COMMUNICATION OPTIMIZATIONS USED IN THE PARADIGM COMPILER FOR DISTRIBUTED-MEMORY MULTICOMPUTERS*

Daniel J. Palermo, Ernesto Su, John A. Chandy, and Prithviraj Banerjee
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.
{palermo, ernesto, jchandy, banerjee}@crhc.uiuc.edu

Abstract — The PARADIGM (PARA*llelizing compiler for* DI*stributed-memory* G*eneral-purpose* M*ulticomputers) project at the University of Illinois provides a fully automated means to parallelize programs, written in a serial programming model, for execution on distributed-memory multicomputers. To provide efficient execution,* PARADIGM *automatically performs various optimizations to reduce the overhead and idle time caused by interprocessor communication. Optimizations studied in this paper include message coalescing, message vectorization, message aggregation, and coarse grain pipelining. To separate the optimization algorithms from machine-specific details,* parameterized *models are used to estimate communication and computation costs for a given machine. The models are also used in coarse grain pipelining to automatically select a task granularity that balances the available parallelism with the costs of communication. To determine the applicability of the optimizations on different machines, we analyzed their performance on an Intel iPSC/860, an Intel iPSC/2, and a Thinking Machines CM-5.*

1. INTRODUCTION

Distributed-memory multicomputers such as the Intel iPSC/860, the Intel Paragon, the IBM SP-1, the NCUBE/2, and the Thinking Machines CM-5 offer significant advantages over shared-memory multiprocessors in terms of cost and scalability. However, lacking a global address space, they present a very difficult programming model in which the user must specify how data and computation are to be partitioned across processors and determine which sections of data need to be communicated among which processors.

To overcome this difficulty, significant research effort has been aimed at source-to-source parallelizing compilers for multicomputers that relieve the programmer from the task of program partitioning and communication generation, while the specification of data distributions remains a responsibility of the programmer. These compilers take a program written in a sequential or shared-memory parallel language, and based on user-specified partitioning of the data, generate code for a given multicomputer. Examples include Fortran D [1], Fortran 90D [2], the SUIF compiler [3], and the Superb compiler [4]. However, many of these research efforts are now looking into automated data partitioning. Many researchers in this area are also currently involved in defining High Performance Fortran (HPF) [5] to standardize parallel programming with data distribution directives.

Some related work on the evaluation of compiler optimizations performed by the Fortran D compiler has previously appeared in [6]. The compiler optimizations that they described were selected manually, applied to one-dimensional partitionings, and only evaluated on an iPSC/860. The novel aspects we present in this paper are the *automatic* selection of multidimensional data partitions, the development of an estimation framework for coarse grain pipelining, and the comparison of existing optimizations on different architectures. The remainder of this paper is organized as follows. Section 2 provides an overview of the PARADIGM compiler. The various communication optimizations used in the compiler, as well as the techniques used to select granularity for the pipelining transformation, are described in Section 3. An analysis of the results using the presented optimizations is performed in Section 4, and conclusions are presented in Section 5.

2. OVERVIEW OF PARADIGM

Figure 1 presents a functional view of the major components in the PARADIGM compiler. The compiler accepts a sequential program (currently FORTRAN 77) and produces a SPMD (Single Program Multiple Data) parallel program with message passing. Following are brief descriptions of some of the major areas in the compilation strategy:

Program Analysis *Parafrase-2* [7] is used as a preprocessing platform to parse the sequential program into an intermediate representation, to perform useful analysis (such as generating flow, dependence, and call graphs), as well as to facilitate various code transformations (such as constant propagation, induction variable substitution, loop distribution, loop interchange, and scalar expansion).

To describe partitioned sets of iterations and regions of data, *Processor Tagged Descriptors* (PTDs) [8] are used to provide a uniform representation of the partitioning for every processor. Operations on PTDs are extremely efficient, capturing the effect on all processors in a given dimension simultaneously. PTDs are easily extended to an arbitrary number of dimensions and are independent of

*This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, and in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

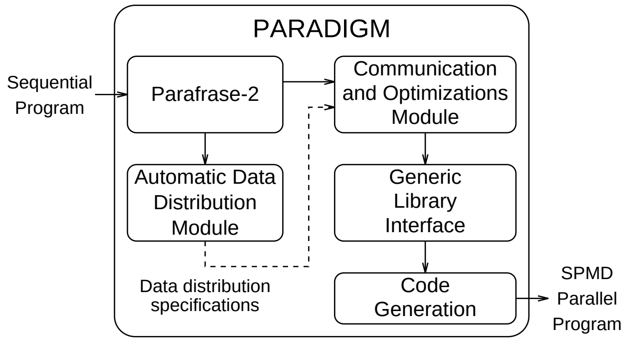


Figure 1: PARADIGM Compiler Overview

the total number of processors.

Data Partitioning Distribution of data is determined automatically by the compiler using a constraint-based algorithm [9, 10], which selects an abstract multidimensional mesh topology along with how program data is to be distributed on the mesh. To minimize the execution time for a particular machine, parameterized models are used to estimate computation and communication costs. For each target machine, there is a set of parameters which interface with the cost models effectively isolating the partitioning algorithm from a specific architecture.

Computation Partitioning Computation is divided among processors using the *owner computes* rule. A direct application of this rule without further optimizations leads to *run-time resolution*, which results in code that computes the ownership and communication for each reference at run time. An efficient implementation of the owner computes rule, however, can avoid the overhead of computing ownership at run time. For computations enclosed in a loop nest, the loops can be partitioned (known as *loop bounds reduction* [1]) allowing processors to execute only those iterations which have assignments that write to local memory.

Communication Analysis The references in assignment statements are also analyzed to detect the need for communication. PTDs are constructed to describe the iterations requiring communication of non-local data, the processors involved, and the exact regions of the arrays to be sent or received. Once the communication descriptors have been computed for individual references, various communication optimizations can be performed (see Section 3). Data dependence and flow information is also used to determine whether a given optimization is applicable, and if so, to what extent.

Processor Mapping The compiler generates code that views the target machine as a multidimensional mesh of processors. The exact configuration of this mesh is chosen during the automatic data-partitioning phase. Since a mesh topology can be easily mapped onto other topologies, machine-dependent processor mapping is achieved through library support to efficiently map the mesh to a given architecture [11].

