Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures [3]

Raja Das Mustafa Uysal Joel Saltz[2] Yuan-Shin Hwang

Department of Computer Science
University of Maryland
College Park, MD 20742
Email : {raja, uysal, saltz, shin}@cs.umd.edu

Abstract

This paper describes a number of optimizations that can be used to support the efficient execution of irregular problems on distributed memory parallel machines. These primitives (1) coordinate interprocessor data movement, (2) manage the storage of, and access to, copies of off-processor data, (3) minimize interprocessor communication requirements and (4) support a shared name space. We present a detailed performance and scalability analysis of the communication primitives. This performance and scalability analysis is carried out using a workload generator, kernels from real applications and a large unstructured adaptive application (the molecular dynamics code CHARMM).

1 Introduction

Over the past few years we have developed a methodology to produce efficient distributed memory code for sparse and unstructured problems in which array accesses are made through a level of indirection. In such problems the dependency structure is determined by variable values known only at runtime. In these cases effective use of distributed memory architectures is made possible by a runtime preprocessing phase. The methods we have developed are implemented using compiler embeddable runtime support procedures. These procedures can be thought of as supporting a type of weakly coherent distributed shared memory that can be closely coupled to distributed shared memory compilers. These primitives (1) coordinate interprocessor data movement, (2) manage the storage of and access to copies of off-processor data, (3) minimize interprocessor communication requirements and (4) support a shared name space.

This paper describes and systematically evaluates a number of new optimizations that have been incorporated into the existing toolkit. Several of the optimizations reduce communication latency and volume. We also describe a paged translation table to efficiently support access to globally indexed distributed arrays that are mapped onto processors in an irregular manner.

[2]UMIACS, University of Maryland, College Park, MD 20742
[3]Research supported by ARPA under contract No. NAG-1-1485, NSF under contract No. ASC 9213821 and ONR under contract No. SC 292-1-22913

We characterize the performance and scalability of our optimizations using (1) a family of test loops and an associated synthetic workload generator, (2) test loops with data access patterns drawn from unstructured applications and templates from real applications. This combination of empirical approaches allows us to characterize the performance of our optimizations under a wide variety of conditions. In order to provide a conceptual framework for understanding ways of evaluating the relative effectiveness of an optimization on varying numbers of processors, we define the concept of *optimization scalability.*

In sparse and unstructured computations, distributed arrays are typically accessed using indirection. In many cases (e.g. distributed arrays referenced in loops with no loop carried dependencies or distributed arrays referenced in loops with accumulation type dependencies) it is possible to prefetch required off-processor data before a loop begins execution. In many cases several loops access the same off-processor memory locations. As long as it is known that the values assigned to off-processor memory locations remain unmodified, it is possible to reuse stored off-processor data. A mixture of compile-time and run-time analysis can be used to generate efficient code for irregular problems [9], [11], [15]. In this paper we give a detailed description of the communication optimizations that prove to be useful for optimizing irregular problem performance but do not further address compilation issues.

1.1 Overview of PARTI

In this section, we give an overview of the functionality of the PARTI primitives described in previous publications [22], [1], [17]. In many algorithms data produced or input during program initialization plays a large role in determining the nature of the subsequent computation. In the PARTI approach, when the data structures that define a computation have been initialized, a preprocessing phase follows. Vital elements of the strategy used by the rest of the algorithm are determined by this preprocessing phase.

In distributed memory machines large data arrays need to be partitioned between local memories of processors. These partitioned data arrays are called distributed arrays. Long term storage of distributed array data is assigned to specific memory locations in the distributed machine. It is frequently advantageous to partition distributed arrays in an irregular manner. For instance, the way in which the nodes of an irregular computational mesh are numbered frequently does not have a useful correspondence to the connectivity pattern of the mesh. When we partition the data structures in such a problem in a way that minimizes interprocessor communication, we may need to be able to assign arbitrary array elements to each processor.

Each element in a distributed array is assigned to a particular processor. In order for another processor to be able to access a given element of the array we must know the processor in which it resides and its local address in that processor's memory. We thus build a *translation table* which, for

```
do i = 1, n
  x(ia(i)) = x(ia(i)) + y(ib(i))
end do
```

Figure 1: Simple Irregular Loop

each array element, lists the host processor address.

The class of problem with which we are dealing consists of a sequence of clearly demarcated concurrent computational phases where data access patterns cannot be anticipated until runtime. They are called static irregular concurrent computations [1]. In these problems, once runtime information is available, 1) data access patterns are known before each computational phase and 2) the same data access patterns occur many times. Adaptive problems can fall into this class of problem as long as data access patterns change relatively infrequently. A typical loop in such computations is shown in Figure 1. In this loop the arrays **x**, **y**, **ia** and **ib** are all distributed arrays. The arrays **ia** and **ib** are used to index the arrays **x** and **y** respectively. At compile time it is not possible to figure out the indices of **x** and **y** that are accessed because they are dependent on the values stored in the arrays **ia** and **ib**. This data access pattern becomes available at runtime.

In distributed memory MIMD architectures there is typically a non-trivial communications latency or startup cost. For efficiency reasons information to be transmitted should be collected into relatively large messages. The cost of fetching array elements can be reduced by precomputing what data each processor needs to send and receive.

In irregular problems, such as solving PDEs on unstructured meshes and sparse matrix algorithms, the communication pattern depends on the input data. This typically arises due to some level of indirection in the code. In this case it is not possible to predict at compile time what data must be prefetched. To deal with this lack of information the original sequential loop is transformed into two constructs, namely the *inspector* and the *executor*.

During program execution the *inspector* loop examines the data references made by a processor and calculates what off-processor data needs to be fetched and where that data will be stored once it is received. The *executor* loop then uses the information from the inspector to implement the actual computation. We have developed a suite of primitives that can be used directly by programmers to generate inspector/executor pairs.

These primitives are named PARTI [5], [1]; they carry out the distribution and retrieval of globally indexed but irregularly distributed data-sets over the numerous local processor memories. Each inspector produces a set of *schedules*, which specify the communication calls needed to either: (1) obtain copies of data stored in specified off-processor memory locations (i.e. *gather*) or, (2) modify

```
 C  Create the required schedules (Inspector)
S1  call localize(dist_of_x, schedule_ia, part_ia, local_ia, n_local_iterations, off_proc_x)
S2  call localize(dist_of_y, schedule_ib, part_ib, local_ib, n_local_iterations, off_proc_y)
 C  The actual computation (Executor)
S3  call zero_out_buffer(x(begin_buffer_x), off_proc_x)
S4  call gather(y(begin_buffer_y), y, schedule_ib)
L1  do i = 1, n_local_iterations
S5     x(local_ia(i)) = x(local_ia(i)) + y(local_ib(i))
S6  end do
S7  call scatter_add(x(begin_buffer_x), x, schedule_ia)
```

Figure 2: Node Code for Simple Irregular Loop

the contents of specified off-processor memory locations (i.e. *scatter*), or (3) accumulate (e.g. add or multiply) values to specified off-processor memory locations, (i.e. *accumulate*).

The primitives that build schedules use hash tables to generate communication calls that, for each loop nest, transmit only a single copy of each off-processor datum [12], [22]. The schedules are used in the executor by PARTI primitives to gather, scatter and accumulate data to/from off-processor memory locations. In this paper, the idea of eliminating duplicates has been taken a step further. If several loops require different but overlapping data references we can avoid communicating redundant data (See Section 4.1.2).

The description of the functionality of the PARTI primitives is presented by transforming the example loop shown in Figure 1 such that it can be executed on a distributed memory machine. The arrays **x**, **y**, **ia** and **ib** are partitioned between the processors of the parallel machine. The partitioned indirection arrays **ia** and **ib** are called **part_ia** and **part_ib** on each processor and they are passed to the function *localize* as shown in statement S1 and S2 in Figure 2. The *localize* calls shown in Figure 2 contains the inspector code for the simple irregular loop shown in Figure 1. On each processor the function *localize* is passed the following:

1. a pointer to the distribution information, (dist_of_y in S2),
2. a list of globally indexed distributed array references for which processor P will be responsible, (**part_ib** in S2), and
3. number of globally indexed distributed array references (n_local_iterations in S2).

