[93] Tankqueray, D., Snelling, D. F., "A Distriubted Memory Implementation of the Shallow Water Equations," *Dawn of Massively Parallel Processing in Meterology*, eds. Hoffman, G.R., Maretis, D. K., Springer-Verlag, 1990.

[94] Thinking Machines Corporation, "Communications Architecture in the Connection Machine System", *Technical Report Series*, HA87-3.

[95] Thinking Machines Corporation, "Connection Machine Model CM-2 Techinical Summary," *Technial Report Series*, HA87-4.

[96] Tuccillo, J. J., "Numerical Weather Prediction on the Connection Machine", *Fourth International Conference on Supercomputing*, Santa Clara, California, April 30-May 5, 1989, pp 334-342.

[97] Tong, C. H., *Parallel Implementation of Multilevel Filtering Preconditioners on the Connection Machine,* 1st Copper Mountain Conference on Iterative Methods, April 1990.

[98] Van Der Vorst, H. A., "Practical Aspects of Parallel Scientific Computing", Future Generation Computer Systems, Vol. 4, No. 4, March 1989.

[99] Walker, D. W., Messina, P., Bailie, C. F., "Performance Evaluation of Scientific Programs on Advanced Architecture Computers", Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, California, January 19-20, 1988.

[100] Washington, W. M., Parkinson, C. L., *An Introduction to Three-Dimensional Climate Modeling,* University Science Books, Mill Valley, California, 1986.

[101] Weicker, R. P., "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, October 1984.

[102] Williamson, D. L., et. al., "Description of NCAR Community Climate Model (CCM1)," NCAR Technical Note NCAR/TN-285+STR, Climate and Global Dynamics Division, National Center for Atmospheric Research, Boulder, Colorado, June 1987.

[103] Wilson, K. G., "The Role of Universities in High-Performance Computing", in *Frontiers of Supercomputers*, eds. Metropolis, N., Sharp, D.H., Worlton, W. J., Ames, K. R., Unversity of California Press, Berkeley, California, 1986, pp. 362-368.

[104] Wubbs, F. W., "Performance Evaluation of Explicit Shallow-Water Equation Solvers on the Cyber 205" in *The Efficient Use of Vector Computers with Emphasis on Computational Fluid Dynamics*, eds. Schonauer, W., Gentzsch, W., Friedr. Vieweg & Sohn, Brauschweig, Germany, 1986, pp. 126-137

[74] Parrallel Computing Forumn, "Parallel FORTRAN Draft," Technical Report, ANSI Committee X3H5, Parallel PRocessing Constrcuts for High Level Programming Languages, 1990.

[75] Parker, S. P. (ed.), *Fluid Mechanics Source Book*, McGraw-Hill, New York, New York, 1987.

[76] Parkinson, D., Liddel, H. M., "The Measurement of Performance on a Highly Parallel System," *IEEE Transactions on Computers,* Vol. C-32, NBo. 1, Jan. 1983, pp. 32-37.

[77] Pierce, P., *Private Communication*, Intel Scientific Computers, Beaverton, Oregon, June 1991.

[78] Pozo, R., MacDonald, A. E., "Performance Characteristics of Scientific Computation on the Connection Machine", CS Tech Report, University of Colorado, Boulder, 1989.

[79] Pozo, R., "Connection Machine Status Report," Report to the Scientific Computing Division of National Center for Atmospheric Research and the Center for Applied Parallel Processing. September, 1989.

[80] Sadourny, R., "The dynamics of finite difference models of the shallow water equations," *Journal of Atmospheric Science*, vol. 32, pp. 680-689, 1975.

[81] Sato, R. , Swarztrauber, P., "Benchmarking the Connection machine 2", *Proceedings of the Supercomputing Conference,* Orlando, Florida, November, 14-18, 1988.

[82] Sato, R. K., internal document of the Scientific Computing Division of the National Center for Atmospheric Science (NCAR), Boulder, Colorado, 1989.

[83] Scott, D. S., Castro-Leon, E., Kushner, E. J., "Solving Very Large Dense Systems of Linear Equations on the iPSC/860," *Proceedings of the Fifth Distriubted Memory Conference,* April 8-12, 1990, Charleston, South Carolina.

[84] Seidel, S., Lee, M., Fotear, S., "Concurrent Bidirectional Communication on the Intel iPSC/860 and iPSC/2," Computer Science Technical Report, CS-TR 90-06, Michigan Technological University, Nov. 1990.

[85] Siegel, L. J., Siegel H. J., Swain, P. H., "Performance Measures for Evaluating Algorithms for SIMD Machines," *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 4, July 1982, pp. 319-330.

[86] Smith, B.T., "Array Processing Features in the next Fortran", in *The Relationship between Numerical Computation and Programming Languages*, North-Holland, 1981, pp. 163-178.

[87] Smith, J. E., "Characterizing Computer Performance with a Single Number," *Communications of the ACM*, Vol. 31, Number 10, October 1988.

[88] Snelling, D. F., "Inherent Parallelism in Numerial Weather Prediction Algorithms," *Dawn of Massively Parallel Processing in Meterology*, eds. Hoffman, G.R., Maretis, D. K., Springer-Verlag, 1990, pp. 159-174.

[89] Stevens, R., McDonough, P., "Instruction Timings and Message Passing Performance of the Connection Machine 2", *Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Monterey, California, March 6-8, 1989, pp. 257-264.

[90] Stone, H. S., *High Performance Computer Architecture*, Addison-Wesley, Reading, Mass., 1987.

[91] Swartztrauber, P. N., "Multiprocessor FFTs", *Parallel Computing,* Vol. 5, 1987, pp. 197-210.

[92] Swartztrauber, P., Sato, R., "Solving the Shallow Water Equations on the Cray X-MP/48 and the Connection Machine 2," *Topics in Atmospheric and Ocean Sciences*, Springer-Verlag 1990.

[55] Levit, C., "Grid Communication on the Connection Machine: Analysis, Performance, and Improvements", *Proceedings of the Conference on Scientific Applications on the Connection Machine,* Moffett Field, California, September 12-14, 1988.

[56] Lillevik, S. L., "Touchstone Program Overview," *Proceedings of the Fifth Distriubted Memory Conference,* April 8-12, 1990, Charleston, South Carolina, Vol. 2, pp. 647-657.

[57] Mader, C. L., Lukas, S., "SWAN - A Shallow Water, Long Wave Code: Applications to Tsunami Models," Joint Institute for Marine and Atmospheric Research report, JIMAR 84-077, 1984.

[58] Mader, C. L, Vitousek, M., Lukas, S., "Numerical Modeling of Atoll Reef Harbors," *Proceedings of the International symposium on Natural and Man-Made Hazards,*, Rimouski, 1986.

[59] Margulis, N., *i860 Microprocessor Architecture*, Osborne, McGraw-Hill, 1990.

[60] McBryan, O., Van De Velde, E., "Hypercube Algorithms and Implementations," *SIAM Journal of Scientific and Statistical Computing,* No. 8, pp 227-87.

[61] McBryan, O., "Connection Machine Application Performance" *Proceedings of the Conference on Scientific Applications on the Connection Machine,* Moffett Field, California, September 12-14, 1988.

[62] McBryan, O., "State of the Art of Mutli-Processors", in in Multiprocessing in Meterological Models, by Joffmann, G. R., Snelling, D. F., Springer-Verlag, Berlin, 1988., pp. 5-40.

[63] McBryan, O., "New Architectures: Performance Highlight and New Algorithms", *Parrallel Computing*, Vol. 7, pp. 477-499, North-Holland, 1988.

[64] McBryan, O., Pozo, R., "Performance Evaluation of the Myrias SPS-2 Computer," Department of Computer Science Technical Report CU-CS-505-90, University of Colorado, Boulder, 1990.

[65] McCreary, C. L., McArdle, M. E., McCreary, J. D., "Broadcast Communication Delay Metric for the iPSC/2 and iPSC/860 Hypercubes," Department of Computer Science and Engineering, *to be submitted,* June, 1991.

[66] McMahon, F., "The Livermore Fortran Kernels," *Performance Evaluation of Supercomputers,* ed. Matrin, J. L., North-Holland, 1988.

[67] McMahon, L. E., "SED – A Non-interactive Text Editor," *UNIX User's Supplementary Documents (USD)*, Computer Systems Research Group, Berkeley, California, April, 1986.

[68] Miller, A., Anthes, R., "Meterology", Charles E. Merril Publishing, Columbus, 1985.

[69] Myers, D., Adams, G., "Benchmarking and Performance Analysis of the CM-2", *Proceedings of the Conference on Scientific Applications on the Connection Machine,* Moffett Field, California, September 12-14, 1988.

[70] National Academy of Sciences, "An Agenda for Improved Evaluation of Supercomputer Performance", *Report to the National Academy of Sciences*, pub. National Academy Press, Washington, D. C., 1986.

[71] Ortega, J. M., *Introduction to Parallel and Vector Solutions of Linear Systems,* Plenum Press, New York, 1989.

[72] Ortega, J. M., Voigt, R. G., *Solution of Partial Differential Equations on Vector and Parallel Computers,* SIAM Press, Philadelphia, 1985.

[73] Parallel Computing Forum, *PCF Fortran: Language Definition*, Draft Version 3.4, Appendix A.

[35] Goede, E., "A Computational model for the 3D shallow water flows on the Alliant FX/4," *Supercomputer*, July 1989, pp. 43-49.

[36] Golub, G. H., Van Loan, C. F., *Matrix Computations*, John Hopkins Press, Baltimore, 1984.

[37] Hack, J., et. al., "Modifications and Enhancements to the NCAR Community Climate Model (CCM1)," NCAR Technical Note NCAR/TN-2336+STR, Climate and Global Dynamics Division, National Center for Atmospheric Research, Boulder, Colorado, June 1989.

[38] Hack, J. J., Ruediger, J., "Parallel MIMD Programming for Global Models of Atmospheric Flow," *Supercomputing 89*, p. 106-112, 1989.

[39] Haltiner, G. J., Williams, R. T., *Numerical Prediction and Dynamic Meterology,* Wiley & Sons, New York, 1980.

[40] Harlow, F. H., "Super Problems for Supercomputers," in in 'Frontiers of Supercomputers', eds. Metropolis, N., Sharp, D.H., Worlton, W. J., Ames, K. R., Unversity of California Press, Berkeley, California, 1986, pp. 231-238.

[41] Hennessy J., Patterson, D., *Computer Architecture: A Quantitative Approach,* Morgran Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[42] Hestenes, M. R., Stiefel, E., "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Standards*, Vol. 49, 1952, pp. 409-436.

[43] Hillis, W. D., *The Connection Machine,* MIT Pres, Cambridge, Mass., 1985.

[44] Hockney, R. W., Jesshope, C. R., "Parallel Computers," Adam Hilger Ltd, Great Britian, 1981.

[45] Hoffman G. R., Snelling, D. F., "Multiprocessing in Meterologlical Models", Springer-Verlag, 1988.

[46] Hoffman, G. R., Swarztrauber, P.N., Sweet, R. A., "Aspects of using multiprocessors for meteorological modeling," in *Multiprocessing in Meteoro- logical Models*, ed. D. Snelling, pp. 126-195, Springer-Verlag, Berlin, 1988.

[47] Hollander, E. D., "Performance Modeling of Supercomputer Architectures and Algoritms", in *Scientific Computing on Supercomputers,* eds. Devresse, J. T., Van Camp, P. E., Plenum Press, New York, 1989, pp. 3-28.

[48] Holton, J. R., *An Introduction to Dynamic Meterology*, Academic Press, New York, 1979.

[49] Houghton, John T., "The Physics of Atmospheres", Cambridge University Press, Cambridge, 1986.

[50] Hwang, K. H, Briggs, F., *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.

[51] Intel Corp., *iPSC/2 and iPSC/860 User's Guide*, Order No. 31152-006, June 1990.

[52] Kamin, R. A., Adams, G. B., "Fast Fourier Transform Algorithm Design and tradeoff on the CM-2", *Proceedings of the Conference on Scientific Applications on the Connection Machine,* Moffett Field, California, September 12-14, 1988.

[53] Karp, A., Flatt, H. P., "Measuring Parallel Processor Performance," *Communication of the ACM,*, Vol. 33. No. 5, May 1990.

[54] Lazou, C., *Supercomputers and their Use,* Clarendon Press, Oxford, 1986.

[16] Chen, M.C., Choo, Y., Jacquemin, M., "Minimizing Communication Cost for Distributed Arrays on Massively Parallel Machines," *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing,*, Chicago, Illinois, December 11-13, 1989, pp. 371-373.

[17] Cunrow, H. J., Wichman, B. A., "A Synthetic Benchmark," *Computer Journal*, February 1976.

[18] Desrochers, G. R., *Principles of Parallel and Multiprocessing*, McGraw-Hill, New York, New York, 1987.

[19] Dongarra, J., Moler, C. B., Bunch, J. R., Stewart, G. W., *LINPACK Users' Guide,* SIAM Press, Philadelphia, 1979.

[20] Dongarra, J., Marti, J. L., Worlton, J., "Computer Benchmarking: paths and pitfalls", *IEEE Spectrum*, Vol 24, No. 7, July 1987, pp. 38-43.

[21] Dongarra J., et. al. "A set of Level 3 Basic Linear Algebra Subprograms", Argonne National Lab., Preprint No. 1, August 1988.

[22] Dongarra, J., Duff, I. S., Sorensen, D., van der Vorst, H., *Solving Linear System on Vector and Shared Memory Computers*, SIAM Press, 1991.

[23] Dongarra, J., Ostrouchov, S., "LAPACK Working Note 24: LAPACK Block Factorization Algorithms on the Intel iPSC/860," CS-90-115, Computer Science Department, University of Tennesseee, October 1990.

[24] Dongarra, J., "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Argonne National Laboratory, Technical Memorandum No. 23, June 4, 1989.

[25] Duff, Ian, S., "The use of supercomputers in Europe," *Computer Physics Communicatons,* NO. 37, pp. 15-25.

[26] *ES-1 Parallel Programmers' Manual*, Technical Publications Department, Evans & Sutherland Computer Division, Mountain View, California, 1989.

[27] Farhat, c., Sobh, N., Park, K. C., "Dynamic Finite Element Simulations on the Connection Machine,"*Proceedings of the Conference on Scientific Applications on the Connection Machine,* Moffett Field, California, September 12-14, 1988.

[28] Farhat, C., Sobh, N., Park, K. C., "Dynamic Finite Element Simulations on the Connection Machine," *Internation Journal of High Speed Computing,* Vol. 2, 1989.

[29] Flemming, P., Wallace, J., "How not to Lie with Statistics: the Correct Way to Summarize Benchmark Results," Communications of the ACM, Vol. 29, No. 3, March 1986.

[30] Federickson, P. O., "Totally Prallel Multilevel Algorithms", *Proceedings of the Conference on Scientific Applications on the Connection Machine,* Moffett Field, California, September 12-14, 1988.

[31] Federickson, P. O., McBryan, O., "Parallel superconvergent multigrid," in *Multigid Methods*, eds. McCormick S. F., Dekker, M., New York adn Basel, 1988.

[32] Gentzsch, W., "Parallel Benchmark Results for Shared Memory Systems," *Supercomputer*, July 1989, pp. 10-16.

[33] Grimsrud, K. S., Archibald, J. K., *Hierarchical Parallelism in the ES-1 Prototype,* Technical Report, Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah, April 1990.

[34] Gilbreadth J., "A High Level Language Benchmark," *BYTE,* September 1981.

Bibliography

[1] Aho, A. V., Kernigham, B. W., Weinberger, P. J., "Awk – A Pattern Scanning and Processing Language," *UNIX User's Supplementary Documents (USD)*, Computer Systems Research Group, Berkeley, California, April, 1986.

[2] Aho, A. V., Kernigham, B. W., Weinberger, P. J., *The AWK Programming Language,* Addison-Wesley, Reading, Mass., 1988.

[3] Almasi, G. S., Gottlieb, *Highly Parallel Computing*, Benjamin Cummins, 1989.

[4] Baker, D., Wilson, T., "First user experiences on the Myrias SPS-2 parallel computer at the University of Calgary," *Supercomputer*, Vol. 7, No. 5, September, 1990.

[5] Bailey, D. H., et. al., "Performance Results on the Intel Touchstone Gamma Prototype," *Proceedings of the Fifth Distributed Memory Computing Conference,* 1990, pp. 1236-1245.

[6] Barton, M. L., Withers, G. R., "Computing Performance as a Function of the Speed, Quantity, and Cost of the Processors," *Proceeedings Supercomputing 89*, Reno, Nevada, Nov. 13-17, 1989, pp. 759-764.

[7] Bengston, L., "Computer Requirements for Atomspheric Modelling", in Multiprocessing in Meterological Models, by Joffmann, G. R., Snelling, D. F., Springer-Verlag, Berlin, 1988., pp. 109-116.

[8] Berry, M., Chen, D., et al, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers", International Journal of Supercomputer Applications, Vol. 3, No. 3, Fall 1989, pp. 5-40.

[9] Bokhari, S., "Communication Overhead on the Intel iPSC-860 Hypercube," ICASE Interim Report 10, NASA Contract No. NAS1-18605, May 1990.

[10] Bomans, L., Roose, D., "Benchmarking the iPSC/2 Hypercube Multiprocessor", *Concurrency: Practice and Experience*, Vol. 1, No. 1, September 1989, pp. 3-18.

[11] Brawer, S., "Introduction to Parallel Programming", Academic Press, Boston, 1989. pp.1-423 QA 76.6 B716 1989.

[12] Browning, G. L., Kreiss, H. O., "Reduced systems for the shallow water equations," *Journal of Atmospheric Science*, to appear.

[13] Burridge, D. M., "The Current State of Meterological Modelling", in Multiprocessing in Meterological Models, by Joffmann, G. R., Snelling, D. F., Springer-Verlag, Berlin, 1988., pp. 109-116.

[14] Capotondi, A., Sonnad, V., Chin, S., "Parallel solution of the shallow water equations using an explicit finite difference algorithm," *Computer Physics Communications,* Vol. 52, No. 2, January 1989.

[15] Chan, T., *Hierarchical Algorithms and Architectures for Parallel Scientfic Computing,* 1990 International Conference on Supercomputing, June 11-15, 1990, Amsterdam, The Netherlands.

7.5 Final Thoughts

In previous sections, we have already discussed various conclusions for the different parallel systems. In this section, we tie together several of the key issues from previous chapters. In summarizing what we have learned by attempting to provide better performance models for parallel architectures, we find that

- synchronous computation on advanced parallel systems can be modeled *accurately*.
- multilevel performance analysis is necessary to capture the complex behavior of these architectures.
- systems such as CMPS demonstrate that the analysis from several levels can be successfully integrated to yield accurate performance models for grid-based applications.
- such models are not only useful to end-users but to various computer science development groups in identifying relative strengths, weaknesses, and bottlenecks in computing systems.

The first point is not as trivial as it may initially sound; the key word here is "accurate." Because the performance of these systems is quite sensitive to various application/architecture considerations (e.g. task granularity, synchronization method, initial layout of distributed data, communication primitives, and communication/computation effects) many people believe that the behavior of performance details exhibited by these systems is simply too complex to be able to describe concisely. We contend that every performance attribute observed stems from a program and architecture operating on a well defined set of principles and it is just a matter of how much detail one is willing to uncover in analyzing and modeling the relationships. The modeling the computational performance of the Intel iPSC/860 is a good example. At first, its behavior observed in applications was quite baffling, but after a detailed analysis of its intricate architecture and its interaction with the various memory hierarchies, we could formulate models that described its performance down to the clock cycle. Similar experiences were encountered when modeling the communication and computation effects of the CM-2.

Hopefully, by know the reader has been convinced that analyzing a single level of computation restricts what one can observe. For example, it would have been nearly impossible to understand the performance of Shallow Water subroutines had we not had a reliable model of the i860 RISC architecture. We also would not have been able to realize the limitations of low-level NEWS grid communication and floating point computations on the CM-2 had we not attempted to integrate such models and apply them at the algorithm and application levels. It was this situation that led to the development of the software prototype CMPS system.

It should also be clear that the parameterized information yielded by these models are useful not only to the programmer fine-tuning a piece of code, but also to individuals involved in other aspects of computer science. It allows compiler writers, hardware designers, and operating system developers to and explore new designs while obtaining immediate feedback on the performance impact of such decisions.

The parallel computing field is a vastly growning and changes daily with the introduction of new hardware components, languages, and programming environments. What will parallel systems of the future look like? Many avenues are being explored and at this stage of development it is perhaps still to early to announce any final decisions. Nevertheless, one thing is certain: performance will always be a fundamental driving force. And with better performance models, designers will better to able to assess the relative strengths and weaknesses of new implementations. In short, performance modeling is essential not only for assessing today's parallel systems but for designing tomorrow's.

7.4 Intel iPSC/860

In summary, we started with a simple low-level model of communication based isolated point-to-point communication. This model agreed well with experimental data and with similar results from other researchers (see section 1.4.3 for related work). After studying numerous subgrid decompositions of the Shallow Water equations, however, we found that the communication latency predicted by this model did not match the experimental measurements. This drove us back to the low-level communication model to try to obtain a better understanding of the communication architecture. We found that the reduced message latency was due to a combination of a 3-step handshaking protocol and a single path between the node CPU and communication hardware (sections 6.3.2 and 6.3.3).

We ran low-level experiments to determine the costs associated with each step of the message protocol (section 6.3.2 and figure 6.16) and analyzed situations where contention could occur between colliding messages used in the communication protocol (section 6.3.3). We conducted furhter experiments on circular shift algorithms which exhibit the same communication pattern of multidimensional data relaxation kernels and finite difference schemes (section 6.4.3). The results verified our hypothesis about message collision behavior and allowed us to formulate a model for the communication latency exhibited by the Shallow Water application. Using such a model (section 6.6.3) we could characterize the performance behavior of the Shallow Water Equations on various subgrid and processor configurations with less than 10% error for most cases (figures 6.40 through 6.47).

Along the way, we found a method to effectively double communication bandwidth to synchronous circular shift algorithms by bypassing the three step protocol for long messages. The algorithm involves a forward-synchronization step which first passes a *synchronization token* around the circular and follows it with a special message type that gets delivered to the destination node without the usual handshaking overhead (section 6.4.4). Our experiments verified that such message algorithms may be beneficial even for messages as short as 4Kbytes (see figure 6.28 and exhibit superior performance for longer messages (figure 6.29).

After an in-depth analysis of the computational architecture of each node, we presented in section 6.2 a computational performance model of the underlying i860 RISC architecture. Section 6.2 describes scalar and pipelined performance of the i860 on vectorizable loops. An analysis of the compiler code generated for such loops shows that many of the hardware optimizations for pipelining floating point operations and dual instruction mode are not being utilized. To remedy this, we provide hand-coded vector operations which achieve over 50 Mflops on a single processor (section 6.2.2). This peak speed, however, is very dependent on the memory traffic patterns used in accessing the vector arguments and in section 6.2.3, we analyze the memory latency interactions between main memory, Fast-Page-Access RAM, and on-chip cache. The performance of the i860 is mainly limited by memory bandwidth; there is only one 64-bit bus to main-memory, while the cache bus is 128 bits wide and limits the use of vector-vector operations, such as `saxpy()`, that requires two read pipes and one write pipe. In section 6.2.3, we present a detail computation model for vectorizable loops for the i860.

Finally, in section 6.6.4 we used performance models to predict and verify efficiencies of over 94% for 128x128 subgrid decompositions of the Shallow Water Equations. Our model predicted the performance of 502 Mflops (128 processors) with less than 3% error.

7.3 Evans & Sutherland ES-1

The performance analysis of the ES-1 has demonstrated a RISC-based shared memory machine capable of achieving around 5-7 Mflops per processor on the Shallow Water equations. The parallelization of iterative loops are quite straightforward using the native compiler directives, or the PFP (section 2.5) primitives. However, as with the other machine studied, the mapping of data into the ES-1 multi-level memory hierarchy (section 5.1.3) plays a fundamental role in application and algorithm performance.

Because the ES-1 became unavailable during the course of experimentation, we could complete all of the necessary analysis; however, we were able to extract many of the fundamental performance characteristics. In section 5.3.1 we saw that parallelism is specified by number of threads rather than the number of physical processors. In section 5.3.2 we established that single precision scalar operations can achieve 7.0 Mflops. To minimize the effects of memory bank conflicts in each cluster, we studied vector/matrix operations using non-power of two grids. The effects of thread spawning and termination show less than linear speedup (section 5.3.3).

The parallel relaxation performance (section 5.3.4) shows a near-linear relationship at about 5 Mflops per processor up to 6 nodes. For multidimensional grids, the efficiency drops to about 65% for 12 processors. Using the techniques described in section 2.4, we tried several grid decomposition strategies and obtained nearly 50 Mflops (12 processors) for 3D grid relaxations (figures 5.7 and 5.8).

Similarly, in section 5.4for the 2D Shallow Water equations we saw 45-60 Mflop ratings for various grid decompositions using 12 processors (section 5.4). The efficiencies ranged from 73% to 92% for the different data decomposition methods (figures 5.9 and 5.11). Using 13 processors, we were able to achieve 57.3 Mflop on a 281x281 grid.

Mflops with 64 processors on a 1024x1024 grid, and performed at .117 Mflos on a single processor with correspondingly smaller grid (128x128,) thus yielding a parallel efficiency of 36%. This lower efficiency can be partly attributed to the boundary copying necessary to implement the periodic boundary conditions. Because fewer processors participate in this section of the code, the boundary data has to be redistributed throughout the system causing significant communication among processors.

7.2 Myrias SPS-2

The SPS-2 was an important machine from the point of view that it demonstrated by virtual shared memory systems could actually be built and work reliably. The SPS-2 was by far, the easiest distributed memory machine to program. Furthermore, it had the capability of running large (> 64 Mbytes) sequential programs by utilizing the extra processors' address space as virtual memory.

Although the SPS-2 was one of the first virtual shared memory computers available, it enjoyed limited success in the scientific computing market. Much of this was related to machine's conservative numerical performance. With each Motorola 68882 producing on the order of 100 Kflops (section 4.6.1), a 64 processor machine yielded a peak performance of roughly 6 Mflops. However, the machine's strength lay in its operating system and programming environment rather than on its specific platform. Although the SPS-2 has shown reasonable efficiency for various parallel computations, it remains to be seen how communication aspect of the virtual shared memory holds up when the computing elements are replaced with more powerful processors.

In characterizing this parallel system, we have analyzed the performance of single nodes to be about 50-60 Kflops for triad vector operations (section 4.6.3.) The virtual memory capability was analyzed in section 4.6.4 where we observed a 40-50% drop in performance due to the overhead of virtual memory access (table 4.3.) In determining the multiprocessor overhead and task granularity for this system we conclude that task sizes less than 100,000 floating point operations will encounter significant losses in efficiency. For scaled vector kernels (section 4.6.5) we saw efficiencies of nearly 100% for fixed-vector/variable-work triad kernels where the task granularity was 128 Kflops. The efficiency drops to about 70% when the task granularity is reduced to 64 Kflops. For non-scaled task granularities we see the common effect of Amdahl's law as the efficiency drops to zero as the number of processors increases (figures 4.1 and 4.2.) The performance of these vector kernel can be accurately modeled as linear relationship to the number of processors, p. For the 128K-flop task size, we can describe the Mflop performance as $M(p) = 0.0594 * p + 0.0029$, with a relative error less than 5% for the interesting domain sizes, $8 \leq p \leq 64$. Similarly, we can model the smaller 64-Kflop task size as $M(p) = 0.0425 * p - 0.048$, with the same relative error.

To get a feel for the performance of iterative solvers, we studied 1D, 2D, and 3D relaxation performance (section 4.7) by varying the per-processor grid shape, the overall grid size and the number of processors. We also investigated the effects of having multiple tasks per processor, but found this to have minimal effects. For a given task granularity, the 3D relaxation turned out the best performance (table 4.5). This is because there is a large ratio of floating point operations per local data distribution and has greater reuse of data. The 1D case we can model the performance as $M(p) = 0.1677 + 0.0313 * p$ with a 10% error (figure 4.7) , the 2D and 3D have larger variations to the linear model, since their memory reference patterns were more complex (figures 4.9 and 4.10). In iterative solvers, the relaxation step is repeated many times; however for a single iteration step there is an increased 20% overhead, since initial copies of data need to be distributed to each processor. Once the data pages have found their processor home, only the pages which contain boundary information need to be merged by the operating system.

The parallelization and data decomposition of these relaxation kernels consisted of replacing parallelizable outer `do` loops with the `pardo` extension, using the `decompose()` and `pardo_decomp()` constructs discussed in section 2.4. Assigning multiple loop iterations per task, termed "loop bundling" had to coded by hand. To help remedy this situation, we developed a Fortran macro preprocessor, PFP (section 2.5), that performs this automatically. PFP also allows the same Fortran program to be run on the Myrias SPS-2, Evans & Sutherland ES-1, and conventional uniprocessor machines with no code modification.

We utilized the strategy for parallelizing the Shallow Water equations on the SPS-2. Each task performs a double DO loop using indices provided by decompose. Similarly, the boundary condition loops were parallelized by treating them as 1D grids and using a single call to decompose for each side of the subgrid. In practice, task granularity for the boundary loops became rather small and so we used a smaller number of tasks to parallelize the boundary than were used in the interior of subgrids. As with the relaxation examples, we studied several methods to decompose the grid among the tasks (section 4.8) and found the most efficient method to be the use of full column slices (section 4.9.) The SPS-2 performed the explicit model at 2.71

Finally, we have used this prototype software system to model the performance of the Shallow Water equations for various virtual processor geometries on the CM-2 with greater than 90% accuracy.

Chapter 7

Conclusion

7.1 Connection Machine CM-2

In characterizing the performance of the Connection Machine CM-2, we developed fundamental models that accurately describe its behavior. Highlighting the major achievements from previous sections, we have developed

- communication models of startup costs and transmission latency are accurate to 5% error for nearly all geometries.
- Computation Model that account for virtual processor ratios and FPU pipelining effects are accurate to 3% error.
- Simulation performance of applications in high-level languages is accurate to less than 10% error.
- Furthermore, we have described an automated system that utilizes such models to a provide an effective development tool for parallel algorithm designers.

We have shown that given a specific virtual processor geometry configuration, for example, the cost of communicating n bytes from a processor to one of its neighbors can be modeled as a linear relationship consisting of a fixed startup cost and an amortized per-word communication latency. By analyzing the various memory hierarchies between virtual, physical and chip processor levels we have identified fundamental machine characteristics (t_E, t_M, t_I) that describe the communication costs between these various levels. By determining relationships for these coefficients, we have extended current models of grid communication to include the effects of arbitrary message length and communication/computation optimizations, as well as arbitrary virtual processor geometries.

By studying the underlying CM-2 floating point architecture and the relationships between the inner components (the Sprint chip, and Weitek 3132 FPU), we have developed a performance model that is roughly 97% accurate. We have shown that this behavior is similar to pipelined floating point vector processors, except that the pipelining effects arises from the virtual processor ratio. We have described the various transformations and stages in the transfer of data from processor memory through the Sprint chip and to the Weitek unit.

We have seen that a paper-and-pencil analytical performance model of the CM-2 does not provide a complete solution; one needs a methodology to apply this model to realistic applications.

Given the various ways in which compilers and software tools translate high level code in the Paris constituents, an integrated and automated performance modeling system is necessary. We have described a prototype system, CMPS, that analyzes and predicts the communication and computation behavior of actual CM-Fortran and C* codes. This system is designed to work transparently with the CM compilers.

Table 6.4: Speedups for 128x128 Subgrid Decomposition of Shallow Water Equations.

Procs	Grid	Observed	Predicted	Ideal	Efficiency
1	128x128	4.0203	-----	4.0203	100.0000
2	256x128	7.8664	7.842	8.0406	97.8343
4	256x256	15.3873	15.684	16.0811	95.6853
8	512x256	30.5745	31.368	32.1623	95.0633
16	512x512	60.9776	62.736	64.3246	94.7968
32	1024x512	121.2230	125.547	128.6492	94.2276
64	1024x1024	242.1816	250.94	257.2983	94.1248
128	2058x1024	487.1949	501.89	514.5966	94.6751

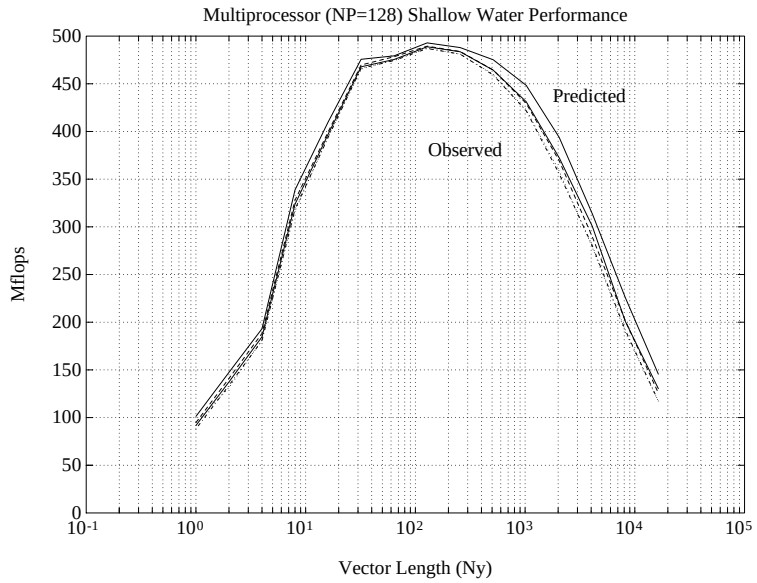


Figure 6.47: Mflop rating for 128 processor version of Shallow Water.

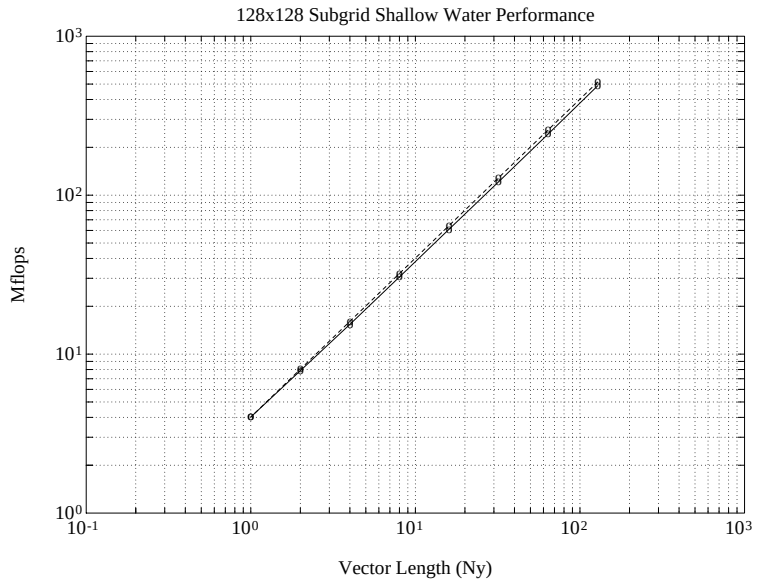


Figure 6.48: Performance of scaled 128x128 problem size for Shallow Water shows near linear speedup.

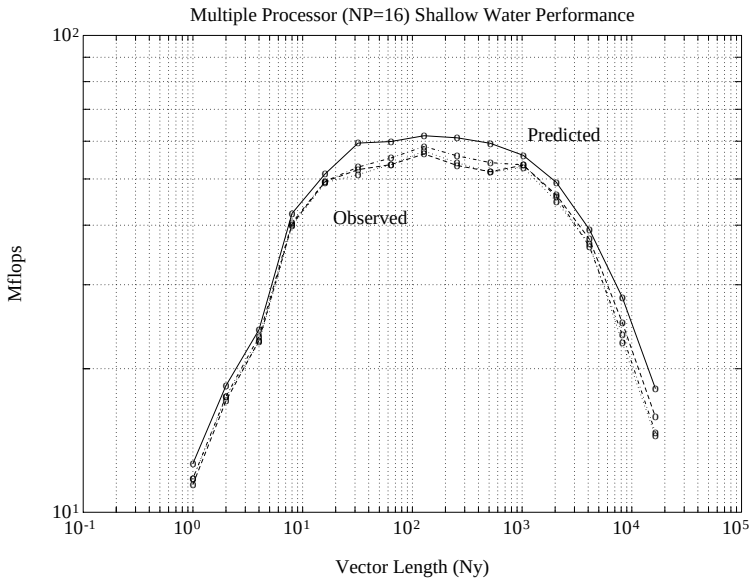


Figure 6.45: Mflops prediction of various 16-processor decompositions (8x2, 4x4, 2,8) of the Shallow Water application.

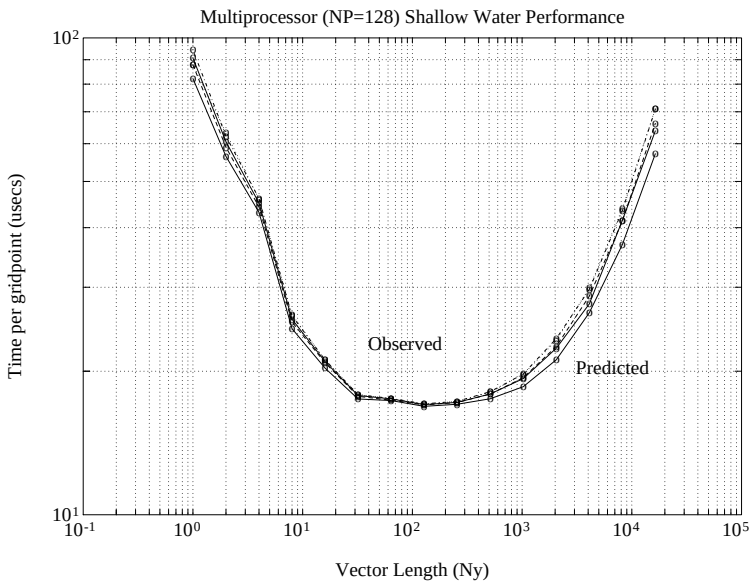


Figure 6.46: Performance model for 128-processor version of Shallow Water.

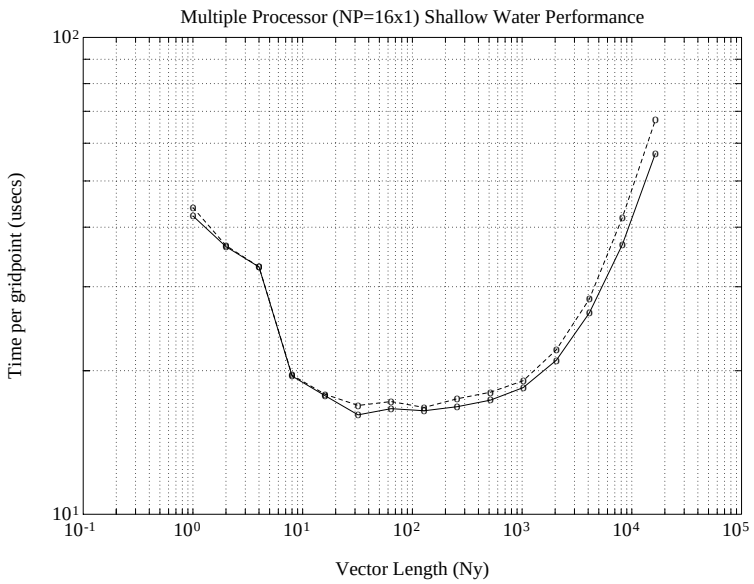


Figure 6.43: Performance model of NP=16x1 version of Shallow Water application.

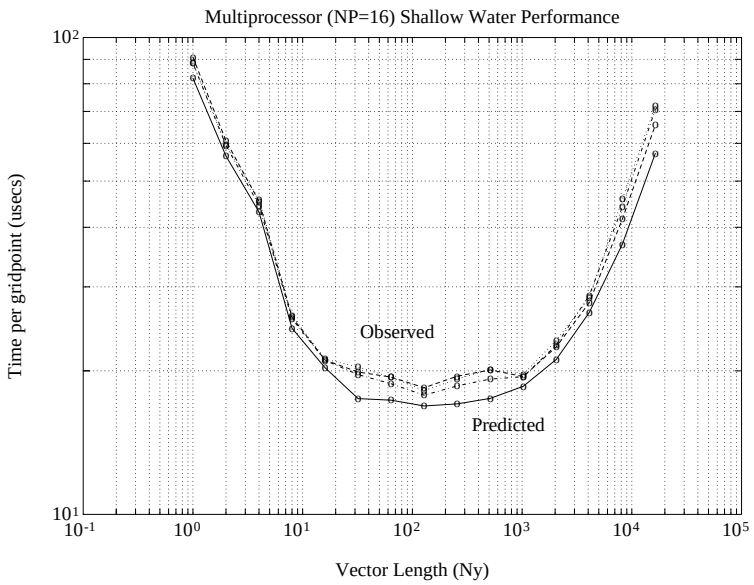


Figure 6.44: Performance model of various 16-processor decompositions (8x2, 4x4, 2,8) of the Shallow Water application.

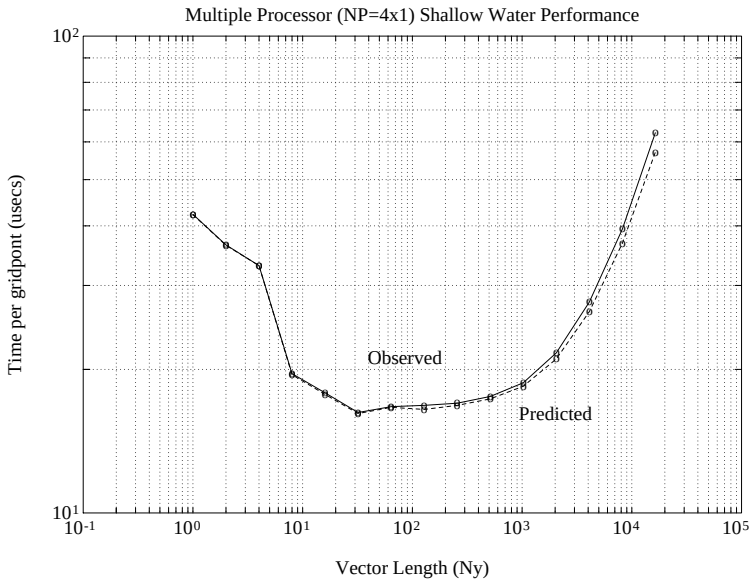


Figure 6.41: Performance model of NP=1x4 version of Shallow Water application.

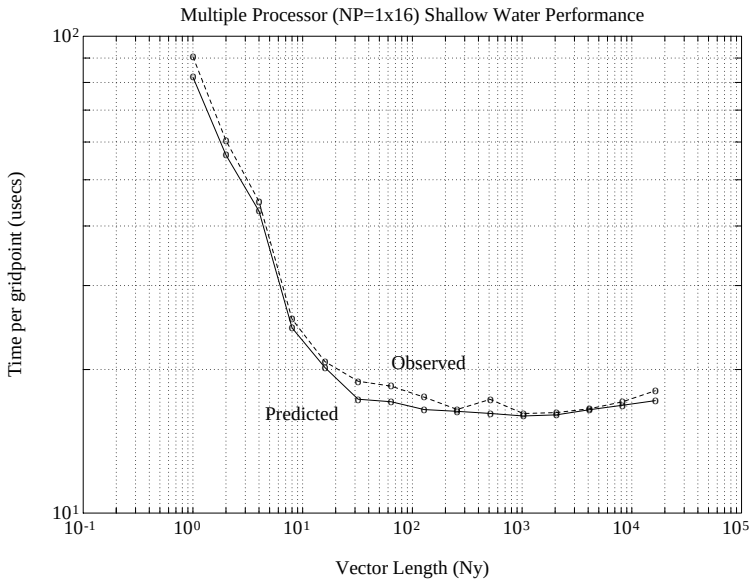


Figure 6.42: Performance model of NP=1x16 version of Shallow Water application.

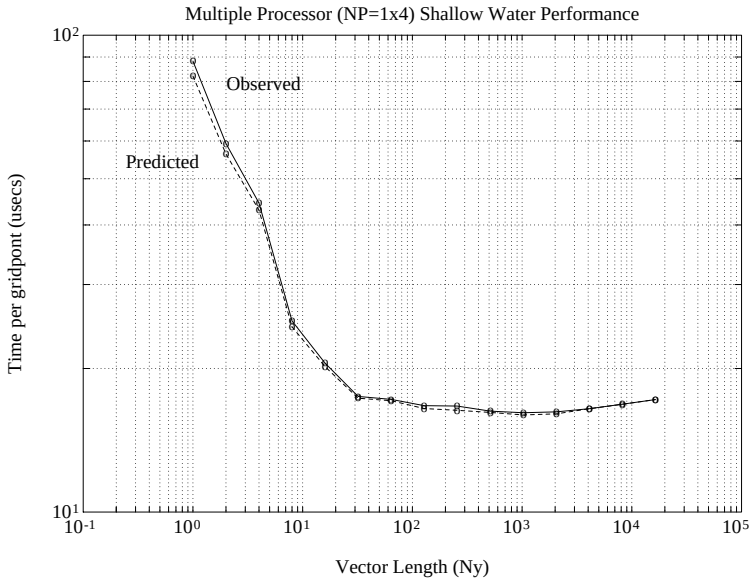


Figure 6.40: Performance model of NP=4x1 version of Shallow Water application.

more difficult.) Similarly, having a simulator of the Intel i860 microprocessor we could have predicted the computation performance with no error (the i860 computation performance model is based on discrete clock cycles, hence it can be make exact. See section 6.2 for details.) Since each processor receives the same calculation as the single processor case, we can predict the latency for an iteration by simply adding the communication time from equation 6.26 to the single processor computation time determined experimentaly. This is valid since we have shown in section 6.2 that such a model can be arbitrarily precise.

Figures 6.40 through 6.47 illustrate predicted vs. observed performance curves for several subgrid decompositions of the Shallow Water equations. In most cases we can predict performance by less than 10% error.

6.6.4 Speedup and Efficiency

Figure 6.48 and table 6.4 shows the relative speedups for scalable 128x128 shallow water problem. We see that efficiencies of over 94% are obtained up to 128 processors. This gives an indication that the communication costs have minimal impact.

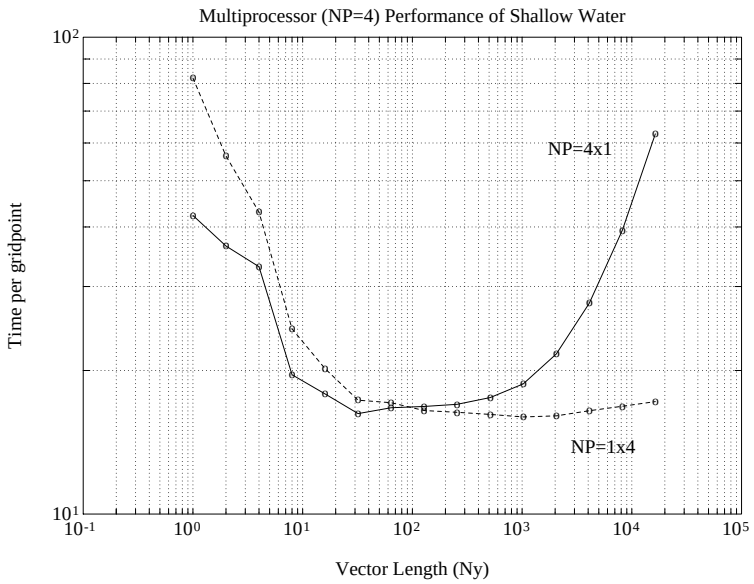


Figure 6.39: Performance of various 4-processor decompositions for Shallow Water.

6.6.3 Performance Model

To model the communication effects of the Shallow Water equations we can use the stencil pattern established in section 6.6.2 and figure 6.35. The Shallow Water application requires each (Ny x Nx) processor subgrid to communicate 7 messages of length Nx and 7 messages of length Ny at each iteration to share the field variables (P, U, V, Cu, Cv, H, and Z,) occurring at the subgrid boundaries. All the communication is to nearest node (assuming we have mapped the grid problem using Gray coding.) Thus, if $T_s(Nx, Ny)$ denotes the time required to complete all communication necessary for one time step of the shallow water equations that, then

$$T(Nx, Ny) = \begin{cases} 0 & \text{if } NPx \text{ and } NPy = 1 \\ 7 * T(Nx) & \text{if } NPy = 1 \text{ and } NPx > 1 \\ 7 * T(Ny) & \text{if } NPx = 1 \text{ and } NPy > 1 \\ 7 * T(Nx + Ny) & \text{if } NPx > 1 \text{ and } NPy > 1 \end{cases} \tag{6.24}$$

given that the processor subgrids as specified as Nx by Ny and the processors are arranged in a NPx by NPy grid.

From the modeling analysis we conducted in section 6.4, we can describe the communication latency for circular-shift type operations encountered in the 2D communication of the shallow water application. We use the standard csend/crev primitives to better illustrate the communication effects. Here the communication bandwidth drops to about 1/2 of 2.8 MB/sec, since each node has two send/recv pairs but can process only one send/recv at a time (see section 6.4.) The communication latency is given by

$$T(n) = 2 * T_f(n) \tag{6.25}$$

$$= 344 + 0.712 * n \ \mu\text{secs}. \tag{6.26}$$

We will use this model to characterize the boundary communication in the following experiments.

Since the communication and computation can be separated, we can use the computation results from section 6.6.1 to analyze the multiprocessor cases. Here we could have used a system similar to CMPS system we developed for the CM-2 (section 3.5); however, the analysis required to assess communication costs is much less, since the communication pattern is explicitly stated by the programmer. (In the CM-2, the gridpoint communication is inserted by the compiler making any paper-and-pencil analysis much

Table 6.3: Multiple Processor Decompositions for shallow. Here NP denotes the total number of processors; NPx, and NPy denotes the processor grid configuration; Mx and My denote the total problem domain; Nx and Ny denote the processor subproblem domain.