Generic Library Interface Support for specific communication libraries is provided through a generic library interface. For each supported library, abstract functions are mapped to corresponding library-specific code generators at compile-time. Library interfaces have been implemented for the Intel iPSC communication library, Thinking Machines CMMD, Parasoft Express [12], PVM [13], and the Portable Instrumented Communication Library (PICL) [14, 15]. Express, PVM, and PICL also provide execution tracing and support for many different machines. The portability of this interface allows the compiler to generate code for a wide variety of machines.

Summary When fully implemented, PARADIGM will be capable of performing all of the following tasks automatically: generation of data partitioning directives [9, 10], partitioning of computation and generation of communication [11], synthesis of high-level communication [16], exploitation of functional parallelism [17], support of a multithreaded execution model [18], and support of irregular computations [19].

3. COMMUNICATION OPTIMIZATIONS

The first three communication optimizations examined in this paper, *message coalescing*, *message vectorization*, and *message aggregation*, are targeted at reducing the overhead associated with communication [21, 22]. These optimizations rely on the fact that the start-up cost of communication for distributed-memory multicomputers is much greater than the per-byte transmission cost (by a factor of over 400 times for the iPSC/860 and 700 times for the CM-5). In general, given the start-up latency and transmission rate for a specific architecture (see Table 1), the compiler uses a communication model in which the transfer cost (in μsec) of a message of m bytes is defined as:

$$transfer(m) \quad = \quad ovhd + rate \times m \tag{1}$$

For a machine such as the iPSC/860, the parameters depend on the length of the message, so the model becomes:

$$transfer(m) \quad = \quad \begin{cases} 60 + 0.50m & (\text{if } m \leq 100) \\ 160 + 0.36m & (\text{if } m > 100) \end{cases}$$

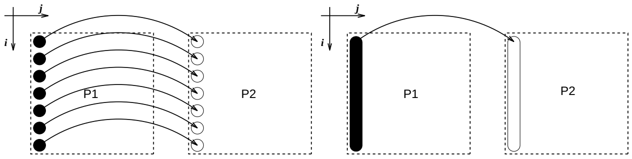
Several optimizations can be employed to increase the performance of the parallel program by combining messages in various ways to reduce the total amount of communication overhead.

3.1 Message Coalescing

Separate communication for different references to the same data is unnecessary if the data has not been modified between uses. When statically analyzing the access

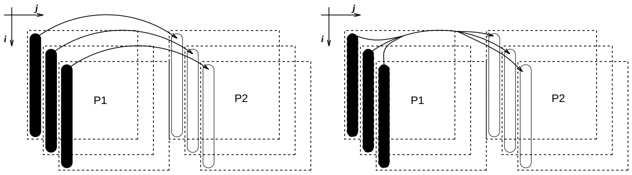
Table 1: Communication Model Parameters [20]

Machine	Size	*ovhd* (μs)	*rate* (μs)
CM-5	$m \leq 16$	23	0.12
	$m > 16$	86	0.12
iPSC/860	$m \leq 100$	60	0.50
	$m > 100$	160	0.36
iPSC/2	$m \leq 100$	390	0.20
	$m > 100$	700	0.36



(a) Before (b) After

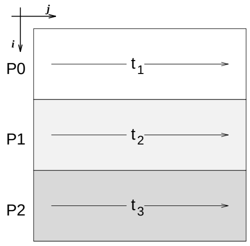
Figure 2: Message Vectorization



(a) Before (b) After

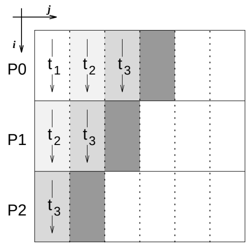
Figure 3: Message Aggregation

```
do i = 1, Y/p
   do j = 1, X
      a(i, j) = a(i-1, j) + a(i, j-1)
   end do
end do
```



(a) Before Transformation

```
do j = 1, X
   do i = 1, Y/p
      a(i, j) = a(i-1, j) + a(i, j-1)
   end do
end do
```



(b) After Loop Interchange

Figure 4: Code Example for Loop Pipelining

patterns, these redundant communications are detected and coalesced into a single message, allowing the data to be reused rather than communicated for every reference. For sections of arrays, unions of overlapping PTD index sets ensure that each unmodified data element is communicated only once. Coalescing is always beneficial since entire communication operations can be eliminated.

3.2 Message Vectorization

Non-local elements of an array that are indexed within a loop nest can also be *vectorized* into a single larger message instead of being communicated individually (see Figure 2). Dependence analysis is used to determine the outermost loop at which the combining can be applied. The "item-wise" messages are combined, or *vectorized*, as they are lifted out of the enclosing loop nests to the selected vectorization level.[a] Vectorization reduces the total number of communication operations, but also increases the message length.

3.3 Message Aggregation

Messages (corresponding to several array sections) to be communicated between the same source and destination can also be *aggregated* into a single larger message. Multiple communication operations (to be performed at the same point) are sorted by their destinations during the communication analysis. Messages with identical destinations can then be collected into a single communication operation (see Figure 3). The gain from aggregation is similar to vectorization in that multiple communication operations can be eliminated at the cost of increasing the message length. Aggregation can be performed on communication operations of individual data references as well as vectorized communication operations. Both of these applications of message aggregation will be examined in Section 4.

3.4 Coarse Grain Pipelining

In loops where there are no cross-iteration dependencies, parallelism is extracted by independently executing groups of iterations on separate processors. However, in cases where there are cross-iteration dependencies due to recurrences, it is not possible to immediately execute every iteration. Often, there is the opportunity to overlap parts of the loop execution, using some form of synchronization to ensure that the data dependencies are enforced.