Localize returns:

1. a schedule that can be used in PARTI gather and scatter procedures (schedule_ib in S2),

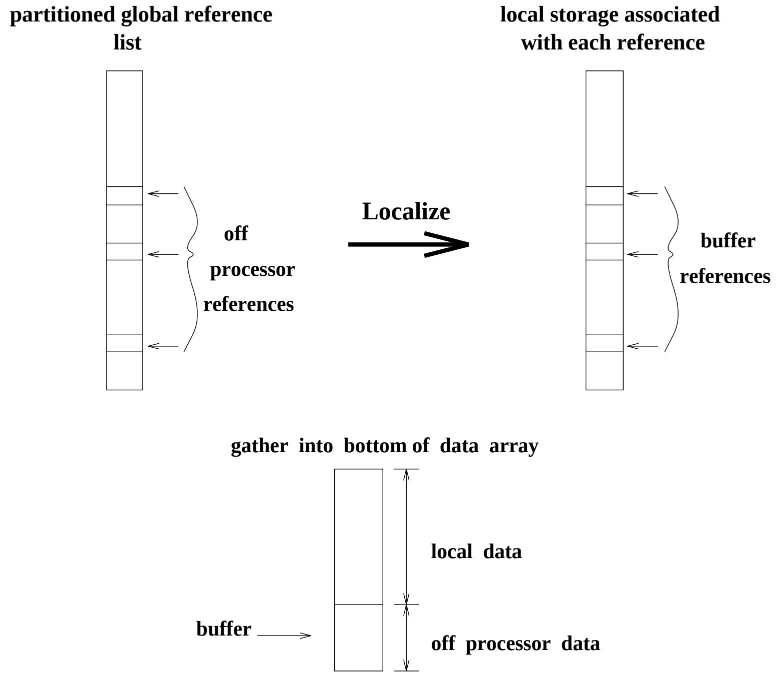

Figure 3: Localize Mechanism

2. an integer array that can be used to specify the pattern of indirection in the executor code (**local_ib** in S2), and
3. number of distinct off-processor references found in part_ia (off_proc_y in S2).

A sketch of how the procedure *localize* works is shown in Figure 3. The array **part_ib** in statement S2 in Figure 2 contains the globally indexed reference pattern used to access array **y**. *Localize* translates **part_ib** so that valid references are generated when the loop is executed. The buffer for each data array immediately follows the on-processor data for that array. For example, the buffer for data array **y** begins at **y(begin_buffer)**. Hence, when *localize* translates **part_ib** to **local_ib**, the off-processor references are modified to point to buffer addresses. When the off processor data is collected into the buffer using the schedule returned by *localize*, the data is stored in a way such that execution of the loop using the **local_ib** accesses the correct data. Similarly, **part_ia** is also translated to point to local addresses during the *localize* call shown in statement S2.

The executor code shown in Figure 2 carries out the actual loop computation. In this computation the values stored in the array **x** are being updated using the values stored in **y**. During the computation, accumulations to off-processor locations of array **x** are carried out in the buffer associated with array **x**. This makes it necessary to initialize to zero the buffer corresponding to off-processor references of **x**. We call the PARTI function *zero_out_buffer* shown in statement S3 to perform this action. After the loop computation, the data in the buffer location of array **x** is communicated to the home processors of these data elements (*scatter_add*). There are two potential communication points in the executor code i.e. the *gather* and the *scatter_add* calls. The *gather* on each processor fetches all the necessary **y** references that reside off-processor. The *scatter_add* calls accumulates the off-processor **x** values. A detailed description of the functionality of these primitives are given in [6].

2 Real Problem Kernels

We have ported a number of scientific codes to parallel machines using the PARTI primitives. In this section we briefly describe two kernels that have been extracted from these codes and how they stress the primitives quite differently. In Section 2.1 we describe a representative kernel from the explicit Euler solver [14], [5] and Section 2.2 briefly describes the kernel from the molecular dynamics code CHARMM [3], [7].

2.1 Unstructured Euler Kernel

Unstructured meshes provide a great deal of flexibility in discretizing complex domains and offer the possibility of easily performing adaptive meshing. However, unstructured meshes result in random data-sets and large sparse matrices which require runtime preprocessing to be executed on a distributed memory machine. The average connectivity of each mesh point is between 6 and 10, hence the number of duplicate data references is low. This connectivity is quite low when compared with the connectivity that is generated for other types of problems such as molecular dynamics or particle dynamics. There are 8 irregular loops in the kernel and the number of indirection arrays is 2. The size of these indirection arrays are 6 to 10 times the size of the data arrays. The communication pattern generated by this kernel is dependent on the type of partitioning method used. When the data and the indirection arrays are block partitioned the number of off-processor data is extremely high, and nearly all processors have to communicate with each other. When the data is partitioned based on the connectivity of the mesh (spectral bisection [16]), we get low volumes of communication between neighboring processors. We have used geometric partitioners with fair amount of success.

2.2 Molecular Dynamics

This kernel from the molecular dynamics code CHARMM has a very different data access pattern from that of the Euler code kernel. Part of this kernel consists of a non-bonded force calculation. Figure 4 depicts the non-bonded force calculation loop. The array **Partners(i, *)** stores all the atoms that interacts with atom i. Inside the inner loop, the three force components between atom i and atom j are calculated (Vander Waal's and electrostatic forces). These forces are then added to the total forces associated with atom i and subtracted from the forces associated with atom j. All atoms within a given cutoff radius interact with each other. The cutoff radius is usually comparable to the geometric size

```
do i = 1, NATOM
    do index = 1, INB(i)
        j = Partners(i, index)
        Calculate dF (x, y and z components).
        Subtract dF from F_j.
        Add dF to F_i
    end do
end do
```

Figure 4: Non-bonded Force Calculation Loop from CHARMM

of the problem, hence each atom interacts with hundreds of other atoms. This causes a large number of duplicate array references in the indirection array for the force calculation. In the input we used to test the kernel the total number of atoms (data array size) is 14,026 while the indirection array size is 6,700,818. There are 4 other irregular loops in the kernel, but the size of those indirection arrays are comparable to the data array sizes. We have used geometric data partitioners to partition the data associated with each atom. Even though the data is partitioned irregularly nearly all processors need to communicate with each other.

The molecular dynamics kernel and the Euler code kernel generate contrasting data access patterns, and stress different aspects of the communication primitives. We will use these two kernels in conjunction with the workload generator to evaluate the primitives.

3 Synthetic Workload

In this section we describe a parameterized synthetic workload generator that has been designed to simulate the kinds of data reference patterns and communication characteristics we have encountered in concurrent irregular scientific problems, examples of which has been presented in section 2.

The synthetic workload consists of two parts, the Communication Pattern Generator (CPG) and the Data Access Pattern Generator (DAPG). The CPG is used to define the communication pattern induced by the problem. The DAPG generates indirection arrays that embody the communication pattern specified by the CPG. Most of the optimizations presented in this paper are methods that can be used to preprocess indirection arrays. Consequently, we have developed a relatively sophisticated DAPG. We have a very simple CPG and we describe it first.

A communication graph $G = (V, E, w)$ is a weighted graph where vertices correspond to individual distributed partitions. For any two partitions $u, v \in V$ there exists an edge $(u, v) \in E$ iff the partitions u and v need to communicate with each other at run time. The volume of the communication is determined by the weight function $w : E \rightarrow N$. The generation of the communication graph is controlled by the following parameters :

- Connectivity (C)

  Connectivity is the degree of vertices in G. C = $(\sum_{v \in V} deg(v))/|V|$, where $deg(v)$ is the degree of vertex v. Each connection between two partitions causes communication to occur at run time between the partitions. The connectivity parameter is translated into the total number of distinct messages for each processor to send or receive at each phase of the computation.

- Total Volume of Communication (V)

  Total volume of communication limits the assignment of weights to the edges in a communication graph so that the sum of the weight of all the edges in a communication graph is equal to V. In the generation of the communication graphs for the experiments of this paper, V is used to distribute the weights to edges in a uniform way. That is, $\forall e \in E : w(e) = V/|E|$.

- Manhattan Distance (D)

  This parameter specifies that no two partitions that are more than D apart can be connected via an edge in the communication graph. The partitions in a real world problem are physically related to each other often with a relationship that is determined by the specific problem. These problem-imposed relationships can be represented as a graph, called a problem topology graph, in which vertices represent partitions and edges represent the connectivity relationships between partitions. It is usually the case that, in the problem domain, partitions are scattered in 2 or 3 dimensional space. The problem topology graphs, often representing the physical proximity of partitions, arise from the fact that distant partitions have little or no relationship with each other. The Manhattan Distance of two partitions is defined to be the sum of the canonical distances between them in the problem topology graph. For example, if the relationships between partitions are represented by a 2D-grid, the Manhattan Distance of u and v is $|u_x - v_x| + |u_y - v_y|$.

It should be noted, however, that more sophisticated communication models can be defined to replicate the communication behavior of irregular problems. One extension is the addition of extra features such as variability of the connectivity and communication volume. However, the current model is general enough to illustrate the key performance parameters of the optimization primitives.

