Evaluating Compiler Optimizations For Fortran D *

Seema Hiranandani *seema@cs.rice.edu*

Ken Kennedy *ken@cs.rice.edu*

Department of Computer Science
Rice University
Houston, TX 77251-1892

Chau-Wen Tseng
tseng@cs.stanford.edu

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

*This research was supported by the Center for Research on Parallel Computation, a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. 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.

Evaluating Compiler Optimizations For Fortran D

Corresponding Author:

Chau-Wen Tseng
Center For Integrated Systems
Stanford University
Stanford, CA 94305
Tel: (415) 723-6838
email: tseng@cs.stanford.edu

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. Communication optimizations reduce communication overhead by decreasing the number of messages and hide communication overhead by overlapping the cost of remaining messages with local computation. Parallelism optimizations exploit parallel and pipelined computations, and may need to restructure the computation to increase parallelism. Profitability formulas are derived for each optimization. Empirical 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. Scalability of communication and parallelism optimizations are analyzed. The effectiveness of communication optimizations is dictated by the ratio of communication to computation in the program. An optimization strategy is developed based on these analyses.

List of Symbols

Symbols used in the text:

prime	$'$
Delta	Δ
Downarrow	$\Downarrow$

Symbols used in tables and figures:

diamond	$\diamond$
circle	$\circ$
star	$\star$
asterisk	$\ast$
bullet	$\bullet$
box	$\Box$
Delta	Δ

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 [24].

Our goal with Fortran D is to provide a simple yet efficient machine-independent parallel programming model by shifting the burden of machine-dependent optimization to the compiler. To evaluate the Fortran D programming model, we are implementing a prototype compiler for MIMD distributed-memory machines. The design and implementation of the prototype compiler are described elsewhere. They include algorithms to partition data and computation [26]; internal representations, program analysis, pipelining and code generation algorithms [27]; interprocedural compilation issues [21]; and integrating partitioning and scalarization optimizations for Fortran 90D [15].

In order to be successful, the Fortran D compiler must be able to generate highly efficient programs. This paper presents our efforts to develop compiler optimizations and evaluate their impact on performance. Contributions include:

- introducing new and existing compiler optimizations in a unified framework;
- evaluating the effect of optimizations for stencil kernels on the Intel iPSC/860;
- ranking the importance of optimizations in the following order:

1. exploit parallelism,
2. reduce communication overhead with message vectorization, coarse-grain pipelining, and collective communications,
3. remaining communication optimizations;

- designing communication and computation models to guide compiler optimizations;
- evaluating the scalability of compiler optimizations with respect to problem size, number of processors, and type of computation.

We note that our goal is not to invent new compiler optimizations, since most have been discovered by other researchers. Instead, we show how these optimizations may be combined in the framework of the Fortran D compilation system, pointing out some interesting design and implementation details. By first describing and classifying each optimization, we prepare the reader for the main result of this paper: an empirical and analytical evaluation of compiler optimizations for MIMD distributed-memory machines. 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 and Compiler

The Fortran D language 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 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. Irregular and dynamic data decomposition are also supported. The complete language is described in detail elsewhere [16, 46].

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. The Fortran D compiler partitions computation across processors using the *owner computes* rule—where each processor only computes values of data it owns [11, 42, 50]. Details of the Fortran D compiler are discussed elsewhere [46]:

1) Reduce Communication Overhead

- Message Vectorization
- Message Coalescing
- Message Aggregation
- Collective Communication
- Run-time Processing
- Relax Owner Computes Rule
- Replicate Computation

2) Hide Communication Overhead

- Message Pipelining
- Vector Message Pipelining
- Iteration Reordering
- Unbuffered Messages

3) Exploit Parallelism

- Partition Computation
- Private Variables
- Reductions and Scans
- Dynamic Data Decomposition
- Pipeline Computations

4) Reduce Storage

- Partition Data
- Message Blocking

Figure 1 Classification of Compiler Optimizations

3 Compiler Optimizations

As the Fortran D compiler is a second generation research project, many of its optimizations have been discussed in the literature. However, previous researchers tend to develop each optimization in isolation, without evaluating their effectiveness or considering their interaction with other elements of the compiler. We find that integrating these optimizations is difficult but feasible.

Figure 1 presents our classification of compiler optimizations. Communication optimizations can reduce communication overhead by eliminating messages or hide communication overhead by overlapping communication and computation. Parallelism optimizations exploit the parallelism in fully parallel and pipelined computations. Optimizations improve partitioning by reducing the number of run-time tests required to partition computation. Optimization may also reduce the amount of storage required for nonlocal data.

Figures 2, 3, and 4 provide brief examples of several compiler optimizations. In these examples, we assume arrays are distributed block-wise onto a four processor machine, and that the two dimensional arrays in Figure 4 are distributed by columns (*i.e.,* block-wise in the second dimension). Detailed examples of these compiler optimizations are discussed elsewhere [28, 46].

We note that several compiler optimizations utilize program transformations such as loop interchange, fusion, distribution, alignment, and strip-mining. Their legality is determined in exactly the same manner as for shared-memory parallelizing compilers [1, 32, 35], since transformations must preserve the meaning of the original program. However, their profitability criteria are now totally different, and are described in greater detail in the remainder of this section.

```
{* No Optimization *}
do i =
  A(i) = B(i+2)
        ⇓
do i =
  send B
  recv B
  A(i) = B(i+2)
```

```
{* Message Vectorization *}
do i =
  A(i) = B(i+2)
        ⇓
send B
recv B
do i =
  A(i) = B(i+2)
```

```
{* Message Coalescing *}
do i =
  A(i) = B(i+1)+B(i+2)
        ⇓
send B
recv B
do i =
  A(i) = B(i+1)+B(i+2)
```

```
{* Message Aggregation *}
do i =
  A(i) = B(i+1)+C(i+1)
        ⇓
send B & C
recv B & C
do i =
  A(i) = B(i+1)+C(i+1)
```

```
{* Collective Comm. *}
do i =
  A(i) = B(100)
        ⇓
broadcast B
recv B
do i =
  A(i) = B(100)
```

```
{* Run-time Processing *}
do time =
  do i =
    A(i) = B(C(i)))
        ⇓
inspect accesses
do time =
  scatter-gather B
  do i =
    A(i) = B(C(i))
```

```
{* Relax Owner Computes *}
do i =
  A(i) = B(i+1)+C(i+1)
        ⇓
do i =
  A(i) = B(i+1)+C(i+1)
send A
recv A
```

```
{* Replicate Computation *}
do i = 1,100
  B(i) = F(i)
  A(i) = B(i-1)+B(i)
        ⇓
B(0) = F(0)
do i =
  B(i) = F(i)
  A(i) = B(i-1)+B(i)
```