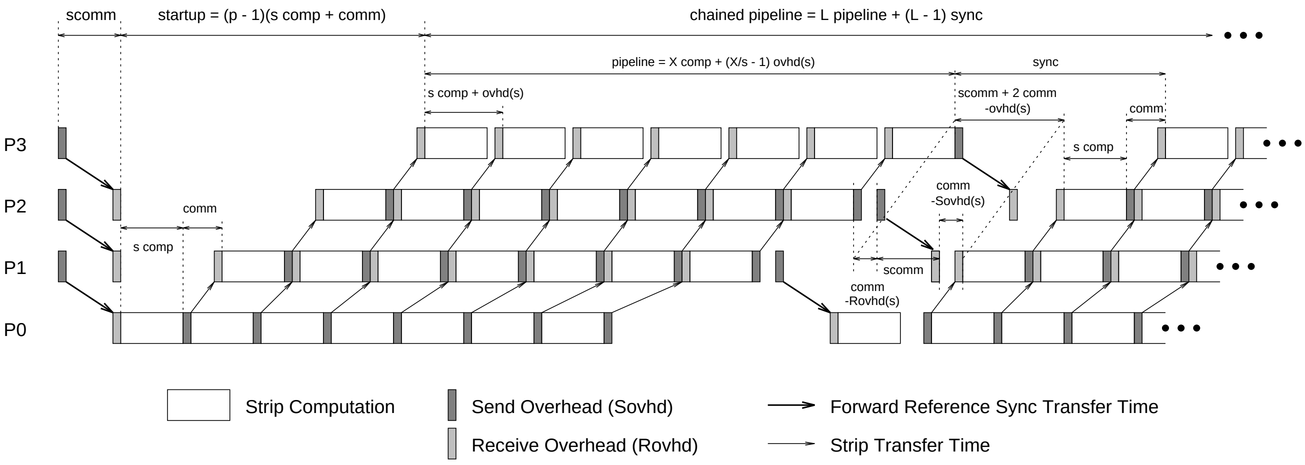
In Figure 4a, the first processor is performing an operation on every element of the rows it owns before sending the border row to the waiting processor, thereby serializing execution of the entire computation. In the example in Figure 4b, a loop interchange has been applied such that the first processor now can compute one partitioned column of elements. It can then send the border element of that column to the next processor so that processor can begin computation immediately. Such techniques have been used in the design of systolic arrays [23, 24] as well as in *software pipelining* [6]. Ideally, if communication has zero overhead, this is the most efficient form of computation, since no processor will wait unnecessarily. Unfortunately, this assumption is not valid for distributed-memory systems. By considering overhead, the cost of performing numerous single element communications can be quite expensive. To address this problem, the total communication overhead can again be reduced by increasing the granularity of the communication. This procedure has become known as *coarse grain pipelining* [6].

3.4.1 Execution Time Analysis

In Figure 5, an execution framework of a generalized two-level pipelined loop nest (similar to that in Figure 6) is presented. In this model, coarse grain pipelining is performed by strip-mining the outer dimension of the two-dimensional loop nest while chaining is applied between consecutive pipelines at an outer iteration level. Definitions of all variables which will be used in the analysis are also listed along with the figure.

Since the amount of available parallelism is reduced as the granularity or strip size, s, is increased, this value must

[a] Management of available memory may require that large regions of data be only partially vectorized. [6]



L = outer loop iterations
X, Y = number of columns and rows
c = cost of inner loop instructions
p = number of processors
s = strip size ($s \cdot b$ = strip size in bytes)

$transfer$(m) = communication time of m elements
$overhead$(m) = communication overhead for m elements
comp = computation time of one column = $\lceil Y/p \rceil \; c$
comm = time for $transfer(s)$ (strip of a row)
scomm = time for $transfer(X)$ (entire row)

Figure 5: Estimation Framework for Coarse Grain Pipelining

```
do l = 1, L
   comm(size = X)
   do j = 1, X
      if (my$p > 0) recv(size = 1)
      do i = 1, ⌈Y/p⌉
         computation
      end do
      if (my$p < p − 1) send(size = 1)
   end do
end do
```

(a) Fine grain pipelining

```
do l = 1, L
   comm(size = X)
   do j = 1, X, s
      bb = min(s,X-j+1)
      if (my$p > 0) recv(size = bb)
      do jj = j, j + bb - 1
         do i = 1, ⌈Y/p⌉
            computation
         end do
      end do
      if (my$p < p − 1) send(size = bb)
   end do
end do
```

(b) Coarse grain pipelining

Figure 6: Cross Processor Loop Pipelining

be carefully selected. If s is one, we have fine grain pipelining, and if s is equal to the bounds of the serial loop, X, the execution is serialized in the inner dimensions (no pipelining). Somewhere in between lies an optimal s that maximizes the overlap of communication and computation. In order to investigate this tradeoff, an execution time estimate is developed from the framework to allow automatic selection of a strip size that yields the highest performance.

The first major phase of execution is the time required to fill the pipeline. This is related to the number of processors as well as the strip size. From the diagram, it can be seen that:

$$\text{startup} = (p - 1)(s \cdot \text{comp} + \text{comm})$$

The next portion of execution is the time spent in the pipeline. Ideally, with no communication overhead, this time should be equal to the amount of computation ($X \cdot comp$). However, because of the presence of communication, the time for each message communicated in the pipeline must also be taken into account. The number of communication operations is $\lceil \frac{X}{s} \rceil$. Therefore,

$$\text{pipeline} = X \cdot \text{comp} + \left(\left\lceil \frac{X}{s} \right\rceil - 1 \right) \mathit{overhead}(s)$$

For loops with flow-dependencies caused by *forward* references in the computation, the execution of a sequence of pipelined loop nests will also generate communication incurring some extra synchronization costs to communicate the required data. Since most parallel machines only support a single channel for memory transfer operations through the communication network, this causes some extra delay which can be seen during multiple sends in Figure 5.[b] The "scomm" term is used to represent the

[b] In some applications there are no forward references, and therefore, no need to synchronize between outer loop iterations. In these cases each processor can proceed without any further synchronization.

amount of *synchronizing* communication (If an entire row is communicated, then this cost is $transfer(X)$). Note that this will also be present in the start-up synchronization for the loop nest (see Figure 5).

$$\text{sync} = s \cdot \text{comp} + 3 \cdot \text{comm} - overhead(s) + \text{scomm}$$

Since the pipeline is entered L times, the total execution time is then:

$$\begin{aligned}\text{total cost} &= \text{scomm} + \text{startup} + L \cdot \text{pipeline} + (L-1)\,\text{sync} \\ &= [LX + s(p+L-2)] \left\lceil \frac{Y}{p} \right\rceil c + \\ &\quad (p+3L-4)\ transfer(s) + \\ &\quad \left[L \left(\left\lceil \frac{X}{s} \right\rceil - 2 \right) + 1 \right] overhead(s) + \\ &\quad L \cdot transfer(X) \qquad (2)\end{aligned}$$

