Performance Modeling and Composition: A Case Study in Cell Simulation

Steve G. Steinberg, Jun Yang, and Katherine Yelick
Computer Science Division
University of California at Berkeley *

Abstract

We present a case study in the use of performance modeling for parallel application development, with a biological cell simulation as our target application. We show that a simple performance model is adequate for determining data layout for arrays and linked structures, and validate our model against experimental results for some application kernels. We quantify the importance of optimizing across program components using information about machine performance and input characteristics. The cell simulation application has two phases, one regular and one irregular. The model closely predicts actual performance within the regular phase and allows for qualitative design comparisons in the irregular one. The resulting application is written in Split-C and runs on multiple platforms.

1. Introduction

Judicious choice of data layouts to balance computation and minimize communication is critical to the performance of parallel programs. Even for regular computational kernels, determining the optimal layout may require detailed performance models [10] or experimentation [6]. For large parallel programs composed of irregular and regular kernels, experimenting with several implementations is not feasible.

In this paper, we apply a simple latency/bandwidth model to the design of a cell simulation application, which has a regular phase that computes over a two dimensional rectangular grid and an irregular phase that computes over a linked structure. We consider two variations on the model, one using constant floating point performance for the machine, and another using a measured rate taken from the sequential program for each major computational kernel. We validate the model against experimental data and use it to evaluate several designs.

This paper is an application study for a large real-world application, but is also used to evaluate the requirements for systems that attempt to automate parallelism or data distribution. We show that optimizations for parallelism and locality must be considered simultaneously using machine performance characteristics; that optimal performance of a combination of program components is often obtained using suboptimal components; and that changing the basic order of dependencies using high-level semantics may also be necessary. A simple abstract machine model addresses these problems. For the regular phase of the application, we show that the model is sufficient for deciding between a set of parallelizations and data layouts, such as blocked, column, or skewed, and is also effective in optimizing across program components. For the irregular phase, we extend the model with a small amount of information about the input, and then use it to correctly make some basic design decisions.

The paper is organized around the cell simulation program, implemented in Split-C [8, 17]. Section 2 provides an overview of the application, and section 3 describes the performance model. We explore parallelization of the regular phase in section 4, the irregular phase in section 5, and the overall program in section 6. We present some related work in section 7, and conclude in section 8.

2. Application Overview

Biologists use computational models of bodies immersed in an incompressible fluid to help understanding blood flow in the heart [16], the growth of embryos [18], platelet aggregation during blood clotting [12], sperm motility [12], and other biological phenomena. This simulation technique, known as the *immersed boundary method*, was first developed by Charles Peskin to model blood flow in the heart in order to aid the design of artificial heart valves. The simulation's key concept is to model the system as a network of elastic fibers immersed in an incompressible fluid.

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contracts DABT63-92-C-0026 and F30602-95-C-0136, by the Department of Energy grant DE-FG03-94ER25206, by NSA grant NSF FD92-20719, and by the National Science Foundation grants CCR-9210260, CDA-8722788, and CDA-9401156. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

The fluid is modeled with a dense, rectangular, periodic grid. Cells are modeled as polygons, whose vertices represent immersed boundary points, and edges represent elastic fibers which join these vertices.

There are six main operations at each time step of the simulation. (1) Calculate forces on fibers according to their current locations and elastic properties. (2) Extrapolate forces from fibers to surrounding grid points. (3) Solve the Navier-Stokes equations over the fluid grid. (4) Interpolate the fluid velocity back to fibers and move them to new positions. (5) Perform interactions between cells. (6) Compute statistics about the simulation and produce output.

3. Performance Model Overview

Our model uses three machine parameters: MFLOPS, millions of floating point operations per second; α, the latency and start-up time to send a message; and β, the inverse of bandwidth, or the cost per double word of a message.

