Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines

Seema Hiranandani
Ken Kennedy
Chau-Wen Tseng

CRPC-TR 91196
November 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

In *Proceedings of the 1992 International Conference on Supercomputing*, Washington DC, July 1992.

Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines

Seema Hiranandani *seema@rice.edu* Ken Kennedy *ken@rice.edu* Chau-Wen Tseng *tseng@rice.edu*

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892
Tel: (713) 527-6077

Abstract

The Fortran D compiler uses data decomposition specifications to automatically translate Fortran programs for execution on MIMD distributed-memory machines. This paper introduces and classifies a number of advanced optimizations needed to achieve acceptable performance; they are analyzed and empirically evaluated for stencil computations. Profitability formulas are derived for each optimization. Results show that exploiting parallelism for pipelined computations, reductions, and scans is vital. Message vectorization, collective communication, and efficient coarse-grain pipelining also significantly affect performance.

1 Introduction

Parallel computing represents the only plausible way to continue to increase the computational power available to computational scientists and engineers. However, parallel computers are not likely to be widely successful until they are also easy to program. MIMD distributed-memory machines such as the Intel iPSC/860 present the most difficult programming model, since users must write message-passing programs that deal with separate address spaces, communication, and synchronization. Even worse, the resulting parallel programs are extremely machine-specific. Scientists are thus discouraged from utilizing these machines because they risk losing their investment whenever the program changes or a new architecture arrives.

To solve this problem, we have developed Fortran D, a version of Fortran enhanced with data decomposition specifications. We consider it to be one of the first of a new generation of *data-placement* programming languages. Its design was inspired by the observation that modern high-performance architectures demand that careful attention be paid to data placement by both the programmer and compiler. As one measure of its relevance, we note that features from Fortran D are being adopted by Cray Research, DEC, IBM, and Thinking Machines for programming their newest generation of parallel machines. Fortran D has also contributed to the development of High Performance Fortran (HPF), a new proposed Fortran standard.

Our goal with Fortran D is to provide a simple yet efficient machine-independent parallel programming model. By shifting much of the burden of machine-dependent optimization to the compiler, we allow the programmer to write data-parallel programs that can be compiled and executed with good performance on many different architectures. To evaluate the Fortran D programming model, we are implementing a prototype compiler in the context of the ParaScope programming environment [8].

Previous work described algorithms for partitioning data and computation in the Fortran D compiler, as well as its optimization and validation strategy [21]. Internal representations, program analysis, message vectorization, pipelining, and code generation algorithms were presented elsewhere [20]. The principal contribution of this paper is to introduce, classify, and evaluate (both empirically and analytically) new and existing compiler optimizations in a unified framework. The rest of this paper briefly reviews the Fortran D language and compiler before presenting each optimization, followed by empirical and analytical evaluations. It concludes by discussing the scalability of compiler optimizations, the overall optimization strategy, and a comparison with related work.

2 Fortran D Language

Fortran D provides users with explicit control over data partitioning using data *alignment* and *distribution* specifications. The `DECOMPOSITION` statement specifies an abstract problem or index domain. The `ALIGN` statement specifies fine-grain parallelism, mapping each array element onto one or more elements of the decomposition. This provides the minimal requirement for reducing data movement for the program given an unlimited number of processors. The alignment of arrays to decompositions is determined by their subscript expressions in the statement; perfect alignment results if no subscripts are used.

The `DISTRIBUTE` statement specifies coarse-grain parallelism, grouping decomposition elements and mapping them and aligned array elements to the finite resources of the physical machine. Each dimension of the decomposition is distributed in a block, cyclic, or block-cyclic manner; the symbol ":" marks dimensions that are not distributed. Both irregular and dynamic data decomposition are supported. Some sample data alignment and distributions are shown in Figure 1, the complete language is described in detail elsewhere [12].

3 Fortran D Compiler

There are two major steps in compiling Fortran D for MIMD distributed-memory machines. The first step is partitioning the data and computation across processors. The second is introducing communication to maintain the semantics of the program. A simple compilation technique known as *run-time resolution* yields code that explicitly calculates the ownership and communication for each reference at run time [9, 32, 38], but resulting programs are likely to execute significantly slower than the original sequential code.

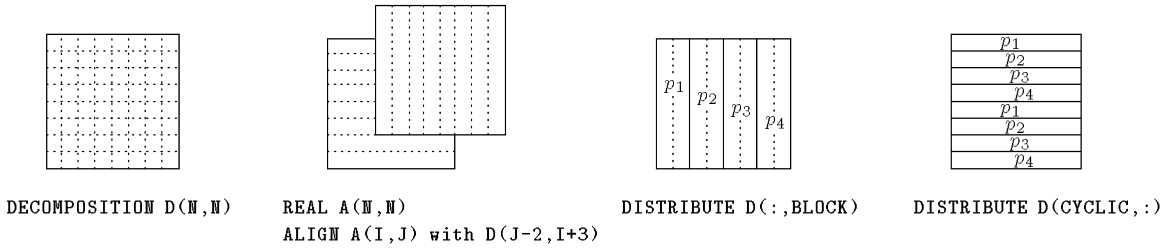


Figure 1: Fortran D Data Decomposition Specifications

```
{* Fortran D Program *}
REAL X(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X WITH D
DISTRIBUTE D(BLOCK)
do i = 1,95
  X(i) = X(i+5)
enddo
```

```
{* Run-time Resolution *}
REAL X(100)
my$p = myproc()  {* 0...3 *}
do i = 1,95
  if (my$p .eq. owner(X(i+5))) then
    send(X(i+5),owner(X(i)))
  endif
  if (my$p .eq. owner(X(i))) then
    recv(X(i+5),owner(X(i+5)))
    X(i) = X(i+5)
  endif
enddo
```

```
{* Compile-time Analysis & Optimization *}
REAL X(30)
my$p = myproc()  {* 0...3 *}
if (my$p .gt. 0) send(X(1:5),my$p-1)
if (my$p .lt. 3) recv(X(26:30),my$p+1)
ub$1 = min((my$p+1)*25,95)-(my$p*25)
do i = 1,ub$1
  X(i) = X(i+5)
enddo
```

Figure 2: Fortran D Compilation

Compile-time analysis and optimization performed in the Fortran D compiler can generate much more efficient programs, as demonstrated in Figure 2. We present a brief overview of the Fortran D compilation algorithm below. The details are discussed elsewhere [20, 21]:

1) Analyze program The Fortran D compiler performs scalar dataflow analysis, symbolic analysis, and dependence testing to determine the type and level of all data dependences [25].

2) Partition data The compiler analyzes Fortran D data decomposition specifications to determine the decomposition of each array in a program. Alignment and distribution statements are used to calculate the array section owned by each processor.

3) Partition computation The compiler partitions computation across processors using the "owner computes" rule—where each processor only computes values of data it owns [9, 32, 38]. The left-hand side (*lhs*) of each assignment statement in a loop nest is used to calculate the set of loop iterations that cause a processor to assign to local data. This represents the work that must be performed by the processor.

4) Analyze communication Once the work partition is computed, it is used to calculate for each right-hand side (*rhs*) reference to a distributed array the nonlocal data accessed by each processor. References that result in nonlocal accesses are marked since they require communication to be inserted.

5) Optimize communication The compiler examines each marked nonlocal reference, using results of data decomposition, symbolic and dependence analysis to determine the legality of optimizations to reduce communication costs. *Regular section descriptors* (RSDs) are built for the sections of data to be communicated. They compactly represent rectangular array sections and their higher dimension analogs [18].

6) Manage storage The compiler collects the extent and type of nonlocal data accesses represented by RSDs to calculate the storage required for nonlocal data. For RSDs representing array elements contiguous to the local array section, the compiler reserves storage using *overlaps* created by extending the local array bounds [15]. Otherwise temporary buffers or hash tables are used for storing instances of nonlocal data.

7) Generate code Finally, the Fortran D compiler uses the results of previous analysis and optimization to generate the *single-program, multiple-data* (SPMD) program with explicit message-passing that executes directly on the nodes of the distributed-memory machine. Array and loop bounds are reduced and guards are introduced to instantiate the data and computation partitions. RSDs representing nonlocal data accesses are used to generate buffer routines, *send* and *recv* calls, and collective communication routines. The compiler applies run-time resolution techniques to references not analyzable at compile time.

4 Compiler Optimizations

The goal of the Fortran D compiler is to generate a parallel program with low communication overhead and storage requirements. We classify each Fortran D compiler optimization by its ability to reduce communication startup costs, hide message copy and transit times, exploit parallelism, and reduce storage. Figure 3 lists Fortran D compiler optimizations in each category.

In the following sections we describe each optimization and provide motivating examples using a small selection of scientific program kernels adapted from the Livermore Kernels and finite-difference algorithms [30]. They contain *stencil computations* and reductions, techniques commonly used by scientific programmers to solve partial differential equations (PDEs) [7, 13].

For clarity we ignore boundary conditions and use constant loop bounds and machine size in the examples, though this is not required by the optimizations. We have also equalized sizes for two dimensional problems used in our Livermore examples (*e.g.,* $n \times n$ instead of $7 \times n$ data arrays).