Four-Processor Decompositions for Shallow Water Application

NPx	NPy	Nx	Ny	Mflops (per proc)	Time/Nx*Ny (usecs)
1	4	1	16384	3.776	17.212
1	4	2	8192	3.861	16.833
1	4	4	4096	3.946	16.471
1	4	8	2048	4.007	16.222
1	4	16	1024	4.024	16.153
1	4	32	512	3.993	16.279
1	4	64	256	3.896	16.683
1	4	128	128	3.889	16.715
1	4	256	64	3.772	17.229
1	4	512	32	3.719	17.477
1	4	1024	16	3.166	20.528
1	4	2048	8	2.583	25.162
1	4	4096	4	1.462	44.458
1	4	8192	2	1.097	59.261
1	4	16384	1	0.736	88.253
4	1	1	16384	1.036	62.714
4	1	2	8192	1.651	39.365
4	1	4	4096	2.348	27.682
4	1	8	2048	3.001	21.660
4	1	16	1024	3.467	18.747
4	1	32	512	3.704	17.550
4	1	64	256	3.826	16.988
4	1	128	128	3.866	16.812
4	1	256	64	3.890	16.708
4	1	512	32	4.002	16.242
4	1	1024	16	3.639	17.862
4	1	2048	8	3.317	19.594
4	1	4096	4	1.969	33.015
4	1	8192	2	1.782	36.485
4	1	16384	1	1.537	42.286

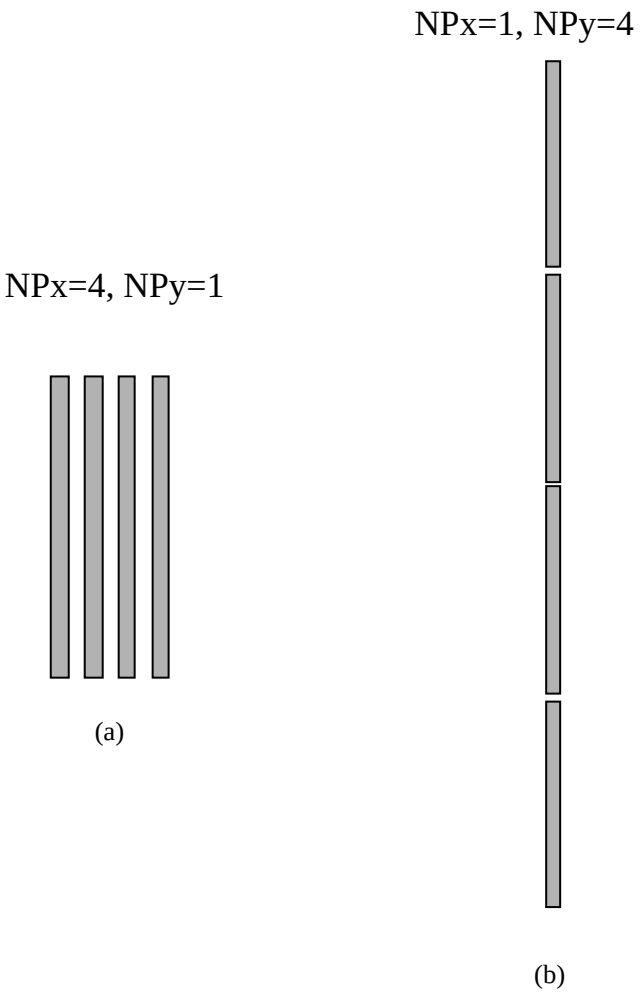


Figure 6.38: Two 4-processor decompositions for Nx=1, Ny=16K processor subgrids. Case (a) has considerably more communication than (b).

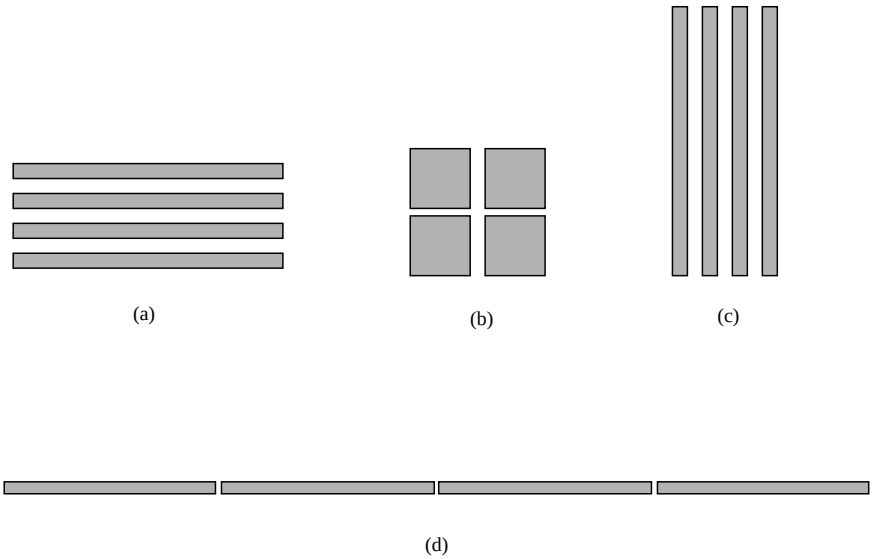


Figure 6.36: Various data decompositions for 2D subgrids.

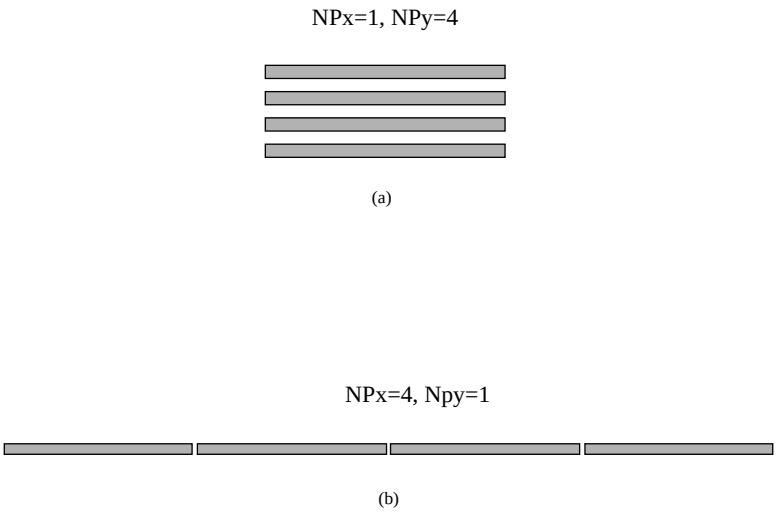


Figure 6.37: Two 4-processor decompositions for Nx=16K, Ny=1 processor subgrids. Case (a) has considerably more communication than (b).

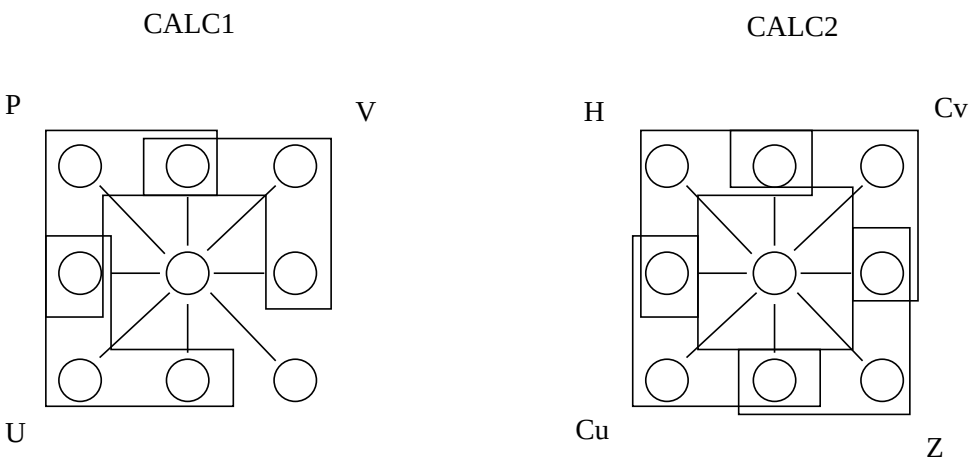


Figure 6.35: Communication patterns for finite difference calculation of primary fields in Shallow Water Application.

6.6.2 Multiple Node Performance

The multiple node section has each node running an similar subgrid size. Here we discuss the various boundary communication effects due to the aspect ratios of the subgrids. By varying NPx, NPy, Nx, and Ny, we can study the communication and computation effects of the parallel decomposition. The communication latency is dependent on the boundary communication (perimeter) of the processor Nx by Ny subproblems , while the computation is dependent on the grid interior (area.) For higher dimensional problem we refer to this as the surface and volume, respectively.

Here we take a Mx by My problem domain and block decompose it onto a NPx by NPy processor grid, each processor receiving a Nx (= Mx/NPx) by Ny (= My/NPy) subproblem. For simplicity, we assume that NPx and NPy are powers of two, and that NPx divides Mx and NPy divides My. Assigning a Nx=1, Ny=16K subgrid to each processor, and laying out the processors in a Px1 column grid, we see that the communication will dominate the performance. Alternately, we can assign a square Nx=128, Ny=128 subgrid to each processor, in an effort to minimize boundary communication. Some of these decompositions are shown in figure 6.36.

These array access patterns exhibited by the finite difference calculations in this application are shown in figure 6.35. Two main subroutines (CALC1, and CALC2) approximate partial derivatives of field variables. (See section 2.6 for details of the shallow water application.)

As an example, consider various 4-processor implementations. The case where NPx=4, NPy=1, Nx=16K, and Ny=1 is shown in figure 6.37(a), requires to communicate only one word along the horizontal, while the second case (NPx=1, NPy=4) requires communication of 16K words at each horizontal boundary. Contrast this with the case where Nx=1, and Ny=16. A similar decomposition yields figure 6.38. The NPx=1 by NPy=4 case (a) has considerable more communication than (b). Due to these variations in the boundary communication, we see the characteristic performance curves as shown in table 6.3 and figure 6.39.

In the case where NPx=1, and NPy=4 (labeled as NP=[NPx]x[NPy]) in the following figures,) the latency per gridpoint starts out at 88.25 μsecs and decreases to roughly 17 μsecs as the vector length of the processor subgrid (Ny) increases from 1 to 16,384. The NPx=4, NPy=1 illustrates a similar behavior: the first half of the graph ($1 \leq Ny \leq 64$) shows little difference from the single processor version (hence communication overhead is minimal,) but soars to almost three times the computation cost when Ny is near 16K.

For a given subgrid decomposition (Nx,Ny), the processor configuration does not matter much , as long as $NPx > 1$ and $NPy > 1$. Due to the cyclic boundary conditions of this application, if one of these is 1, then the 2D communication primitives realize that a physical communication is not necessary. (We saw a similar behavior with the CM in chapter 3.)

```
.
.
.
fld.l -140(fp), f18
ld.l -164(fp), r29
ixfr r5, f26
fmlow.dd f18, f16, f20
ixfr r10, f8
ld.l -176(fp), r23
fxfr f20, r28
adds r28, r29, r30
shl 2, r30, r16
fld.l r11(r16), f22
fld.l r14(r16), f23
adds -164, fp, r16
fadd.ss f23, f22, f24
fmul.ss f24, f2, f25
fmlow.dd f18, f26, f28
fxfr f28, r17
adds r17, r29, r18
shl 2, r18, r19
fld.l r12(r19), f30
adds -140, fp, r17
fmul.ss f30, f25, f31
fmlow.dd f18, f8, f10
.
.
.
```

Figure 6.34: The poor floating point performance of each iPSC/860 can be traced to the current limitations of optimizing compilers for RISC architectures. Here an excerpt of code generation of calc1(), using PGI 1.3a Fortran compiler on on the iPSC/860 shows no use of pipelined floating point operations or pipelined floating point loads.

the i860 is capable. This limitation stems mainly from the underutilization of memory access and lack of effective pipelining of floating point operations. As shown in figure 6.34, the assembly code generated by the PGI vectorizing Fortran compiler (version 1.3a) does not employ pipelined floating point operations, nor pipelined floating point loads.

To determine if the performance was memory bandwidth limited, we ran a completely cache-contained version of shallow on a single i860 processor. We used small 6x6 grids, so that all 14 variable fields could easily fit within the 8K on-chip data cache. The performance results yielded only 3.65 Mflops. This lead us to believe that the performance is limited by the lack of pipelined floating point computation (4 cycles for double precision multiplied and 3 cycles for double precision adds) and redundant array index calculations. Indeed, the Fortran PGI 1.3a compiler generates 233 instructions for the main loop of calc1(), but only 23 (10%) of these are actually floating point operations. The rest (90%) are load, writes, and integer operations related to the array index computations. Assuming the best case, where the integer operations can be effectively pipelined to one instruction per cycle, and double precision floating point computations take 3.5 cycles on average, the 23 floating point operations would take roughly 80 cycles and the remaining 210 integer/load/store instructions would take 210 cycles, yielding only 3.1 Mflops for the calc1() subroutine. The other two subroutines, calc2() and calc3(), have much higher flop ratings thus bringing the total performance up to 3.65 for the complete application.

Table 6.2: Performance of Shallow Water equations on single node of iPSC/860 for various data decompositions.

Nx	Ny	Mflops	Time/gridpoint (microseconds)
1	16384	3.796	17.124
2	8192	3.883	16.740
4	4096	3.969	16.376
8	2048	4.072	15.963
16	1024	4.099	15.859
32	512	4.058	16.016
64	256	4.036	16.104
128	128	4.040	16.089
256	64	3.960	16.412
512	32	4.063	15.999
1024	16	3.692	17.605
2048	8	3.352	19.390
4096	4	1.974	32.923
8192	2	1.789	36.323
16384	1	1.542	42.149

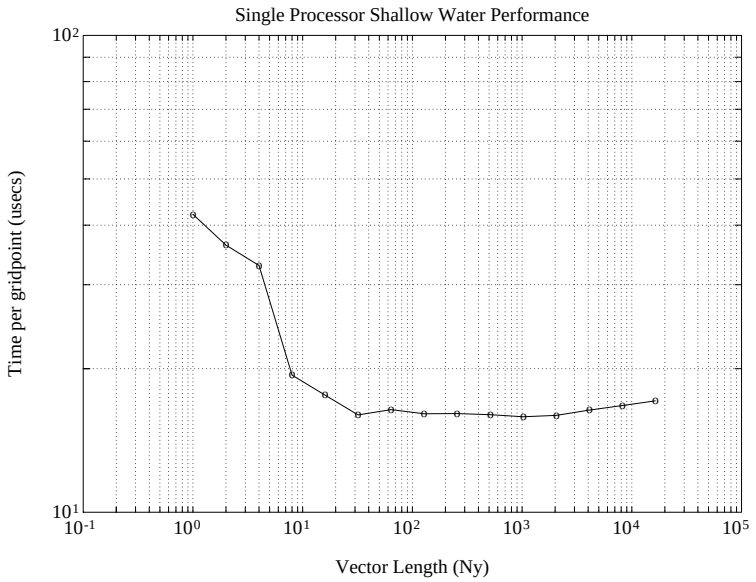


Figure 6.33: Single processor shallow water performance.

```
      do 100 j=1,Nx
          do 100 i=1,Ny
              Cu(i,j) = .5 *(P(i,j) + P(i-1,j)) * U(i,j)
              Cv(i,j) = .5 *(P(i,j) + P(i,j-1)) * V(i,j)
              Z(i,j) = (fsdx*(V(i,j)-V(i-1,j)) - fsdy*(U(i,j)
     >                 -U(i,j-1)))/(P(i-1,j-1)+P(i,j-1)+P(i,j)
     >                 +P(i-1,j))
              H(i,j) = P(i,j) + 0.25*(U(i+1,j)**2 + U(i,j)**2
     >                  +V(i,j+1)**2 + V(i,j)**2 )
100     continue
```

Figure 6.32: One of the main loops of shallow

6.6 Performance Analysis of Shallow Water Equations

To study the various data mapping schemes of this application onto the iPSC/860, we utilize the tools developed in section 2.4.1 to decompose the 2D problem domain of the Shallow Water application onto the hypercube. The spatial decomposition of a Mx*My problem size gets mapped into a 2D NPx by NPy grid of processors, each processor receiving Nx by Ny subgrid, where Nx = P/NPx and Ny = NP/NPy. This allows us to study the effects of communication (along boundary of processor subgrids) and computation (area of processor subgrids.) In the case where NPx or NPy is 1, we have a simple 1D decomposition. In all other cases, we have a 2D block ordering of the original problem grid.

In this study, we scale the problem with the machine size. This is generally more useful, since we attempting to solve larger problems with parallel machines. We assign to each processor a fixed number of grid points, 16K as a Nx*Ny array. By varying the "aspect ratio" of this grid together with the processor grid (NPx * NPy) one can simulate different computation/communication ratios and study the effects of various ways of decomposing the same problem for a given machine size.

We ran a double precision version of shallow on the Intel iPSC/860 at Beaverton, Oregon, and at NASA Ames using the Portland Group's PGI Fortran compiler v. 1.3 with optimization flags `"-O3 -Mvect"`.

6.6.1 Results of Single Node Performance

To determine the computational behavior of shallow, we first study the performance on a single processor. This allows us to isolate the communication effects from the computational performance, since the application uses blocking message passing primitives. The communication routines of the 2D grid package developed in section 2.4.1 have the capability to bypass communications if the destination processor and source processor are identical. In this manner, we can study the single-node behavior of a parallel program without any modification to the source code. Thus we have a concise method of factoring out the communication from the computation.

The performance results for a single node are shown in Table 6.2; a graphical representation appears in figure 6.33. With a traditional pipelined vector processor, one would see the best performance for the longest vector case (Ny = 16K); however, we notice that the best performance occurs for shorter vectors. This is due in part to effects of the 8Kbyte on-chip cache. The calculations in this application tend to reuse some data and hence benefit from smaller vectors that remain in cache longer. For example, in subroutine `calc1()`, shown in figure 6.32, the variables P(i,j), V(i,j) and U(i,j) are reused at least three times.

Thus we see that the single node performance involves a subtle tradeoff between an affinity for longer vectors (to increase pipeline throughput) and shorter vectors (to retain data in cache longer.) The floating point performance observered of 4 Mflops is only 10% of the peak 40 Mflops (double precision) of which

subgrid. Also note that, unlike the sequential version above, zero offset indices appear on the left-hand of the assignment. This translates to performing 'reads' as opposed to 'writes' of neighboring boundary values, and provides a more uniform local view of the per-processor computation.

```
    call SendSouth(P)
    call SendSouth(V)
    call SendEast(P)
    call SendEast(U)
    call SendNorth(U)
    call SendWest(V)

    do 100 j=1,Ny
        do 100 i=1,Nx
        Cu(i,j) = .5 *(P(i,j) + P(i-1,j)) * U(i,j)
        Cv(i,j) = .5 *(P(i,j) + P(i,j-1)) * V(i,j)
        Z(i,j) = (fsdx*(V(i,j)-V(i-1,j)) - fsdy*(U(i,j)
     >          -U(i,j-1)))/(P(i-1,j-1)+P(i,j-1)+P(i,j)
     >          +P(i-1,j))
        H(i,j) = P(i,j) + 0.25*(U(i+1,j)**2 + U(i,j)**2
     >           +V(i,j+1)**2 + V(i,j)**2 )
100 continue
```

Figure 6.31: iPSC/860 version of subroutine CALC2.

6.5 Shallow Water Equations

6.5.1 Implementation Issues

Because of certain implementation choices made in the original F77 version (see previous section), the hypercube version differs to some extent. Since the model uses 2D doubly periodic grids, the shallowf77 implementation actually allocates (M+1,N+1) size grids. This supposedly facilitates the updating of the borders, although the periodic conditions still have to be processed seperately.

The iPSC version of shallow consists of five major parts:

1. computes the initialization routine (INTIAL) and the first time step (CALC1, CALC2, and CALC3Z) on the front end.

2. It then distributes each field (U, V, P, and so on) among the nodes so that each processor gets a proper sublock of the original field.

3. The host then sends the number of iterations (actually, nit-1, since the first time step has already been computed) to each node and the processors crank away at the remaining iterations.

4. 2D grid node communication is handled by the packaged developed in section 2.4.1.

5. Finally, each node sends its subgrid of each field back to the host for printing and final processing.

Steps (1) - (3) are performed on the host, running the program `shallow_host.f`, while the nodes execute the `shallow_node.f` program which performs the shallow water calculations in block-ordered form on the nodes.

Slight modifications were made to the sequential Fortran loop codes to accommodate the block-ordered distribution of grids onto processor nodes. For example, the (i,j) indices appearing in the first computation loop of each time step, CALC1, appear in figure 2.11.

Periodic boundary conditions are enforced by manually copying the boundary edges in the second half of the subroutine. In the iPSC version, each node computes a similar calculation, but over a limited domain local to that processor only (shown in figure 6.31).

Notice that the periodic boundary conditions are handled *before* the loop update. Furthermore, the (i,j) indices run over 1 to Nx and 1 to Ny, respectively, where Nx and Ny denote the sizes of the per-processor

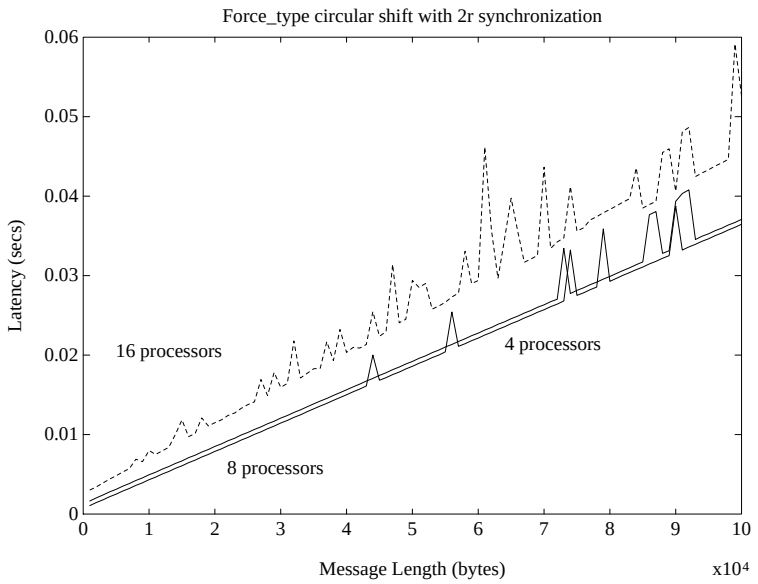


Figure 6.30: Forward synchronizing FORCE-TYPE communication latencies for 4, 8, and 16 processors.

relationship,

$$T(n) = 1345 + 0.357 * n \ \mu\text{secs}. \tag{6.17}$$

The few straggling data points occur because there is the rare opportunity on the second time around the ring that the incoming `FORCE_TYPE` message gets processed before the second synchronization `csend()` gets sent out. (See figure 6.27. This is possible because the `irecv()` first posted can be service via an interrupt. This happens rarely, so that the average behavior over many runs becomes minimal. Modulo the five (out of 100) data points that stray from the straight line for `FORCE_TYPE` messages in figure 6.29, the data fits to the above linear model with less than 1% relative error.

Since the forward synchronization mechanism we have described for `FORCE_TYPE` message requires two complete trips around the ring, we can decompose the total latency, $T_{2\mathrm{r}}$, as

$$T_{2\mathrm{r}} = subTsynch + T_{\mathrm{r}} \tag{6.18}$$

where

$$T_{\mathrm{synch}} = 2 * P * T_{\mathrm{s}}(0). \tag{6.19}$$

By experimentation, we determine the time to send a null message, $T_{\mathrm{s}}(0)$, to be

$$T_{\mathrm{s}}(0) = 74.38\,\mu\text{secs} \tag{6.20}$$

which agrees with similar experiments by Bokhari [9]. Using the model we developed in section 6.3.2, we can predict the behavior of the circular shift communication pattern on the iPSC/860 as follows. For the example of P=8, our model predicts

$$\begin{aligned} T_{2\mathrm{r}} &= 2 * P * T_{\mathrm{s}}(0) + T_{\mathrm{f}} && (6.21)\\ &= (2 * 8 * 74.38) + (175.0 + 0.356 * n) && (6.22)\\ &= 1372.3 + 0.356 * n \ \mu\text{secs} && (6.23) \end{aligned}$$

which deviates from the observered data (eqn. 6.17) by less than 2% error for short messages, and less than 1% error for long messages. Moreover, the peak bandwidth predicted by the model in expression 6.23, 2.81 MB/sec, agrees with quoted peak bandwidth, 2.80 MB/sec, with less than 1/2% error.

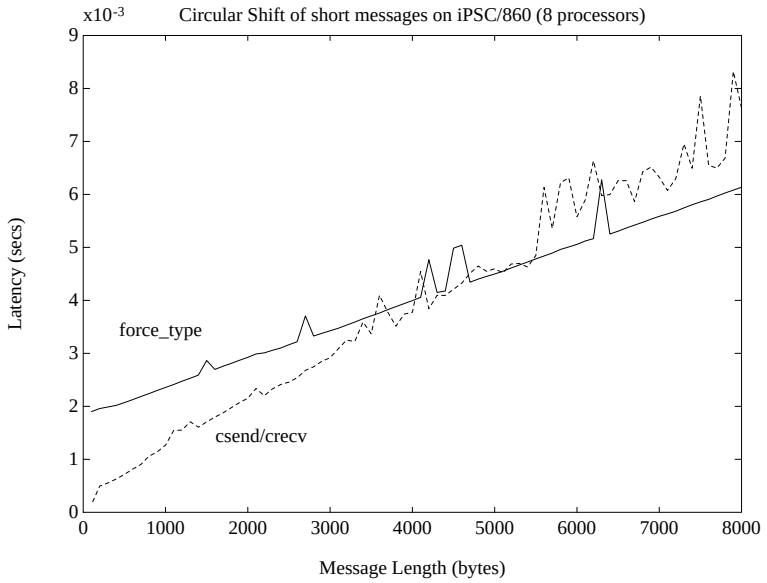


Figure 6.28: Circular Shift of Short Messages on iPSC/860 (8 processors.)

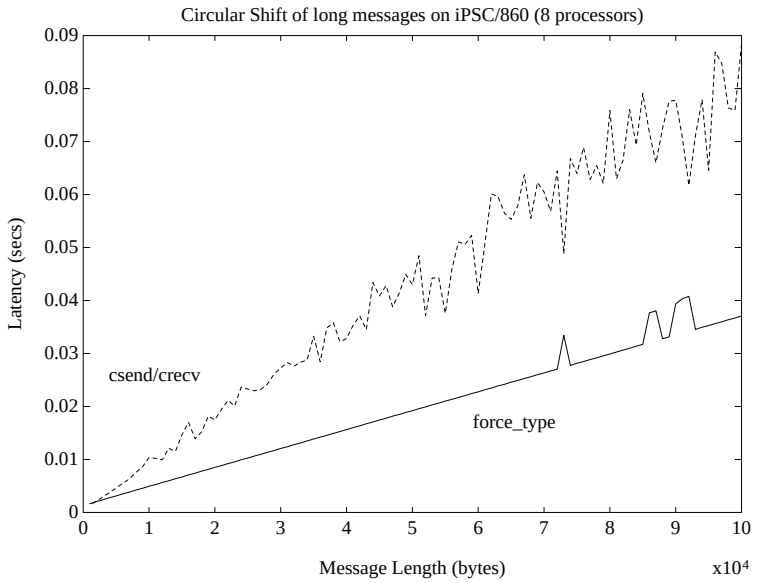


Figure 6.29: Circular Shift of Long Messages on iPSC/860 (8 processors.)

```
            do 150 i=1, N
                msgid = irecv(FORCE_TYPE, msgbuff, msglen)

c..............send null message once around ring TWICE (forward only)
c               to synchronize.

                if (menode.eq.0) then
                   call csend(SYNC_TYPE, msgbuff, 0, nextnode, mepid)
                   call crecv(SYNC_TYPE, msgbuff, 0)
                   call csend(SYNC_TYPE, msgbuff, 0, nextnode, mepid)
                   call crecv(SYNC_TYPE, msgbuff, 0)
                else
                    call crecv(SYNC_TYPE, msgbuff, 0)
                    call csend(SYNC_TYPE, msgbuff, 0, nextnode, mepid)
                    call crecv(SYNC_TYPE, msgbuff, 0)
                    call csend(SYNC_TYPE, msgbuff, 0, nextnode, mepid)
                endif
c..............

c.......make this a forcetype, so that it bypasses 3-way protocol
                call csend(FORCE_TYPE, msgbuff, msglen, nextnode,
     >                  nextpid)

                call msgwait(msgid)
150         continue
```

Figure 6.27: Implementation of a forward-only synchronization algorithm for maximizing communication bandwidth for circular shifts.

6.4.4 A Forward Synchronization Model

To remedy this situation we investigate the use of a forward synchronization model that attempts to decrease communication latency by eliminating the conventional 3-step protocol associated with long messages. A message may be declared as a special `FORCE_TYPE` which will be sent directly to the receiving processor without the exchange of probes and acknowledgements. The catch is that it is then the programmer's responsibility to ensure that a matching `irecv()` has already been posted on the receiving node. If not, the incoming message is lost since there is no designation for where it should go. (This is part of the purpose of the 3-step protocol.) This synchronization is rather tricky to achieve, however, due to asynchronicity of the nodes' clocks. A simple global synchronization call, `gsynch()` , will generate "backward" messages that cause collisions and hence will be of no use to decrease latency. One solution is to use a "forward only" synchronization mechanism that ensures the matching `irecv()`'s have been posted, while avoiding the generation of any "backward" messages. The simplest of these methods involves passing a synchronization token around the ring, so that each processor knows when it is safe to send a `FORCE_TYPE` message without worrying about it being lost. A sample code is given in figure 6.27.

The comparative results for both methods are shown in figure 6.28 for short messages, and 6.29, for long messages. For long messages, we see that the benefits of the improved communication bandwidth far outweighs the slight overhead in passing the synchronization token (ideally a null message) twice around the ring. We see that for messages longer than 4Kbytes, it is advantagous to use the forward synchronizing algorithms.

Figure 6.30 illustrates the communication latencies for forward synchronizing `FORCE_TYPE` messages using 4, 8, and 16 processors. We notice that both figures shown do not exhibit the communication behavior as straight lines; there are nonlinear variations in the data. This is due in part to the asynchronicity of message collisions explained in section 6.3.2. We note, however, that the `FORCE_TYPE` message more closely exhibits linear behavior. For example, in figure 6.29, only a few data points deviate from the underlying linear

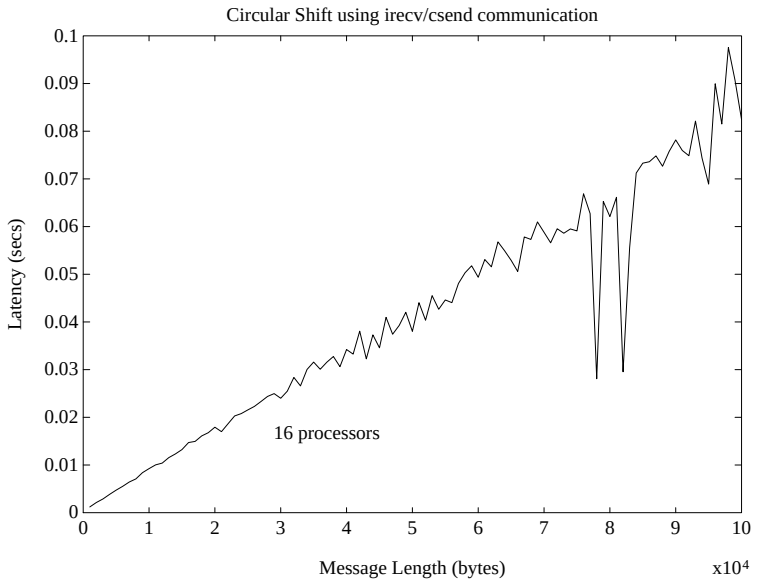


Figure 6.25: Using irecv/csend communication pair illustrates that latency has been increased due to race conditions between colliding messages of the 3-step protocol.

```
            do 150 i=1, ringcount
                msgid =irecv(NODETYPE, msgbuff, msglen)
                msgid2 = isend(NODETYPE, msgbuff2, msglen, nextnode,
     >                  nextpid)
                call msgwait(msgid)
call msgwait(msgid2)
150         continue
```

Figure 6.26: Example code of irecv/isend communication pair.

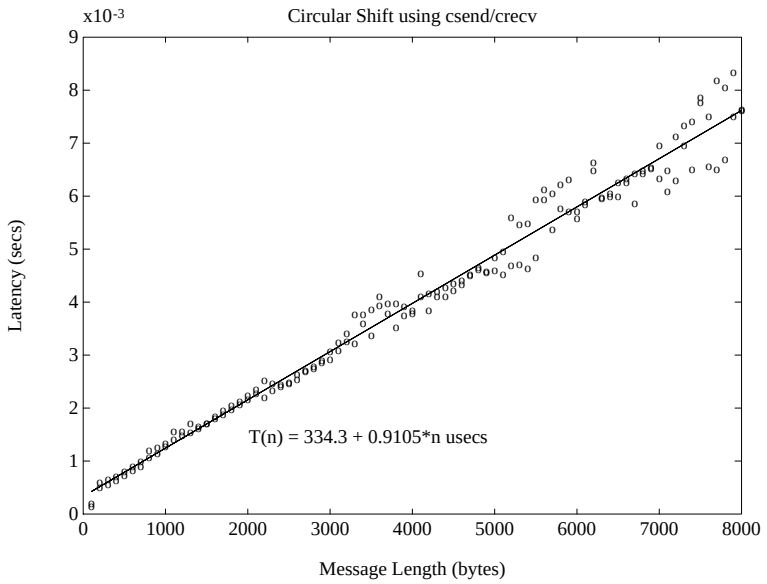


Figure 6.23: Circular Shift using csend/crecv for various number of processors

```
            do 150 i=1, ringcount
                msgid =irecv(NODETYPE, msgbuff, msglen)
                call csend(NODETYPE, msgbuff2, msglen, nextnode,
     >                     nextpid)
                call msgwait(msgid)
150         continue
```

Figure 6.24: Example code of irecv/csend communication pair.

Notice that this latency is over 2.5 times greater than that exhibited by the similar sequential ring experiment (eq. 6.7.) The maximum bandwidth predicted by the data is

$$B_\infty = \frac{1}{0.9105} \approx 1.1 \text{ MB/sec}, \tag{6.16}$$

which is quite lower than the 2.8 MB/sec bandwidth expected for long messages.

Using irecv() and isend() primitives

Figure 6.25 illustrates the communication latency when using the `irecv/csend` communication primitives. The actual communication loop is shown in figure 6.24 . Notice that the two low values near 80,000 byte messages shows that indeed fast communication is possible, but generally not obtainable due to race conditions mentioned above. Using `irecv/isend` communication pair (code sample 6.26) produced similar results.

The problem in both of these cases is due to the colliding traffic between message "probes" and "acknowledgements." Because each node can process at most one incoming and one outgoing message, an "acknowledgement" cannot be sent out if the processor is currently receiving a long message. It must be wait for the long to complete before it can return an acknowledgement. This essentially sequentializes incoming and outgoing messages. (See section 6.3 for complete details.)

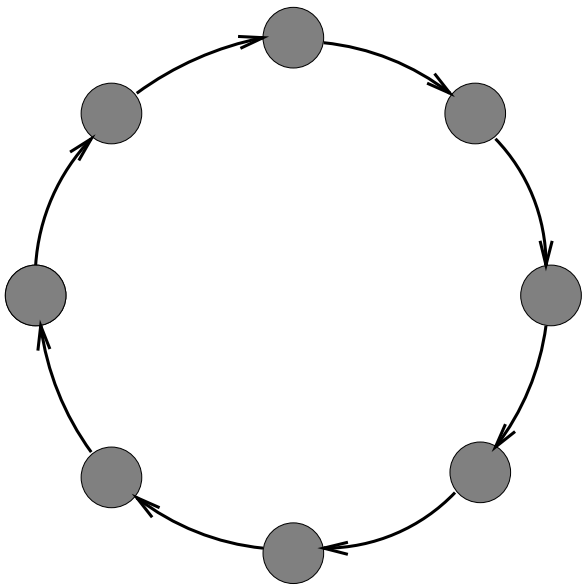


Figure 6.21: One dimensional circular shift operation in which all processors cooperate in communication. Here each processor communicates simultaneously.

```
            do 150 i=1, N
                call csend(NODETYPE, msgbuff, msglen, nextnode,
     >                 nextpid)
                call crecv(NODETYPE, msgbuff, msglen)
150         continue
```

Figure 6.22: Sample iPSC/860 for unidirectional circular shift operation.

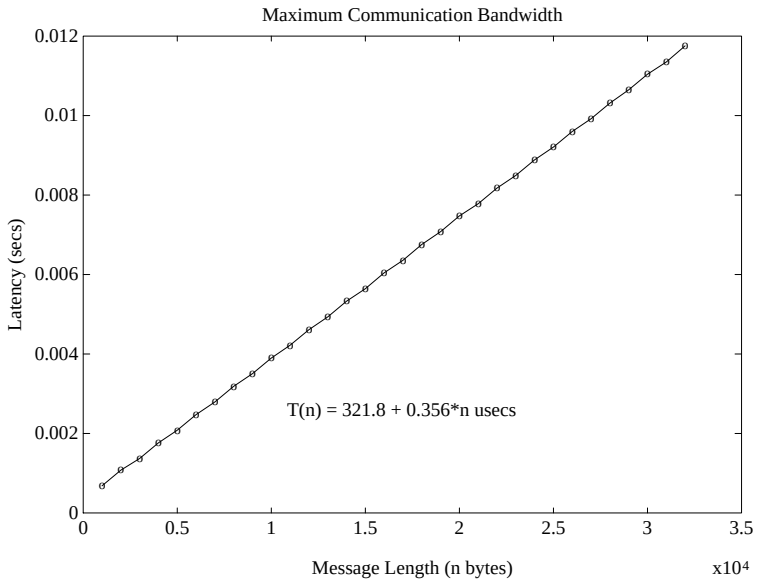


Figure 6.19: Sequential Ring Bandwidth (8 Processors) on the iPSC/860

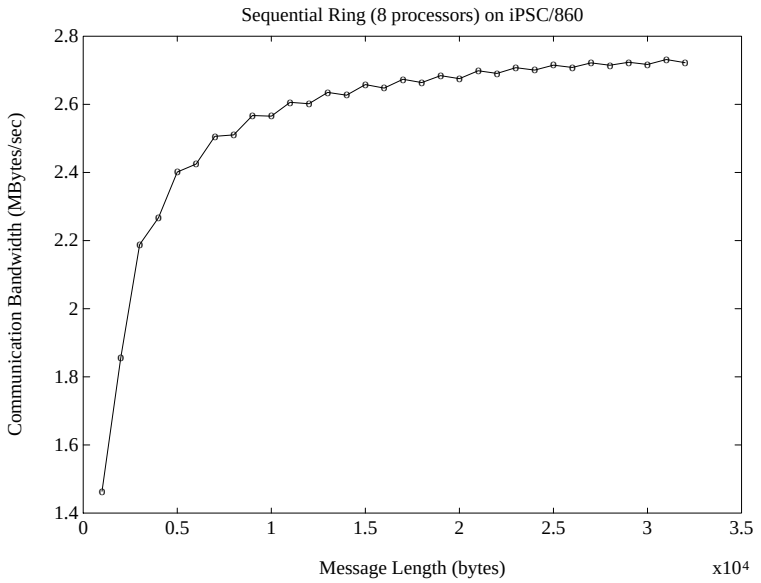


Figure 6.20: Communication bandwidth vs. message length.

data in figure 6.19 was obtained using an 8 processor ring, although rings of other sizes yielded identical results. Because there is no contention for resources (no two processors attempt a send at the same time), we can conclude that this is the maximum communication bandwidth obtained in practice. The communication latency for each data channel can be given by

$$T_l(n) = 321.8 + 0.356 * n \ \mu\text{secs}. \tag{6.7}$$

The predicted communication bandwidth for significantly long messages is

$$B_\infty \ = \ \frac{1}{0.356} \ = \ 2.81\,\text{MB/sec} \tag{6.8}$$

which agrees closely with the maximum specification bandwidth of 2.8 MB/sec. The relative error of the least squares fit of this data to a linear model is less than 4% and generally less than 2% for the longer (5K - 35K) messages.

For short messages (< 100 bytes) that do not entail the 3-step protocol overhead, the communication latency has been shown by Bokhari [9] to be

$$T_s(n) = 75 + 0.425 * n \ \mu\text{secs} \tag{6.9}$$

for $0 \leq n \leq 100$.

Given that for long messages a 3 step protocol is encountered, we can deduce that there are three message startup costs (see section 6.3.2.) The probe and acknowledgement messages are rather short (about 28 bytes) and hence their transmission costs are dominated by startup time, $T_s(0)$. Thus we can approximate that T_l consists of three message startups (probe + acknowledgement + actual long message.) If we denote the cost of sending a long message *without* the 3-step protocol as T_f, we have

$$T_l(n) = T_s(0) + T_s(0) + T_f(n) \tag{6.10}$$

thus

$$\begin{aligned} T_f(0) &= T_l(0) - 2T_s(0) && (6.11)\\ &= 322 - 150 \ \mu\text{secs} && (6.12)\\ &= 172 \ \mu\text{secs} && (6.13) \end{aligned}$$

giving

$$T_f(n) \approx 172 + 0.356 * n \ \mu\text{secs}. \tag{6.14}$$

6.4.3 Circular Shift Experiment

To understand the communication behavior with relaxation-type patterns, we study a circular shift example in which every processor simultaneously communicates along a 1D ring. This is a simplification of the higher 2D and 3D relaxation kernels.

To minimize contention, we map the processors of the 1D ring using a Binary Gray Reflected Code so that adjacent neighbors on the ring get mapped to adjacent processors in the hypercube. Each processor then sends a message to its left (or right) neighbor. This is one of the simplest communication patterns where *every* processor participates. Its communication pattern in shown in figure 6.21. Each node sends to its next neighbor up the line and receives from its neighbor down the line, using the `csend()` and `csend()` communication primitives as shown in the code excerpt in figure 6.22. Here `nextnode` refers to the next gray mapped neighbor, and `N` denotes the number of repetitions.

The results of this experiment for various number of processors are shown in figure 6.23. A least squares fit to the data shows that the communication latency can be modeled as a linear function of the message length as

$$T(n) = 334.3 + 0.9105 \ \mu\text{secs}. \tag{6.15}$$

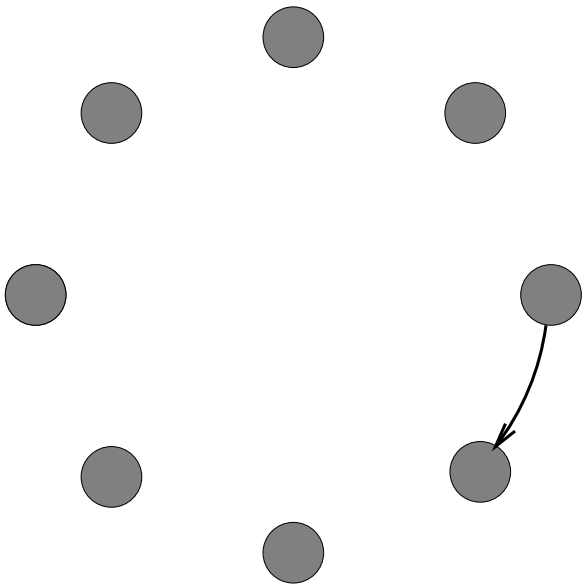


Figure 6.18: One dimensional sequential ring communication example. Communication occurs at only one node at a time.

6.4 Communication Model

6.4.1 Linear Models

Assuming no contention for resources, the transmission of an n-byte message between nodes can generally be modeled as

$$T(n) = T_{\text{startup}} + T_{\text{transmission}} * n \tag{6.3}$$

where T_{startup} denotes a fixed overhead for establishing the connection and $T_{\text{transmission}}$ denotes a per-byte transmission cost. The effective bandwidth, $B(n)$, defined as

$$B(n) \equiv \frac{n}{T(n)} \tag{6.4}$$

denotes the number of bytes/sec transmitted and is a convenient way to measure effectiveness of the communications network. For sufficiently long messages, the peak bandwidth, B_∞, is given by

$$B_\infty \equiv \lim_{n \to \infty} B(n) = \frac{1}{T_{\text{transmission}}} \tag{6.5}$$

6.4.2 Sequential Ring Experiment

To determine the peak communication bandwidth sustained between nodes, we devise a simple experiment where we arrange the nodes into a 1D ring and circulate a single token around the ring. Each node is Gray mapped onto the 1D ring, and simply passes the message from its left neighbor to its right neighbor, as shown in figure 6.18. If a ring with P processors takes T seconds for a message of L bytes to complete one cycle, then we can conclude that the interprocessor communication bandwidth is given by

$$L/(T * P)\text{bytes/sec}. \tag{6.6}$$

In figure 6.19 we show the results for such an experiment. Because the results are normalized by P in equation 6.6, the outcome of the experiment is independent of the actual number of processors used. The

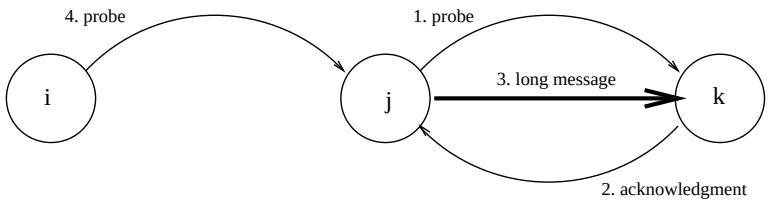


Figure 6.17: An exmaple of colliding messages under the 3-step protocol for long messages.

4. node i send a probe to node j

Because there can be only one outgoing message from a node, node j must complete the transmission of its long message to node k before returning an acknowledgment to node i. Thus, node i cannot begin transmitting before node j completes its message, and the messages are effectively sequentialized. The effect this has to synchronous circular-shift communication patterns is further discussed in section 6.4.3.

6.3.4 Buffering of Messages

In certain cases, the NX node operating system may buffer messages into temporary locations, while the completion of that operation is pending. For example, if two processors require to exchange information both could execute a `csend()` followed by a `crecv()`. The `csend()` sends a message to the receiving processor, but since the final destination is yet undetermined (the corresponding `crecv()` has not been executed yet,) the incoming message must be copied into a system buffer until the corresponding `recv` is encountered. At that point the message is copied from the system buffer into the application buffer [51].

This message copying can avoided by using a non-blocking `irecv()`, first and then use `csend()` primitive to send the message. Once the message arrives at the destination processor the message can be copied directly into the application buffer if the corresponding `irecv()` has already been posted.

We have conducted several experiments using blocking and non-blocking sends and recvs and noticed little difference between both methodologies. Performance improvements would be perhaps be more noticable on the iPSC/2 because of the slower Intel 80386 CPU, but given the speed of the Intel i860 microprocessor this memory-copying aspect of the communication latency is less noticable.

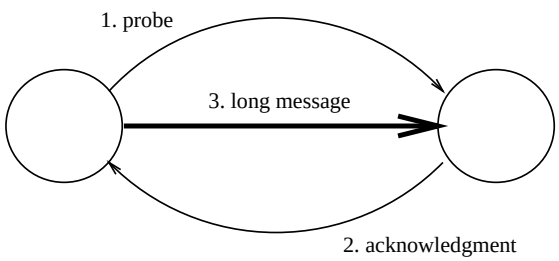


Figure 6.15: Three step protocol encountered for messages longer than 100 bytes. First, a probe message is sent to receiving node, it then responds with an acknowledgement before the transmission of the full message.

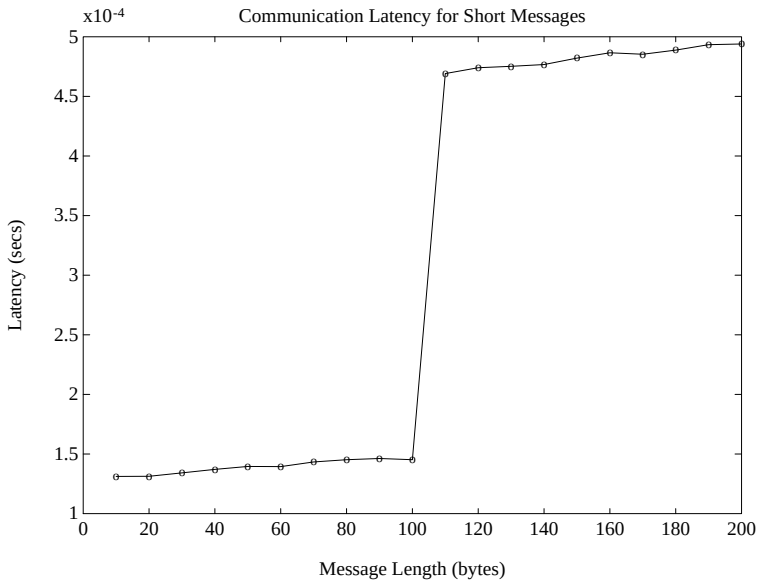


Figure 6.16: Increased communication overhead is encountered for messages longer than 100 bytes due to invocation of the 3-step handshaking protocol.

6.3 Communication Architecture of the iPSC/860

6.3.1 Gray Coding and E-cube Message Routing Algorithm

Message routing on the iPSC/860 is governed by the e-cube algorithm. Given a departure node, n_i, and a destination node, n_j, the path taken through intermediate nodes is unique and determined by the following algorithm: starting with the right hand side of the processor number (in binary), move to the processor whose number most closely matches the number of the destination processor n_j.

For example, the path taken when processor #4 (100_2) sends a messge to processor #79 (1001111_2) is

$$0000100_2 \rightarrow 0000101_2 \rightarrow 0000111_2 \rightarrow 0001111_2 \rightarrow 100111_2 \tag{6.2}$$

which corresponds to the path $4 \rightarrow 5 \rightarrow 7 \rightarrow 15 \rightarrow 79$.

It is most efficient, therefore, to map subproblem onto processors in such a manner that nodes that communicate between each other have minimum paths.

Specifically, Binary Gray Reflective Coding allows one to map a list of integers into contiguous processor labels. It is important to utilize such mapping strategies when implementing N-dimensional grids on a hypercube. In this manner, nearest neighbor subgrids are on adjacent processors. (see section 2.4.1 for further discussion of the Gray-mapped 2D decomposition package for distributed memory machines.

6.3.2 Message Passing Protocol

For messages over 100 bytes long, the NX operating system imposes a three-step "handshaking" protocol. This is required for long messages, to ensure that there is enough space in the system buffers to copy the incoming message. Assuming that processor i needs to send a message to processor j, the protocol consists of the following steps (illustrated in figure 6.15):

1. i first emits a "probe" message to j containing the message length and other information
2. j returns a short acknowledgment message
3. i then transmits long message.

Figure 6.16 illustrates how the communication latency jumps at 100 bytes as this crossover point is traversed.

6.3.3 Channel Limitations of the iPSC/860

The communication subsystem of the iPSC/860 can has only one incoming (and only one outgoing) hardware path between the routing logic and node memory [77]. Many messages can be going by a node but only one coming in and also one going out at a time.

This can degrade the communication performance of long messages by more than a factor of two (see section 6.4.3 , since there is the possibility of colliding messages during the 3-step handshaking protocol employed for messages longer than 100 bytes.

The following example illustrates how the communication performance could inadvertently be cut in half. Suppose node i wants to send a message to node j, and node j also needs to send a message to node k. A simple example of such a pattern is a circular shift communication where processor i simultaneously sends a message to processor $i+1$. The we could have the following situation (depicted in figure 6.17):

1. Node j sends a probe to node k,
2. node k returns acknowledgement,
3. node j begins transmission of long message

```
.L9:
   mov     r6,     r25             //Line 1        r25 = A
   adds    -4,     r25,    r25     //Line 2        r25 = A-4
   mov     r4,     r26             //Line 3        r26 = i
   shl     2,      r26,    r26     //Line 4        r26 = 4*i (byte offset)
   adds    r25,    r26,    r26     //Line 5        r26 = (A-4)+ 4*i
   fld.l   0(r26), f27             //Line 6        f27 = "A(i)"
   fadd.ss f27,    f4,     f4      //Line 7        sum = sum + A(i)
   adds    r5,     r4,     r4      //Line 8        i += stride
   bla     r17,    r16,.   L9      //Line 9        repeat loop if i < N
     nop
```

Figure 6.14: Fortran-860 assembly output (no optimization) for single precision vector sum. Each iteration of the loop requires 12 clock cycles (2.75 Mflops @ 33 MHz.)

