Compiler and run-time support for semi-structured applications

Nikos Chrisochoides, Induprakas Kodukula, and Keshav Pingali
Department of Computer Science,
Cornell University, Ithaca, NY 14853.
EMail: {nikosc,prakas,pingali}@cs.cornell.edu

Abstract

Adaptive mesh refinement (AMR) is a very important scientific application. Several libraries implementing specific distribution policies have been written for AMR. In this paper, we present a "fully general block distribution" which subsumes these distributions, and discuss compiler and run-time tools for supporting these distributions efficiently in the context of a restructuring compiler. We also present performance numbers which suggest that in comparison with library code written for a particular distribution policy, the overhead arising from the generality of our approach is small.

1 Introduction

Semi-structured methods such as adaptive mesh refinement and multigrid are used in applications which are computationally intensive. It is difficult to implement these methods efficiently even on a sequential machine; parallelism adds an order of magnitude overhead to the complexity.

The computation in semi-structured methods is characterized by irregularly organized regular computations on the underlying data. The underlying data is specified as *grid components* which are organized in an irregular fashion in a *grid hierarchy* which itself changes dynamically. The computation in the application consists of stencil operations for relaxation on grid components, and interpolation and projection operations for transferring data between grid components at different levels of the grid hierarchy. The computation is regular (that is, data access functions are affine functions of surrounding loop indices). Parallelism is obtained by distributing the grid components in the hierarchy among various processors. However, this also leads to communication between various grid components. The volume and patterns of communication depend critically on the data distribution. The irregular and evolving nature of the application also leads to load imbalance, requiring redistribution of data at run time.

Even though the underlying computation is dense, HPF-like compiler technology is inadequate for AMR applications. There are two reasons for this:

- Block-cyclic distributions are inadequate to obtain proper load balance for these applications. Obtaining good load balance requires grid components to be distributed in complex ways among the processors, as explained below.
- The communication between various grid components is determined by the connectivity of the grid hierarchy. Due to the irregular nature of this connectivity, communication patterns are irregular.

Because of the inadequacy of present compiler technology, several libraries have been developed to make the application programmer's job easier. The most important of these are Multi-block PARTI [1], P++/AMR++ [3], LPARX [2], DAGH [14]. All these libraries hide the nature of the data distribution from the application programmer as much as possible. They provide constructs such as *forall* loops to enable application writing at a high level, and provide library calls to take care of parallelization issues. Each of these libraries makes certain assumptions regarding how the underlying data is distributed, which affects application performance. *Multi-block PARTI* [1] uses block-cyclic distributions as in HPF, but allows data arrays to be mapped to a subspace of all the computing processors. This works well for Multigrid codes, but for adaptive mesh refinement, the limitations of HPF apply here as well. *AMR++* [3] is an AMR class library layered on top of a parallel array library P++. AMR++ treats P++ as a black box and uses the distributions that P++ provides. These distributions reduce load-imbalance by allowing arrays to be distributed by columns of variable size, but are still limited in expressiveness. *LPARX* [2] allows data arrays to be distributed in irregularly shaped and irregularly sized blocks onto processors. While in principle it is possible in LPARX for multiple blocks of data to be assigned to one processor, in practice there is only one data block per processor. For adaptive mesh refinement, LPARX helps in reducing the communication between grid components at the same level, but the communication between grid components at different levels increases. Finally, *DAGH* [14] uses a space-filling curve enumeration to distribute the blocks of an array onto processors. Space filling curves ensure spatial locality, which means that this distribution policy reduces communication between grid components at different levels of the grid hierarchy.

From an examination of all the above libraries, it is clear that there is no single universal distribution policy that is superior to all others. This makes it hard for the application programmer to experiment with different distribution policies. The libraries also need to be extended when they do not satisfy the users' needs. For example, DAGH comes with predefined stencils for relaxation schemes which cover many, but not all relaxation methods. Successful compiler support here would be of benefit to the application writers as well as library writers.

To this end, there have been some generalizations of the distributions allowed in HPF, the most general of which is the generalized block distribution in HPF-2 [8]. The generalized block distribution for an array is defined by a cartesian product of intervals, where each interval partitions one dimension of the array. This generalization captures the distributions in libraries such as Multi-block PARTI and AMR++, but it is inadequate to express distributions in libraries such as LPARX and DAGH.