The second part of the synthetic workload generator is the DAPG which is responsible for generating the data access patterns utilizing the communication graph. The actual communication takes place in a way that is determined by the communication graph. The data access pattern is defined to be a permutation of a subset of the global index space. It specifies which global data indices have been accessed locally. The output of the DAPG is a set of indirection arrays that will be used in accessing

```
do i = 1, n
  x(ia(i), 1) = x(ia(i), 1) + z(ia(i), 1)
  x(ia(i), 2) = x(ia(i), 2) + z(ia(i), 2)
end do
do i = 1, n
  x(ib(i), 1) = x(ib(i), 1) + z(ib(i), 1)
  x(ib(i), 2) = x(ib(i), 2) + z(ib(i), 2)
end do
```

Figure 5: Synthetic Workload Loops

the distributed arrays, whereas the input is the communication graph generated by the CPG and the following parameters :

- Number of Loops (N_{loop})

  The number of loops determines the number of consecutive test loops associated with the DAPG. The test loops are of the form depicted in Figure 5, where we have two consecutive computational phases, so , N_{loop} is equal to 2. For each computational phase the DAPG produces N_{loop} indirection arrays to access the loop's distributed arrays.

- Intersection Ratio (R_{int})

  The intersection ratio of two indirection arrays is defined to be the ratio between the number of identical global data indices they contain and the size of the indirection arrays. Its result is the degree to which data usage patterns in two indirection arrays are similar. If R_{int} is zero, then the global data indices stored in two different indirection arrays are completely disjoint; and two indirection arrays are exactly the same if R_{int} is 1.0.

- Number of Duplicates (N_{dup})

  The number of duplicates for a given indirection array is defined to be the number of distinct occurrences of the same off-processor data reference. If the number of duplicates for an indirection array is 2, then each unique reference in the indirection array will occur twice. Note that this parameter has no effect on the total volume of unique data communicated.

- Number of Dimensions(N_{dim})

  N_{dim} measures the degree of reuse of the same data access pattern across the dimensions of a distributed array.

Table 1: Summary of Symbols used in the paper.

Symbol	Meaning
C	Connectivity
V	Total Volume of Communication
D	Manhattan distance between partitions
R_{int}	Intersection Ratio
N_{dup}	Duplication Factor
N_{loop}	Number of test loops
N_{dim}	Number of identically referenced distributed array slices in each test loop
P	Total Number of Processors
R	Replication Factor of Paged translation table
B	Number of blocks (pages) in the translation table

An example of the type of work load generated is shown in Figure 5. For this case the various inputs to the DAPG are $R_{int} = 0.5$, $N_{dup} = 2$, $N_{loop} = 2$, and $N_{dim} = 2$. Since R_{int} equals 0.5, half of the values stored in array **ia** are also present in the array **ib**. Since N_{loop} equals 2, there are two indirection arrays namely, **ia** and **ib**. N_{dim} equals 2 makes the the upper bound of the compressed dimension of all the data arrays be 2 (in this case **x(*,2)** and **z(*,2)**). Since N_{dup} equals 2 each reference in **ia** is repeated twice. The same follows for **ib**.

The parameters defined for the workload and other symbols used throughout the rest of the paper are summarized in Table 1.

4 Communication Optimizations

In this section we present a set of communication optimizations. We have implemented these optimizations in a set of primitives that we call the PARTI runtime library. Section 4.1 shows how *software caching* can be used to reduce the volume of communication between processors. One optimization is to remove redundant off-processor accesses associated with a particular indirect array reference. A more aggressive optimization is to remove redundant off-processor accesses associated with several indirect array references. Section 4.2 describes the optimizations that have been developed to reduce communication startups by coalescing communications into a decreased number of messages.

4.1 Software Caching

During the execution of irregular loops on distributed memory (or distributed shared memory) machines, the same off-processor data may be accessed repeatedly. In many cases, we can prefetch all data needed by an array reference before a loop's computation begins. In other cases, we can prefetch all data needed by a set of irregular references to the same array. In either case, the same off-processor data may be accessed multiple times, but only a single copy of that data must be fetched from off-processor. In this paper, we do not address issues associated with identifying when these prefetches can be carried out; See von Hanxladen [11] and Das [9] for a detailed discussion.

Informally, the prefetches can be carried out when (1) it is possible to predict array reference patterns prior to a loop's execution, and (2) it is known that all array data subject to prefetch remains live, i.e. there is no possibility that the prefetched values are no longer valid.

4.1.1 Simple Software Caching

A hash table is utilized to identify duplicate off-processor data accesses associated with the indirect references to a single data element. Communication *schedules* are generated from the lists of unique off-processor data accesses. These schedules store the communication patterns to be used by the gather and scatter primitives. During the schedule generation process each processor sends the lists of data it needs from all other processors, it also receives the lists of data it must send to other processors. These lists contain the indices of the data that need to be communicated. Each schedule is associated with a distribution and a data access pattern, rather than being tied to specific data arrays. Hence, if we encounter two references to arrays where the arrays are distributed in the same way and where data access patterns are identical, we can use the same schedule to gather or scatter data to these arrays. The PARTI primitives described in earlier papers and outlined in Section 1.1 provide runtime support for simple software caching.

4.1.2 Incremental Scheduling

Data communication volume is reduced by tracking and reusing live off-processor data copies. In a number of application codes there occur multiple indirect references to the same data array. When it is known that no array assignments can occur between some sets of indirect references, i.e. the array in question remains live in that portion of the code, we only have to fetch a single copy of each unique off-processor value accessed by any of the indirect references.

Assume there are N different indirect array references to distributed data array D. From each reference we can obtain off-processor indices used to access data from the array D. Let IA_I be the set of off-processor indices from reference I. Hence, $IA = \{IA_1, IA_2, ..., IA_N\}$ is the set of sets of off-processor indices used to access data from D. The use of an incremental schedule allows us to bring in only those data that are not available locally:

$$\bigcup IA_I = \{\ ia : ia \in IA_I \text{ for some set } IA_I \in IA\ \}$$

The number of indices belonging to this set is potentially smaller than the number of indices we would get if we simply concatenated the indices obtained from separately applying simple software caching to each distributed array reference. If every index listed in each of the sets IA is different then there is no advantage in doing incremental scheduling. On the other hand, if there is significant overlap in the off-processor references obtained from the reference sets then a large reduction in communication volume is achieved by using incremental scheduling.

The incremental scheduling procedure removes duplicate off-processor data references by using hash tables. This procedure is invoked with a set of references and a set of *schedules* that have to be considered during the generation of the incremental schedule.

4.2 Communication Coalescing

One can frequently collect many data items destined for the same processor into a single message This kind of optimization is sometimes called *communication coalescing.* The object of communication coalescing is to reduce the number of message startups. For many distributed memory systems there is a substantial latency associated with message passing. For instance, Bokhari [2] measured the time to communicate a message of size k (bytes) between two nodes of an Intel iPSC/860 as:

$T = 65.0 + 0.425k + 10.0h$, for $0 < k \leq 100$, and

$T = 147.0 + 0.390k + 30.5h$, for $k > 100$

where T is the time in μsecs and h is the number of hops between the communicating processors. On this architecture, the cost of a startup latency is equal to the cost of sending one to several hundred bytes. We perform three types of communication coalescing :

- Simple communication aggregation
- Communication vectorization
- Schedule merging

4.2.1 Simple Communication Aggregation

It is frequently possible to anticipate which data must be communicated before a loop executes. We can carry out the preprocessing needed to characterize the data required by a given right hand side array reference. Prior to loop execution, we pack into a single message all the data each pair of processors need to exchange. In a similar manner, we can often defer until after a loop the communication (and accumulations) associated with left hand side array references. We refer to this optimization as *simple communication aggregation.*

4.2.2 Communication Vectorization

If a number of columns of a multi-dimensional array are distributed in a conforming manner and if the data access pattern from these columns are the same, then the primitives gather and scatter data from all the columns using a single communication phase. This does not reduce the communication volume but reduces the latency by the number of columns. Hence, if any processor P is to receive data from N processors for L columns then the reduction of latency time is given by,

- Latency Reduction = $N * Time_{latency} * (L - 1)$

The PARTI primitives for multi-dimensional arrays perform this optimization.

4.2.3 Schedule Merging

When data is gathered from, or scattered to, the same data array using a number of different schedules, then these schedules can be merged together to reduce the number of message startups and thereby the latency. Schedule merging is orthogonal to the software caching optimizations; for instance one can merge sets of schedules that arise from simple software caching or sets of schedules obtained from incremental scheduling. The total reduction in latency is by the factor $(S - 1)$, where the number of merged schedules is S. PARTI provides primitives that merge a number of schedules to form a single communication schedule.