An Example

We can use the assembly code for the vector sum studied in section 6.2.1, shown here in figure 6.14. This was the code generated by the Greenhills Fortran i860 compiler. Assuming a cache-contained vector, we can describe the instruction pipelining, integer core and FPU overlap, and memory traffic associate with this computation.

Looking at figure 6.14, we notice that all instructions can complete in one cycle, except the floating point load (line 6) and the floating point add (line 7). Since there are 10 instructions in the basic loop (the `nop` in line 10 is still executed by the `bla` instruction since it is a delayed branch) and would require 10 cycles to complete if there were no sources of delay. The floating point add instructions (`fadd.ss`) in line 7 is not pipelined and hence requires three clock cycles to complete. However, since the three instructions following `fadd.ss` are integer core instruction and do not involve the FPU, they are effectively overlapped and there are no associated stalls. The only remaining source of instruction delay is the `fld.l` in line 6.

We utilize the model as follows. There is only one load (instruction 6), therefore nl=1; In this example the operand for this instruction comes from cache, hence Lt(1) = 2. We then have for each iteration

1. Dl(1) = ∞ (since it is greater than Lt(1))
2. Du(1) =1
3. Dc(1) = MIN (∞ , 1-1) = 0
4. T(1) = MAX (2-0 , 0) = 2

Hence the total execution time is 2 + 10 = 12 cycles, which agrees with the analysis in section 6.2.1.

Synthesizing the Model

The freeze conditions 1.a, 1.b, 2.a, and 2.c can be detected in the following way:

1. Determine the load-time of the load being examined.
2. Check if there is another load in the window defined by the load-time.
3. Check if there is an instruction using the operand being loaded in the window defined by the load-time.
4. Determine the delay time imposed by 2) or 3), whichever occurs first.

The delay obtained, either for conditions 1.a /2.a or 1.b/2.c, will be charged to the memory load as follows. Each memory load in the loop will be numbered from 1 to nl, where nl is the total number of loads in the loop. Two distances will be assigned to each load: the distance to the closest load (dl: load-load distance) and the distance to the closest use of the operand being load (du: load-use distance). The minimum of these two distances will be called the critical distance (dc). If the load-time is longer than the dc, then a freeze condition will occur, and it will take (lt - dc) clocks. If we define Dl , Du, Dc, and Lt to be vectors containing the distances dl, du, and dc, and the load-time lt for each load 'i', we can express this procedure as follows. For cache-loads we have

1. Determine the Lt vector: Lt(i) = 2 if cache hit, Lt(i) = 10 if cache miss and near access, Lt(i) = 13 if cache miss and far access.
2. Determine the Dl vector: Dl(i) = distance to next load.
3. Determine the Du vector: Du(i) = distance to next use of operand loaded in 'i'. Since stalls are only possible in the next immediate operation, or the one following, possible values for Du are $\{1, 2, \infty\}$, where ∞ denotes that the distance is greater than 2.
4. Determine the Dc vector: Dc(i) = min(Dl(i), Du(i)-1).
5. Determine the total time charged to the load: T(i) = 1 + MAX(Lt(i) - Dc(i), 0).

For Pipelined Loads: Lt(i) will contain the access time of the load 'i' and LTt(i) will contain the total pipeline latency that the load 'i' will be subject to.

1. Determine the Lt vector: Lt(i) = 2 if near access, Lt(i) = 5 if far access.
2. Determine the LTt vector: LTt(i) = Lt(i) + Lt(i-1) + Lt(i-2), (i-1, i-2 taken MOD(nl))
3. Determine the Dl vector: Dl(i) = distance to the load 'i+3'
4. Determine the Du vector: Du(i) = distance to the next use of operand loaded in 'i'. Again, possible values for Du are $\{1, 2, \infty\}$. (See cache-load condition above.)
5. Determine the Dc vector: Dc(i) = min(Dl(i), Du(i)-1)
6. Determine the total time charged to the load: T(i) = 1 + MAX(LTt(i) - Dc(i), 0)

For both cases the Total Execution Time of the loop would be:

$$T(1) + T(2) + \ldots + T(nl-1) + T(nl) + REST$$

where REST is the number of branch and execution instructions.

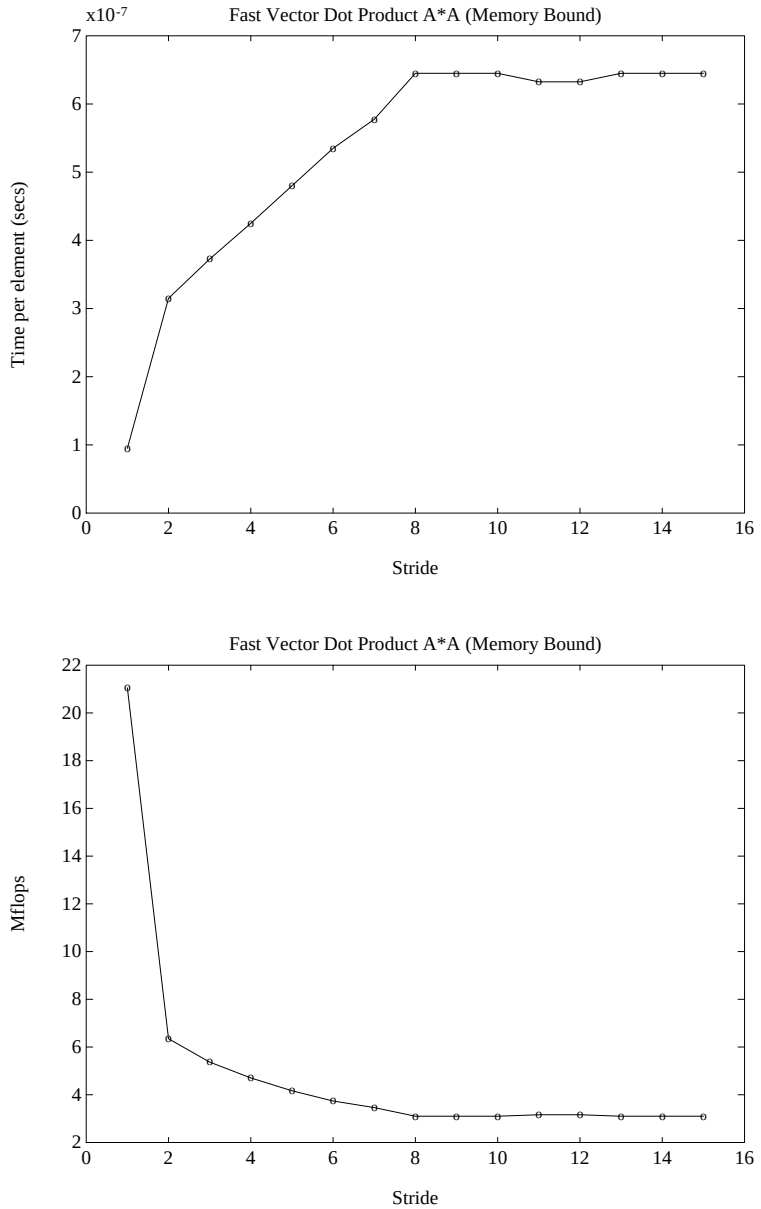


Figure 6.12: Dot Product Performance of A*A is superior to that of A*B since memory traffic is greatly reduced.

1. Cache-Loads	Cache Hit:	2 clocks
	Cache Miss / Near Access:	10 clocks
	Cache Miss / Far Access:	13 clocks
2. Pipelined Loads:	Near Access:	2 clocks
	Far Access:	5 clocks

Figure 6.13: Memory reference latencies for Page Acesses

each vector element reference costs 13 clock cycles (1 "far" access of 5 clock cycles, 3 "near" accesses of 2 clocks each to fill the cache line, and finally 2 clock cycles to bring the data from cache into a floating point register.)

To illustrate how the reduction of memory traffic improves performance, Figure 6.12 shows the performance of the dot product A*A. In this case, the FPA memory page buffer is not thrashed and the overall memory traffic is reduced by half. Thus, we see improved performance: 21 Mflops for unit stride, as opposed to 14 Mflops for the similar A*B example.

Thus, we see that for strides of 4 or greater, data caching actually works against us. Using a pipelined floating load (`pfld`) instruction, one can save other overhead of filling a cache line (3 extra 64-bit loads) and reduce the memory latency to just 5 clock cycles per floating point argument.

6.2.3 Intel i860 Performance Modeling

This section presents some analytical models that characterize the performance of the i860 on numerical loops. The Intel i860 architecture is rather complex and while we do not model every aspect of the architectures, we can capture the main characteristics for vector and matrix loops often found in scientific code. We model the behavior of the 4-level memory hierarchy, the effects of caching and pipeline loading, and instrution shceduling.

Load Instruction Related Freezes

Any instruction on the Intel i860 RISC processor can be executed in one clock unless a freeze condition is invoked. Pipelined floating point operations, for example, complete at one per clock cycle, since they can be fed to the floating point pipeline at a rate of one per clock. To predict the execution time of any loop that is run on the i860, we count the number of instructions and detect the freeze conditions. For simplicity, we only consider the freeze conditions related to operand loads, since preliminary studies have shown that operand fetches are the key factor driving the performance of the system for vector loops. There are two basic types of freeze (delays) that could occur with loading and storing memory elements:

1. Cache-Loads
   (a) Another cache-load is attempted before a previous one is completed: the second load is delayed until the first one is finished.
   (b) An operand is used before its load has completed: the instruction involving the operand being load is delayed until the destination register is written.
2. Pipelined Loads
   (a) A pipelined load is issued when there are three pfld outstanding: the pipelined load is delayed until the data of the first load is returned.
   (b) A pipelined load hits the cache: two clocks.
   (c) An operand is used before it is loaded: same as for the cache load situation.

To simplify our analysis, we will consider loops where all the memory loads are either cache-loads or pipelined loads (note that a cache-load and a pipelined load are transparent to each other: a cache-load can proceed even if a pfld was issued in the previous clock.). The condition 2.b will not be included in this analysis since it involves both types of loads, and it should rarely happen if programming is done correctly. The model that we propose is not a complete one, its purpose is mainly to serve a the basis for further development.

Load-Time: timings of load instructions

The times presented below indicate the number of clocks that a load will be outstanding when utilizing Fast Page Access memory. We will refer to this number as the load-time (lt).

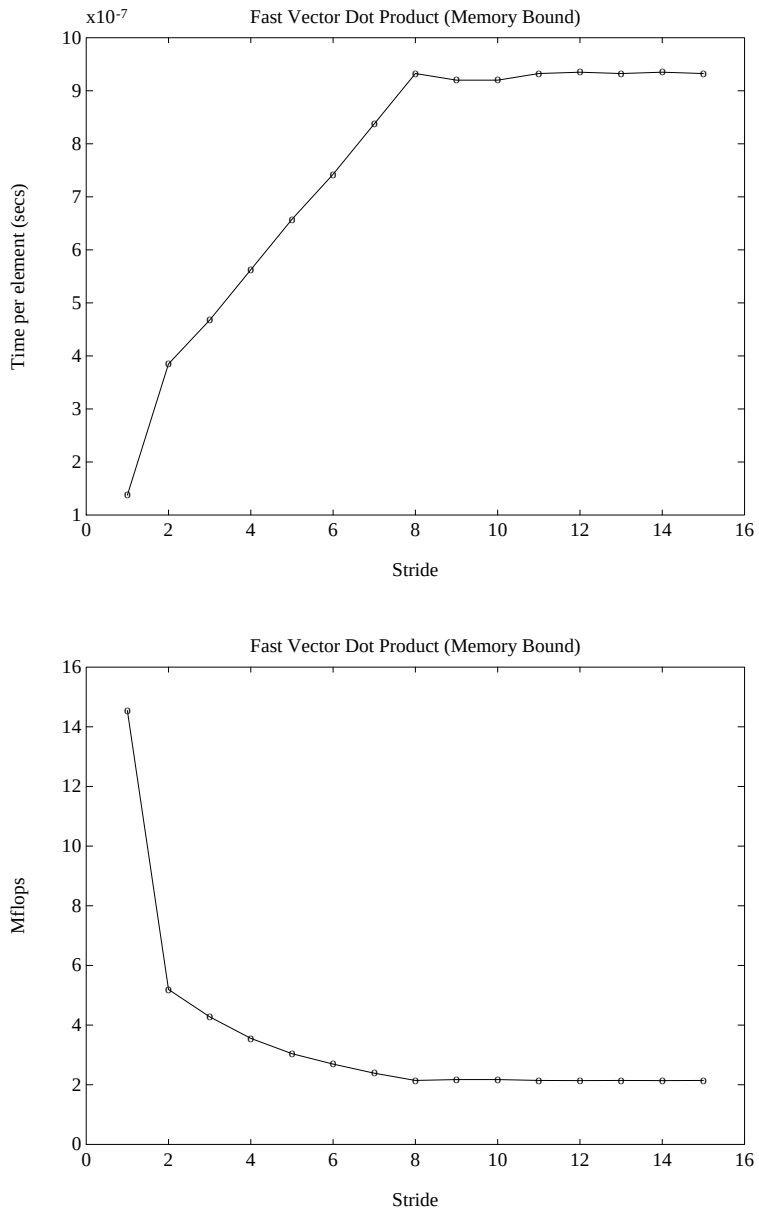


Figure 6.11: Cache hit ratio effects on the performance of the hand coded vector dot product.

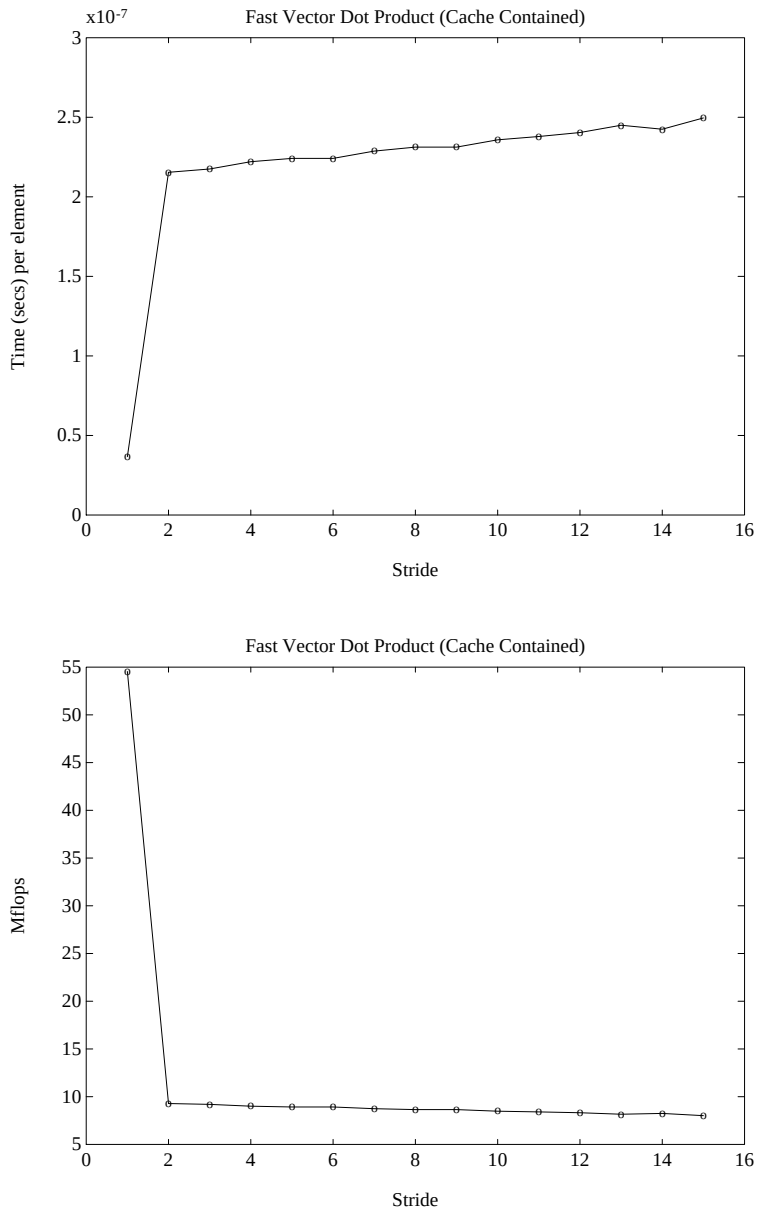


Figure 6.10: Hand coded dot product performance for small cached vectors.

Although 56 Mflops is rather impressive for an inexpensive microprocessor, we see that such high performance is very sensitive to memory access patterns. Figure 6.10 illustrates the sensitivity of this code to the cache hit ratio. Here we have taken two small vectors (N=600) that easily fit in cache and have studied the performance as strides are linearly increased from 1 to 15 elements. The actual dot product was repeated enough times to ensure that the elapsed execution time was of significant granularity, and that inital overhead to bring the vector into cache is negligible.

We see similar performance patterns in FPA page buffer hit ratio when we utilize the same code to compute the dot product of two 2,000,000 element vectors (non-cached). Figure 6.11 shows the performance drop to 14 Mflops. And furthermore, when the stride is changed from 1 to 8, performance drops to around 2 Mflops, as the figure shows. Such a high performance ratio (56:2) indicates the efficient memory utilization is what really dictates the performance of the i860.

The reason for such a wide performance gap is centered around the FPA scheme for retrieving values from main memory to cache. In the normal mode of operation, where caching is enabled, every cache miss copies one complete cache line (32 bytes or 8 single precision floating point values.) The memory traffic of the dot product computation, `sum = A(1)*B(1) + ... + A(N)*B(N)`, is as follows

```
   need A(1)        FPA page miss,
                    5 clocks to get the pair A(1)/A(2),
                    2 extra clocks for every other 64 bit pair:
A(3)/A(4), A(5)/A(6), and  A(7)/A(8).

                       Total: 11 cycles

  need B(1)         FPA page miss, 5 clocks to get B(1) and 2
                    extra clocks for each B(2),...B(8).

                       Total: 11 cycles

  need (A+stride)   if stride is less than 8, this is also in
                    the cache (2 clock cycles)
                    otherwise another FPA page fault (because B
                    was the last quantity in the FPA)

                        Total: 2 or 11 cycles.

  need (B+stride)   if stride is less than 8, this is also in
                    the cache (2 clock cycles)
                    otherwise another FPA page fault (because B
                    was the last quantity in the FPA)

                        Total: 2 or 11 cycles.

        ......
```

Here we see the performance bottleneck to be number of ports to main memory. Due to the small size (4 Kbytes) of the FPA memory page, it is unlikely that both vector reside on the same page. We encounter a page fault on every successive iteration where one needs alternate elements of A() and B(). In the case where both the stride of A() and B() is eight or greater (as when we are traversing multidimensional grids),

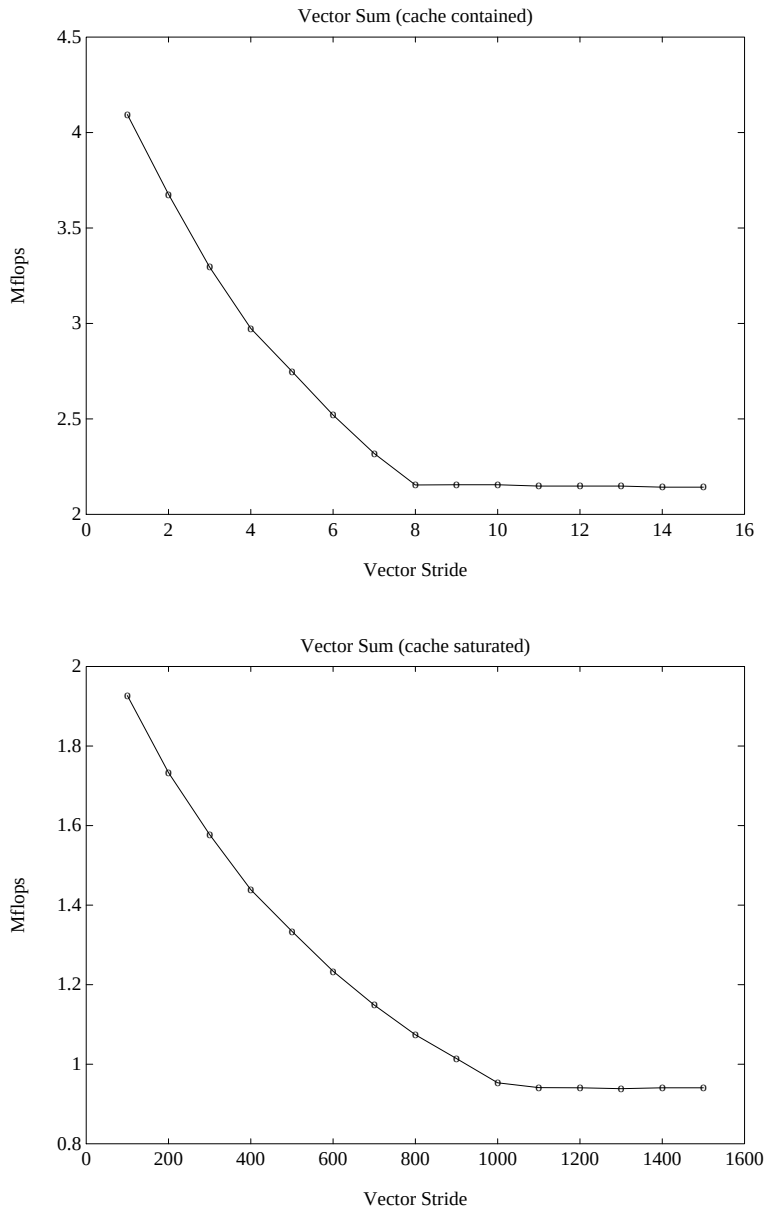


Figure 6.9: Memory performance effects of a vector summation on the i860. (Top) Short vector strides affect the cache hit ratio, (Bottom) long vector strides affects the FPA page hit ratio.

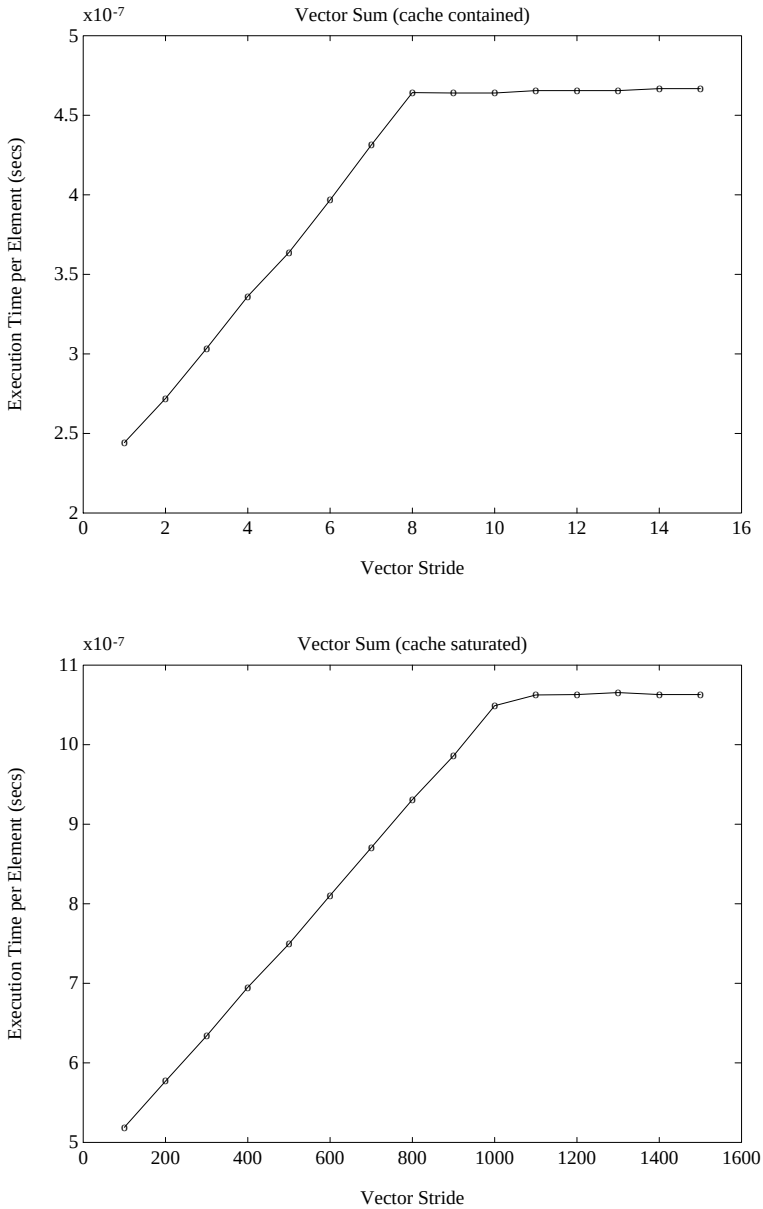


Figure 6.8: Memory performance effects of a vector summation on the i860. Top: short vector strides affect the cache hit ratio; bottom: long vector strides affects the FPA page hit ratio.

```
.L5:

        fld.l   0(r18), f18             //line 1   f18 = *p
        adds    r29,    r18,    r18     //line 2   p += (4*stride)
        adds    r28,    r16,    r16     //line 3   i+=stride
        subs    r16,    r17,    r0      //line 4   i < N?
        fadd.ss f18,    f8,     f8      //line 5   sum += *p
        bc      .L5
```

Figure 6.7: Scalar 860 code for single precision vector sum. Each loop iteration requires 7 clock cycles (4.7 Mflops @ 33 MHz.)

A similar phenomena is seen at the second level of the memory hierarch, the fast page mode RAM access. This Fast Page Access (FPA) mode appears to the user as a second-level DRAM cache. It requires 5 clock cycles to "activate" a page and return the first memory reference. Further memory references that lie within that 4K page boundary require only 2 clock cycles. With large vector strides, then, one sees the familiar characteristics of data caching. As an example, we study a 4,000,000 element vector and sum it using strides of 100,200, 300, ... , 1500. That is, we perform

```
                do 100 i=1,N,stride
                     sum = sum + A(i)
    100         continue
```

for the various stride values. In a similar manner to the caching performance, Figure 6.9 shows the execution time increasing linearly, until the stride is 1,000 elements (4,000 bytes.) At that point each memory reference requires a "far" access, hence the performance reduces to 0.95 Mflops.

6.2.2 Pipelined Performance

In this section we examine how the effective use of instruction and operand pipelining can dramatically improve performance. Using dual instruction and dual operation mode, one can obtain roughly 56 Mflops for a cached vector dot product on a 33 MHz i860. The high-level Fortran source code resembles

```
            subroutine dotpr(A, incA, B, incB, sum, count)
            integer count, incA, incB
            real A(), B(), sum

            i = 1
            j = 1
            sum = 0.0
            do 100 i=1,count
               sum = sum + A(i)*B(j)
               i = i + incA
               j = j + incB
    100     continue
            return
            end
```

The 56 Mflops performance was obtained by implementing a handcoded vector library subroutine.

```
.L9:
   mov     r6,     r25             //Line 1        r25 = A
   adds    -4,     r25,    r25     //Line 2        r25 = A-4
   mov     r4,     r26             //Line 3        r26 = i
   shl     2,      r26,    r26     //Line 4        r26 = 4*i (byte offset)
   adds    r25,    r26,    r26     //Line 5        r26 = (A-4)+ 4*i
   fld.l   0(r26), f27             //Line 6        f27 = "A(i)"
   fadd.ss f27,    f4,     f4      //Line 7        sum = sum + A(i)
   adds    r5,     r4,     r4      //Line 8        i += stride
   bla     r17,    r16,.   L9      //Line 9        repeat loop if i < N
    nop
```

Figure 6.6: Fortran-860 assembly output (no optimization) for single precision vector sum. Each iteration of the loop requires 12 clock cycles (2.75 Mflops @ 33 MHz.)

6.2 Computation Modeling of the Intel i860

To illustrate these memory performance issues on a concrete example, we study the effects on a simple vector summation loop

$$s = \sum_{i=1}^{N} a_i \tag{6.1}$$

using various stride values. For a on 4,000,000 element vector, we study a Fortran generated output, running at 2.59 Mflops, and a hand coded scalar example, running at 4.04 Mflops.

6.2.1 Fortran Performance

We studied the code generated from an early Greenhills/Oasis Fortran compiler. The i860 code generated for main loop of the vector sum operation is shown in figure 6.6. The comments (delimited to end-of-line by //) in that program listing are our own and are not part of the compiler output.

By a careful analysis of the output code, we find that the execution of the main loop requires 12 clock cycles for each iteration. The floating point add instruction, **fadd.ss**, requires 3 clock cycles, but this is overlapped with other RISC core instructions. There is a 1 clock freeze between loading the next vector element (line 6) and accumulating into the total sum (line 7.) This is because there is an extra clock delay when accessing the data cache. This could have been eliminated by swapping lines 7 and 8, reducing the total loop execution to 11 cycles.

Lines 1 and 2 are not necessary –they merely initialize beginning of the vector of A() to be A(-1). This is done because in Fortran the first vector element is A(1), not A(0) as in C. By readjusting the beginning of array in this manner, then the address of A(i) is computed as `(A-1) + 4*i`, since each single precision element occupies 4 bytes. However, lines 1 and 2, are necessary to the loop and we could save 2 clock cycles by moving them outside the loop.

Figure 6.7 denotes the code used for experiments with a small, cached vector of size 400. The expected theoretical performance of 4.72 Mflops (33 MHz/7 clocks) was obtained. Figures 6.8 and 6.9 denote the performance degradation, measured in execution time per vector element and Mflops, respectively, incurred when using vector strides from 1 to 15. Figure 6.8 illustrates how the execution time per vector element increases linearly, until a stride of 8. At this point, only one element per cache line is utilized and further increase in the vector stride merely reuses another cache line. Fig 6.9(c) denotes the Mflop degradation of the same experiment.

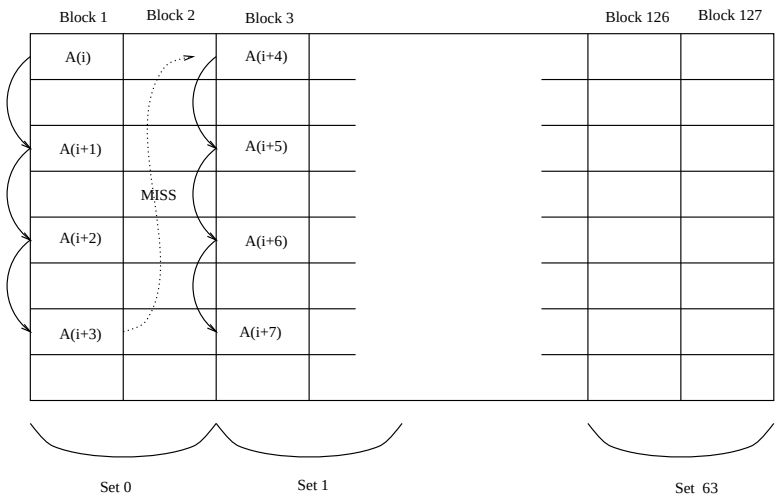


Figure 6.4: Accesses to 8 consecutive elements of a vector with stride two. For each block brought from memory, there will be 4 cache hits.

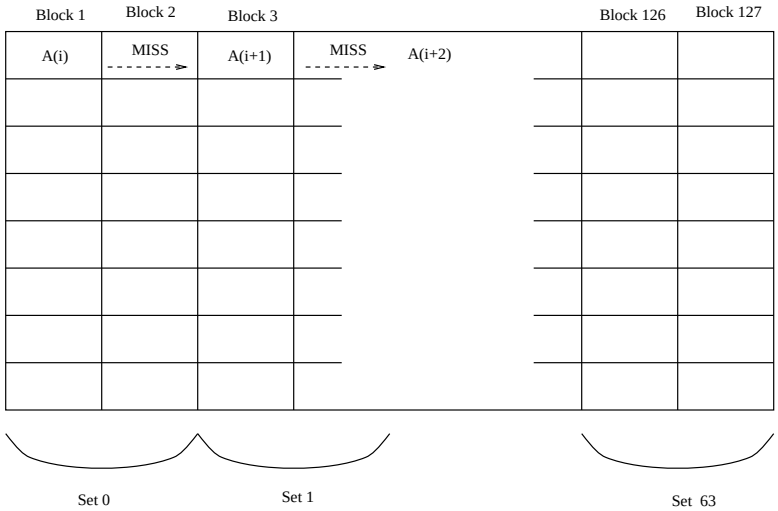


Figure 6.5: Access of 8 consecutive elements of a vector with stride 8. Only one cache hit per block is obtained.

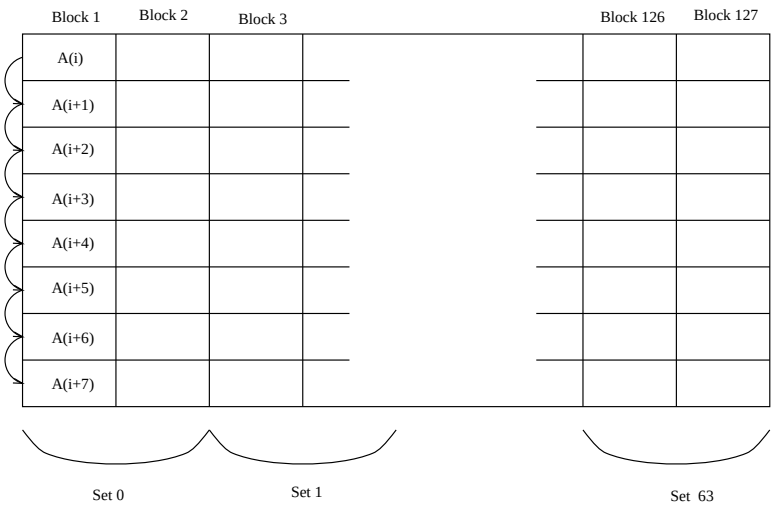


Figure 6.3: Access to 8 consecutive elements of a vector with stride one. If the first element is at the beginning of a block, 8 consecutive cache hits occur.

A near access takes 2 clock cycles (transfer from RAM memory to cache), and a far access takes 5 clocks. Clearly the Fast Page Access technique exploits special locality on the data.

6.1.2 Caching and Memory Paging Effects

The system performance of the i860 is heavily influenced by data mapping and the effective utilization of the memory hierarchy: excessive transfers between cache and external memory, together with continuous access across page boundaries, decrease throughput dramatically. Here we illustrate how the operand availability influences the performance. We consider the interaction among the three levels of memory hierarchy in the i860 architecture: register files, on-chip cache, fast access page , and main memory.

In the normal caching of data, new memory reference cause a filling of the 32 byte line in the data cache. If contiguous memory locations are needed then they can be retrieved directly from the cache. That is, for unit strides, the other seven contiguous vector elements will also be in that cache line. However, if the stride is larger, then those data elements brought into the cache line will not be used. This is illustrated in figures (6.3, 6.4, 6.5.)

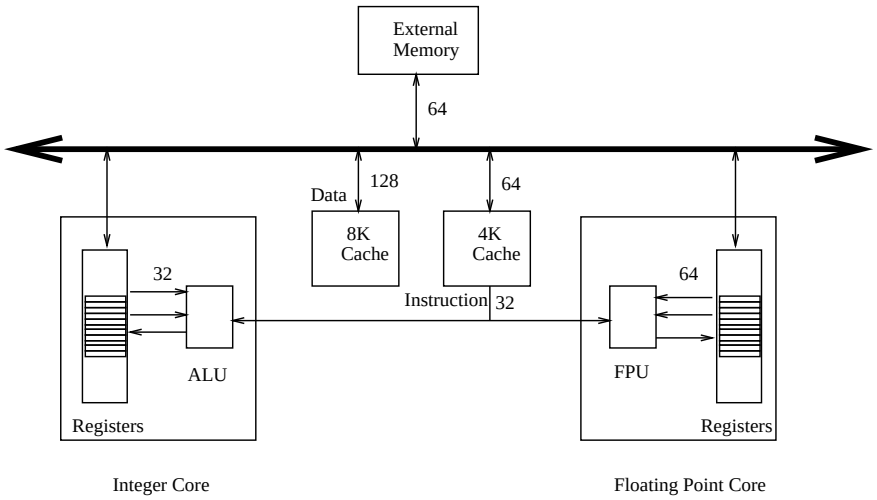


Figure 6.1: An architectural overview of the Intel i860 RISC Microprocessor.

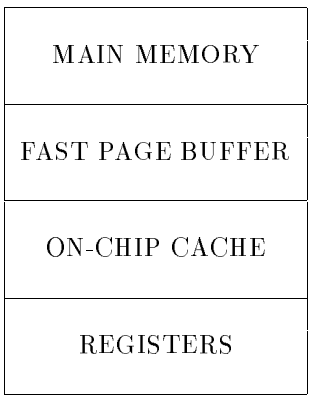


Figure 6.2: Hierarchical memory levels of the Intel i860.

6.1 Processor Architecture: Intel i860

Each node of the iPSC/860 consists of an Intel i860 Reduced Instruction Set Computer (RISC) microprocessor capable of producing 80 MFLOPS (million floating point operations per second) and 40 MIPS (million instructions per seconds.) The Intel i860 microprocessor is mass-produced and easily available; it has been used in workstations, add-in boards for PC's, dedicated graphics cards, and realtime systems.

The basic architecture of the i860 is shown in figure 6.1. By utilizing effective pipelining techniques and exploiting microarchitecture parallelism, two floating point instructions and one integer instruction can be completed every clock cycle. Characteristic of RISC designs, the i860 has a load/store architecture: instructions operate only on internal register operands and data must be explicitly transferred to and from main memory.

Besides the multistage instruction pipelining discussed below, integer and floating point operations can be performed simultaneously. For example, address calculation for vector operands can be performed simultaneously with the floating point operation on the vector elements. Each i860 instruction consists of four basic stages:

- **FETCH:** selects the instruction from the 4Kbyte instruction cache and moves into an internal storage buffer.
- **DECODE:** retrieves instruction operands from internal registers and decodes instruction.
- **EXECUTE:** performs ALU (or FPU) operations and address calculation for memory references, if necessary.
- **WRITE:** Results of operation (if any) are written to the register file. The data cache is accessed here for memory operations.

Although these stages can be effectively overlapped, there are conditions which may stall the pipeline: resource conflicts (two operations require the same resource), data dependencies (one instruction requires data from previous instruction), and control dependcies (e.g. branch instructions that change the execution sequence.) Of these three cases, resource conflicts with the memory system accounts for the most significant problem (see section 6.2.2).

Although the peak speed of the i860 is quoted at 80 Mflops, it is nearly impossible to such figure in practice. Such speeds assume only register-to-register operations. Limited bandwidth to main memory is the primary obstacle in sustaining peak speeds. Furthermore, the complex microarchitecture of the i860 poses a challenge to optimizing compilers. Currently, we are seeing only 4-6 (64-bit) Mflops out of C and Fortran codes (see sections 6.2.1 and 6.6.1.) This translates into 5-8% efficiency of the RISC hardware and illustrates that optimizing compilers still have far to go in generating effective code.

6.1.1 Memory System Architecture

In this section, we discuss how the data cache and the FPA Buffer miss ratios most important parameters for memory access optimization. The basic memory architecture of the i860 can be visualized as a four-level hierarchy in figure 6.2. Coupled to the integer core CPU and floating point unit, the Intel i860 has an on-chip data cache of 8 Kbytes and an instruction cache of 4 Kbytes. The processor is connected to the data cache through a 128 bit bus, which allows the simultaneous access to four single precision of two double precision floating point numbers. The cache is two way set associative, with 128 blocks (64 sets) of 32 bytes. Each block fits eight single precision or four double precision floating point numbers.

To reduce the memory latency a technique known as Fast Page Access (FPA) is used. When a word is accessed that corresponds to a page different from the one of the previous access, the entire page is activated and its row becomes a virtual buffer: the FPA Active Page. If successive accesses are to the same page, this buffer will be used for faster response since the "row selection" is no longer necessary. Accesses to words in the FPA buffer are called near accesses, accesses to data not in the buffered page are called far accesses.

Table 6.1: Message passing primitives on the iPSC/860.

```
Blocking:

     csend(type, buf, len, node, pid)
     crecv(type buf, len)

Non-blocking:

     msgid = isend(type, buf, len, node, pid)
     msgid = irecv(type, buf, len)
     status = msgdone(msgid)
         msgwait(msgid)
```

Chapter 6

Intel iPSC/860 Hypercube

The Intel iPSC/860 is a distributed memory MIMD parallel machine. It is part of the Touchstone family of computers that will target parallel architectures towards 100 Gflops by 1995 [56]. The iPSC/860 is referred to as the Gamma prototype; it consists of up 128 nodes connected in a hypercube topology with worm-hole routing communication.

Each node of the iPSC/860 consists of an Intel 40 MHz i860 RISC microprocessor, capable of producing 40 MIPS and 80 Mflops. (See section 6.1 for details of the i860 microprocessor architecture.) Nodes can be fitted with 8 to 64 Mbytes of memory and a routing module (DCM) capable of generating up to 2.8 Mbytes/sec transfer rate between processors. The system is controlled by a front end host computer: System Resource Manager (SRM). Compiling and program development can be carried out on Sun workstations to relieve the workload of the SRM. The SRM is a PC/AT compatible microcomputer running UNIX System V. The node processors run a limitied operating system, NX, which supports message passing primitives and I/O to the host.

The peak unidirectional bandwidth between processors is 2.8 MB/sec. Communication primitives are based on the asynchronous send/receive message passing model, and are available in blocking, and non-blocking modes, (table 6.1.) They are callable from Fortan or C applications. The blocking versions wait until the message buffers are freed before continuing. The non-blocking versions continue executing the next statement immediately. It is the programmer's responsibility to check that the message operation has completed by using, for example, the `msgwait()` primitive. Further details of the message passing primitives can be found in section 6.3 and [51].

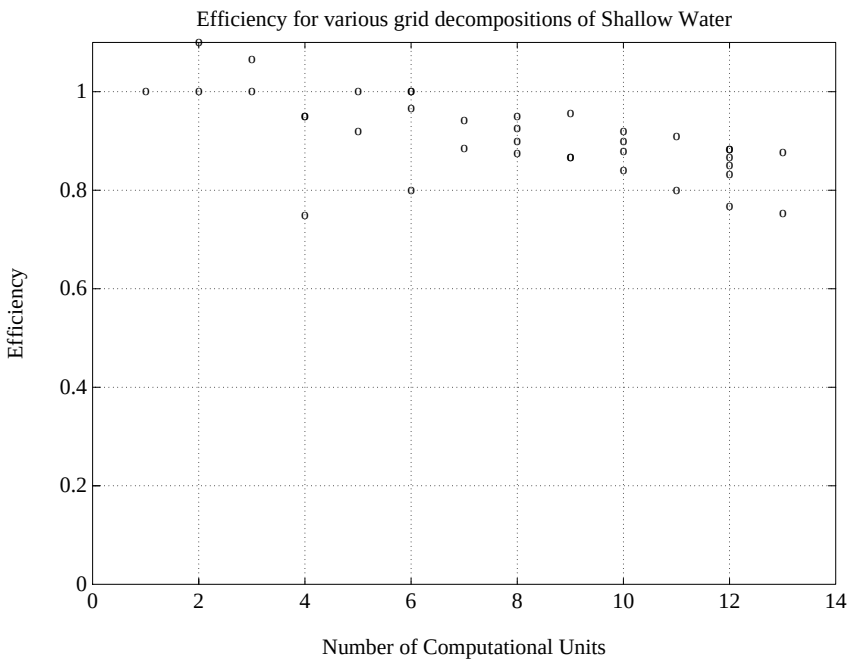


Figure 5.11: Multiple efficiency values represent different grid decomposition methods for a given node set.

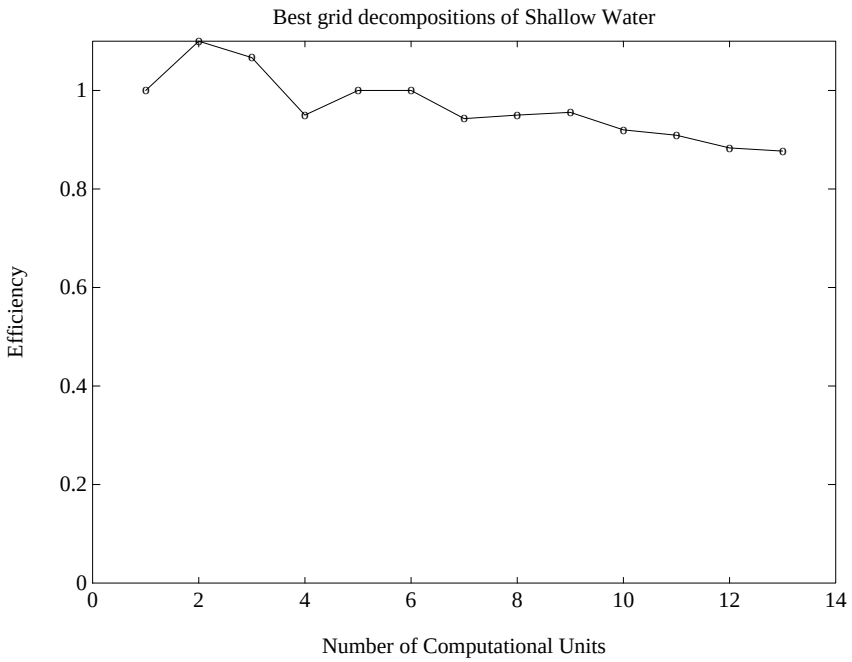


Figure 5.9: Efficiency values for best grid decomposition method (full column slices) of the Shallow Water equations.

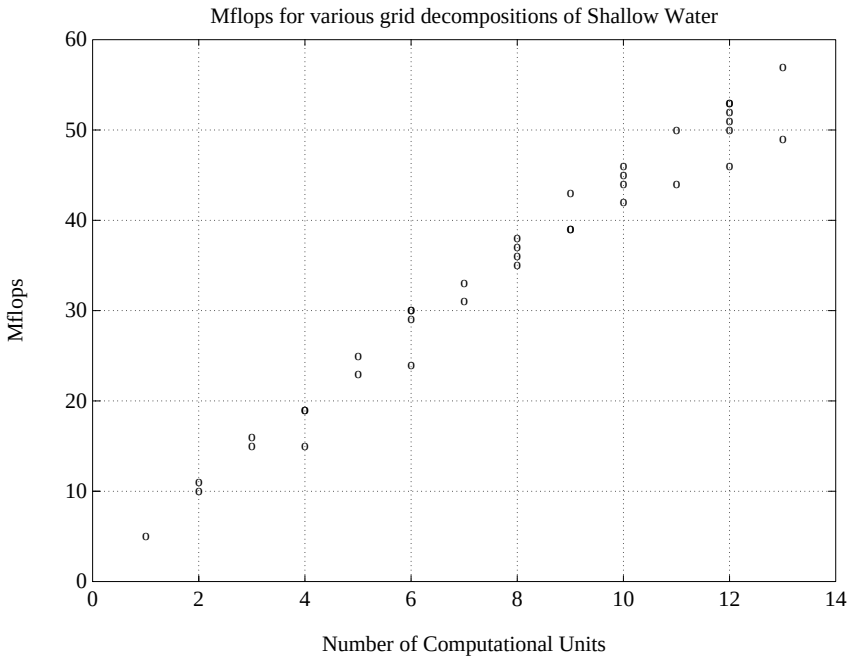


Figure 5.10: Mflops for various data distributions of Shallow.

5.4 Implementation and Performance of Shallow Water Application

The Fortran code implementing the Shallow Water equations involves a 2D rectangular grid with variables: u(i,j), v(i,j), p(i,j), z(i,j), psi(i,j), h(i,j). A typical code sequence for the CALC2 subroutine (section 2.6) on the ES-1 is given by

```
c$es parallel do
    do 200 j=1,n
    do 200 i=1,m
       unew(i+1,j) = uold(i+1,j)+
          tdts8*(z(i+1,j+1)+z(i+1,j))*(cv(i+1,j+1)+cv(i,j+1)+cv(i,j) +
          cv(i+1,j))-tdtsdx*(h(i+1,j)-h(i,j))
       vnew(i,j+1) = vold(i,j+1)-tdts8*(z(i+1,j+1)+z(i,j+1))
          *(cu(i+1,j+1)+cu(i,j+1)+cu(i,j)+cu(i+1,j)) -
          tdtsdy*(h(i,j+1)-h(i,j))
       pnew(i,j) = pold(i,j)-tdtsdx*(cu(i+1,j)-cu(i,j)) -
          tdtsdy*(cv(i,j+1)-cv(i,j))
200 continue
c$es end do
```

Each such loop is followed by code to implement the periodic boundary conditions. Note that there are loops for both the horizontal and vertical boundaries, and in addition some corner values are copied as single items.

For the ES-1 implementation, we simply parallelize the code by the use of directives to identify parallel loops as described in section 5.2.2. That is the only code modification required for the main loops. Similarly, the boundary condition loops can be parallelized; however, its effectiveness is limited by the task granularity.

The ES-1 performed the explicit model at 57.3 Mflops with 13 nodes on a 281x281 grid, and ran the same model at 5 Mflops on a single node. Thus parallel efficiency is about 88%. As with the relaxation examples, we studied many potential ways to decompose the grid among the nodes. Figure 5.10 shows the Mflops performance of this model over varying number of nodes. Each abscissa coordinate has several performance values, corresponding to different grid compositions as described in section 5.3.5. The most efficient proved to be using full column slices.

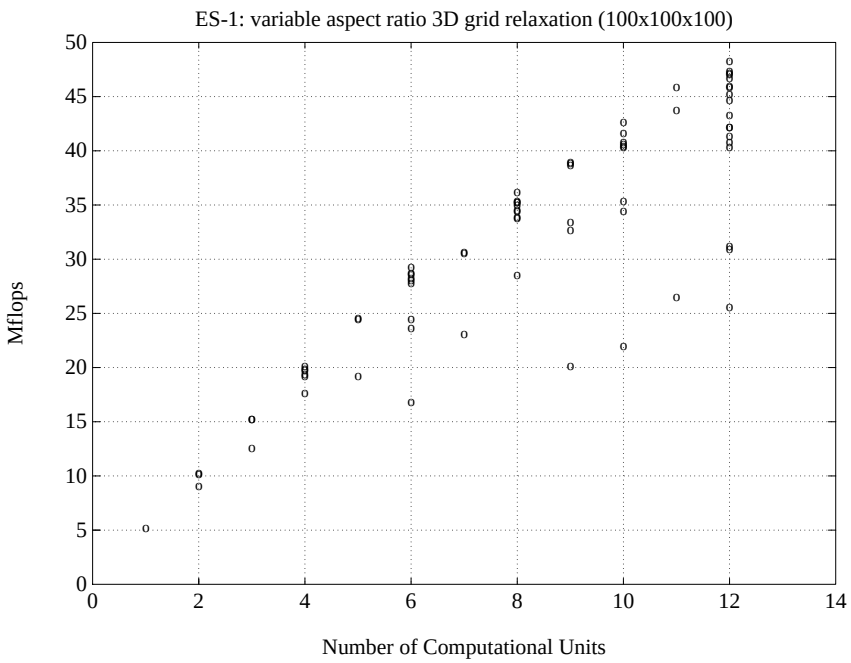


Figure 5.8: Performance variation in 3D relaxation schemes where grids have been decomposed by blocks.

