time-step of the simulation was largely determined by local node performance. Figure 32 shows that the HPs are fastest and, even when using the vector units, the CM-5 is the slowest.

On all three machines about 70% of the running time was spent in the Navier-Stokes solver. So, not surprisingly, total speedups are similar to those reported for the Navier-Stokes solver alone (figure 33). The HPs obtain the best speedup of the three machines because of their high write bandwidth and because their large cache pays off during the FFTs. Using the vector units allows the CM-5 to reach a "speedup" of 4.1.

Finally, the scaled efficiency for the CM-5 (not using vector units) is shown in figure 34. Again, it roughly parallels the efficiency of just the Navier-Stokes solver, which had an imme-

Figure 32: Total time per time-step

Figure 33: Total speedup per time-step

Figure 34: Total scaled efficiency

diate 25% drop- off due to the layout conversions (section 4.3.2).

6.1.1 Related and future work

There has been little previous research on efficiently porting parallel programs. The work most similar to ours is [?], which analyzes a Monte Carlo simulation of carcinogenesis on a four-node CM-5 and four-node network of HPs. They achieved best overall performance on the HPs, however, they obtained significantly lower communication bandwidth on the HPs than on the CM-5. This was due to their use of very large messages that caused the buffers on the HPs to overflow, which in turn resulted in time- outs and re-transmissions. Although we did not observe this effect in our application, it points to one of the drawbacks of using a LAN for a multiprocessor network.

In another related paper, Demmel and Smith describe the parallelization of an atmospheric simulation and use a detailed model to predict the application's running time on a number of different machines [?]. It is interesting to note that a skewed grid layout turned out to be advantageous for that application as well [?]. The authors also showed with their model that I/O dominated running time. Although we have largely ignored I/O in this paper, it is a cost that does exist and should be studied further. Our cell simulation dumps data to disk at regular intervals so that a graphical "movie" showing cell movement can be constructed. It would be interesting to compare the performance of the CM-5's disk array with the software RAID possible with a NOW.

Another interesting extension to our work would be to port the application to machines such as the CRAY T3D or Intel Paragon that have very high bandwidths. While the three machines we examined have widely varying latencies, their bandwidths are all roughly equivalent. A machine with very high communication bandwidth might encourage even more complex data-packing schemes.

Finally, the most important future work will be to analyze this application on a larger NOW. It seems unlikely that performance on the HPs will scale as well as on the CM-5 since the FDDI LAN is essentially a shared bus. However, future NOWs may use ATM networks that provide better scalability. We hope to analyze performance on a 64 node NOW sometime next year in order to examine this issue.

6.2 Summary

There was a single, overarching goal for this project: to construct an application that allows biologists to investigate why cells move the way they do. But achieving this goal entailed three others. First, the application had to be able to quickly simulate realistic scenarios. This required the use of parallel computers and close attention to efficiency. Second, the application needed to be easy to modify so that biologists could use the same base application to test multiple hypotheses. This goal was met by hiding the complexity of parallelism with general, re-usable data structures. Third, the application could not be tied to a single machine since a supercomputer's useful life span is only a few years. This meant using a portable language and designing the program to take advantage of different machine characteristics. We believe that we successfully achieved all three goals.

Achieving good performance required choosing the optimal algorithm and data layout scheme. For this project we used analytical models to narrow the possible choices, and then timed multiple implementations of important kernels in order to determine the optimal method.

The second, empirical step is necessary because models are only approximate. This was demonstrated by our analysis of grid layout in section 4.1 where a model that ignored memory costs and communication contention led to incorrect layout choices for small N. We also used the two-step methodology in section five to choose between cell partitioning schemes. We first used analytical arguments in support of both blocked and skewed partitioning, and then, by timing the two dominant cell operations under both partitioning regimes, were able to conclude that skewed cell partitioning is superior.

