Compilation of Scientific Programs into Multithreaded and Message Driven Computation

John Holm Antonio Lain Prithviraj Banerjee

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

Many programs written in the SPMD programming model send messages asynchronously, and block when receiving messages. Multiple threads can make use of the processor while other threads wait for messages. This paper describes and evaluates two techniques for multithreading on the nodes of distributed memory message passing systems. One method is a purely runtime threads package. The second method requires the SPMD code to be systematically transformed into message driven code which can be run under a message driven model. The multithreading of scientific applications is evaluated on the iPSC2 and the CM5.

1 Introduction

Distributed Memory Multicomputers (DMM's) offer the advantage of easy scalability over shared memory systems, but the lack of a global address space makes DMM's more difficult to program. One of the simplest models used to program DMM's is the message passing Single Program Multiple Data (SPMD) programming model. While many programs written in this model send messages asynchronously, most programs block when receiving messages. This is especially true for compilers that generate message passing code from sequential code [1, 2, 3]. Depending on the application, this can lead to significant idle cycles while the processors wait for messages [4]. If more than one thread is run on a node at the same time, then another thread can be swapped in while the processor is waiting for the message. This paper describes and evaluates two techniques for multithreading on the nodes of a DMM message passing system. The first method is a purely runtime threads package. The second method uses a message driven model in which SPMD code needs to be transformed into message driven code using compiler based transformations. The performance of both methods is evaluated on a 16-node iPSC2 hypercube and on the CM5 on several scientific applications.

This paper presents multithreading as part of the PARADIGM compiler project at Illinois [1]. PARADIGM is targeted to generating efficient message passing code from sequential applications. Its unique features are:

- Automatic generation of data partitioning [5, 6].
- Automatic generation of high level communication [7].
- Automatic exploitation of functional and data parallelism [8, 9, 10].
- Support for irregular computations [11].
- Use of multithreading to overlap computation and communication.

The next section discusses the background of multithreading and message driven computation. Section 3 discusses the methodology and run-time systems required by each package. Section 4 presents the performance improvement which results from multithreading. Section 5 concludes.

2 Background

Although there has been much work on multithreading in multicomputers [12, 13, 14, 15, 16, 17, 18],

*This research was supported in part by NASA under contract NASA NAS 1-613, the ONR under contract N00014-91J-1096, and in part by an AT&T graduate fellowship and a Fulbright/MEC fellowship.

little effort has been done on integrating multithreading with a distributed memory compiler. Most distributed memory compilers assume an SPMD model of computation with only one thread per node [19, 20].

Early work on multithreading centered around operating systems in which context switching was used for time sharing and to hide I/O latency. In multitasked operating systems, the processes are usually independent and context switching time is usually orders of magnitude smaller than time slices and I/O time [21]. More recently, multithreading has been used in shared-memory machines to avoid idling when the cache misses. Since cache misses can occur as frequently as one every few cycles [22], special hardware support is required to support fast context switching [23]. Some processors used in DMM's also provide support for multithreading including the Transputer [12], the START project [17], Monsoon [15], and the Message-driven Machine [16]. The Transputer and START both provide processor level hardware support for context switching and threads in general purpose computations. Monsoon and the Message-driven Machine require extensive hardware support to make the very fast context switches required to support functional languages or some object oriented languages.

In scientific applications written with more conventional languages, the message inter-arrival times on DMM's is several hundred cycles [4]. In this case, multithreading on DMM's does not necessarily require the special hardware support for very fast context switches, but it still can not absorb the cost of heavy-weight operating system context switching. The purpose of this paper is to show that using software techniques, it is possible to improve the performance of programs on DMM's using multithreading.

Threads packages that run as user code are a good alternative to operating system context-switching in multi-threaded applications [24, 25, 26]. Context-switching is handled entirely by the application and the amount of state that need to be saved and restored is smaller. Recent trends of light-weight threads packages try to get some advantage of the operating system as well using a hybrid approach [27]. Most of these packages provide a flexibility that we do not need in our basic threads model. Due to its simplicity, CSIM [25] was chosen as our threads package.