Using the communication cost model previously presented (see Equation 1) the total cost becomes:

$$\begin{aligned}\text{total cost} &= [LX + s(p+L-2)] \left\lceil \frac{Y}{p} \right\rceil c + \\ &\quad (p+3L-4)(ovhd + s \cdot b \cdot rate) + \\ &\quad \left[L \left(\left\lceil \frac{X}{s} \right\rceil - 2 \right) + 1 \right] ovhd + \\ &\quad L(ovhd + X \cdot b \cdot rate)\end{aligned}$$

The total cost can then be minimized with respect to the strip size to select the optimal granularity. Verification of the second derivative will show that this value of s is indeed a minimum. Since *ovhd* and *rate* are actually functions of s, Equation 3 is evaluated iteratively if necessary.

$$\begin{aligned}\frac{\partial}{\partial s}\text{total cost} &= (p+L-2) \left\lceil \frac{Y}{p} \right\rceil c + \\ &\quad (p+3L-4) b \cdot rate - L \cdot ovhd\ \frac{X}{s^2} = 0\end{aligned}$$

$$s = \sqrt{\frac{X \cdot L \cdot ovhd}{(p+L-2)(b \cdot rate + \lceil \frac{Y}{p} \rceil c) + 2b(L-1)rate}} \qquad (3)$$

3.4.2 Serial Computation Cost Estimation

Since the cost, c, of the inner loop computations appears in this expression, it is necessary to estimate the computation costs. Two techniques can be used: source-level [9] and assembly-level cost estimation. For each method of estimation, a given machine's operation costs are expressed in terms of clock cycles. In the case of source-level estimation, these costs must take into account any support instructions (address computation, register loads, stores, etc.) performed for the actual computation. Assembly-level estimation requires the timing costs of individual machine instructions as well as being able to perform pre-compilation of the source under examination. In both cases, estimation of the cost of a fixed block of code (which may contain loops and other control flow structures) requires computing a dynamic cycle count.

Table 2: Data Partitioning for Initial Tests

Test	iPSC/860	CM-5	Distribution
ADI	$4 \times 1000 \times 2$	$4 \times 16000 \times 2$	`*, block, *`
EXPL	1000×7	16000×7	`block, *`
Jacobi	500×500	1000×1000	`block, block`

4. EVALUATION OF OPTIMIZATIONS

A group of small scientific program kernels are used to examine the performance of the presented communication optimizations. The selected program fragments include

- ADI Integration (ADI, kernel 8) [25]
- 2-D Explicit Hydrodynamics (EXPL, kernel 18) [25]
- Jacobi's Iterative Method

Three other programs which exhibit cross-iteration dependencies are selected to examine the pipelining optimization:

- Implicit Hydrodynamics (IMPL, kernel 23) [25]
- Successive Over-Relaxation Iterative Method (SOR)
- Block Lower Triangular Solver (BLTS) [26]

These programs will be examined separately in Section 4.6.

Array dimensions are statically specified and loop bounds are determined at compile time when possible. Both the virtual mesh configuration as well as the data partitioning of the arrays are automatically selected by the compiler. The sizes of the major arrays and the chosen distributions are shown in Table 2 for each of the first three programs. The evaluation of the overhead optimizations is performed with 8 processors of an Intel iPSC/860 as well as 64 processors of a Thinking Machines CM-5. Larger arrays were used on the CM-5 to reliably obtain measurable times.

Traces taken using PICL [15] are analyzed using the ParaGraph [14] visualization tool to examine the effect of each optimization. The ParaGraph "Spacetime diagram" is used to further examine the execution profile viewing the frequency and amount of communication that takes place. Continuous lines indicate uninterrupted execution while a break in a line indicates that a processor has blocked awaiting communication. Communication operations are indicated as vertical lines between the cooperating processors' execution profiles (see Figures 7 to 10). Snapshot views of the execution of EXPL compiled with the selected optimizations are shown in each figure while performance data is presented for all three test programs.

For comparison purposes, the reported execution times have been normalized to the serial execution of the corresponding program and are further separated into two quantities:

- Busy – amount of time spent on *useful* computation (where *useful* refers to only the code which carries out the actual computation)
- Overhead – time spent executing code related to computation partitioning and communication

The relative effectiveness of each optimization is determined by examining the amount of overhead eliminated as the optimizations are incrementally applied.