4.1 Reducing Communication Overhead

Many compiler optimizations focus on reducing communication overhead. Computer architects usually characterize communication by latency and bandwidth. For evaluating compiler optimizations, we find it convenient to divide communication overhead into three components:

1. Reducing Communication Overhead
   (a) Message Vectorization
   (b) Message Coalescing
   (c) Message Aggregation
   (d) Collective Communication
2. Hiding Communication Overhead
   (a) Message Pipelining
   (b) Vector Message Pipelining
   (c) Iteration Reordering
   (d) Nonblocking Messages
3. Exploiting Parallelism
   (a) Partitioning Computation
   (b) Reductions and Scans
   (c) Dynamic Data Decomposition
   (d) Pipelining Computations
4. Reducing Storage
   (a) Partitioning Data
   (b) Message Blocking

Figure 3: Fortran D Compiler Optimizations

- T_{start}, the startup time to send & receive messages.
- $T_{copy}(n)$, the time to copy a message of size n into & out of the program address space.
- $T_{transit}(n)$, the transit time for a message of size n between processors.

We assume T_{start} is relatively fixed with respect to message size, but that both T_{copy} and $T_{transit}$ grow with n. Using this communication model we can cast latency as $T_{start} + T_{copy}(1) + T_{transit}(1)$ and bandwidth as $n/(T_{copy}(n) + T_{transit}(n) - T_{transit}(1))$.

We begin with optimizations to reduce T_{start}, the startup cost incurred to access nonlocal data. For most MIMD distributed-memory machines, the cost to send the first byte is significantly higher than the cost for additional bytes. For instance, the Intel iPSC/860 requires approximately 95 μsec to send one byte versus .4 μsec for each additional byte [6]. The following optimizations seek to reduce T_{start} by combining or eliminating messages.

Message vectorization *Message vectorization* uses the results of data dependence analysis [1, 28] to combine element messages into vectors. It first calculates *commlevel*, the level of the deepest loop-carried true dependence or loop enclosing a loop-independent true dependence. This determines the outermost loop where element messages resulting from the same array reference may be legally combined [3, 15]. Vectorized nonlocal accesses are represented as RSDs and stored at the loop at commlevel. They eventually generate messages at loop headers for loop-carried RSDs and in the loop body for loop-independent RSDs.

Message coalescing Once nonlocal accesses are vectorized at outer loops, the compiler applies *message coalescing* to avoid communicating redundant data. It compares RSDs from different references to the same array, merging RSDs that contain overlapping or contiguous elements. If two overlapping RSDs cannot be be coalesced without loss of precision, they are split into smaller sections in a manner that allows the overlapping regions to be merged precisely.

For instance, consider the kernel in Figure 4. Communication analysis discovers that references U(k+1)...U(k+6) access nonlocal data. These references do not cause loop-carried true dependences, so their nonlocal RSDs are vectorized outside both the l and k loops, resulting in the RSDs

```
{* Fortran D Program *}
REAL U(100), X(100), Y(100), Z(100), R, T
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN U, X, Y, Z with D
DISTRIBUTE D(BLOCK)
do l = 1,time
  do k = 1,94
    X(k) = F(Z(k),Y(k),U(k)...U(k+6))
end
{* Compiler Output *}
REAL U(31), X(25), Y(25), Z(25), R, T
my$p = myproc() {* 0...3 *}
if (my$p .gt. 0) send(U(1:6),my$p-1)
if (my$p .lt. 3) recv(U(26:31),my$p+1)
do l = 1,time
  do k = 1,25
    X(k) = F(Z(k),Y(k),U(k)...U(k+6))
end
```

Figure 4: Livermore 7–Equation of State Fragment

[26:26]...[26:31]. These RSDs may be coalesced without loss of precision, resulting in [26:31]. Calls to copy routines are inserted during code generation to pack noncontiguous data into message buffers, but is not needed for this example since the data is contiguous. Finally, explicit *send* and *recv* statements are placed at loops headers to communicate nonlocal data [20].

Message aggregation Message coalescing ensures that each data value is sent to a processor only once. In comparison, *message aggregation* ensures that only one message is sent to each processor, possibly at the expense of extra buffering. After message vectorization and coalescing, the Fortran D compiler locates and aggregates all RSDs representing data being sent to the same processor. During code generation, these array sections are copied to a single buffer so that they may be sent as one message. The receiving processor then copies the buffered data back to the appropriate locations.

Consider the kernel in Figure 5. The lack of true dependences for nonlocal references to ZP, ZQ, and ZM allows their communication to be vectorized and coalesced outside the l loop. The compiler discovers they are being sent one processor to the right, and aggregates them in the same message. Nonlocal references to ZR and ZZ cause true dependences carried on the l loop. Their nonlocal accesses are vectorized and communications inserted in loop l just after the loop header, allowing values from the previous iteration to be fetched at the beginning of each new iteration. Once RSDs are placed at the header, the compiler easily recognizes that the messages may be coalesced and aggregated as one message each for the left and right processors. Finally, nonlocal references to ZA cause loop-independent dependences. Communication is vectorized and coalesced at the level of the l loop, since it is the only loop common to the endpoints of the dependence. Messages are later inserted in front of the loop accessing ZA.

Collective communication Message overhead can also be reduced by utilizing fast *collective communication*, such as broadcast, all-to-all, or transpose, instead of generating individual messages [6, 29]. Collective communication opportunities are recognized by comparing the subscript expression of each distributed dimension in the *rhs* with the aligned dimension in the *lhs* reference. For instance, loop-invariant subscripts in distributed array dimensions correspond to broadcasts, and differing alignment between the *lhs* and *rhs* may require transpose or broadcast. Collective communication routines are also useful for accumulating results of *reductions* and *scans*, as shown later in Section 4.3.

```
{* Fortran D Program *}
REAL ZP(100,100), ZQ(100,100), ZM(100,100), S, T
REAL ZR(100,100), ZZ(100,100), ZA(100,100)
REAL ZU(100,100), ZV(100,100), ZB(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN ZA, ZB, ZM, ZP, ZQ, ZR, ZU, ZV, ZZ with D
DISTRIBUTE D(BLOCK,:)
do l = 1,time
  do k = 2,99
    do j = 2,99
      ZA(j,k) = F1(ZP(j-1,k),ZQ(j-1,k),ZR(j-1,k),...)
      ZB(j,k) = F2(ZP(j-1,k),ZQ(j-1,k),...)
  do k = 2,99
    do j = 2,99
      ZU(j,k) = F3(ZZ(j-1,k),ZZ(j+1,k),ZA(j-1,k),...)
      ZV(j,k) = F4(ZR(j-1,k),ZR(j+1,k),ZA(j-1,k),...)
  do k = 2,99
    do j = 1,25
      ZR(j,k) = F5(ZR(j,k),ZU(j,k))
      ZZ(j,k) = F6(ZZ(j,k),ZV(j,k))
end
{* Compiler Output *}
REAL ZP(0:25,100), ZQ(0:25,100), ZM(0:25,100), S, T
REAL ZR(0:26,100), ZZ(0:26,100), ZA(0:25,100)
REAL ZU(25,100), ZV(25,100), ZB(25,100)
my$p = myproc() {* 0...3 *}
if (my$p .lt. 3)
  send(ZP(25,2:100),ZQ(25,2:100),ZM(25,2:100),my$p+1)
if (my$p .gt. 0)
  recv(ZP(0,2:100),ZQ(0,2:100),ZM(0,2:100),my$p-1)
do l = 1,time
  if (my$p .gt. 0) send(ZR(1,2:99),ZZ(1,2:99),my$p-1)
  if (my$p .lt. 3) send(ZR(25,2:99),ZZ(25,2:99),my$p+1)
  if (my$p .lt. 3) recv(ZR(26,2:99),ZZ(26,2:99),my$p+1)
  if (my$p .gt. 0) recv(ZR(0,2:99),ZZ(0,2:99),my$p-1)
  do k = 2,99
    do j = 1,25
      ZA(j,k) = F1(ZP(j-1,k),ZQ(j-1,k),ZR(j-1,k),...)
      ZB(j,k) = F2(ZP(j-1,k),ZQ(j-1,k),...)
  if (my$p .lt. 3) send(ZA(25,2:99),my$p+1)
  if (my$p .gt. 0) recv(ZA(0,2:99),my$p-1)
  do k = 2,99
    do j = 1,25
      ZU(j,k) = F3(ZZ(j-1,k),ZZ(j+1,k),ZA(j-1,k),...)
      ZV(j,k) = F4(ZR(j-1,k),ZR(j+1,k),ZA(j-1,k),...)
  do k = 2,99
    do j = 1,25
      ZR(j,k) = F5(ZR(j,k),ZU(j,k))
      ZZ(j,k) = F6(ZZ(j,k),ZV(j,k))
end
```

Figure 5: Livermore 18–Explicit Hydrodynamics

4.2 Hiding Communication Overhead

The previous section discussed techniques to decrease communication costs by reducing T_{start}. This section presents optimizations to hide $T_{transit}$, the message transit time, by overlapping communication with computation. The same optimizations can also hide T_{copy}, the message copy time, by using *nonblocking messages.*

Message pipelining *Message pipelining* inserts a *send* for each nonlocal reference as soon as it is defined [32]. The *recv* is placed immediately before the value is used. Any computation performed between the definition and use of the value can then help hide $T_{transit}$. Unfortunately, message pipelining prevents optimizations such as message vectorization, resulting in significantly greater total communication cost. It is thus generally undesirable for completely parallel programs, but may be useful for exploiting parallelism for pipelined computations, as shown in Section 4.3.