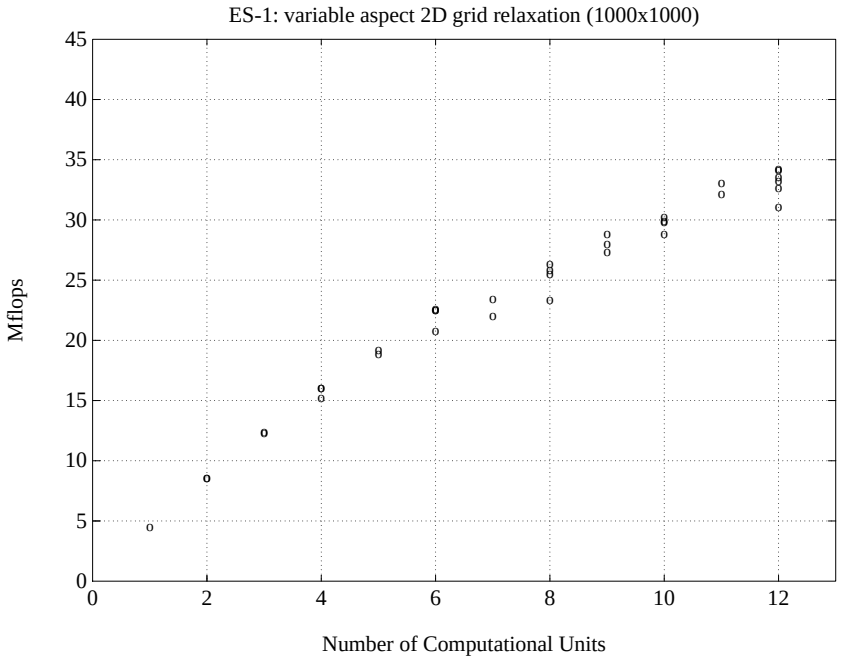


Figure 5.7: Performance variation in 2D relaxation schemes where grids have been decomposed by blocks.

The procedure `decompose(a,b,t,istart,iend)` decomposes an interval [a,b] into t near-equal length subintervals as above, and which initializes arrays istart(t), iend(t) with the start and end indices of each subinterval. The procedure `gridproc(p, 2, qx, qy)` returns the x and y processor indices that reference the grid block assigned to node `p`.

The above procedure provides many different parallelizations of a given problem, one for each possible factorization of the number of tasks T. At one extreme are decompositions by rows (case T1=1), and at the other extreme are decompositions by columns (T2=1), with intermediate values representing decompositions by rectangles. Figures 5.7 and 5.8 depict scatter plots such decomposed grid domains.

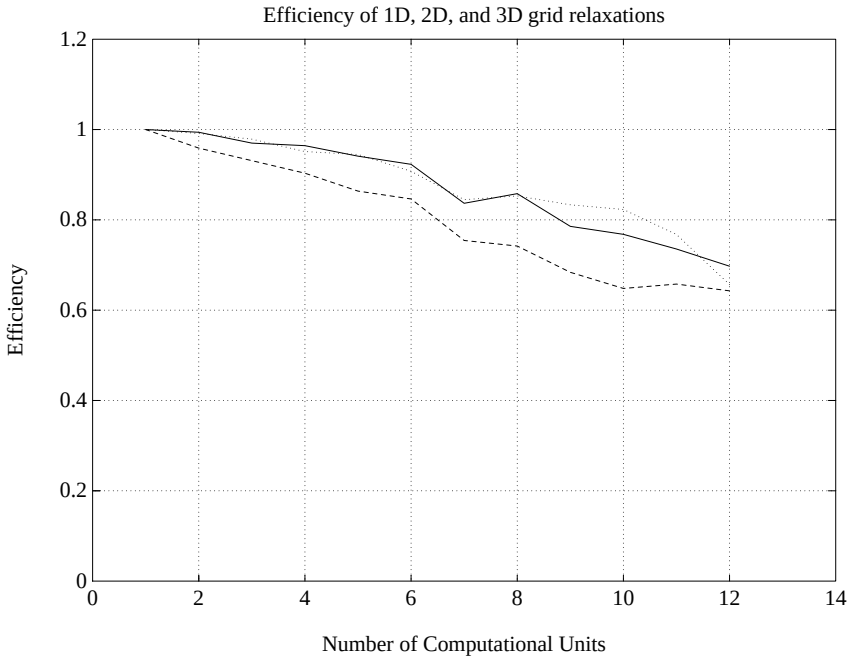


Figure 5.5: Dashed line denotes 3D (100x100x100) single precision grid relaxation; dotted line denotes 2D (1000x1000), and solid line denotes 1D (10**6) grid relaxation

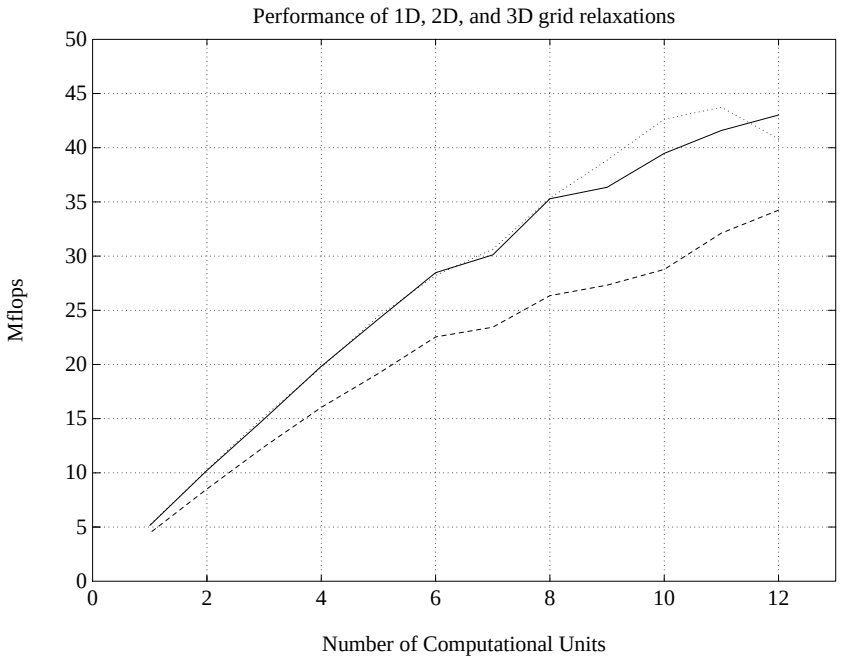


Figure 5.6: Dashed line denotes 3D (100x100x100) single precision grid relaxation; dotted line denotes 2D (1000x1000), and solid line denotes 1D (10**6) grid relaxation

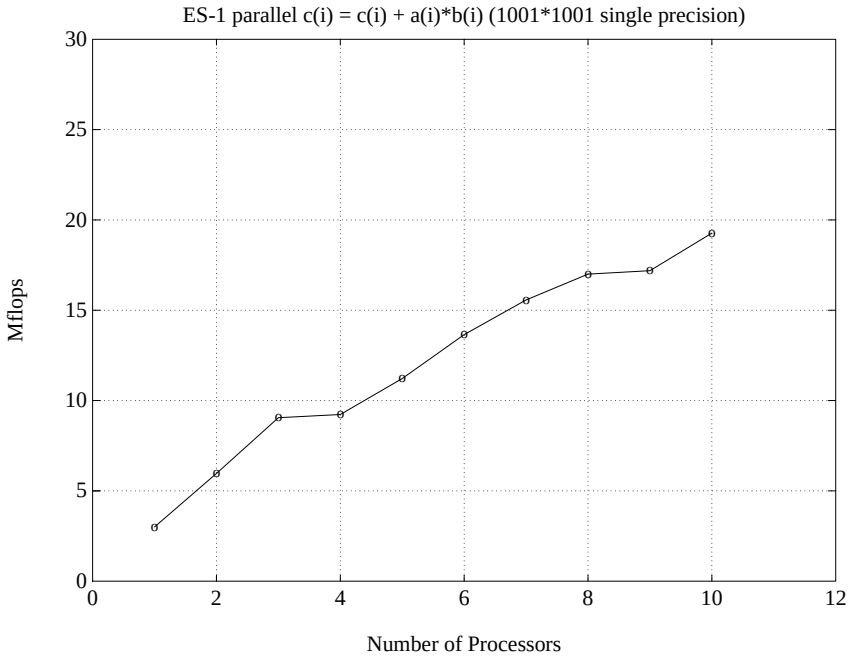


Figure 5.4: Parallel vector/vector triad performance on the ES-1.

We also notice, that while the performance curve for the 2D relaxation is lower than 1D, it shares the essential features of the 1D curve: near-linear up to 6 CU's, a slight dip at 7 CU's, and so on. Figure 5.6 illustrates the efficiency curves for the 1D, 2D, and 3D grid relaxations.

5.3.5 Variable Aspect Ratio Grid decompositions

Several data partitioning strategies can be explored for n-dimensional grid relaxation by decomposing the grid G into T contiguous rectangular subgrids and each of T tasks will be assigned to process a different subgrid.

For each unique factorization of T = T1 * T2, we divide the index interval [2,n1-1] into T1 essentially equal pieces and similarly we divide [2,n2-1] into T2 pieces. The tensor product of the interval decompositions defines the 2D subgrid decomposition.

The ES-1 Fortran code to perform these computations takes the form

```
        decompose(2,n1-1,t1,istart1,iend1)
        decompose(2,n2-1,t2,istart2,iend2)
c$es parallel do
c$es local qx, qy, it, i, j
        pardo 10 p=1,numtasks
        call gridproc(p, 2, qx, qy)
        do 15 j=isty(qy), ieny(qy)
            do 15 i=istx(qx), ienx(qx)
15              b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1) +
                              a(j-1,i) + a(j+1,i)
10      continue
c$es end do
```

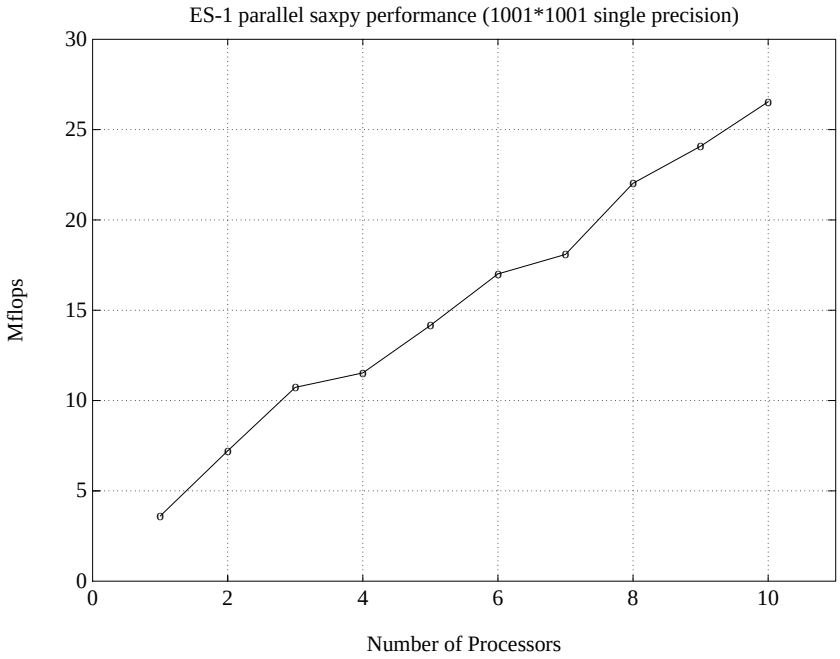


Figure 5.3: Parallel scalar/vector triad performance on the ES-1.

3. near-linear performance relationship (although smaller slope) is again established between 4 and 10 CUs.

5.3.4 Parallel Relaxation Performance

Experimental results for 1D, 2D, and 3D grid relaxations are shown in figure 5.5. The three results have been superimposed to highlight their common characteristics. In this case, the memory size of each grid is the same, it is just distributed evenly among its dimensions — the size of the 1D grid is 10^6, the size of the 2D is (1000x1000), and the 3D is (100x100x100.) For the 1D grid we have decomposed it into T=P tasks, 2d grids are decomposed into Px1, and 3D grids into Px1x1 task decompositions. Section 5.3.5 below presents some relaxation results using variable aspect-ratio decomposition of grids.

We note that near-linear relationship is maintained up to 6 CUs. The performance difference 1D and 3D relaxations is almost indistinguishable up to 6 CUs, where the 3D grid relaxations perform better than the 1D relaxations. The performance of 2D relaxations lag significantly behind those of 3D and 1D. This characteristic is not entirely what one would expect. For example, we would expect the effectiveness of data caching to be reduced for the 3D relaxation, since only three of the seven memory references in

$$\begin{aligned} b(k,j,i) &= 6.0 * a(k,j,i) + a(k,j,i-1) + a(k,j,i+1) + \\ &\quad a(k,j-1,i) + a(k,j+1,i) + a(k-1,j,i) + a(k+1,j,i) \end{aligned}$$

are consecutive and hence could fit inside a data cache page.

We would also expect the 3D relaxation to have a higher cost associated with the computation of the memory address from the array indices. This could be an indication that the integer calculations involved in indexing arrays has been successfully overlapped with the floating-point computation, by using the integer and floating point pipes. This, however, would not explain why the 2D performance is significantly lower; or why at some stages (between 8 and 11 CUs) 3D performance is actually *better* than the 1D performance.

Table 5.1: Single Node Performance of the ES-1

Operation	Mflops
c = a + c	7.021
c = a * c	7.021
c = a * c (unrolled)	2.922
c = c + a*c	1.925
c=c+a*c; a=a+c*b	1.921

```
c = a * c
c = a * c
```

The second statement, for example, cannot begin executing until the first completes, since the resulting value of the first statement, c, is used is referenced in the second, and so on. It is still not clear why this effect is not seen in the unrolled version and this will one issue to be investigated — one conjecture is that the predictable computation involved in the loop overhead somehow allows the compiler to make better assumptions about pipelining effects.

5.3.3 Vector Triad Performance

Although vector operations do not involve explicit interprocessor communication, data migration is still noticeable, as there is interaction between the various memory hierarchies. As in, for example, the memory space of large vectors gets copied into the caches of individual processors. Here we study the performance of vector triad operations which occur as the low-level computation intensive core of applications. The two operations of study are (1) scalar/vector triads,

$$y_i = a * x_i + y_i,$$

and vector/vector triads,

$$y_i = b_i * x_i + y_i$$

We first study vector lengths which are non-powers of two to minimize the effects of memory bank conflicts in each cluster. Since each ES-1 cluster 16 CUs share 8 memory banks, there is the potential for bank conflicts for data structures which whose number of elements are a multiple of eight.

Figure 5.3 shows the performance for a scalar/vector triad (saxpy) using for one to ten CUs. These tests were performed on single precision vectors of length 1001*1001. We see two dips in the graph, one occurring at four CUs, and the other occurring when using seven CUs. Notice that near-perfect speedup is achieved using up to 3 CUs. The fourth CU adds relatively little computational power. One can see a linear relationship between Mflops and CUs again from four CUs to 10 CUs, except for a small dip when using seven CUs. We would expect to see some performance degradation when using more than 8 CUs, since then we have potential memory bank contention (16 CUs vs. 8 memory banks), although this is somewhat offset by the use of data per CU data caches.

The performance of vector/vector triads, figure 5.4, lags behind the scalar/vector triads. This is expected, since the scalar multiplier of the scalar/vector triads can be kept in a fast access register. We notice that figures 5.3 and 5.4 share common characteristics:

1. near-perfect linear speedup is observed when using one to three CUs.
2. the computational power added by the fourth CU is relatively small — flattest slope of both graphs occur between 3 and 4 CUs.

5.3 Performance Analysis of the ES-1

Although the ES-1 was specifically designed as a parallel machine, its individual CUs provide significant computational power (2-7 Mflops) to be effective at running small programs on a single CU as a conventional sequential uniprocessor. For the parallel program developer, this provides an ideal environment where existing sequential codes can be executed with little or no modification, establishing baseline results where program correctness can be tested as successive transformations and optimizations are applied in fine-tuning the code for maximum performance.

5.3.1 Relation between Tasks and Processors

A fundamental difference between this machine and others included in this study is that parallelism is specified by the number of tasks, rather than the number of processors, to use. Thus, one has no explicit control over the number of physical processors used; nor does one have any control over the mapping of data to individual nodes. This leads to interesting challenges for the performance modeler to adequately describe the machine's behavior.

Since a CU is not a user-allocatable resource, parallelism is specified by requesting the number of concurrent *threads* to use. When a parallel section of code (such as a parallel do) is encountered, parallel threads are created (see section 5.2.1) and placed on the global (per cluster) run queue. Then, any idle CU can be assigned that thread and continue its execution. Once the CU has taken the thread then when its time slice is over, it gets placed on a local (per CU) queue. Only when a thread blocking for I/O (or some other system service) does it get placed back on the global run queue. Consequently, next time that thread becomes active, it may not necessarily be running on the same CU. This creates some automatic load balancing across the machine, but complicates any analysis of its performance.

Among the consequences this has for benchmarking, is that the number of *threads* do not necessarily correspond to the number of CU elements used. For example, in a system with many users running parallel program, there will be many more threads created then there are physical nodes. In this case, threads are just sequentially time sliced. With a heavily used system, the migration of users' tasks to processors is completely unpredicatable. For this reason, all of our experiments were conducted on a dedicated system with no other users competing for resources.

At the time of our experiments, a software bug concerning the use of locks occasionally caused deadlock when using 15 or more processors for a single job. This problem was studied by E&S engineers and researchers at the Brigham Young University (BYU)[33], however the software patch was never completed. To avoid performance pertubations dues to this software bug, we restricted our experiments to 13 processors or less.

5.3.2 Single CU Scalar Performance

Single-node performance is measured by setting the environment variable N_THREADS to have the value of one. This allows us to automate the benchmarking process by implementing a simple shell script that increments N_THREADS to create a parametrized study of a specific problem over a given processor domain.

Performance numbers for simple scalar operations are given in table 5.1, where many loop iterations were performed to ensure that loop overhead is minimized and that the discrete effects of the clock resolution is negligible.

Each algorithm was implemented as a Fortran program and compiled with the -O -a option to f77, yielding the highest optimization available. Results shown here are for single precision data.

The unrolled version of `c = a*c` was obtained by unrolling 4 times. Notice that the unrolled version gives less than half of the performance of the unrolled $c = a * c$ loop. One possible explanation could center around the effective pipelining in the code sequence

```
c = a * c
c = a * c
c = a * c
```

several hundred) there will many more loop iterations than there are physical CUs and the problem can be effectively parallelized with a single PARALLEL DO construct wrapped around the outer-level loop. Some reorganization of the original code may be necessary, however, when attempting to parallelize sections of code which resemble

```
          do 10 i=1,3
             do 10 j=1,4
                 call bigroutine(i,j)
10        continue
```

Assuming that separate calls to **bigroutine()** do not have any interdependencies and can executed concurrently, the above code exhibits 12-way parallelism. Since parallel constructs cannot be nested on the ES-1, some potential parallelism cannot be utilized. Full parallelism can restored, however, by a simple code transformation which essentially 'fuses' muti-nested loops into a single parallel loop, such as

```
c$es parallel do
          do 10 t=1,12
            call bigroutine( (t-1)/3+1, imod(t-1,4) + 1 )
10        continue
c$es end do
```

Critical sections which require to synchronize the activities of independent threads using LOCK and UNLOCK directives. LOCK acquires an interlock before entering a critical region, while UNLOCK frees an interlock after leaving the critical region.

In addition to parallel do loops, general multithread series/parallel control structures can be specified by the PARALLEL SECTIONS directive. Each section of a PARALLEL SECTIONS construct executes concurrently on different threads. For example, the following code segment demonstrated the use of these directives.

```
c$es  parallel sections
c$es  section
          call routine1
c$es  section
          call routine2
c$es  section
          do 10 i=1,n
            a(i)=0
10        continue
c$es  end parallel sections
```

In this example, three concurrent threads are spawned ; code following the PARALLEL SECTION construct is not executed until the last thread finishes execution of its work assignment.

5.2.2 Programming Environment

The ES-1 will run standard Fortran-77 programs (with VAX extensions) without modification on a single computational unit (CU.) This allows one to get baseline programs up and running with minimal effort. Once this is done, then successive optimization/parallelization techniques can be performed to further increase execution performance. Such approaches are ideal, since one always has the correct program results to very that each successive program transformation does not destroy correctness.

5.2.3 ES-1 Parallel Fortran Compiler Directives

A straightforward method to parallelize existing Fortran scientific codes for the ES-1 involves inserting specific compiler directives to the potentially parallel sections of the code. The basis for these directives were based on the Parallel Computing Forum document *PCF FORTRAN: Language Definition* [73] [74]. These directives control the management of multiple thread execution, critical sections, synchronization primitives, locks, and activation/deactivation of threads. This discussion will cover the most relevant constructs; readers are referred to the *ES-1 Programmer's Reference Manual* [26] for complete details.

The number of threads to be used for a given program is specified by the environment variable N_THREADS. The number of available threads is determined at runtime and cannot be changed for that run of the program. However, there are directives for inactivating threads so that they do not consume physical resources (see ACTIVATE directive below.) Concurrent threads (parallel execution) is specified by the **-m** f77 compiler flag. In this way, one can run "parallelized" code on a single computational unit without any code modification. This greatly simplifies the debugging phase of parallel program development. All ES-1 Fortran compiler directives begin with an **C$ES** in the first column and have the name of the directive immediately following it.

The ES-1 Fortran directives enable the parallel execution of successive iterations of a DO loop on multiple threads. Parallel Fortran DO loops are implemented with the `PARALLEL DO` directive. All iterations of a `PARALLEL DO` loop execute concurrently on different threads. In the absence of any other synchronization mechanisms, each iteration of the loop must independent from all others. `PARALLEL DO` loops are terminated by the `END DO` compiler directive. Temporary loop variables not used outside a parallel do loop can be declared `LOCAL`, thus keeping its reference local to the executing thread. For example, the code segment for a parallel matrix multiply could appear as

```
C$ES PARALLEL DO
C$ES LOCAL t, j, k
       do 10 j=1,N
          do 20 i=1,N
             t = 0.0
             do 30 k=1, N
                t = t + a(i,k) * c(k,j)
30           continue
             c(i,j) = t
20        continue
10     continue
C$ES END DO
```

Here the `LOCAL` directive specifies that each thread has a separate copy of the variables t, j, and k. Without the LOCAL directive, load and stores of these specific variables would reference the same memory address and produce incorrect results. (The loop indices of the outermost parallel loop are automatically declared LOCAL.)

It is important to note that although parallel constructs can be nested, the compiler only recognizes the outermost level of parallelism. For large data structures and small number of CU's, this does not pose much of a problem. In the matrix multiply code above, for example, with large values of **N** (say,

intercluster memory traffic. the Zone Cache (Z-buffer) is under software control and may be user-optimized at the assembly language level, or used by the operating system to keep task information and reduce the overhead for context switches (see section 5.2.1.) Memory performance can be further increased by the use of 32 64-bit floating point registers and 32 32-bit integer registers. This large number of register reduces memory accesses by holding temporary results during computation.

For floating point computation, multiply and adder floating point pipelines are implemented in ES-1, using the Weitek WTL 2264/WTL 2265-80 chipset [33]. Each floating point pipeline can produce a new result every 2 clock cycles using a 40ns cycle time. With both pipes working simultaneously, each processor is capable of 25 Mflops at peak speed. The interface to the floating point pipelines is 32 bits wide and consists to 2 input buses and a single output bus which connect the adder and multiplier.

5.1.3 Memory Hierarchy

Since the ES-1 also supports a virtual memory address space of 4 Gbytes (of which up to 2 Gbytes may be physical memory), the complete memory hierarchy may be seen as a five-level system:

1. (288 bytes) Floating-Point & Integer Registers
2. (128 Kbytes) Instruction, Data, and Zone Caches
3. (256 Mbytes) local cluster memory (first-level crossbar)
4. (2 Gbytes) intercluster memory (second-level crossbar)
5. (4 Gbytes) virtual memory

Although the ES-1 employs a uniform address space, locality of reference is still critical. For example, each cluster contains 16 CU's but only 8 memory banks, giving rise to memory bank conflicts. The extensive use of the data and instruction caches on each CU helps alleviate this contention.

5.2 Programming Environment

5.2.1 ES1/Mach Operating System

The parallel programming model of the ES-1 is based on the Mach Operating System – an extension of the Unix operating system to support parallel programming. The Mach operating system uses lightweight processes, *threads*, as the basic schedulable unit. There may be several concurrent threads of execution within a program.

Unlike the standard Unix process model, Mach threads have a much smaller overhead for creation. For example, the parent's code text is not copied but shared among its children. This cuts down on the amount of data copying that occurs during task creation, and hence reduces associated start-up costs.

For systems programming, the basic mechanism for specifying parallel tasks is quite similar to the Unix fork/join model. Several routines are available in the C/Mach library for management of multiple threads, mutual exclusion, locking shared variables, and broadcasting condition signals.

The operating system maintains two types of tables to determine when a thread gets scheduled for executions: each CU has its own local run queue in addition to a global run queue. Normal thread preemption by the scheduler places that thread on the local run queue, keeping the context (state information) of that thread local to the CU it was running on. In some cases, register file contents can be saved by the operating system onto the Zone Cache for fast retrieval when the preempted thread gets rescheduled. This scheme helps maintain locality of reference to within each CU.

Threads that are blocking for I/O or other events get placed on the global run queue. Threads on the global run queue can be scheduled on the first available CU. This has the effect of balancing the work load across all the CUs in the system.

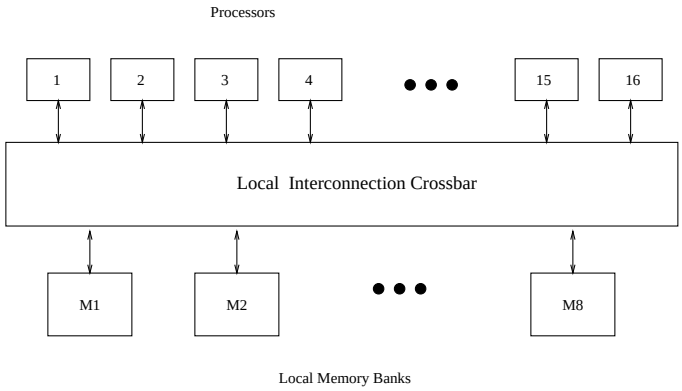


Figure 5.1: ES-1 Computational Unit Architectural Overview

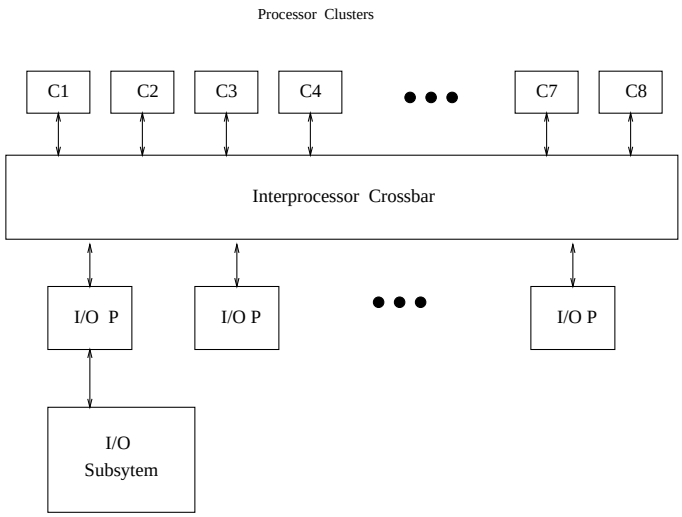


Figure 5.2: ES-1 Computational and I/O Processor Overview

Chapter 5

Evans & Sutherland ES-1: A Shared Memory MIMD System

The Evans & Sutherland ES-1 is a shared memory MIMD parallel supercomputer based on a bilevel hierarchal crossbar interconnection scheme. The ES-1 design allows configurations up to 128 processors, or computational units (CU), resulting in a system with 2 Gbytes of memory and an aggregate peak speed of 1600 Mflops and 1600 MIPS. Each CU is a Reduced-Instruction-Set (RISC) microprocessor with its own cache, RISC core, and floating-point unit. Computational Units (CU) are packaged in clusters of 16 together with 256 Mbytes of memory (16 MBytes each CU.)

Systems can be configured with 2, 4, 6, or 8 clusters, each cluster providing peak speeds of 200 MIPS and 200 64-bit scalar Mflops. Memory configurations range from 256 to 512 MBytes per cluster pairs. In short, a full 8 cluster (128 CU) system provides 2 Gbytes of memory, 8-450 GB disk capacity, and peaks speeds of 1600 MIPS, and 1600 Mflops.

5.1 Hardware Overview

5.1.1 Hierarchical Crossbar Architecture

The ES-1 employs a bilevel hierarchical crossbar interconnection scheme which divides the memory in this uniform address space between two levels. At the first level, each CU is connected to nearby memory banks residing with its cluster through a crossbar switch capable of reaching aggregate data transfer rates of up to 800 Mb/sec (see Fig.5.1). Each cluster can access the memory of other cluster in system over the second-level crossbar switch with aggregate speeds of up to 3.5 Gbytes/sec (see Fig. 5.2.)

The ES-1 also contains dedicated I/O Processors (IOPs) that provide 200 MB/second bandwidth by having each IOP support up to eight 25 Mb/sec full-duplex channels capable of attaching to disk, tape, or interactive graphics systems (figure 5.2) . Up to eight IOPs can be supported in parallel to provide a 1600 Mbyte/sec I/O subsystem.

5.1.2 Node Processors

Each node processor (CU) contains a proprietary RISC microprocessor with its own instruction cache to feeds two integer and three floating-point pipelines. The integer and floating point pipelines within each CU are independent and can execute instructions in parallel. Scoreboarding hardware keeps track of the data dependencies in the pipelined instructions being executed.

To reduce the memory contention within each cluster (16 CU's and 8 memory banks) three types of caches are included on each CU: the Instruction Cache, containing 16K entries (64 Kbytes) is maintained by the IB to keep the pipelines busy; a Data Cache (32 Kbytes) maintains locally referenced data values to reduce

4.9 Shallow Water Equations

The Fortran code implementing the shallow water algorithm involves a 2D rectangular grid with variables: u(i,j), v(i,j), p(i,j), z(i,j), psi(i,j), h(i,j). There are three main loops, two corresponding to the leap-frog time propagation of various quantities, and one for the filtering step. A typical code sequence, used in the updating of the U, V and P variables, is:

```
      do 200 j=1,n
      do 200 i=1,m
         unew(i+1,j) = uold(i+1,j)+
                       tdts8*(z(i+1,j+1)+z(i+1,j))*
                       (cv(i+1,j+1)+cv(i,j+1)+cv(i,j) +
                        cv(i+1,j))-tdtsdx*(h(i+1,j)-h(i,j))
         vnew(i,j+1) = vold(i,j+1)-tdts8*(z(i+1,j+1)+z(i,j+1))
                       *(cu(i+1,j+1)+cu(i,j+1)+cu(i,j)+cu(i+1,j))
                       -tdtsdy*(h(i,j+1)-h(i,j))
         pnew(i,j) = pold(i,j)-tdtsdx*(cu(i+1,j)-cu(i,j))
                       -tdtsdy*(cv(i,j+1)-cv(i,j))
  200 continue
```

Each such loop is followed by code to implement the periodic boundary conditions. Note that there are loops for both the horizontal and vertical boundaries, and in addition some corner values are copied as single items.

For the SPS-2 implementation, we simply parallelize the code in the way described in section 4, introducing two calls to decompose followed by a double PARDO over tasks. Each task performs a double DO loop using indices provided by decompose. That is the only code modification required for the main loops. Similarly the boundary condition loops are parallelized by treating them as 1D grids and using a single call to decompose for each side of the subgrid. In practice task granularity for the boundary loops can become very small and so we have used a smaller number of tasks to parallelize the boundary than were used in the interior of subgrids.

The SPS-2 performed the explicit model at 2.71 Mflops with 64 processors on a 1024x1024 grid, and performed at .117 Mflops on a single processor with a correspondingly smaller grid. Thus parallel efficiency is about 36%. As with the relaxation examples, we studied many potential ways to decompose the grid among the tasks. The most efficient proved to be to use full column slices.

Table 4.10: SPS-2 Block 2D Relaxation (T=T1)

SPS-2 Block 2D Relaxation (Mflops) ((512x512) Grid			
Tasks	Normal	Index	Subroutine
12	0.27	0.14	0.51
15	0.36	0.16	0.60
18	0.44	0.19	0.72
21	0.52	0.22	0.80
24	0.54	0.24	0.79
27	0.86	0.27	0.85
30	0.89	0.28	1.03
33	0.90	0.30	0.97
36	0.99	0.31	1.14
39	0.98	0.31	1.18
42	1.05	0.32	1.22
45	1.11	0.33	1.24
48	1.15	0.33	1.23
51	1.24	0.35	1.24
54	1.24	0.34	1.26
57	1.27	0.33	1.16
60	1.29	0.33	1.19
63	1.30	0.35	1.36

Table 4.9: SPS-2 Block 2D Relaxation

SPS-2 Block 2D Relaxation Comparisons
((Mflops)

D	T	T1	T2	Normal	Index	Subroutine
15	15	15	1	0.36	0.16	0.60
15	15	5	3	0.21	0.18	0.27
15	15	3	5	0.14	0.16	0.32
30	30	30	1	0.89	0.28	1.03
30	30	15	2	0.59	0.33	1.02
30	30	10	3	0.48	0.33	0.83
30	30	6	5	0.36	0.33	0.87
30	30	5	6	0.34	0.32	0.78
45	45	45	1	1.11	0.33	1.24
45	45	15	3	0.62	0.44	1.16
45	45	9	5	0.49	0.43	1.11
60	60	60	1	1.3	0.34	1.20
60	60	30	2	1.10	0.48	1.40
60	60	15	4	0.81	0.54	1.36
60	60	12	5	0.67	0.53	1.39
60	60	10	6	0.56	0.54	1.28
63	63	63	1	1.29	0.34	1.26
63	63	21	3	0.81	0.56	1.25
63	63	9	7	0.51	0.54	1.18
63	63	7	9	0.43	0.55	1.17
63	315	315	1	1.12	0.03	1.04
63	315	105	3	0.73	0.33	1.07
63	315	63	5	0.60	0.44	1.11
63	315	45	7	0.48	0.48	1.14
63	315	35	9	0.39	0.50	1.09

Table 4.8: Effects of 3D subgird aspect ratios.

3D Effect of Subgrid Aspect Ratio

Dim.	Grid	D	T1	T2	T3	Mflops	Efficiency
3	128x128x64	64	64	1	1	3.465	1.000
3	128x128x64	64	32	2	1	3.292	.950
3	128x128x64	64	16	4	1	3.082	.889
3	128x128x64	64	8	8	1	2.723	.786
3	128x128x64	64	16	2	2	2.286	.660
3	128x128x64	64	8	4	2	2.292	.661
3	128x128x64	64	4	4	4	1.424	.411

which case Fortran allocates the data contiguously by columns. Each task processes a subarray, and clearly the columns in subarrays are also then contiguous, although successive columns in any subarray are not contiguous, and involve a substantial stride, except in the case T1=T. Of course the virtual memory system plays havoc with the concept of contiguity at the word level, but it will certainly preserve the concept at the page level, and with a reasonable implementation, contiguous columns should mostly fall in contiguous pages.

The second data layout is one where we store the data for each subgrid contiguously: i.e. the sub-columns are stored contiguously and successive sub-columns are contiguous within each grid block. The second layout is equivalent to the first in case T1=T but not otherwise. The data storage in this case may be implemented as a single large linear array or as a linear array of 2D subarrays. Since individual tasks work almost entirely within a single block, one might expect better performance with this layout due to the improved locality of data.

The three computational methods used are

1. the "normal" method computes the start and end indices of gridblocks within the large 2D data arrays (layout 1) as described above, using multiple calls to the procedure decompose. Standard DO loops are then performed with bounds determined by decompose.

2. the "index" method uses an explicit address calculation to retrieve data elements from a large 1D array.

3. the "subroutine" method, treats the large one-dimensional array as a 1D array of smaller 2D arrays, and passes a pointer to the appropriate 2D array to a subroutine. These declarations are used to equivalence the 1D input parameter array to a 2D array.

Table 4.10 shows the relative performance of these three methods in the case T1=T, where they all involve the same data layout. The index method recommended in E&S actually degrades performance. This is due in part to the fact that the address calculation for each array reference involves 13 integer operations and hence overshadows savings in page referencing/merging. The other two approaches are more or less comparable, with an advantage for the subroutine method in the case of small task number (i.e. large subgrids).

In Table 4.9 we present some examples of performance of the three methods in cases where T1=T, i.e. for non full-column decompositions. Here we have varied the number of processors in steps of 3 to show better resolution. As expected, the subroutine method shows a definite advantage the further we move from the case T1=T.

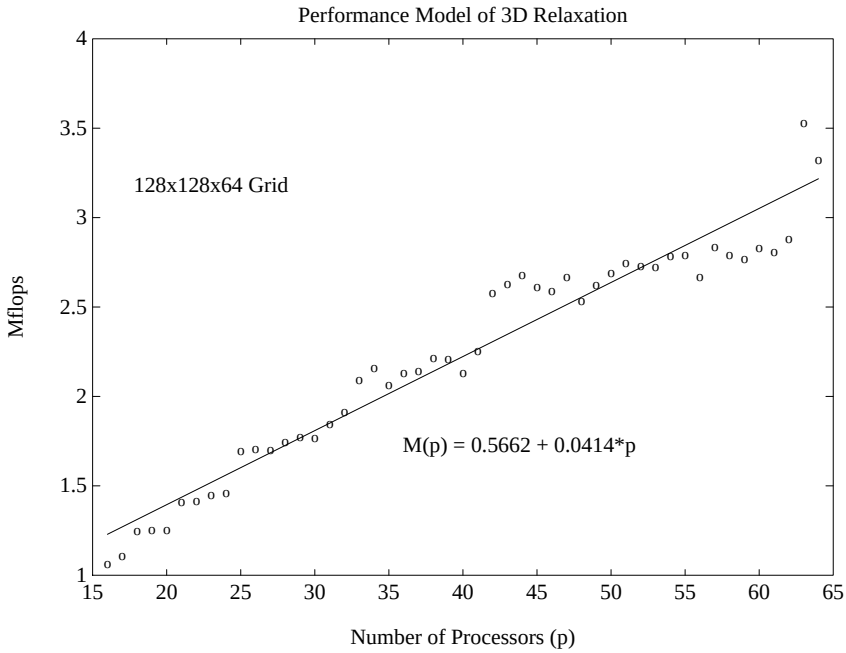


Figure 4.10: Performance Model of 3D Relaxation

It is instructive to study the variation in performance for a given task number T as the task decomposition varies – we call this varying the subgrid aspect ratio, although in fact it is the task subgrid aspect ratio. We present sample results for 2D and 3D relaxations in Table 4.7. The efficiency measures the deviation from the optimal case.

4.8 Implementations of Block Ordering

We have studied two data layouts and three computational methods for implementing block-ordering of matrices.

The first data layout is the one described in section 4.7 where we declare array data as a 2D array, in

Table 4.7: Effects of 2D subgird aspect ratios.

SPS-2 2D Effect of Subgrid Aspect Ratio

Dim.	Grid	D	T1	T2	Mflops	Efficiency
2	1024x1024	64	64	1	2.783	1.000
2	1024x1024	64	32	2	1.962	.705
2	1024x1024	64	16	4	1.190	.428
2	1024x1024	64	8	8	.5495	.197

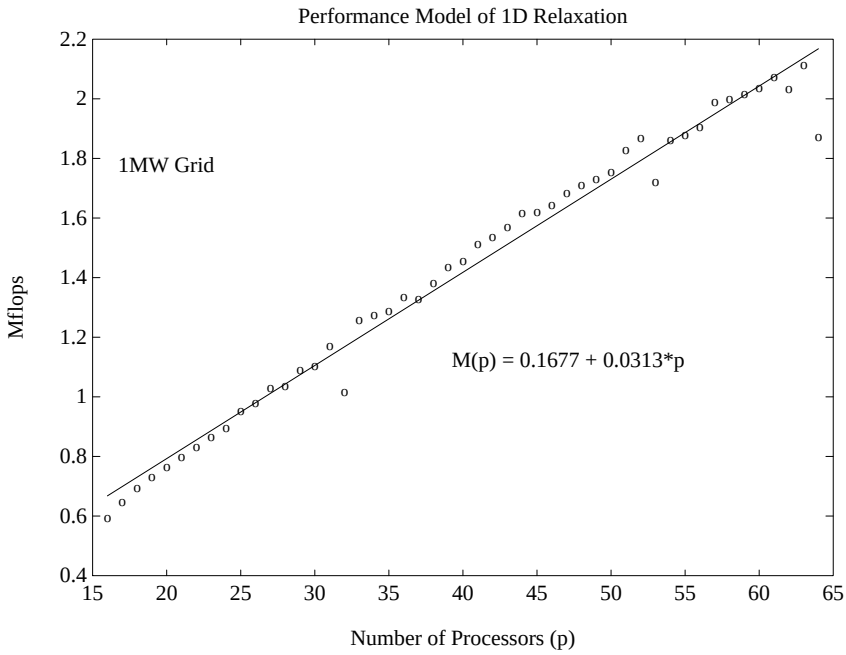


Figure 4.8: Performance Model of 1D Relaxation

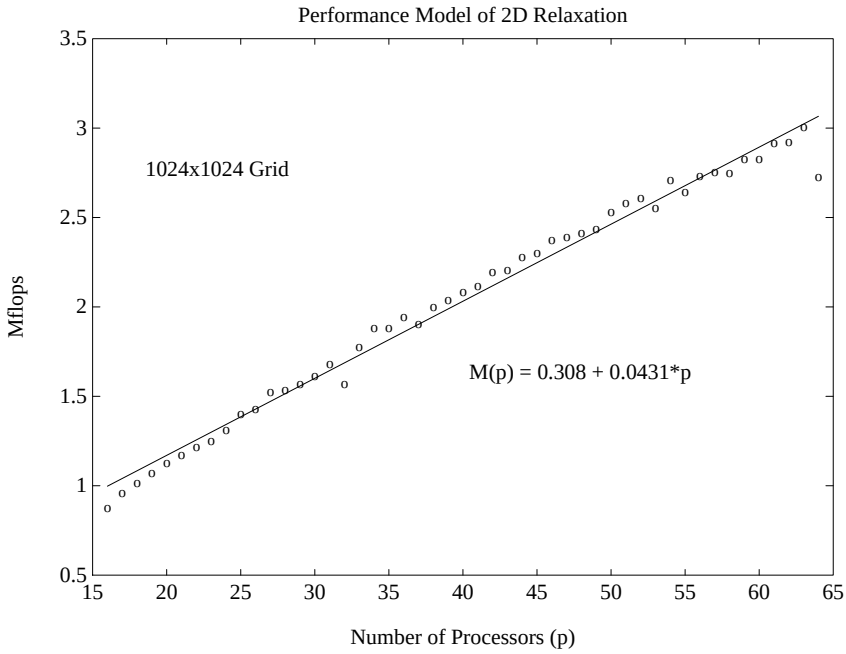


Figure 4.9: Performance Model of 2D Relaxation

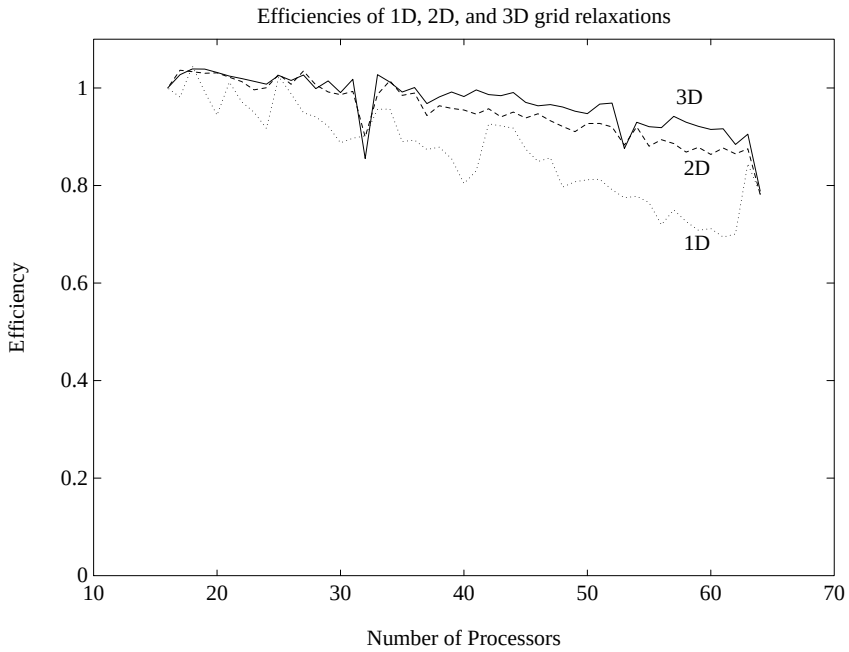


Figure 4.6: Relaxation Efficiency. Grid sizes for given in previous table.

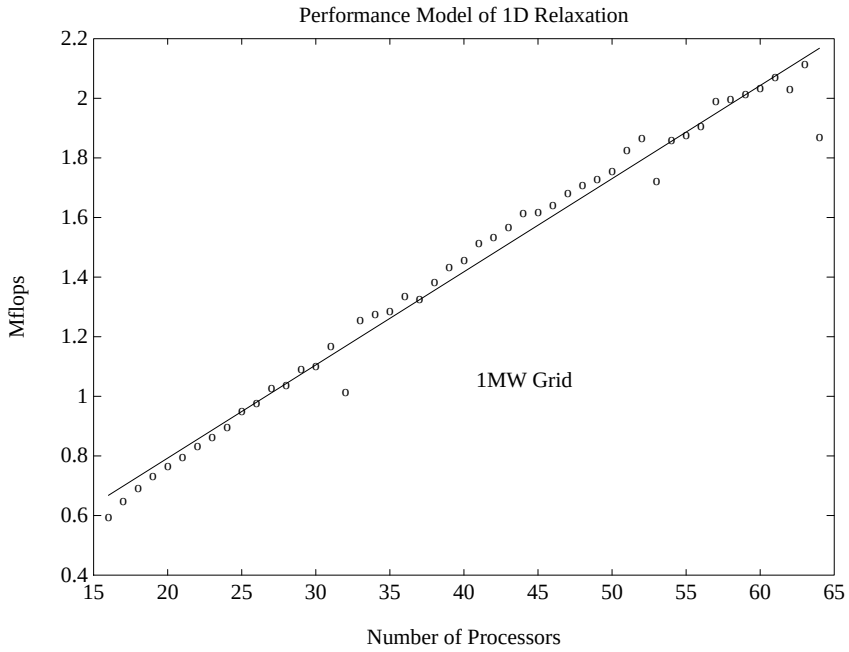


Figure 4.7: Performance Model of 1D Relaxation

Table 4.6: SPS-2 Relaxation Performance

SPS-2 Relaxation Performance

Dim.	Grid	D	T1	T2	T3	Mflops	Efficiency
1	1048576	16	16			0.584	1.000
1	1048576	32	32			1.025	.878
1	1048576	48	48			1.700	.970
1	1048576	64	64			1.913	.819
2	1024x1024	16	16	1		.879	1.000
2	1024x1024	32	32	1		1.554	.884
2	1024x1024	48	48	1		2.420	.918
2	1024x1024	64	64	1		2.783	.792
3	128x128x64	16	16	1	1	1.086	1.000
3	128x128x64	32	32	1	1	1.970	.907
3	128x128x64	48	48	1	1	2.643	.811
3	128x128x64	64	64	1	1	3.465	.798

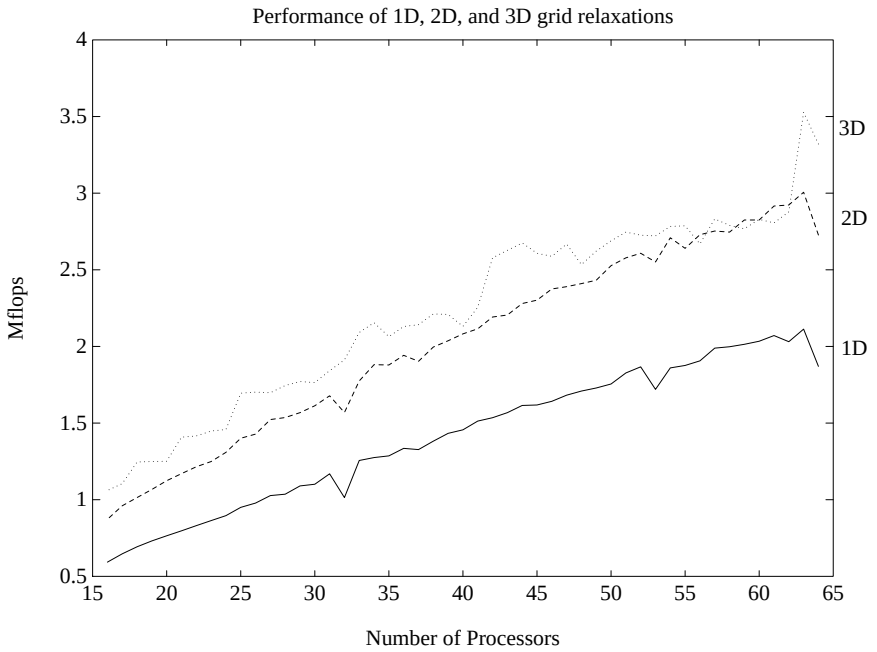


Figure 4.5: Relaxation Performance. Grid sizes for given in previous table.

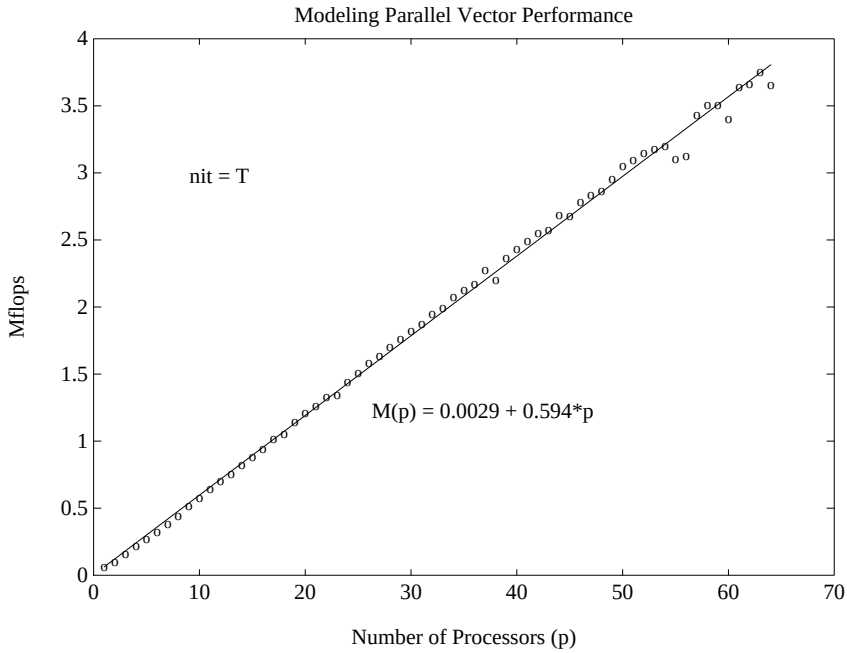


Figure 4.4: Modeling multiprocessor saxpy performance for nit=T case.

each gridsize, the 16-processor result is regarded as efficiency 1.0, and the higher processor number efficiency is computed relative to the 16-processor case (the problems are too large to fit on a single processor with T=D=1).