4.3 Example Test Codes

The placement of the runtime support depends on the nature of the communication optimization. For instance, in Figures 6 and 7 we compare the test loops associated with simple communication aggregation and schedule merging. The simple communication aggregation case shown in Figure 6 does the preprocessing with the various indirection arrays right at the beginning. It returns four schedules, one for each of the indirection arrays. The **z** values are fetched immediately before each loop executes the schedule for **ic** is employed before the first loop and the schedule for **id** is employed before the second loop. After the execution of the first loop the off-processor **x** values are accumulated using the schedule for **ia**. Similarly, after the second loop computation, off-processor accumulation of **x** is done using the schedule for **ib**.

The schedule merging code is shown in Figure 7. As in the previous case, schedules are built using all the indirection arrays. In this case the schedules are merged and instead of four we have two, one for **ia** and **ib** and one for **ic** and **id**. All the required values of **z** are fetched using vectorized communication (**z** being a multi-dimensional array) before execution of the loops. We accumulate the off-processor values of **x** using the schedule for **ia** and **ib** after both loops execute. We can do

Preprocessing for indirection arrays **ia**, **ib**, **ic** *and* **id**
gather the values of array **z** *using schedule for* **ic**
do i = 1, n
  x(**ia(i)**) = x(**ia(i)**) + z(**ic(i)**)
end do
accumulate values of **x** *using schedule for* **ia**
gather the values of array **z** *using schedule for* **id**
do i = 1, n
  x(**ib(i)**) = x(**ib(i)**) + z(**id(i)**)
end do
accumulate values of **x** *using schedule for* **ib**

Figure 6: Simple Communication Aggregation Case

this because of the commutative property of the '+' operator. The executor communication cost when schedule merging and vectorization is performed is much lower than that of the simple software caching case. The inspector cost for schedule merging is higher than the inspector cost of the software caching case.

Note that while the software caching and communication coalescing optimizations are orthogonal, on distributed memory machines it makes sense to use certain optimizations together. For instance, if we employ incremental scheduling, we can easily produce a single merged schedule to perform the communication of the unique off-processor elements identified by the incremental scheduling process.

5 Paged Distributed Translation Table

In distributed memory machines, large data arrays need to be partitioned between the local processor memories. These partitioned data arrays are called *distributed arrays*. Long term storage of distributed array data is assigned to specific memory locations in the distributed memory machine. It is frequently advantageous to partition distributed arrays in an irregular manner. For instance, the way in which the nodes of an irregular computational mesh are numbered frequently does not have a useful correspondence to the connectivity pattern of the mesh. Figure 8 depicts an unstructured mesh. The node numbers are generated during the mesh generation process and they do not follow any regular pattern. In unstructured computational fluid dynamics solvers data is associated with each node of the mesh. When we partition the data structures in such a problem in a way that minimizes interprocessor communication, we may need to be able to assign arbitrary array elements to each processor. Once distributed arrays have been partitioned between processors, each processor ends up with a set of globally indexed distributed array elements that must be accessed.

Preprocessing to build a single schedule using arrays **ia** *and* **ib**
Preprocessing to build a single schedule using arrays **ic** *and* **id**
Gather for **z** *using the single schedule for arrays* **ic** *and* **id**

```
do i = 1, n
  x(ia(i), 1) = x(ia(i), 1) + z(ic(i), 1)
  x(ia(i), 2) = x(ia(i), 2) + z(ic(i), 2)
end do
do i = 1, n
  x(ib(i), 1) = x(ib(i), 1) + z(id(i), 1)
  x(ib(i), 2) = x(ib(i), 2) + z(id(i), 2)
end do
```

Accumulate **x** *using the single schedule for arrays* **ia** *and* **ib**

Figure 7: Schedule Merging Case

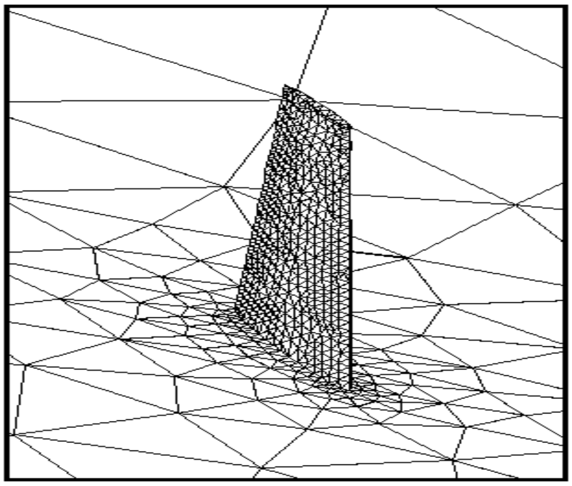

Figure 8: Unstructured Mesh

Each element in a size S distributed array, A, is assigned to a particular home processor. In order for another processor to be able to access a given element, $A(i)$, of the distributed array we must know the home processor where $A(i)$ resides, and we must know the local address of $A(i)$. We build a *translation table* which, for each array element, lists the home processor and the local address in the home processor's memory.

Memory considerations make it clear that it is not always feasible to place a copy of the translation table on each processor. A translation table can be distributed between processors. Earlier versions of PARTI supported a translation table that was partitioned between processors in a blocked fashion [8], [22]. This was accomplished by putting the first N/P elements on the first processor, the second N/P elements of the table on the second processor, etc ., where P is the number of processors. If we are

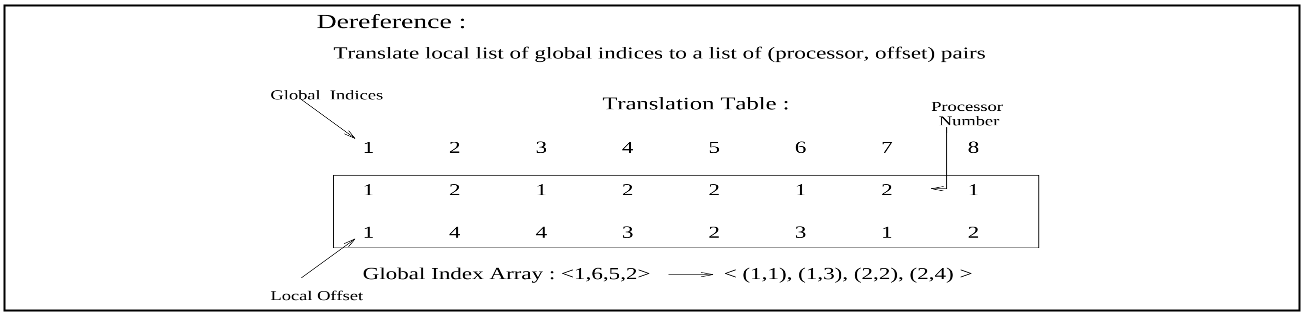

Figure 9: Global Index Tanslation

required to access an element $A(m)$ of distributed array A, we find the home processor and local offset in the portion of the distributed translation table stored in processor $((m-1)/N) * P + 1$. We will refer to a translation table lookup, aimed at discovering the home processor and the offset associated with a global distributed array index, as a *dereference request*.

In many cases, the naive translation table described above tends to be costly to use because (1) The distribution of the translation table between processors is fixed. The distribution of the translation table bears no particular relationship to the distribution of dereference requests; and (2) Some distributed array elements are included in a number of reference requests. In many cases there is enough memory to partially (or completely) replicate the translation table. The naive distributed translation table is not able to replicate portions of the translation table in order to trade memory for improved performance.

In this paper, we introduce a *paged* translation table. In this scheme the translation table is decomposed into fixed-sized pages. Each page lists the home processors and offsets associated with a set of B contiguously numbered distributed array indices. Each processor P stores Pa pages, and at least one processor maintains a copy of each page; consequently the total number of stored pages P*Pa must be greater than or equal to the distributed array size S divided by B. We follow the convention in the virtual memory literature and call the memory location associated with each page a *page frame*. Each processor maintains a page table; for each page, the page table lists a processor and a page frame.