Figure 2 Examples of Optimizations To Reduce Communication Overhead

```
{* Message Pipelining *}
do i =
  B(i) =
do i =
  A(i) = B(i+2)
        ⇓
do i =
  B(i) =
  send B
do i =
  recv B
  A(i) = B(i+2)
```

```
{* Vector Mesg Pipelining *}
do i =
  B(i) =
local computation
do i =
  A(i) = B(i+2)
        ⇓
do i =
  B(i) =
send B
local computation
recv B
do i =
  A(i) = B(i+2)
```

```
{* Iteration Reordering *}
do i =
  B(i) =
do i =
  A(i) = B(i+2)
        ⇓
do i = nonlocal iterations
  B(i) =
send B
do i = local iterations
  B(i) =
do i = local iterations
  A(i) = B(i+2)
recv B
do i = nonlocal iterations
  A(i) = B(i+2)
```

```
{* Unbuffered Messages *}
local computation
do i =
  A(i) = B(i+2)
        ⇓
post recv B
start send B
local computation
wait recv B
do i =
  A(i) = B(i+2)
wait send B
```

Figure 3 Examples of Optimizations To Hide Communication Overhead

```
{* Reductions *}
do i = 1,100
  sum = sum + A(i)
        ⇓
do i = 1,25
  sum = sum + A(i)
global-sum sum
```

```
{* Scans *}
A(1) = B(1)
do i = 2,100
  A(i) = A(i-1) + B(i)
        ⇓
do i =
  S(my$p) = S(my$p)+B(i)
global-concat S
A(1) = B(1)
do i = 0, my$p-1
  A(1) = A(1) + S(i)
do i = 2,25
  A(i) = A(i-1) + B(i)
```

```
{* Dynamic Data
   Decomposition *}
do i =
  do j =
    A(j,i) = A(j,i-1)
do i =
  do j =
    A(j,i) = A(j-1,i)
        ⇓
map-to-rows A
do i =
  do j =
    A(j,i) = A(j,i-1)
map-to-columns A
do i =
  do j =
    A(j,i) = A(j-1,i)
```

```
{* Fine-grain
   Pipelining *}
do i = 1,100
  do j = 1,100
    A(j,i) = A(j,i-1)
        ⇓
do j = 1,100
  recv A
  do i = 1,25
    A(j,i) = A(j,i-1)
  send A
```

```
{* Coarse-grain
   Pipelining *}
do i = 1,100
  do j = 1,100
    A(j,i) = A(j,i-1)
        ⇓
do jj = 1,100,S
  recv A
  do i = 1,25
    do j = jj,jj+S-1
      A(j,i) = A(j,i-1)
  send A
```

Figure 4 Examples of Optimizations To Exploit Parallelism

3.1 Reducing Communication Overhead

We first present compiler optimizations that attempt to reduce 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:

- 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 [7]. The following optimizations seek to reduce T_{start} by *eliminating* messages, reducing the total number of messages sent. In the next section we describe optimizations that try to *hide* T_{copy} and $T_{transit}$ by overlapping communication with computation.

3.1.1 Message Vectorization

Message vectorization is a loop-based optimization fundamental to the Fortran D compilation process [5, 19]. It uses the results of data dependence analysis [1, 35] to extract communication from within loops, combining element messages per loop iteration with one vectorized message preceding the loop. Message vectorization first calculates *commlevel*, the level of the deepest loop-carried true dependence or loop enclosing a loop-independent true dependence. This level determines the outermost loop where element messages resulting from the same array reference may be legally combined. *Regular section descriptors* (RSDs), compact representations of rectangular array sections [23], are built for vectorized nonlocal accesses 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.

3.1.2 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 the RSDs possess the same communication type (*e.g.,* shift, broadcast). If two overlapping RSDs cannot be coalesced without loss of precision, they can be split into smaller sections in a manner that allows the overlapping regions to be merged precisely.

3.1.3 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 that have the same communication type. 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.

Though not implemented, message coalescing and aggregation can also be applied across loop nests. RSDs representing nonlocal data to be communicated for a loop nest may be combined with RSDs at previous loop nests at the same level, if the data represented in the RSD has not be redefined by an intervening write. During dependence analysis, the Fortran D compiler summarizes array definitions using RSDs. During optimization, the compiler steps backwards through the program for each RSD, seeking targets for message coalescing and aggregation. The search halts if a merge is found, the RSD intersects an RSD representing an intervening definition, or if no more statements exist at the appropriate loop level.

3.1.4 Collective Communication

Message vectorization, coalescing, and aggregation determine the extent to which communication for nonlocal accesses may be combined into a single message. For stencil computations these are point-to-point interprocessor communication and can be performed quite efficiently by individual calls to *send* and *recv* primitives. However, when communication takes place between groups of processors in regular patterns, message overhead can be reduced by utilizing fast *collective communication* routines instead of generating individual messages [7, 36]. These routines can significantly reduce the number of messages required for global communications without affecting parallelism. For instance, computing the global sum of an array on N processors can be reduced from $O(N^2)$ to $O(N \log N)$ messages. Collective communication routines are also very useful for accumulating results of reductions and scans, described in Section 3.3.3.