Vector message pipelining We describe a new optimization, *vector message pipelining*, that hides $T_{transit}$ without increasing total communication cost. After message vectorization, pairs of vectorized *send* and *recv* statements have been gathered either inside or outside of loop headers.

```
{* Fortran D Program *}
REAL V(1000,1000)
PARAMETER (n$proc = 10)
DECOMPOSITION D(1000,1000)
ALIGN V with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  {* compute red points *}
  do j = 3,999,2
    do i = 3,999,2
S1    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
  do j = 2,998,2
    do i = 2,998,2
S2    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
  {* compute black points *}
  do j = 2,998,2
    do i = 3,999,2
S3    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
  do j = 3,999,2
    do i = 2,998,2
S4    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
end
{* Compiler Output *}
REAL V(1000,0:101)
my$p = myproc() {* 0...9 *}
if (my$p .lt. 9) send(m$3,V(3:999:2,100),my$p+1)
do l = 1,time
  if (my$p .gt. 0) send(m$4,V(2:998:2,1),my$p-1)
  if (my$p .gt. 0) recv(m$3,V(3:999:2,0),my$p-1)
  do j = 1,99,2
    do i = 3,999,2
S1    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
  if (my$p .gt. 0) send(m$1,V(3:999:2,1),my$p-1)
  if (my$p .lt. 9) recv(m$4,V(2:998:2,101),my$p+1)
  do j = 2,100,2
    do i = 2,998,2
S2    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
  if (my$p .lt. 9) send(m$2,V(2:998:2,100),my$p+1)
  if (my$p .lt. 9) recv(m$1,V(3:999:2,101),my$p+1)
  do j = 2,100,2
    do i = 3,999,2
S3    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
  if (my$p .lt. 9) send(m$3,V(3:999:2,100),my$p+1)
  if (my$p .gt. 0) recv(m$2,V(2:998:2,0),my$p-1)
  do j = 1,99,2
    do i = 2,998,2
S4    V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
if (my$p .gt. 0) recv(m$3,V(3:999:2,0),my$p-1)
end
```

Figure 6: Red-Black SOR

Vector message pipelining uses data dependence information to move vector *send* and *recv* statements towards their definitions and uses respectively in order to hide $T_{transit}$.

Vector message pipelining may be considered to be *macro-instruction scheduling*, where macro-instructions consist of vectorized *send*, *recv* statements and entire inner loop nests. Since *send* and *recv* statements interlock, they must be scheduled apart in order to avoid idle cycles. A simple application of vector message pipelining is to invoke all *send* statements before *recv* when a number of messages are sent at the same time.

Figure 6, Red-Black successive over-relaxation (SOR), demonstrates a more complex case. Values of red points computed by statement S_1 are used by S_3, corresponding to a loop-independent true dependence from S_1 to S_3. Similarly, S_2 computes red points used by S_4. Message vectorization creates communication for S_3 and S_4 at the level of the l loop, since it is the deepest loop enclosing these loop-independent dependences. We label these messages as `m$1` and `m$2`. Vector message pipelining then places the vector *send* for S_3 and S_4 after the j loops enclosing S_1 and S_2, respectively, using them to hide $T_{transit}$.

In addition, values of black points computed by statement S_3 are used by S_1, corresponding to a loop-carried true dependence from S_3 to S_1. Similarly, S_4 computes black points used by S_2. Message vectorization creates communication for S_1 and S_2 at the level of the l loop, since it is the loop with the deepest loop-carried dependences. We label these messages as `m$3` and `m$4`. Vector message pipelining places the vector *recv* for S_2 just before the j loop enclosing S_2, using S_1 to hide $T_{transit}$.

Hiding transit time for the values needed by S_1 is more complicated, since the communication needs to cross iterations of the l loop. Scheduling *send* and *recv* statements across iterations of the outer time-step loop is analogous to *macro-software pipelining*. Vector message pipelining places the vector *send* for S_1 after S_3, using S_4 to hide $T_{transit}$. Matching vector *send* and *recv* statements must be inserted outside the loop. The final result is shown in Figure 6.

Iteration reordering Red-Black SOR is a computation structured so that careful placement of vector *send* and *recv* statements using vector message pipelining can effectively hide communication costs. Where this is not the case, *iteration reordering* may be applied to change the order of program execution, subject to dependence constraints. This allows loop iterations accessing only local data to be separated and placed between *send* and *recv* statements to hide $T_{transit}$ [26].

We demonstrate how the Fortran D compiler finds local loop iterations for the Jacobi algorithm in Figure 7. First, communication analysis calculates [2:99,0] and [2:99,26] to represent nonlocal accesses to array B. These accesses are caused by the references B(j,i-1) and B(j,i+1). Applying their inverse subscript functions yields the iteration sets [1,2:99] and [25,2:99]. Subtracting these nonlocal iterations from the full iteration set yields [2:24,2:99] as the set of local loop iterations. These iterations are placed into a separate loop nest between the vector *send* and *recv* statements to hide $T_{transit}$. More aggressive iteration reordering would also extract iterations from the second j loop to be placed before the vector *recv* statement.

Nonblocking Messages The Fortran D compiler generally uses *blocking messages*. Invoking a blocking *send* causes the calling process to block until the data has been copied out of the program address space into a system buffer. Upon return the process may overwrite the original data. This does not mean that the process must wait for the message to be actually received by another processor, just that the content of the message is no longer affected by the sending processor. Invoking a blocking *recv* causes the calling process to block until the data has been received and copied into the program address space.

Nonblocking messages, provided by machines such as the iPSC/860, permit computation and message copying to be performed in parallel on the same processor. A nonblocking *send* returns immediately, allowing computation to be performed on the sending processor concurrently with copying the data into a system buffer. A nonblocking *recv* posts a message destination, enabling computation to be performed on the receiving processor while the data is being received and copied. It also avoids an extra copy into a system buffer, since the message body may be placed directly at the posted address. An additional system call must be made for each nonblocking message to block the computation until the copy is complete.

Vector message pipelining and iteration reordering with blocking messages can only hide $T_{transit}$, since the processor must remain idle while copying the data. By using nonblocking messages, the Fortran D compiler also hides T_{copy}, the message copy time. This is important since copying is a major component of communication overhead for large messages. However, nonblocking messages should be utilized selectively. Message startup time for nonblocking messages is generally higher than for blocking messages, since the number of system calls is doubled. Nonblocking *sends* may also require multiple buffers for noncontiguous data. Note that in our model the only source of savings for a nonblocking *send* is the time to copy data to the system buffer. After the copy is performed, both blocking and nonblocking messages can overlap communication and computation.

```
{* Fortran D Program *}
REAL A(100,100), B(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  do j = 2,99
    do i = 2,99
      A(i,j) = F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
  do j = 1,99
    do i = 2,99
      B(i,j) = A(i,j)
end
{* Compiler Output *}
REAL A(100:25), B(100,0:26)
my$p = myproc() {* 0...3 *}
do l = 1,time
  if (my$p .lt. 3) send(B(2:99,1),my$p-1)
  if (my$p .gt. 0) send(B(2:99,25),my$p+1)
  {* perform local iterations *}
  do j = 2,24
    do i = 2,99
      A(i,j) = F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
  if (my$p .lt. 3) recv(B(2:99,26),my$p+1)
  if (my$p .gt. 0) recv(B(2:99,0),my$p-1)
  {* perform non local iterations *}
  do j = 1,25,24
    do i = 2,99
      A(i,j) = F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
  do j = 1,25
    do i = 2,99
      B(i,j) = A(i,j)
end
```

Figure 7: Jacobi

4.3 Exploiting Parallelism

Parallelism optimizations restructure the computation or communication to increase the amount of useful computation that may be performed in parallel.

Partitioning computation Most scientific applications are completely parallel in either a *synchronous* or *loosely synchronous* manner [13]. If this can be determined by the compiler, partitioning the computation using the "owner computes" rule yields a fully parallel program. To successfully exploit parallelism in these basic cases, the compiler must be able to intelligently partition the work at compile-time. The Fortran D compiler achieves this through *loop bounds reduction* and *guard introduction* [20, 21].

An exception to the "owner computes" rule must be made for *private* variables, scalars or arrays that are only defined and used in the same loop iteration. Since private variables are usually replicated, naive compilation would cause their computation to be performed on all processors. The Fortran D compiler needs to recognize these private variables and partition their computation based on where their values are used [20].

Compile-time partitioning of parallel computations is key to any reasonable compilation system, and should not really be considered an optimization. *Cross-processor* dependences point out sequential components of the computation that cross processor boundaries. These dependences disable parallel execution by forcing processors to remain idle, waiting for their predecessors to finish computing. We show

```
{* Fortran D Program *}
REAL X(100), Z(100), Q
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, Z with D
DISTRIBUTE D(BLOCK)
do l = 1,time
  Q = 0.0
  do k = 1,100
    Q = Q + Z(k)*X(k)
end
```

```
{* Compiler Output *}
REAL X(25), Z(25), Q
do l = 1,time
  Q = 0.0
  do i = 1,25
    Q = Q + Z(k)*X(k)
  Q = global-sum(Q)
end
```

Figure 8: Livermore 3–Inner Product

```
{* Fortran D Program *}
REAL X(100), Y(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, Y with D
DISTRIBUTE D(BLOCK)
do l = 1,time
  X(1) = Y(1)
  do k = 2,100
    X(k) = X(k-1) + Y(k)
end
```

```
{* Compiler Output *}
REAL X(25), Y(25), S(0:3)
my$p = myproc() {* 0...3 *}
do l = 1,time
  S(my$p) = 0.0
  do k = 1,25
    S(my$p) = S(my$p) + Y(k)
  global-concat(S)
  X(1) = Y(1)
  if (my$p .ne. 0) then
    do k = 0,my$p-1
      X(1) = X(1) + S(k)
  endif
  do k = 2,25
    X(k) = X(k-1) + Y(k)
end
```

Figure 9: Livermore 11–First Sum

```
{* Fortran D Program *}
REAL A(100), B(100), X(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN X with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  {* Phase 1: sweep along columns *}
  do j = 1, 100
    do i = 2, 100
      X(i,j) = F1(X(i,j),X(i-1,j),A(i),B(i))
  {* Phase 2: sweep along rows *}
  do j = 2, 100
    do i = 1, 100
      X1(i,j) = F2(X(i,j),X(i,j-1),A(i),B(i))
end
```

```
{* Compiler Output *}
REAL A(100), B(100), X(25,100), X1(100,25)
EQUIVALENCE (X,X1)
do l = 1,time
  do j = 1, 25
    do i = 2, 100
      X(i,j) = F1(X(i,j),X(i-1,j),A(i),B(i))
  redistribute-row-to-col(X)
  do j = 2, 100
    do i = 1, 25
      X1(i,j) = F2(X(i,j),X(i,j-1),A(i),B(i))
  redistribute-col-to-row(X1)
end
```

Figure 10: ADI Integration

how optimizations may extract parallelism in the presence of cross-processor dependences.