We base most of our analysis on a CM-5 with estimated 2.0 MFLOPS per processor, $\alpha = 80\mu sec$ per message, and $\beta = 1.8\mu sec$ per double. Communication parameters are based on the measured performance of Split-C bulk store operation, with message sizes greater than 16 doubles.

4. Navier-Stokes Solver

The computational core of the cell simulation is the Navier-Stokes solver. At each time step, discrete versions of the incompressible Navier-Stokes equations 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 [13]. The Navier-Stokes solver consists of the following four major steps:

- Tridiagonal solver (read U; read/write F)
- Divergence calculation (read F; write Re(P))
- FFT-based Poisson solver (read/write Re(P), Im(P))
- Final step (Read F, Re(P); write U)

where U is the $N \times N \times 2$ fluid velocity array, F is the $N \times N \times 2$ force array, and Re(P) and Im(P) are the $N \times N$ real and imaginary components of the pressure array.

4.1. Tridiagonal Solver

The tridiagonal solver operates in a series of wave-fronts: *tridgx* sweeps toward positive x and then negative x, and *tridgy* sweeps toward positive y and then negative y. An obvious choice for the grid layout in *tridgx* is a row layout, which has perfect parallelism and no communication. Likewise, a column layout is the ideal choice for *tridgy*. However, when these two are combined, an expensive remap is required. To remap an $N \times N$ grid on p processors, every processor must send N^2/p^2 data to each of the other $p-1$ processors. We therefore look for a single layout to avoid this expensive remap.

To maintain the full p-degree parallelism, we consider a skewed layout, in which each processor owns p blocks of size N^2/p^2 arranged in diagonals. Each sweep now has p stages of communication, and in each stage every processor sends a message of size N/p to the next.

To further reduce communication, we consider a blocked layout, with only $\sqrt{p}$ blocks in each dimension. In each sweep, a total of only $N\sqrt{p}$ data is sent across block boundaries, compared to Np for the skewed layout. However, this layout provides only $\sqrt{p}$-degree parallelism for both computation and communication.

Which layout is superior depends on the values of N and p, the machine's performance parameters, and relationship to other program components. We build a model based on the discussion above, along with machine constants. Figure 1 shows the results for $N^2/p = 2^{12}$, a typical value of the problem size per processor. For $p < 128$, the skewed layout is preferred, while for larger values of p the row/column layout is better. In contrast, on an IBM SP-1 with higher α, blocked layout outperforms both skewed and row/column for $p > 64$, because it uses fewer messages.

To confirm the results of the model, we compare the predicted performance of blocked and skewed layouts for *tridgx* against implementations. On a 64-processor CM-5, the model underpredicts the running time by 60%, primarily due to an overestimate of floating point performance. The tridiagonal computation performs roughly one memory access per floating point operation, and most of these accesses are uncached. Replacing the MFLOPS rate with a value of 1.2, measured from the sequential code, the model comes within 18% for the skewed layout and 30% for blocked.

4.2. Fast Fourier Transform

The dominating cost of the Navier-Stokes solver is the time required for two 2-D FFTs. To implement the 2-D FFT, we may choose a row-column technique [4] with $2N$ 1-D FFTs, using a row layout for the row FFTs and a column layout for the column FFTs. Each processor performs 1-D FFTs on its local rows, followed by a global remap, and then 1-D FFTs on its local columns. After some local computation, the reverse is carried out with inverse FFTs. Computation for each 1-D FFT is $\frac{1}{p}N^2 \log_2 N$ per processor, and the only communication is the row-to-column and column-to-row layout transposes, in which each processor sends N^2/p^2 data to each of the other $p-1$ processors.