We propose a distribution called the *fully general block distribution* to capture all the distributions used in the libraries mentioned above. We extend current compiler techniques to solve the standard parallelization problems encountered in generating parallel code for such distributions. A genuine concern is the potential cost of handling a general data distribution mechanism, and we present preliminary performance numbers suggesting that this overhead can be minimal with proper choice of algorithms. We describe data structures from computational geometry that enable us to manipulate information about these distributions efficiently. We also discuss suitable modifications to these techniques that allow us to handle the case of dynamic data distributions efficiently.

The rest of the paper is organized in two parts. Section 2 describes the new data distribution. Section 3 discusses how parallel code can be generated if the parameters of the data distribution are known completely at compile time. Section 4 describes how these techniques are modified when data distributions change dynamically. Section 5 describes incremental algorithms for computing communication sets. We present some performance results in Section 6 and summarize in Section 7.

2 Fully general block distributions

Intuitively, a fully general block distribution partitions the elements of an array into rectangular blocks whose sides are parallel to the data coordinate axes. Different blocks can have different sizes. More formally, we have the following definition.

A *block* in d-dimensions is a rectangular parallelopiped in d-dimensions, and can be specified by two vectors: $\vec{o} = [o_1, \ldots, o_d]$ and $\vec{e} = [e_1, \ldots, e_d]$, where $\vec{o}$ represents one of the vertices of the parallelopiped, and $\vec{e}$ represents the "extent" of the parallelopiped in each of the d-dimensions. By requiring that all the entries of $\vec{e}$ be non-negative, we end up with a canonical representation of a block. A *fully general block distribution* for a data item D is specified as a set of tuples of the form (B_i, P_i), where B_i is a block as defined above, and P_i is a processor assignment of this block. In addition, it must be the case that every element of D must occur in precisely one block, and any element of a block B_i must be a valid element of D (*i.e.*, it must be within array bounds of D).

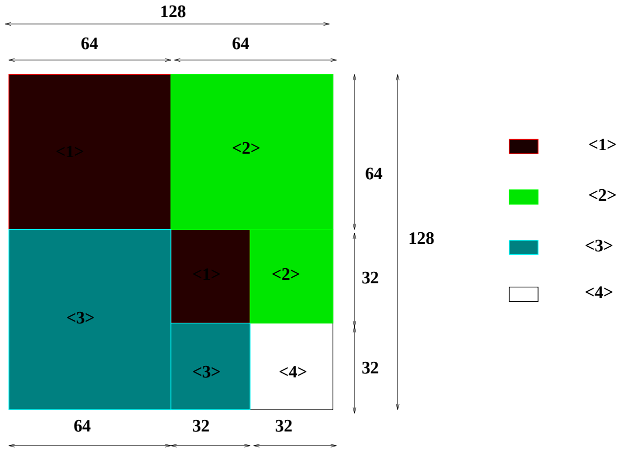


Figure 1: Example of a valid fully general block distribution

Figure 1 shows an instance of a *fully general block distribution*. The general block distributions described in the literature [7, 8, 9, 17] are special cases of this distributions.

2.1 Distribution descriptor

We introduce the notion of a *distribution descriptor* that is associated with each distributed data item. This descriptor is a data structure which provides information about the distribution of the associated data item in a structured form. In particular, the following information is provided to each processor.

- Number of blocks of the data item on a particular processor
- For each block, the global indices of the data object contained in the block, specified by the origin ($\vec{o}$) and extent ($\vec{e}$) vectors.

We assume that this information is replicated; *i.e.*, that every processor has access to the complete distribution information of the data item under consideration.

A particular representation for the distribution descriptor is as an ordered set of sets each describing the local allocation of each processor. For example, the distribution descriptor for the distribution in Figure 1 consists of the four sets $\{([1,1],[64,64]), ([65,65],[32,32])\}$, $\{([1,65],[64,64]), ([65,97],[32,32])\}$, $\{([65,1],[64,64]), ([97,65],[32,32])\}$ and $\{([97,97],[32,32])\}$ for processors 1, 2, 3 and 4 respectively.