3.1.5 Run-time Processing

The Fortran D compiler cannot precisely determine at compile-time what communication is required for irregular computations, even though it may be able to apply message vectorization can extract communication out loops. Message coalescing and aggregation cannot be performed at compile-time because the actual array elements accessed are unknown. However, a combination of compile-time analysis and run-time processing can be applied to optimize communication. If no loop-carried true dependences are present, *inspectors* and *executors* may be created at compile-time during code generation to combine messages at run-time [33, 39]. The inspector performs the equivalent of message coalescing and aggregation at run-time. The executor then utilizes collective communication specialized for irregular computations. Special all-to-all *scatter* and *gather* routines collect all the nonlocal data with a small number of messages.

3.1.6 Relax Owner Computes Rule

The *owner computes* rule provides the basic strategy of the Fortran D compiler. We may also relax this rule, allowing processors to compute values for data they do not own. For instance, suppose multiple *rhs* of an assignment statement are owned by a processor that is not the owner of the *lhs*. Computing the result on the processor owning the *rhs* and then sending the result to the owner of the *lhs* could reduce the amount of data communicated. This optimization is a simple application of the "owner stores" rule proposed by Balasundaram [4]. In particular, it may be desirable to partition loops amongst processors so that each loop iteration is executed on a single processor, such as in KALI [33] and ARF [48], in order to eliminate the need for guards and improve load balance.

3.1.7 Replicate Computation

The Fortran D compiler considers scalar variables to be replicated. All processors thus perform computations involving assignments to scalar variables. This causes redundant computation to be performed, but is profitable if it reduces communication costs. A similar approach may be taken for computations on elements of distributed arrays. It may be more efficient to replicate local computations on multiple processors, rather than incur the expense of communicating the value from the owner of that element.

3.2 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. These optimizations can also hide or eliminate T_{copy}, the message copy time, by using *unbuffered messages*.

3.2.1 Message Pipelining

Message pipelining inserts a *send* for each nonlocal reference as soon as it is defined [42]. 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 the next chapter.

3.2.2 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. 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 within a single loop iteration 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. Scheduling *send* and *recv* statements across iterations of the outer time-step loop is analogous to *macro-software pipelining*.

3.2.3 Iteration Reordering

Iteration reordering, also known as *index set splitting*, transforms a loop into multiple loops, each of which contains some of the iterations of the original loop, subject to dependence constraints. It may be used to hide communication cost by separating loop iterations accessing only local data and placing them between *send* and *recv* statements, hiding $T_{transit}$ [33].

3.2.4 Unbuffered Messages

The Fortran D compiler generally uses *buffered messages*, such as the *csend*() and *crecv*() routines from the Intel NX/2 message-passing library. Invoking a buffered *send* causes the calling process to block until the data has been copied out of the program address space into a system buffer. Invoking a buffered *recv* causes the calling process to block until the data has been received and copied into the program address space. In comparison, *unbuffered messages*, such as the *isend*() and *irecv*() routines found in the Intel NX/2 library, can eliminate copying the message to/from the system buffer. An unbuffered *send* returns immediately, sending the message in place. An unbuffered *recv* posts a message destination, allowing the message to be received directly in the user address space while computation is being performed in parallel. To avoid inadvertent overwrites, an additional system call must be made for each unbuffered message to synchronize the computation.

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

3.3 Exploiting Parallelism

3.3.1 Partitioning Computation

Most scientific applications are completely parallel in either a *synchronous* or *loosely synchronous* manner [17]. In these computations all processors execute SPMD programs in a loose lockstep, alternating between phases of local computation and synchronous global communication. These problems achieve good load balance because all processors are utilized during the computation phase. For instance, Jacobi and Red-black SOR are loosely synchronous computations.

If a computation can be determined by the compiler to belong to this class of parallel programs, 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.

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. The following sections describe optimizations that can extract parallelism in the presence of cross-processor dependences.

3.3.2 Private Variables

Statements performing assignments to scalar and replicated array variables present a special challenge for the Fortran D compiler. Naive application of the *owner computes* rule would cause every processor to execute the assignment on all iterations. However, often the assignment can be partitioned because its value is used only in the current loop iteration. These cases are readily recognized, since the variable being assigned will have been labeled *private* during dependence analysis.

3.3.3 Reductions and Scans

Some computations with cross-processor dependences may be parallelized directly. *Reductions* are associative 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 are used to solve a number of computations in scientific codes, including linear recurrences and tridiagonal systems [14, 34].

If a reduction or scan accesses data in a manner that sequentializes computation across processors, it may be parallelized by relaxing the *owner computes* rule and providing methods to combine partial results. 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. 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 additional parallelism yields major performance improvements.

3.3.4 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 [5]. For instance, dynamic data decomposition can internalize the computation wavefront in both phases of ADI integration, allowing processors to execute in parallel without communication [33]. 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, because the computation wavefront crosses both spatial dimensions. Even when dynamic data decomposition is applicable, it may not be efficient, as shown in Section 5.

3.3.5 Pipelined Computations

A technique known as *pipelining* can extract partial parallelism from many *pipelined computations* containing cross-processor dependences. Consider the difference in program execution between parallel and pipelined computations illustrated in Figure 5. 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 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 par-

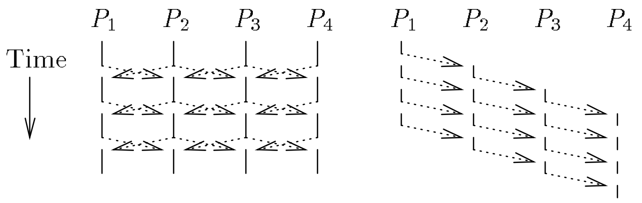


Figure 5 Parallel & Pipelined Computations

allel computations by discovering *cross-processor loops*—loops that cause computation wavefronts to sweep across processor boundaries [27].

3.3.6 Fine-grain Pipelining

We present two optimizations to exploit pipeline parallelism. The key observation we make is that the amount of pipeline parallelism is determined by the amount of computation $\mathcal{C}$ enclosed by the *send* and *recv* primitives inserted around cross-processor loops. Small values for $\mathcal{C}$ exploit more parallelism, but also result in more messages, resulting in higher communication overhead. The Fortran D compiler must thus balance pipeline parallelism with communication overhead, choosing the degree of pipelining that results in the most efficient execution.