An alternative is to let $q = \sqrt{p}$ processors cooperate to perform each 1-D FFT simultaneously. Starting with bit-reversed data, we first use a blocked layout so that the first $\log_2 \frac{N}{q}$ stages of the FFT butterfly operation are local. Then we remap data into a cyclic layout so that the latter stages are local as well. Inverse FFTs can be done in the reverse

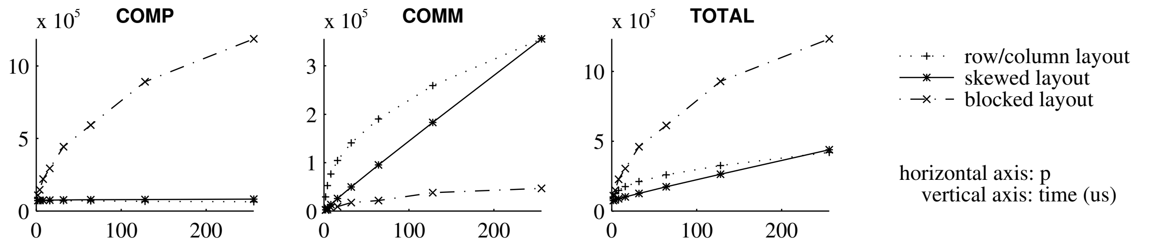

Figure 1. Modeled performance for the tridiagonal solver on the CM-5.

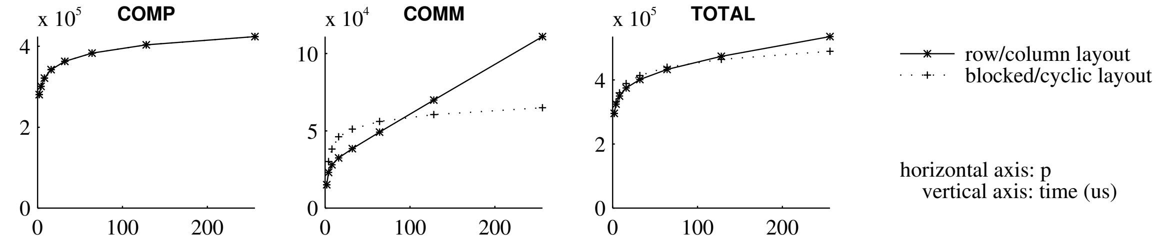

Figure 2. Modeled performance for the FFT-based Poisson solver on the CM-5.

order. The computation cost is the same as before, but the communication cost is the cost of four blocked-to-cyclic or cyclic-to-blocked remaps, one for each of the four 1-D FFT phases. Each processor sends a total of $q-1$ messages and $\frac{q-1}{q}\frac{N^2}{p}$ data. Since $q=\sqrt{p}$, each remap requires a smaller number of messages and less bandwidth than the ones in the row/column layout scheme.

Modeled performance of the complete Poisson solver is shown in figure 2. Again, we fix the problem size per processor to be 2^{12} and vary p. Because of the $\frac{1}{p}N^2\log_2 N$ computation requirement for FFT, computational load per processor increases with p. For the values of p available, the row/column layout scheme performs better than blocked/cyclic because it has only two remap phases and communicates less data overall. However, as p increases, the latency term dominates, and the blocked/cyclic layout scheme, with its $O(\sqrt{p})$ number of messages, becomes better than row/column with $O(p)$ latency.

To validate our model, we compare the predicted running times with the actual times of a row/column implementation on the CM-5. This time, 2.0 MFLOPS are too low for the highly optimized FFTs, and the total running time for $p=64$ is overestimated by 26%. When we replace the MFLOPS value by 2.8, as measured on the sequential kernel, the model closely matches the actual performance with only 5% error.

4.3. Putting Together the Navier-Stokes Solver

The rest of the Navier-Stokes solver involves mostly local manipulations. Some are embarrassingly parallel, and others are nearest-neighbor computations on the grid. We use ghosts on partition boundaries and include their update costs in our model. A blocked layout provides the lowest communication cost in this phase, because it has only $4N\sqrt{p}$ ghost values, as opposed to $2Np$ for row or column layout and $4Np$ for skewed.