A fully general block distribution is clearly more complex to specify than a block or cyclic distribution in HPF. However, it appears that in most AMR applications, the distributions of data arrays at the start of

Relaxation code

```
{
    Atp1 and A are N-by-N matrices
    do I = 2, 127
      do J = 2, 127
        S1: Atp1(I,J) = c* (A(I,J+1)+A(I,J-1)+
                 A(I-1,J)+A(I+1,J)-4.0*A(I,J))
}
```

Figure 2: Relaxation code in 2-d wave equation

program execution are usually very simple, but become complex during program execution. Therefore, the fully general block distribution needs to be supported by the compiler and runtime system, but it may not be necessary to have directives in the source language to specify such distributions.

For the rest of the paper, we use the relaxation code in Figure 2 as our running example, with the data distribution for `A` and `Atp1` as specified in Figure 1. This is a simplified version of the actual relaxation code that is used in the solution to the 2-D wave equation using the DAGH library [14, 6], and illustrates all the principles behind our approach.

2.2 Issues for a Parallelizing Compiler

Given a program with data distribution specifications, a restructuring compiler performs the following tasks:

- Assignment of computations to processors.
- Generation of code to enumerate local iteration sets.
- Local storage allocation for distributed arrays.
- Determination of communication sets.
- Placement of communication in the program.

For dense programs, block and cyclic distributions (as in HPF [7]) are standard, and a simple rule like the *owner-computes* rule [15] is used to determine the iterations to be performed on each processor. In this case, closed form linear integer constraints can be used to express the local storage requirements, the local iteration sets, communication sets as well as the placement of communication. While the efficient placement of communication is a major concern for programs with dependences, it is straightforward in the case of *do-all* loops, since all communication for the loop nest can be performed before the loop nest begins execution. We will not worry about communication optimizations such as combining communication from different loop nests to reduce the volume of communication.

In the rest of this paper, we solve the first four problems listed above for the case of fully general block distributions.

3 Parallelization in the static case

In this section, we examine the static case (*i.e.*, when the parameters of the distribution are known at compile time), which is relevant in codes like multi-block Euler solvers [5]. A more general treatment of some of the material of this section can be found in [11].

Code executed by processor 4

```
{
    Atp1 and A are N-by-N matrices
    do I = 2, 127
      do J = 2, 127
        If I'm data-centric w.r.t. A(I,J+1)
        if ((97 ≤ I ≤ 128) && (97 ≤ J+1 ≤ 128))
          S1: Atp1(I,J) = c* (A(I,J+1)+A(I,J-1)+
                   A(I-1,J)+A(I+1,J) - 4.0*A(I,J))
        endif
}
```

Figure 3: Naive code to be executed by processor 4 after being data-centric w.r.t. `A(I,J+1)`.

Simplified code executed by processor 4

```
{
    do I = 97, 127
      do J = 96, 127
        S1: Atp1(I,J) = c* (A(I,J+1)+A(I,J-1)+
                 A(I-1,J)+A(I+1,J) - 4.0*A(I,J))
}
```

Figure 4: Simplified code to be executed by processor 4

3.1 Local iteration Set

For any loop nest, every processor must determine the set of iterations it will execute. This requires the determination of the *computation decomposition*. In our framework, we drive this using the *data-centric* approach. The data-centric approach requires choosing a single reference of a distributed data item from a loop nest and requiring all accesses caused by that reference to be local. In other words, only those iterations of the loop nest are executed for which the chosen reference causes data accesses to be local to the processor. We illustrate this with an example. We can drive the data-centric approach using the reference `A(I,J+1)` in our running example. This causes processor 4 (which owns the block(97..128, 97..128) of `A`) to execute the code in Figure 3. Note that this code is still written in a *shared-memory* style, because the indices used to access the elements of all the arrays are global indices. This code is similar to that produced by *run-time resolution* [15] for distributed memory multiprocessors. We refer to such conditionals introduced to limit a specific access to a specific block as *localization constraints*.

Since all *localization constraints* are simple affine constraints on surrounding loop indices and constants representing block bounds, they can be simplified and folded into loop bounds using any polyhedral algebra tool. Doing this produces code shown in Figure 4.