The second goal, making the code easy to modify, was addressed by providing general and re-usable data structures that hid the complexities of distributed memory. The two main data structures we provided were a grid and a network. The grid data structure hid the complexity of partitioning and presented the illusion of a single, global grid. However, the user could still take advantage of how data was laid out for optimal performance. Similarly, the network of cells data structure was able to largely hide partitioning and consistency issues from the user. But, because the added latency of remote operations can not be hidden, some information about locality was made visible to the user. Both of these data structures were relatively straightforward to implement with Split-C. The language's notion of spread arrays allowed for skewed layouts to be defined and operated on efficiently, while the concept of global pointers provided the necessary mechanism for the irregular cell data structure.

Split-C was also invaluable for achieving the final goal, portability, because the language is available on many platforms and its syntax and semantics are well defined. We were able to port the application to the SP-1 and network of HPs without making any changes to the original CM-5 code. However, optimizing the application to fit the different machines required two types of changes. First, the efficacy of bulk operations varied across machines. It is a standard optimization to amortize latency costs by packing data into a buffer and sending it in bulk. However, while this always led to better performance on the SP-1 due to its fast computation and high-latency, in one instance it led to slower performance on the CM-5. The second portability pitfall is with synchronization primitives. Programs designed on MPPs such as the CM-5 often implicitly rely on the low-cost of synchronization; using `barrier()` as a form of communication flow-control, or using `bulk_store()` followed by an `all_store_sync()` even when there is little computation to overlap. These programs will perform poorly on NOWs since synchronization exposes the full latency of a network.

The bottom line is that the application currently works and achieves reasonable speedups on all three machines. We are currently working on porting the application to a larger NOW and to a state-of-the- art MPP, the CRAY T3D. But most importantly, biologists at UC Berkeley have begun to use the simulator. How useful they find the program – what knowledge is gained – will be the final arbiter of whether we achieved our stated goals.

Acknowledgments

I am indebted to Kathy Yelick for extensive advice and guidance during this project. George Oster and Lance Davidson helped me in understanding the goals and theory of the application. Jun Yang did the initial conversion from FORTRAN to C. Additional assistance was provided by Remzi Arpaci, Eric Brewer, Arvind Krishnamurthy, Lok Tin Liu, and Richard Martin.

6.3 Introduction

The difficulty of programming the CM-5 vector units is well known. Although a number of languages have been designed in an attempt to hide the complexity of the vector units (such as AC and CDPEAC), obtaining good performance with the VUs still requires an extremely detailed understanding of the hardware and an almost obsessive attention to data layout. The result has been that most programmers forgo using the vector units altogether.

Thinking Machines partially addressed this issue with the CM Scientific Subroutine Library (CMSSL), a library of mathematical routines such as matrix-multiply and Fast-Fourier Transform that have been carefully engineered to take full advantage of the CM-5 and its vector units. Unfortunately, the library is intended only for use with CMF and, to a lesser extent, C*, data-parallel languages with sophisticated run-time systems that support the notion of "parallel arrays" that live in vector space.

In this appendix I describe how CMSSL can be called from two different SPMD languages, C with message passing and Split-C. While conceptually a simple task, the details of getting CMSSL and non- data-parallel languages to cooperate proved to be difficult. I will describe some of the subtle problems but will primarily give a high-level overview of the method; those interested in the details can consult the source code of the examples. In section two of this appendix I sketch the architecture of the vector units and explain how arrays are mapped into their memory. Then, in section three I describe how to call nodal CMSSL and point out a few of its limitations. I conclude by drawing a few lessons and pointing toward future work that could be done.

6.4 VU Architecture

Each node of the CM-5 has six main components: the CPU (currently a SPARC processor), the network interface, and four vector units (figure 35). The vector units act as both memory controllers and floating- point units. This scheme offers very high memory bandwidth, about 500 MB/sec per node, but is also the cause for much of the difficulty inherent in programming the VUs. Because each VU can only address its associated memory unit, data layout becomes critical. Also, because each access is uncached, scalar performance of the VUs is poor. This leads to time-consuming swapping of data between scalar and vector memory.