Other software methods for multithreading include Active Messages [14], Chare Kernel [18], Chores [28], and NewThreads [13]. Active messages are the most similar to our message driven scheme described in this paper because it uses existing hardware and keeps some information about firing the thread within the message. The Chare Kernel [18] is a new language for dynamically creating and scheduling chares across processors on both shared and non-shared memory machines and is suitable for problems that involve tree searches. The Chores system provides a message driven model, but only runs on shared memory systems. The message driven model produced by transforming SPMD codes does not require the extensive support these packages provided. We use our own simple, portable user level run time system to support our model.

3 Methodology

This section describes two different techniques to integrate multithreading into message passing SPMD code. The first one relies on a run-time support package to support fast context-switching between user-level threads. The second multithreading system uses a compiler transformation from SPMD code to message driven code. The transformation reduces the complexity and increases the portability of the system. In both cases, the grid of virtual processors (SPMD Threads) is mapped onto a smaller grid of physical processors using block distribution.

3.1 Basic Threads Model

Figure 1 shows the basic threads model of computation. Threads 1 and 2 are SPMD Threads mapped

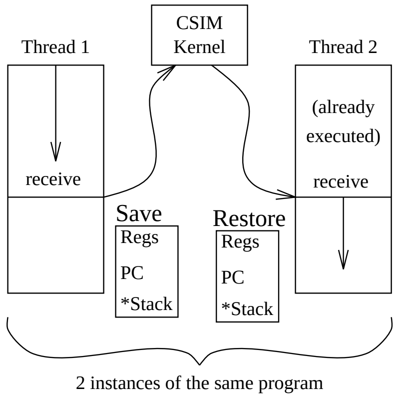

Figure 1: Basic Model of Threading.

to the same processor. When Thread 1 executes its receive it first checks if the message has arrived. If the message has arrived it will continue execution, otherwise CSIM [25] will:

1. Save Thread 1 state.
2. Post an interrupt handler that will enqueue Thread 1 in the ready queue when its message arrives.
3. Restore Thread 2, a thread already in the ready queue.
4. Resume execution of Thread 2 after its receive.

Saving and restoring the state of a thread should be fast. In the original version of CSIM this was an expensive operation because both the stack and the registers, were saved during a context-switch. CSIM was modified to avoid saving the stack and to handle network interrupts. These modifications reduced the time to context-switch, post an interrupt handler and resume another thread to less than $200\mu s$ on the iPSC2.

No special communication support is needed. Message tags are used to distinguish among threads in a particular processor. Interrupt driven asynchronous receives are used to execute an interrupt handler when its corresponding message arrives. This complexity is hidden using C macros for sending and receiving messages, making a trivial task to multi-thread SPMD programs with explicit send/receive calls.

This technique minimizes compiler support required to obtain multithreaded code, but at the expense of a complex, machine-dependent run-time support. The next section describes the message driven model that eliminates some of the complexity and overhead of the run-time support using compiler transformations.

3.2 Message Driven Model

This section describes the methods and run time support needed to convert SPMD message passing code into SPMD message driven code. In the message driven model, everything is done in terms of blocks of code which are actually subroutines. The compiler transforms the original SPMD code into a group of subroutines each of which end in a receive. Because everything is done in subroutines, only a pointer to the heap needs to be saved between subroutine calls. Each subroutine will have its own stack and allocate its own registers which do not need to be saved and restored between subroutines. Figure 2 shows and example running under the message driven model of threading. As presented in Figure 1, Thread 2 has already executed Code Block 1 which ended in a receive. While Thread 2 waits for its message, Thread 1 can execute Code Block 1 until the receive. After the message for Thread 2 arrives, it fires off Code Block 2 and in the process restores the heap of Thread 2. The code blocks execute without being preempted.