We note that it is possible for a processor to contain multiple blocks of the data item used to partition computational work (in our example, processor 1 is an instance of this). Since the distribution descriptor is available at compile time, we can take one of two approaches: (i) generate a different loop nest for each local block, or (ii) generate an outer loop that enumerates local blocks, with inner loops generating the iteration set for each block. The code generated for processor 1 using approach (i) is shown in Figure 5. Note that this approach cannot be used if the descriptor is not known to the compiler. Figure 6 shows the code generated for

Simplified code executed by processor 1

```
{
do I = 1, 64
   do J = 1, 63
      S1: Atp1(I,J) = c*(A(I,J+1)+A(I,J-1)+
                A(I-1,J)+A(I+1,J)-4.0*A(I,J))
do I = 65,96
   do J = 64, 95
      S1: Atp1(I,J) = c* (A(I,J+1)+A(I,J-1)+
                A(I-1,J)+A(I+1,J) - 4.0*A(I,J))
}
```

Figure 5: Code executed by processor 1 with multiple blocks of A

Simplified code executed by processor 1

```
{
Let D(A) be the distribution descriptor of A
D(A).first is first block of A, D(A).last is last
lo$dim and hi$dim are bounds for dimension $dim
do bl = D(A).MYPROC.first, D(A).MYPROC.last
   do I = max(D(A).lo1(bl), 2), min(D(A).hi2(bl), 128)
      do J = max(D(A).lo2(bl), 2)-1,
                     min(D(A).hi2(bl), 128)-1
         S1: Atp1(I,J) = c* (A(I,J+1)+A(I,J-1)+
                A(I-1,J)+A(I+1,J) - 4.0*A(I,J))
}
```

Figure 6: Alternative code executed by a processor with multiple blocks of A

processor 1 using approach (ii).

3.2 Local storage allocation

It is easy to see that while the reference that we choose to be data-centric with accesses only local data, other references to the data item could necessitate access to data not local. We need to allocate storage locally for all data items that a processor requires to perform its iterations. In this section, we describe the analysis required for local storage allocation.

Let us assume that A has been distributed in a fully general block distributed manner onto a set of processors. Let A_{pj} denote block j of A on processor p. Let R be the reference to A which we have chosen to be data-centric. Let $\mathbf{F}(r)$ be the access matrix [16] for any reference r, i.e. iteration $\vec{i}$ touches data item $\mathbf{F}(r) * \vec{i}$ through reference r. We define the following concepts:

Definition 1 *The* per-block owned data $\mathbf{OB}(p, A_{pj})$ *of a block is the elements of the data item it contains.*

Definition 2 *The* per-block local iteration space $\mathbf{LB}(p, j)$ *is defined as*

$$\mathbf{LB}(p, A_{pj}) = \{\vec{i} | F(R) * \vec{i} \in \mathbf{OB}(p, A_{pj})\} \qquad (1)$$

Definition 3 *The* per-block per-reference view set *for block j for reference r on processor p, written as $\mathbf{vB}(r, p, j)$, is defined as*

$$\mathbf{vB}(r, p, A_{pj}) = F(r) * \mathbf{LB}(p, A_{pj}) \qquad (2)$$

Definition 4 *The* per-block view set *for block j on processor p for* A, *written as $\mathbf{VB}(p, A_{pj})$, is defined as*

$$\bigcup_r \mathbf{vB}(r, p, A_{pj})$$

Localized index sets

```
{
do I = 65, 96
   do J = 64, 95
      S1: Atp1(I,J) = c* (lA12(I-63,J+1-62)+
            lA12(I-63,J-1-62) + lA12(I-1-63,J-62) +
            lA12(I+1-63,J-62) - 4.0*lA12(I-63,J-62))
}
```

Figure 7: Code executed by processor 1 after translation to local index sets

where r ranges over all the references to A in the loop nest.

The *per-block view set* represents the set of elements that are required for the execution of a block for the given loop nest. In general, this set is arbitrarily shaped. To allocate storage, we take the smallest enclosing isothetic rectangle (sides parallel to the axes) and allocate storage for that.