With a reasonably accurate performance model of each component, we now look for the most promising overall layout strategies for the complete Navier-Stokes solver. First of all, if we choose a blocked/cyclic layout for the FFTs, two remaps of Re(P) are unavoidable, since only the FFTs use bit-reversed data. To reduce communication, the blocked layout for the tridiagonal solver is a reasonable choice. The remap of Re(P) between bit-reversed order and normal order can each be implemented in two phases: First each processor row cooperates to reverse all rows; then each processor column cooperates to reverse all columns. This remap has an $O(\sqrt{p})$ latency term, lower than most remaps of other kinds.

If we use the row/column layout scheme for the FFTs, there are many possibilities. The tridiagonal solver may use a blocked or a skewed layout, with Re(P) remapped before and after the FFTs. Although the blocked layout is not as good as skewed for the tridiagonal solver, combining it with row/column FFTs is less costly, since a greater number of grid values would already be correctly positioned. If the tridiagonal solver uses row/column layout, things become more interesting. The remap of Re(P) after the FFTs can be saved, since the *tridgx* that follows continues to use the row layout. However, because the tridiagonal solver takes U and F in the row layout and leaves them in the column layout upon exit, they would need to be converted back to the row layout before the final step. Fortunately, *tridgx* and *tridgy* are commutative; the 1-D FFTs on rows and the 1-D FFTs on columns are also commutative. With some

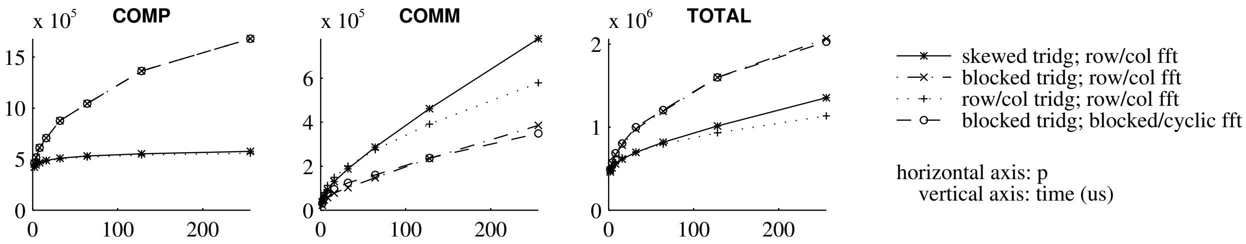

Figure 3. Modeled performance for the complete Navier-Stokes solver on the CM-5.

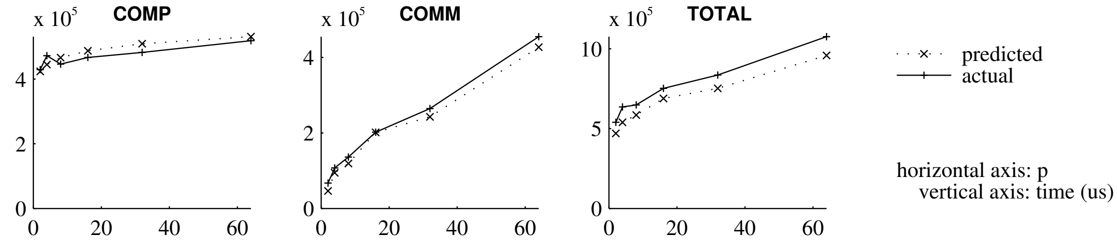

Figure 4. Predicted vs. actual running times on the CM-5 for the complete Navier-Stokes solver.

global reorganization of these components and unrolling the simulation loop, we obtain a scheme where the grid layouts alternate every time step between row and column, and these remaps can be avoided altogether. This is an example of how a suboptimal layout for a component can become advantageous when combined with other components. This transformation would require a high-level understanding of the algorithms.

We present the performance model for the complete Navier-Stokes solver in figure 3 with $N^2/p = 2^{12}$. Two layout schemes—row/column tridiagonal solver with row/column FFTs, and skewed tridiagonal solver with row/column FFTs—are close in the low range of p. However, as p exceeds 32, the row/column scheme begins to demonstrate a clear advantage. For the blocked layout schemes, the communication time they save is too small to make up for what is lost during the tridiagonal solver.