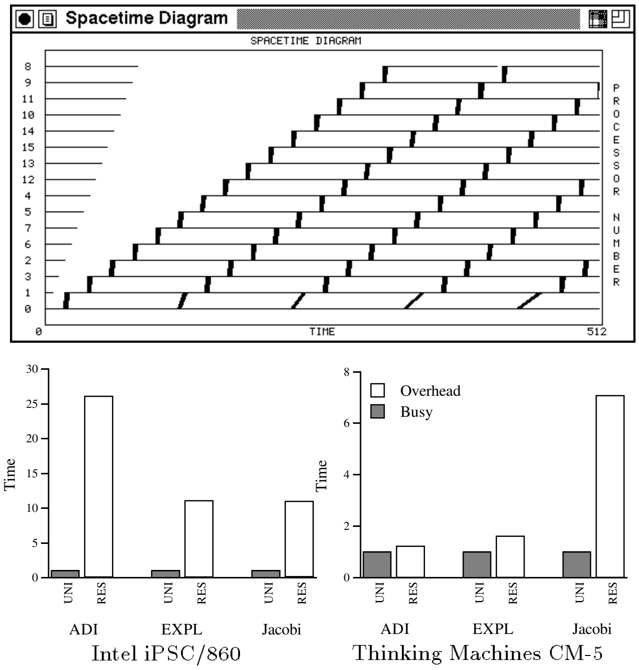


Figure 7: Evaluation of Run-time Resolution

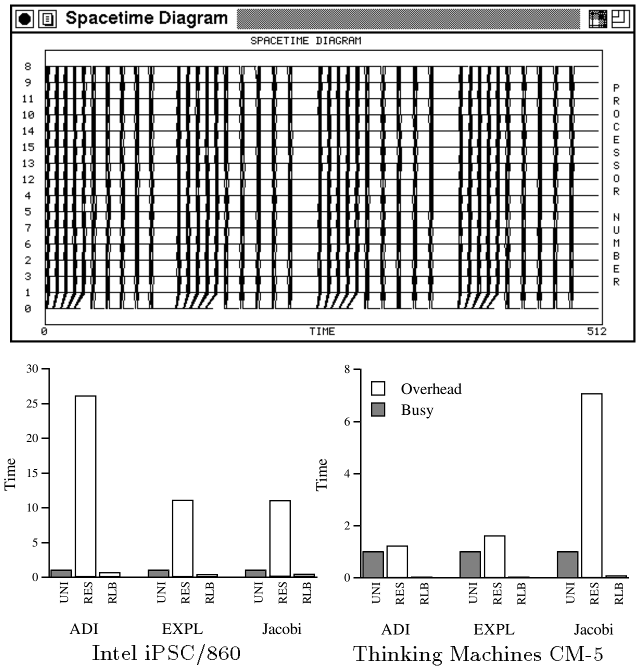


Figure 8: Reduced Loop Bounds & Coalescing

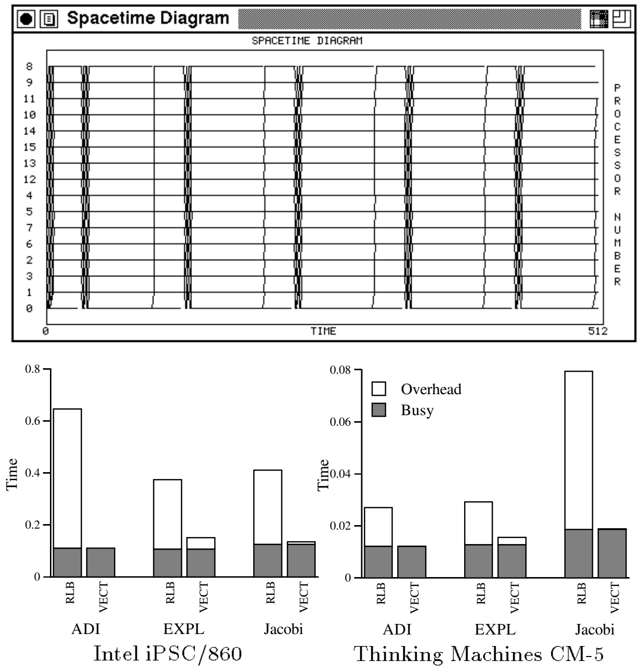


Figure 9: Evaluation of Message Vectorization

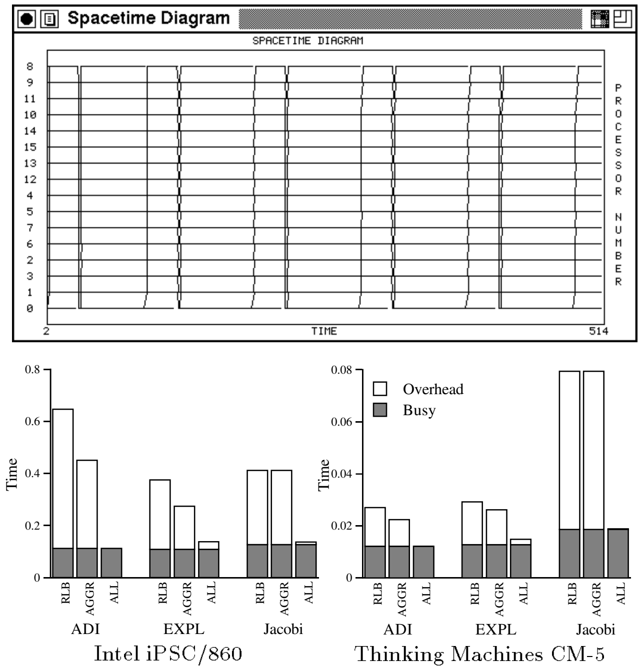


Figure 10: Message Vectorization & Aggregation

UNI	Uniprocessor Execution	VECT	Communication Vectorization
RES	Run-time Resolution	AGGR	Communication Aggregation
RLB	Reduced Loop Bounds	ALL	All Optimizations applied

Traces taken from Explicit Hydrodynamics (Livermore Kernel 18)
using a 16 processor iPSC/2 (time in μs)

4.1 Results of Run-time Resolution

A direct application of the owner computes rule without any further optimization leads to *run-time resolution.* Since each processor must execute the entire iteration space (to determine if any other processor needs locally owned data), different instances of a communication operation are effectively serialized across the processors in a given mesh dimension (see Figure 7). Multiple communication operations appear to be pipelined, but the time spent between successive messages can be quite large.

Examining the execution time for each program, it can be seen that traversing the entire iteration space to compute ownership results in large amounts of overhead. Communication for resolution programs is also very inefficient since it is comprised of a large number of small (sometimes even redundant) messages resulting in high communication overhead. The net result is a reduction in performance when compared to the serial case.

4.2 Results of Loop Bounds Reduction and Message Coalescing

By applying loop bounds reduction and statically generating communication operations, the serialization present in the baseline resolution cases can be eliminated. The statically generated communication operations are effectively collapsed in time as compared to the serialized communication present in run-time resolution (see the spacetime diagram in Figure 8). Static analysis of communication allows the loop bounds to be partitioned instead of requiring every processor to check the ownership of each reference for the entire iteration space.

The overhead is dramatically reduced as all ownership and communication is now statically determined. Note, however, that the size of the messages is still identical to that in run-time resolution (single elements), but redundant messages have been eliminated by application of the coalescing optimization. It is still apparent that there is an excessive amount of small messages being communicated (as seen in Figure 8) since communication overhead is now the dominant factor in the execution overhead.