Reductions and scans Some computations with cross-processor dependences may be parallelized directly. *Reductions* are associative and commutative operations that may be applied to a collection of data to return a single result. For instance, a sum reduction would compute and return the sum of all elements of an array. *Scans* are similar but perform parallel-prefix operations instead. A sum scan would return the sums of all the prefixes of an array. Scans may be used to solve a number of computations in scientific codes, including linear recurrences and tridiagonal systems [11, 27].

The Fortran D compiler applies dependence analysis to recognize reductions and scans. If the reduction or scan accesses data in a manner that sequentializes computation across processors, the Fortran D compiler may parallelize it by relaxing the "owner computes" rule and providing methods to combine partial results. This requires changing the order in which computations are performed, which is why the operations must be both associative and commutative.

Reductions are parallelized by allowing each processor to compute in parallel, later accumulating the partial results. Communication using individual *send/recv* calls can be used to calculate the global result. Broadcast may be used in place of *send* for efficiency, and specialized collective communication routines such as *global-sum*() can reduce communication overhead even further for common reductions. Figure 8 shows how a sum reduction may be parallelized using a *global-sum* collective communication routine to combine the partial sums.

Scans may also be parallelized by reordering operations. Each processor first computes its local values in parallel, then communicates the partial results to all other processors. The global data is used to update local results. Though extra communication and computation is introduced during parallelization, the improvement in parallelism yields major performance improvements. Figure 9 demonstrates how a prefix sum may be computed, using a *global-concat* collective communication routine to collect the partial sums from each processor in S. The partial sums of all preceding processors are combined locally and used as a basis for computing local prefix sums [11].

Dynamic Data Decomposition Other computations contain parallelism, but are partitioned by the "owner computes" rule in a way that causes sequential execution. In these cases *dynamic data decomposition* may be used to temporarily change the ownership of data during program execution, exposing parallelism by *internalizing* cross-processor dependences [3].

For instance, consider the two substitution phases in the Alternating Direction Implicit (ADI) integration example in Figure 10. The computation wavefront only crosses one spatial dimension in each phase. A fixed column or row data distribution would result in one parallel and one sequential phase. By applying dynamic data decomposition using collective communication routines to change the array decomposition after each phase, the Fortran D compiler can internalize the computation wavefront in both phases, allowing processors to execute in parallel without communication [26].

However, dynamic data decomposition is only applicable when there are full dimensions of parallelism available in the computation. For instance, it cannot be used to exploit parallelism for SOR or Livermore 23 in Figure 12, because the computation wavefront crosses both spatial dimensions. Even when dynamic data decomposition is applicable, it may not be efficient, as shown in Section 6.

Pipelining computations For many computations containing cross-processor dependences, a technique known as *pipelining* can extract partial parallelism. Consider the difference in program execution between parallel and *pipelined computations* illustrated in Figure 11. Solid lines denote computation, and dotted arrows represent communication from sender to recipient. For parallel computations, all processors can execute concurrently, communicating data when necessary. In pipelined computations, a processor cannot begin execution until it receives results computed by its predecessor. However, by sending partial results to their successors earlier, processors may overlap their computations. When used in this fashion, messages both transmit data and

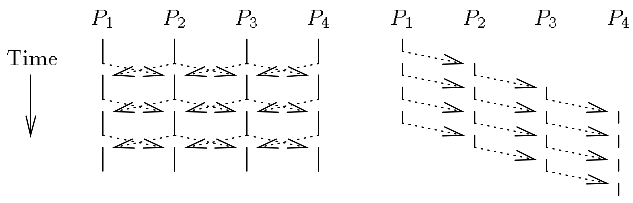


Figure 11: Parallel & Pipelined Computations

```
{* Fortran D Program *}
REAL ZA(100,100), ZB(100,100), ZR(100,100), QA
REAL ZU(100,100), ZV(100,100), ZZ(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN ZA, ZB, ZR, ZU, ZV, ZZ with D
DISTRIBUTE D(:,BLOCK)
do l = 1,time
  do j = 2,99
    do k = 2,99
S1      QA = F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
S2      ZA(k,j) = F2(ZA(k,j),QA)
end

{* Compiler Output 1:  Fine-grain pipelining *}
REAL ZA(100,0:26), ZB(100,25), ZR(100,25), QA
REAL ZU(100,25), ZV(100,25), ZZ(100,25)
my$p = myproc() {* 0...3 *}
do l = 1,time
  if (my$p .gt. 0) send(ZA(2:99,1),my$p-1)
  if (my$p .lt. 3) recv(ZA(2:99,26),my$p+1)
  do k = 2,99
    if (my$p .gt. 0) recv(ZA(k,0),my$p-1)
    do j = 1,25
      QA = F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
      ZA(k,j) = F2(ZA(k,j),QA)
    if (my$p .lt. 3) send(ZA(k,25),my$p+1)
end

{* Compiler Output 2:  Coarse-grain pipelining *}
REAL ZA(100,0:26), ZB(100,25), ZR(100,25), QA
REAL ZU(100,25), ZV(100,25), ZZ(100,25)
my$p = myproc() {* 0...3 *}
do l = 1,time
  if (my$p .gt. 0) send(ZA(2:99,1),my$p-1)
  if (my$p .lt. 3) recv(ZA(2:99,26),my$p+1)
  do kk = 2,99,B
    if (my$p .gt. 0) recv(ZA(kk:kk+B-1,0),my$p-1)
    do j = 1,25
      do k = kk,kk+B
        QA = F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
        ZA(k,j) = F2(ZA(k,j),QA)
    if (my$p .lt. 3) send(ZA(kk:kk+B-1,25),my$p+1)
end
```

Figure 12: Livermore 23–Implicit Hydrodynamics

serve as data synchronization. The degree of pipeline parallelism depends on how soon each processor is able to begin work after its predecessor starts.

The Fortran D compiler can distinguish pipelined computations from fully parallel computations by discovering *cross-processor loops*—loops that cause computation wavefronts to sweep across processor boundaries [20]. The compiler finds cross-processor loops as follows. First, it considers all pairs of array references that cause loop-carried true dependences. If non-identical subscript expressions occur in a distributed dimension of the array, all loop index variables appearing in the subscript expressions belong to cross-processor loops. In most cases, cross-processor loops are loops carrying true dependences whose iterations have been partitioned across processors.

Consider the loop-carried true dependence between S_1 and S_2 caused by ZA(k,j) and ZA(k,j-1) in Figure 12. Since the second dimension of ZA is distributed, the compiler compares j and $j-1$, the subscripts in the second dimension. These are not identical, so the j loop is labeled as cross-processor. No other loops are cross-processor.

The presence of any cross-processor loop in a loop nest indicates that it is a pipelined computation. The granularity of pipeline parallelism is determined by the amount of computation enclosed by cross-processor loops. *Fine-grain pipelining* interchanges all cross-processor loops as deeply as possible, so that they enclose the least amount of computation. The resulting program execution order generates values needed by other processors in the shortest time, achieving the finest granularity of pipelining. Unfortunately, it also results in high message overhead since a message is sent for each iteration accessing nonlocal data.

Coarse-grain pipelining remedies this problem by applying loop interchange and strip-mining to adjust the amount of computation $\mathcal{C}$ enclosed by cross-processor loops. Increasing $\mathcal{C}$ reduces communication, since all nonlocal data accessed by $\mathcal{C}$ may be communicated in a single message. However, parallelism is also reduced since processors must wait longer before beginning to compute. Section 6 discusses how to choose an efficient granularity for pipelining based on the ratio between computation and communication costs.

Figure 12 shows examples of both fine and coarse-grained pipelining. Fine-grain pipelining interchanges the cross-processor loop j to the innermost position to maximize pipelining. In comparison, coarse-grain pipelining strip-mines the k loop by a factor B, then interchanges the iterator loop kk outside the j loop. This allows communication for B iterations to be vectorized at the j loop.

The legality of loop interchange and strip-mine is determined exactly as for shared-memory programs [1, 25, 28]. The Fortran D compiler first permutes loops in *memory order* to exploit data locality on individual processors [24], then applies coarse-grain pipelining to adjust the degree of pipeline parallelism.

4.4 Reducing Storage

Most optimizations increase the amount of temporary storage required by the program. *Storage optimizations* seek to reduce storage requirements. Compile-time partitioning of the data so that each processor allocates memory only for array sections owned locally is fundamental. Otherwise the problem size is limited by the amount of data that can be place on a single processor. We view *partitioning data* as fundamental for any reasonable compiler; like partitioning computation, it should not be merely viewed as an optimization.

Once data has been partitioned, storage must be provided for nonlocal data. Choosing overlaps, buffers, or hash tables for storing nonlocal data involve many tradeoffs. Overlaps are the most convenient, but also consume the most space. Buffers may be preferred because they may be reused. If insufficient storage is available, *message blocking* strip-mines loops by a block factor B. Each vectorized message of size n is then divided into n/B messages of size B. This reduces the buffer space required by a factor of n/B at the expense of additional messages.