We demonstrate the computation of view sets for our running example next. We choose block A_{12}, represented by the tuple $\{([65,65],[32,32])\}$. Let `A(I,J+1)` be the data-centric reference. $\mathbf{LB}(1, A_{12})$ is the set of iterations for which I is in the range $65 \ldots 96$, and J is in the range $64 \ldots 95$. The *per-block per-reference view set* for A_{12} for the references `A(I,J)`, `A(I+1,J)`, `A(I-1,J)`, `A(I,J+1)` and `A(I,J-1)` are $\{([65,64],[32,32])\}$, $\{([66,64],[32,32])\}$, $\{([64,64],[32,32])\}$, $\{([65,65],[32,32])\}$ and $\{([65,63],[32,32])\}$ respectively. The smallest rectangle that encloses the union of these five view sets is the set $\{([64,63],[34,34])\}$. This is precisely what the *ghost region* support in the libraries such as LPARX and DAGH do and is a generalization of overlap analysis introduced by Gerndt [10]. Note that this storage can be determined automatically given the data distribution. In particular, no information about the type of stencils used, etc need be conveyed to the compiler, since the data usage information can be extracted from the input code. Note that the storage allocated for each block on a processor is enough to contain the *view set* of the block, which is a *superset* of the elements *owned* by the block. Finally, note that if an element of an array occurs in the view sets of multiple blocks, space is allocated for it multiple times.

3.2.1 Global and Local index translation

In the node program on each processor of a distributed memory machine, all array accesses must be to local indices. The storage for the view set of each block introduces a specific index translation from the global indices to local indices. Each local index of a given block is at a fixed offset from the corresponding global index, which is determined by the global index of the *first location* of the view set.

For example, the view set of A_{12} requires a local array of size 34-by-34 elements. Element (1,1) of this local array (call it lA12) corresponds to element A(64,63) of the global array. Consequently, a local index of A_{12} is related to the corresponding global index by the offset vector $(63, 62)$. Given this information, it is easy to see that the enumeration of the *per-block iteration space* for A_{12} corresponds to Figure 7.

3.3 Communication sets

The final step in the parallelization process is one in which every processor determines what data to send and what data to receive before the start of execution of a loop. To mathematically formulate this problem, we define the following:

Definition 5 *The* owns set *for data item* `A` *on processor p,* $\mathbf{O}(p, A)$*, is defined as the union of* per-block owned data *sets of all blocks of A on p. In other words,*

$$\mathbf{O}(p, A) = \bigcup_j \mathbf{OB}(p, A_{pj})$$

where j ranges over all blocks of A on p.

Definition 6 *The* view set *for data item* `A` *on processor p, written as* $\mathbf{V}(p, A)$*, is defined to be the union of the* per-block view sets *over all the blocks of* `A` *on p. In other words,*

$$\mathbf{V}(p, A) = \bigcup_j \mathbf{VB}(p, A_{pj}),$$

where j ranges over all blocks of A on p.

Two processors p and q need to communicate elements of A if $\mathbf{V}(p, A) \cap \mathbf{O}(q, A) \neq \phi$. Let $\mathbf{S}(p, q, A)$ denote the elements of `A` that processor p needs to send to processor q. Let $\mathbf{R}(p, q, A)$ denote the elements of `A` that processor p needs to receive from processor q. Then,

$$\mathbf{S}(p, q, A) = \mathbf{R}(q, p, A) = \mathbf{O}(p, A) \cap \mathbf{V}(q, A) \qquad (3)$$

Once the communication sets are available, a generic communication routine enumerates over all the elements of these sets and performs the appropriate send (and recv) operations. In Section 5, we present a fast algorithm for computing the communication sets efficiently.

4 Modification for dynamic and adaptive distributions

In general, the grid hierarchy may be known only when the program begins execution (dynamic case), and it may even be modified during the course of program execution (adaptive case). How does this affect the techniques described in Section 3?

To enumerate over the local iteration space, the compiler must generate code similar to that in Figure 6, rather the code in Figure 5. Local storage allocation and computation of communication sets is dependent on knowing the data distribution. While the compiler cannot compute the appropriate sets at compile time, it is easy to write code that computes these sets at runtime and performs storage allocation and communication as needed. This code is part of the runtime library. Before each loop nest begins execution, the library code determines communication sets, and a generic communication routine enumerates over all elements of the communication sets, performing the appropriate send (and recv) operations.

There are two considerations if the data distributions of arrays can change at run time:

1. Since data descriptors are replicated, they must be updated consistently across all processors.

2. A more important concern is the efficiency of computing sets such as communication sets. In the static or dynamic case, these sets are computed once, so efficiency of this computation is not a major issue. In the adaptive case, this computation is performed repeatedly, so it is important that it be performed efficiently. We address this problem next.