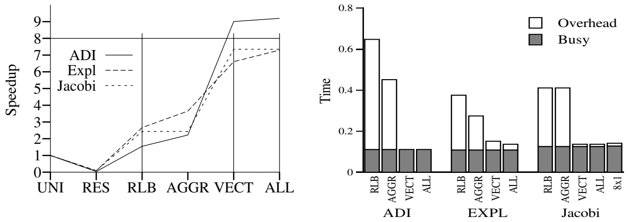
4.3 Results of Message Vectorization

It is also possible to vectorize the communication operations after loop bounds reduction and message coalescing have been applied. Using dependence information to determine when vectorization is applicable, communication operations can be lifted out of the inner-most loops thereby reducing the communication frequency (this can be seen in Figure 9). Recall that this also increases the size of the messages, but since the start-up cost is much greater than the per-byte cost, there is a large gain in performance.

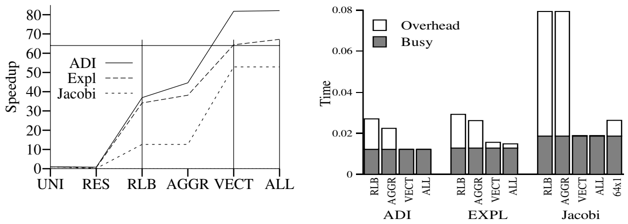
For ADI, it is possible to vectorize all communication completely out of the loop nest. Since there is no longer any synchronization due to communication within the loops, the reduction of the array bounds resulted in super-linear speedup, which can be attributed to a gain in cache performance (also see Figure 11).

4.4 Results of Message Aggregation

Aggregation also reduces communication frequency by grouping multiple communication operations, increasing the size of the resulting message. (see Figure 10). Applying aggregation after loop bounds reduction and message coalescing, a performance gain is seen for both ADI and EXPL. There was no improvement using aggregation on Jacobi's method since there was only one message communicated for each source/destination pair.



(a) 8 processor Intel iPSC/860



(b) 64 processor Thinking Machines CM5

Figure 11: Comparison of Combined Optimizations

By applying aggregation after vectorization reduces the communication overhead as can be seen in the "thinning" of the communication in the execution profiles when comparing Figures 9 and 10. Since groups of messages are combined, the performance improvement is related to the number of different array sections that need to be communicated at the same point in the program. For the small test programs examined, only a few arrays were involved in the communication, and hence the overall performance improved only slightly.

4.5 Results of Combined Optimizations

Figure 11 shows the relative speedup and the amount of overhead measured with each optimization. Eliminating the overhead of traversing the entire iteration space to compute ownership, the combination of static communication generation and loop bounds reduction attained roughly half of the total available performance for most programs. By applying message vectorization it was possible to reduce the majority of the remaining communication overhead resulting in near ideal performance. Message aggregation was beneficial for the two programs (ADI, EXPL) which contained references to a number of different arrays. Due to the more limited scope of application, its effect tended to not be as dramatic as the other optimizations.

Since the compiler selected a two-dimensional partitioning for Jacobi's method, an extra run is shown in Figure 11 where the compiler was forced to generate a one-dimensional partitioning. For the eight processor iPSC/860, the execution time increased by about 5%. For the larger, 64 processor CM-5, the execution time has increased by over 30% and will become more apparent with larger numbers of processors. The highest performance was achieved by allowing the compiler to select the best distribution based on cost models to estimate communica-

Table 3: Mesh Configurations

Test	N	Mesh	Test	N	Mesh
ADI/ EXPL	1	1×1	Jacobi	1	1×1
	2	2×1		2	2×1
	4	4×1		4	4×1
	8	8×1		8	4×2
	16	16×1		16	4×4
	32	32×1		64	8×8
	64	64×1		32	8×4
	128	128×1		128	16×8
	256	256×1		256	16×16

tion and computation.

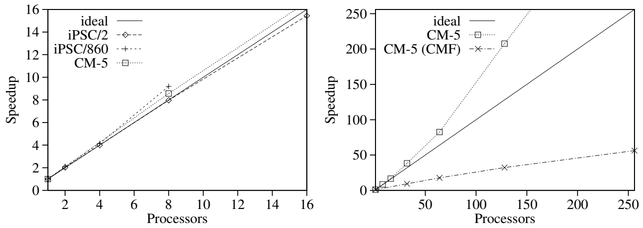
Each of the fully optimized test programs were also executed with larger numbers of processors to examine the scalability. In Table 3, each program's mesh configuration is shown as selected by the automatic partitioning pass. The speedup curves for each of the test programs run on an iPSC/2, an iPSC/860, and a CM-5, can be seen in Figure 12. The super-linear speedup is again observed for ADI and can be attributed to the cache effect previously described. The execution of these three programs was also compared to an existing data-parallel compiler available on the CM-5 using a language known as Connection Machine Fortran (CMF 2.1 Final, CMOST 7.3).[(c)] The reduction in performance of the programs compiled with CMF can be attributed to the fact that it uses a SIMD (Single Instruction Multiple Data) model of execution for compilation (carried over from the CM-2). Simulating a SIMD architecture on a MIMD multicomputer (such as the CM-5) incurs fairly high synchronization costs between blocks of computation. Future versions of the CMF compiler will most likely use a more asynchronous model of computation (i.e. SPMD) which is better suited to the CM-5.

4.6 Results of Coarse Grain Pipelining

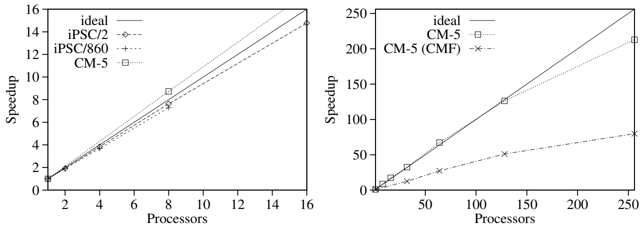
To evaluate the quality of the strip size estimate developed in Section 3.4 (see Equation 3), each of the test programs (IMPL, SOR, BLTS) was executed with varying strip sizes to compare the estimate with the actual minimum. We also examined a simpler one-level strip size estimate developed in the Fortran D project [6]. It is interesting to note that our Two-Level estimate reduces to their One-Level estimate when several assumptions are applied. In its most general form, the simpler estimate assumes that chaining does not occur between consecutive pipelines, communication is modeled as constant overhead, and the array dimensions are square.