5 Empirical Performance Evaluation

To evaluate the usefulness of each compiler optimization, we applied them where appropriate to the Livermore and PDE kernels used as examples in this paper. SOR is simply one of the four inner loop nests in Red-Black SOR with unit step. Table 1 shows the optimized versions of each program. Message vectorization, coalescing, aggregation, and fine-grain pipelining were applied by the prototype Fortran D compiler; other optimizations were performed by hand. *Nc* is a parallel version of the program with all communication removed. It is meant to provide a baseline for measuring communication overhead. We also use $nc \times P$ to estimate the sequential execution time, since most problem sizes are too large for a single processor. Parallel speedup is then simply $\frac{nc \times P}{time}$.

The experiments were performed on a 32 node Intel iPSC/860 with 8 Meg of memory per node. Each program was compiled under -O4 using Release 2.0 of *if77*, the iPSC/860 compiler. Timings were taken using *dclock*() for one iteration of *l*, the time step loop. The results are presented in milliseconds for several machine and problem sizes. P indicates the number of processors. N describes the total problem size and its dimensionality; N/P yields the problem size on an individual processor. All arrays are double precision and distributed block-wise in one dimension. In addition to the timings, each table contains ratios of execution times for some selected optimizations, illustrating their relative usefulness.

5.1 Optimizations for Communication Overhead

We begin by measuring the effect of optimizations to reduce and hide communication overhead. We found that the nature of the computation and data partition significantly affects the utility of each optimization. For instance, we omitted execution times for Livermore 7, a 1D stencil computation, since data movement is limited and optimizations have little effect for reasonable problem sizes. The three kernels presented in Table 2 are 2D stencil computations with 1D data distributions. Enough communication is required to make optimizations significant.

For these stencil computations, message vectorization is clearly the most important optimization. The numbers computed for $\frac{mp}{mv}$ (2.1–8.9) demonstrate that message vectorization significantly improves execution compared to sending element messages. Message aggregation provides a small fixed gain. Vector message pipelining and iteration reordering help, but are most effective when used in tandem with nonblocking messages. Nonblocking messages alone are insufficient, since the original program may not provide enough computation to hide all copying costs. Optimizations to hide communication lose effectiveness for small problem sizes, since insufficient computation exists to hide all message copy and transit overhead. Optimizations should not be applied in all cases. For instance, iteration reordering actually degraded performance for Livermore 18.

To evaluate the profitability of optimizations beyond message vectorization, we compute $\frac{mv}{best}$, where *best* is defined as the best time among all optimizations. The results show that other optimizations can improve somewhat on message vectorization (1.1–2.6), but the differences are less dramatic and drop quickly with increasing problem size. From $\frac{best}{nc}$ we see that optimizations can reduce communication overhead to a small percentage of total computation cost as problem size increases (5.3 to 1.01). This translates into close to linear speedup for larger problem sizes, as shown by the speedup values calculated for $\frac{nc \times P}{best}$.

These timings lead us to conclude that for parallel computations, communication optimizations can significantly reduce communication overhead, depending on the amount and nature of computation performed by each processor. The number of processors appears to have little effect, except indirectly by changing the amount of computation per processor. For larger problem sizes, message vectorization seems to yield most of the available improvement.

5.2 Optimizations for Reductions and Scans

Table 3 illustrates the performance of optimizations for parallelizing reductions and scans. In *seq*, the computation is sequentialized by requiring each processor to wait for the partial result from the previous processor before performing the local computation. In *sr, br,* and *cc* the partial results are computed in parallel by each processor, then accumulated using individual send/receives, broadcast/receives, or collective communication, respectively.

Version	Optimizations Performed
nc	no communication
mp	message pipelining (element messages)
mv	message vectorization
mc	mv + message coalescing
ma	mc + message aggregation
vmp	ma + vectorized message pipelining
ir	vmp + iteration reordering
mc', vmp', ir'	versions w/ nonblocking messages
seq	sequential reduction/scan
sr	accumulate using send/receive
br	accumulate using broadcast/receive
cc	accumulate using collective communication
dyn	dynamic data decomposition
fgp	fine-grain pipelining
cgp	coarse-grain pipelining w/ block size B

Table 1: Optimized Versions of Test Kernels

The largest improvements (4–22) were measured for $\frac{seq}{sr}$, making discovering and extracting parallelism the most important optimization for reduction and scan operations. As expected, the benefit of exploiting parallelism increases with both the problem size and number of processors. Timings show that broadcasts can accumulate partial results quicker than sending individual messages, and specialized collective communication is even more efficient.

The values for $\frac{sr}{cc}$ (1–3.3) show that collective communication can provide large improvements over simple messages. Unlike other communication overhead optimizations, the impact of collective communication increases with the number of processors, even when the amount of computation per processor remains constant. From the values for $\frac{cc}{nc}$ (1–3.7) we conclude that communication overhead can become a major component of execution time for reductions and scans, especially when employing large numbers of processors. The values calculated for $\frac{nc \times P}{cc}$ show that close to linear speedups were measured for reductions. Scans achieved only about half of linear speedup, probably because computation is doubled in the parallel scan.

5.3 Optimizations for Pipelined Computations

Table 4 shows the timings for pipelined computations. In all three kernels, the original loop structure and data distribution is such that message pipelining (*mp*) yields parallelism only for the last outer loop iteration, and message vectorization (*mv*) sequentializes the computation. Loop interchange is needed in order to enable fine-grain pipelining (*fgp*) in these kernels. We present measurements for these worst-case examples of *mv* and *mp* to illustrate potential pitfalls if the compiler cannot reorder computation through loop interchange. Results for nonblocking messages are not displayed. They degraded the performance of both fine and coarse-grain pipelining since message sizes are too small to compensate for increased startup costs.

The values for $\frac{mv}{fgp}$ (2.7–14) show that it is essential to exploit parallelism for pipelined computations, particularly as the number of processors increases. The best overall timings, *best*, were achieved using coarse-grain pipelining. A block size of eight resulted in the best times for Livermore 23; a block size of twelve proved best for SOR and ADI integration. Results for $\frac{fgp}{best}$ (1.3–3.7) show that coarse-grain pipelining can significantly improve performance when compared to fine-grain pipelining. Values for $\frac{nc \times P}{best}$ indicate coarse-grain pipelining can achieve respectable speedup for

Kernel	*P*	*N*	*nc*	*mp*	*mv*	*mc*	*ma*	*vmp*	*ir*	*mc'*	*vmp'*	*ir'*	$\frac{mp}{mv}$	$\frac{mv}{best}$	$\frac{best}{nc}$	$\frac{nc \times P}{best}$
Livermore 18 *Explicit Hydrodynamics*	16	32x32	1.2	62.1	9.7	7.4	5.7	5.7	5.7	5.3	4.1	4.3	6.40	2.37	3.58	4.7
		64x64	3.9	62.7	16.2	12.7	11.4	11.0	11.6	11.1	8.7	9.8	3.87	1.86	2.23	7.2
		128x128	14.4	137	35.5	29.7	27.5	25.6	28.0	26.5	22.5	24.9	3.86	1.58	1.57	10.2
		256x256	54.3	317	90.2	80.9	77.6	77.8	79.3	76.6	69.7	73.6	3.51	1.29	1.30	12.5
		512x512	211	732	277	260	257	255	262	253	239	248	2.64	1.15	1.13	14.1
	32	32x32	0.7	62.5	9.5	7.1	5.5	5.4	5.5	5.0	3.7	4.3	6.58	2.57	5.29	6.1
		64x64	2.5	60.5	14.7	11.1	9.9	9.9	9.9	9.9	7.3	7.5	4.12	2.01	2.92	11.0
		128x128	8.0	128	29.3	23.6	21.3	20.7	21.7	20.6	16.5	18.3	4.37	1.76	1.50	15.5
		256x256	28.8	287	64.8	55.4	53.8	51.6	53.9	52.8	43.7	48.1	4.43	1.48	1.52	21.1
		512x512	109	630	177	159	156	155	160	152	137	146	3.56	1.29	1.25	25.5
Jacobi	16	128x128	1.3	29.3	4.1	-	-	3.9	3.7	3.8	3.6	2.9	7.15	1.41	2.23	7.2
		256x256	5.3	64.0	9.8	-	-	8.9	9.2	9.0	8.7	6.2	6.53	1.58	1.16	13.7
		512x512	20.7	145	28.3	-	-	29.0	27.6	27.4	27.3	23.2	5.18	1.22	1.10	14.3
		1Kx1K	96.9	349	110	-	-	107	112	109	109	99.1	3.17	1.11	1.02	15.6
		2Kx2K	385	889	412	-	-	417	411	410	410	391	2.16	1.05	1.02	15.8
	32	128x128	0.7	29.9	3.5	-	-	3.1	3.1	3.4	3.2	3.0	8.54	1.17	4.29	7.5
		256x256	2.7	64.2	7.2	-	-	7.2	6.6	7.2	7.1	4.9	8.92	1.47	1.81	17.6
		512x512	10.4	136	18.2	-	-	18.7	18.2	18.7	18.5	12.3	7.56	1.48	1.20	27.1
		1Kx1K	48.5	296	64.1	-	-	64.2	64.0	61.0	60.9	51.8	4.63	1.24	1.08	30.0
		2Kx2K	193	693	220	-	-	224	224	217	217	198	3.15	1.11	1.03	31.2
Red-Black SOR	16	128x128	1.7	29.8	4.8	-	-	4.9	5.1	4.8	3.9	3.6	6.21	1.33	2.12	7.6
		256x256	6.7	66.7	11.8	-	-	10.5	10.6	11.4	9.7	9.3	5.65	1.27	1.39	11.5
		512x512	26.3	158	33.9	-	-	31.5	31.8	33.9	30.8	29.7	4.65	1.14	1.15	14.2
		1Kx1K	109	397	122	-	-	118	118	122	116	114	3.25	1.07	1.05	15.3
		2Kx2K	437	971	462	-	-	453	454	457	448	442	2.10	1.04	1.01	15.8
	32	128x128	0.8	30.5	4.1	-	-	4.0	4.2	4.1	3.3	3.1	7.44	1.32	3.88	8.3
		256x256	3.3	63.6	8.6	-	-	7.3	7.4	8.0	6.3	5.8	7.40	1.48	1.76	18.2
		512x512	13.2	148	21.0	-	-	18.5	18.9	20.8	17.7	16.6	7.05	1.26	1.30	25.4
		1Kx1K	54.3	342	68.3	-	-	63.9	64.7	67.3	62.2	59.5	5.03	1.15	1.11	29.2
		2Kx2K	217	766	245	-	-	236	238	241	231	226	3.13	1.09	1.04	30.7