To apply *fine-grain pipelining*, the Fortran D compiler 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 as quickly as possible. Because loop-carried true dependences are carried by the cross-processor loop, the standard message vectorization algorithm will place calls to *recv* and *send* primitives before and after the cross-processor loop, respectively, during code generation. This produces the maximum parallelism with the finest granularity of pipelining. The resulting program is a major improvement over sequentialized computation, but incurs the most communication overhead since a message is sent for every iteration accessing nonlocal data.

3.3.7 Coarse-grain Pipelining

Coarse-grain pipelining, in comparison, attempts to balance parallelism with communication overhead, using a combination of loop interchange and strip-mining where needed to adjust the granularity of pipelining. A simple heuristic for coarse-grain pipelining is implemented in the prototype Fortran D compiler. It first applies fine-grain pipelining, then strip-mines the deepest loop enclosing the cross-processor loops. Communication is inserted outside of the newly strip-mined loop. The granularity of pipelining is determined by the strip size, discussed later in Section 5.2.3. Because loop interchange may affect usage of registers and cache, the Fortran D compiler restores the original innermost loop after strip-mining. More sophisticated approaches would permute inner loops into *memory order* to exploit data locality for the local computation [31].

3.4 Reducing Storage

3.4.1 Partitioning Data

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 placed 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.

3.4.2 Message Blocking

If insufficient storage is available, buffers must be used to store nonlocal data. *Message blocking* may then be applied to reduce the buffer storage needed. A loop carrying communication is strip-mined by a block factor B. Each vectorized message of size n is then divided into n/B messages of size B and sent inside the strip-mined loop. This reduces the buffer space required by a factor of n/B at the expense of additional messages.

4 Empirical Performance Evaluation

To evaluate the usefulness of each compiler optimization, we applied them where appropriate to a small selection of scientific program kernels adapted from the Livermore Kernels [37] and finite-difference algorithms. These kernels, shown in Figure 6, contain *stencil computations* and reductions, techniques commonly used by scientific programmers to solve partial differential equations (PDEs) [9, 17]. Details of how optimizations were applied are discussed elsewhere [28, 46].

Most optimizations were performed by hand, simulating algorithms we later implemented in the compiler. At this point the prototype Fortran D compiler performs message vectorization, coalescing, aggregation, vector message pipelining, unbuffered messages, reductions, broadcasts, fine-grain pipelining, and coarse-grain pipelining (with a preset granularity). For simplicity, it restricts array sizes and loop bounds to be compile-time constants, and requires array subscripts to of the form $i + c$ or $i - c$, where c is a compile-time constant and i is a loop index variable.

Table 1 shows the optimized versions of each program. *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

Parallel Kernels

```
— Livermore 18 —
do k = 2,N-1
   do j = 2,N-1
     ZA(j,k) = F1(ZP(j-1,k),ZQ(j-1,k),ZM(j-1,k),...)
     ZB(j,k) = F2(ZP(j-1,k),ZQ(j-1,k),...)
do k = 2,N-1
   do j = 2,N-1
     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,N-1
   do j = 2,N-1
     ZR(j,k) = F5(ZR(j,k),ZU(j,k))
     ZZ(j,k) = F6(ZZ(j,k),ZV(j,k))

— Jacobi —
do j = 2,N-1
   do i = 2,N-1
     A(i,j) = F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
do j = 2,N-1
   do i = 2,N-1
     B(i,j) = A(i,j)

— Red-Black SOR —
do j = 3,N-1,2       {* compute red points *}
   do i = 3,N-1,2
     V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
do j = 2,N-2,2
   do i = 2,N-2,2
     V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
do j = 2,N-2,2       {* compute black points *}
   do i = 3,N-1,2
     V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
do j = 3,N-1,2
   do i = 2,N-2,2
     V(i,j) = F(V(i,j-1),V(i-1,j),V(i,j+1),V(i+1,j))
```

Reduction & Scan Kernels

```
— Livermore 3 —
Q = 0.0
do k = 1,100
   Q = Q + Z(k)*X(k)

— Livermore 11 —
X(1) = Y(1)
do k = 2,100
   X(k) = X(k-1) + Y(k)
```

Pipelined Kernels

```
— Livermore 23 —
do j = 2,N-1
   do k = 2,N-1
     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)

— SOR —
do j = 2,N-1
   do i = 2,N-1
     A(i,j) = F(A(i,j-1),A(i-1,j),A(i+1,j),A(i,j+1))

— ADI Integration —
do j = 1, N      {* Phase 1: sweep along columns *}
   do i = 2, N
     X(i,j) = F1(X(i,j),X(i-1,j),A(i),B(i))
do j = 2, N      {* Phase 2: sweep along rows *}
   do i = 1, N
     X1(i,j) = F2(X(i,j),X(i,j-1),A(i),B(i))
```

Figure 6 Stencil Kernels Employed To Evaluate Optimizations

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. All arrays are double precision and distributed block-wise in one dimension. Timings were taken using *dclock*() for one iteration of l, the time step loop. Results presented are both tabulated and plotted graphically.

In Tables 2, 3, and 4, 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. In addition to the timings, each table contains ratios of execution times for some selected optimizations, illustrating their relative usefulness. Figure 7 presents the same timings graphically. In the figure, program execution times in seconds are plotted logarithmically along the Y-axis. Optimizations are plotted along the X-axis, ranging from sequential execution (no optimization) to ideal parallel execution (no communication). Dotted, dashed, and solid lines represent execution times for 8, 16, and 32 processors, respectively. Lines are marked with $\circ$, $\bullet$, $\star$ and other symbols to represent the problem size.

Table 1 Optimized Versions of Test Kernels

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/ unbuffered 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

4.1 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 for which we present results are 2D stencil computations with 1D data distributions. Enough communication is required to make optimizations significant.

Table 2 presents the performance of communication optimizations for parallel stencil kernels. For parallel 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 unbuffered messages. Unbuffered 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.