$$\text{One-Level} = \sqrt{\frac{p}{p-1}\frac{ovhd}{c}}$$

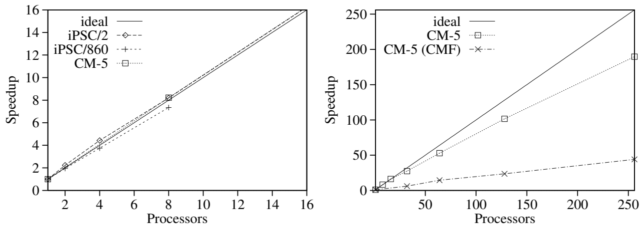
Given the two forms of strip size estimates, the optimal strip size is compared with the estimated minimums in Table 5. The execution time using the Two-Level estimate can be seen to be no more than 8% away from the optimal time while the most general form of the One-Level estimate was, at times, more than 30% worse than the actual



(a) ADI Integration



(b) Explicit Hydrodynamics



(c) Jacobi's Iterative Method

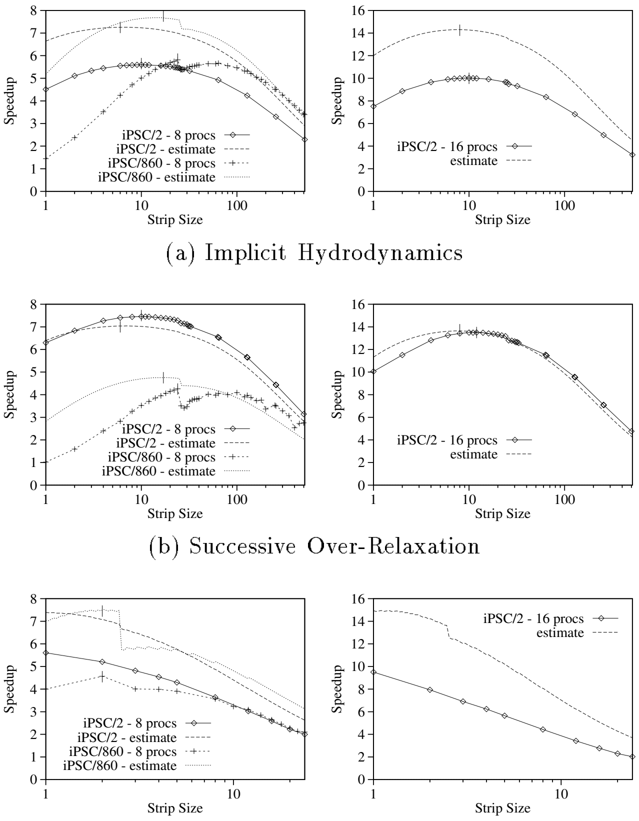
Figure 12: Performance Comparison

minimum. The One-Level estimate predicted strip sizes of roughly half the size of the Two-Level estimate. In fact, a further approximation is made in Fortran D, resulting in an estimate of $\sqrt{\frac{ovhd}{c}}$ which proves to be even farther from the minimum.

The major gain seen with the Two-Level estimate comes from the chaining of consecutive pipelines. In the One-Level estimate, which only models a single pipeline, the pipeline start-up costs have a more significant contribution and tend to reduce the strip size in order to minimize the total execution time. When chaining is taken into account, as in the Two-Level estimate, the overall contribution of the start-up phase is much less. For both machines examined, however, the communication *rate* did not have a significant effect on the Two-Level estimate since only small messages were communicated (the overhead of communication was two orders of magnitude greater than the rate). For programs in which larger amounts of data need to be communicated within the pipeline, the communication rate would most likely become more significant.

In Figure 13, speedup curves are shown for the measured data as well as that predicted by the Two-Level estimate (with the optimal point indicated for each). On the iPSC/860 the ratio of communication to computation is fairly high whereas this ratio is much lower on the

[(c)]The CM-5 vector units were not utilized for either the CMF or message passing runs since the node compilers would not produce vector code for the message passing programs.



(a) Implicit Hydrodynamics

(b) Successive Over-Relaxation

(c) Sparse Block Lower Triangular Solver

Figure 13: Speedup of Pipelining vs. Granularity

iPSC/2.[(d)] For this reason, the performance curves tend to be steeper on the iPSC/860, making the selection of the correct strip size more critical. In Figure 14, traces from the SOR kernel also show the correlation of the execution profile to the framework presented in Section 3.4.1.

Since there tends to be a range of strip sizes which all provide similar performance, predicting the trend is more important than predicting the exact time of the execution. With the correct trend, it is possible to automatically select a granularity that approaches the minimum execution time. Estimates for the iPSC/860 followed the trend quite well, but did not match the measured times so they are scaled in magnitude to facilitate comparison. This is expected due to the advanced optimizations performed by the i860 target compiler. The estimates for the iPSC/2 were fairly accurate when directly compared to the measured data (except for the BLTS estimate which was also scaled). For IMPL and SOR, communication is more tightly coupled with computation than in BLTS. For these programs, control of the granularity had a greater effect on the resulting performance. On the other hand, the computation in BLTS was not as fine grained and required little or no increase in grain size.

[(d)] Comparing the iPSC/860 to the iPSC/2 the change in communication overhead is about a factor of 5 while the change in computation is about a factor of 100.

Table 4: Data Partitioning for Pipeline Tests

Test	Array Sizes	Distribution
IMPL	512×512	∗, `block`
SOR	500×500	∗, `block`
BLTS	$5 \times 5 \times 32 \times 26 \times 5$	∗, ∗, `block`, ∗, ∗

Table 5: Comparison of Estimates (Strip Size, Time)

Machine	Test	L	Optimal		Two-Level		One-Level	
iPSC/860 (8 proc)	IMPL	100	24	7.75	14	8.37	7	10.09
	SOR	100	24	3.44	17	3.66	8	4.49
	BLTS	3	2	0.061	2	0.061	1	0.069
iPSC/2 (8 proc)	IMPL	10	10	19.98	7	20.06	3	20.96
	SOR	10	10	13.21	7	13.27	3	13.99
	BLTS	3	1	1.21	1	1.21	1	1.21
iPSC/2 (16 proc)	IMPL	10	10	11.16	8	11.19	3	12.10
	SOR	10	12	7.30	8	7.34	3	8.13
	BLTS	3	1	0.76	1	0.76	1	0.76