The measurements described here were performed by repeating the PARDO loop for the relaxation process 10 times. However only the last 9 iterations were actually timed. The first iteration plays the role of an initializer - pages of data that are randomly scattered in memory are brought to their most desirable locations during this first iteration of the PARDO. This seems to be the appropriate measurement approach to give a feeling for likely performance of iterative solvers or time evolution equations, both of which involve perhaps hundreds of iterations of a basic step similar to the one studied here. For a "one-shot" relaxation operation performance will be of order 20% lower due to overheads related to brining in initial copies of pages. Times varied by only a percent or so if 1 iteration was timed rather than 9.

The relaxation performance can also be modeled by linear relationships between the number of processors and the aggregate Mflop performance. For the 1D case above, we can describe the performance as

$$M(p) = 0.1677 + 0.0313 * p$$

A least squares fit, shown in figure 4.7, shows that the relative error is bounded by 10% except for a few cases, most noticeably where p=32 and 64. The actual error of the least squares fit is shown in figure 4.8. This is most likely related to crossing over to a higher communication hierarchy. The performance behavior of the 2D and 3D relaxations show similar characteristics; for the 2D relaxation case we have

$$M(p) = 0.308 + 0.0431 * p$$

likewise, for the 3D relaxation case we have

$$M(p) = 0.5662 + 0.0414 * p$$

The model results for the 2D and 3D cases are shown in figures 4.9 and 4.10, respectively.

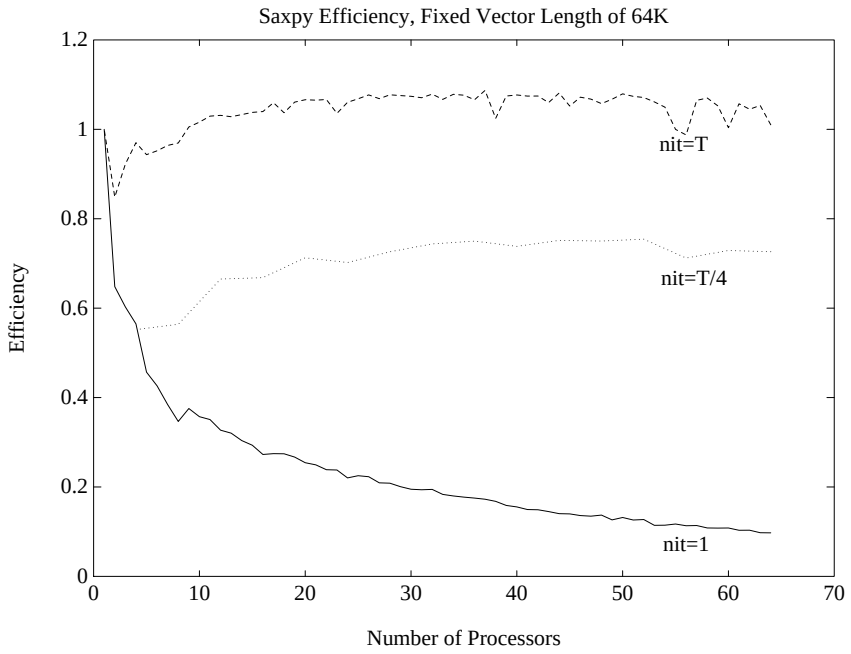


Figure 4.2: Efficiency effects of fixed length, variable work saxpy efficiency. (Same data as previous figure).

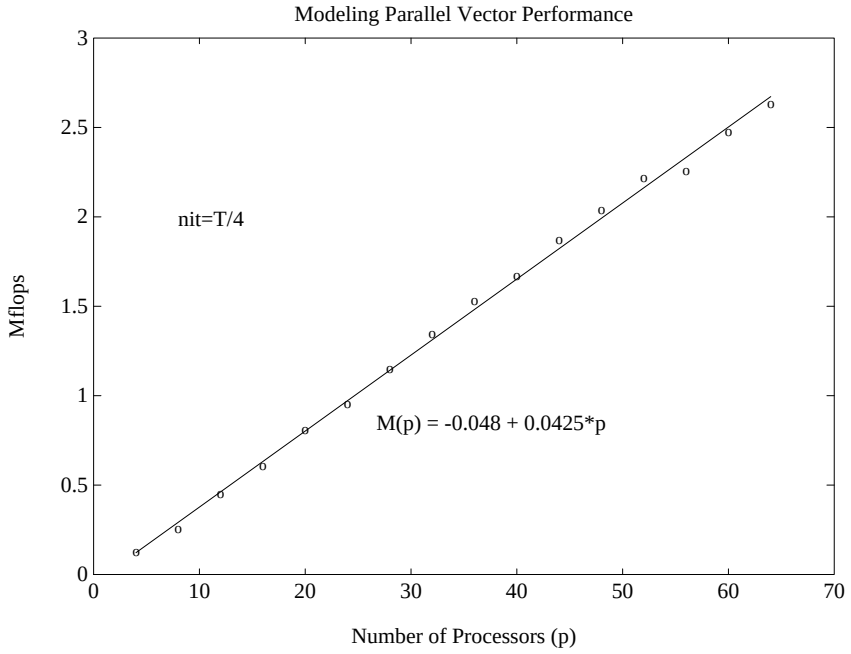


Figure 4.3: Modeling multiprocessor saxpy performance for nit=T/4 case.

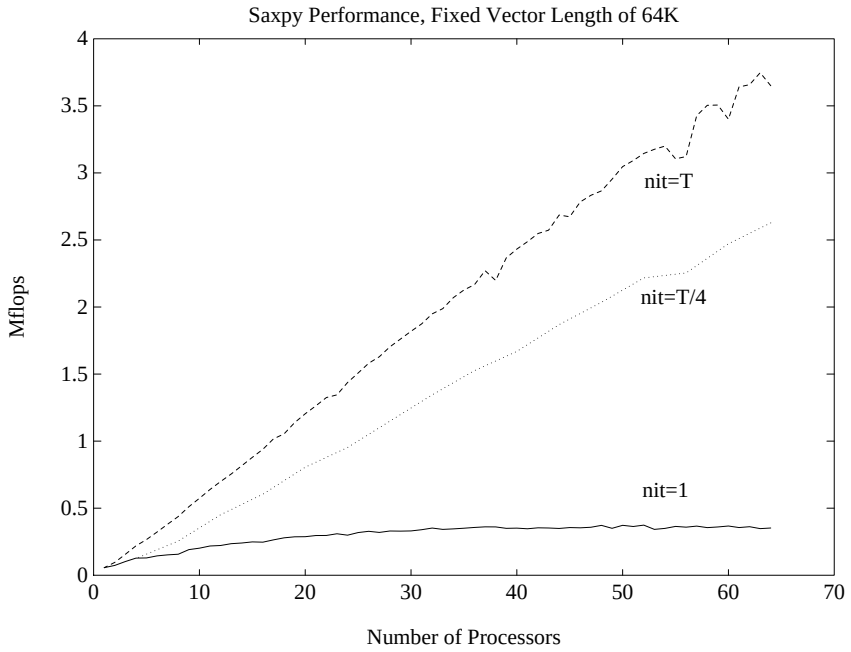


Figure 4.1: Performance effects of fixed length, variable work saxpy performance. Here the loop granularity is set to three different values: fixed, scaled to 1/4 of processors, and scaled to full number of processors.

In the second interpretation of "large vector" we take the vector length with T=D tasks to be T times the vector length with one processor. This a more appropriate measure of parallel efficiency in that vector size (i.e. work to be performed) is being matched to the domain size. In this case we observe results as in table 4.4 below. We note that performance is about 35% of linear in scaling from 1 to 64 processors.

We repeat a similar experiment using nit=T and nit=T/4 and increasing the data resolution by sampling at every processor number. The results in figures 4.1 and 4.2 illustrates the performance and efficiencies, respectively, of this case at the higher processor resolutions. Here we see that the nit=T case yields nearly linear growth with perfect efficiency. The nit=T/4 also yields near linear growth, although the efficiency remains constant around 0.7 for 10 processors or more.

We can model the floating point performance in Mflops as a function of the number of processors (p) for the nit=T/4 case as

$$M(p) = -0.048 + 0.0425 * p$$

as shown in figure 4.3. The maximum relative error of this model is less than 5% for $p > 8$. Similarly, the nit=T case can be modeled as

$$M(p) = 0.0029 + 0.0594 * p$$

as shown in figure 4.4 with a relative error of less than 5% for $p > 10$.

4.7 SPS-2 Relaxation Performance

In Table 4.6 below, we present the results for 1D, 2D and 3D relaxation processes on arrays of varying size and using various number of processors. Figures 4.5 and 4.6 show more subtle effects through the parameterization of the number of processors. We have presented the best performance measured for each task value T, which in all cases occurred with T1=T. We have found only slight variations from using multiple tasks per processor (i.e. T>D) and have presented only results for the cases T=D as a result. For

Table 4.4: SPS-2 Fixed Vector Triad Performance

Fixed Vector Saxpy Performance

Length	Tasks	Domain	nit = 1 Mflops	Effic.	nit = 10 Mflops	Effic.	nit = 10*T Mflops	Effic.
262144	1	1	.054	1.000				
262144	4	4	.107	.495	.243	1.125	.208	.963
262144	16	16	.320	.370	.882	1.021	.956	1.106
262144	64	64	.867	.251	3.053	.883	4.078	1.180
65536	1	1	.051	1.000	.069	1.000	.069	1.000
65536	4	4	.108	.529	.231	.837	.272	.986
65536	16	16	.220	.270	.809	.733	1.106	1.002
65536	64	64	.292	.089	1.831	.415	4.292	.972

Table 4.5: Variable Vector Length SPS-2 Saxpy Performance

Benchmark	Length	Tasks	Domain	Mflops	Efficiency
a = a+s*c	16384	1	1	.054	1.000
a = a+s*c	65536	4	4	.12	.556
a = a+s*c	262144	16	16	.32	.370
a = a+s*c	1048576	64	64	1.21	.350

Table 4.3: Virtual Memory Performance for 1MW Vector.

Single Node Multiple Domain
Saxpy Performance (Mflops)

Benchmark	Tasks	Domain	Mflops	Efficiency
a = a+s*c	1	1	.054	1.000
a = a+s*c	1	16	.0316	.585
a = a+s*c	1	32	.0329	.609
a = a+s*c	1	48	.0332	.615
a = a+s*c	1	64	.0327	.606
a = a+b*c	1	1	.052	1.000
a = a+b*c	1	16	.0250	.481
a = a+b*c	1	32	.0258	.496
a = a+b*c	1	48	.0257	.494
a = a+b*c	1	64	.0257	.494

4.6.5 Modelling Multiprocessor Overheads and Task Granularity

As discussed in section 4.3, the SPS-2 operating system copies parent data spaces to child tasks and merges pages of data on task completion that have been written into by multiple processes. Since the indices i in operations (1,2) are all distinct, it follows that if the work is divided among several tasks, no two tasks ever operate on the same element a(i) and thus the merging rules are satisfied trivially in all cases. Basically this means that no merging between children should be required on task completion. This is not quite true since, except in exceptional cases, a division of the vector a(i) among T tasks will generally not provide segments restricted to disjoint sets of pages. Thus some limited page inter-child merging should be expected, except in the special case that a is chosen to be page-aligned and where N is a multiple of T and N/T is a multiple of the page size.

In this study we allocate large vectors a, b, c and we study performance as the number of tasks T is increased. We use two possible interpretations of "large vectors". In the first case, we choose fixed size vectors and vary T. As per-task data granularity decreases with increasing T, performance decreases relatively – i.e less than linear speedup is exhibited. Linear speedup may be restored by increasing the work performed per task. This we do by repeating the operations (1,2) nit times in each task. For constant nit we find that performance is sublinear, but the deviation from linear is less the larger nit becomes. With the choice nit = 10*T completely linear growth is exhibited. Table 4 exhibits performance measured for several vectors lengths and using from 1 to 64 processors. The three columns correspond to the choices nit = 1, nit = 10 and nit = 10*T. For each choice we provide both Mflops and computational efficiency.

The data for the case nit = 10*T are essentially linear - i.e. efficiency is 100% to within measurement accuracy. Thus we conclude that the work involved in the case nit = 10*T dominates the PARDO overheads. Within a single processor this work amounts to applying a saxpy operation at each point of a vector of length 262144/64 and repeating this 640 times, or a total of 5.2 million floating point operations (task granularity). Linearity was not achieved with the choice nit = 10 which corresponds to a task granularity of 82,000 floating point operations. Our conclusion is that tasks with granularity below 100,000 operations will encounter severe inefficiencies.

Table 4.2: SPS-2 Single Node Vector Triad Performance

SPS-2 Single Node Saxpy Performance (Mflops)				
Benchmark	1-Node	64-Node *	Sun Sparc	MIPS
a = a+s*c	.054	3.456	.9979	.9790
a = a+b*c	.052	3.328	.7266	.6919

chapter 6, there would be no point in measuring multiprocessor versions of these two benchmarks. Neither involves any interprocessor communication, and so one can predict exact linearity of performance over the single-processor case. Because of interaction with the virtual memory system this is no longer true on the SPS-2.

4.6.3 Single Node Performance

Single-node performance is measured by ensuring that all of the vectors a, b, c are resident in the node memory. Because of the virtual memory, one can run a program on a single processor (i.e. PARDO of length 1, or simply a DO), but with data spread over a larger number of processors (the domain). We will denote the number of tasks actually computing by T, and the domain size of processors requested to provide virtual memory by D in the sequel. To ensure true single processor behavior, one must therefore run on a single processor with the domain size also set to 1, i.e. T=1 and D=1. The performance results, presented in Table 4.2 are in agreement with the scalar benchmarks presented earlier:

We have also studied (1-2) in two more interesting situations:

a) a single processing node where N is so large that the vectors cannot reside in a single node ($T = 1, D > 1$).

b) the true parallel processing case where we allow more than one processor to perform the work ($T > 1$ and $D > 1$).

Test a) gives a measure of the cost of accessing virtual memory, while test b) involves spawning child tasks (since $T > 1$) and therefore measures the overheads of task creation and merging. The simplest case discussed above with all vectors resident in one node and only one task is useful as a comparison point.

4.6.4 Single Processor Access to Virtual Memory

In case a) we examine the effectiveness of the virtual shared memory environment of the SPS-2 from the viewpoint of a single processor. A remarkable feature of the SPS-2 is that a program computing on only one processor can solve a problem that is too large to fit in a single processor. The ability to do so stems from the SPS-2 virtual memory. For this study we take increasingly larger vectors in (1-2) and complete the operation using only one processor. Table 4.3 shows the degradation in performance resulting from this use of virtual memory.

The efficiency number given here is the ratio of performance with D processor domain to the performance on 1 processor. Thus we observe about a 40-50% drop in performance due to the overhead of virtual memory access. Of course the overhead would be reduced if more significant work were performed at each vector element.

Table 4.1: SPS-2 Scalar Performance

SPS-2 Scalar Performance (Mflops)

Benchmark	1-Node	64-Node *	Ext. Prec. *	Sun Sparc	MIPS
c = a+c	.0545	3.488	6.515	2.617	4.939
c = a*c	.0510	3.264	6.180	2.589	2.952
c = a*c unrolled	.0874	5.594	8.211	3.256	3.529
c = c + a*c	.0753	4.819	8.570	3.254	3.737
c = c + a*c;a = a + c*b	.0874	5.594	8.730	3.937	3.969
super-unroll	.266	17.024	18.266		

4.6 Performance Analysis of the Myrias SPS-2

4.6.1 Scalar Performance

In order to get a feeling for the likely peak machine rates of the system, we have analyzed performance of several simple scalar computations. These were performed on a single SPS-2 processor, with a loop sufficiently long (e.g., 10,000,000) to ensure that loop overhead or similar effects are not being measured, and that timing resolution is not causing perturbations. For comparison, results from a SUN Sparc-station 4/110 and from a MIPS workstation have been included. The results are shown in Table 4.1.

Each algorithm was implemented as a Fortran program and compiled with the -O2 option of the Myrias Fortran compiler, mpfc, giving the highest level of standard optimization. The data types were real*4. Slightly higher performance would be obtained with double precision data, and higher still with extended precision, which is however not compatible with standard Fortran. Extended precision is obtained using the -O2f compiler option. For uniformity of results we quote only the single precision values. Double precision data would have decreased the largest problems that we could study in the PDE solution and application benchmarks. For comparison, we include above the extended precision benchmark values. The 64-Node figures and extended precision figures are simply linear extrapolations of the single node values and are useful in comparisons with the real 64-node parallel benchmarks presented later.

The unrolled version of c = a*c was obtained by unrolling 4 times. The super-unroll is a special benchmark designed to make maximum use of the 68881 registers and is totally unrepresentative of any normal computation. Since it is specific to the SPS-2 it was not measured on other systems. The actual code consists of the three lines:

```
fi = fi * fi
sum = sum * fi
sum = sum - fi
```

repeated 16 times and surrounded by a DO loop of length 1 million. Its only purpose is to show the peak obtainable speed of the node chipset.

4.6.2 Computational Core Performance

We have performed three studies of various vector kernels encompassing both single processor and multiple processor cases. We remark that on a standard distributed processor, such as an Intel iPSC/860 described in

```
            pardo 10 i=1,n
                sum = sum + expr(i)
10          continue
```

as a reduction operation and would generate appropriate code.

4.5 Programming Environment

The ideal parallel software development environment is one where one can take a serial program from a CRAY, SUN or other system, and run it on the SPS-2 without change. The program will likely run with poor performance: for example no parallelism has been introduced, but at least it is running on the new hardware. Then, one begins the refinement process of optimizing the software to the SPS-2. The current Myrias system comes close to attaining this goal – far closer than any other massively parallel system. Serial programs indeed run immediately on the system. In fact even programs with massive memory needs will run without modification: while the programs run on a single processor, they can access memory stored on other processors. The only requirement is to request sufficient SPS-2 processors (domain size) before running.

To begin the refinement process, one next introduces PARDO instead of DO wherever independent iterations are observed in DO loops. Performance is generally then greatly enhanced since multiple processors will be working on the code previously handled by one. Unfortunately this is the end of the line on SPS-2 — no further optimizations are available. It is desirable that further levels of refinement be made available. For example, users may in the end want to specify which processors specific tasks should run on, or to lock and share specific memory locations for write access without incurring the overhead of a complete merge.

An annoying feature of PARDO is that it is a Fortran extension. An SPS-2 parallelized program will not run on any other system. Had Myrias chosen to use Comment Directives such as

```
c$sps   pardo
        do 10 i=1,n
            ...
10      continue
```

instead of a Fortran extension, the programs would have retained portability to serial machines.

```
            <Code to be executed>
10      continue
```

Any Fortran DO loop may be replaced by a PARDO provided the body of the loop is independent across all loop iterations. Each thread of execution performs completely independently in its own address space, starting with a copy of the parents memory. Execution of a child proceeds in normal sequential mode, except that PARDO's may be nested recursively. On completion of all children, the memory states of the children are merged to form the new memory state of the parent. Thus a child can never affect the memory of another child, but can affect the memory state of its parent, but only after all children merge.

A second available facility for parallel task execution is the PARBEGIN which implements a parallel IF-THEN-ELSE or case statement. However the PARBEGIN may be thought of as a special case of PARDO where the different cases are simply different loop iterations, and so we will ignore PARBEGIN in the sequel.

4.3 Task Memory Merging Rules

The rules for merging of child memories, at a parent memory address, on child task completion are

- If no child stored a value at the address, the location in the parent memory retains its original value.
- If exactly one child changed the value at the address, the location in the parent receives the last value from the child.
- If more than one child stores a value at the address the result is unpredictable (undefined) unless all values stored are the same.

Efficiency is maintained throughout the process by using a copy-on-write approach which ensures that most of the global address space is never really replicated. While the hardware ignores the issue of whether a given memory location is undefined, the latest version of the Myrias debugger may be used to detect undefined values. A significant possibility for efficiency in merging occurs in connection with merge timing. Logically, merging occurs upon termination of the last child task and is completed before the parent executes the next instruction. In practice, merging can begin as soon as a child task completes, with various merging activities proceeding on the very processors which would otherwise be idled by the terminated task(s). Furthermore it is not essential that the merge actually complete for a page until that page is referenced in a subsequent instruction by the parent – indeed in many circumstances some pages may never be referenced before the parent terminates, and thus need never be merged!

4.4 Use of Recursion

The programming approach to the system is functional in nature and is based on parallel recursion. In fact recursion has been added to Fortran as a Myrias extension, and is strongly recommended to users in the various manuals. As an example, we describe an implementation of a global vector sum, assuming the vector is initially of length N. The parent creates two tasks, each intended to sum half of the elements, storing the results in variables left and right. Each child task similarly spawns two more children, storing results into two local variables of the principal child, and so on. On completion of a pair of child tasks, its parent has the two partial sums in separate variables, and therefore it proceeds to add these, providing the sum to the next parent above. In this way the result is obtained at the global parent in time $O(\log N)$ without ever sharing memory. All assignment of work to processors is handled automatically.

This programming style is undoubtedly elegant, but because of the recursion involved, the resulting program is not recognizable as a standard Fortran program. A far more effective solution for this example would be provided if Myrias supported Parallel Reduction Operations in some form. For example, the summation variable, perhaps sum, might be declarable to be a candidate for reduction, in which case the compiler would recognize the construct

Chapter 4

Myrias SPS-2: A Distributed/Shared Memory Hybrid System

The Myrias SPS-2 computer, built by Myrias Research Corp. of Edmonton, Alberta, is a massively parallel distributed memory computer with a virtual shared memory software environment. The largest system built contained 1024 processing elements , although the design allowed for even larger sizes.

Each processor is a 32-bit Motorola 68020 microprocessor, with a 68881 floating point coprocessor, and 4 Mbytes of local memory. The architecture is a three-level hierarchical bus design, utilizing 33 Mbytes/sec busses to interconnect processors within clusters and clusters to each other. Processors are assembled in groups of four on a board connected among each other by a bus, along with an I/O port controller. At the second level in the hierarchy is the card cage, containing 16 processor boards and thus 64 processors, as well as one or two off-cage communication boards. A pair of busses connect the 16 boards of a cage. Each communication board supports four off-cage links which can be connected to other cages or to the front end computer, which is currently a SUN 3 work-station. While the host provides user access to the SPS-2, all SPS-2 tasks execute entirely on SPS-2 nodes. The host is used only for compilation, program initiation and access to I/O devices.

4.1 SPS-2 Software and Programming Model

All system software has been developed for an abstract system called the G Machine. This is intended to clarify the programming model and to provide a layer of insulation from hardware details. The SPS-2 supports a global 32-bit virtual address space. Processes achieve parallelism by spawning independent subtasks, each executing in a copy of the parent's memory. There is no explicit concept of shared access to memory locations by multiple processors. However upon completion of child processes their memories are merged with the parent's memory according to specific rules explained in detail below. The subtask creation and subsequent merging process represent an implicit shared memory concept.

4.2 The PARDO construct

A single facility (but see PARBEGIN below) is available to effect subtask creation, execution and merging: the PARDO construct. This is implemented as Fortran and C language extensions: pardo and parfor respectively). The PARDO model used by Myrias is somewhat unusual in that there are no possibilities for multiple processors to directly modify shared data (see below). A PARDO is executed by specifying a code segment to be executed and the number of child tasks to be run.

```
pardo 10, I = 1, N
```

Table 3.4: Results of CMPS on Shallow Water equations.

Performance of SWM on the CM-2				
Grid Size	Observed (secs)	Predicted (secs)	Error	Gflops (scaled to 64K)
64 x 64	5.1	4.7	8.6%	0.9
64 x 128	8.0	8.0	0.5%	1.1
128 x 128	13.5	12.7	6.1%	1.2
128 x 256	23.2	22.7	2.0%	1.5
256 x 256	41.5	39.6	4.5%	1.6
256 x 512	77.1	75.0	2.7%	1.8
512 x 512	144.0	135.1	6.2%	2.0
512 x 1024	??	263.1		2.1
1024x1024	??	504.1		2.1

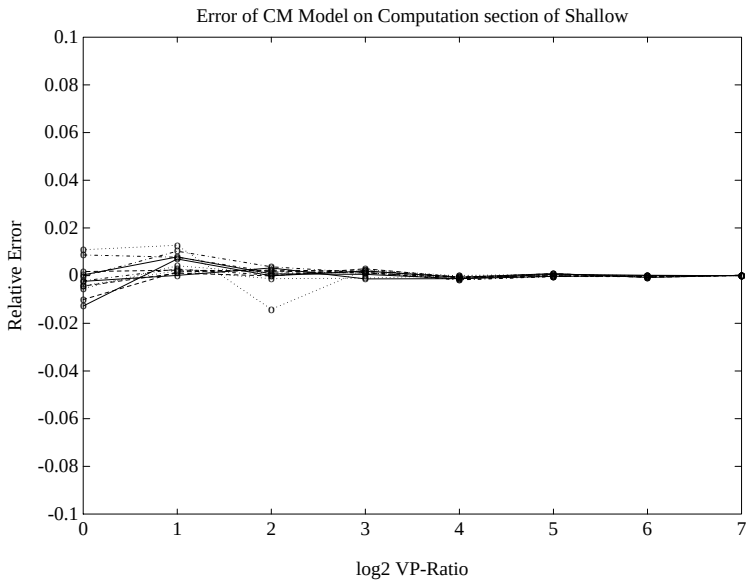


Figure 3.29: CMPS model error on computational section of the Shallow Water equations.

The last two entries in table 3.4 illustrate an important use of these performance models. The Connection Machine we used for these tests did not have sufficient memory to hold the larger versions of the application. We notice that, as CMPS predicts, larger problem sizes generally result in more effective use of the computing hardware. The last column in table 3.4 illustrates this. We were interested in seeing whether going a larger memory machine would improve performance. With the CMPS system, we could answer this question instantly before making the decision to purchase the memory upgrade.

```
      subroutine calc1()
      include 'input.cb'
      real pnorth(m,n)

      pnorth = cshift(p, 1,-1)
      cu=.5*(p + pnorth)*u
      cv=.5*(p + cshift(p, 2,-1))*v

      z = (fsdx*(v-cshift(v,1,-1)) -
     >       fsdy*(u-cshift(u,2,-1)))/
     >       (cshift(pnorth,2,-1)+
     >       cshift(p,2,-1)+p+pnorth)

       h = p+.25*(cshift(u,1,1) *
     >       cshift(u,1, 1)+ u*u +
     >       cshift(v,2, 1)*
     >       cshift(v,2, 1)+ v*v)

      return
      end
```

Figure 3.27: Example CM-Fortran subroutine from Shallow Water Equations.

```
%   Paris function              Occurence     function-specific parameters
%
 CM_f_add_always_2_1L              7
 CM_f_add_always_3_1L              1
 CM_f_add_mult_const_1L            4
 CM_f_divide_3_1L                  1
 CM_f_move_1L                      9
 CM_f_mult_add_1L                  4
 CM_f_mult_const_add_1L            4
 CM_f_mult_const_sub_1L            1
 CM_f_mult_const_subf_1L           7
 CM_f_mult_subf_1L                 1
 CM_f_multiply_3_1L                2
 CM_f_multiply_always_3_1L         1
 CM_f_news_add_always_2_1L         2             dim=1
 CM_f_news_add_always_3_1L         2             dim=1
 CM_f_news_sub_always_3_1L         2             dim=0
 CM_f_news_sub_always_3_1L         1             dim=1
 CM_f_sub_mult_const_1L            1
 CM_f_subtract_always_3_1L         2
 CM_get_from_news_1L               3             dim=0
 CM_get_from_news_always_1L        3             dim=0
 CM_get_from_news_always_1L        4             dim=0
```

Figure 3.28: Example of CMPS output for floating point computation/communication in the Shallow Water application.

```
      SUBROUTINE CALC1
C
C        COMPUTE CAPITAL  U, CAPITAL V, Z AND H
C
      PARAMETER (N1=65, N2=65)
      COMMON  U(N1,N2),V(N1,N2),P(N1,N2),UNEW(N1,N2),VNEW(N1,N2),
     1        PNEW(N1,N2),UOLD(N1,N2),VOLD(N1,N2),POLD(N1,N2),
     2        CU(N1,N2),CV(N1,N2),Z(N1,N2),H(N1,N2),PSI(N1,N2)
C
      COMMON /TIME/ T100, T200, T300
      COMMON /CONS/ DT,TDT,DX,DY,A,ALPHA,ITMAX,MPRINT,M,N,MP1,
     1              NP1,EL,PI,TPI,DI,DJ,PCF
C
      FSDX = 4./DX
      FSDY = 4./DY

      DO 100 J=1,N
      DO 100 I=1,M
      CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
      CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
      Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
     1          -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
      H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1               +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE
C
C     PERIODIC CONTINUATION
C
      DO 110 J=1,N
      CU(1,J) = CU(M+1,J)
      CV(M+1,J+1) = CV(1,J+1)
      Z(1,J+1) = Z(M+1,J+1)
      H(M+1,J) = H(1,J)
  110 CONTINUE
      DO 115 I=1,M
      CU(I+1,N+1) = CU(I+1,1)
      CV(I,1) = CV(I,N+1)
      Z(I+1,1) = Z(I+1,N+1)
      H(I,N+1) = H(I,1)
  115 CONTINUE
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)

      RETURN
      END
```

Figure 3.26: Example Fortran 77 subroutine from Shallow Water Equations.

3.7 Shallow Water Application Example

As an example of the application performance analysis the CMPS system provides, we describe the implementation of the Shallow Water equations in CM-Fortran. The code implementing the explicit algorithm involves a 2D rectangular grid with three main loops: two corresponding to the leap-frog time propagation of various quantities, and one for the filtering step.

One of the original Fortran 77 subroutines, taken from Hoffman, et. al [46], is shown in figure 3.26. Here the main loop runs over the 2D arrays which update of the U, V and P variables. The second half of the code handles periodic boundary conditions by copying rows and columns that lie on the edge.

The corresponding CM-Fortran subroutine is shown in figure 3.27. Here the expressions use the Fortran 90 array syntax. Since the expressions involve matrices, rather than scalars, no subscripts appear in this code. Furthermore, array shifting is performed by the Fortran 90 circular shift,`cshift()`, operation. For example,

```
A = cshift(B, 1, -1)
```

is equivalent to writing

```
      do 100 j=1,N
          do 200 i=2,M
              A(i,j) = B(i-1,j)
200       continue
          A(1,j) = B(M,j)
100   continue
```

Each shift operation is essentially a grid communication on the CM.

3.7.1 CMPS Analysis

We use the CMPS compile, link, and run steps outlined in section 3.6 to study the performance behavior of the Shallow Water equations.

The CMPS analysis yields interesting information. For example, in the subroutine shown in figure 3.27, we see that there are 10 `cshift()` operations. In reality, the CM-Fortran (v. 0.6) compiler performs only 7 such operations – it recognizes that some `cshift()`s are redundant and stores them into temporary variables to reduce the number of physical communications. The CMPS system automatically models this compiler behavior. This is yet another example for the need to have an intergrated CMPS software environment rather than just a paper-and-pencil analytical performance model. A typical output of the CMPS system is shown in figure 3.28.

From this structured profiler information, we study the relative errors of the model for computation phase of the application. (We can study the computation phase in isolation, since it is independent of the vp geometry.) Figure 3.29 illustrates the relative error of the CMPS model on the computational section of the Shallow Water equations. For all vp ratios the relative error is bounded by about 2%.

Finally, the results for the complete application are shown in table 3.4. This table show various problem sizes for the application and the respective differences between the observed and predicted performance. In all cases the overall application performance generated by the CMPS model is below 10% relative error. We notice that the CMPS model always underestimates the running time of the application. This is due to the fact that we are only modeling the floating point aspect of the application. (We do not, for example, model Paris integer functions.) This capability could always be added, but the results from the CMPS are sufficient to illustrate that the basic concept works. Likewise the relative errors are slight higher than they were for individual Paris functions for the same reason. We also notice that the overall CMPS error is slightly less for the square NxN problem sizes. This is due to the fact that the communication parameters determined by the original analytical model (t_E, t_M, and t_I) fit square geometries better.

Paris Function	Time/Function (secs)	#Calls	Total Time (sec)
CM_f_add_always_2_1L	1.521e-01	4	6.0820e-01
CM_f_add_mult_const_1L	2.089e-01	7	1.4619e+00
CM_f_divide_3_1L	7.485e-01	3	2.2456e+00
CM_f_move_1L	1.387e-01	4	5.5472e-01
CM_f_mult_add_1L	2.256e-01	2	4.5112e-01
CM_f_mult_const_add_1L	1.988e-01	4	7.9532e-01
CM_f_mult_const_sub_1L	1.988e-01	3	5.9649e-01
CM_f_mult_const_subf_1L	1.988e-01	3	5.9649e-01
CM_f_mult_subf_1L	2.256e-01	2	4.5112e-01
CM_f_multiply_3_1L	1.587e-01	2	3.1746e-01
CM_f_multiply_always_3_1L	1.287e-01	3	3.8595e-01
CM_f_news_add_always_3_1L	3.258e-01	6	1.9549e+00
CM_f_news_sub_always_3_1L	3.258e-01	6	1.9549e+00
CM_f_sub_mult_const_1L	2.089e-01	5	1.0443e+00
CM_f_subtract_always_3_1L (dim=1)	1.521e-01	2	3.0410e-01
CM_get_from_news_1L (dim=1)	3.024e-01	2	6.0484e-01
CM_get_from_news_always_1L (dim=0)	2.757e-01	1	2.7569e-01
		-----	----------
Total		59	1.4602e+01

Figure 3.25: An example of the output from the CMPS analyzer shows Paris instructions, the number of seconds required to complete the instructions (based on the performance model), the number of times each instruction is called, the total time spent executing each instruction, and a final sum of the elapsed time required to execute to complete code.

Alternately, they can be updated or changed by simply editing this file. The format of this file, "comps.arch", is

```
<Function-name>          <alpha>      <beta>
```

where α denotes startup cost (in μseconds), and β denotes the slope of linear relationship. An example listing would include

```
t-M                                  2.40    5.60
t-I                                 20.50   10.40
t-E                                  0.01    5.50
CM-f-add-2-1L                       10.60   33.16
CM-f-add-3-1L                       10.78   33.47
CM-f-add-constant-2-1L              12.09   28.35
CM-f-divide-2-1L                     1.57   88.34
CM-f-divide-3-1L                    13.92  139.95
CM-f-divide-constant-2-1L           19.00   88.28
CM-f-multiply-3-1L                  10.93   29.17
CM-f-multiply-constant-2-1L         12.04   24.05
...
```

The three special instructions (t-M, t-I, t-E) describe the communication times involved in virtual, on-chip, and off-chip communications, respectively. For these functions, the time is given by $T(n) = \alpha + \beta * n$, where n is the number of bytes transmitted. That is, β has dimensions of μsecs per byte. This information is used by the CMPS analyser to compute the elapsed time required for communication operations. For the Paris floating point functions, β has dimensions of μsecs per vp-ratio.

Output of CMPS

The analysis phase (figure 3.24) produces a final performance "spec sheet" as output. This text file lists the specific Paris functions traced under CMPS, the number of seconds required to complete the instructions (based on the performance model), the number of times each instruction is called, the total time spent executing each instruction, and a final sum of the elapsed time required to execute to complete code. The a sample output of the CMPS "analyzer" is shown in figure 3.25.

By trying various virtual CM geometries, one can see how different data mapping strategies affect overall application performance. This process could be aided by an "optimizing" module which allows for iterative improvements or some linear constraint optimization solver. Furthermore, one can also experiment with different hardware characteristics (for example, changing values of t_E or constants) and we see how different component designs would improve the performance of the parallel machine. For example, a hardware designer can ask questions such as, "What if we doubled the virtual processor communication bandwidth (t_M)?" and instantly see what performance impact this would have at the *application* level. In short, the CMPS could be used by

- end users
- algorithm designers
- hardware designers
- compiler and language developers

The next section demonstrates how the CMPS system is used on a CM-Fortran application to analyze its performance.

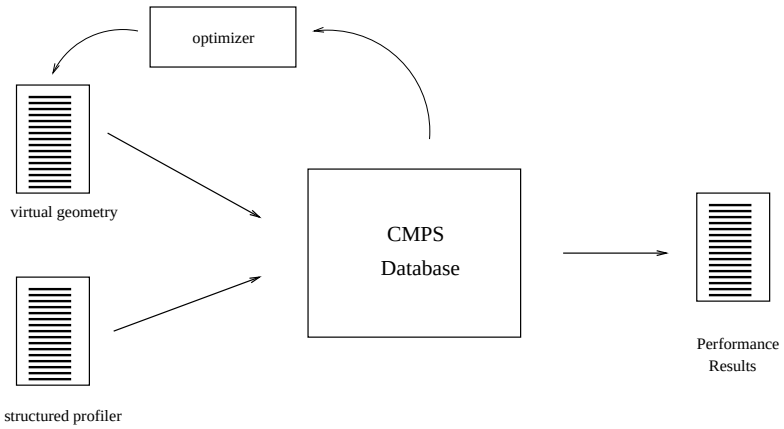


Figure 3.24: Analysis phase of the CMPS system. See text for details.

Specifying virtual geometry information

The file specifying a virtual geometry mapping may have either been generated automatically by the CMPS run phase (in the file "cmps.geometry") or may be simply generated by hand. The format of this output text file describes an N-dimensional vp geometry as an 3xN matrix

$$v_1 \quad o_1 \quad f_1 \tag{3.16}$$
$$v_2 \quad o_2 \quad f_2 \tag{3.17}$$
$$\vdots \tag{3.18}$$
$$v_n \quad o_n \quad f_n \tag{3.19}$$

where v_i, o_i, and f_i, denote the sizes (in base two, since each value must be a power of two) of the virtual, on-chip, and off-chip sizes for dimension i. The total size of the vp-geometry grid $d_1 \times d_2 + \times \cdots \times d_n$, is then given by

$$d_i = 2^{(v_i + o_i + f_i)}$$

Thus, the following three lines

$$\begin{array}{ccc} 2 & 1 & 1 \\ 1 & 1 & 2 \\ 0 & 2 & 3 \end{array}$$

defines a 3-D vp-geometry mapping of a 16x16x32 grid with the following virtual, on-chip, and off-chip dimensions

Dimension	Total	Virtual	On	Off
0	16	4	2	2
1	32	2	2	4
2	32	1	4	8

Specifying Architecture Information

The architecture "database" is a text file consisting of the α and β parameters that are given in sections 3.4 and 3.3. These parameters can be found by conducting baseline experiments on existing hardware.

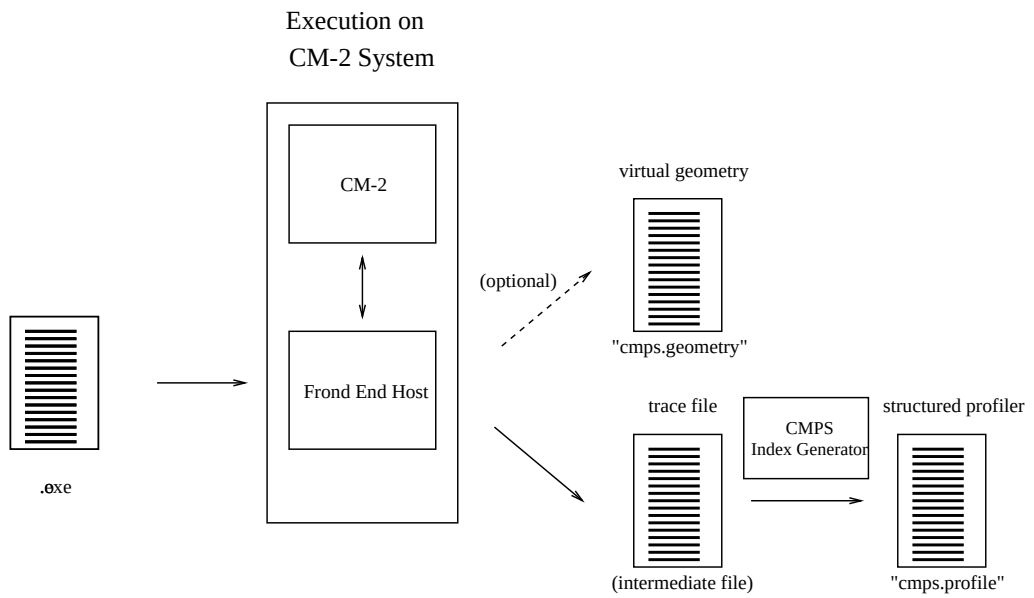


Figure 3.23: Run phase of the CMPS system. See text for details.

where `[frequency]` denotes the number of times called. For example,

```
_CM_f_add_always_3_1L              1
_CM_f_add_mult_const_1L            4
_CM_f_divide_3_1L                  1
_CM_f_move_1L                      9
_CM_f_mult_add_1L                  4
_CM_f_mult_const_add_1L            4
_CM_f_mult_const_sub_1L            1
_CM_f_mult_const_subf_1L           7
_CM_f_mult_subf_1L                 1
_CM_f_news_sub_always_3_1L         3          dim = 1
_CM_get_from_news_1L               3          dim = 0

...
```

The last two entries describe Paris functions which involve communication, hence they also record the axis is which the communication took place. (The direction is irrelevant, since the communication cost is identical.) This information, will then be coupled with t_E, t_I and t_M values in the "architecture database" (see next section) to provide performance estimates.

3.6.5 Analysis

The final analysis phase, shown in figure 3.24, utilizes input files that describe (1) a virtual geometry mapping, (2) an application profile, and (3) an architecture description to provide final performance estimates. The CMPS "engine" is an **awk** script that estimates the Paris floating point performance by having a database of the various α and β parameters as appears in tables

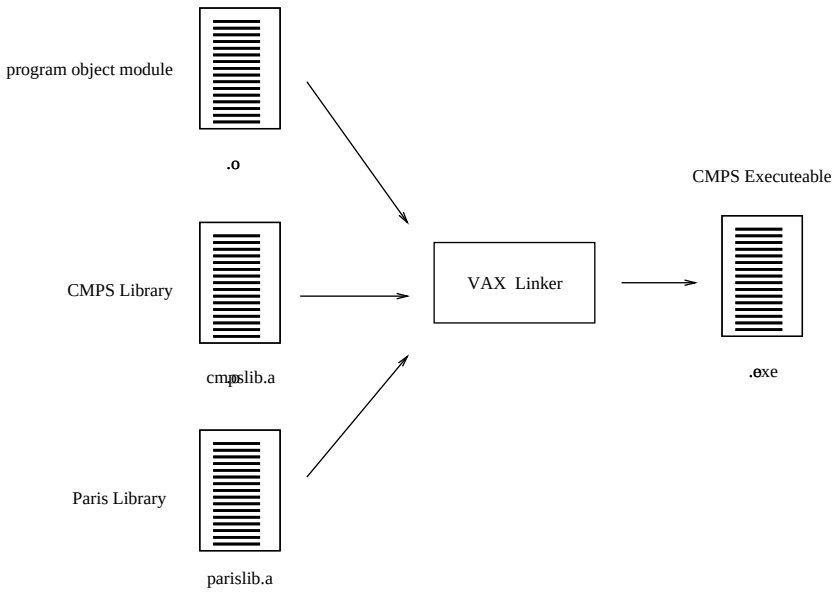


Figure 3.21: Link phase of the CMPS system. See text for details.

```
static int CMPS_trace = 1;
static FILE *fcmps;

        .
        .
        .

void CMPS_f_add_always_2_1L(dest, source1, s, e)
     CM_field_id_t dest;
     CM_field_id_t source1;
     unsigned s;
     unsigned e;
{
    if (CMPS_trace) fprintf(fcmps,"%% %-35s \n", "CM_f_add_always_2_1L" );
    CM_f_add_always_2_1L(dest, source1, s, e);
}

        .
        .
        .
```

Figure 3.22: An example of CMPS wrapper function. It is functionally equivalent to its underlying Paris counterpart, except that is has the capability to record run-time trace information as well.

```
      pnorth = cshift(p, 1,-1)
      cu=.5*(p + pnorth)*u
      cv=.5*(p + cshift(p, 2,-1))*v

      call CMPStraceOn
      z = (fsdx*(v-cshift(v,1,-1)) -
     >      fsdy*(u-cshift(u,2,-1)))/
     >      (cshift(pnorth,2,-1)+
     >      cshift(p,2,-1)+p+pnorth)
      call CMPStraceOff

       h = p+.25*(cshift(u,1,1) *
     >      cshift(u,1, 1)+ u*u +
     >      cshift(v,2, 1)*
     >      cshift(v,2, 1)+ v*v)
```

only the fourth statement (with `z` on the left hand side) produces tracing information. The tracing is controlled by a static boolean flag within the cmpslib.a library (see next section).

3.6.3 Link Phase

At this point, we link the modified program object modules with the CMPS library, `libcmps.a`, to create a final executable. This is shown in figure 3.21. This library consists of CMPS functions which are similar to Paris functions, but perform the extra task of recording trace information into the file "cmps.profile" in the current directory. (See run phase in section 3.6.4 for details.)

Besides the individual Paris functions in the CMPS library, `libcmps.a`, there is also a static boolean variable which turns tracing on or off. As described in section 3.6.1 this tracing flag can be explicitly controlled by inserting calls to `CMPStraceOn()` and `CMPS_traceOff()`. This can eliminate the volume of trace output for a large application by concentrating only on relevant parts. An example of a CMPS wrapper function is shown in figure 3.22. Together with the CMPS library, the original Paris library must also be linked in since the underlying Paris functions are still called. There are also other functions in "cmpslib.a", including functions to record the current detailed geometry layout of an array (in CM-Fortran) or a vp-setup (in C*).

3.6.4 Run Phase

Once the CMPS executable is created from the link phase, it is ready to run on the CM-2 using the usual CM commands. This run phase is shown in in figure 3.23. After termination, the trace information (frequency list of Paris functions called) is captured in the file "cmps.profile" residing in the current directory.

The vp-geometry can also be captured at run-time into "cmps.geometry" by explicitly inserting calls to `CMPS_describe_geometry(A)`, where `A` is a parallel variable. This file can be used as input to the analysis phase of CMPS.

This "cmps.profile" text file has all the Paris call information necessary for the analytical model to reproduce the performance information. Its contents are then fed into the analysis phase of CMPS to provide performance estimations. The output itself is rather large; every time a CMPS Paris call is made, a new record is added to the trace file. For this reason, it is recommended that only one or two iterations of complex loops be made to reduce the volume of output. This file is then processed by the CMPS Index Generator (another `awk` program) to produce a compact frequency list of the CMPS functions called. The resulting file, "cmps.profile", is in the format

```
[Paris-function-name]  [frequency]
```

Ideally, one would include all Paris functions in this file; however, one can also use this file to select only a subset of Paris functions to be studied. With over 1,000 Paris functions available, this allows one to run the CMPS system with a handful of the most useful Paris functions used, without having to first write a complete database.

To execute the compile phase of the CMPS system on a given program source (e.g. "myprog.fcm" or "myprog.cs") one simply types

```
% cmpsc -c myprog1.fcm
```

for CM-Fortran programs, or

```
% cmpsf -c myprog1.cs
```

for C* programs. These commands execute the CMPS "compiler" which is a shell script that uses the "cmps.select" file and the underlying CM compilers to produce a CMPS object module as follows (see figure 3.20)

1. The corresponding CM compiler (`cmf` or `cs`) is called to produce an assembly language file.
2. The list in file "cmps.select" is fed into a short `awk` program that generates a `sed` command file, "cmps.sed." The contents of this text file looks like

   ```
   s/CM_f_add_always_2_1L/CMPS_f_add_always_2_1L/
   s/CM_f_add_mult_const_1L/CMPS_f_add_mult_const_1L/
   s/CM_f_divide_3_1L/CMPS__divide_3_1L/
   s/CM_f_move_1L/CMPS__f_move_1L/
   s/CM_f_mult_add_1L/CMPS_f_mult_add_1L/
   s/CM_f_mult_const_add_1L/CMPS_f_mult_const_add_1L/
   ...
   ```

   These `sed` commands change the "CM_" prefix of those Paris functions to the "CMPS_" prefix.
3. This `sed` program is then run on the assembly language input as "`sed -f cmps.sed`" which in turn provides the translation tables to perform the proper name substitution in the source code.
4. this text output is then fed into the front-end assembler to produce the final object module code. This object code is identical to the original object code, except Paris functions appearing in "cmps.select" have been modified to link with their CMPS counterparts.

When using C*, the situation is similar except that internally there is an extra step which converts the output of the C* compiler into assembly code. This is due to the fact that CM-Fortran compiles to VAX assembly directly, whereas C* compiles to C source with Paris calls and then uses the VAX C compiler for final compilation.

Once the CMPS system completes its modification, compilation continues normally to produce a final object module. This object module differs from the original only in that Paris calls have been replaced by their CMPS counterparts. An example of a CMPS wrapper function is given in the next section.

The CMPS systems allows one to turn tracing information on and off, thus allowing one to study certain portions of code. This is quite useful, for example, when one wants to isolate the initialization portion of an application from the main section. The tracing mechanism is on by default when using CMPS. However, the `CMPStraceOn` and `CMPStraceOff` subroutines can select run-time tracing. For example, in the CM-Fortran code fragment

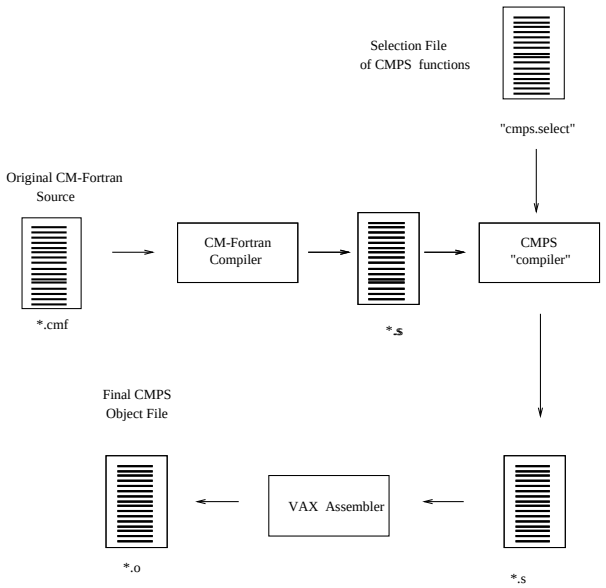


Figure 3.20: Compile phase of the CMPS system. See text for details.

The individual steps (compile, link, run, and analyze) are easy to perform and the whole process can be automated by the use of shell scripts. For illustration purposes, however, we provide detailed examples of the various CMPS components.