4.2 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.

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, because computation is doubled in the parallel scan.

Table 2 Performance of Communication Optimizations (in milliseconds)

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 3 Performance of Optimizations to Parallelize Reductions and Scans (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 4 Performance of Optimizations to Exploit Pipeline Parallelism (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	431	135	-	145	50	40	39	52	2.97	3.72	3.25	9.8
		1Kx1K	53	1800	382	-	321	121	99	96	123	5.61	3.34	1.81	17.7
		2Kx2K	213	7010	1180	-	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

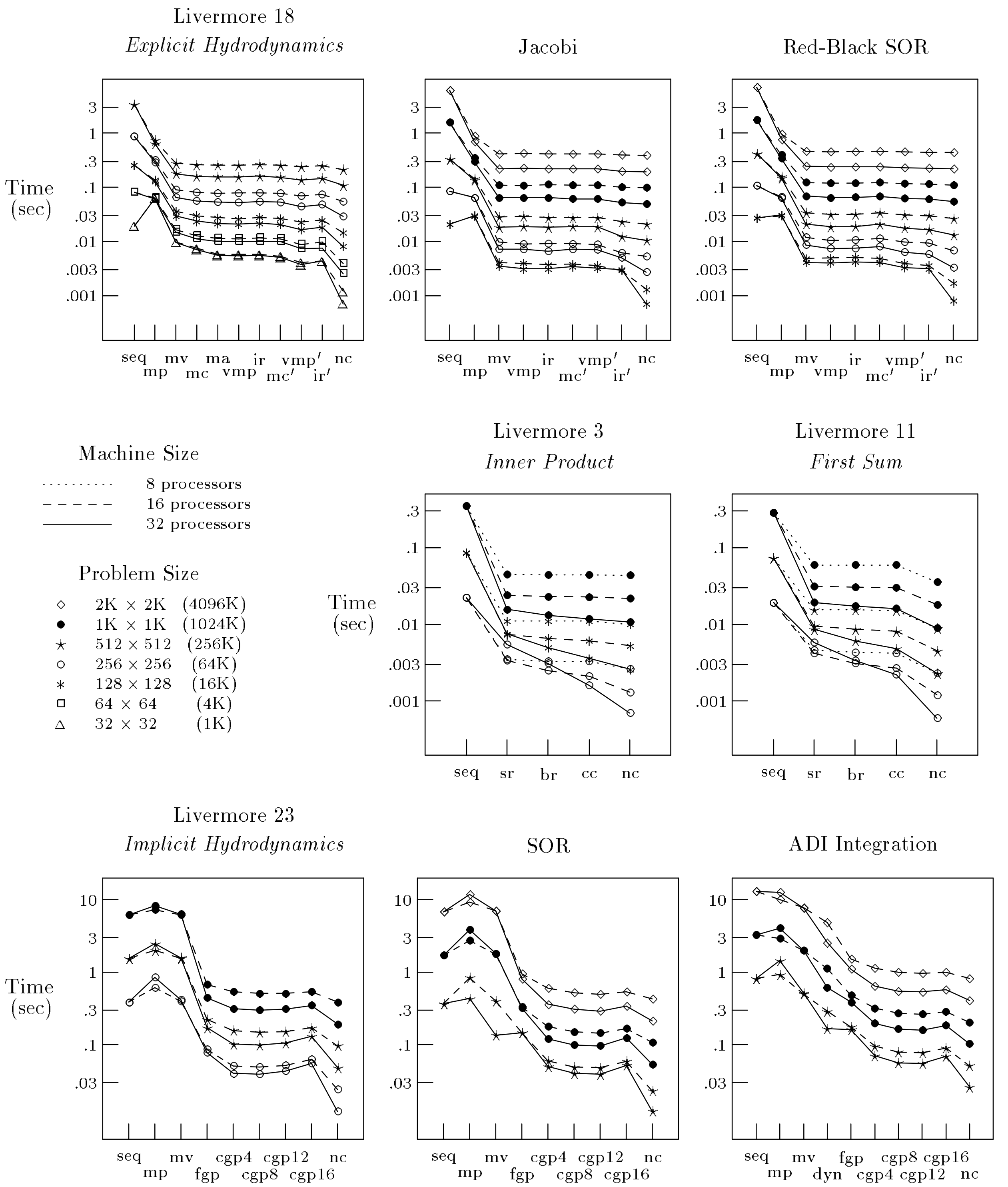


Figure 7 Results for Communication and Parallelism Optimizations

4.3 Optimizations for Pipelined Computations

Timings for pipelined computations are tabulated in Table 4. 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 unbuffered 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 pipelined computations.

The performance of dynamic data decomposition displays almost linear speedup with respect to the number of processors (speedup of 1.7–1.9 going from 16 to 32 processors), at least for the problem sizes tested. However, applying dynamic data decomposition to redistribute arrays in ADI proved to be undesirable and required significantly more time than pipelining, especially as problem size increases. Dynamic data decomposition should prove to be a profitable optimization on machines employing larger numbers of processors, but more experimentation is needed.

4.4 Ranking Optimizations

We can summarize our empirical evaluation by ranking the effectiveness of compiler optimizations in the following order:

1. *Exploit parallelism* — The key step in improving performance, particularly with large problems or many processors.

2. *Reduce communication overhead* — Three optimizations are notable because they can eliminate large numbers of messages: message vectorization, coarse-grain pipelining, and collective communications. These optimizations can significantly improve overall performance because of high message startup costs on MIMD distributed-memory machines such as the Intel iPSC/860.

3. *Remaining communication optimizations* — Though not as dramatic, the remaining optimizations are useful in reducing or hiding the cost of communication. This may improve overall performance, especially when communication comprises a significant percentage of total execution time even after applying message vectorization.

5 Analysis of Optimizations

This section presents analysis and decision algorithms to evaluate the cost and effectiveness of the communication and parallelism optimizations presented in the previous sections. They may be used by the Fortran D to determine when an optimization may be profitably applied. The success of these decision algorithms depends on how accurately we can estimate the cost of different computation and communication operations for the underlying machine.