Table 2: Performance of Optimizations to Reduce and Hide Communication Overhead (in milliseconds)

Kernel	*P*	*N*	*nc*	*seq*	*sr*	*br*	*cc*	$\frac{seq}{sr}$	$\frac{sr}{cc}$	$\frac{cc}{nc}$	$\frac{nc \times P}{cc}$
Livermore 3 *Inner Product*	8	64K	2.6	22	3.5	3.3	3.3	6.29	1.08	1.27	6.3
		256K	10.0	86	11.0	11.0	11.0	7.82	1.03	1.10	7.3
		1024K	43.5	348	44.5	44.1	44.2	7.82	1.01	1.02	7.9
	16	64K	1.3	23	3.4	2.5	2.1	6.76	1.59	1.62	9.9
		256K	5.3	86	7.4	6.6	6.1	11.6	1.21	1.15	13.9
		1024K	21.7	347	23.8	22.9	22.6	14.6	1.05	1.04	15.3
	32	64K	0.7	24	5.5	3.1	1.6	4.36	3.33	2.29	14.0
		256K	2.6	86	7.5	5.0	3.6	11.5	2.07	1.38	23.1
		1024K	10.7	345	15.6	13.1	11.7	22.1	1.33	1.09	29.3
Livermore 11 *First Sum*	8	64K	2.3	19	4.6	4.3	4.2	4.13	1.10	1.82	4.4
		256K	8.9	73	15.5	15.4	15.1	4.71	1.03	1.70	4.7
		1024K	35.6	286	59.0	58.9	58.8	4.85	1.00	1.65	4.8
	16	64K	1.2	20	4.2	3.1	2.7	4.76	1.54	2.25	7.1
		256K	4.5	74	9.5	8.6	8.1	7.79	1.31	1.80	8.9
		1024K	17.9	287	31.2	30.5	30.0	9.20	1.08	1.68	9.5
	32	64K	0.6	21	5.8	3.4	2.2	3.62	2.66	3.67	8.7
		256K	2.3	75	8.6	6.1	4.8	8.72	1.77	2.09	15.3
		1024K	8.9	290	19.3	17.2	15.9	15.0	1.22	1.79	19.0

Table 3: Performance of Optimizations to Parallelize Reductions and Scans (in milliseconds)

Kernel	*P*	*N*	*nc*	*mp*	*mv*	*dyn*	*fgp*	*cgp* *B=4*	*cgp* *B=8*	*cgp* *B=12*	*cgp* *B=16*	$\frac{mv}{fgp}$	$\frac{fgp}{best}$	$\frac{best}{nc}$	$\frac{nc \times P}{best}$
Livermore 23 *Implicit Hydrodynamics*	16	256x256	24	613	395	-	86	51	49	52	63	4.59	1.76	2.04	7.8
		512x512	96	1990	1550	-	222	156	148	152	171	6.98	1.50	1.54	10.4
		1Kx1K	383	7190	6160	-	677	537	507	508	539	9.10	1.34	1.32	12.1
	32	256x256	12	847	412	-	78	40	39	43	55	5.28	2.00	3.25	9.8
		512x512	48	2480	1580	-	171	102	99	105	129	9.24	1.72	2.06	15.5
		1Kx1K	191	8280	6210	-	441	311	298	308	348	14.1	1.48	1.56	20.5
SOR	16	512x512	23	834	400	-	146	60	49	48	59	2.74	3.04	2.09	7.7
		1Kx1K	107	2730	1750	-	330	177	150	143	167	5.30	2.31	1.34	12.0
		2Kx2K	429	9280	6950	-	947	598	519	493	534	7.34	1.92	1.14	13.9
	32	512x512	12	135	431	-	145	50	40	39	52	2.97	3.72	3.25	9.8
		1Kx1K	53	382	1800	-	321	121	99	96	123	5.61	3.34	1.81	17.7
		2Kx2K	213	1180	7010	-	806	360	308	288	340	8.70	2.80	1.35	23.7
ADI Integration	16	512x512	51	933	496	288	175	95	79	77	90	2.83	2.27	1.51	10.6
		1Kx1K	204	2920	1960	1110	475	315	270	261	285	4.13	1.82	1.28	12.5
		2Kx2K	817	10100	7710	4820	1520	1140	991	959	1000	5.07	1.59	1.17	13.6
	32	512x512	26	1440	515	166	162	70	56	55	69	3.18	2.95	2.12	15.1
		1Kx1K	102	4020	1970	614	383	196	163	158	183	5.14	2.42	1.55	20.7
		2Kx2K	408	12500	7630	2530	1100	644	547	531	573	6.94	2.07	1.30	24.6

Table 4: Performance of Optimizations to Exploit Pipeline Parallelism (in milliseconds)

Opt	Resulting Communication Overhead (For n Elements)
none	$n(T_{start} + T_{copy}(1) + T_{transit}(1))$
mp	$n(T_{start} + T_{copy}(1) + \text{pos}(T_{transit}(1) - T_{comp}))$
mv	$T_{start} + T_{copy}(n) + T_{transit}(n)\ [+ T_{buf}(n)]$
ma	$T_{start} + T_{copy}(mn) + T_{transit}(mn) + mT_{buf}(n)$
vmp	$T_{start} + T_{copy}(n) + \text{pos}(T_{transit}(n) - T_{comp})$
ir	$T_{start} + T_{copy}(n) + \text{pos}(T_{transit}(n) - T'_{comp}) + \Delta T_{comp}$
mv'	$T'_{start} + T_{copy}(n)/2 + T_{transit}(n)$
vmp'	$T'_{start} + \text{pos}(T_{copy}(n)/2 + T_{transit}(n) - T_{comp})$
ir'	$T'_{start} + \text{pos}(T_{copy}(n)/2 + T_{transit}(n) - T'_{comp}) + \Delta T_{comp}$

Table 5: Effect of Compiler Optimizations

pipelined computations. Dynamic data decomposition to redistribute arrays in ADI proved to be undesirable and required significantly more time than pipelining, especially for large problem sizes.

6 Analysis of Compiler Optimizations

Our empirical results show that compiler optimizations can be used to improve program performance. This section presents analysis and decision algorithms to determine when these optimizations can be profitably applied.

6.1 Communication Optimizations

We begin by analyzing optimizations to manage communication overhead. Table 5 provides the cost of sending one message with n elements for each optimization (the cost for message aggregation (ma) represents m messages). These formulas for communication overhead are presented using T_{start}, T_{copy}, $T_{transit}$ and some new terms. $T_{buf}(n)$ describes the cost of buffering n noncontiguous data elements for message vectorization (mv). It is placed in square brackets [ ] because it is only incurred if data is noncontiguous. T_{buf} may also be be ignored if the underlying architecture can efficiently communicate noncontiguous data. We assume it is not needed for optimizations that include message vectorization.

$Pos()$ is a function that returns the value of its argument if it is positive, zero otherwise. T_{comp} represents the amount of computation between a pair of calls to *send* and *recv* that may be used to hide communication cost. T'_{comp} includes the computation available after applying iteration reordering. ΔT_{comp} describes the increase in computation time caused by iteration reordering. T'_{start} is the startup cost of using nonblocking messages.

The Fortran D compiler always applies message coalescing, vector message pipelining, and collective communication where applicable, since these optimizations improve performance in all cases. In the following sections, we describe profitability criteria for other communication optimizations. These criteria are derived directly from Table 5, but are simplified where possible. These formulas can also be used to calculate the expected savings of each optimization. For simplicity we regard copy time as linear, treating $T_{copy}(n)$ and $nT_{copy}(1)$ as equal quantities.

Message vectorization To send n elements, message vectorization is profitable over message pipelining (assuming mp can hide $T_{transit}$) when:

$$mp > mv$$
$$\Downarrow$$
$$(n-1)T_{start} > T_{transit}(n)\ [+ T_{buf}(n)]$$

The compiler thus needs to compare the reduction in startup time against the transit time and cost of buffering noncontiguous data. When startup costs are high, as on the iPSC/860, message vectorization will significantly outperform message pipelining for large values of n.

Message aggregation To send m messages of size n, message aggregation is profitable over message vectorization when:

$$mv > ma$$
$$\Downarrow$$
$$(m-1)T_{start} + mT_{transit}(n) > T_{transit}(mn)\ [+ mT_{buf}(n)]$$

If the transit time for m messages of size n is similar to that for one message of size mn, the primary overhead of message aggregation is the cost of copying all messages to a single buffer. If the individual messages are not contiguous, then message aggregation is always profitable since message vectorization performs buffering in any case. Otherwise it is profitable only if the reduction in startup time is greater than the extra buffering cost.