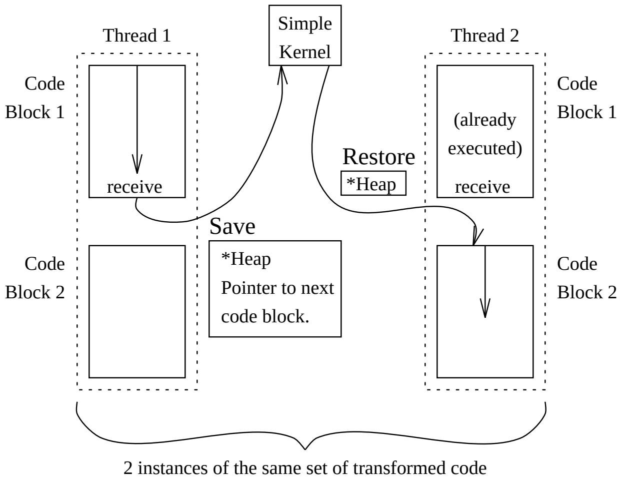

Figure 2: Message Driven Model of Threading.

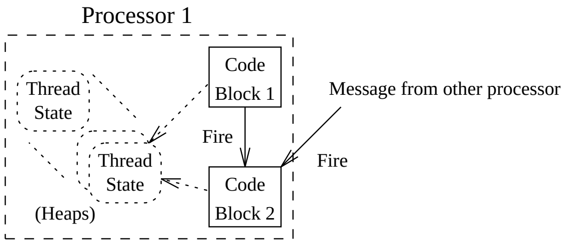

Figure 3: Message Driven Model of Computation.

The message driven model discussed in this paper requires two dependence arcs to fire before the code block is allowed to execute. One of the arcs comes from a previously executed code block within the same thread. The other arc is from a message needed to drive the next code block in the thread. Figure 3 shows the arcs that cause a code block to fire. Figure 3 also shows the heaps which comprise a thread's state. The heap is modified by the execution of code blocks. The code blocks are fired in an order to guarantee correct execution of the code, but the arcs which cause the code blocks to fire might arrive in any order.

A key component to the message driven computation is the transformation from SPMD code to the message driven model. As described earlier, transforming the message passing code to the message driven code involves putting the receives at the end of each task and returning right after the receive. This is simple for code without conditionals, but becomes more complex when conditionals and loops are

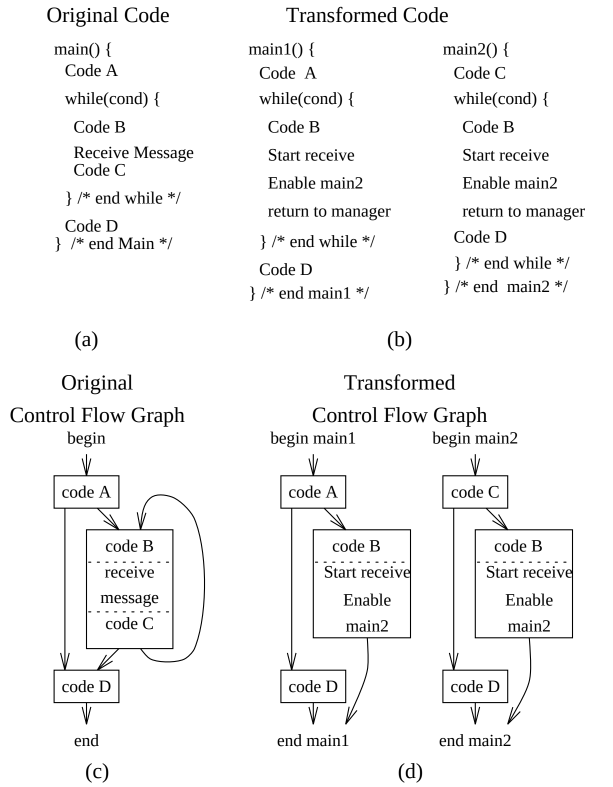

Figure 4: Transformation of the While Statement.