To validate the model, we implemented skewed layout for the tridiagonal solver and row/column layout for the FFTs. Figure 4 compares the actual performance of the complete Navier-Stokes solver on the CM-5 with the performance predicted by our model, again with $N^2/p = 2^{12}$. The MFLOPS value of 2.0 happens to describe the Navier-Stokes solver quite well as a whole, and the model provides a fairly accurate prediction of the actual running time.

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, but in the case of blood platelet and epithelial cell simulations, relevant structures are cells, adhesives, and walls, all built out of segments or groups of segments linked together as polygons. The cells, interconnected by adhesives, form a network that is partitioned among p processors, with global pointers to link data across processors. In the following discussion, we use the term *cell layout* to refer to the mapping of cells to processors according to their centers of mass. Thus, a row layout of cells means that one processor owns all of the cells centered in the upper p^{th} fraction of the domain.

5.1. Cell/Cell Interaction

In platelet simulation, cells interact with each other by forming and breaking adhesives. Adhesives are created for pairs of cells whose segments are within a given formation distance, and are broken when they move beyond a given breaking distance. Breaking adhesives is inexpensive and can be implemented by independently traversing local adhesives. Forming adhesives requires testing pairs of cells and segments and therefore fairly heavy computation. Our current implementation uses a naive $O(n^2)$ algorithm, where n is the total number of cells.

We use a model that accepts any distribution of cells and counts the numbers of local and remote cell pairs under different layouts. Total running time is estimated using the average processing cost obtained through measurement. Although our model can only provide a crude estimate of the actual running time, it is still helpful in determining layout strategies.

Predicted running times for cell/cell interaction under blocked, skewed, and column layouts of the cells are presented in figure 5. The input to the model is typical in a platelet simulation, with $N^2/p = 2^{12}$ and $n/p = 20$. The cells are randomly distributed in a blood vessel which runs

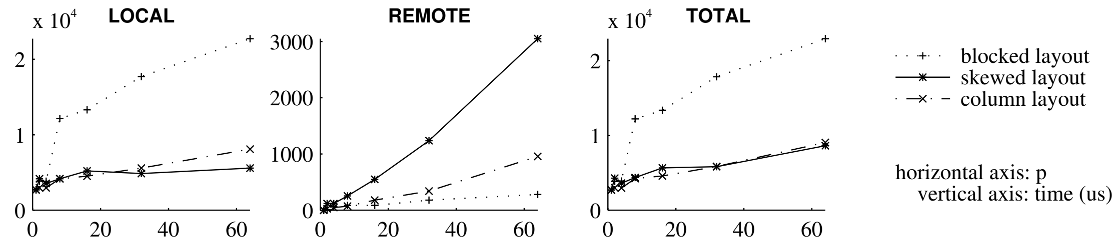

Figure 5. Modeled performance for cell/cell interaction on the CM-5.

horizontally across the center of the fluid grid. Because cells are not uniformly distributed, skewed layout offers better load balance than blocked. Due to the particular structure of the vessel, column layout also provides good load balance. Blocked layout has a lower surface to volume ratio and therefore requires less communication for typical problems, whereas skewed is the worst. The model shows that on the CM-5, local interaction is the dominating cost, so load balance becomes all-important. Overall, the blocked layout is clearly inferior, while skewed and column are very close.

5.2. Cell/Fluid Interaction

Cells interact with the underlying fluid grid during two routines: *spread* and *move*. *Spread* drives the fluid flow by extrapolating elastic forces from each cell segment to its surrounding sixteen grid points in F, while *move* performs the inverse operation, moving each segment by interpolating the fluid velocities at the surrounding sixteen grid points in U. Cell/fluid interaction involves the interplay between the layouts of the irregular cell structure and the regular grid structure. These two layouts can be aligned, i.e., a processor that owns a section of the grid also owns all cells centered in that section, or they can be unaligned.