5.1 Communication Optimizations

We begin by analyzing the effects of communication optimizations on 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 unbuffered 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.

Table 5 Effect of Compiler Optimizations

Optimization	Resulting Communication Overhead (For n Elements)
none	$n(T_{start} + T_{copy}(1) + T_{transit}(1))$
mp	$n(T_{start} + T_{copy}(1) + \mathrm{pos}(T_{transit}(1) - T_{comp}))$
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	$T_{start} + T_{copy}(n) + \mathrm{pos}(T_{transit}(n) - T_{comp})$
ir	$T_{start} + T_{copy}(n) + \mathrm{pos}(T_{transit}(n) - T'_{comp}) + \Delta T_{comp}$
mv′	$T'_{start} + T_{transit}(n)$
vmp′	$T'_{start} + \mathrm{pos}(T_{transit}(n) - T_{comp})$
ir′	$T'_{start} + \mathrm{pos}(T_{transit}(n) - T'_{comp}) + \Delta T_{comp}$

5.1.1 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.

5.1.2 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.

5.1.3 Unbuffered messages

Using unbuffered messages eliminates T_{copy} by eliminating copying on the sending and receiving processors. However, the resulting program incurs a higher startup cost T'_{start}. It is profitable to use unbuffered messages with vector message pipelining when:

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

The compiler will use unbuffered 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, unbuffered messages become more useful as message size increases.

5.1.4 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 buffered messages, iteration reordering can profitably enhance vector message pipelining when the following conditions hold:

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

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 unbuffered messages is profitable when:

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

The criteria are similar to that of buffered 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.

5.2 Parallelism Optimizations

In this section we analyze optimizations to exploit parallelism. It is essential that computation be partitioned, even for private variables. Reductions and scans should always be identified and parallelized, using collective communication to accumulate results. Dynamic data decomposition may extract parallelism, but usually with high cost. We show analytically that exploiting pipeline parallelism through either fine-grain or coarse-grain pipelining is both effective and scalable.

5.2.1 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 more likely to be profitable 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. On the other hand, dynamic data decomposition becomes more effective as the number of processors increases, because it may be performed concurrently [15]. It may also eliminate communication in other phases of the computation not part of the pipelined computation. As a result, determining when dynamic data decomposition should be performed remains an open research area.

5.2.2 Fine-grain Pipelining

Consider the simple examples presented in Figure 8. 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 block-wise 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

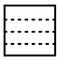
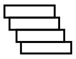
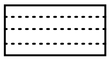
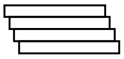
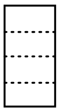
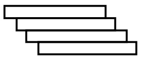
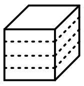
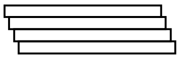
Data Arrays & Data Partition	Computation & Elapsed Time	Sequential Time	Parallel Time (fgp) & Parallel Time Blocked (cgp)	Block Size Preferred
		n^2	$fgp = \frac{n^2}{p} + nC + (p-1)(\frac{n}{p} + C)$ $cgp = \frac{n^2}{p} + \frac{nC}{B} + (p-1)(\frac{nB}{p} + C)$	$\sqrt{C}$
		$2n^2$	$fgp = \frac{2n^2}{p}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

Figure 8 Effectiveness of Pipelining

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 8. 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.

5.2.3 Coarse-grain Pipelining

The same model may also be used to calculate an efficient buffered 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. 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 8.

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 [7]. The Fortran D compiler will employ a flexible and precise approach using *training sets* to estimate communication and computation costs [6, 25, 30]. Accurate static estimates of communication and computation are also needed by the compiler to calculate block sizes for coarse-grain pipelining.

6 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.

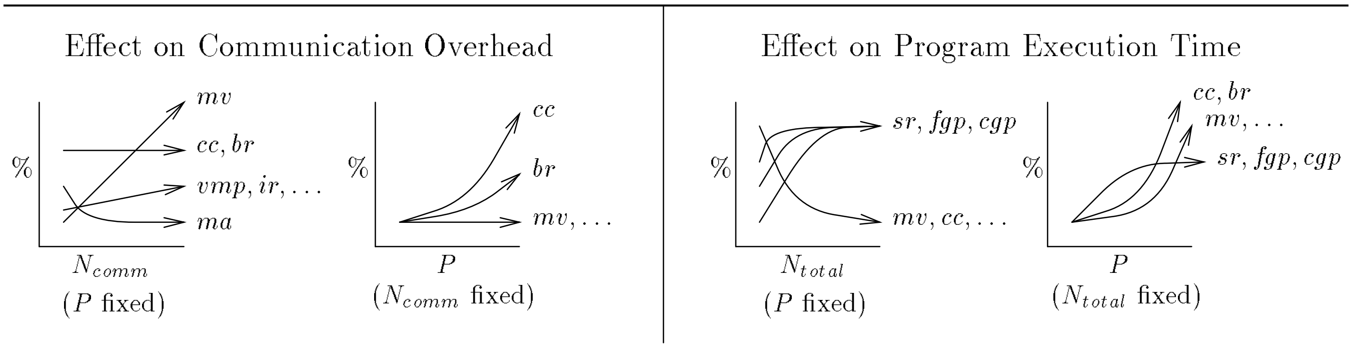


Figure 9 Scalability of Optimizations with Respect to N_{comm}, N_{total}, and P

6.1 Communication overhead

The first half of Figure 9 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, ...) 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, ...) 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.

6.2 Program execution

The second half of Figure 9 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, ...) shrink in relative usefulness because N_{comm} grows slowly compared to N_{total} for stencil computations.

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.

6.3 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

Table 6 Data Communication Requirements

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

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 10. 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 distributed-memory compiler that emphasizes compile-time analysis and optimization instead of language extensions and run-time support. Unlike earlier systems such as Callahan & Kennedy [11] and Superb [19, 50], it applies analysis and optimization up front before code generation, rather than first inserting guards and element-wise messages then optimizing via program transformations and partial evaluation. The Fortran D compiler is similar to other *owner computes* systems such as Adapt [38], Adaptor [8], Aspar [29], Forge90 [3], P^3C [18], and Vienna Fortran [12], but performs more compile-time optimization and requires less run-time support.