5. CONCLUSIONS

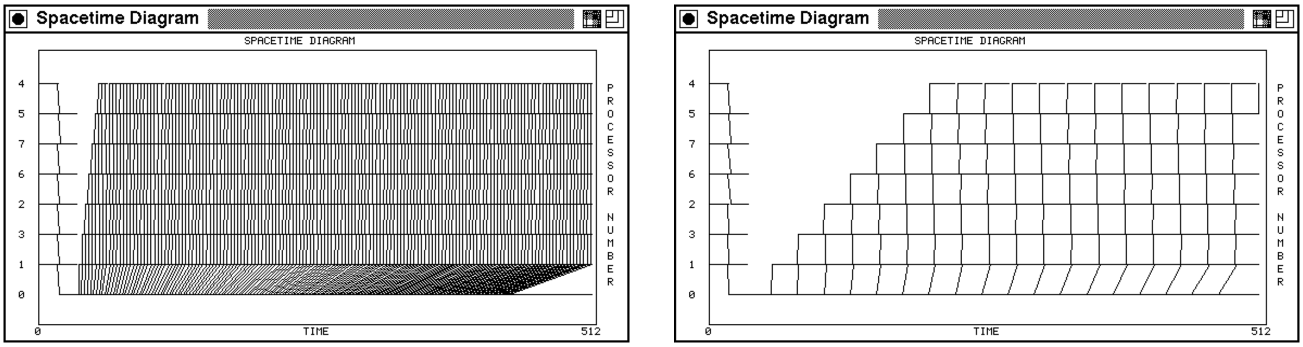
One of the most complex tasks facing a user in parallelizing serial programs is dealing with interprocessor communication. In this paper, it has been shown that this task can be performed by the compiler through the use of good estimates for communication and computation costs. It has also been shown that the application of the presented optimization techniques yields high performance on several different distributed-memory multicomputers.

By applying the presented optimizations, it was possible to amortize communication overhead obtaining near ideal performance. Through use of good computation and communication estimates, the compiler was able to select the best distribution for even small differences in performance. For larger machine sizes and more complex programs, the utility of automatic data distribution becomes more apparent as the communication costs become greater for inferior data distributions.

The performance of coarse grain pipelining is directly influenced by the relative costs of communication and computation on a given machine. The estimates presented in this paper allow for variation in both computational power as well as the communication latency and bandwidth of different machines. Comparing the estimated minimum to the measured data, it is apparent that the compiler is able to automatically select a granularity that gives near-optimal performance.

Currently, PARADIGM automatically performs loop bounds reduction, message coalescing, message vectorization, and message aggregation. The coarse grain pipeline transformation will be integrated into the compiler in the near future.

Acknowledgements: We would like to thank the reviewers for their helpful input, and to thank Antonio Lain, Christy Palermo, and Shankar Ramaswamy for their insightful comments and suggestions.



Fine Grain Coarse Grain

Figure 14: Traces of SOR with Pipelining

REFERENCES

[1] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD Distributed Memory Machines," *Communications of the ACM*, vol. 35, pp. 66–80, Aug. 1992.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, "Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers, Design, Implementation, and Performance Results," *Proceedings of the 1993 ACM International Conference on Supercomputing*, pp. 351–360, July 1993.

[3] S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 126–138, June 1993.

[4] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," in *Third Workshop on Compilers for Parallel Computers*, pp. 145–164, 1992.

[5] High Performance Fortran Forum, "High Performance Fortran Language Specification, version 1.0," Tech. Rep. CRPC-TR92225, CRPC, Rice University, Houston, TX, May 1993.

[6] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines," in *Proceedings of the 1992 ACM International Conference on Supercomputing*, (Washington, DC), July 1992.

[7] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors," in *Proceedings of the 1989 International Conference on Parallel Processing*, pp. II:39–48, Aug. 1989.

[8] E. Su, D. J. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers," to appear in the *1994 International Conference on Parallel Architectures and Compilation Techniques*, 1994.

[9] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 179–193, Mar. 1992.

[10] M. Gupta and P. Banerjee, "PARADIGM: A Compiler for Automated Data Partitioning on Multicomputers," in *Proceedings of the 1993 ACM International Conference on Supercomputing*, (Tokyo, Japan), July 1993.

[11] E. Su, D. J. Palermo, and P. Banerjee, "Automating Parallelization of Regular Computations for Distributed Memory Multicomputers in the PARADIGM Compiler," in *Proceedings of the 1993 International Conference on Parallel Processing*, pp. II:30–38, Aug. 1993.

[12] Parasoft Corporation, Pasadena, CA, *Express Reference Guide for FORTRAN Programmers*, 1992.

[13] G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, "PVM 3.0 User's Guide and Reference Manual," Oak Ridge National Laboratory, Oak Ridge, TN, Feb. 1993.

[14] M. T. Heath and J. A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, vol. 8, pp. 29–39, Sept. 1991.

[15] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "PICL: A Portable Instrumented Communication Library, C reference manual," Tech. Rep. ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[16] J. Li and M. Chen, "Compiling Communication-Efficient Programs for Massively Parallel Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 361–376, July 1991.

[17] S. Ramaswamy, , S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," to appear in the *1994 International Conference on Parallel Processing*, 1994.

[18] J. G. Holm, A. Lain, and P. Banerjee, "Compilation of Scientific Programs into Multithreaded and Message Driven Computation," to appear in the *1994 Scalable High Performance Computing Conference*, 1994.

[19] A. Lain and P. Banerjee, "Techniques to Overlap Computation and Communication in irregular Iterative Applications," to appear in the *ACM International Conference on Supercomputing 1994*, 1994.

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 256–266, May 1992.

[21] M. Gerndt, "Updating Distributed Variables in Local Computations," *Concurrency Practice and Experience*, vol. 2, pp. 171–193, Sept. 1990.

[22] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "An Interactive Environment for Data Partitioning and Distribution," in *Proceedings of the 5th Distributed Memory Computing Conference*, (Charleston, SC), pp. II,1160–1170, Apr. 1990.

[23] H. T. Kung, "Why Systolic Architectures?," *Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[24] D. I. Moldovan, *Parallel Processing: From Applications to Systems.* Morgan Kaufman, 1993.

[25] F. McMahon, "The Livermore Fortran Kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[26] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," Tech. Rep. RNR-91-002, NASA Ames Research Center, 1991.