3.6.2 Compile Phase

To create a trace of Paris calls used within a particular code, the CMPS system needs to know how the compiler interprets the code example and hence which Paris calls get generated. When using CM-Fortran, the CMPS intercepts the assembly output of the compiler and alters the symbol table information. It alters names of certain Paris functions called; for example, `CM_f_add_2_1L()` becomes `CMPS_f_add_2_1L()`. The CMPS version of the function is a wrapper which calls the real Paris function but also saves state information (current vp geometry, direction of communications, etc.) necessary to reconstruct its performance characteristics from the analytical CM model. (See section 3.6.3 for details of CMPS wrapper functions.) Thus the CMPS system keeps a careful record of each Paris function called and under what conditions. The interaction of this system with the compile phase is shown in figure 3.20.

Only Paris functions listed in the "cmps.select" file are translated to corresponding CMPS functions. This file is simply a text file with one Paris function per line. For example, the file "cmps.select" could have lines such as

```
CM_f_add_always_2_1L
CM_f_add_mult_const_1L
CM_f_divide_3_1L
CM_f_move_1L
CM_f_mult_add_1L
CM_f_mult_const_add_1L
CM_get_from_news_1L
CM_get_from_news_always_1L
...
```

coefficients for floating point operations (such as given in Table 3.1). The application description is essentially a program trace of the specific Paris primitives and vp-geometries used.

The description of the application is a record of the specific Paris primitives called during the execution of a given code segment. This trace is provided by the CMPS system, through its interaction with the high level language CM compilers.

The description of an N-dimensional vp geometry can be described by an 3xN matrix,

$$\begin{matrix} v_1 & o_1 & f_1 \\ v_2 & o_2 & f_2 \\ & \vdots & \\ v_n & o_n & f_n \end{matrix}$$

where, as before, v_i, o_i, and f_i, denote the virtual, on-chip, and off-chip dimensions, respectively.

The vp-geometry can be chosen arbitrarily or can be recorded from an actual program run. The performance of the floating point primitives is dependent only on the virtual processor ratio of the vp-geometry. However, the communication performance depends not only on the specific vp-geometry used, but also on the specific parameters to the Paris communication primitive. This is because one needs to know the specific axis of communication used in order to use the communication model developed in section 3.3.

The description of the architecture consists of the various communication parameters t_E, t_I, and t_M, together with tables of the α and β coefficients for floating point operations (such as given in Table 3.1). This essentially creates a performance "database" which parameterizes the architecture.

Generally, CMPS is used by keeping the architecture description fixed and varying the data layout (vp-geometry information) for a given algorithm or code section. This indicates how the algorithm behaves under different data distributions. Alternately, the CMPS system can be used to study architectural effects, by keeping the application description fixed and varying the architecture parameters.

The components of the CMPS system are grouped in four stages

1. **Compile Phase.** This phase intercepts assembly language output of CM compilers (C* and CM-Fortran) and uses a `sed` script to alter names of Paris functions called. For example, `CM_f_add_2_1L()` becomes `CMPS_f_add_2_1L()`. After this modification of the assembly file, the native assembler is called to complete the final step and produce an object module. Optionally, CMPS primitives may be inserted into the application code directly to control tracing for only a particular portion. See section 3.6.2 for details.

2. **Link Phase.** Here the object file created above is linked with cmpslib.a library. This library has the CMPS wrapper functions which are functionally equivalent to the Paris counterparts except that they have the capability to provide runtime trace information. Thus the name changing step in the compile phase above is essential to avoid symbol conflicts, since CMPS functions generally call their Paris counterpart as well. This is discussed further in section 3.6.3.

3. **Run Phase.** The resulting executable image is executed normally on the Connection Machine. Paris functions that were modified at the compile phase of CMPS provide a trace of themselves in the file "cmps.out", which is created in the current directory. CMPS primitives can be inserted into the application code directly to provide detailed virtual-processor geometries as well, although this can be specified in the analysis phase. See section 3.6.4 for details.

4. **Analysis Phase.** Once the application description has been captured in the output file "cmps.out", various vp-geometries may be tried. The CMPS analyzer is an `awk` program that reads various input files (architecture database, structured profiler, and vp-geometry description) and performs the actual calculation given the analytic expressions for the communication and computation models. See section 3.6.5 for details.

3.6 CMPS

Even with simple numerical expressions, we see that there are several ways in which compilers can parse such constructs. Which particular Paris code gets generated depends on the sophistication and maturity of compilers. Even between different versions of the same compiler, some may perform some different optimizations (e.g. vector triads, or overlapped computation/communication Paris primitives.)

This presents a practical problem to implementing our analytical performance model. We simply do not know how the compiler will interpret numerical expressions. Ideally, we would like to have a performance modeling system that mimics not only the underlying parallel hardware but also the software environment that utilizes such hardware. For example, a programmer would like to know how code performance improves when the compiler optimization is turned on via the -O flag.

Even if the parsing/implementation strategy of the compiler was empirically known, there is another fundamental problem in using the analytical model in practice: no one is going to work through a several thousand line CM-Fortran program parsing complex numerical expressions by hand to catalog the Paris primitives called. A paper-and-pencil analytical performance is clearly not a complete practical solution.

Because of these considerations, we have developed a prototype software environment, the Connection Machine Performance Simulator (CMPS in the sequel), which essentially solves this problem. This software prototype is aimed at demonstrating functionality — it is primarily a vehicle to illustrate that the basic ideas work, rather than being a commercial tool.

The basic function of CMPS is to record the underlying Paris instructions used for communication and computation in a C* or CM-Fortran code segment then utilize previous NEWS communication and floating point performance models to automate the prediction process. One simply runs a normal CM-Fortran program or C* program on the CM-2; by linking to the CMPS library, a trace file, "cmps.profile," is generated at run time. The CMPS system can be thought as a "trace" mechanism which allows one to go back and investigate how various vp geometry configurations affect the performance of a given piece of scientific code.

Although such an environment can become quite sophisticated, we can demonstrate its functionality by utilizing familiar programming tools available on most Unix systems (including the front end host to the CM). We utilize two such tools, (1) `sed` [67], the stream editing tool, and (2) `awk`[1],[2]), a text-processing language, to allow for rapid prototyping. The CMPS performance tool focuses on grid-based computation with NEWS type of communication. A sophisticated system would be much more complex, nevertheless it would employ the same basic strategies. Among the characteristics of the CMPS system are

- transparent integration with the C* and CM-Fortran compilers
- requires no source code modification
- uses the CM analytical performance model as database to yield accurate analysis
- works with arbitrary NEWS vp-geometries
- can be used as an analytical tool for finding optimal data mappings

One of the current limitations of the prototype system is that it works with code that uses only one fixed vp-geometry throughout its execution. All of the code examples discussed in this work and most matrix/operations fall under this category. This limitation can be removed by making the CMPS even more complex; however, the current system illustrates the basic concepts and presents underlying design and strategy.

3.6.1 Overview of Functionality

The CMPS model consists of three parts: (1) a description of the architecture, (2) a description of the application, and (3) a description of the data decomposition for N-d grids (vp-geometries). The architecture description consists of the communication parameters t_E, t_I, and t_M, together with tables of the α and β

which performs nearly all possible Paris optimizations! (For some reason, the last `CM_f_news_add` operation is not implemented in the `always` version, but this is relatively insignificant in the overall performance of the expression.)

```
pushl   $32
calls   $1, _CM_allocate_stack_field
movl    r0, r6
pushl   $8
pushl   $23
cvtfd   $0F4.000000000000e+00, -(sp)
pushl   r7
pushl   r6
calls   $6, _CM_f_multiply_const_always_3_1L
pushl   $8
pushl   $23
pushl   r6
calls   $3, _CM_f_negate_1_1L
pushl   $8
pushl   $23
pushl   $1
pushl   $1
pushl   r7
pushl   r6
calls   $6, _CM_f_news_add_always_2_1L
pushl   $8
pushl   $23
pushl   $0
pushl   $1
pushl   r7
pushl   r6
calls   $6, _CM_f_news_add_always_2_1L
pushl   $8
pushl   $23
pushl   $0
pushl   $2
pushl   r7
pushl   r6
calls   $6, _CM_f_news_add_always_2_1L
```

Figure 3.19: A section of VAX assembly code generated by the CM-Fortran v. 1.0 compiler showing the Paris primitives being called.

```
CM_f_multiply_const
CM_get_from_news
CM_f_add
CM_get_from_news
CM_f_add
CM_get_from_news
CM_f_add
```

As another possibility, the compiler could recognize overlapped communication/computation Paris primitives and generate instead

```
CM_f_multiply_const
CM_f_news_add
CM_f_news_add
CM_f_news_add
CM_f_news_add
```

Alternately, the compiler could recognize NEWS/triads operations and realize that the three operations `-4.0*A + chsift(A,1-1)` can be done by a single Paris operation,

```
CM_f_news_mult_const_add
CM_f_news_add
CM_f_news_add
CM_f_news_add
```

Of course, which specific decision the compiler takes will affect the performance (and hence the performance model.) As an exercise, we described performance models for all three possibilities and then actually compared them to the above compiled to determine which of the three combinations was actually being generated. (Remember, that our analytical model is precise enough to distinguish between the different cases.)

To our surprise, none of our predictions were correct! A closer look at the CM-Fortran compiler (version 0.6) output, shown in fig 3.19, shows the that actual Paris code generated looks like

```
CM_f_multiply_const_always
CM_f_negate_1_1L
CM_f_news_add
CM_f_news_add_always
CM_f_news_add_always
CM_f_news_add_always
```

Here the compiler is doing something very smart (using the faster `always`) versions of Paris primitives) and something rather foolish (multiplying `A` in parallel by 4.0, then multiplying the result in parallel by -1.0 !) Furthermore, the subexpression `-4.0*A + cshift(A,1,-1)` can parsed into a single NEWS/triad Paris function, but this is not being recognized. If we make simple modification to the CM-Fortran code by surrounding the -4.0 constant by parenthesis,

```
      A = (-4.0)* A  + cshift(A,1,-1) +
     >      cshift(A,1,1) + cshift(A,2,1) +
     >      cshift(A,2,-1)
```

the compiler then generates

```
CM_f_news_mult_const_add
CM_f_news_add_always
CM_f_news_add_always
CM_f_news_add
```

3.5 Putting the Models to Work

Now that we have analytical performance models for floating point computation and grid communication for the CM-2, the next natural step is to see how such models can be used on basic numerical algorithms and higher level scientific applications.

This section begins with the study of simple relaxation kernels encountered in for several solution techniques for PDEs and continues with a two-dimensional atmospheric application often used in benchmarking supercomputer architectures – the Shallow Water Equations.

There is a fundamental problem, however, in applying these analytical models to real problems: the algorithms and applications one is interested in modeling are usually in high level languages such as Fortran or C. It is completely unreasonable to expect that large CM-2 applications be rewritten into Paris in order to apply our performance model.

We present a prototype software environment, the Connection Machine Performance Simulator (CMPS), that essentially solves this problem. We should stress that the objective of this work is not to develop a commercial-level tool, but rather to develop a prototype vehicle to demonstrate that our ideas basically work. The CMPS systems works in integrated environment with the high-level CM-Fortran and C* compilers. It works with completely *unmodified* CM-Fortran and C* codes. Although the system is tightly coupled (for example, compiling with an optimization flags turned on, yields different predictions,) it is still flexible in the sense it can utilize new releases of the compilers as soon as they become available. The results in this paper were done under CM-Fortran VAX Fortran version 0.6.

Finally, we use the CMPS system to study the performance of the Shallow Water Equations for various grid decompositions on the CM-2. We illustrate the various problem sizes and decompositions, the predictions generated by the automated CMPS system are within 10% error.

3.5.1 Relaxation Example

We begin by studying simple relaxation processes for two dimensional Poisson equations, typical of processes occurring in many applications codes involving either elliptic PDE solution or time evolution equations. The code kernels we will describe are essentially those used in relaxation, multigrid and conjugate gradient solution of the Poisson equation.

Although there are significant performance differences among relaxation methods, they nevertheless utilize nearly identical computational operations. Since the Poisson equation has constant coefficients, the ratio of computational work per grid point to memory traffic is severe, thus illustrating the task granularity synchronization characteristics of parallel architectures.

Consider the simple two-dimensional 5-point stencil

$$b(j,i) = -4.0 * a(j,i) + a(j,i-1) + a(j,i+1) + a(j-1,i) + a(j+1,i)$$

defined at each grid point (i,j). For those points on the border, we will define periodic boundary conditions; this captures the essential flavor of the finite difference computations in the Shallow Water Equations applications (see section 3.7.)

This expression can be coded in the Fortran 90 array syntax as

```
      B = -4.0* A  + cshift(A,1,-1) +
>       cshift(A,1,1) + cshift(A,2,1) +
>       cshift(A,2,-1)
```

Depending on how the compiler parses such as expression, we could have several possibilities for the Paris code generated. For example, the CM-Fortran compiler could decompose the expression into single vector expressions (no triads) and separate communication and computation (no overlap). We can illustrate this by listing the respective Paris primitives called:

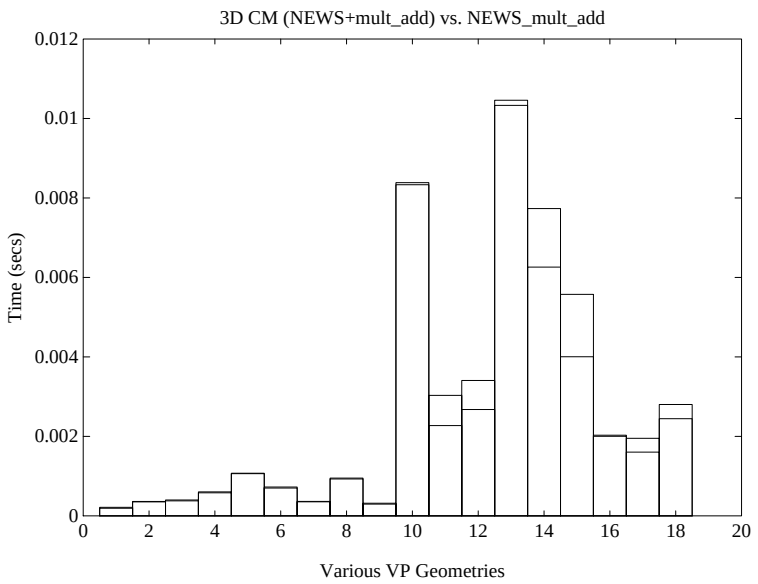


Figure 3.18: Performance effects of overlapping computation with communication on the CM-2. Some vp-geometries (11,12,14, 15, 17, 18) show performance improvement. Each bar is split into two regions: (1) the cost of performing overlapped communication/computation instructions (lower bar), and (2) the extra cost by separating these two aspects (top bar). See text for details.

Table 3.3: Model for CM Triad Floating Point Operations

Operation	Paris Function	α (μsecs)	β (μsecs/vp)
$\vec{y} = \vec{a}(\vec{x} + \lambda)$	CM:f-add-const-mult-1L	13.98	32.31
$\vec{y} = \omega(\vec{x} + \lambda)$	CM:f-add-const-mult-const-1L	14.67	32.02
$\vec{y} = \vec{a}(\vec{x} + \vec{b})$	CM:f-add-mult-1L	12.23	37.42
$\vec{y} = \omega(\vec{x} + \vec{b})$	CM:f-add-mult-const-1L	31.19	37.38
$\vec{y} = \vec{a}\vec{x} + \vec{b}$	CM:f-mult-add-1L	12.08	41.70
$\vec{y} = \vec{a}\vec{x} + \lambda$	CM:f-mult-add-const-1L	28.54	41.70
$\vec{y} = \omega\vec{x} + \vec{b}$	CM:f-mult-const-add-1L	13.23	36.62
$\vec{y} = \omega\vec{x} + \lambda$	CM:f-mult-const-add-const-1L	13.70	36.34
$\vec{y} = \omega\vec{x} - \vec{b}$	CM:f-mult-sub-1L	12.43	41.70
$\vec{y} = \omega(\vec{x} - \vec{b})$	CM:f-sub-mult-1L	11.85	37.43

```
    D(i,j) = A(i,j) * B(i+1,j) + C(i,j)
endfor
```

The `23,8` arguments denote single precision operands, `CM_upwards` distinguishes between +1 or -1 offset, and the `0` denotes along dimension zero (first dimension.) The above expression is similar to

```
D= A * cshift(B, 1, 1) + C
```

using Fortran 90 syntax.

To see how well the communication is hidden in these operations, we show the comparison results between using an optimized `NEWS_mult_add` operation and calling the two instructions separately. We chose to study the operations on several three dimensional arrays; the results are shown in figure 3.18, where vp geometry #1, #2, and #3 correspond to the three dimensions of the first 3D array, geometries #4, #5, and #6, correspond to the three dimensions of the second 3D array, and so on.

Here we notice that for each 3D array, there is a considerable performance improvement when overlapping computation with communication in certain situations. Upon studying the situation closely, we noticed the pattern that only the first first dimension showed no performance improvement when using the optimized NEWS/triads. (This is more noticeable, for example, in vp geometries #10, #13, and #16, which are all first dimensions of different examples.)

The reason was found to be that the Sprint node is always assigned the first dimension. Thus, when traversing *across* it one can communicate directly from Sprint node to Sprint node, bypassing redundant transformations. In a sense, the Sprint transposition step becomes effectively pipelined.

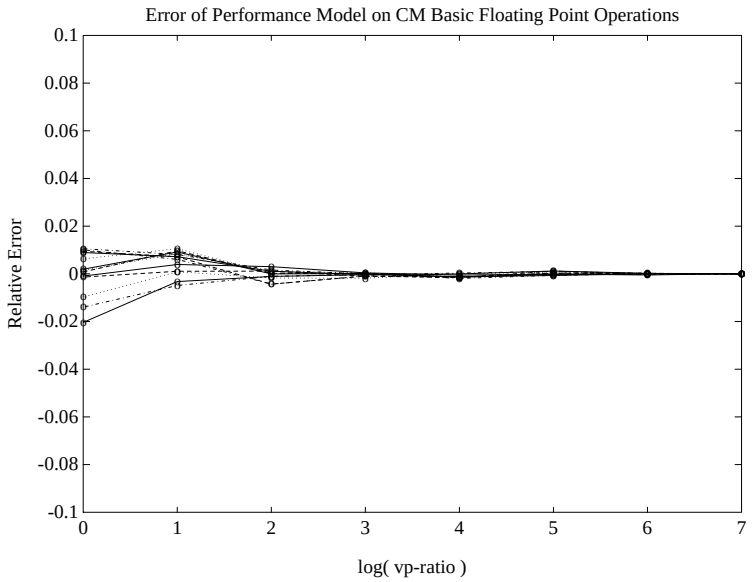


Figure 3.17: Error of performance model diminishes at higher vp ratios.

Table 3.2: Model for CM Transcendental Floating Point Operations

Operation	Paris Function	α (μsecs)	β (μsecs/vp)
$\vec{x} = \cos \vec{x}$	CM:f-cos-1-1L	23.15	233.57
$\vec{y} = \cos \vec{x}$	CM:f-cos-2-1L	22.89	233.57
$\vec{x} = \exp \vec{x}$	CM:f-exp-1-1L	38.46	243.76
$\vec{y} = \exp \vec{y}$	CM:f-exp-2-1L	37.54	243.78
$\vec{x} = \ln \vec{x}$	CM:f-ln-1-1L	15.84	293.90
$\vec{x} = \sin \vec{x}$	CM:f-sin-1-1L	21.37	232.27
$\vec{y} = \sin \vec{y}$	CM:f-sin-2-1L	12.31	236.96
$\vec{x} = \sqrt{\vec{x}}$	CM:f-sqrt-1-1L	1.63	188.27
$\vec{x} = \tan \vec{x}$	CM:f-tan-1-1L	20.77	276.46
$\vec{y} = \tan \vec{y}$	CM:f-tan-2-1L	12.27	281.18

Table 3.1: Model for CM Basic Floating Point Operations

Operation	Paris Function	α (μsecs)	β (μsecs/vp)
$\vec{x} = \vec{x} + \vec{a}$	CM:f-add-2-1L	10.60	33.16
$\vec{x} = \vec{a} + \vec{b}$	CM:f-add-3-1L	10.78	33.47
$\vec{x} = \lambda + \vec{x}$	CM:f-add-constant-2-1L	12.09	28.35
$\vec{x} = \lambda + \vec{a}$	CM:f-add-constant-3-1L	12.03	28.38
$\vec{x} = \vec{x}/\vec{a}$	CM:f-divide-2-1L	1.57	88.34
$\vec{x} = \vec{a}/\vec{b}$	CM:f-divide-3-1L	13.92	139.95
$\vec{x} = \vec{x}/\lambda$	CM:f-divide-constant-2-1L	19.00	88.28
$\vec{x} = \vec{a} * \vec{b}$	CM:f-multiply-3-1L	10.93	29.17
$\vec{x} = \vec{x} * \lambda$	CM:f-multiply-constant-2-1L	12.04	24.05
$\vec{x} = \vec{a} * \lambda$	CM:f-multiply-constant-3-1L	12.17	24.06
$\vec{x} = \vec{a} - \vec{b}$	CM:f-subtract-3-1L	10.77	33.47
$\vec{x} = \vec{x} - \lambda$	CM:f-subtract-constant-2-1L	12.08	28.35
$\vec{x} = \vec{a} - \lambda$	CM:f-subtract-constant-3-1L	12.09	28.38

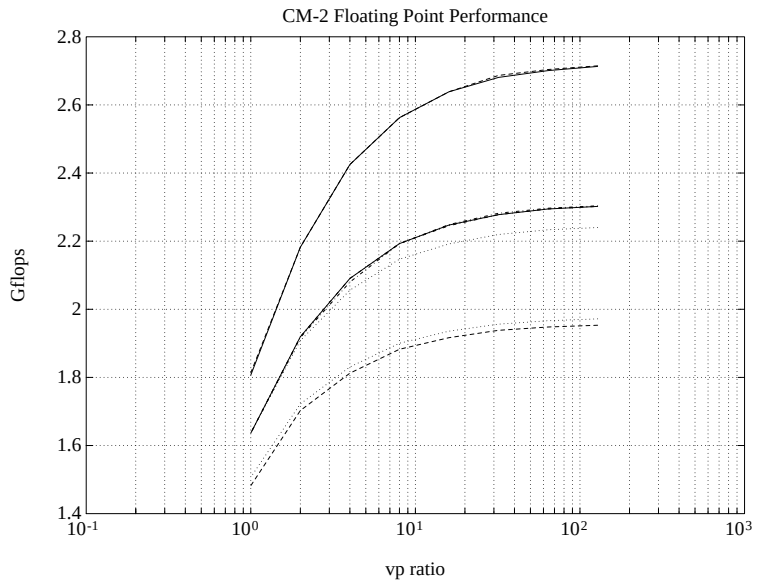


Figure 3.16: Performance of various basic floating point operations falls between 1.5 and 3.0 GFLOPS.

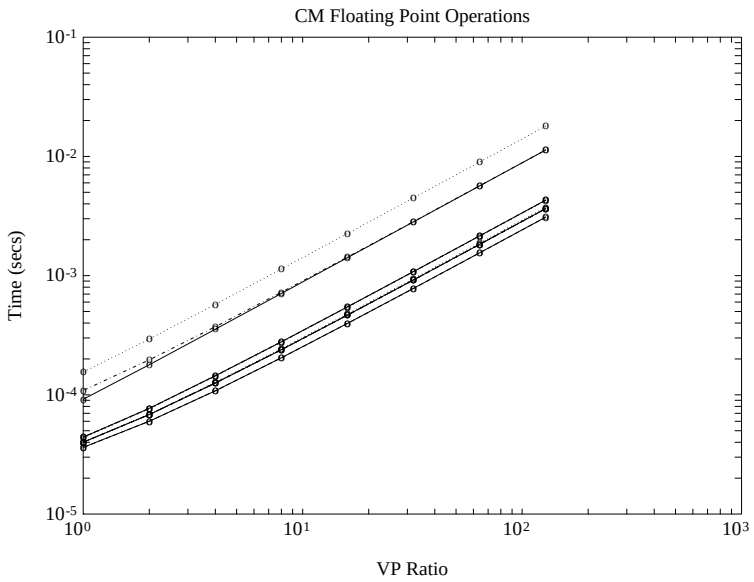


Figure 3.15: Performance of various basic floating point operations on the CM-2 show a linear relationship the vp ratio.)

Running several experiments for the basic floating point operations, the results in fig. 3.15 show that the performance is indeed linear with respect to the vp ratio. Some of the specific operations studied are listed in figure 3.1. Here the first column specifies the vector operations such as $\vec{x} = \lambda + \vec{a}$, where λ denotes a scalar. The second column denotes the PARIS instruciton name, the third column denotes the startup costs, α, (in μsecs) and the fourth column denotes the pipelined costs, β, (in μsecs/vp.) The ratios between α and β are not exactly 3/2 since the stages do not each take the same amount of time. For example, in the case of vector division, $\vec{x} = \vec{a}/vb$, the ratio β/α is roughly 10:1; this is due to the fact the floating point stages take much longer than the Sprint stages. As an indication of computational speed of these operations, fig. 3.16 illustrates that most of the vector add and multiply operations operate between 1.5 to 2.5 GFLOP range; the vector divisions require much more work and operate in the 0.5 GFLOP range.

Figure 3.17 denotes the error distribution the model of these basic vector operations. The relative error of the model is never greater than 2%, and for vp ratios of two or greater the relative error is essentially neglible. Figures 3.2 and 3.3 give the α and β parameters for vector transcendental functions and triads functions, respectively. Both groups of functions are modeled to 2% error. The triad vector operations combine multiplication and an addition in a single function (the Weitek 3132 vector unit has an add and multiply pipe that can work in unison) thus effectively doubling the GFLOP performance rate in many cases.

3.4.4 Overlapping Communication with Computation

It is possible to overlap NEWS communication while computing on the vector units. This allows one to reduce communication costs by hiding the latency behind a binary or triad vector operation. This useful in many numerical applications, such as finite difference methods for solving partial differential equations. For example, let `a, b, c,` and `d` be single precision arrays of size $N \times M$. The PARIS primitve

```
CM_f_news_mult_add_4_1L(d, a, b, c, 0, CM_upward, 23,8)
```

performs the following psuedocode

```
for (i=1 to N, j=1 to M)
```

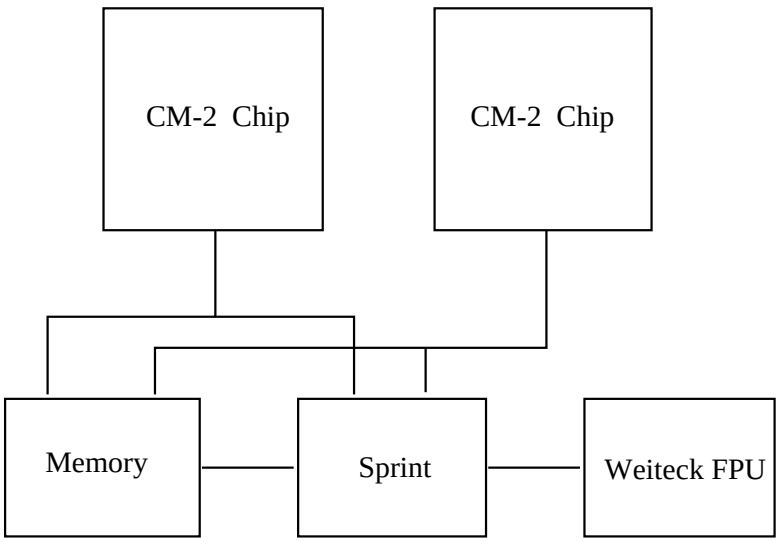


Figure 3.13: CM-2 Floating point node architecture.

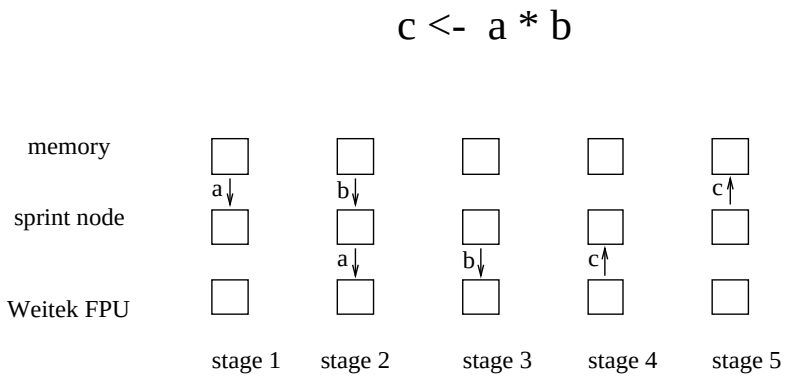


Figure 3.14: Pipelining effects between memory, Sprint, and vector chips.

3. the second vector operand is forwarded to the vector unit. Here the vector operation is performed and the vector contains the vector result

4. the result of the vector operation is forwarded back to the Sprint, which transposes them

5. finally, the vector result is sent back to the main processor memory.

Notice that the pipelining effect is related to vp ratio; if this ratio is n, then this process is pipelined to require only $3n+2$ stages rather than $5n$. This is depicted in figure 3.14, where the stages of vector operation $c = a + b$ are shown and the asterisks denote that a new vector value has been produced. (Each vector is stripmined into components of length 32; the subscripts in this figure refer to these components.) Since this model closely resembles traditional pipelined vector processors, it is reasonable to expect that the floating point latency will consist of a basic startup cost and an amortized costs in one "pipelined" mode. Thus, we exepct the model to have a linear relationship between vp ratio (which plays the role of vector length in traditional pipelined processors) and execution time.

3.4.3 Observed Measurements

3.4 Modeling Floating Point Operations

Here we present a performance model for the basic floating point kernel operations which constitute the computational core of all numerical applications: basic operations (+,*,-,/), transcendental library functions (`sin, exp, sqrt`), and triad vector operations (`ax + y`). The model predicts these operations to within 2% error over all sample CM-2 geometries. Furthermore, we present a model of the PARIS primitives that effectively overlapp communication and computation for improved performance.

3.4.1 CM-2 Floating Point Architecture

To understand the floating point performance of the CM-2, we need to view its underlying architecture and understand how its constituent parts work together. This architecture is rather unique in that it attempts to integrate two rather mismatched architectures. It essentially couples fine-grained bit-serial processors (which can produce *one bit* per cycle) with powerful pipelined vector processors (that can produce a 32-bit result every cycle.)

This architectural curiosity arose mainly from the CM's history: the motivation for original CM-1 architecture came from artificial intelligence applications [43] where computation is dominated by fine-grained, bit-serial processing (e.g. neural networks, scene recognition, lists, boolean tables, etc.) Consequently, the original design did not include floating point hardware and all such computations were carried out in software by the bit-serial processors. When the potential of the CM was realized for scientific computing, the original CM design was augmented with pipelined vector processors (Weitek 3132s) to significantly increase the CM's floating point computational power.

This addition of numeric coprocessors, however, posed a fundamental architectural challenge due to the tremendous differences in processor bandwidth. In one clock cycle, for example, a bit-serial processor can send out one bit of a single-precision (32-bit) floating point number; n the same clock cycle, however, the Weitek coprocessors can consume a 32-bit argument. Simply attaching a Weitek 3132 to each bit-serial processor, therefore, is not an effective solution.

Since the effective bandwidth ratio between the two processors 32:1, it seems reasonable to attach 32 bit-serial processors to each vector unit to balance bandwidth mismatch. Since it requires at least 32 stages for each CM processor to send its argument to the vector units, the CM architects arranged to have 32 physical processors (2 CM chips) attached to each vector unit. In this manner, the Weitek 3132 receives 32 single precision (32-bit) arguments which amortize to one single precision argument each step.

Figure 3.13 illustrates a typical CM-2 floating point node consisting of two CM chips (32 processors) coupled to a common memory area which connects to a single Wietek 3132 and its interface chips. The "Floating Point and Memory Interface" chip residing between the processor memory and floating point unit is commonly referred to the Sprint chip, and is responsible for "transposing" the arguments from main memory into a suitable format for the Wietek chip. The "transposition" is necessary, since the 32-bit argument produced each stage by the 32 processors does not correspond to any single floating point number — each bit comes from a different processor.

3.4.2 Basic Model for Floating Point

The basic model for floating point computation on the CM-2 is based on the movement of data from the processor memory (256Kx44 RAM in figure 3.13) to the Sprint chip, and onto the vector units. For example, consider performing a parallel **add**, or **multiply** with two vector operands. Since the data must pass through and back the Sprint interface chip, the operation can be broken down into five stages [94]:

1. the first 32 data values (one from each processor) of the first operand are moved from main memory into the Sprint node

2. the Sprint nodes transposes the first vector operand and forwards it to the Weitek vector unit while simultaneously, the second vector operand is moved from main memory to the Sprint node

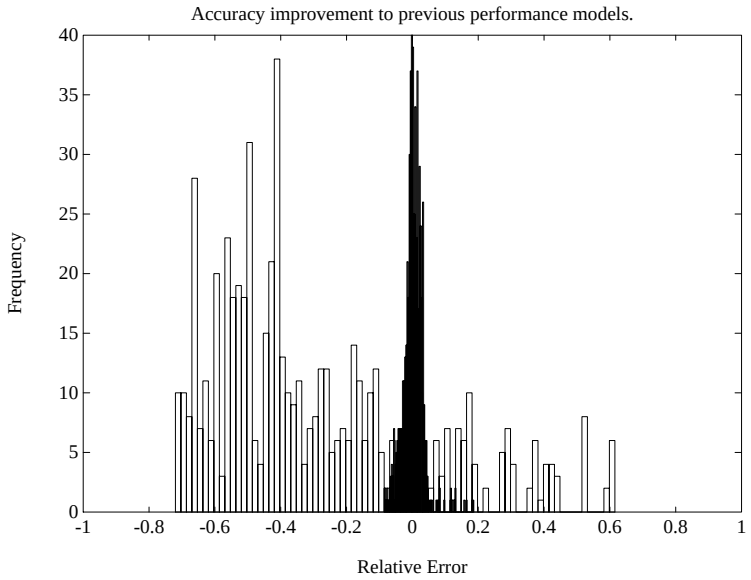


Figure 3.12: A comparison to previous CM-2 communication models. The error distribution of predictions has been reduced from a +/-60% span to a +/- 5% span.

3.3.3 Related Work

Chen, Choo & Jacquemin [16] discuss similar communication models for the CM-2 . They provide an example of how accurate communication models can help optimizing compilers rearrange distributed data structures to improve performance. Because dependencies among distributed arrays often translate into processor communication, their goal is to determine the allocation of arrays that minimizes the total communication cost. They have developed techniques where given the parallel arrays and their dependencies, linear transformations are applied to the domains of the arrays to produce the best alignment that reduces the total communication costs.

An earlier communication model for the CM-2 was proposed by Levit [55] which focused on with sending fixed-sized messages over power-of-two distances. Our model is an extension to this work. By modeling virtual processor communications (t_M) and geometries which bypass physical communications ($o_i = 1$), we have been able to significantly improve the prediction accuracy .

Now that we have an accurate grid communication model for the CM-2, our next step is to formulate an equally accurate performance model for the fundamental floating point computations. Together, these two models are then adequate to describe the computation and communication aspects of grid-based numerical applications. The next section presents a computation model for the CM-2.

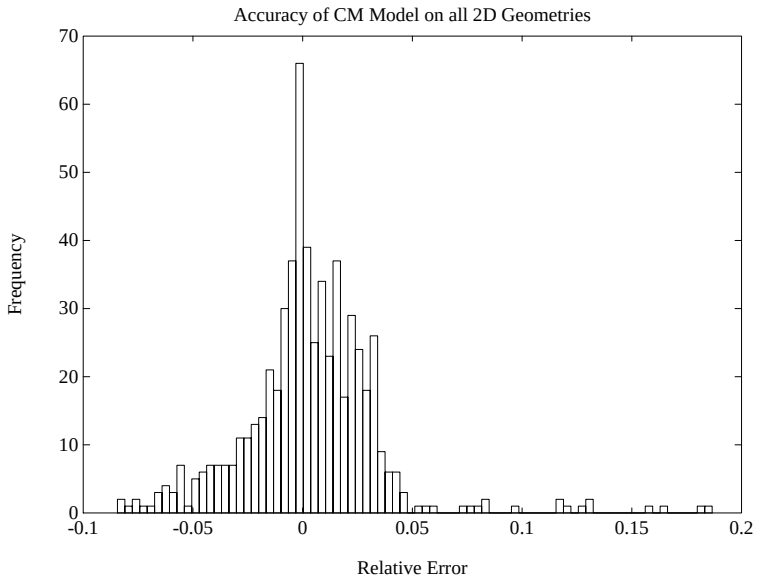


Figure 3.10: Error distribution of CM NEWS model on CM measurements of various 2D geometries.

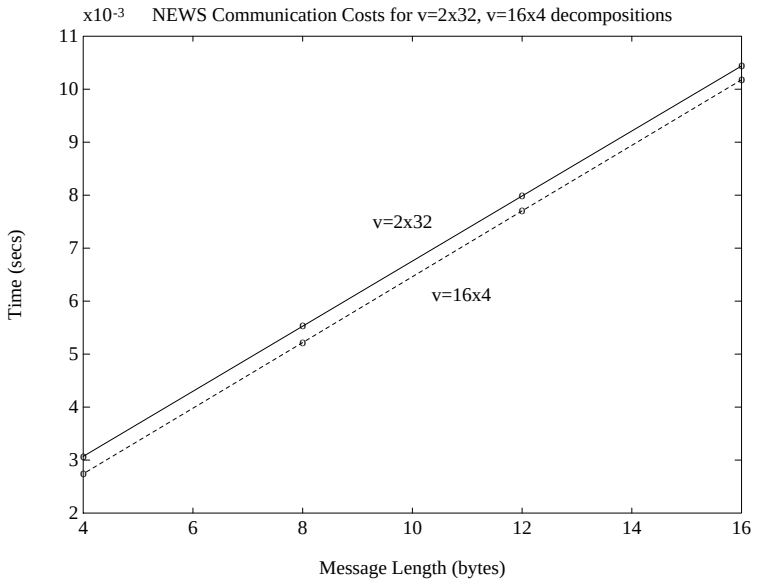


Figure 3.11: NEWS communication latency for two ways of decomposing the same problem.

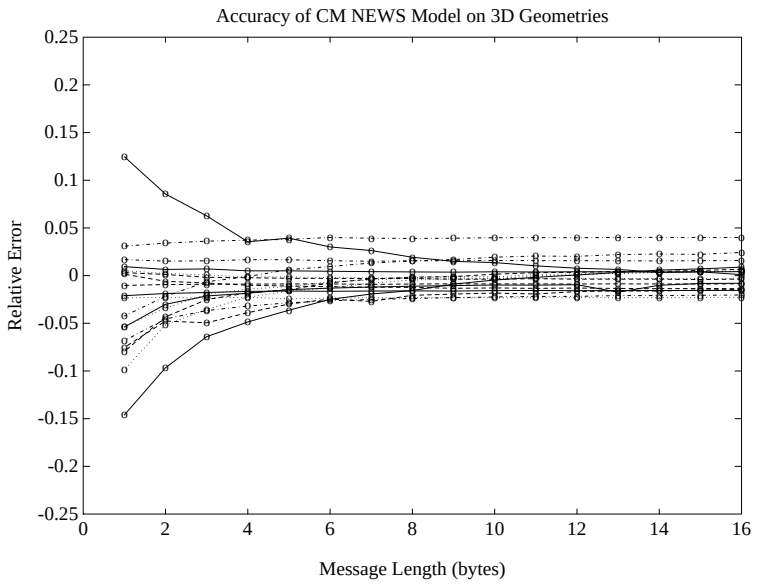


Figure 3.9: Relative error of CM NEWS model on CM measurements of various 1D and 3D geometries.

Most predicted values fall within 5 % relative error of the actual measurement. When this model is applied to the 2D data (which was not included in the least-squares fit) all predicted values fall within 6 % relative error of the measurements. Figure 3.10 depicts the error distribution of the model on the 2D data.

Altogether, the model has been verified on 33 1D, 2D, and 3D geometries. This comprises over 500 geometry/message-length combinations. Each of data point (circle) in these figures represents the average of five measurements (all experimental measurements agreed to within 1% relative error of each other), totaling over 2,500 CM measurements.

Note that since $t_E < t_M < t_I$, finding the optimal processor geometry by simply minimizing the number of off-chip communications does not always work. For example, consider mapping a 4096x4096 grid onto a CM-2 with 16K physical processors. Furthermore, assume that the physics of the posed problem involves asymmetric communication: one axis, say the x-axis, has 5 times more communication than the other. (These characteristics are not unusual, in fact this example was based on an actual application.) Given that the communication is dominated by one axis, we would like to map more of this axis onto each physical (and hence, virtual) processor. In this way, off-chip communications turn into on-chip communications, and on-chip communications become less expensive virtual (memory-to-memory) transfers. Decomposing the grid as

$$\vec{f} = \{128, 128\} \tag{3.10}$$
$$\vec{o} = \{16, 1\} \tag{3.11}$$
$$\vec{v} = \{2, 32\} \tag{3.12}$$

and given that the first dimension has 5 times more communication than the second, we have (5 x 32 + 32) = 192 off-chip communications per iteration. Consequently, mapping the same problem as

$$\vec{f} = \{16, 1024\} \tag{3.13}$$
$$\vec{o} = \{16, 1\} \tag{3.14}$$
$$\vec{v} = \{16, 4\} \tag{3.15}$$

results in generating (5 x 4 + 256) = 284 off-chip communications. How do these compare? It is the second mapping that has the lower communication latencies. This is shown in figure 3.11.

the number of on-chip communications is given by

$$C_M = (v_i - 1) \prod_{j \neq i}^{N} v_i,$$

and the number of off-chip communications is given by

$$C_E = \begin{cases} \prod_{j \neq i}^{N} o_i v_i & \text{if } f_i > 1 \\ 0 & \text{otherwise} \end{cases}$$

The total communication latency for a NEWS communication is then the dot product of the coefficient vector $\{C_I, C_M, C_E\}$ with $\{t_I, t_M, t_E\}$.

Notice that the performance was augmented to include a zero value for C_E coefficient if the off-chip dimension o_i is 1. This is due to the cyclic characteristic of the Binary-Reflected-Gray-Coding that maps CM processor chips onto the hypercube topology: the "neighboring" chip in that direction is the same chip, hence no physical (off-chip) communication takes place. The CM hardware detects this special case and bypasses the forwarding of the message into the hypercube network. Hence that communication is really on-chip and is accounted for in t_I. This important point was overlooked in earlier models [55] and consequently caused performance estimates to be off by a factor of 4 or more (see page 58).

Performing a least squares fit to the data for 1D and 3D geometries, we determine single communication latencies of the `CM_get_from_news_1L()` to be

$$t_M(n) = (2.4 + 5.6n)\ \mu\text{secs} \quad (3.4)$$
$$t_I(n) = (20.5 + 10.4n)\ \mu\text{secs} \quad (3.5)$$
$$t_E(n) = (0.0 + 5.5n)\ \mu\text{secs}. \quad (3.6)$$

When all virtual processors are participating, one can use a slightly faster version, (`CM_get_from_news_always_n1L()`), which does not test the processor context flag [1] . These similar routines are roughly 15% percent faster. The α and β parameters for the `_always_` version are

$$t_M(n) = (2.6 + 4.6n)\ \mu\text{secs} \quad (3.7)$$
$$t_I(n) = (24.8 + 8.8n)\ \mu\text{secs} \quad (3.8)$$
$$t_E(n) = (0.0 + 5.5n)\ \mu\text{secs}. \quad (3.9)$$

3.3.1 Extending the Model to Power-of-Two Distances

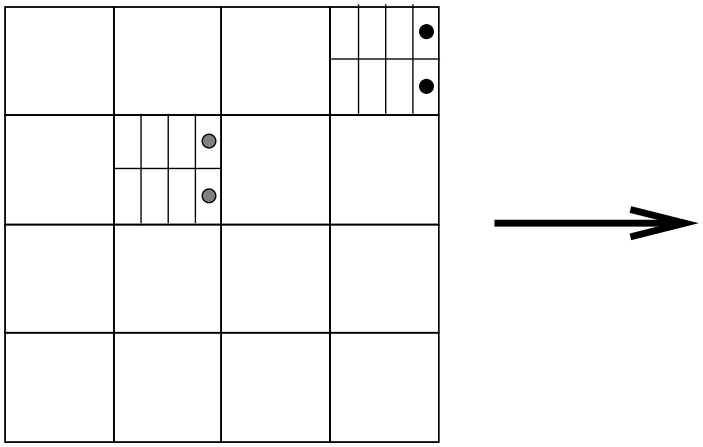
Our model so far has focused on variable-length messages sizes over unit distances (as encountered in finite difference computations or SOR methods.) By a similar analysis to Levit[55], we can extend it to communicate over arbitrary power-of-two distance lengths, d, as

$$T(n,d) = \begin{cases} t_I(n) \prod_{j \neq i} v_i + t_M(n)(v_i - 1) \prod_{j \neq i} v_i + & \\ \quad t_E(n) \prod_{j \neq i} o_i v_i & \text{if } d < o_i v_i \\ 16 t_E(n) \prod_i v_i & \text{if } d = o_i v_i \\ 32 t_E(n) \prod_i v_i & \text{if } d > o_i v_i \end{cases}$$

3.3.2 Accuracy of NEWS Communication Model

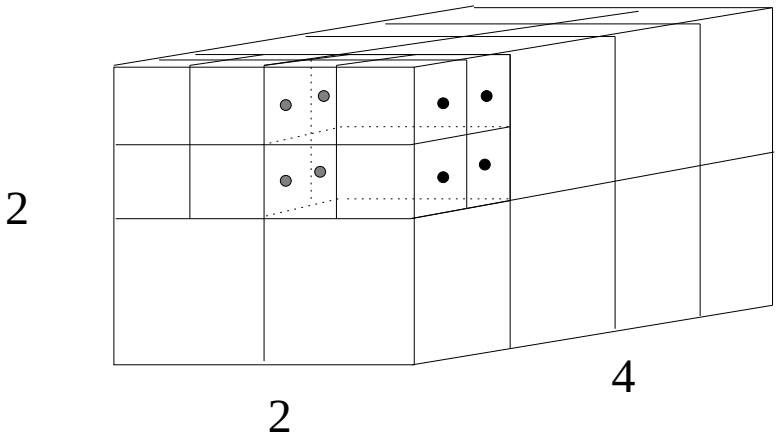
Figure 3.9 provides an error distribution for the accuracy of the NEWS communication model. Here we measure the relative error of observed versus predicted values for 345 points over 23 different geometries.

[1] Because of its SIMD architecture, each processor in the CM has a context flag that dictates whether or not that processor will participate in the current instruction.



6tm + 2ti + 8te

Figure 3.7: An example of a 2D configuration where the 16 on-chip processors are arranged as a 4x4 array, each simulating 2x4 virtual processors.



4tm + 4ti + 32te

Figure 3.8: An example of a 3D configuration where the 16 on-chip processors are arranged as a 2x2x4 array, each simulating 2x2x2 virtual processors.

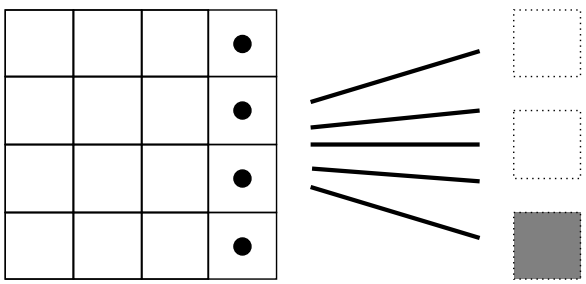


Figure 3.6: An example of a 2D configuration where the 16 on-chip processors are arranged as a 4x4 array, each simulating 1x1 virtual processors.

processors on the interior of the chip (the 12 processors shown in white) can communicate in parallel, since there is no contention for resources. However, since we are on an SIMD machine, we can either be executing on-chip communications or off-chip communications, but not both. If we let $t_i(n)$ denote the time required for *all* interior processors on a physical chip to communicate a value to their on-chip neighbors, then the total time required for our example is

$$T(n) = t_i(n) + 4t_E(n).$$

Consider a more typical example, shown in fig. 3.7, where again the 16 physical processors are arranged in a 4x4 array ($\vec{o} = \{4, 4\}$), but each physical processor simulates a 2x4 array of virtual processors, ($\vec{v} = \{2, 4\}$.) Much of the communication will be inexpensive, since a good portion of it remains within a physical processor. Let us denote the time required to communicate between virtual processors (on the same physical processor) as t_M. Then we have the following situation. Each physical processor on the interior of the chip will have 6 of its virtual processors send virtual communications (inexpensive memory transfers), while the two boundary processors (shown dotted in fig. 3.7) will undergo an on-chip communication. Thus the time required for the interior processors to communicate will be $6t_M + 2t_I$. There are eight virtual processors residing on the edge which will perform a physical communication, and hence the total communication latency is given by $6t_M + 2t_I + 8t_E$.

Finally, consider the 3D example shown in fig. 3.8. The 16 on-chip processors are arranged in a 2x2x4 array, with each physical processor simulating 8 virtual processors arranged in an 2x2x2 array. Again assume that we wish to have every processor send a value to its East neighbor. Each physical processor, then, will send 4 values to neighboring virtual processors (virtual communications) and 4 values to processors on the same CM-2 chip. The remaining 32 virtual processors on the right-side of the box must be sequentially transmitted over the same hypercube edge to the destination CM-2 chip. Hence the total time required for this communication is given by $4t_M + 4t_I + 32t_E$.

From these examples, one observes that by studying the boundary effects among the three communication hierarchies one can obtain the proper coefficients

$$\vec{C} \equiv \{C_M, C_I, C_E\}$$

that dedicate the number of various communication performed. Given a CM geometry specification consisting of $G = \{\vec{f}, \vec{c}, \vec{v}\}$ specifying off-chip, on-chip, and virtual processor mapping, we can then determine the number of each type of communications performed along a given dimension by the following formulas.

The number of memory-to-memory communications is given by

$$C_I = \prod_{j \neq i}^{N} v_i,$$

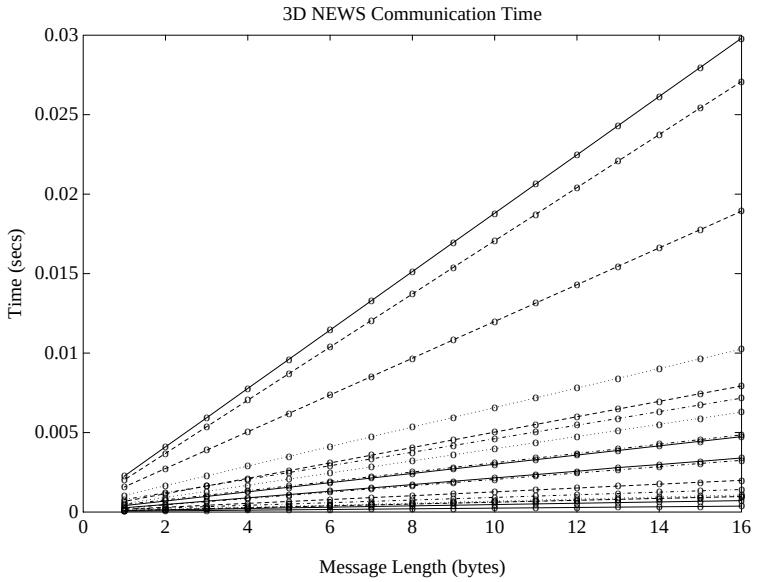


Figure 3.5: Observed NEWS communication latency for various 3D geometries illustrate the communication costs are linear in relation to the message length.

virtual communication: The communication between two virtual processors on the same physical processor. This is really a memory transfer within each physical processor, and requires no explicit communication.

on-chip communication: The communication between two physical processors on the same processor chip. This occurs over dedicated hardware on each chip and does not require use of the off-chip hypercube network.

off-chip communication: The communication between two processors on different processor chips. This requires the off-chip hypercube network.