included. This paper only presents the transformation for converting *while* loops to message driven code. *For* loops and *repeat-until* loops can all be rewritten in terms of while loops. Figure 4(a) shows a message passing program which contains a receive in a *while* loop. This receive is intended to be executed many times in this *while* loop. Figure 4(b) shows the transformed code. In *main1*, code A is executed followed by the *while* condition check. If the *while* loop is condition is true, code B is executed, the receive is executed, the routine enables *main2*, and *main1* returns. If the *while* loop condition is false, code D is executed and the program ends. If *main2* is enabled, it will receive its message and execute code C which is the code after the receive inside the *while* loop. At this point, the original code would check for loop completion. Therefore, the transformed code must also do this check. The *while* condition is checked. If it is true, then *main2* enables another invocation of *main2* and returns. Otherwise, code D is executed and the program ends.

The CFG of the *while* transformation in Figures 4(c) and (d) show that the loop has been broken at the point of a message needing to be received. Upon the reception of the message, the rest of the loop body is executed and the check is done to execute more iterations.

	GS	QR	IMPL-2D	IMPL-1D
iPSC2	51%	22%	52%	51%
CM5	52%	25%	70%	61%

Table 1: Idle Times for the Benchmarks.

The transformation can be defined such that it handles any level of nesting of *if* statements or loops. The basic algorithm for conversion is straightforward.

4 Results

This section presents results for four scientific applications originally written in the SPMD programming model with blocking receives:

- Gauss-Seidel Iterative Partial Differential Equation Solver (GS).
- QR factorization of a dense matrix (QR).
- One-dimensional data distribution of Implicit Hydrodynamics (IMPL-1D) which is based on Livermore kernel 23 [29].
- Two-dimensional data distribution of Implicit Hydrodynamics (IMPL-2D).

All programs are run on a 16-node iPSC2 hypercube and on the CM5 on large matrices.

Using the Portable Instrumented Communication Library (PICL) from Oak Ridge National Laboratories [30, 31] the communication patterns for the programs are generated. Figures 5(a) shows the communication pattern for the QR and IMPL-1D. Processors communicate with neighbors on either side. The idle cycles occur when one processor needs a result a that its neighbor has not finished computing. As Table 1 shows, both machines running QR sit idle about 25% of the time and both machines are idle over half of the time while running IMPL-1D. Figure 5(b) shows the communication pattern for GS and IMPL-2D. Both programs are iterative method to solve partial differential equations using a five-point stencil. The points of a grid are mapped using a rectangular blockwise

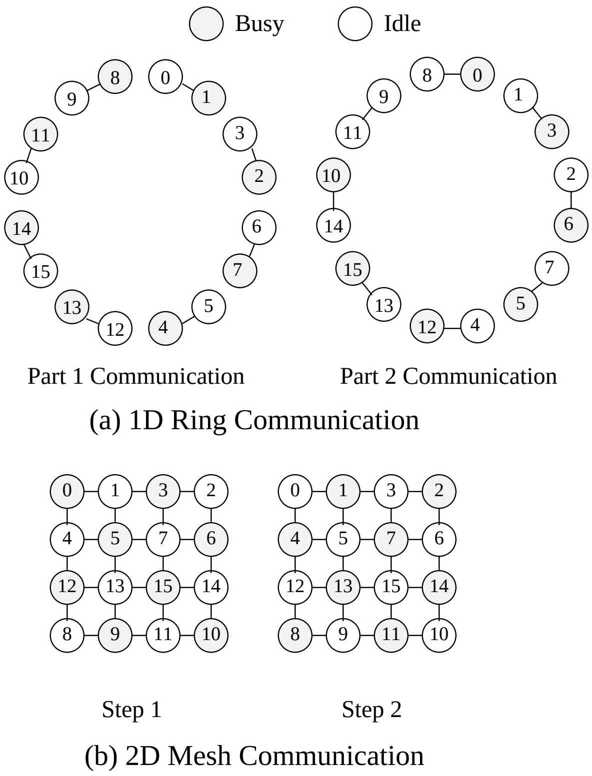

Figure 5: Communication Patterns of the Benchmarks.

data distribution to a two dimensional mesh of processors. The computation proceeds in a wavefront along the diagonal of the matrix. These codes run are very similar to those generated by a compiler without advanced message pipelining optimizations. Table 1 shows that both machines sit idle over half of the time while running these programs.