Translation table information for each index must be stored somewhere; we make the simplifying assumption that each processor must store at least $S/(B*P)$ pages. In our current paged translation table implementation, we statically bind $S/(B*P)$ pages to each processor and then dynamically assign copies of additional pages to each processor. In the absence of any memory constraints, each processor could dynamically store $S*(P-1)/(B*P)$ pages; in this cases, the entire translation table would be replicated. The *replication factor* (RF) is defined as the fraction of the maximum number of pages for which frames are allocated by each processor. The user (or compiler) sets the page size B and a RF. Figure 9 shows the index translation process. Figure 10 depicts a highly simplified scenario in which we have 2 processors, an 8-element distributed array (S=8), a page size of 2 (B = 2) and a replication factor of 0.0 (RF=0.0). Since no pages are replicated, each processor has the same page table. In

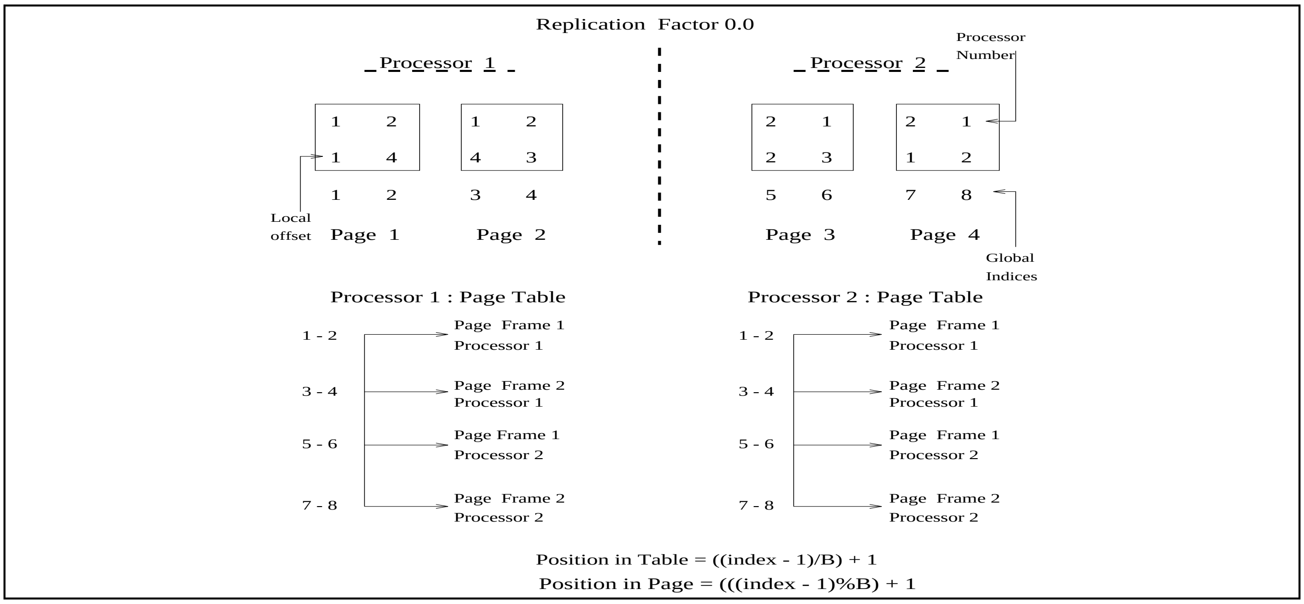

Figure 10: Paged Translation Table (Replication = 0.0)

Figure 11 we depict a scenario that is identical to the one shown in Figure 9, except that we now have a replication factor of 0.5. In this case, Processor 1 contains a dynamic copy of page 3, and processor 2 contains a dynamic copy of page 1.

Our runtime support allows each processor to choose which pages to replicate based on the characteristics of a user (or compiler) specified distributed array access pattern, specified by integer array **IA**. Each index i of **IA** is dereferenced by consulting page $\frac{IA(i)-1}{B} + 1$. On each processor, we choose the most heavily accessed pages as the dynamically assigned pages.

6 Optimization Scalability

Various models, metrics and analysis techniques for scalability of parallel programs have been proposed in the literature [20, 21, 13]. Under constant problem size scalability model, the application is run on varying sized systems without changing any of its parameters, including problem size. In the memory-constrained scalability model, the application parameters are scaled so that the memory requirements grow linearly with the increase in system size [10]. Finally, in time-constrained scaling, the application parameters are scaled so that the running time does not change while the system size changes. All of these models have some deficiencies and none of them are universal for all applications as pointed out by [19].

There are a wide range of optimizations designed to improve the performance of problems on multi-processor architectures. Any given optimization targets a certain class of problems and the effectiveness of the optimization varies with problem characteristics. We believe that it is also important to char-

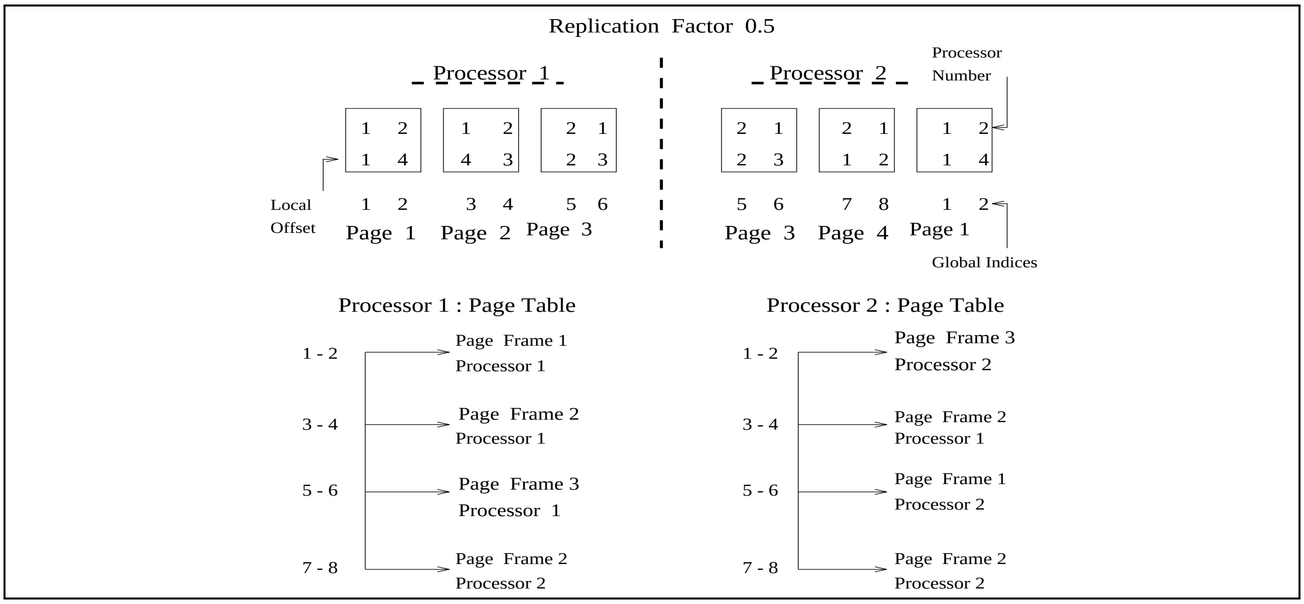

Figure 11: Paged Translation Table (Replication = 0.5)

acterize the relative effectiveness of an optimization on varying system sizes. For instance, consider a mechanism that is capable of exploiting parallelism that may exist between a heterogeneous set of atomic tasks (i.e. no task can be divided between processors). An optimization of this kind may be very effective when the number of processors used is less than or equal to the number of tasks. In such a case, increasing the number of processors greater than the number of tasks would show no gain in performance. As we shall see below, there are other cases in which the effectiveness of an optimization may remain constant or increase as one makes use of increasing numbers of processors.

We define *optimization scalability function* for a given problem and architecture as

$$osf(A, P) = \frac{T_{base}(s(A, P), P)}{T_{opt}(s(A, P), P)}$$

where $s(A, P)$ is the scaled version of problem A on P processors using any of the known scaling models, $T_{base}(s(A, P), P)$ is the time taken by the base code to solve $s(A, P)$ on P processors and $T_{opt}(s(A, P), P)$ is the time taken by the optimized code to solve $s(A, P)$ on P processors. It is important to note that optimization scalability can be viewed in the context of any problem scaling model. For instance, in computing an optimization scalability function, we can fix problem size and vary the number of processors. In this case $s(A, P) = A$. Alternately, we can increase the problem size or alter problem characteristics with increasing numbers of processors.

We examine the effect of our optimizations on unstructured problem communication requirements. In the analysis presented in this section we make use of the synthetic workload described in Section 3, which employs a set of loops of the type depicted in Figure 5. In the experimental analysis presented in the following sections we make use of both the synthetic workload and data access patterns derived from real applications.

Our optimizations are applied to a data access pattern generated when a given unstructured problem is mapped onto a multiprocessor. Measured performance on a given architecture consequently depends on (1) the nature of the unstructured code (e.g. the real codes outlined in Section 2 or the test loops in Figure 5), (2) the dataset (e.g. the data structures used to represent unstructured meshes and molecular interactions described in Section 2), and (3) the way in which the dataset is partitioned among processors.