We denote the respective communication latencies as

t_M virtual communication (Memory-to-memory)

t_I on-chip communication (Internal)

t_E off-chip communication (External)

Consider now a vp geometry in which the 16 on-chip processors have been configured as 4x4 arrays, ($\vec{\sigma} = \{4, 4\}$); each physical processor simulates one virtual processor, ($\vec{v} = \{1, 1\}$.) Suppose we are performing a second-order centered difference calculation where each processor is required to communicate with its four nearest neighbors to compute $\partial/\partial x$ and $\partial/\partial y$. For simplicity, let us focus on obtaining a neighboring along one dimension although the discussion is analogous to all directions and dimensions. In this case we have the situation depicted in figure 3.6, where some of the hypercube edges are also drawn.

The virtual processors on the right edge (shown shaded in figure 3.6) must be sent out into the hypercube network. Since N-dimensional grids can be mapped into a hypercube so that neighboring grid points get mapped into neighboring nodes on the hypercube, the destination node lies directly at the end of the one of the hypercube edges. If we denote the time to send a message of n bytes over one of these edges as $t_E(n)$, then the time required to send the four values on the chip boundary is simply $4t_E(n)$. The remaining

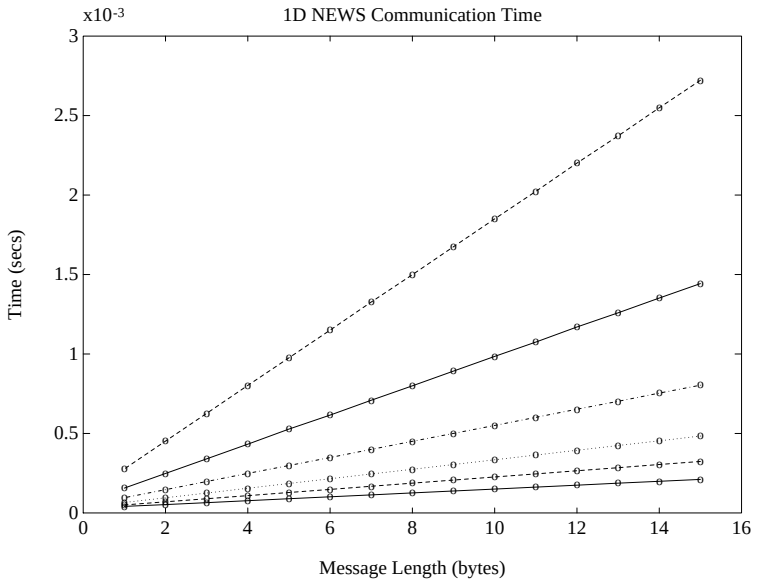


Figure 3.3: Observed NEWS communication latency for various 1D geometries illustrate the communication costs are linear in relation to the message length.

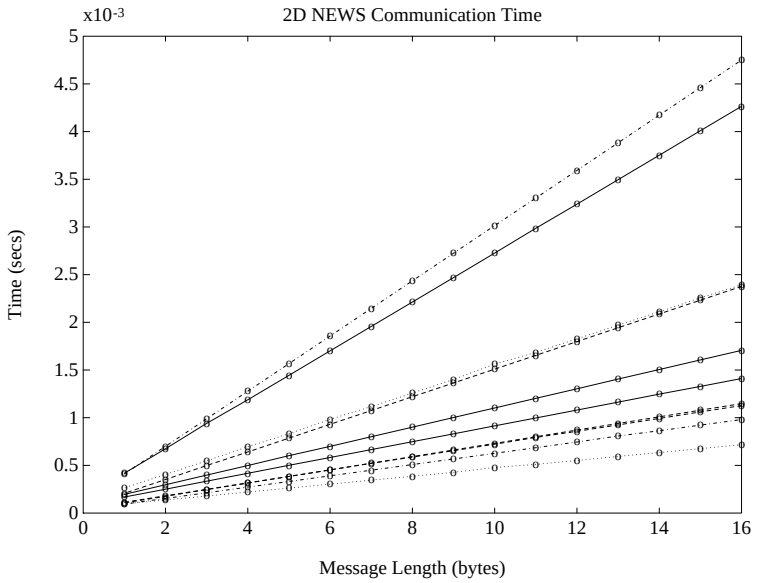


Figure 3.4: Observed NEWS communication latency for various 2D geometries illustrate the communication costs are linear in relation to the message length.

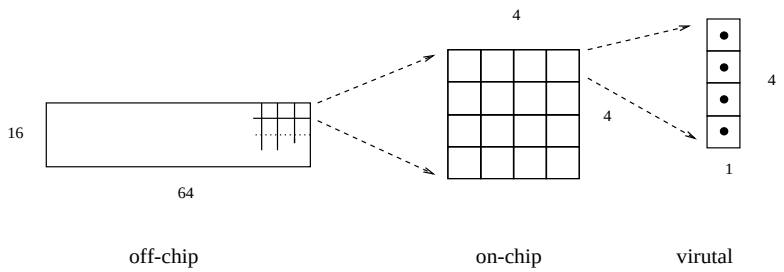


Figure 3.2: An example of one of several grid decompositions of a 256x256 grid onto a 16K processor CM-2 with virtual dimensions 4x1, on-chip 4x4, and off-chip dimensions of 16x64.

2. the total on-chip dimensions must equal the number of processors on a single CM chip

$$\prod_{i=1}^{N} c_i = 16$$

3. the total off-chip dimensions must match the size of the CM machine

$$\prod_{i=1}^{N} f_i = 2^{d-4}$$

3.2.1 First Experiments

To obtain some insight on how communication latency depends on geometry decomposition, we first ran communication experiments on various multidimensional geometries with varying message lengths. Figures 3.3, 3.4, and 3.5 illustrate the communication latency for several 1D, 2D, and 3D geometries, respectively. This data strongly suggests that NEWS communication latency can be modeled as a simple linear relationship

$$T(n) = \alpha + \beta * n$$

where the α and β are dependent on the given CM geometry, $\{\vec{f}, \vec{c}, \vec{v}\}$. The natural question, then, is how are the α and β parameters dependent on a specific geometry G; that is, what is $\alpha(G)$ and $\beta(G)$?

3.3 A Grid Communication Model

We now describe an analytical model of how the different communication hierarchies (virtual, on-chip, off-chip) interact in regular grid communications. By understanding the associated costs between these hierarchies, we can develop a communication model that is accurate to about +/- 5% in over 97% of the vp geometries and message length combinations, and is never off by more than 20% even for the most extreme configuration.

We study the effect of varying message lengths since they arise in several contexts. In scientific grid-based applications, for example, grid points need to communicate with their local neighbors when computing finite differences or performing local relaxation steps. Often, several parameters need to be communicated (the solution of shallow water equations, discussed in section 3.7, provides a concrete example of this,) and by careful "bundling" of such values into a single, longer message, we need pay for the startup cost only once. Such communication optimization strategies have been shown to be quite effective on distributed memory machines.

The NEWS communication phase on the CM-2 can be broken down into three parts, each corresponding to the off-chip, on-chip, and virtual processor phases:

3.2 Mapping N-dimensional Grids

As a mechanism for optimizing processor communications, the mapping of N-dimensional grids onto the pseudo-hypercube architecture of the CM-2 are specified by the use of virtual processor geometries. This mechanism specifies how the dimensions of the grid are mapped onto the CM-2's architecture components. Each CM-2 processor chip constist of 16 processors which can be arranged in an N-dimensional array. For example, the 16 processors can be arranged as a 4x4 square, or a 2x4x2 cube. In general, one can arrange the processors in an N-dimensional array with dimensions $\{c_1, c_2, \ldots, c_N\}$ where

$$\prod_{i=1}^{N} c_i \;=\; 16.$$

Furthermore, each physical processor may simulate several virtual processors v_i in each dimension. Thus we have the possibility of assigning each processor a $\{v_1 \times v_2 \times \ldots \times v_N\}$ subarray of virtual processors. The product $V = \prod_{i=1}^{N} v_i$ is termed the *vp-ratio* of the geometry and since it is always an exact power of two, one can denote the integer $v \equiv \log_2 V$.

Viewing the CM-2 as a d-Dimensional hypercube with 2^d processors, we see that 2^4 or 4 dimensions of the hypercube are embedded in the processor chip, and the remaining $d-4$ dimensions constitute the physical hypercube network connecting these chips. Furthermore, each processor can simulate 2^v virtual processors. One can then map a grid of 2^m points onto a d-dimensional CM-2 as

$$\underbrace{2^m}_{\text{total}} = \underbrace{2^{d-4}}_{\text{off-chip}} * \underbrace{2^4}_{\text{on-chip}} * \underbrace{2^{(m-d)}}_{\text{virtual}}.$$

For example, consider mapping a 256x256 grid onto a CM-2 with 16K physical processors. One such decomposition is shown in Fig 3.2. In this case, the sixteen processors on each CM-2 chip are configured as a 4x4 array, each physical processor simulates a 4x1 array of virtual processors, and the 1024 chips as configured as a 16x64 array. We denote this geometry as $G = \{\vec{f}, \vec{o}, \vec{v}\}$, where $\vec{f}$, $\vec{o}$, $\vec{v}$ denote the off-chip, on-chip, and virtual dimensions, respectively. In the example above, the components of G are given by

$$\vec{f} \;=\; \{16, 64\} \tag{3.1}$$
$$\vec{o} \;=\; \{4, 4\} \tag{3.2}$$
$$\vec{v} \;=\; \{4, 1\} \tag{3.3}$$

Hence this geometry dictates that the original problem size (256x256) can decomposed on the CM-2 follows:

	dimension		off-chip		on-chip		virtual
dim1	256	=	16	x	4	x	4
dim2	256	=	64	x	4	x	1

Of course, there are many other geometries that map the same 256x256 problem onto the CM-2 and depending on how often communication takes place along a given axis, some mappings may be more effective than others.

Software primitives exist that allow one to specify how the dimensions of the original problem will be mapped to these three levels of the CM-2 architecture. Thus, one can specify such a *geometry* by using `CM_create_detailed_geometry()` Paris primitives. Given a problem with an N-dimensional of size $n_1 \times n_2 \times \ldots n_N$ or $\prod_{i=1}^{N} n_i$, we can specify a mapping of off-chip components, $\vec{f} = \{f_1, f_2, \ldots, f_N\}$, on-chip components, $\vec{c} = \{c_1, c_2, \ldots, c_N\}$, and virtual components, $\vec{v} = \{v_1, \ldots, v_N\}$, where

1. for each dimension, the product of geometry lengths must match the problem dimension,

$$n_i = f_i * c_i * v_i, \;\; i = \{1, \ldots, N\},$$

3.1.8 The Paris Programming Model

Paris is a library of atomic instructions which the front end host executes to direct the actions of the CM. This is the lowest level programming model interface available.

Having access to this low-level interface allows programmers to specify precise control over the operations executed on the Connection Machine, and provides language developers a common base from which new languages and environments can be built upon.

The Paris programming model presents a large number of homogeneous processors, (similar in power to a conventional microprocessor) each with some local memory and able to address any other processor. Each processor carries out orders from the same instruction stream; optionally, some processors may choose to turn themselves "inactive" and remain idle during the duration of an instruction.

Paris augments conventional sequential languages (C, FORTRAN) available on the front end by providing library routines to perform all of the parallel actions. Some Paris commands, for example, instruct each processor to perform a floating point multiplication, or retrieve data from a neighboring processor.

The programmer can call Paris instructions directly as functions using the standard C, or FORTRAN compilers normally available on the front end, provided they link with the proper library. For example, in C one can call the Paris function

```
CM_f_cos_2_1L(y, x, 23, 8);
```

which instructs the active processors in the CM to calculate the cosine of all floating point `x` values (in parallel) and store the result in the parallel variable `y`, (i.e., $y = \cos(x)$.) The name of the function signifies 1) a "CM" Paris function, 2) "f" floating point function, 3) "cos" cosine function, 4) requiring "2" arguments (as opposed to $x = \cos(x)$), and 5) requiring a length "1L" argument. Since the Connection Machine can perform floating point arithmetic in arbitrary precision, the length argument determines the precision requested. For parallel integer variables, the length is a single argument: its bit length. For parallel floating point variables, the mantissa and exponent field lengths are specified independently. In the example above, the last two arguments indicate that the floating point variables are in IEEE single-precision (23 bits for the mantissa, 8 bits for the exponent.)

As another example, the following sequence of functions load each processor in the CM with parallel values and performs a floating point addition in each processor.

```
CM_f_move_constant_1L(A, 3.1, 23, 8);
CM_f_move_constant_1L(B, 1.7, 23, 8);
CM_f_add_3_1L(C, A, B, 23, 8);
```

This code is similar to

```
A = 3.1
B = 1.7
C = A + B
```

in CM-FORTRAN.

3.1.5 Floating Point Hardware

During a clock cycle each bit-serial processor can move only one bit, whereas the Wietek FPU can move a single-precision (32-bit) floating result. In other words, it requires at least 32 clock cycles for a single-precision floating point value to be moved out of a bit-serial processor. This potential bottleneck between bit processors and the FPU is avoided by dedicated hardware which, at each clock cycle, extracts a bit from *each* of the 32 bit-processors sharing an FPU. Thus, at the end of 32 clock cycles, there are 32 single-precision (32 bit) floating point values available for the FPU. This is discussed in more detail in section 3.4.1

3.1.6 Language Support

The CM-2 software support includes SIMD extensions to standard languages. *LISP is an extension of Common Lisp that supports parallel values (pvars), C* is an extension of the C programming language, and CM Fortran is an extension to Fortran 77 which incorporates most of the array operations introduced in the Fortran 8x standard.

These high level language extensions are built upon a common low-level virtual machine interface –a set of atomic commands collectively name Paris (PARallel Instruction Set), discussed in Section 3.1.8.

3.1.7 CM Fortran

The CM Fortran programming language is a superset of Fortran 77 employing many of the proposed Fortran 8x array extensions. Several books and articles are available on the proposed standard, in addition to the official Fortran Standards Committee X3J3 documents of the American National Standards Institute (ANSI).

Fortran 8x treats arrays as single entities. That is, one can operate on arrays in a syntax quite similar to other fundamental data structures such as reals or integers. Arrays can appear in numerical expressions, can be passed to functions, and can return the value of a function. DO loop constructs, while still accessible, are no longer necessary. The Fortran 8x program style is more concise, resembling more of the underlying mathematical notation and eliminating the need to introduce loop indices or labels. Furthermore, parallel machine architectures can exploit this notation and perform such matrix operations in natural parallel manner.

Consider the following differences between DO loops in Fortran 77 and array expressions in Fortran 8x:

```
          Fortran 77                        Fortran 8x

      DO 50 I = 1,100                    A = A + 1.7 + SIN(B)
      DO 50 J = 1,100
      DO 50 K = 1,100
50    A(I,J,K) = A(I,J,K) + 1.7 + SIN(B(I,J,K))
```

Some features of Fortran 8x, such as pointers, structures, modules, and precision control, are not part of CM Fortran; however, all of the language features discussed here are common to both Fortran 8x and CM Fortran, unless otherwise noted. Some of these common features include

- Array sections and vector-valued subscripts, e.g. A(1:N), B(P).
- Attributed type declarations, e.g., REAL, ARRAY(N,M)::A,B,C
- Array extensions of scalar operators and intrinsic functions, e.g., A = SIN(B), or C = A + B, where A, B and C are arrays.
- WHERE constructs to restrict array statements, e.g. WHERE (A>0.0) B = 1/A
- New array functions for transformation and inquiry, e.g. TRANSPOSE(A).

This theoretical peak, however, is not obtained in practice. The performance model we develop in this paper is aimed at showing what performance *can* be expected from the CM-2 when solving real problems.

3.1.2 Processor Memory

In the original CM-2, each bit processor had 8 Kbytes associated with its local memory space, resulting in 0.5 Gbytes for a 64K processor machine. Memory upgrades are available to expand the memory to 32K bytes per processors, resulting in a full machine configuration of 2 Gigabytes.

Each processor memory is divided into stack and heap regions. The management of these memory regions is performed by the front-end host computer. No code is stored in processor memory —only parallel data. Each bit processor can access its local memory at 5 megabits per second, providing a memory bandwidth of 300 gigabits per second [94], [95], [81].

3.1.3 Processor Communication

Communication between bit-processors on the same physical chip takes place through a Benes network. The 4096 physical CM *chips* in are connected in a hypercube topology: each chip forms the node of a 12 dimensional hypercube. Hypercube topologies have the attractive feature that *ring*, *tree* and *N-dimensional grid* topologies can be efficiently embedded in them [3].

In a machine with P processors, each processors has its own unique address, numbered from 0 to $P-1$. General communication between CM chips occurs through "routers"; every group of 16 bit-serial processors share a router which is responsible for forwarding outgoing requests from the chip onto the network, and servicing incoming message requests from other chips residing in the Connection Machine. A processor can request a value from any processor by specifying that processor's cube address, the beginning memory location of the data object, and its bit length.

NEWS Communication

For applications that utilize regular communication patterns along N-dimensional grids, faster communication primitives exist that do not entail the overhead of using the chip router. For example, in a 2-dimensional grid topology, each processor can simultaneously retrieve data from its immediate south neighbor. This type of regular communication is termed NEWS (each letter representing the four respective directions: north, east, west, south) to distinguish it from the more general (router) communication.

3.1.4 Virtual Processors

An elegant feature of the Connection Machine is its availability to utilize virtual processors. The CM can be programmed as if it had many more processors than it physically has. Normally, one associates a single data point (e.g. a grid point) with each processor; although 64K processors may at first appear overwhelming, many scientific applications (specifically 3D models) require literally millions of data points.

The ratio of physical processors present on a specific CM to the number of virtual processors requested by the application is termed the **vp ratio.** Each physical processor can *simulate* more than one *virtual* processors by dividing its memory space into equal parts and sequentially serving the virtual processors.

This virtual processor capability is an elegant feature of the CM-2 which essentially allows one to map the machine to the size of the problem, rather than the other way around. It facilitates programming by eliminating the intermediate-level decomposition (mapping of data points to processors) for regular structures altogether. Furthermore, the use of high vp ratios not only allows one to run extremely large distributed data objects, but actually *enhances* the performance of the CM (see Section 3.4.)

Chapter 3

Connection Machine CM-2

The Connection Machine CM-2 is a massively parallel single instruction multiple data stream (SIMD) distributed memory hypercube computer. A fully configured machine consists of 64K (65,536) bit-serial processors utilizing 2,048 Weitek 3132 floating point units (FPU) for an aggregate peak performance of over 25 Gflops.

To discuss the performance issues of this machine, we need first to understand a few fundamental principles of its basic operation. The Connection Machine is not a standalone computer: it stores data but no code. Control flow execution is carried out on the front-end or host computer: a Sun workstation, Symbolic 3600 , or DEC VAX mainframe. Program development (compiling, editing) and its *execution* occur on the front-end. At runtime, the front-end attaches to the CM and takes control of it, issuing a command stream in a synchronized, lock step fashion.

3.1 Overview

3.1.1 Processors

The basic architectural unit of the CM-2 is the CM bit-serial processor. Sixteen bit-serial processors are fabricated on a single physical chip; a full CM-2 contains 4096 such chips. Every pair of CM-2 chips (32 bit processors) shares a Weitek 3132 floating point unit (FPU). These Weitek chips provide the CM-2 with its tremendous number-crunching strength. When properly pipelined, each Weitek 3132 can produce one (single-precision) floating point multiply and one addition result per clock cycle. A CM-2 with an 6.7 MHz clock can run each Weitek 3132 at 13.3 MFLOPs, thus delivering a machine *peak* speed of 26.7 GFLOPs.

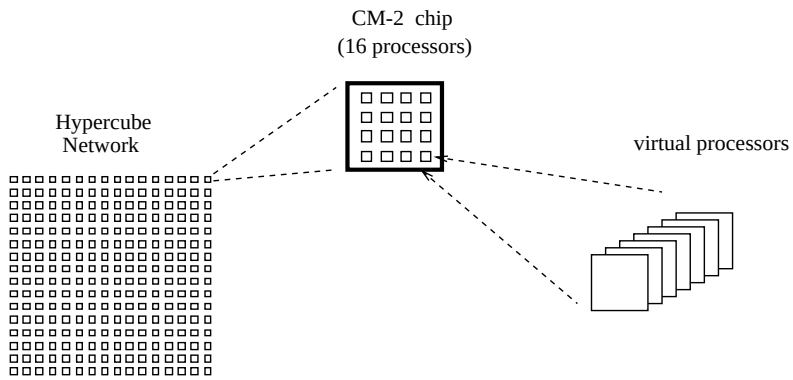


Figure 3.1: Hierarchical view of the Connection Machine 2 Architecture

2.7 Overview

This work covers a lot of ground; it provides an analysis of RISC architectures, operating systems, memory hierarchies, numerical algorithms, parallel programming languages, performance metrics, parallel data decomposition strategies, distributed memory and shared memory schemes, synchronization protocols, and multilevel performance models. Nevertheless, the continuing theme running throughout this research remains the same: how to develop accurate performance models for these complex parallel architectures.

The next four chapters are each devoted to the specific performance analysis of the CM-2, the Myrias SPS-2, the ES-1, and the Intel iPSC/860. Each chapter discusses details of the architecture, the experiments, and their respective performance models. Because of the large scope of material presented, there is not one conclusion to this work, but five: a separate conclusion for each of the four architectures studied and a a final conclusion that ties the chapters together and summarizes what we have learned in attempting to provide better performance models of parallel architectures.

```
   subroutine calc1()
   include 'input.cb'
   real pnorth(m,n)

   pnorth = cshift(p, 1,-1)
   cu=.5*(p + pnorth)*u
   cv=.5*(p + cshift(p, 2,-1))*v

   z = (fsdx*(v-cshift(v,1,-1)) -
>       fsdy*(u-cshift(u,2,-1)))/
>       (cshift(pnorth,2,-1)+
>       cshift(p,2,-1)+p+pnorth)

    h = p+.25*(cshift(u,1,1) *
>       cshift(u,1, 1)+ u*u +
>       cshift(v,2, 1)*
>       cshift(v,2, 1)+ v*v)

 return
 end
```

Figure 2.15: A Shallow Water coding example using CM-Fortran.

```
      SUBROUTINE CALC1
      include 'pfp.h'
      include 'shallow.h'
       real fsdx,fsdy

      FSDX = 4./DX
      FSDY = 4./DY

      pardo_decomp(100,j,1,N, numtasks)
      pardo_local i
      do 100 j = 1,N
      do 100 i = 1, M
      CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
      CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
      Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
     1          -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
      H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1               +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE
      end_pardo
C
C     PERIODIC CONTINUATION
C
      DO 110 J=1,N
         CU(1,J) = CU(M+1,J)
         CV(M+1,J+1) = CV(1,J+1)
         Z(1,J+1) = Z(M+1,J+1)
         H(M+1,J) = H(1,J)
  110 CONTINUE
      DO 115 I=1,M
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)
  115 CONTINUE
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)

      RETURN
      END
```

Figure 2.14: A Shallow Water coding example using PFP Fortran.

```
      SUBROUTINE CALC1
      include 'shallow.h'
       real fsdx,fsdy

      FSDX = 4./DX
      FSDY = 4./DY
      PARDO 1000 ip=1,numtasks
      DO 1000 ip=1,numtasks
      call gridproc(ip,2,qx,qy,0)
      istartx = istx(qx)
      iendx = ienx(qx)
      istarty = isty(qy)
      iendy = ieny(qy)
      do 100 j = istartx,iendx
      do 100 i = istarty,iendy
      CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
      CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
      Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
     1          -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
      H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1               +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE
 1000 CONTINUE
C
C     PERIODIC CONTINUATION
C
      DO 110 J=1,N
         CU(1,J) = CU(M+1,J)
         CV(M+1,J+1) = CV(1,J+1)
         Z(1,J+1) = Z(M+1,J+1)
         H(M+1,J) = H(1,J)
  110 CONTINUE
      DO 115 I=1,M
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)
  115 CONTINUE
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)

      RETURN
      END
```

Figure 2.13: A Shallow Water coding example on the Myrias SPS-2.

```
      SUBROUTINE CALC1
       include 'shallow.h'
       real fsdx,fsdy

      FSDX = 4./DX
      FSDY = 4./DY

C$ES  PARALLEL DO
C$ES  local qx,qy,i,j,istartx,iendx,istarty,iendy
      DO 1000 ip=1,numtasks
      call gridproc(ip,2,qx,qy,0)
      istartx = istx(qx)
      iendx = ienx(qx)
      istarty = isty(qy)
      iendy = ieny(qy)
      do 100 j = istartx,iendx
      do 100 i = istarty,iendy
      CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
      CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
      Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
     1          -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
      H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1               +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE
 1000 CONTINUE
C$ES  end do
C
C     PERIODIC CONTINUATION
C
      DO 110 J=1,N
         CU(1,J) = CU(M+1,J)
         CV(M+1,J+1) = CV(1,J+1)
         Z(1,J+1) = Z(M+1,J+1)
         H(M+1,J) = H(1,J)
  110 CONTINUE
      DO 115 I=1,M
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)
  115 CONTINUE
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)

      RETURN
      END
```

Figure 2.12: Shallow Water coding example in ES-1 Fortran.

```
      DO 100 J=1,N
       DO 100 I=1,M
         CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
         CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
         Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
     1          -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
         H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1               +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE
C
C     PERIODIC CONTINUATION
C
      DO 110 J=1,N
         CU(1,J) = CU(M+1,J)
         CV(M+1,J+1) = CV(1,J+1)
         Z(1,J+1) = Z(M+1,J+1)
         H(M+1,J) = H(1,J)
  110 CONTINUE
      DO 115 I=1,M
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)
  115 CONTINUE
      CU(1,N+1) = CU(M+1,1)
      CV(M+1,1) = CV(1,N+1)
      Z(1,1) = Z(M+1,N+1)
      H(M+1,N+1) = H(1,1)
```

Figure 2.11: Fortran 77 subroutine CALC1 of shallow water application.

At further steps, a time filter is applied to handle the weak instabilities of the leapfrog scheme for $f = u, v$, and P

$$F^{(n)} = f^{(n)} + \alpha(f^{(n+1)} - 2f^{(n)} + f^{(n-1)}) \tag{2.23}$$

where α is a filtering parameter. The filtered values of the variables at the previous time-step are used in computing new values at the next time step.

Each time step of this nonlinear system consists of three parts:

1. subroutine Calc1
   (a) Formulate mass fluxes U, V using a staggered grid averaging for pressure P.
   (b) Formulate vorticity, ζ, using second-order centered difference approximation to $\partial v/\partial x$ and $\partial u/\partial y$. Compute $Z = \zeta/P_a$ where P_a is the staggered four-point average of pressure.
   (c) Formulate H as given by equation 2.17 using two-point average value for u and v.
2. subroutine Calc2
   (a) compute $\partial u/\partial t$ using equation 2.19
   (b) compute $\partial v/\partial t$ using equation 2.20
   (c) compute $\partial P/\partial t$ using equation 2.21
3. subroutine Calc3
   (a) perform leapfrog time differencing for u, v, and P.

An example subroutine of the sequential Fortran 77 code is shown in figure 2.11. There are two parts, the first handles the updating of the grid variables and the second parts handles the updating of the periodic boundary conditions.

To implement the Shallow Water Equations onto the iPSC/860, several modifications to the sequential version need to be made. First, we realize that there will be two parts of the program: one runs on the host, or front end, and is responsible for initializing the data, and the second runs on each node, performing the shallow water calculations on a local submatrix representing the subfields of U, V, and H. These fields are distributed in a block-ordered form, such that each node receives a representative submatrix. Since grid points need only access their nearest neighbors in a 2D lattice, most of the data already resides locally in each node. Only on the boundaries will there be a need to access a neighboring processor's values. In such cases, specific communication routines (discussed below) handle the transfer of boundary information between nodes. The basic communication pattern between the primary field variables is illustrated in Fig 6.35. See section 6.5 for details.

For the Myrias SPS-2 and Evans & Sutherland ES-1 shared memory machines, we use the `decompose()` mechanisms of section 2.4.2 to map the 2D grids onto the system. The code samples for the ES-1, Myrias SPS-2, and PFP Fortran in figures 2.12, 2.13, and 2.14, respectively. In each case, only the doubly nested DO loop is parallelized; the boundary-copying section exhibits too little granularity to be effectively parallelized. In the ES-1 version, all of the variables used in the `decompose()` mechanism need to be declared local so that each processor has its own value. Details are described in section 5.4. The Myrias programming model is presented in section 4.1.

The SIMD version of CALC1 in CM-Fortran is shown in figure 2.15. Here the boundary copying section is not necessary, since it occurs as part of the `cshift()` Fortran 90 array function. No subscripts appear in the code, since all of the operations are applied to arrays. The specific mapping of array data throughout the machine is discussed in detail in section 3.2.

2.6 Shallow Water Equations

To further examine the current capabilities of parallel architectures, we describe the implementation of a standard two-dimensional atmospheric model - the Shallow Water Equations. These equations provide a primitive but useful model of the dynamics of the atmosphere or of certain ocean systems. For example, versions of such models have been used to study the wave motion resulting from island tsunamis [57] and to numerically model atoll reef harbors [58]. Because the model is simple, yet captures features typical of more complex codes, the model is frequently used in the atmospheric sciences community to analyze performance of computers by Capotondi [14], Wubbs [104], Hoffman & Snelling [46], Sato & Swarztrauber [81], Goede [35], Hack & Ruediger [38], and Barton & Withers [6]. Furthermore, the model has been extensively analyzed mathematically and numerically [80], [12].

The Cartesian model of the Shallow Water Equations is based on Sadourny [80] and the parallel implementations in this section are based on the original program presented in Hoffmann, Swartzrabuer, and Sweet [46]. (See Snelling [88] for dataflow version of these equations in SISAL and Tankqueray [93] for a parallel implementation on the FPS T Series Transputer machine.) Mathematically, the Shallow Water equations take the form

$$\frac{\partial u}{\partial t} - \zeta v + \frac{\partial H}{\partial x} = 0 \tag{2.14}$$

$$\frac{\partial v}{\partial t} + \zeta u + \frac{\partial H}{\partial y} = 0 \tag{2.15}$$

$$\frac{\partial P}{\partial t} + \frac{\partial Pu}{\partial x} + \frac{\partial Pv}{\partial y} = 0 \tag{2.16}$$

where u and v are the x and y velocities, respectively; P is the pressure; H, related to the height field is defined as

$$H = P + (u^2 + v^2)/2 \tag{2.17}$$

and the vorticity, ζ, is given by

$$\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}. \tag{2.18}$$

It is required to solve these equations in a rectangular region $a \leq x \leq b$, $c \leq y \leq d$. Periodic boundary conditions are imposed on u, v, and P, each of which satisfies $f(x+b, y) = f(x+a, y)$, $f(x, y+d) = f(x, y+c)$.

These equations can be somewhat simplified by introducing mass fluxes $U \equiv Pu$, $V \equiv Pv$, and potential velocity $Z \equiv \zeta/P$ in terms of which reduce to

$$\frac{\partial u}{\partial t} - ZV + \frac{\partial H}{\partial x} = 0 \tag{2.19}$$

$$\frac{\partial v}{\partial t} - ZU + \frac{\partial H}{\partial y} = 0 \tag{2.20}$$

$$\frac{\partial P}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} = 0. \tag{2.21}$$

The above equations are discretized on a rectangular staggered grid with periodic boundary conditions. The variables P and H have integer subscripts, Z has half-integer subscripts, U has integer and half-integer subscripts, and V has half-integer and integer subscripts, respectively.

Initial conditions are chosen such that $\nabla \cdot \vec{v} = 0$ at all times. Time differencing is implemented using the Leap-frog method. The values at t_1 are computed using one step of the Euler's method

$$f^{(1)} = f^{(0)} + \frac{\partial f}{\partial t}^{(0)} \Delta t \tag{2.22}$$

```
#define pardo(label, varindex, a, b) PARDO label varindex = a, b

/* no local declarations in mpfc, so code as comment */
#define pardo_local     ^c    local
#define endpardo        continue

/*  Parallel sections in mpfc on SPS-2 */
/* parbegin is not redefined */
#define parsection      parallel
#define parend          endpar

#else SPS2


/* Default to sequential f77 code */

#define pardo(label, varindex, a, b)    do label varindex = a, b
#define pardo_decomp(label, varindex, a, b, T)    do label varindex = a, b
#define pardo_local     ^c   local
#define endpardo        continue
#define parbegin
#define parsection
#define parend

#endif
#endif
```

```
/*                                                                      */
/* NOTE: the output of this macro needs to be passed through            */
/*             tr '^~' '\012\011'                                       */
/*                                                                      */
/* so that the ^ character can be translated into a newline             */
/*  and the ~ character converented to a physical tab                   */
/*                                                                      */
/* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/
/* LIMITATIONS:                                                         */
/* (1) cannot nest these types of pardos (duplicate local loop variables*/
/*     will definitely cause problems.)                                 */
/* (2) the following are keywords: pardo, parbegin, parend, parsection  */
/*        decompstart, decompend.                                       */
/* (3) for the SPS-2, remember that mpfc considers the following as      */
/*     keywords also : parallel, endpar.                                */
/* (4) when using pardo_decomp, the following variables are defined:    */
/*          mystart, myend, myprocnum                                   */
/* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/

/* decompose macros: sequence a, a+1, a+2, ..., b  into T semiequal parts */
/* macros below calculate the start and end indexes for task t */

#define decompstart(a, b, T, t)   (a) + (((b)-(a)+1)*(t-1))/(T)
#define decompend(a, b, T, t)     (a) - 1 + (((b)-(a)+1)*(t))/(T)

/*:::::::::::::::::::::::::: ES-1 macros ::::::::::::::::::::::::::::::*/
#ifdef ES1
#define pardo_decomp(label, varindex, a, b, T) \
        ^c$es parallel do  \
        ^c$es local mystart myend \
        ^~       do label myprocnum = 1, T\
        ^~       mystart = a + ((b-a+1)*(myprocnum-1))/T \
        ^~       myend = a -1 +((b-a+1)*myprocnum)/T \
        ^~       do label varindex = mystart, myend

#define pardo(label, varindex, a, b) \
        ^c$es parallel do \
        ^~       do label varindex = a, b

#define pardo_local     ^c$es LOCAL

/* by inserting a 'continue'on current line, we can label endpardo's */
/* see above example.   */
#define endpardo        \
        continue \
        ^c$es end do

/* Parallel sections in f77 on ES-1 */
#define parbegin        \
        ^c$es parallel sections \
        ^c$es section
#define parsection      ^c$es section
#define parend          ^c$es end parallel sections

#else ES1

#ifdef SPS2

#define pardo_decomp(label, varindex, a, b, T) \
        ^~ PARDO label myprocnum = 1, T\
        ^~     mystart = a + ((b-a+1)*(myprocnum-1))/T \
        ^~     myend = a -1 +((b-a+1)*myprocnum)/T \
        ^~     do label varindex = mystart, myend
```

There are also portable parallel-sections macros.

To use the new "parallel Fortan Preprocessor" (pfp) include pfp.h in the Fortran source and use the following Bourne shell script:

```
#   pfp: parallel fortran preprocessor for Myrias SPS-2 or ES-1
#
#   source file should have included "pfp.h"
#
/lib/cpp $* | grep -v '^#' | tr '^~' '\012\011'
```

The pass through the translator (tr) is necessary, since newlines and tabs have to "coded" so that they are not ignored during macro substitution. This is shown in the listing of pfp.h below.

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<[ CUT HERE ]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
/* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/
/*    P F P:   P a r r a l l e l   F o r t r a n   Preprocessor                   */
/* ------------------------------------------------------------------------------*/
/*     Generic Parallel Fortran Preprocessor for SPS-2 and ES-1                  */
/* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/
/*                                                                               */
/*   Two constructs available:                                                   */
/*     (1) parallel do                                                           */
/*           to perform a loop from i=1,N+1 among P tasks:                        */
/*                                                                               */
/*              pardo_decomp(110, i, 1, N+1, P)                                  */
/*              pardo_local temp1, temp2                                         */
/*                 ...use index i...                                             */
/*      100     endpardo                                                         */
/*                                                                               */
/*                                                                               */
/*     (2) parallel sections                                                     */
/*                                                                               */
/*                                                                               */
/*    More examples:                                                             */
/*      ) parallel global reduction with add                                     */
/*         (sum up elements of v(i) from i=1 to N.)                              */
/*                                                                               */
/*              total = 0.0                                                      */
/*              pardo(110, p, 1, numprocs)                                       */
/*      pardo_local i, istart, iend, sum                                         */
/*         istart = decompstart(1,N, p)                                          */
/*         iend = decompend(1,N,p)                                               */
/*         sum = 0.0.                                                            */
/*         do 120 i=istart, iend                                                 */
/*  115     sum = sum + v(i)                                                     */
/*                 s(p) = sum                                                    */
/*      110     endpardo                                                         */
/*              do 120 i=1,numprocs                                              */
/*      120        total = total + s(i)                                          */
/*                                                                               */
/*                                                                               */
/* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/
/*                      Parallel Do Loops                                        */
/* break up a do loop ranging from a to b into T tasks on the ES-1               */
/*  creates three new (local) variable: myprocnum, mystart, myend               */
/* These can be used inside the parallel loop if needed.                         */
/*                                                                               */
/* Alternately, this macro can use a predefined variable 'numtasks'              */
/* rather than pass the number of tasks, T, explicitly.                          */
/* ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*/
```

2.5 Parallel Fortran Preprocessor

In aiding the program development of maintaining portable shared memory codes between the ES-1 and SPS-2, we have developed a set of machine independent parallel loop constructs that support various decomposition methods for 1D and 2D grids. These constructs are in the form of macros in which the source is first passed through `cpp` or some equivalent preprocessor. This macro package has been termed as the PFP for Parallel Fortran Preprocessor; it is a tiny extension to Fortran 77 syntax which allows for prescheduled parallel loops. PFP contains a much smaller subset of constructs than found in larger projects; however, it has the advantage of being able to run on pseudo-shared memory machines (such as the Myrias SPS-2) which lack explicit variable-locking synchronization primitives.

The PFP parallel constructs have the ability to fuse multiple loop iterations into one task. This "loop bundling" capability is performed automatically for the ES-1, but must be defined explicitly for the SPS-2.

For example, consider a simple example to compute a vector operation $\vec{c} = \vec{c} + \vec{a} + \vec{b}$. In maintaining code between the ES-1 and Myrias SPS-2, one would express this operations as follows

```
      call decompose(1,n1x,numtasks,ist,ien)

#ifdef SPS
      PARDO 100 p=1,numtasks
#else  SPS
C$ES    PARALLEL DO
C$ES    local i
      DO 100 p=1,numtasks
#endif SPS
            DO 150 i=ist(p),ien(p)
                cv(i) = cv(i) + av(i)*bv(i)
150         CONTINUE
100 CONTINUE
C$ES    END DO
2000    CONTINUE
```

Using the PFP macros the same expression becomes

```
      include 'pfp.h'

      pardo_decomp(100,i,1,n1x,numtasks)
          cv(i) = cv(i) + av(i)*bv(i)
100 endpardo
```

The `pardo_decomp` macro reads as

```
      pardo_decomp 100 i=1,n1x  using numtasks
```

and performs the same task decomposition as above. Furthermore, there are no function calls (like to decompose()), everything is done inline. There is also no need to declare extraneous arrays (like ist[] or iend[].)

We then make the first r1 intervals of length $h_1 + 1$ and the remaining $T_1 - r_1$ intervals of length h_1, and similarly in the other dimension(s). This is conveniently done with a procedure

```
decompose(a,b,t,istart,iend)
```

which decomposes an interval [a,b] into t near-equal length subintervals as above, and which initializes arrays `istart(t)`, `iend(t)` with the start and end indices of each subinterval. Thus, for example, the complete code to parallelize the above loop on the Myrias SPS-2 takes the form

```
      decompose(2,n1-1,t1,istart1,iend1)
      decompose(2,n2-1,t2,istart2,iend2)
      pardo 10 q1=1,t1
      pardo 10 q2=1,t2
          do 10 i= istart1(q1),iend1(q1)
          do 10 j= istart2(q2),iend2(q2)
              b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1)
     >                    + a(j-1,i) + a(j+1,i)
10    continue
```

In practice we collapse the nested parallel DO loops into a single parallel loop of product length T=T1*T2 which minimizes parallel loop overhead and also allows implementation on systems, such as the ES-1, which restrict parallelism to a single level.

The above procedure provides many different parallelizations of a given problem, one for each possible factorization of the number of tasks T. At one extreme are decompositions by rows (case T1=1), and at the other extreme are decompositions by columns (T2=1), with intermediate values representing decompositions by rectangles. As we will see, performance is strongly influenced by which of these choices is made (see sections 4.7 and 5.3.5).

```
subroutine SendEast(A)
include 'gridsize.h'
include 'ipsc.h'
include 'block.h'
real*LENR A(0:Ny+1,0:Nx+1)
integer i,j
integer eastproc, westproc, bxlen, bylen
integer blen, rproc, rpid

westproc = west(myproc, NPx, NPy)
eastproc = east(myproc, NPx, NPy)

c.......perform actual communication only if destination is another
c.......processor.  Otherwise, there is no point in sending data to
c.......yourself.

if (eastproc. eq. myprocnum) return

c.......otherwise, continue with the explicit communication


bxlen = LENR*(Nx+2)
bylen = LENR *(Ny+2)

call sendw(chan1, EASTMSG, A(0,Nx), bylen, eastproc, pid)
call recvw(chan2, EASTMSG, A(0,0),  bylen, eastproc, pid)

c.......make sure there are no communication errors
if (blen.ne.bylen) then
    write(systring,*)'SendEast: bad message length of ',blen
    call syslog(pid,systring)
    return
endif

if (rproc.ne.west(myproc,NPx,NPy)) then
    write(systring,*)'SendEast: mesg from wrong proc ',rproc
    call syslog(pid,systring)
    return
endif

return
end
```

Figure 2.10: Example of 2D grid communication subroutine for the iPSC/860.

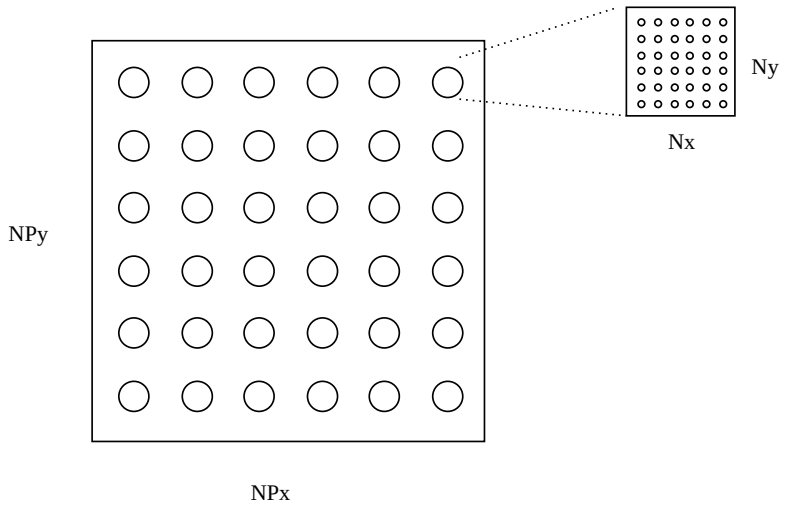


Figure 2.9: Decomposition of NP processors as a NPx by NPy grid, each processor receiving a Nx by Ny grid subproblem.

and can be tailored to perform efficient mapping (see section 6.3.1.) The specific is user defined and must be consistent with the distribution routines below.

The following functions,

```
integer function SendNorth(node, NPx, NPy)
integer function SendSouth(node, NPx, NPy)
integer function SendWest(node, NPx, NPy)
integer function SendEast(node, NPx, NPy)
```

let each processor know who its neighbors are in 2D array configuration of the hypercube. These functions are used in a modular manner by the `SendNorth()`, `SendSouth()` routines discussed above. Figure 2.10 illustrates a sample code for the `SendEast()` function.

There are routines that allow the host to distribute large matrices to hypercube nodes in a block-ordered manner. These routines take the form

```
subroutine distblock(A)
subroutine gatherblock(A)
```

where `distblock()` ships submatrices to respective nodes, and `gatherblock()` performs the complimentary operation of gathering the submatrix pieces from each node. When the above routines are encountered in a node program, they perform the expected function; i.e., `distblock()` causes a node to stop and wait for its submatrix piece, while `gatherblock()` ships out that submatrix to the host.

2.4.2 Shared Memory

For partitioning 2D grids on shared memory models, such as the Evans & Sutherland and Myrias SPS-2, the partitioning scheme used is straightforward. Let T=T1*T2 be a factorization of T. Then we divide the index interval [2,$n_1 - 1$] into T1 essentially equal pieces and similarly we divide [2,$n_2 - 1$] into T2 pieces. The tensor product of the interval decompositions defines the 2D subgrid decomposition. In case T1 does not divide $n_1 - 2$ evenly, we can write:

$$n_1 - 2 = h1 * T1 + r1, \;\; 0 < r1 < T1$$

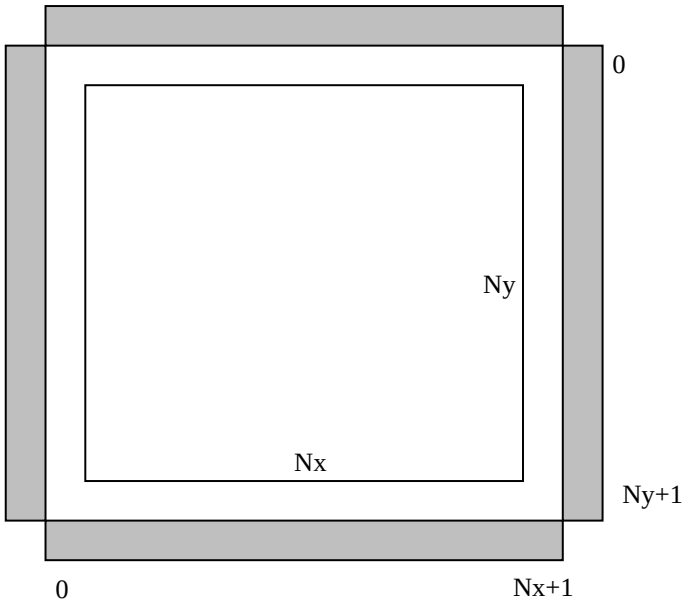


Figure 2.8: Communication of block decomposed subgrid occurs on the boundary.

munication routines:

```
subroutine SendSouth(A)
subroutine SendNorth(A)
subroutine SendWest(A)
subroutine SendEast(A)
```

These routines ship the boundaries of an (0:Ny+1,0:Nx+1) submatrix between the hypercube nodes. Each node contains an (Ny,Nx) matrix with data and extra boundaries on each side to hold the values from neighboring nodes, as shown in figure 2.8. Furthermore, these routines send corner values of neighboring nodes as well. For example, when a node executes

```
SendSouth(A)
SendWest(A)
```

the value of the processor directly north-east is produced in A(0,0). Thus, the access pattern for these communication primitives includes access to corner values as well. The pattern of communication between each lattice point is exemplified by the following star pattern shown in figure 2.6. Each node holds a respective (i,j) block-ordered submatrix of some larger (M,N) matrix originating on the front end. We are assuming that there are NP processors mapped as an NPy x NPy 2D processor grid. (For simplicity, we also assume that NPy divides M and NPx divides N.)

The mapping of subgrids to processors in shown in figure 2.7. Here we take a Mx by My problem domain and block decompose it onto a NPx by NPy processor grid, each processor receiving a Nx (= Mx/NPx) by Ny (= My/NPy) subproblem, as shown in figure 2.9. For simplicity, we assume that NPx divides Mx and NPy divides My. The mapping of hypercube nodes to elements of the NPy x NPy processor array are handled by functions:

```
integer function getmyprocx(node, NPx, NPy)
integer function getmyprocy(node, NPx, NPy)
integer function nodefromxy(myprocx, myprocy, NPx, NPy)
```

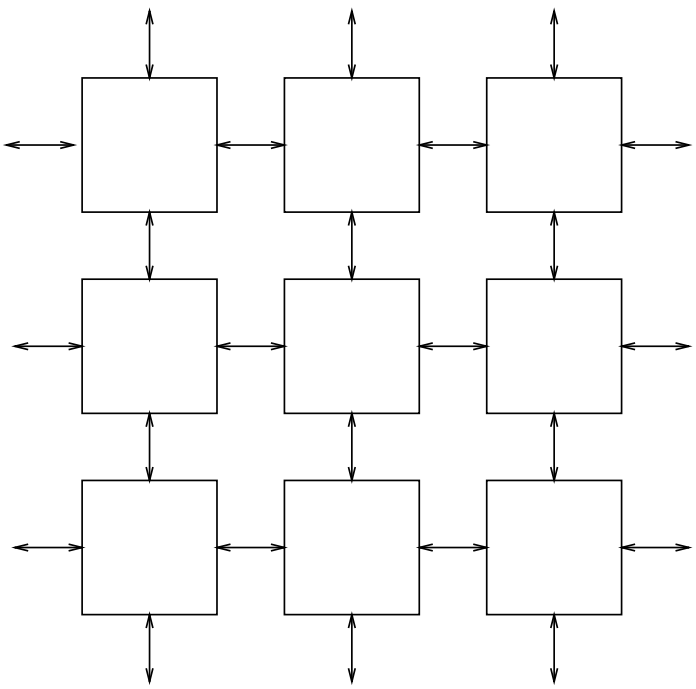


Figure 2.7: Mapping of subgrids to processors.

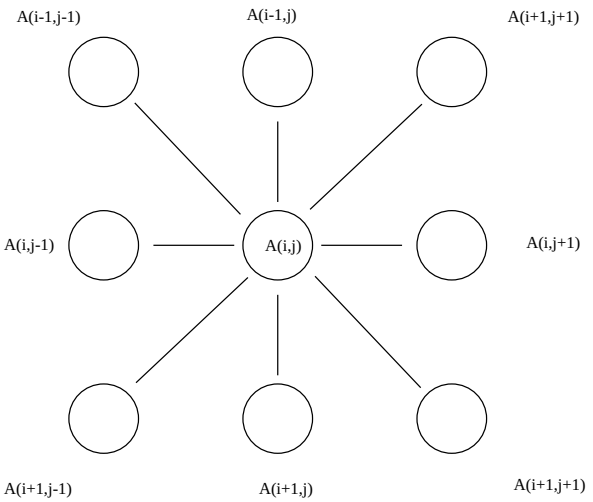


Figure 2.6: Basic 2D grid communication pattern.

2.4 Partioning of Two Dimensional Grids

2.4.1 Distributed Memory

For Intel iPSC/860, we present various mapping strategies for 2D grids. The basic idea is to decompose the large problem grid onto various processors connected through the network. For stencil-type communication patterns where each gridpoint requires communication with its 8 neighbors (figure 2.6) the subgrids are mapped onto individual processors as shown in figure 2.7.

We study two different mapping strategies for assigning processors to subgrids: one uses a simple row-ordering scheme similar to how the C language stores matrices, and the other uses Binary Gray Reflected Coding (BGRC) to ensure that neighboring subgrids map onto neighboring processors within the hypercube topology. These communication primitives can be easily extended to general distributed memory machines. It is essential to have *both* mapping capabilities, since there may be situations where the numbers of processors required for an application is not a integer power of two. For example, a user may wish to allocate a 32-processor subcube, but assign 25 processors to work on one program, while the remaining 7 processors work on some other parallel application. Or, one may wish to request a 10x11 processor grid on a 128 node hypercube.

The importance of supporting non-power of two mappings is even more relevant to future versions of the iPSC that will implement 2D meshes. On a hypercube, one has two choices as to how to implement non-power-of-two processor grids. One could simply increase the number of processors in each dimension to the next power of two and use a BGRC into this larger grid. Thus, a 3x5 processor grid would be mapped into as a 4x8 processor grid in a 32 node hypercube. Although the locality is preserved, we now require more than twice as many processors than what we will actually use. An alternate strategy would be to map a 3x5 processor grid into a 16 node hypercube, although nearest neighbor subgrids would not correspond to nearest neighbor processors. Thus we have a tradeoff: one can preserve locality at the expense of requesting larger subcubes, or we sacrifice locality (communications more expensive) but utilize fewer processors (better machine efficiency.)

Figure 2.6 illustrates the array reference pattern commonly encountered in finite difference calculation and relaxation operations. The element $A(i, j)$ needs to reference some or all of its eight nearest neighbors. This means that elements on the boundary of a processor subgrid will require to share boundary values among neighboring processors,

To aid the implementation of 2D grid-based processes, we have defined the following higher level com-

```
c$es parallel do
c$es local i
         do 10 j = 2,n1-1
         do 10 i = 2,n2-1
             b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1)
                + a(j-1,i) + a(j+1,i)
 10      continue
c$es end do
```

Figure 2.4: ES-1 Fortran code sample of 2D relaxation step.