All of these programs were originally written for the iPSC2 and then ported over to the CM5. They use only basic message passing primitives. Although the programs run faster on the CM5, the CM5 is idle a higher percentage of the time than the iPSC2 as shown in Table 1. This is because the processors on the CM5 are much faster than the iPSC2, but the communication is not proportionally faster. The computations are communication bound on the CM5 more so than on the iPSC2. We do not use the CM5 vector units because they would further increase the disparity between the computation and communication times.

All four programs are run using both threading methods on a 16 node iPSC2. The improvement ratio is defined as:

$$\text{Improvement\_Ratio} = \frac{T_{\text{SPMD}}}{T_{\text{Threaded}}} \qquad (1)$$

where T_{SPMD} is the execution time of the SPMD code and T_{Threaded} is the execution time of the threaded code. The improvement ratio is shown in Figure 6 (Note that the scales among the graphs differ). Each program is run with one thread to give an idea of how much overhead the threading gives. For the grid based computations, the overhead of the Basic Threads is very small while the overhead for message driven threads ranges from 5% to 12%. QR is exceptional in that the Message Driven Threads exhibits greater improvement than the Basic Threads. Basic Threads have lower computation overhead, but higher context switching time. QR has many more context switches than the other programs. For multiple threads, both systems show improvements over the SPMD versions. In all cases, two threads are sufficient to get much of the improvements. Four threads are sometimes minimally better than two threads. For eight threads, the cost of threading overtakes the benefits and improvement tends to decrease.

Even though Basic Threads shows lower overhead, it is far less portable than the Message Driven Threads. On the CM5, the programs were run using only the the message driven threads. For QR factorization and IMPL-1D, a 64 node partition was used to keep the number of iterations greater than the number of processors. For GS and IMPL-2D, a 256 node partition was used and the processor were mapped to a 16 by 16 node grid. The improvement ratio for the programs is shown in Figure 7. (Again, the scales differ.) The results are fairly consistent with the iPSC2. The relative overhead is higher for the Message Driven Threads for most of the programs due to the higher cost of maintaining all the data on the heap instead of the stack. While it is possible to optimize some of this cost out, these results do not show it. As is true on the iPSC2, QR is exceptional on the CM5. It performs better with one thread than in the SPMD version. This improvement occurs partly because of disparity among the different communication primitives for sending messages on the CM5. It does not occur in the other programs which are more computationally bound. The multithreaded versions of the code outperform the SPMD versions in all cases except IMPL-1D. When multithreading IMPL-1D, the number of messages is doubled as well as the number of bytes sent. The overhead is too great to be overcome by multithreading. A lower overhead system would likely show improvement for this application as well. For the other cases four threads seems to work the best. In the case of IMPL-2D, eight threads does the best. Although not shown here, runs with sixteen

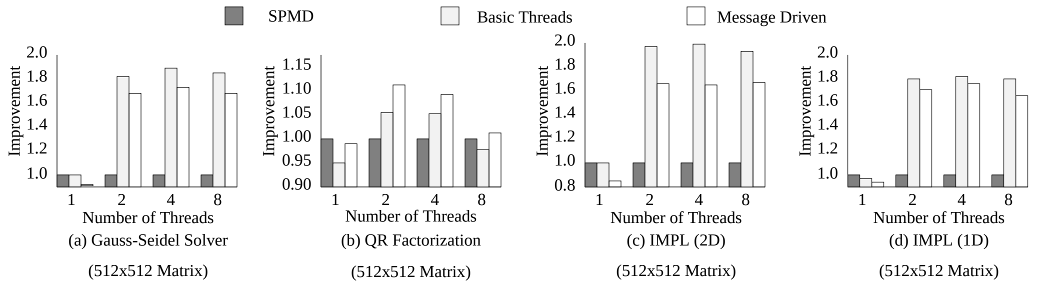

Figure 6: Improvement of Basic and Message Driven Threads on iPSC2.