As previously stated, most optimizations in this paper have been discovered by other researchers. Message vectorization was first developed by Gerndt [19] and Balasundaram *et al.* [5]. Most distributed-memory compilers implicitly perform message vectorization by extracting messages out of user-specified parallel regions, usually performing message coalescing and aggregation as well. Compilers that take this approach require the user to specify parallel loops (Al [47], Arf [48], Kali [33]), parallel functions (C* [43], Dataparallel C [22], Dino [44]), parallel code blocks

```
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 unbuffered messages
      else if T'_comp can be profitably created & used then
         reorder iterations
         apply vector message pipelining
         insert unbuffered messages
      else
         insert buffered messages
      endif
   else        {* must be pipelined computation *}
      select efficient granularity for pipelining
      apply strip-mining & loop iterchange
      vectorize, coalesce, and aggregate messages
      insert buffered messages
   endif
   if insufficient storage is available then
      apply storage optimizations
   endif
enddo
```

Figure 10 Fortran D Optimization Algorithm

(Oxygen [45]), or parallel array operations (CM Fortran [9], Paragon [13]). Other compilers identify parallelism and vectorize messages automatically using the single assignment semantics of their high-level functional languages (Crystal [36], Id Nouveau [42]). In comparison, compilers that explicitly use data dependence information (such as the Fortran D compiler) can extract communication even out of sequential regions such as those found in ADI or SOR.

Other optimizations are less common. Crystal pioneered the strategy of identifying collective communication opportunities through syntactic analysis. Gupta & Banerjee describe a sophisticated framework for analyzing collective communication and estimating pipelining costs in Parafrase-2 [20]. Id Nouveau applies message vectorization, message pipelining, and recognizes reductions. Kali performs iteration reordering for individual parallel loops and suggests using array transposition as a form of dynamic data decomposition for ADI integration [33]. Aspar [29] and P^3C [18] perform simple dependence analysis to extract communication from parallel loops, and rely on portable run-time libraries to support collective communication and reductions. The Suif compiler uses precise data-flow information for arrays from *last-write-trees* to avoid over-communication [2].

Few optimizations have been evaluated either analytically or through empirical measurements to determine their effectiveness. Rogers & Pingali performed experiments on the Intel iPSC/2 that showed message pipelining is needed to extract parallelism from a cyclically distributed Gauss-Seidel computation [42]. Strip-mining the computation by hand to increase the granularity of pipelining also improved performance. However, the Id Nouveau compiler does not distinguish between parallel and pipelined computations. It is thus unable to determine when message pipelining is needed to exploit pipeline parallelism. The Fortran D compiler, in comparison, can automatically recognize and optimize pipelined computations in a more sophisticated manner, including modifying the program execution order.

The CM Fortran compiler extracts communications from array operations and handles reductions expressed as array intrinsics. Bromley *et al.* develop optimization techniques for CM-2 stencils that avoid unnecessary intra-processor data motion, insert communication only for nonlocal data, and improve register usage, resulting in significant improvements in execution speed [9]. Except for optimizations to improve register reuse, these techniques appear to be subsumed by standard Fortran D optimizations.

Burns *et al.* developed techniques for guiding the use of unbuffered messages on the Alliant CAMPUS/800 using data dependence information [10]. They show that unbuffered messages improve overall performance for a collection of hand-parallelized scientific programs. Olander & Schnabel show that Dino programs can be significantly improved through iteration reordering and pipelining [41]. Their studies validate the effectiveness of selected compiler optimizations for complete programs. Rühl performed studies on a variety of parallel architectures, demonstrating excellent speedups for the Oxygen compiler [45].

The optimizations described in this paper have been targeted towards regular computation on dense matrices. The same principles apply, but in a different form, for irregular or adaptive computations on sparse matrices. Compilers such as Arf [48] and Kali [33] support irregular computations by creating *inspectors* and *executors* for each loop nest to detect and combine messages for nonlocal accesses at run-time [39]. This approach may be viewed as a sophisticated version of message vectorization, coalescing, and aggregation.

Finally, two groups have analyzed the scalability of optimizations to exploit pipeline parallelism. Naik explores parallelization and optimization of pipelined computations in the context of CFD applications [40]. Results show that in addition to increases in problem size and processor speed, improvements in algorithms and data partitioning are also necessary to yield scalable parallelism.

Yeung & Agarwal examine the scalability of fine and coarse-grain pipelining for pipelined computations in the context of the forward and backward substitution stages of a preconditioned conjugate gradient solver [49]. They find that coarse-grain pipelining is not scalable if the granularity of pipelining is proportional to the problem size divided by the number of processors. In comparison, we find that coarse-grain pipelining is highly scalable when the granularity of pipelining is selected according to the analytical models presented in this paper.

9 Conclusions

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, coarse-grain pipelining, and collective communication also significantly affect performance by eliminating large numbers of messages. The remaining optimizations yield less dramatic results, but are still important when the proportion of nonlocal to local data is high. This is the case when attempting to speed up a problem with fixed size. Profitability formulas may be used to intelligently choose between optimization options, but rely on accurate measurements of machine parameters through the use of training sets.

High performance is essential to the success of Fortran D as 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. We have incorporated many of the optimizations described in this paper in the prototype Fortran D compiler. Preliminary experiences show it can produce highly optimized code for stencil programs that is competitive with hand-optimized code [46]. Ongoing work to provide environmental support for automatic data decomposition and static performance estimation will also enhance the usefulness of the Fortran D compiler [5, 6, 25].

10 Acknowledgements

The authors wish to thank Joel Saltz, Robert Schnabel and Robert Weaver for several enlightening discussions. We are also grateful to the ParaScope research group for providing the software infrastructure for the Fortran D compiler.

References

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

[2] Amarasinghe, S., and Lam, M. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation* (Albuquerque, NM, June 1993).