```
       call sendsouth(A);
       call sendnorth(A);
       call sendeast(A);
       call sendwest(A);

         do 10 j = 1,Nx
         do 10 i = 1,Ny
             b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1)
                + a(j-1,i) + a(j+1,i)
 10      continue
```

Figure 2.5: Distributed Memory Fortran code sample of 2D relaxation step.

Intel iPSC/860

The programming model of the iPSC/860 is similar to most MIMD distribute memory systems. Parallelism is achieved by having nodes run their own program and communicate explicitly through `send` and `receive` communication primitives. These communication primitives are discussed in detail in section 6.3. For many problems, however, these communication primitives are at a low level and there is great difficulty in implementing them correctly in high level applications. For grid-based communication patterns, there have been several software packages developed; we describe one such package in section 2.4.1. It contains higher-level communication primitives which can intelligent communicate rows and columns of distributed 2D structures.

For example, the 2D relaxation code for the iPSC/860 is shown in figure 2.5. Here the `sendsouth()`, `sendnorth()`, `sendeast()`, and `sendwest()` routines performs similar operations to the `cshift()` array operators in CM-Fortran described above. The loops iterate not over the total grid size (n1,n2), but over the local per-processor subgrid size (Nx,Ny). If the total grid size is not evenly divisible among the processors, then Nx (as well as Ny) may have different values on various nodes. The details of this mapping is explained in the next section.

```
      where (interior)
        B = 4.0 * A + eoshift(A,1,1) + eoshift(A,1,-1) +
     >              eoshift(A,2,1) + eoshift(A,2,-1)
      endwhere
```

Figure 2.2: Parallel version of 2D relaxation step in CM-Fortran for the CM-2.

.

```
      pardo 10 j = 2,n1-1
      do 10 i = 2,n2-1
          b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1)
             + a(j-1,i) + a(j+1,i)
10    continue
```

Figure 2.3: Myrias Fortran code sample of 2D relaxation step.

```
      pardo 10, I = 1, N
          <Code to be executed>
10    continue
```

generates N tasks. Any Fortran DO loop may be replaced by a PARDO provided the body of the loop has no data dependencies that prohibit it from executing in parallel.

Thus, the parallelization of the 2D Fortran code relaxation example on the Myrias SPS-2 is given in figure 2.3. The code is nearly identical to the sequential version (figure 2.1) except the outer `do` construct has been changed to a `pardo`. The second level of loop nesting could also be made parallel, if enough processors were available and task granularity is sufficiently large.

One problem with the code in figure 2.3 is that in some cases it may generate too many tasks, particularly if n1 is much larger than the number of processors available. The next section (2.4) discuss strategies for fusing loop iterations and reducing this effect.

Evans & Sutherland ES-1

The ES-1 uses compiler directives inserted into traditional Fortran 77 codes to achieve parallelism. A full discussion is presented in section 5.2.3. These directives control the management of multiple thread execution, critical sections, synchronization primitives, locks, and activation/deactivation of threads. The ES-1 Fortran compiler directives begin with an **C$ES** in the first column and have the name of the directive immediately following it. For example, the parallel ES-1 Fortran code for the previous 2D relaxation example is shown in figure 2.4. As before, the compiler directives do not check for any data dependencies between loop data, hence the programmer must ensure that the loop is free of such dependencies. The statement "`c$es local i`" is necessary to specify that each parallel thread spawned by the task has its own local copy of the variable. Unlike the SPS-2 programming model, the ES-1 Fortran does allow nested parallelization of loops. This may limitation may be removed, however, by fusing multidimensional loops into large single-level loop and redistribute the iteration space into a linear array (see section 5.2.3 for details).

2.3.1 Programming Models

This section discusses various programming models used in expressing parallelism in various numerical algorithms. We illustrate these methods using the relaxation example presented in the previous section. A straightforward parallelization scheme replaces each DO loop with a parallel DO loop. Unfortunately this scheme is not successful — parallel loops cannot be nested on the ES-1, and the SPS-2 compiler generates too many tasks (one for each loop iteration). Other partitioning strategies can be explored, however, by decomposing the grid G into T contiguous rectangular subgrids and each of T tasks will be assigned to process a different subgrid. The next section (2.4) discusses various decomposition techniques for 2D grids.

CM-2

On the Connection Machine CM-2, parallelism is expressed implicitly by the use of parallel data arrays in CM-Fortran and C*. The CM-2 utilizes a subset of Fortran 90 (see section 3.1.7 for details) and uses array-based operations to express basic SIMD parallelism. Communication is implicit in several of the intrinsic Fortran 90 array functions. For example, the parallel version of a stencil code example given in the previous section (figure 2.1) is shown in figure 2.2.

The `cshfit()` array function performs implicit processor communication by performing a circular index shift in the given dimension and direction specified by the parameter list. For example, the expression

```
cshfit(A, 1, 1)
```

performs an index shift of the first dimension (row dimension) a unit step in the positive direction. That is, one can visualize taking the each row and moving it donward (+1 in the j-direction is downward.) Array elements moved off the array border get mapped to the top of the array in a cyclic manner. Thus if the array A has its elements defined as

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 32 & 33 & 34 & 35 \end{bmatrix}$$

then

$$\text{cshift}(A, 1, 1) = \begin{bmatrix} 31 & 32 & 33 & 34 & 35 \\ 1 & 2 & 3 & 4 & 5 \\ 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$$

That is, the array elements in each row move in a +1 direction, with the bottom row "wraps around" to the top. This, then, is how communication between processors in performed. Section 3.5 illustrate with more details the specific communication instructions generated by the CM-Fortran compiler for such relaxation code examples. Section 3.1.7 has more discussion on comparisons between sequential Fortran 77 constructs and similar Fortran 90 (CM-Fortran) parallel constructs.

Myrias SPS-2

The basic mechanism for expressing parallelism on the Myrias SPS-2 is the parallel loop, or `PARDO`, construct in its C and Fortran extensions. This is discussed in detail in section 4.1, but here we provide a flavor of the Myrias SPS-2 programming style. A `PARDO` statement is executed by specifying a code segment to be executed and the number of child tasks to be run. Each iteration of a PARDO statement can, in principle, be executed in parallel. For example, the code fragment

```
      do 10 j = 2,n1-1
      do 10 i = 2,n2-1
          b(j,i) = 4.0*a(j,i) + a(j,i-1) + a(j,i+1)
             + a(j-1,i) + a(j+1,i)
10    continue
```

Figure 2.1: Fortran code sample of 2D relaxation step.

by computing vector triads on a single SPS-2 node. Two studies are performed: (1) virtual memory effects using the largest floating point vector (64K) that will fit comfortably in one node, and (2) effects when the vector size is scaled to the domain size (section 4.6).

2.3 Relaxation Performance

Simple relaxation processes for 1D, 2D and 3D Poisson equations are typical of processes occurring in many applications codes involving either elliptic PDE solution or time evolution equations. The code kernels we will describe are essentially those used in relaxation, multigrid and conjugate gradient solution of the Poisson equation. We study two such systems: (1) Poisson equation kernels which have constant coefficients, and (2) more general 3pt, 5pt, and 9pt stencils used in finite differencing schemes and image smoothing techniques.

Although there are significant performance differences among relaxation methods, they utilize nearly identical computational operations. Since the Poisson equation has constant coefficients, the ratio of computational work per grid point to memory traffic is severe, thus illustrating the task granularity synchronization characteristics of parallel architectures.

The basic relaxation kernel we study is

$$b(i) = 2.0 * a(i) + a(i-1) + a(i+1) \tag{2.8}$$

$$b(j,i) = 4.0 * a(j,i) + a(j,i-1) + a(j,i+1) + a(j-1,i) + a(j+1,i) \tag{2.9}$$

$$\begin{aligned} b(k,j,i) &= 6.0 * a(k,j,i) + a(k,j,i-1) + a(k,j,i+1) + \\ &\quad a(k,j-1,i) + a(k,j+1,i) + a(k-1,j,i) + a(k+1,j,i) \end{aligned} \tag{2.10}$$

The multidimensional grid stencils also studied consists of more floating point operations per assignment, thus giving alternate indication of performance variations associated with larger task granularity. The 3pt, 5pt, and 7pt stencil equations consist of

$$b(i) = C_1 * a(i) + C_2 * a(i-1) + C_3 a(i+1) \tag{2.11}$$

$$\begin{aligned} b(j,i) &= C_1 * a(j,i) + C_2 * a(j,i-1) + C_3 * a(j,i+1) + \\ &\quad C_4(a(j-1,i) + C_5 * a(j+1,i) \end{aligned} \tag{2.12}$$

$$\begin{aligned} b(k,j,i) &= C_1 * a(k,j,i) + C_2 * a(k,j,i-1) + C_3 * a(k,j,i+1) + \\ &\quad C_4 * a(k,j-1,i) + C_5 * a(k,j+1,i) + C_6 * a(k-1,j,i) + \\ &\quad C_7 * a(k+1,j,i) \end{aligned} \tag{2.13}$$

where each of the corresponding coefficients C_i may be different at each grid point. Here the arrays are of dimensions N_1, $N_1 \times N_2$, $N_1 \times N_2 \times N_3$, respectively . The equations above are to be applied at each point of the interior of the corresponding 1D, 2D or 3D rectangular grid, which we will denote generically as G. Were the equations to be applied at the boundary of G, then they would index undefined points on the right hand side. This choice of relaxation scheme corresponds to imposition of Drichlet boundary conditions in a PDE solver.

For shared memory programming models, such as those on the ES-1 and the SPS-2, we look at several data partitioning methods of the corresponding grids. Consider the 2D case. We implement the above algorithm by enclosing the expression in a nested set of DO loops, one for each grid direction. For the 2D case the code is given in figure 2.1.

difference methods for solving partial differential equations) are typical of other scientific fields as well.

2.1 Scalar Benchmarks

To measure an estimate for the likely peak machine rates of these systems, we first study several simple scalar benchmarks. These performance numbers are not likely to be sustained in applications, but provide an upper bound on ideal system performance which can be used as a yardstick when determining how well systems achieve their peak in practice.

On MIMD machines, such as the ES-1 and the SPS-2, scalar benchmarks are conducted on a single node in two ways: (1) as a pure sequential program (no parallel constructs) and (2) as a parallel program utilizing only one node. This allows us to see whether that machine's compilers and operating system are "smart" enough to avoid the usual parallel constructs overhead (such as placing tasks on the global run queue for the ES-1, or executing a pardo for SPS-2) when running on a single node. On SIMD parallel processor arrays (such as the CM-2), single node measurement is not as meaningful since each *node* does not represent a full thread of execution.

Each scalar operation is implemented in a loop sufficiently long (e.g. 10,000,000) to ensure that the timing resolution is not causing any perturbations. The scalar operations consist of

$$c = a + c \tag{2.1}$$
$$c = a * c \tag{2.2}$$
$$c = a * c(\text{unrolled}) \tag{2.3}$$
$$c = c + a * c \tag{2.4}$$
$$c = c + a * c \; ; \; a = a + c * b \tag{2.5}$$

Each algorithm is compiled with the highest optimizations available. The unrolled version of $c = a * c$ was obtained by unrolling to a depth of four. Where possible, we measure the performance for single and double precision computations.

2.2 Vector Triad Kernel Performance

Vector triad operations appear in many situations as the low-level computationally intensive core of applications. They are the basic computational units (*saxpy* in the BLAS-1 literature) of the popular LINPACK [19] library of matrix computations. Examples include direct solvers for systems of equations and conjugate gradient iterative solvers for PDEs. As test cases we have used the vector operations

$$a(i) = a(i) + s * c(i), \tag{2.6}$$
$$a(i) = a(i) + b(i) * c(i) \tag{2.7}$$

which we refer to as scalar/vector triads and vector/vector triads, respectively. The s in equation 2.6 denotes a floating point scalar. We perform studies of these operations for single and double precision floating point data.

Although the vector triad kernels do not involve explicit communication they exhibit unique machine-dependent properties. On shared memory systems, like the ES-1, the effects of thread spawning and termination show less than linear speedup (see section 5.3.3). On hybrid distributed/shared memory systems, such as the SPS-2, the effects of memory page merging and virtual memory access are distinguished (section 4.6.2). On the Connection Machine, we will see the effects of virtual processor ratios (section 3.4.3).

The vector experiments also illustrate the effectives of *virtual memory* on the SPS-2. One particular feature of the SPS-2 is that a program computing on only one processor can solve a problem that is too large to fit in a single processor. This virtual memory feature (see section 4.7) stems from the details of the virtual G architecture (section 4.1). The effectiveness of virtual memory on the SPS-2 will be studied

Chapter 2

Numerical Algorithms

Since the aim of this study is to analyze and model the performance of parallel architectures on scientific computing, we need examples of scientific computations. Because this field is large and diverse, we have purposely concentrated on synchronous computations on structured grids with the underlying motivation being that many scientific applications take place on two-dimensional and three-dimensional structured grids. Finite difference solvers for partial differential equations normally require processors to communicate with their local neighbors on the grid. Relaxation processes for elliptic PDEs and hyperbolic time-dependent solvers occurring in computational fluid dynamics and atmospheric modeling exhibit this type of communication pattern when discretized over regular domains. Quantum Chromodynamics (QCD) and mesoscale weather forecasting, for example, are two common areas of scientific computing that utilize rectangular lattice domains for their numerical simulation.

It is certainly true that not every scientific application exhibits these features. There are other important fields that deal exclusively with irregular grids and asynchronous styles of computation. Nevertheless, it is clear that we have no hope of understanding dynamic, irregular systems unless we do not first have a firm understanding of regular, synchronous systems.

In comparing various architectures we have attempted to keep the tests and benchmarks consistent. Although different aspects of each architectural class drive the research in different directions, we have concentrated on performing similar tests across the various machines.

In section 2.1, we discuss simple single node performance benchmarks which provide baseline measurements on upper bound peak performance rates. In section 2.2, we describe performance experiments for simple parallel computations, vector triad operations, which form the low-level computational intensive core of many applications. The communication patterns exhibited by some of these operations yield interesting properties related to page-merging on the SPS-2, thread management on the ES-1, vp ratio effects on the CM-2, and memory traffic effects on the iPSC/860.

In section 2.3 we study simple relaxation processes for multidimensional Poisson equation solvers. These processes are representative of application codes involving either elliptic PDE solution or time evolution equations. By analyzing various ways to partition this problem, several subtle performance effects are brought to light: (ES-1) process granularity, cache effects, data mapping, memory bank conflicts; (SPS-2) page merging, page migration, data layout (subgrid aspect ratio), granularity effects; (CM-2) effects of NEWS communication network, effects of overlapping computation with communication (NEWS/Triads), vp ratio effects, and coprocessor pipelining.

Section 2.6 discusses an important 2D atmospheric application – the solution to a finite-difference version of the Shallow Water Equations. These equations provide a primitive yet useful model of the dynamics of the atmosphere or of certain ocean systems. In choosing areas of applications to be used in this thesis, we have concentrated on models of atmospheric flow and primitive equation models used in numerical weather prediction. These fields have been classically recognized as requiring supercomputing resources for solving very important daily problems of studying the Earth's atmosphere, and their solution techniques (finite

with a local neighbor). This limitation is due to the fact that each node can handle only one incoming and one outgoing message at a time (see section 6.3.3).

Dongarra & Ostrouchov [23] have studied the communication costs involved in block factorizations on the iPSC/860. Using hand-coded BLAS3 subroutines, they were able to achieve over 1.4 Gflops for a pipelined LU factorization on 8,000 x 8,000 matrices. Scott, et. al. [83] have also demonstrated such performance can sustain on even larger 20K x 20K matrices using the out-of-core solvers that utilize Intel's Concurrent File System (CFS).

1.4 Related Work

1.4.1 Connection Machine CM-2

Much of the early performance analysis work on the CM-2 was not aimed at generating detailed models, but at reporting basic benchmarking and timing information on various applications or on the low-level Paris and *LISP instructions. No work really provided a thorough analysis of underlying architecture issues or attempted to model performance behavior. Myers & Adams [69], for example, provide timing information in the form of a long table of various integer and floating operations. No study was made of the vp ratio effect for these operations.

An earlier communication model for the CM-2, proposed by Levit [55], focused on sending fixed-sized messages over power-of-two distances. Our model is an extension to this work. By modeling virtual processor communications (t_M) and geometries which bypass physical communications ($o_i = 1$), we have been able to significantly improve the prediction accuracy.

Chen, Choo & Jacquemin [16] discuss similar communication models for the CM-2 . They provide an example of how accurate communication models can help optimizing compilers rearrange distributed data structures to improve performance. Because dependencies among distributed arrays often translate into processor communication, their goal is to determine the allocation of arrays that minimizes the total communication cost. They have developed techniques where given the parallel arrays and their dependencies, linear transformations are applied to the domains of the arrays to produce the best alignment that reduces the total communication costs.

1.4.2 Myrias SPS-2 and Evans & Sutherland

Because both the Myrias SPS-2 and Evans & Sutherland have had limited commercial exposure, relatively little has been written about them in the literature.

Baker & Wilson [4] have reported on usage of the Myrias SPS-2 at the University of Calgary in Alberta, Canada. Their report on computer utilization is written from the system administrator's point of view and is not focused on performance modeling per se, but on system utilization over load averages.

Grimrud, et. al., [33] have provided an overview description of the Evans & Sutherland ES-1 and describe the underlying architecture although there is no specific performance analysis made of this machine.

1.4.3 Intel iPSC/860

There have been several communication models proposed for the iPSC/860; however, most models are based on isolated point-to-point communications where only a few processors in a machine are actively engaged in communication. In those cases, there is little contention, hence the communication performance is quite favorable. Unfortunately, such models do not adequately capture the more complex, yet common, cases where *all* processors (or at least a significant percentage of them) are engaging in communications. One of the simplest examples occurs in Jacobi-like relaxation kernels, where processor boundary information must be exchanged with neighboring subgrids. In such cases the processors are effectively engaging in a circular shift operation.

Other communication studies include McCreary et. al.[65], who have conducted experiments for broadcast communication patterns on the iPSC/860. Bokhari [9] has conducted point-to-point communication experiments for multihop communications and has noted that the effect of the number of hops (hamming distance) between two nodes has an insignificant effect ($< 2\%$ in most cases) on the communication latency. This has been also noted by Bomans [10] and McCreary [65]. Contention of edges, however, can be considerable (Bokhari [9], Bomans & Roose [10].)

The work of Seidel, et al.[84] studies bidirectional communication on pairwise exchanges between isolated nodes. Although each channel of an iPSC/860 node can support bidirectional communication (5.6 MB/sec), it is not possible to achieve this for "circular" exchanges (that is, where each node exchanges information

1.3.4 Intel iPSC/860

The Intel iPSC/860 is a distributed memory MIMD hypercube developed from the earlier iPSC/1 and iPSC/2 prototypes. The iPSC/860, often referred to as the Gamma prototype, is part of the Intel's Touchstone family of computers that will target parallel architectures towards 100 Gflops by 1995 [56]. The iPSC/860 consists of up 128 nodes with a worm-hole routing message passing communication system. Nodes can be fitted with 8 to 32 Mbytes of memory and a routing module (DCM) capable of generating up to 2.8 Mbytes/sec transfer rate between the hypercube nodes.

Each node of the iPSC/860 consists of an Intel 40 MHz i860 RISC microprocessor, capable of 40 MIPS and 80 Mflops. We have measured various vector routines, such as BLAS1 saxpy, at over 40 Mflops (section 6.1) while other studies have shown this microprocessor can sustain over 40 Mflops for matrix multiplication and Fast Fourier Transform (FFT) routines [59]. Compiler generated code, however, does not exhibit such performance; similar operations perform at only 3-5 Mflops. In section 6.1, we study the code generated by various compilers and provide a critique as to why the performance is less than expected.

In section 6.3, we present communication models for the iPSC/860. We compare our work to previous communication models which successfully described isolated "point-to-point" cases but had difficulty describing the behavior when several or all processors simultaneously participate in communication patterns typical of iterative solvers of explicit systems.

When we first studied the communication performance of the shallow water application on the iPSC/860, the message latency between nodes was roughly double of what conventional communication models predicted. After a careful modeling and analysis of the underlying message passing protocol, we were able to improve application communication performance by a factor of two.

Although it is impossible to characterize completely random or nondeterministic communication patterns, many of the communication patterns used in relaxation techniques and the shallow water equations can be modeled rather accurately. This is possible because of the synchronicity of these algorithms.

There are complications, however, due to the handshaking protocol imposed by the operating system for long (> 100 bytes) messages. Because of the extra traffic generated by system messages, the communication bandwidth for simple synchronous operations, such as a circular shift or relaxation kernels, can cause severe performance degradation. To remedy this situation, we have designed new communication algorithms which bypass the operating system protocol and generate "forward only" synchronization so that the full communication bandwidth can be obtained.

In section 6.5, we utilize the models developed in the earlier sections to predict the performance of the Shallow Water application quite accurately. We see that performance scales nicely, yielding nearly 95% efficiency for scaled applications up to 128 processors.

difficulties encountered in explicit data mapping and synchronization for conventional distributed memory systems.

Parallelism in the SPS-2 is based on the basic process task model. The native operating system copies data spaces to child tasks and merges pages of data upon task completion that have been written into by multiple processes. The migration of pages is demand driven and completely dynamic.

This immediately raises questions about how distribution of parallel data objects within the machine affects machine performance. Even operations which normally do not entail communication, such as parallel vector operations, exhibit page merging effects due to the overlapping of distributed data objects across memory page boundaries.

In chapter 4, we describe the underlying architecture of the SPS-2 and the details of its unique operating system, including the page merging models, task spawning, and task assignments. After investigating the computational characteristics of individual nodes and conducting various baseline experiments, we analyze the virtual memory and implicit communication effects introduced by the programming model. In section 4.6.5, we design linear performance models which describe the multiprocessor overhead and task granularity effects for page merging of distributed vector operations which straddle page boundaries. These models are over 95% accurate for the interesting domain sizes ($10 < p \leq 64$).

In section 4, we study several mapping schemes to reduce the volume of page merging and message traffic for relaxation processes for 2 and 3 dimensional grids. By varying the size and shape of each processor subgrid, we can control the amount of boundary information that requires merging. We also investigate several blocking schemes for 2d grids (index, subroutine, column striping) and illustrate the subtle performance tradeoffs between these methods. Finally, in section 4.9 we discuss the performance of the Shallow Water application on the SPS-2.

1.3.3 Evans & Sutherland ES-1

The Evans & Sutherland ES-1 was one of the first massively parallel shared memory multiprocessors. The design of the ES-1 allowed for up to 128 processors to share a global address space. Although no longer commercially available (E & S cut off their supercomputing project just as the machine was completed), several prototype systems were installed and used regularly until the lack of software and hardware maintenance and support made their continued operation unfeasible. Nevertheless, the ES-1 provided a unique research testbed to study and analyze the characteristics and behavior of massively parallel shared memory systems.

A fundamental difference between this machine and others included in this study is that parallelism is specified by the number of tasks, rather than the number of processors. Thus, the programmer has no explicit control over the number of processors to use in an application nor does he have any control over the mapping of data to individual nodes. This leads to interesting challenges for the performance modeler to adequately describe the machine's behavior.

The ES-1 was slated as a large computing resource. The full system configuration included 2 GBytes of memory, 8-450 GByte disk capacity, and aggregate peak speeds of 1600 MIPS and 1600 Mflops. Our studies revealed that each processor produced more like 3-6 Mflops for compiled code; however, the efficiency and scalability was quite good. Even for the Shallow Water application, we saw efficiencies of over 80% for 12 processor configurations.

In chapter 5 we provide a detailed description of the underlying hierarchical crossbar architecture, followed by a description of the native Mach operating system. We discuss the basic synchronization primitives and parallel constructs of the programming model. In section 5.3.1 we discuss the relationship between tasks and processors and describe the details of the global and local run queues maintained by the operating system. This mechanism controls task migration and hence is important in studying the dynamics of data locality. After describing the relevant details of the programming environment we focus on performance of 1D, 2D, and 3D relaxation processes and other kernel vector operations. We investigate various ways to distribute grids and assign the subgrid aspect ratio to individual processors. We conclude by describing the performance of the Shallow Water application on this architecture.

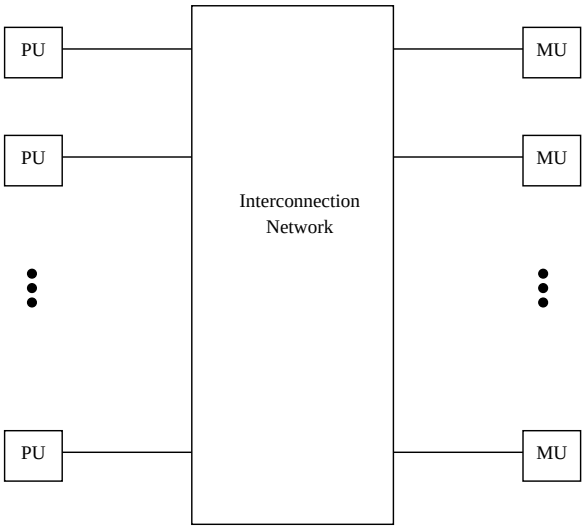


Figure 1.4: Shared Memory Multiple Instruction Multiple Data (MIMD) parallel architecture.

boundary surfaces of per-processor subgrid, we can formulate accurate models that predict communication latency to within 5% for nearly all 1D, 2D, and 3D virtual processor configurations.

In section 3.4, we discuss the underlying floating point hardware and conclude that its performance behavior may be modeled as a pipelined vector unit except that the role of the "vector length" is played by the virtual processor ratio (sections 3.4.1 and 3.4.2). Various experiments are conducted in section 3.4.3 to establish the pipeline length and overhead, and accurate performance models are described that predict floating point performance to within 2% error. We also discuss the important aspect of overlapping communication with computation (section 3.4.4).

In section 3.5, we discuss a performance modeling software prototype, the Connection Machine Performance Simulator (CMPS), that works with the C* and CM-Fortran compilers to produce profiler-type information. The system maintains an internal database of operations used within an application and uses `awk` and shell scripts to keep track of application performance information.

Finally, in section 3.7 we illustrate how these low-level models can be integrated to provide accurate performance predictions of the Shallow Water equations over various virtual processor geometries with less than 10% total error.

The native programming model of the CM-2 is termed Paris (PARallel Instruction Set) and consists of various atomic instructions (section 3.1.8) that direct processors to perform elementary boolean, floating point, integer, and communication operations. Recently, a transitional programming model has been introduced, the the so-called "slice-wise" programming model, which distributes atomic data values (such as a double precision float) across several physical processors. This reduces the role of glue chips that interface the physical processors to the floating point units (see section 3.4.1) and improves floating point performance. This programming model, however, is available only for double precision hardware and hence not available on many CM-2 systems. The Paris programming model described here more closely abstracts a true SIMD architecture and is available on all CM-2s.

1.3.2 The Myrias SPS-2

The SPS-2 was one of the first distributed memory MIMD machines to support virtual shared memory constructs in its native operating system. Although no longer commercially available, the Myrias SPS-2 has incorporated some rather successful features that will resurface in future machines. The ability to provide a shared memory environment facilitates rapid prototyping of conventional software and removes the usual

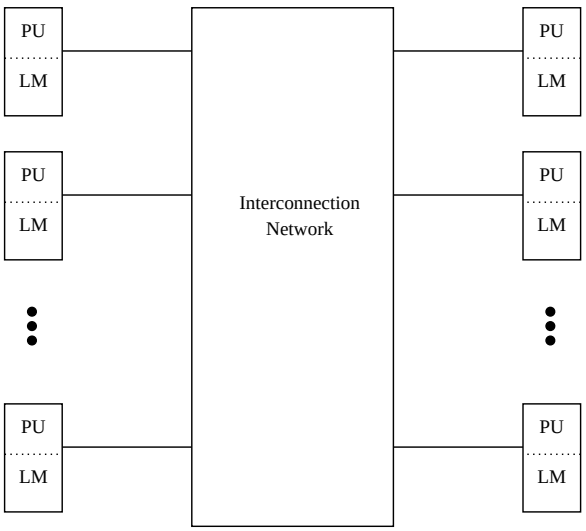


Figure 1.3: Distributed Memory Multiple Instruction Multiple Data (SIMD) parallel architecture.

processors (less than 16) it can be possible to have a full crossbar topology. The Cray supercomputers and the IBM 3090 have such features.

1.3 Parallel Systems Studied

Demonstrating the usefulness of such multilevel performance models requires applying them to concrete examples of existing parallel machines. To provide a fairly complete picture of the architectural landscape, we perform such analysis on machines representative of the major architectural categories:

1. Connection Machine CM-2, an SIMD distributed memory processor array,
2. Myrias SPS-2, an MIMD shared/distributed memory hybrid,
3. Evans & Sutherland ES-1, a RISC-based shared memory parallel processor
4. Intel iPSC/860, a message-passing distributed memory MIMD hypercube.

Although our research has focused on the more popular machines, the Connection Machine CM-2 and the Intel iPSC/860, we have also included analyses of Myrias SPS-2 and the Evans & Sutherland ES-1 to provide a more complete picture of architectural landscape. An introduction of these machines is given in section 1.3.

1.3.1 Connection Machine CM-2

The Connection Machine CM-2 has been widely used in massively parallel computing. Consisting of up to 65,536 SIMD processors, the CM-2 is capable of delivering Gflop performance for various scientific applications. In section 3.1, we provide an overview of the underlying CM-2 architecture and software environment. One important feature of the CM-2 is the ability to configure the machine to fit the problem. By having each processor simulate many virtual processors, one may decompose a problem grid or array so that each array corresponds to a different virtual processor. This facilitates the programming tremendously, since the programmer sees only a one-level data decomposition. (Contrast this with the iPSC/860 discussed below and in detail in chapter 6). In section 3.3 we present NEWS communication models for the CM-2. By analyzing

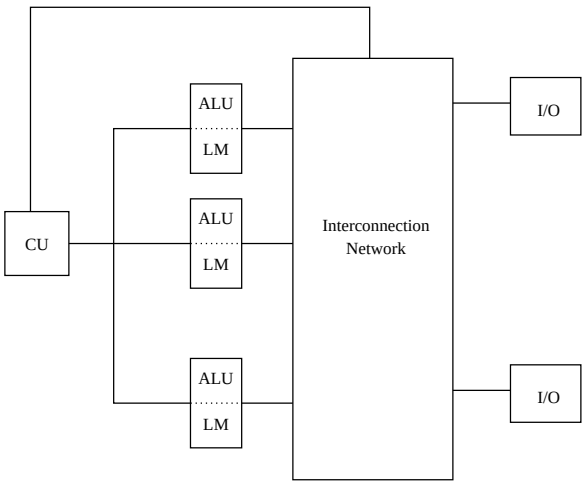


Figure 1.2: Single Instruction Multiple Data (SIMD) parallel architecture.

1.2.1 SIMD

In Single Instruction Multiple Data stream (SIMD) parallel architectures each processing unit executes the same instruction. (See figure 1.2.) In other words, there is only one program counter (PC) dictating program control flow. Each processing element consists of Arithmetic Logic Unit (ALU) and some local memory (LM). Here a computational unit (CU) maintains control flow and issues instructions to all processing elements (ALU/LM pairs). Sometimes this architecture is also referred to as a "processor array." The individual processing elements are connected via an interconnection network. The most popular choices are hypercubes, and 2D meshes. The CU plays the role of a "host" which issues the instructions; it has direct access to the interconnection network, input/output (I/O) channels, and its own memory. Machines that fall under this category include the Connection Machine CM-2, the MasPar MP-1, and the AMT DAP.

1.2.2 Distributed Memory MIMD

In distributed memory Multiple Instruction Multiple Data (MIMD) parallel architectures, each processing unit (PU) has its own local memory (LU) and executes instructions independently of other processing units. (See figure 1.3.) The PU can be thought of as an independent computer in that it has its own program counter (can follow its own control flow), and ALU units. PUs do not have direct access to memory on other processing units. To request a copy of the memory contents of some other PU in the system, an explicit communication must first take place. There are various possibilities for the interconnection network: linear arrays, rings, stars, trees, meshes, hypercube, cube-connected cycles, omega and butterfly networks, etc. However the most commonly used interconnection network for distributed memory MIMD machines is the hypercube, with a recent focus on 2D meshes. Examples of current distributed memory MIMD machines that have been built include the Intel iPSC/860, the NCube, the Ametek 2010, and FPS T Series.

1.2.3 Shared Memory MIMD

In shared memory systems, all processors share a common global memory address space. (See figure 1.4.) Here each processing unit can access any of the memory units (MU) through an interconnection network. It is quite often that each PU has its own cache or semi-private local memory. In such cases, some care must be taken by the system to insure cache coherency. Various network topologies have been used to connect the processors to the memory units: the Evans & Sutherland ES-1 used a hierarchical crossbar topology (see section 5.1), both the Alliant FX/8 and FX/2800 used bus topology. For systems using a small number of

Table 1.2: Performance of Advanced Architectures

- Connection Machine CM-2
  - data decomposition onto hypercube network
  - overlapped communication/computation
  - virtual processors
- Myrias SPS-2
  - page merging effects
  - data layout and migration
- Evans & Sutherland ES-1
  - memory bank conflicts
  - caching effects
  - RISC pipelined core
- Intel iPSC/860
  - communication effects
  - granularity, data decomposition
  - load balancing
- Intel i860
  - parallel microarchitecture
  - cache management
  - dual-instruction and dual-operation
  - prefetching in pipelined loads

subject of this work. These four machines (the Connection Machine, Evans & Sutherland ES-1, Myrias SPS-2, and Intel iPSC/860) are introduced in the next section.

1.2 Parallel Architectures

This section presents a brief overview of some of the basic parallel architectures. More complete tutorial introductions can be found in the standard texts, e.g. Almasi & Gottlieb [3], Hwang & Briggs [50], Hockney & Jesshope [44], Hennessy & Patterson [41], Stone [90]. Breifly, parallel architectures can be be broken down into several categories. Among the important differences are (1) how the multiple units are connected (connection topology), (2) whether each unit has its own instruction stream (MIMD vs. SIMD), and (3) whether the processors share a global memory pool or have their own private memory space (shared memory vs. distributed memory.) Near all commercial machines are one of three types: (1) Single Instruction Multiple Data (SIMD), (2) Multiple Instruction Multiple Data Stream with shared memory (Shared Memory MIMD) and (3) Distributed Memory MIMD systems.

Table 1.1: Performance of Shallow Water Equations

Performance of Shallow Water Equations			
Machine	Processors	Grid Size	Mflops
CM-2	65,536	8096x8096	1714.00
CRAY-YMP	8	512x512	1532.00
CRAY-XMP	4	512x512	560.00
iPSC/860	128	1024x2048	543.00
SUPRENUM-1	16	512x512	72.00
Ncube II	32	1024x1024	32.30
Alliant FX/8	6	256x256	22.10
Micro i860	1	128x128	6.80
SPS-2	64	1024x1024	2.70
MIPS	1	128x128	1.43
SUN Sparc 1	1	256x256	1.39
386 PC	1	128x128	0.15
SPS-2	1	128x128	0.12

Such methods have been recommended to National Academy of Sciences [70], setting guidelines and standards for supercomputer evaluation. Nevertheless, most performance studies presented in the literature still address only one or two of these levels without really providing a complete view of the complex performance personalities exhibited by advanced parallel architectures.

The results of multilevel performance analyses can be used in several ways. First, the preliminary studies of the low-level computational kernels can be used to develop performance models for various machines. These performance models can then be used to hypothesize how the next higher-level computations will behave. Integrating the analysis and modeling from several computational levels provides a more complete understanding of a parallel computer's performance characteristics.

To illustrate these points, consider the benchmark results for an atmospheric simulation, the Shallow Water equations, on various computer architectures as shown in table 1.1. We provide a detailed discussion of the Shallow Water equations and some of these architectures in the next few sections, but for the moment we wish to address the interpretation of performance numbers. Although such performance tables are often useful for obtaining rough estimations, they provide only a static snapshot and yield very little information about how these architectures behave under different situations. Even though each architecture is running the same mathematical algorithm, there are various ways to code the algorithm and express possible vectorization and/or parallelization. Furthermore, there are various ways to describe the mapping of distributed data objects onto a machine, and to direct communication for message passing systems. For example, increasing the problem grid size by one in each dimension, causes the CM-2 performance to drop by a factor of four; interchanging the order of nested loops can severely cripple the performance on Crays; using an alternate version of the communication primitives can effectively double communication performance on the iPSC/860; remapping matrices into block can increases cache utilization and significantly effect performance of the Alliant FX/8.

The point is that, unlike scalar uniprocessors, the performance exhibited by an advanced parallel architecture can be extremely sensitive to the way in which a problem has been implemented. Parallel architectures can behave more like chaotic systems, in the sense that a small perturbation in the program implementation (input) can have a dramatic impact on system performance (output). Figure 1.2 illustrates some of the implementation considerations that effect overall performance for the four parallel systems that form the

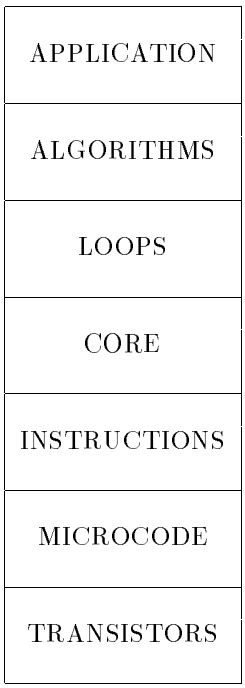


Figure 1.1: Hierarchical levels of scientific computation on computer hardware.

in figure 1.1. The different levels in this figure represent the various layers that lie between the mathematical problem statement and its solution (execution) on a digital computer. At the top level, is how a mathematician, scientist or engineer specifies their problem; at the bottom level, is how the digital bits flow through the transistors and electrical devices that latch address buses and perform boolean logic at the basic circuitry level to solve that problem. In between, there are various intermediate levels. Scientific applications generally consist of several numerical algorithms (matrix factorizations, multigrid, FFT's, finite difference and finite element solvers, etc.) with each algorithm containing certain fundamental loops that run over a matrix or grid. These form core vector/matrix operations, which in turn consist of individual floating point instructions that execute a certain sequence of microcode and control the underlying hardware.

In the multilevel view, there is a considerable distance between the application and transistor level. One of the main points of this thesis is that studying machine performance at only one particular level does not provide a complete view of performance capabilities. Computational kernels alone do not capture the complexity of data access and computation patterns exhibited by large scientific code; alternately, analyzing large applications without an understanding of how simple routines contribute to overall performance characteristics limits what we can learn about the effectiveness of these machines. In short,

- single level benchmarks do not reveal the complete performance picture that captures subtle characteristics of parallel supercomputers.
- high level applications benchmarks alone have limited value since it is difficult to analyze the complete source code and determine the influence of individual factors.
- low level benchmarks alone may give one rudimentary knowledge about a parallel machine, but simple loops often do not exhibit the complexity of computational patterns of real applications.

Indeed, the importance of analyzing supercomputer performance at several computational levels has been repeatedly advocated by many of the leading researchers in this field (Dongarra [20], Kuck [8], Wilson [103]).

Chapter 1

Introduction

The primary motivation for the existence of the parallel computing field can be characterized by a single word – *performance.* Large scale scientific computing problems typically utilize tremendous computing resources and the ability to solve these problems in a reasonable time frame requires supercomputers with ever increasing computational speeds. Traditionally, these applications have been implemented on single processor machines but the speeds of such conventional uniprocessor machines are quickly approaching their physical limitations. Experts agree that a viable approach to solving challenging problems in aerodynamics, climate modeling, structural analysis, biochemistry, particle physics, medicine, vision, and other important fields is to harness the power of many processors in parallel to overcome the physical performance barrier of uniprocessor computers.

Because performance is the primary motivation for parallel processing, performance analysis and modeling is the important yardstick by which we measure how well parallel systems achieve their goals. Characterizing the performance of parallel supercomputer architectures has become an increasingly complex task. Several years ago, performance of numerical applications on large systems concerned itself with issues related mainly to pipelined vector processing, such as identifying vectorizable loops within scientific code. With the advent of truly parallel systems, however, several dimensions of complexity have been added to the list of performance considerations: task granularity, synchronization issues, domain decomposition, locality of reference, communication effects, and resource contentions. Often, a relatively small change in the program specification or problem description (such as altering the order of nested loops, or even rearranging expressions) can have a tremendous impact on performance. Traditional performance analysis techniques are simply not robust enough to capture the increased complexity exhibited by these computer systems.

In addressing this problem, we introduce new multilevel performance models and analyses which characterize computation and memory utilization at the instruction, loop, algorithm, and application level. We demonstrate that these approaches are necessary to adequately evaluate distributed and shared memory systems. By closely modeling the underlying architectures, communication, and memory systems, we identify machine characteristics that capture the performance behavior exhibited by numerical computations such as matrix/vector operations, relaxation techniques, and PDE applications on various parallel architectures.

The information provided by these performance models is not just limited to the end-user who wishes to analyze the running time of a program; this information is also useful to algorithm designers, compiler writers, hardware and firmware architects. For example, by having a parameterized performance model of the communications network, a hardware architect can investigate various component designs and have a characterization of its impact at the application level.

1.1 Multilevel Performance Models

The fundamental notion of multilevel performance models is based on the fact that numerical and scientific computation occurs at various computational levels. Consider a typical computational hierarchy, as shown

5.3.5 Variable Aspect Ratio Grid decompositions . . . 111
5.4 Implementation and Performance of Shallow Water Application . . . 115

6 Intel iPSC/860 Hypercube 119
6.1 Processor Architecture: Intel i860 . . . 121
6.1.1 Memory System Architecture . . . 121
6.1.2 Caching and Memory Paging Effects . . . 123
6.2 Computation Modeling of the Intel i860 . . . 125
6.2.1 Fortran Performance . . . 125
6.2.2 Pipelined Performance . . . 126
6.2.3 Intel i860 Performance Modeling . . . 132
6.3 Communication Architecture of the iPSC/860 . . . 136
6.3.1 Gray Coding and E-cube Message Routing Algorithm . . . 136
6.3.2 Message Passing Protocol . . . 136
6.3.3 Channel Limitations of the iPSC/860 . . . 136
6.3.4 Buffering of Messages . . . 138
6.4 Communication Model . . . 139
6.4.1 Linear Models . . . 139
6.4.2 Sequential Ring Experiment . . . 139
6.4.3 Circular Shift Experiment . . . 140
6.4.4 A Forward Synchronization Model . . . 145
6.5 Shallow Water Equations . . . 148
6.5.1 Implementation Issues . . . 148
6.6 Performance Analysis of Shallow Water Equations . . . 150
6.6.1 Results of Single Node Performance . . . 150
6.6.2 Multiple Node Performance . . . 154
6.6.3 Performance Model . . . 158
6.6.4 Speedup and Efficiency . . . 159

7 Conclusion 165
7.1 Connection Machine CM-2 . . . 165
7.2 Myrias SPS-2 . . . 167
7.3 Evans & Sutherland ES-1 . . . 169
7.4 Intel iPSC/860 . . . 170
7.5 Final Thoughts . . . 171

3.2.1 First Experiments . . . 50
3.3 A Grid Communication Model . . . 50
3.3.1 Extending the Model to Power-of-Two Distances . . . 55
3.3.2 Accuracy of NEWS Communication Model . . . 55
3.3.3 Related Work . . . 58
3.4 Modeling Floating Point Operations . . . 59
3.4.1 CM-2 Floating Point Architecture . . . 59
3.4.2 Basic Model for Floating Point . . . 59
3.4.3 Observed Measurements . . . 60
3.4.4 Overlapping Communication with Computation . . . 61
3.5 Putting the Models to Work . . . 66
3.5.1 Relaxation Example . . . 66
3.6 CMPS . . . 70
3.6.1 Overview of Functionality . . . 70
3.6.2 Compile Phase . . . 72
3.6.3 Link Phase . . . 74
3.6.4 Run Phase . . . 74
3.6.5 Analysis . . . 76
3.7 Shallow Water Application Example . . . 80
3.7.1 CMPS Analysis . . . 80

4 Myrias SPS-2: A Distributed/Shared Memory Hybrid System 85
4.1 SPS-2 Software and Programming Model . . . 85
4.2 The PARDO construct . . . 85
4.3 Task Memory Merging Rules . . . 86
4.4 Use of Recursion . . . 86
4.5 Programming Environment . . . 87
4.6 Performance Analysis of the Myrias SPS-2 . . . 88
4.6.1 Scalar Performance . . . 88
4.6.2 Computational Core Performance . . . 88
4.6.3 Single Node Performance . . . 89
4.6.4 Single Processor Access to Virtual Memory . . . 89
4.6.5 Modelling Multiprocessor Overheads and Task Granularity . . . 90
4.7 SPS-2 Relaxation Performance . . . 92
4.8 Implementations of Block Ordering . . . 98
4.9 Shallow Water Equations . . . 102

5 Evans & Sutherland ES-1: A Shared Memory MIMD System 103
5.1 Hardware Overview . . . 103
5.1.1 Hierarchical Crossbar Architecture . . . 103
5.1.2 Node Processors . . . 103
5.1.3 Memory Hierarchy . . . 105
5.2 Programming Environment . . . 105
5.2.1 ES1/Mach Operating System . . . 105
5.2.2 Programming Environment . . . 106
5.2.3 ES-1 Parallel Fortran Compiler Directives . . . 106
5.3 Performance Analysis of the ES-1 . . . 108
5.3.1 Relation between Tasks and Processors . . . 108
5.3.2 Single CU Scalar Performance . . . 108
5.3.3 Vector Triad Performance . . . 109
5.3.4 Parallel Relaxation Performance . . . 110

Contents

1 Introduction **7**
- 1.1 Multilevel Performance Models . . . 7
- 1.2 Parallel Architectures . . . 10
  - 1.2.1 SIMD . . . 11
  - 1.2.2 Distributed Memory MIMD . . . 11
  - 1.2.3 Shared Memory MIMD . . . 11
- 1.3 Parallel Systems Studied . . . 12
  - 1.3.1 Connection Machine CM-2 . . . 12
  - 1.3.2 The Myrias SPS-2 . . . 13
  - 1.3.3 Evans & Sutherland ES-1 . . . 14
  - 1.3.4 Intel iPSC/860 . . . 15
- 1.4 Related Work . . . 16
  - 1.4.1 Connection Machine CM-2 . . . 16
  - 1.4.2 Myrias SPS-2 and Evans & Sutherland . . . 16
  - 1.4.3 Intel iPSC/860 . . . 16

2 Numerical Algorithms **19**
- 2.1 Scalar Benchmarks . . . 20
- 2.2 Vector Triad Kernel Performance . . . 20
- 2.3 Relaxation Performance . . . 21
  - 2.3.1 Programming Models . . . 23
- 2.4 Partioning of Two Dimensional Grids . . . 26
  - 2.4.1 Distributed Memory . . . 26
  - 2.4.2 Shared Memory . . . 29
- 2.5 Parallel Fortran Preprocessor . . . 32
- 2.6 Shallow Water Equations . . . 36
- 2.7 Overview . . . 43

3 Connection Machine CM-2 **45**
- 3.1 Overview . . . 45
  - 3.1.1 Processors . . . 45
  - 3.1.2 Processor Memory . . . 46
  - 3.1.3 Processor Communication . . . 46
  - 3.1.4 Virtual Processors . . . 46
  - 3.1.5 Floating Point Hardware . . . 47
  - 3.1.6 Language Support . . . 47
  - 3.1.7 CM Fortran . . . 47
  - 3.1.8 The Paris Programming Model . . . 48
- 3.2 Mapping N-dimensional Grids . . . 49

Abstract

Performance modeling and analysis is essential to assess the effectiveness of parallel and pipelined architectures in scientific computing applications. Traditional performance analysis has focused on separate levels of computation (e.g. instruction, loop, algorithm, or application) but such approaches often do not a provide a complete picture of the performance behavior. New multilevel performance models which integrate these various levels are necessary to adequately evaluate distributed and shared memory systems. We demonstrate this with concrete architectural examples – four computing systems, each representing one of the major parallel architecture classes: a distributed memory SIMD system (the Connection Machine CM-2), a shared memory MIMD system (Evans & Sutherland ES-1), a shared/distributed memory hybrid system (Myrias SPS-2), and a distributed MIMD system (Intel iPSC/860.) Using an integrated multilevel approach we can accurately describe the behavior of matrix/vector operations, relaxation techniques, and PDE applications on these architectures. To demonstrate the applicability of multilevel performance models, we utilize an application example of a simple fluid code describing atmospheric and oceanic systems and show how one can predict application performance to within ten percent error for all data decomposition methods over various parallel architectures.

Performance Modeling of Parallel Architectures for Scientific Computing

Roldan Pozo
Center for Applied Parallel Processing
Department of Computer Science
University of Colorado, Boulder

January 22, 1992