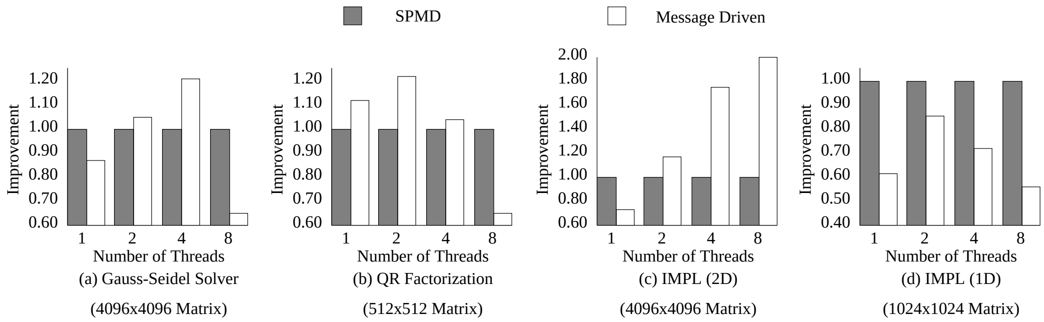

Figure 7: Improvement of Message Driven Threads on CM5.

threads show that eight is the maximum.

To measure the goodness of multithreading, T_{SPMD} and $T_{Threaded}$ from equation 1 can be broken down respectively into:

$$T_{SPMD} = T_{comp} + T_{sys} + T_{idle} \quad (2)$$

$$T_{Threaded} = T_{comp} + T_{sys} + T_{thread\_overhead} + T_{idle} - T_{idle\_reduction} \quad (3)$$

For equation 2, T_{comp} is the computation time, T_{sys} is the system overhead, and T_{idle} is the idle time. For equation 3, $T_{thread\_overhead}$ is the additional computation and system overhead introduced by multithreading and $T_{idle\_reduction}$ is the time saved by reducing in idle time.

For purposes of analysis, assume that multithreading adds no additional overhead, and all the idle time is eliminated which gives:

$$T_{thread\_overhead} = 0$$
$$T_{idle} = T_{idle\_reduction}$$

With these assumptions and equations 2 and 3 substituted into equation 1, the maximum improvement ratio obtainable through multithreading is:

$$\text{max\_improvement} = \frac{T_{comp} + T_{sys} + T_{idle}}{T_{comp} + T_{sys}} \quad (4)$$

	GS	QR	IMPL-2D	IMPL-1D
iPSC2	0.84	0.39	0.92	0.79
CM5	0.19	0.67	0.44	-0.10

Table 2: Fraction of Maximum Improvement.

Normalizing the numerator of equation 4 to one, the maximum improvement possible over the SPMD code can be calculated from the fraction of time idle using the following formula:

$$\text{max\_improvement} = \frac{1}{1 - \text{fraction\_idle}}$$

This formula assumes that the system overhead in the SPMD version remains constant in the threaded version. Even though the maximum improvement is unattainable since threading does add overhead, the fraction of the maximum improvement obtained can be calculated by using:

$$\text{fraction\_max\_improvement} = 1 - \frac{\text{max\_improvement} - \text{actual\_improvement}}{\text{max\_improvement} - 1}$$

The results using the best actual improvement for each program are shown in Table 2. In some cases, threading almost eliminates non-useful processor time. In cases where there is no improvement due to threading overhead, such as the CM5 running IMPL-1D, the fraction is negative. Even though lower overhead implementations could alleviate idle time even further, the simple methods do reasonably well at decreasing non-useful processor cycles.

5 Conclusion

This paper compares two compiler transformations and supporting run-time systems that produce multithreaded code for distributed memory multicomputers. The first approach minimizes compiler transformations at a cost of an increased complexity of the run-time system. The second approach simplifies the design of the run-time system, moving the complexity to the compiler, and also increases portability.