We first examine the volume of communication and the number of communication startups associated with bringing in off-processor data. This data is presented in Table 2. In this table, we assume that problem size is fixed as we increase the number of processors. The row labeled "naive" stands for no optimization at all; each processor requests its data whenever that data is needed locally. In the "naive" case,the number of communication startups is equal to the number of data elements communicated. Using the notation defined in Section 3, we assume that the unoptimized or base problem requires communication volume V/P to be sent and received by each processor, N_{loop} represents the number of test loops employed by the DAPG and N_{dim} represents the number of identically referenced array slices. When targeted at distributed memory architectures, the naive implementation is extremely inefficient [18].

The row labeled "simple communication aggregation" gives the communication characteristics associated with the optimization described in Section 4.2.1. This optimization reduces the number of messages that must be transmitted. For each array slice (N_{dim}) and each test loop (N_{loop}) every processor must communicate with each of C neighboring processors. Note that this optimization does nothing to reduce communication volume. This optimization reduces latency costs compared to the naive implementation, but incurs two costs. The costs are the memory overhead of storing the schedules associated with the communication, and the pre-processing overhead for precomputing the communication requirements in the irregular computation.

The next optimization labeled "simple software caching" includes both simple software caching (Section 4.1.1) along with simple communication aggregation. Simple software caching involves eliminating intra-loop duplicates. The addition of this optimization reduces the communication time and space requirements compared to the simple communication aggregation case. The trade-off is the extra preprocessing required by the inspector and the memory required for the hash table. The communication volume for simple software caching is a factor of N_{dup} smaller than the communication volume for simple communication aggregation.

The next optimization labeled "communication vectorization" includes communication vectorization (Section 4.2.2) along with simple software caching and simple communication aggregation. The addition of communication vectorization optimization leaves the communication volume unchanged but reduces the number of startups by an additional factor of N_{dim}. The next row of the table, "schedule merging" adds the schedule merging optimization (Section 4.2.3) to the optimizations represented in the rows above. This optimization makes it possible to prefetch all data needed by the entire set of test loops before carrying out the first of the test loops. The number of startups in this case is reduced by a factor of N_{loop} and is equal to C.

Finally, we add the incremental scheduling optimization (Section 4.1.2) to the list above. Incremental scheduling allows us fetch from off-processor only the unique data values needed by any one of the test loops. This produces a savings when more than one test loop uses the same datum.

The left hand side array references in the test loops in Figure 5 involve accumulations. In most cases, our experience with applications scientists has indicated that it is permissible to defer off-processor accumulations until after a loop. This does have the effect of changing the order in which the accumulation is carried out. In the authors' experience, this change in operation order does not usually cause problems, since such loops are routinely vectorized, and vectorization also changes the order in which values are accumulated. If we limit ourselves to carrying out deferred accumulations after each loop, we find that schedule merging and incremental scheduling optimizations cannot be employed. In some applications areas, such as molecular dynamics, some applications programmers find that they can defer accumulations until after a sequence of (non-dependent) loops are carried out. In these cases one could make use of schedule merging and incremental scheduling optimizations.

Using the executor communication requirements presented in Table 2 we can show that the optimization scalability function for the executor is of the form

$$osf(A,P) = \frac{\frac{\alpha_1}{P} + \beta_1}{\frac{\alpha_2}{P} + \beta_2}$$

where, $\frac{\alpha_1}{P}$ is the communication volume of base code, $\frac{\alpha_2}{P}$ is the communication volume of optimized code, β_1 is the number of startups for the base code and β_2 is the number of startups for the optimized code. Recall that under the constant problem size scalability model $s(A,P) = A$. The base code in this case is taken to be the code with "simple communication aggregation" optimization (the "naive" code is so inefficient that in practice no one would employ this method!). When we take the derivative of the *osf* with respect to the number of processors P we get,

$$osf\prime(A,P) = \frac{\beta_1 * (\alpha_1 + \beta_1 * P)}{\alpha_2 + \beta_2 * P} - \frac{\beta_2 * (\alpha_1 + \beta_1 * P)}{(\alpha_2 + \beta_2 * P)^2}$$

The function $osf\prime(A,P) > 0$ because $\alpha_1 \geq \alpha_2$ and $\beta_1 \geq \beta_2$. Hence we conclude that the optimization scalability function for the executor primitive is monotonically increasing with the increase in the number of processors. We can show similar characteristics for the optimization scalability function of the inspector primitives.

Table 2: Executor Communication Requirements (Gather, Scatter or Accumulate)

Optimization	*Communication*	
	Volume	Number Startups
Naive	$\frac{N_{dim} N_{loop} V}{P}$	$\frac{N_{dim} N_{loop} V}{P}$
Simple communication aggregation	$\frac{N_{dim} N_{loop} V}{P}$	$N_{dim} N_{loop} C$
Simple software caching	$\frac{N_{dim} N_{loop} V}{P N_{dup}}$	$N_{dim} N_{loop} C$
Communication vectorization	$\frac{N_{dim} N_{loop} V}{P N_{dup}}$	$N_{loop} C$
Schedule merging	$\frac{N_{dim} N_{loop} V}{P N_{dup}}$	C
Incremental scheduling	$\frac{N_{dim} V}{P N_{dup} (2 - R_{int})^{N_{loop} - 1}}$	C

In CHARMM, one can increase problem size by varying the size of a molecule being simulated or by varying the cutoff radius (Section 2.2). Increasing the cutoff radius has the effect of increasing N_{dup}. As is clear in Table 2, the effectiveness of optimizations that include software caching improve with increasing N_{dup}.

The communication requirements associated with preprocessing are very closely tied to the communication requirements needed to carry out off-processor gathers, scatters and accumulations. Table 3 depicts these communication requirements. We gain some advantage from the fact that we can reuse the same schedule each time we carry out communications in which we encounter identically referenced, identically distributed arrays (or array sections). From Table 3, we see that the inspector costs do not scale with the number of processors in the constant size problem scaling model. Consequently, when we include the inspector overhead, the optimization scalability function decreases with the number of processors.

We have developed a substantial body of compile time analysis dedicated to identifying such situations [11], [9]. In the case of the test loops, it is clear that we do not have to repeat our preprocessing for identically distributed array sections. This advantage is reflected in the communication volume and startup numbers depicted in Table 3.

7 Experiments and Results

This section describes the experiments performed and the corresponding results. A number of different experiments were performed using the synthetic workload generator and the application code kernels.

Table 3: Inspector Communication Requirements (Gather, Scatter or Accumulate)

Optimization	*Communication*	
	Volume	Number Startups
Naive	-	-
Simple communication aggregation	$\frac{N_{loop}V}{P+N_{loop}C}$	$2N_{loop}C$
Simple Software caching	$\frac{N_{loop}V}{PN_{dup}+N_{loop}C}$	$2N_{loop}C$
Communication vectorization	$\frac{N_{loop}V}{PN_{dup}+N_{loop}C}$	$2N_{loop}C$
Schedule merging	$\frac{N_{loop}V}{PN_{dup}+N_{loop}C}$	$2N_{loop}C$
Incremental scheduling	$\frac{V}{PN_{dup}(2-R_{int})^{N_{loop}-1}+C}$	$2C$

We present the results to show the performance of the primitives and also how they scale as we go from a small to a large number of processors.

7.1 Scalability of the Primitives

The communication optimizations presented in section 6 have been tested in various synthetic and real computational problems. In this section, we characterize the scalability of problems implemented using our runtime support. We also carry out experiments that make it possible to characterize a few of the optimization scalability functions associated with our optimizations.

Experiments based on the memory constrained scaling model were used to demonstrate the scalability of the optimizations. In our experiment we increased the size of the problem together with the size of the machine. A useful property of our workload generator is that it can be used to produce communication patterns whose communication structure is preserved when the number of processors is increased. If we scale the synthetic workload in the above manner we did not observe any significant changes in performance with increasing numbers of processors.

Table 4 presents the absolute timings for schedule merged incremental gathers using the communication pattern for the Manhattan distance, $D = 1$, connectivity $C = 4$. The total communication volume V is scaled up in proportion to the number of processors employed, this maintains a volume per processor of $\approx 2K$ floating point numbers. We are able to obtain a roughly fixed communication cost as the problem size grows linearly with the number of processors. We have observed this pattern for a number of different communication patterns.

Table 4: Results Supporting Scalability (Time in secs.)