Vector memory is mapped into virtual memory as shown in figure 36. A single physical

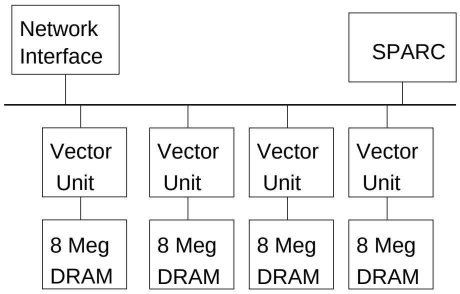

Figure 35: One node fo the CM-5

address can be accessed by either its instruction address, which triggers the execution of an instruction in the associated vector unit, or by its data address. For now we will only be concerned with data addresses. The data address space can be broken down further: there are heap and stack segments (each grows upwards), and these in turn are broken down into regions for each VU and combination of VUs. The segments shaded gray can only be written to, this allows a write to be broadcast to multiple VUs with a single instruction.

Given this memory architecture, the problem now becomes how to best map an arbitrary array into vector space. In order to achieve peak performance we are faced with two main constraints: we must divide the array as evenly as possible so that each VU has an equal amount of data and we must minimize communication between VUs. The latter is necessary because data movement within a VU is faster than between VUs, which in turn is faster than data movement between nodes. The CM Run Time System (CMRTS) adds a further constraint: the size of the data owned by each VU must be a multiple of eight.

Suppose we have an 8 x 12 array that we wish to map onto a machine with four nodes, as shown in figure 37. We must divide the array into sixteen equal subgrids, each with a size that is a multiple of eight. This is impossible to do for an 8 x 12 array so we must add garbage

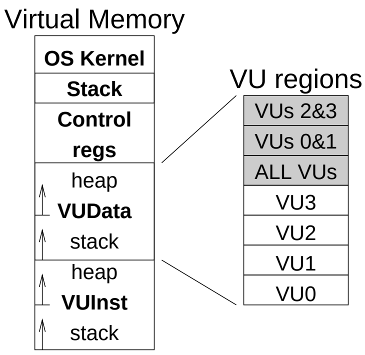

Figure 36: Vector memory

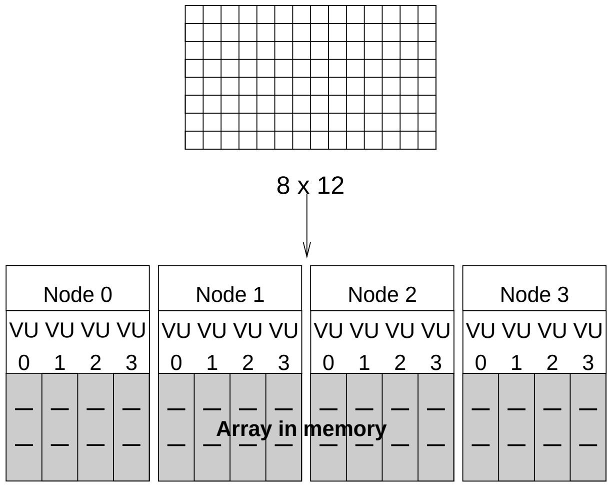

Figure 37: Mapping an array

positions (array elements that do not contain useful data) to pad axis one to 16. The resulting array could then be laid out a number of ways, two of which are shown in figure 38. Notice that in the right example axis 0 is serial — all positions along the axis are located on the same VU — so communication will be fastest along this axis.

Another layout decision must be made concerning the numbering of the vector units. In figure 38 the vector units could be numbered so that the horizontal axis varies fastest (so node 0 would own the first top-four subgrids), or so the vertical axis varies fastest (node 0 would own the left-most four subgrids). CMRTS will attempt to make these layout decisions optimally, but to achieve peak performance with CMSSL the programmer must be prepared to control the layout decisions explicitly. Fortunately, this can be easily done and is described in the following section.