Aligned layouts have lower communication costs than unaligned, although even with an aligned layout, some of the 16 updates/reads on the grid may be remote. By the usual surface-to-volume-ratio argument, a blocked layout for the fluid grid with an aligned cell layout requires less communication than other alternatives, but it does not effectively balance the work load.

When cell and grid layouts are unaligned, computational load may be distributed according to either the cell or the grid distribution. In “owner-computes” terminology, either the cell owner or the subgrid owner may perform the computation. Load distribution by cells may be better balanced with judicious choice of cell layout, but distribution by grid requires less communication, since transferring a segment is less expensive than accessing 16 remote grid points.

Modeled performance for cell/fluid interaction is shown in figure 6 with $N^2/p = 2^{12}$ and $n/p = 20$. The model reveals that load balance is, once again, the most important factor. Blocked cells, even when aligned with a blocked grid, perform poorly. Unaligned layouts are also costly as shown by the plots for column cells with row grid. Not only do the communication costs go up dramatically, but also the computational overhead. When layouts are not aligned, whether to distribute load by cells or by grid depends primarily on the load balance they offer. In the case of column cells and row grid, because cells are more evenly distributed under the column layout, it is better to go by cells, despite slightly higher communication cost.

Due to the irregular nature of cell operations, it is difficult to validate our model quantitatively, but we can still do so qualitatively. We have implemented *spread* by distributing load by grid, and *move* by distributing load by cells, for a blocked cell layout and a skewed grid. As predicted, *move* is profoundly unbalanced because of the blocked cell layout. By changing the application’s cell creation routines, we have also experimented with situations where cells are aligned with the grid, and found that 95% of the remote grid point accesses can be avoided, confirming the value of alignment.

5.3. Putting Together the Cell Operations

There are other operations in the irregular phase, such as force calculations on the cells, which mainly involve local data. In addition, aligned layouts require communication to transfer cells across processors as they move during the simulation. The cost of this communication is highly dependent on the type of the simulation and the speed and direction of the fluid flow. Once again, blocked layout has lower communication costs but worse load balance.

According to our model, load balance is the most important factor in the performance of cell/cell interaction, cell/fluid interaction, and most of the other cell operations. Therefore, the preferred layout for cells would be column (best combined with column grid) or skewed (best combined with skewed grid), as they both offer fairly good load balance. The result may be quite different for other patterns of cell distribution and movement. In particular, column works well for the platelet simulation because most activities occur within a band that spans the middle rows of grid, while skewed is generally better if cells are clumped.

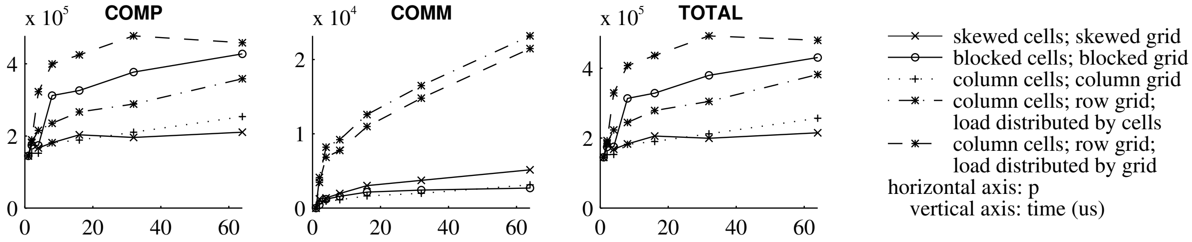

Figure 6. Modeled performance for cell/fluid interaction on the CM-5.

6. Putting Together the Whole Application

In this section, we finally turn to the problem of combining the Navier-Stokes solver and the cell operations to construct the complete cell simulation program. First we eliminate blocked cells and blocked grids from consideration, since both perform significantly worse than the alternatives. Therefore, we are left with two candidates for the solver—alternating row/column layout or skewed layout—and also two candidates for the cell layout—column or skewed.