Intersection ratio	32 processors		64 processors		128 processors	
	Inspector	Executor	Inspector	Executor	Inspector	Executor
0.0	0.8	6.2	0.9	6.2	0.9	6.1
0.2	0.8	5.9	0.9	5.7	0.9	5.7
0.4	0.8	5.5	0.8	5.5	0.9	5.5

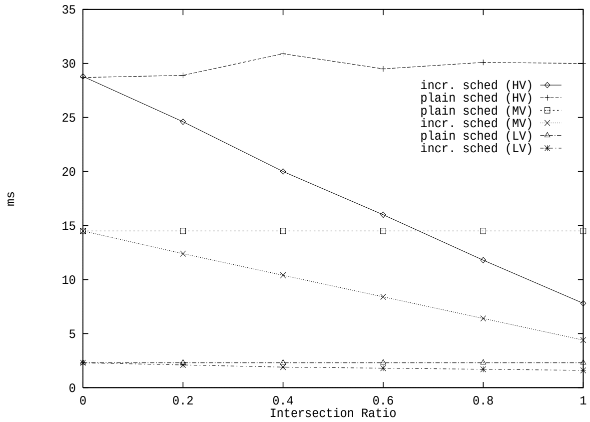

Figure 12: Cost vs. volume of communication and Intersection Ratio (gather, N_{loop}=4, P = 32)

7.2 Synthetic Workload Performance Results

We now present empirical performance results to characterize the effectiveness of the communications optimizations presented in Section 4.

The reduction in communication time associated with incremental schedules is shown in Figure 12. We compare performance of a code which employs schedule merging with incremental scheduling versus simple software caching carried out separately for each loop. We used four loops in our test loop code ($N_{loop} = 4$). We keep the communication graph constant (C=4) and vary the R_{int} parameter in order to vary the number of shared off-processor accesses. The loop structure is similar to the one presented in Figure 7. We repeated the experiment for low ($\approx$ 100 floating point numbers), medium ($\approx$ 1000 floating point numbers), and high ($\approx$ 2000 floating point numbers) communication volume. Figure 12 gives the timings for the *gather* calls both for incremental scheduling and simple software caching. For both high and medium communication volumes, communication time for the incremental case drops rapidly as the intersection ratio becomes close to 1. The inspector times are given in Figure 13. The inspector time for the incremental scheduling is usually higher than with simple software caching because of the larger volume of data that has to be hashed.

We now quantify the performance effects of simple software caching. We keep the communication

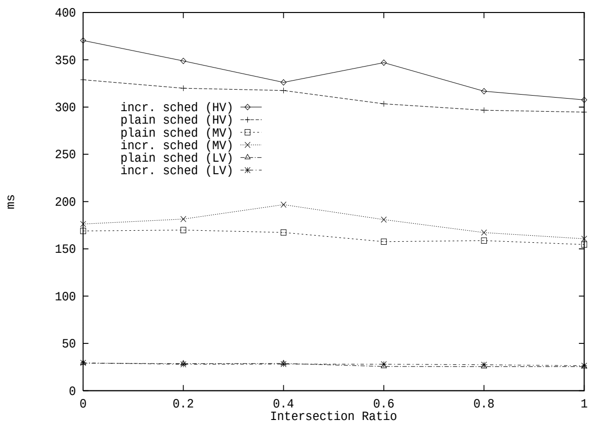

Figure 13: Cost vs. volume of communication and Intersection ratio (Inspector, N_{loop}=4, P = 32)

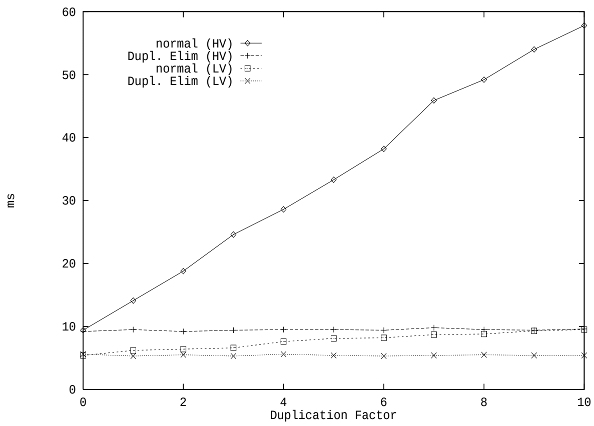

Figure 14: Cost vs Number of duplicates (Executor, low N_{dup}, 32 Processors)

graph constant and vary N_{dup} and the volume of communication. The structure of the test loop associated with the duplicate elimination version and the pre-scheduled communication version are very similar to the one in Figure 6. Figure 14 shows the results when the duplication factor N_{dup} is low (range 0 to 10). This is the case usually found in unstructured mesh computations computational fluid dynamic calculations, in these calculations; the connectivity of the mesh ranges from 6 to 10. Figure 15 shows the case where the duplication factor is very high (range 0 to 500). This case is similar to the data access pattern found in molecular dynamics and particle dynamics codes, where each particle interacts with a large number of other particles (usually within a cut-off radius). We find the expected result: performance improvement associated with software caching increases with the duplication factor, except when the communication volume and duplication factor are both low.

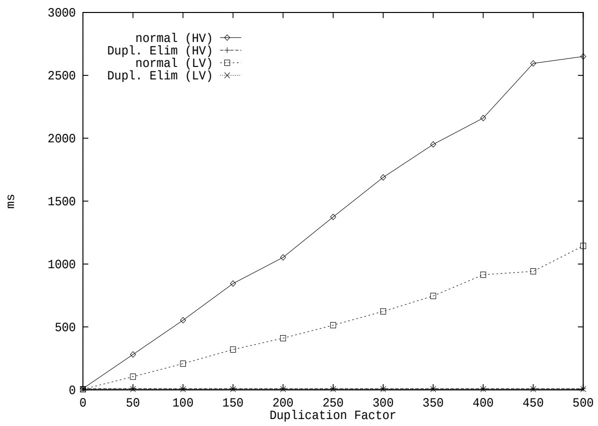

Figure 15: Cost vs Number of duplicates (Executor, high N_{dup}, 32 Processors)

7.3 Performance Results Derived from Applications

7.3.1 Comparison of Communication Optimizations

The Euler kernel was timed varying the number of processors from 16 to 128. All timings presented are for 10 iterations of the outermost loop. The communication times for the different levels of optimizations are shown in Table 5. We see that for both the 53K and 100K mesh input, schedule merging together with vectorization makes the communication time decrease slightly as we go to a larger number of processors. Similarly, the total running time, presented in Table 8, also goes down significantly as more processors are used. It is also of interest to note that when we view simple software caching as our base case, our optimizations taken together exhibit an increasing optimization scalability function (as predicted by the discussion in Section 6. For the 53K mesh, the ratio between the last and first row of Table 5 increases monotonically from 1.4 for the 16 processor case to 2.9 for the 128 processor case. For the 100K mesh, the ratio between the first and last row of Table 5 increases from 1.4 to 1.9 in a monotonic fashion. This tells us that for a fixed sized problem, our optimizations become increasingly effective as one increases the number of processors.

7.3.2 Behavior of Paged Translation Table

We performed several experiments to measure the performance of the paged translation table. Table 6 shows the effects of replication factor on the scheduling time for a 53K node unstructured mesh, and a benchmark input for CHARMM (MbCO + 3,830 water molecules; 14,026 atoms) on a 64-processor iPSC/860. The column labeled "Before" corresponds to performance with the initial block distribution of pages across processors. The column labeled "After" corresponds to the performance after a re-organization of replicated pages according to access behavior on each processor. In this experiment,

Table 5: Euler kernel, 53k&100kMesh (Time in secs)

Optimization	Total Communication (Executor)						
	53K Mesh				100K Mesh		
	16	32	64	128	32	64	128
Simple Software Caching	22.4	22.7	29.1	37.3	29.2	29.9	34.7
Schedule Merging (SM)	19.1	20.1	24.7	28.5	25.0	25.1	26.4
Vectorized (Vect) + SM	15.9	15.7	13.1	12.8	20.7	19.3	18.1
Incremental + SM	18.9	20.2	24.3	27.9	24.3	25.1	26.7
Incremental + SM + Vect	16.1	15.7	12.9	12.7	21.2	19.1	18.0

Table 6: Effects of Replication Factor (Time in secs.)

Replication	Euler kernel (53k)		CHARMM kernel	
	Before	After	Before	After
0.0	0.4	0.4	7.9	7.2
0.1	0.4	0.3	6.8	5.1
0.2	0.4	0.3	6.0	3.2
0.3	0.4	0.3	5.7	2.0
0.4	0.4	0.2	5.3	1.4
0.5	0.3	0.2	5.0	1.1
0.6	0.3	0.2	4.3	1.0
0.7	0.2	0.2	4.0	0.9
0.8	0.2	0.1	2.9	0.9
0.9	0.2	0.1	2.2	0.9
1.0	0.1	0.1	0.9	0.9

we vary the number of pages replicated on each processor. As we expected, performance improves as the replication factor increases. For the unstructured mesh, reshuffling of translation table pages does not make much difference in scheduling time. For the molecular dynamics case the reshuffling makes a large difference, especially for low replication factors.