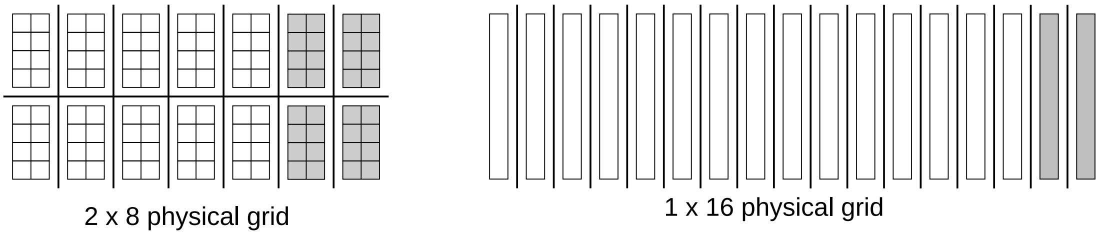

Figure 38: Mapping an array

6.5 Nodal CMSSL

Nodal CMSSL provides a model slightly different than the one I have described. Operation still follows a data-parallel model, but now each node acts as the host while the vector units are the PEs. This allows each node to operate independently which more closely resembles the SPMD model of Split-C and C with message passing. A nodal CMSSL routine is called with arrays that live entirely in the associated four vector units. For some applications, where each node operates independently, this model works very well. For other applications, where the size of the data is large enough so that it makes sense to use the power of the entire machine, this model is less appropriate. But in some cases, such as matrix-multiply or 2-D FFT's, it is possible to use blocking to effectively perform global operations with nodal CMSSL. We have used this last method in a Navier-Stokes solver.

6.5.1 Calling nodal CMSSL

There are three main steps to using nodal CMSSL from C: allocation of arrays, manipulating those arrays, and then actually calling the CMSSL routine. Allocation of vector arrays is done using the primitives provided by the CMRT library. After a few required steps to initialize the run-time system, a geometry that describes the desired layout of the array is created and then passed to the allocation routine. This routine returns a pointer, of type `CMRT_desc_t`, that completely describes the layout and position of the array. A few of the fields in this descriptor (`geometry_order_offsets_ptr`, `axes_layout_maps_ptr`, and `axes_extents_ptr`) must be modified so that it can be understood by CMSSL. I have implemented a library of routines that allows allocation of parallel arrays to be done from C while hiding most of details. These routines support a range of control over the array geometry, from basic to very detailed.

Reading and writing parallel arrays is awkward for two reasons: first, determining the address for an element depends on the layout and can be complex to calculate. Second, because vector memory is uncached, each access is two to four times more expensive than a normal memory access. Therefore it is best to read and write in bulk using the CMRT routines:

```
CMRT_read_array( CMRT_desc_t dest, double *src )
CMRT_write_array( CMRT_desc_t dest, double *src )
```

These allow an entire array to be copied out of or into vector memory.

Finally the CMSSL routine may be called, but the programmer must be careful to use

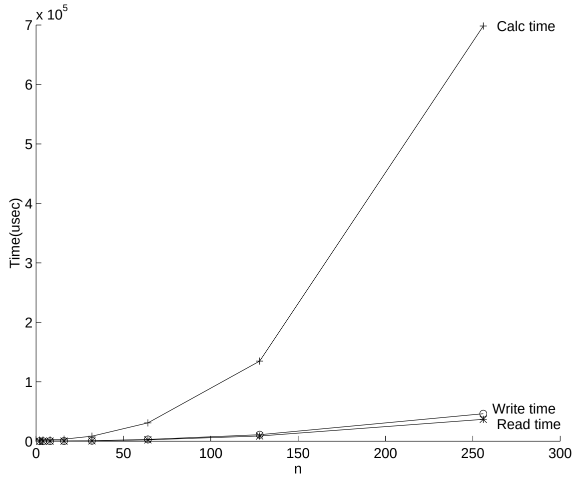

Figure 39: Running time for CMSSL matrix multiply

FORTRAN calling conventions. This means that an underscore is appended to the function name, all parameters must be passed by reference, arrays must be passed in column-major order, and a length must be passed following any string. The following shows how the FFT routine is called:

```
fft_(a_desc, "CTOC", &type, &fftsetup, &ierr, 4);
```

6.5.2 Performance of nodal CMSSL

Nodal CMSSL offers the same performance whether called from CMF or Split-C, the only difference lies in the overhead costs of manipulating vector memory. This overhead is implicit and unavoidable in CMF but is explicit when CMSSL is called from Split-C. The magnitude of the overhead determines when it makes sense to call CMSSL.