Both techniques can efficiently reduce idle time. The first one is best suited for applications in which idle times are large and uniform. The second one is preferred when context switching time becomes critical.

Improvements can still be made to further reduce idle times in DMM's. Hardware and operating system support for multiple threads within processors of a DMM might be useful in bringing down threading overhead more.

6 Acknowledgments

The authors would like to thank the National Center for Supercomputing Applications (NCSA) for the use of the Thinking Machines CM5 supercomputer.

References

[1] E. Su, D. J. Palermo, and P. Banerjee. Automating Parallelization of Regular Computations for Distributed Memory Multicomputers in the PARADIGM Compiler. *International Conference on Parallel Programming*, 1993.

[2] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report COMP TR90-141, Rice University, Houston, Texas, December 1990.

[3] High Performance Fortran Forum. *High Performance Fortran Language Specification*, CITI/CRPC, Rice University, Houston, TX, November 1992. version 0.4, draft.

[4] J. M. Hsu. *Performance Measurement and Hardware Support for Message Passing in Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[5] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, Department of Computer Science, University of Illinois, September 1992.

[6] M. Gupta and P. Banerjee. Compile-Time Estimation of Communication Costs on Multicomputers. In *Proceedings of 6th International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[7] M. Gupta and P. Banerjee. A Methodology for High-Level Synthesis of Communication on Multicomputers. In *Proceedings of 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[8] K. P. Belkhale and P. Banerjee. Approximate Algorithms for the Partitionable Independent Task Scheduling Problem. In *International Conference on Parallel Programming*, pages 72–75, 1990.

[9] K. P. Belkhale and P. Banerjee. A Scheduling Algorithm for Parallelizable Dependent Tasks. In *International Parallel Processing Symposium*, pages 500–506, 1991.

[10] Shankar Ramaswamy and Prithviraj Banerjee. Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler. In *Proceedings of the International Conference on Parallel Processing*, 1993.

[11] Antonio Lain and Prithviraj Banerjee. Techniques to Overlap Computation and Communication in Irregular Iterative Applications . In *Proceedings of the 8th ACM International Conference on Supercomputing*, 1994.

[12] C.Whitby-Strevens. The Transputer. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 292–300, 1985.

[13] Edward Felten and Dylan McNamee. Improving the performance of message-passing aplications by multithreading. Technical Report 92-09-07, University of Washington, Seattle, 1992.

[14] T. von Eicken, D. E. Culler, S. C. Goldstein. Active messages: a mechanism for integrated

communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[15] G. M. Papadopulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–90, 1990.

[16] W. J. Dally, L. Chao, A. Chein, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196, 1987.

[17] Micheal J. Beckerle. Overview of the START(*T) Multithreaded Computer. In *Proceedings of the IEEE COMPCON*, February 1993.

[18] L. V. Kale'. The Chare-Kernel Parallel Programming Language and System. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 17–25, 1990.

[19] S. Hiranaandani, K. Kennedy, C. W. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *The International Conference on Supercomputing*, pages 1–14, 1992.

[20] S. Hiranaandani, K. Kennedy, C. W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[21] L.A. Belady and C.J. Kuehner. Dynamic space-sharing in computer systems. *Communications of the ACM*, 5:282–288, May 1969.

[22] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, 1989.

[23] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Computers*, 3:525–539, September 1992.

[24] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, 1988.

[25] Herb. Schwetman. CSIM Reference Manual (Revision 14). Technical Report ACA-ST-252-87, MCC, 1990.

[26] D. Stein and D. Shah. Implementing Lightweight Threads. In *Summer-92 USENIX*, 1992.

[27] T.E. Anderson and B.N. Bershad and E.D. Lazowska and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 1:53–79, February 1992.

[28] D. L. Eager and J. Zahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Transactions on Computer Systems*, 11:1–32, February 1993.

[29] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[30] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[31] G. A. Geist, M. T. Heath, and P. H. Worley. PICL: a portable instrumented communication library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratories, 1990.