Table 7 shows the performance of dereferences with varying block sizes for a fixed replication factor, $R = 0.5$. We have observed that reasonable communication times can be obtained with relatively large page sizes. When the page size is decreased, the communication efficiency of the fully replicated case can be achieved without having to replicate all the data associated with the translation table.

7.3.3 Performance of Optimizations on Large Scale Application

The PARTI primitives were also used to port CHARMM (120K lines) to the iPSC/860. The entire energy calculation portion of CHARMM has been parallelized. This involves both internal and external energy calculations. The internal energy calculation is not very expensive and accounts for 1% of the total energy calculation time. It calculates the bonded interactions of atoms. The external energy

Table 7: Effects of Page Size, $R = 0.5$, (time in secs)

Euler kernel (53k)			CHARMM kernel		
Page Size	Before	After	Block Size	Before	After
85	0.3	0.2	89	5.0	1.2
43	0.3	0.2	44	5.0	1.2
29	0.3	0.2	22	5.0	1.1
22	0.3	0.2	15	5.0	1.1
17	0.3	0.2	11	5.0	1.1
9	0.3	0.1	6	5.4	1.0
5	0.4	0.1	5	5.4	1.0
3	0.4	0.1	3	5.3	1.1

Table 8: Euler kernel, 53k&100kMesh (Time in secs)

Optimization	Total Running Time						
	53K Mesh				100K Mesh		
	16	32	64	128	32	64	128
Simple Software Caching	104.3	63.9	50.0	48.9	108.5	67.4	52.6
Schedule Merging (SM)	100.3	60.5	46.8	39.3	104.7	62.3	45.4
Vectorized (Vect) + SM	97.5	57.3	34.8	24.1	99.7	57.1	37.2
Incremental + SM	100.6	60.7	46.3	38.7	103.6	61.9	44.6
Incremental + SM + Vect	97.1	57.9	34.5	23.8	100.3	56.8	36.7

calculation is the most computationally intensive part of the code and accounts for more than 90% of the total energy calculation time. It first generates the nonbond list and then calculates the nonbonded interactions of atoms. But the nonbond list is not updated every iteration. It is updated every **n** iterations where **n** is a variable that can be fixed by the user. Currently, **n** is set to 25.

Atoms are partitioned based on geometric positions to reduce communication time by using recursive coordinate bisection. Since the execution time of external energy calculation depends on the size of the nonbond list, we must partition the nonbond list as balanced as possible. If we simply consider geometrical positions of atoms and assign equal numbers of atoms to processors, it is very likely that the load will be not balanced (see Table 9). The index of load balance is calculated by

$$\frac{(\max_{i=1}^{n} \mathit{execution\ time\ of\ processor\ i}) \times (\mathit{number\ of\ processor\ n})}{\sum_{i=1}^{n} \mathit{execution\ time\ of\ processor\ i}}$$

In order to keep the load balanced, i.e. index close to 1, we generate the nonbond list for every atom and use the size of nonbond list as the weight (or load) of each atom. Then we use the weighted recursive coordinate bisection to partition atoms by considering both the positions and loads.

During the internal and external energy calculations, the iteration partitioning is done based on the atom partitioning. The iterations of internal energy calculations are assigned to the processors with the

Table 9: Real Charmm Data (MbCO + 3830 water molecules)[1]

	Unweighted Coordinate Bisection				Weighted Coordinate Bisection			
Processors	16	32	64	128	16	32	64	128
Preprocessing:								
Partition[2]	0.75	0.94	1.40	2.04	10.12	5.94	4.25	3.75
Schedules[3]	1.27	0.92	0.63	0.60	1.15	0.71	0.41	0.41
Communication	14.19	15.77	17.10	21.47	14.77	16.67	19.43	24.15
Execution Time[4]	538.75	350.53	226.95	136.63	485.63	259.38	134.58	82.87
Load Balance[5]	1.14	1.43	1.79	1.75	1.02	1.03	1.04	1.05

[1] Intel iPSC/860 timings in seconds, 100 steps, 4 nonbond list updates, 12 Å energy cutoff, 14 Å list cutoff
[2] Partition and remap atoms
[3] Generate schedules for gather and scatter functions
[4] Execution Time = (Internal and External Energy Calculation Time) + Communication Time + Nonbond List Update Time
[5] Load Balance = $\frac{(\max_{i=1}^{n} \textit{execution time of processor } i) \times (\textit{number of processor } n)}{\sum_{i=1}^{n} \textit{execution time of processor } i}$

Table 10: Weighted and Unweighted Binary Dissection v.s. Data Replication

Processors	Unweighted (sec/iteration)	Weighted (sec/iteration)	Replicated[1] (sec/iteration)
16	5.38	4.86	4.95
32	3.51	2.59	2.65
64	2.27	1.35	1.53
128	1.37	0.83	0.94

[1] Results from [4]

largest number of atoms for the calculations. For the external (nonbonded) energy calculation, every processor generates the nonbond list for the atoms that have been assigned to the processor. And then the Van der Waals and electrostatic force calculations are performed.

We ran a benchmark case (MbCO + 3830 water molecules) and present the timing results in Table 9 and Table 10. We present these results to make clear that we have found that our runtime support and optimizations can be, and are, being used to port challenging application codes. These results are comparable to all other implementations of which we are aware [4]. We should also note that the optimization scalability function for the weighted coordinate bisection load balancing optimization function takes on values of 1.11, 1.35, 1.69 and 1.65 for 16,32,64 and 128 processor respectively. As one would expect, a better load balancing procedure becomes increasingly important as the number of processors increases.

8 Conclusion

This paper has described and systematically evaluated a number of new communication optimizations aimed at irregular problems. These optimizations reduce communication latency and volume. We also describe a paged translation table to efficiently support access to globally indexed distributed arrays that are mapped onto processors in an irregular manner. This paged translation table can be configured to reduce the time required to determine processor assignments for irregularly distributed data. The optimizations that are described in this paper are part of the PARTI primitives. These primitives can be used by parallelizing compilers to generate parallel code from programs written in data parallel languages like HPF, Fortran D etc. The choice of the optimization is done by the compiler after indepth analysis of the sequential code.

We presented a framework for characterizing the performance and scalability properties of our optimizations. We introduced the concept of optimization scalability function to characterize the relative effectiveness of an optimization as one employs varying numbers of processors. Optimization scalability can be viewed in the context of any problem scaling model.

Performance was measured using a family of test loops and an associated synthetic workload generator, using the test loops with data access patterns drawn from unstructured applications, and through the use of templates from real applications.

Acknowledgements

We wish to thank Alan Sussman for many valuable suggestions, and for carefully editing this document. We also would like to thank NASA Ames Laboratory and National Institute of Health for providing access to their 128-node iPSC/860.

References

[1] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[2] S. Bokhari. Communication overhead on the intel ipsc-860 hypercube. Report 90-10, ICASE Interim Report, 1990.

[3] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[4] B. R. Brooks and M. Hodoscek. Parallelization of charmm for mimd machines. *Chemical Design Automation News*, 7:16, 1992.

[5] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. In *To appear AIAA Journal,AIAA-92-0562*, Jan 1992.

[6] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, 1992. Elsevier.

[7] R. Das and J. Saltz. Parallelizing molecular dynamics codes using parti software primitives. In *Parallel Processing for Scientific Computation, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk VA, March 1993*, 1993.

[8] R. Das, J. Saltz, D. Mavriplis, and R. Ponnusamy. The incremental scheduler. In *Unstructured Scientific Computation on Scalable Multiprocessors*, Cambridge Mass, 1992. MIT Press.

[9] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[10] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal of Scientific and Statistical Computing*, 9(4):609–638, July 1988.

[11] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[12] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12:415–422, August 1991.

[13] V. Kumar and A. Gupta. Analysis of scalability of parallel algorithms and architectures : A survey. In *Proc. of 1991 International Conference on Parallel Processing*, June 1991.

[14] D. J. Mavriplis. Three dimensional multigrid for the Euler equations. *AIAA paper 91-1549CP*, pages 824–831, June 1991.

[15] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. Technical Report CS-TR-3055 and UMIACS-TR-93-32, University of Maryland, Department of Computer Science and UMIACS, April 1993. To Appear Supercomputing '93.

[16] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11:430–452, 1990.

[17] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[18] J. Saltz, K. Crowley, R. Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[19] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, pages 42–50, July 1993.

[20] X. Sun and D. T. Rover. Scalability of parallel algorithm-machine combinations. Report IS-5057, April 1991.

[21] Patrick H. Worley. The effect of time constraints on scaled speedup. *SIAM Journal of Scientific and Statistical Computing*, 11(5):838–858, September 1990.

[22] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 26–30, 1991.