Figure 39 and table 6.1 show the break down of time spent manipulating vector memory and time spent performing a N x N matrix multiply. It is important to realize that these times are averages over many iterations — the first occurrence will be significantly more expensive due to initialization and set-up time.

The lesson from these numbers is clear: manipulating vector memory is a significant cost and should be minimized. We can think of vector memory in terms of a new level in the memory hierarchy, and the same rules that apply to optimizing programs to extract maximum cache reuse apply here as well.

One use of nodal CMSSL where the time spent performing memory manipulation is par-

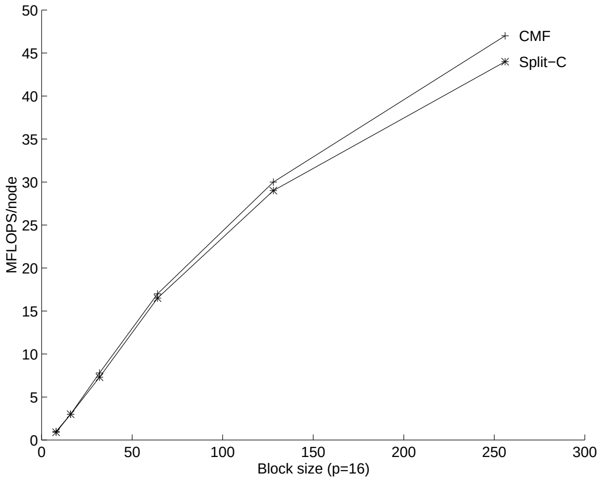

Figure 40: Comparison between CMSSL called from CMF and Split-C

ticularly important is when nodal CMSSL is used for global operations. This is done using *blocking*, an array is split into blocks that are then transferred between nodes using message passing. I implemented blocked matrix multiply using nodal CMSSL and initially found it to be fairly inefficient. The problem was that CMMD is extremely slow at transferring portions of vector arrays (it first creates a new array, copies the original portion in, and then sends the data.) However, Jean-Marc Adamo modified the code so that the lower-level CMAML library was used and this dramatically improved running time.

Figure 40 compares the speed of this blocked CMSSL matrix multiply against the global CMSSL routine called from CMF and shows that the global CMSSL version is faster by only about 10%.

N	Write (μsec)	Calculate (μsec)	Read (μsec)	% overhead
2	33	2024	42	3.57%
4	48	2135	65	5.03%
8	85	2342	126	8.26%
16	211	3524	302	12.71%
32	671	8468	877	15.46%
64	2243	30763	2942	14.42%
128	8814	134791	10912	12.77%
256	36723	698328	46262	10.62%

Table 6.1: Breakdown of time for CMSSL matrix multiply

6.6 Conclusion

This appendix has described how to call CMSSL from Split-C, as well as showing why *and when* it is a good idea. Using CMSSL is an easy way to harness the power of the vector units and thereby achieve significantly higher floating-point performance on common mathematical routines, from matrix-multiply to LU to FFT.

Nodal CMSSL, where each node operates independently, can be used quite easily with the help of the library I have built. There are some initial costs, but for problems of moderate size calling CMSSL should result in far higher performance than could be achieved with the SPARC CPU alone, even with hand tuned assembly. However, the programmer must be aware of the expense of transferring data between vector and scalar memory.

Further work could be done on making CMSSL even easier to use from C and Split-C. Wrappers could be written for each function that hide the details of FORTRAN calling conventions. And, ideally, Split-C should be able to call global CMSSL as well (although TMC claims this is impossible due to memory policies). But in general, I believe this work marks a significant advance toward routinely attaining super-computer performance from Split-C on the CM-5.