A column cell layout is better combined with an alternating row/column grid than with a skewed grid, not only because row/column is the better layout for the solver, but also because cell and grid layouts will at least be aligned for a half of the time, which implies lower communication costs in the cell/fluid interaction. On the other hand, a skewed cell layout may be better combined with a skewed grid than with an alternating row/column grid, due to the advantage of alignment. Thus, two candidates for the overall layout are: (1) column cell layout with alternating row/column grid layout; and (2) skewed cell layout with skewed grid layout.

A third layout comes into consideration when we allow for major reorganization: (3) column cell layout with column grid layout. It is derived from a variation of (1). For (1), cell and grid layouts are aligned for only a half of the time in the cell/fluid interaction. In order to keep them aligned, we need to remap U and F before and after the cell/fluid interaction whenever the solver produces them in row layout. But instead of doing these four remaps in every two time steps, we can use just one remap per time step, noting that cell/fluid interaction only reads U and only writes F. First we convert U from row layout to column, then proceed with cell/fluid interaction and obtain F in column layout. Hence the solver always receives U and F in column layout and produces U in row layout, instead of operating under alternating layouts.

We present the modeled performance for these three layout combinations in figure 7, with $N^2/p = 2^{12}$ and $n/p = 20$. With no extra cost for combining the phases, the skewed layout offers the best overall performance.

Our implementation, which was completed before considering some cases in our model, uses a blocked layout for the cells, and a skewed layout for the grid during the tridiagonal solver, which is remapped to row/column for the FFTs. Compared to the sequential version of the platelet simulation with 256×128 fluid grid and 16 cells, our parallel implementation achieves a total speedup of 2.83 for $p = 4$ on the CM-5. Speedup numbers for larger p values are difficult to obtain, because it is impossible to run on one processor a problem whose size would be meaningful for a large p. To provide an indication of scalability, we keep the amount of memory used on each processor constant, and vary p, as we have done in previous plots. For $N^2/p = 2^{12}$ and $n/p = 1$, the average time required for one simulation time step is $533.3ms$ for $p = 1$ (sequential version), $726.9ms$ for $p = 4$, and $1059.3ms$ for $p = 16$.

7. Related Work in Automating Layout

Many researchers have developed compilation algorithms or systems to support automatic alignment and data distribution for data-parallel or automatically parallelized sequential programs [11, 7, 3, 14, 1, 15]. The global optimization problem for both locality and parallelism is NP-complete, and has been addressed using heuristics [3, 11, 15] and 0-1 integer programming techniques [5]. Some systems are interactive [11, 1]. All of this work covers regular parallel programs and is applicable only to the Navier-Stokes solver phase of our application. The tridiagonal solver is used as a benchmark for some systems [14, 3, 15], although none consider the skewed layout. In addition, transformations to globally rearrange FFTs and other computations using high-level knowledge of the semantics are probably beyond the scope of these compilers.

The problem of compiling irregular applications is typically addressed using a combination of compiler and runtime analysis [2, 9]. The information from this analysis is useful in eliminating redundant communication, converting small messages into larger ones, and in generating efficient communication schedules, but is not yet applicable to automatic alignment and distribution of irregular structures across program phases.

8. Conclusion

We have shown several instances in which the the best design for a composition of modules uses suboptimal imple-