[3] Applied Parallel Research. *Forge 90 Distributed Memory Parallelizer: User's Guide*, version 8.0 ed. Placerville, CA, 1992.

[4] Balasundaram, V. Translating control parallelism to data parallelism. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing* (Houston, TX, Mar. 1991).

[5] Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference* (Charleston, SC, Apr. 1990).

[6] Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. 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, Apr. 1991).

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

[8] Brandes, T. Efficient data parallel programming without explicit message passing for distributed memory multiprocessors. Internal Report AHR-92-4, High Performance Computing Center, GMD, Sept. 1992.

[9] Bromley, M., Heller, S., McNerney, T., and Steele, Jr., G. 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).

[10] Burns, C., Kuhn, R., and Werme, E. Low copy message passing on the Alliant CAMPUS/800. In *Proceedings of Supercomputing '92* (Minneapolis, MN, Nov. 1992).

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

[12] Chapman, B., Mehrotra, P., and Zima, H. Programming in Vienna Fortran. *Scientific Programming 1*, 1 (Fall 1992), 31–50.

[13] Chase, C., Cheung, A., Reeves, A., and Smith, M. Paragon: A parallel programming environment for scientific applications using communication structures. *Journal of Parallel and*

Distributed Computing 16, 2 (Oct. 1992), 79–91.

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

[15] Choudhary, A., Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Ranka, S., and Tseng, C. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation* (McLean, VA, Oct. 1992).

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

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

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

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

[20] Gupta, M., and Banerjee, P. A methodology for high-level synthesis of communication for multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing* (Washington, DC, July 1992).

[21] Hall, M. W., Hiranandani, S., Kennedy, K., and Tseng, C. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92* (Minneapolis, MN, Nov. 1992).

[22] Hatcher, P., and Quinn, M. *Data-parallel Programming on MIMD Computers.* The MIT Press, Cambridge, MA, 1991.

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

[24] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, Jan. 1993.

[25] Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., and Tseng, C. An overview of the Fortran D programming system. In *Languages and Compilers for Parallel Computing, Fourth International Workshop* (Santa Clara, CA, Aug. 1991), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Springer-Verlag.

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

[27] Hiranandani, S., Kennedy, K., and Tseng, C. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM 35*, 8 (Aug. 1992), 66–80.

[28] Hiranandani, S., Kennedy, K., and Tseng, C. 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).

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

[30] Kennedy, K., McIntosh, N., and McKinley, K. S. Static performance estimation in a parallelizing compiler. Tech. Rep. TR91-174, Dept. of Computer Science, Rice University, Dec. 1991.

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

[32] Kennedy, K., McKinley, K. S., and Tseng, C. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice & Experience 5*, 7 (Oct. 1993), 575–602.

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

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

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

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

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

[38] Merlin, J. ADAPTing Fortran-90 array programs for distributed memory architectures. In *First International Conference of the Austrian Center for Parallel Computation* (Salzburg, Austria, Sept. 1991).

[39] Mirchandaney, R., Saltz, J., Smith, R., Nicol, D., and Crowley, K. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing* (St. Malo, France, July 1988).

[40] Naik, V. Scalability issues for a class of CFD applications. In *Proceedings of the 1992 Scalable High Performance Computing Conference* (Williamsburg, VA, Apr. 1992).

[41] Olander, D., and Schnabel, R. 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, Apr. 1992).

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

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

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

[45] Rühl, R. Evaluation of compiler-generated parallel programs on three multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing* (Washington, DC, July 1992).

[46] Tseng, C. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Dept. of Computer Science, Rice University, Jan. 1993.

[47] Tseng, P.-S. 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).

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

[49] Yeung, D., and Agarwal, A. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993).

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

Biographies

Seema Hiranandani

Seema Hiranandani graduated *cum laude* from Smith College in 1988 with a B.A. in Computer Science and was awarded the Bert Mendelson Prize for excellence in Computer Science at graduation. She continued her studies and earned an M.S. in Computer Science from the University of Massachusetts at Amherst in 1989. She worked at Yale University until April 1990. Since then she has been a member of the Center for Research on Parallel Computation at Rice University, where she has been actively involved in the Fortran D compiler project. Her research interests include programming environments and compilation techniques for high performance architectures.

Ken Kennedy

Ken Kennedy received a B.A. in mathematics from Rice University in 1967, an M.S. in mathematics from New York University in 1969, and a Ph.D. in computer science from New York University in 1971. Since 1971, he has been a faculty member at Rice University. Professor Kennedy served as Chair of the Department of Computer Science in 1984–88 and 1990–92. He has held a Noah Harding Professorship since 1985. Professor Kennedy is currently Director of the Center for Research on Parallel Computation, an NSF Science and Technology Center. In 1990, he was elected to the National Academy of Engineering. He chairs the High Performance Fortran Forum, which is seeking to define an informal standard for machine-independent parallel programming extensions to Fortran 90.

Professor Kennedy has published over ninety technical articles and supervised more than twenty Ph.D. dissertations on programming support software for high-performance computer systems. He has supervised the construction of two substantial software systems for programming parallel machines: an automatic vectorizer for Fortran 77 and an integrated scientific programming environment. His current research focuses on extending techniques developed for automatic vectorization to programming tools for parallel computer systems and high-performance microprocessors. Through the Center for Research on Parallel Computation, he is seeking to develop new strategies for supporting architecture-independent parallel programming, especially in science and engineering.

Chau-Wen Tseng

Chau-Wen Tseng received the A.B. degree in computer science from Harvard University in 1986, and the M.S. and Ph.D. degrees in computer science from Rice University in 1992 and 1993. He spent a year as a Research Scientist at Rice University and is currently a Research Associate at Stanford University. Dr. Tseng's research interests are in the field of parallel and distributed systems, in particular optimizing and parallelizing compilers. He has contributed to the design and implementation of the dependence analyzer and parallelizing editor in the ParaScope parallel programming environment. His dissertation focuses on compiling and optimizing Fortran D for MIMD distributed-memory machines.