Nonblocking messages Using nonblocking messages halves T_{copy} by eliminating copying on the receiving processor, and the remaining T_{copy} may be overlapped with computation. However, the resulting program incurs a higher startup cost T'_{start}. It is profitable to use nonblocking messages with vector message pipelining when:

$$vmp > vmp'$$
$$\Downarrow$$
$$T_{comp} \geq T_{copy}(n) > (T'_{start} - T_{start})$$

The compiler will use nonblocking messages if sufficient local computation exists to hide copy cost, and the copy cost is greater than the increased startup cost. Since the savings in copy time increases with n, nonblocking messages become more useful as message size increases.

Iteration reordering Iteration reordering makes additional local computation available, but may also affect code size, data reuse, and conventional scalar optimizations, increasing the total computation time. For instance, empirical results show that iteration reordering does not affect computation costs for Jacobi and Red-Black SOR, but slightly degrades performance for Livermore 18, a kernel that contains significant amounts of computation and data reuse. With blocking messages, iteration reordering can profitably enhance vector message pipelining when the following conditions hold:

$$vmp > ir$$
$$\Downarrow$$
$$T'_{comp} \geq T_{transit}(n) > T_{comp}$$
$$T_{transit}(n) - T_{comp} > \Delta T_{comp}$$

Iteration reordering should thus be applied if the message transit time is not completely hidden by vector message pipelining, and iteration reordering can extract sufficient local computation to hide the remaining transit time. In addition, the savings in transit time must be greater than the increased computation time. Iteration reordering using nonblocking messages is profitable when:

$$vmp' > ir'$$
$$\Downarrow$$
$$T'_{comp} \geq T_{copy}(n) + T_{transit}(n) > T_{comp}$$
$$T_{copy}(n) + T_{transit}(n) - T_{comp} > \Delta T_{comp} + T'_{start} - T_{start}$$

The criteria are similar to that of blocking messages, except that both copy and transit times are considered.

The usefulness of iteration reordering hinges on the value of ΔT_{comp}, which is quite difficult to predict. Our strategy is to simply estimate ΔT_{comp} as some small fixed percentage of the total computation time. It can then be compared against the message copy and transit times to determine whether iteration reordering is worthwhile.

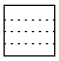
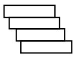
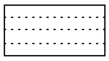
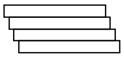
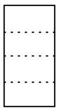
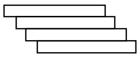
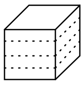
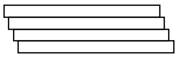
Data Arrays & Data Partition	Computation & Elapsed Time	Sequential Time	Parallel Time (fgp)	Parallel Time Blocked (cgp)	Block Size Preferred
		n^2	$\frac{n^2}{p} + nC + (p-1)(\frac{n}{p} + C)$	$\frac{n^2}{p} + \frac{nC}{B} + (p-1)(\frac{nB}{p} + C)$	$\sqrt{C}$
		$2n^2$	$\frac{2n^2}{p} + 2nC + (p-1)(\frac{n}{p} + C)$	$\frac{2n^2}{p} + \frac{2nC}{B} + (p-1)(\frac{nB}{p} + C)$	$\sqrt{2C}$
		$2n^2$	$\frac{2n^2}{p} + nC + (p-1)(\frac{2n}{p} + C)$	$\frac{2n^2}{p} + \frac{nC}{B} + (p-1)(\frac{2nB}{p} + C)$	$\sqrt{\frac{C}{2}}$
		n^3	$\frac{n^3}{p} + n^2C + (p-1)(\frac{n}{p} + C)$	$\frac{n^3}{p} + \frac{n^2C}{B} + (p-1)(\frac{nB}{p} + C)$	$\sqrt{nC}$

Figure 13: Effectiveness of Pipelining

6.2 Parallelism Optimizations

In this section we analyze optimizations to exploit parallelism. Reductions and scans should always be identified and parallelized, using collective communication to accumulate results. Empirical results prove that pipelining is vital for pipelined computations. We show analytically that pipeline parallelism is both effective and scalable.

Fine-grain pipelining Consider the simple examples presented in Figure 13. We define n as the number of elements along one dimension, p as the number of processors, and C as the communication overhead for each message. We normalize all costs by the cost required to compute one element, so the sequential computation time is equal to the number of data elements.

For simplicity we restrict our analysis to cases where we can interchange cross-processor loops to the *innermost* position, allowing program execution to first proceed along the distributed dimensions. This enables both fine-grain and coarse-grain pipelining. Fortunately, most if not all pipelined computations meet this requirement. For instance, loop interchange of cross-processor loops is legal for both SOR and ADI integration.

Using these assumptions, we can now calculate the time required to compute an $n \times n$ data array distributed blockwise in one dimension. Each processor begins execution exactly $\frac{n}{p} + C$ units later than its predecessor, where $\frac{n}{p}$ is the time for its predecessor to compute one column and C is the communication overhead. The time it takes each processor to finish its computation is $\frac{n^2}{p}$, the total computation time, plus nC, the time spent to send and receive n messages. The total parallel execution time is $(p-1)(\frac{n}{p} + C)$, the delay before the last processor begins, plus $\frac{n^2}{p} + nC$, the time required by the processor to finish computing.

Similar calculations for the $n \times 2n$, $2n \times n$, and $n \times n \times n$ example arrays result in the formulas shown in Figure 13. Examining the expressions, we see that the dominating term in the parallel execution time is simply $(\textit{sequential time})/p$. Pipeline parallelism under these conditions thus approaches perfect speedup for large problem sizes.

Coarse-grain pipelining The same model may also be used to calculate an efficient blocking factor for coarse-grain pipelining. Assume we strip-mine and interchange the outer loop in the pipelined computation of an $n \times n$ array by a constant block factor B, as in Figure 12. The delay between processors increases to $\frac{nB}{p} + C$, since B columns each costing $\frac{n}{p}$ are computed before sending a message. However, the total communication overhead for one processor drops from nC to $\frac{nC}{B}$. The total parallel execution time is thus $\frac{n^2}{p} + \frac{nC}{B} + (p-1)(\frac{nB}{p} + C)$. The times for other examples are shown in Figure 13.

As we can see, the asymptotic speedup is unchanged by B, but the total communication overhead can be significantly decreased at the expense of some parallelism. To determine the minimal cost while holding n and p constant, we differentiate the expression for parallel execution time with respect to B and set the result to zero. This yields the following equation and solution for B:

$$-\frac{nC}{B^2} + \frac{n(p-1)}{p} = 0$$

$$B = \sqrt{\frac{pC}{p-1}} \approx \sqrt{C} = \sqrt{\frac{\text{block communication cost}}{\text{element computation cost}}}$$

Since C has been normalized by the computation required to calculate one array element, it is actually the ratio of communication to computation cost. As expected, the results show that larger block sizes are preferred when the ratio of communication to computation cost is high; smaller blocks are desirable when communication cost is relatively low. More importantly, these formulas allow the compiler to calculate efficient block sizes and estimated execution times for pipelined computations.

Our analysis for pipelined computations is somewhat imprecise since it assumes that communication cost is fixed as the message size increases. Fortunately this is relatively true for the small block sizes that are selected. More accurate analytical models can be developed, but may be hindered by unpredictable system discontinuities. For instance, communication cost increases abruptly past 100 bytes on the iPSC/860 [6]. The Fortran D compiler will employ a flexible and precise approach using *training sets* to estimate communication and computation costs [4, 19, 23]. Accurate static estimates of communication and computation are also needed by the compiler to calculate block sizes for coarse-grain pipelining.

Dynamic data decomposition The previous sections show how parallel computation time can be estimated for pipelined computations. The compiler needs to compare it with the estimated cost for dynamic data decomposition (based on training sets) to determine whether it is more profitable than applying pipelining. Dynamic data decomposition is likely to be profitable only for small problems, because communication to redistribute data becomes less efficient as problem size increases. In comparison, the efficiency of pipelining improves with larger problem sizes.

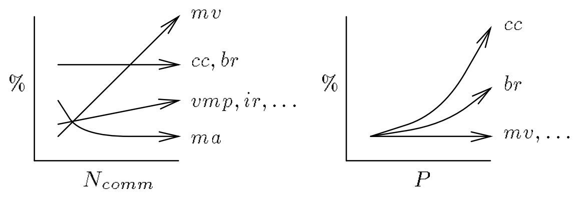


Figure 14: Effect on Communication Overhead

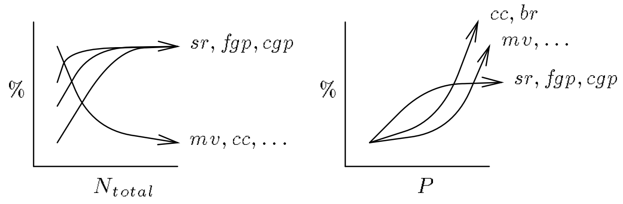


Figure 15: Effect on Program Execution Time

6.3 Scalability

The *scalability* of an optimization describes whether its effectiveness increases, decreases, or remains constant in proportion to some characteristic. In this section we use scalability to summarize our insights concerning the usefulness of communication and parallelism optimizations. Our conclusions are derived from the empirical and analytical results presented in the previous sections. In the following discussion we define N_{comm} to be the number of elements communicated by each processor and N_{total} to be the total number of elements. For convenience, we also use N_{local} to describe the number of elements on each processor. It is simply N_{total}/P, where P is the number of processors.

Communication overhead Figure 14 shows the scalability of optimizations in eliminating communication overhead. The effectiveness of message vectorization (mv) is displayed as improvement over message pipelining. Collective communication (cc) and broadcast/receive (br) are shown as gains over send/receive (sr). The effectiveness of other optimizations are displayed as improvement compared with message vectorization. All improvements are shown as percentages.