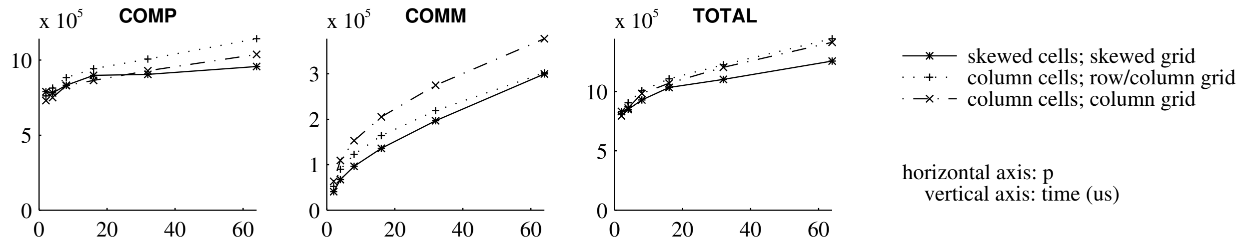

Figure 7. Modeled performance for the complete simulation on the CM-5.

mentations of individual modules, and cases where optimizing for either locality or parallelism separately would result in a poor design. This argues for better global analysis techniques or interactive systems that simultaneously optimize for locality and load balance. In addition, the problem size and the number of processors, as well as machine-specific performance characteristics, can sometimes lead to fundamentally different designs.

We have also demonstrated that a simple performance model is useful for determining data layouts of programs with both regular and irregular access patterns. The model correctly predicts the relative benefits of various layouts, although precise performance prediction requires either measured floating point rates for each kernel or a more detailed model of the local memory hierarchy. The model has been most useful in optimizing the Navier-Stokes solver, particularly in narrowing the set of designs to a reasonable number. Layouts that are clearly not suitable for our application, such as cyclic and blocked-cyclic, are not included in this presentation. We have found a surprising advantage of using the skewed layout, which is often overlooked in other work. Most importantly, our parallel implementation allows biologists to simulate larger and more complex systems than would otherwise be possible.

References

[1] V. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing*, San Diego, CA, Dec. 1995.

[2] G. Agrawal and J. Saltz. Interprocedural compilation of irregular applications for distributed memory machines. Technical Report CS-TR-3447, University of Maryland, College Park, MD, June 1995.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 23–25, Albuquerque, NM, June 1993.

[4] G. Angelopoulos and I. Pitas. Two-dimensional fft algorithms on hypercube and mesh machines. *Signal Processing*, 30, 1993.

[5] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Montréal, Québec, Aug. 1994.

[6] E. Brewer. High-level optimization via automated satistical modeling. In *Principles and Practice of Parallel Programming*, July 1995.

[7] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings, 20th Annual ACM Symposium on Principles of Programming Languages*, pages 16–28, Jan. 1993.

[8] D. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93*, pages 262–273, Portland, OR, Nov. 1993.

[9] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, Sept. 1994.

[10] J. Demmel and S. Smith. Parallelizing a global atmospheric chemical tracer model. In *IEEE Conference for Scalable High Performance Computation*, May 1994.

[11] T. Fahringer, R. Blasko, and H. P. Zima. Automatic performance prediction to support parallelization of fortran programs for massively parallel systems. In *6th ACM International Conference on Supercomputing*, pages 347–356, Washington, D.C., July 1992.

[12] F. Fauci and A. Fogelson. Truncated newton methods and the modeling of complex immersed elastic structures. *Communications on Pure and Applied Mathematics*, XLVI, 1993.

[13] S. Greenberg. Three-dimensional fluid dynamics in a two-dimensional amount of central memory. *Wave Motion: Theory, Modeling, and Computation*, 1987.

[14] K. Kennedy, C. Koelbel, and U. Kremer. Automatic Data Layout for High Performance Fortran. Technical Report TR94498, CRPC, Rice University, Dec. 1994.

[15] D. J. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995.

[16] C. Peskin and D. McQueen. Cardiac fluid dynamics. *Critical Reviews in Biomedical Engineering*, 20, 1992.

[17] S. Steinberg. Parallelizing a cell simulation: Analysis, abstraction, and portability. Master's thesis, University of California, Berkeley, Computer Science Division, Dec. 1994.

[18] M. Weliky. Notochord morphogenesis in xernopus laevis: simulation of cell behavior underlying tissue convergence and extension. *Development*, 113, 1991.