We first consider how communication optimizations scale with respect to N_{comm}, the amount of data communicated by each processor. When we increase N_{comm} for a fixed number of processors, message vectorization (mv) improves most rapidly because it eliminates entire messages. Other optimizations ($vmp, ir, \ldots$) improve less quickly since they only affect message transit and copy times. The effectiveness of collective communication (cc) and broadcast/receive (br) remains unchanged at a level determined by the number of processors. The percentage improvement for message aggregation (ma) decreases because its usefulness is set by the number of arrays communicated to the same processor.

In comparison, when N_{comm} is fixed, most communication optimizations (mv, vmp, $\ldots$) are not enhanced by increasing the number of processors. Only collective communication (cc) and broadcast (br) improve in their ability to eliminate communication cost as P grows.

Program execution Figure 15 displays the scalability of optimizations in reducing total execution time. We assume that computation cost is proportional to N_{total}. Optimizations to exploit parallelism (sr, fgp, cgp) are expressed as improvements relative to the sequential execution time. For a fixed number of processors P, they increase in effectiveness as N_{total} grows, reaching a plateau at the number of processors. In comparison, communication optimizations (mv, cc, $\ldots$) shrink in relative usefulness because N_{comm} grows slowly compared to N_{total} for stencil computations.

Problem Dimension	Dimensions Distributed	N_{comm}	N_{comm}/N_{local} for $p=8$ & $n=10^3$	$n=10^4$	$n=10^5$
3D	1D	$2\sqrt[3]{n^2}$	1.0	.74	.35
3D	2D	$4\sqrt[3]{n^2}/\sqrt{p}$	1.0	.52	.24
2D	1D	$2\sqrt{n}$	.50	.16	.05
2D	2D	$4\sqrt{\frac{n}{p}}$	.36	.11	.04
3D	3D	$6\sqrt[3]{\frac{n}{p}}$	.24	.05	.01
1D	1D	2	.016	.0016	.00016

Table 6: Data Communication Requirements

The situation is more complex when a problem with fixed size is parallelized using an increasing number of processors. Initially the amount of communication is small relative to the local problem size ($N_{comm} \ll N_{local}$) so parallelism optimizations achieve excellent speedup, increasing linearly with P. At this stage communication optimizations only attain modest improvements, though collective communication and broadcast/receive improve more quickly.

As we show in the next section, eventually the problem is divided among enough processors that N_{comm} becomes a large percentage of N_{local}. When this point is reached, communication overhead begins to have a significant impact on execution time. Growth in the effectiveness of parallelism optimizations slows because of communication costs, while communication optimizations quickly increase in importance. How soon this point is reached depends on the communication overhead relative to computation costs.

Communication vs. computation We have seen that parallelism optimizations are critical for improving overall program execution time, regardless of the problem or machine size. In comparison, the effectiveness of communication optimizations is dependent on N_{comm}, the amount of data that must be communicated. Understanding the relationship between N_{comm}, N_{total}, and N_{local} is thus crucial to determining the impact of communication optimizations.

Simple geometric analysis shows that the growth of N_{comm} relative to N_{total} varies for different data distributions. For instance, when a 2D array with n elements is distributed 1D block-wise across p processors, each processor owns a $\sqrt{n}\times\frac{\sqrt{n}}{p}$ section of the array. Assuming a stencil computation that only accesses boundary elements, a processor needs to send $\sqrt{n}$ array elements to each neighboring processor, communicating $2\sqrt{n}$ elements. Similar analyses for other examples result in the formulas for calculating N_{comm} displayed in Table 6.

Table 6 also presents relative values of N_{comm} for three different problem sizes on a machine with eight processors. Though it varies depending on the problem and machine dimensionality, N_{comm} always grows less rapidly than N_{local}. This implies that communication optimizations become less important as problem size grows. For large problems, message vectorization and collective communication are likely to yield most of the available benefits.

On the other hand, consider the situation when we attempt to speed up a problem of size N_{total} by increasing the number of processors. Similar analysis makes it clear that N_{comm} becomes an increasingly large percentage of N_{local}. Eventually communication overhead becomes the limiting factor, and all of the communication optimizations discussed become important for achieving good speedup.

7 Optimization Algorithm

The overall Fortran D compiler optimization algorithm is shown in Figure 16. It is intended only to provide a rough outline of how optimizations are organized. The compiler will decide at each point which optimizations are actually worth performing.

8 Related Work

The Fortran D compiler is a second-generation compiler that emphasizes compile-time analysis and optimization instead of language extensions and run-time support. By using dependence analysis, the Fortran D compiler can detect and exploit parallelism automatically, without requiring the user to specify single assignment (Crystal [29], Id Nouveau [32]), all parallel loops (Al [36], Arf [37], Kali [26]), parallel functions (C*/Dataparallel C [17, 33], Dino [34]), parallel code blocks (Oxygen [35], Pandore [2]), or parallel array operations (CM Fortran [7], Paragon [10]). The Fortran D compiler is similar to Callahan & Kennedy [9] and Superb [15, 38], but applies analysis and optimization up front before code generation, rather than inserting guards and element-wise messages then optimizing via program transformations and partial evaluation.

Most distributed-memory compilers reduce communication overhead by extracting communication out of user-specified parallel regions (*e.g.,* loops, code blocks, array operations, procedures). Fortran D is similar to Superb and Vienna Fortran [5] in that it vectorizes messages using data dependence information, which can extract communication even out of sequential regions such as those found in ADI or SOR. Crystal and Id Nouveau identify parallelism automatically and vectorize messages using the single assignment semantics of their high-level functional languages. Crystal pioneered the strategy of identifying collective communication opportunities.

Aspar [22] and P^3C [14] extract communication from parallel loops and rely on portable run-time libraries to support collective communication and reductions. CM Fortran extracts communications from array operations and handles reductions expressed as array intrinsics. Dino programs can be significantly improved through iteration reordering and pipelining [31]. Id Nouveau applies message pipelining and recognizes reductions. Kali performs iteration reordering for individual parallel loops and suggests using array transposition for ADI integration. Gupta & Banerjee estimate collective communication and pipelining costs in Parafrase-2 [16].

9 Conclusions

A usable yet efficient machine-independent parallel programming model is needed to make large-scale parallel machines useful for scientific programmers. We believe that Fortran D, one of the first data-placement languages, can provide such a portable data-parallel programming model. The key to achieving good performance is applying advanced compiler analysis and optimization to automatically extract parallelism and manage communication.

In this paper we described and categorized a large number of compiler optimizations for MIMD distributed-memory machines. For the kernels tested, we found that extracting parallelism from reductions, scans, and pipelined computations is vital. Message vectorization, collective communication, and efficient pipelining can yield significant improvements. The remaining optimizations acquire greater importance as the amount of computation per processor decreases and communication overhead becomes a larger percentage of total program execution time. We derive profitability formulas each optimization that depend on accurate knowledge of the communication and computation costs of the underlying machine.

The existing Fortran D compiler prototype parallelizes reductions, pipelined computations, and performs message vectorization, coalescing, and aggregation for block-distributed arrays. We are implementing the remaining optimizations and intend to evaluate their effectiveness for larger and more varied programs on different MIMD architectures, including networks of workstations. Ongoing work to provide environmental support for automatic data decomposition and static performance estimation will also enhance the usefulness of the Fortran D compiler [3, 4, 19, 23].

```
partition data across processors
partition computation using "owner computes" rule
detect and parallelize reductions & scans
compute cross-processor loops
for each loop nest L do
  if L is fully parallel (i.e., no cross-processor loops) then
    vectorize, coalesce, and aggregate messages
    select and insert collective communications
    if sufficient T_comp exists to hide T_copy, T_transit then
      apply vector message pipelining
      insert nonblocking messages
    else if T'_comp can be profitably created & used then
      reorder iterations
      apply vector message pipelining
      insert nonblocking messages
    else
      insert blocking messages
    endif
  else      {* must be pipelined computation *}
    select efficient granularity for pipelining
    apply strip-mining & loop iterchange
    vectorize, coalesce, and aggregate messages
    insert blocking messages
  endif
  if insufficient storage is available then
    apply storage optimizations
  endif
enddo
```

Figure 16: Fortran D Optimization Algorithm

10 Acknowledgements

The authors wish to thank Mary Hall, Uli Kremer, and Kathryn McKinley for their assistance on this paper, as well as Joel Saltz, Robert Schnabel and Robert Weaver for several enlightening discussions. We are also grateful to the ParaScope and Fortran D research groups for their assistance in implementing the Fortran D compiler. This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center. Use of the Intel iPSC/860 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement Nos. CCR-8809615 and CDA-8619893 with support from the Keck Foundation.

References

[1] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] F. André, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[3] 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, April 1990.

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[5] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[6] S. Bokhari. Complete exchange on the iPSC-860. ICASE Report 91-4, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[7] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[8] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[9] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[10] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[11] S. Chatterjee, G. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[12] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[13] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[14] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[15] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice & Experience*, 2(3):171–193, September 1990.

[16] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[17] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[18] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[19] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[20] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[21] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[22] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[23] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report TR91-155, Dept. of Computer Science, Rice University, April 1991.

[24] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[25] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[26] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[27] P. Kogge and H. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[28] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[29] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

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

[31] D. Olander and R. Schnabel. Preliminary experience in developing a parallel thin-layer Navier Stokes code and implications for parallel language design. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[32] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[33] J. Rose and G. Steele, Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[34] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[35] R. Ruhl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[36] P.-S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[37] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[38] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.