5 Incremental Communication Set Generation and Maintenance

Data redistribution in the adaptive case is usually incremental, so an incremental computation of communication sets is preferable to complete recomputation. We show how techniques from computational geometry can be used to accomplish this.

First, we examine the case when the *per-block per-reference view sets* are all rectangular parallelopipeds. This will be the case for any reference for which the access matrix can be written as the product of a diagonal matrix and a permutation matrix. The exact set of indices that need to be communicated between every non-local block which *owns* some data in the *per-block per-reference view set* of some local block is the intersection of the rectangles representing these two sets. The general problem to determine the communication for a single block is the following: Given a set of input blocks on all processors representing data ownership, determine the intersection of the *per-reference per-block view set* rectangle with the input set. This is a variant of the well known multidimensional range-search problem in computational geometry [12]. The rectangles representing data ownership correspond to *input rectangles* and the rectangle representing the *per-reference per-block view set* is called the *query rectangle*.

There is a naive solution to the above problem - enumerate over all the rectangles in the input set and test each rectangle for intersection with the query rectangle. Two rectangles in d- dimensions intersect if and only if their projections on all d axes intersect. Evaluating this takes $O(d)$ time. Thus, a naive solution to the query formulated above takes $O(Nd)$ time, where N is the number of input rectangles (the number of blocks of data distributed).

Pre-processing the input set of rectangles representing ownership information, as described below, allows us to answer each query in time $(log^d(N) + k)$ time [13], where k is the number of rectangles that intersect. In other words, given a certain number of input rectangles, the query time grows as the size of the actual number of positive results for the given query, rather than the size of the input set. Since these queries have to be solved quite often, the savings can be substantial in large problems.

We finally note that every time we perform a query intersection with the input set of rectangles, we can *cache* this result, so that in future identical queries, we

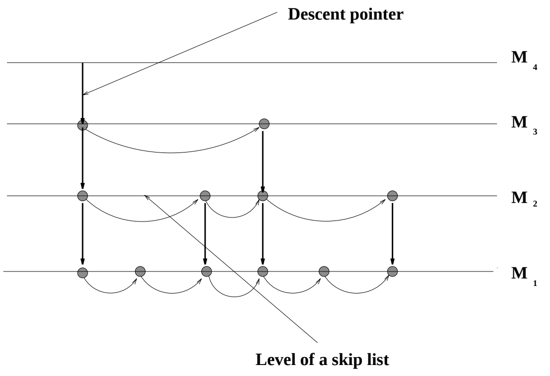


Figure 8: An example skip list

Procedure BuildSkipList(M)

```
{
  M is an input set of coordinates on the real line
  M_1 = M, i = 1
  while (M_i) is non-empty do
    create M_{i+1} = φ
    for (all elements e ∈ M_i do)
      randomcoin() is a fair coin
      toss = randomcoin()
      if (toss == head)
        M_{i+1} = M_{i+1} ∪ e
      endif
    endfor
    i++
    endwhile
  nlevels = i
  for j = nlevels, 1, -1 do
    for (all elements e ∈ M_j do)
      create a descent pointer to element at same location
    endfor
  endfor
}
```

Figure 9: Procedure for constructing a skip list from a set of input coordinates

can simply reuse the result of the earlier evaluation. Thus, we can save on computations for view sets for same references in different loop nests, for example.

5.1 Orthogonal range-search problem

The *orthogonal range-search problem* is defined as follows: In d- dimensions, an orthogonal object is defined to be simply the cartesian product of d intervals. Given an input set $\mathbf{I}$ of input orthogonal objects and a query orthogonal object q, the solution to the *orthogonal range search problem* is required to report all objects in $\mathbf{I}$ that intersect with q. Because of space constraints, we discuss only the solution of this problem when d = 1.

In 1-dimension, an orthogonal object is a line segment, which is completely specified by its endpoints. The problem then is, given a set of input line segments, to report all line segments from this set that intersect a query line segment. This is done as follows.

Procedure searchSkipList(L, q)

```
{
  L is the skip list, and q is a query coordinate
  identify the interval at every level of L containing q
  level = toplevel; child = descent pointer from top level
  q is located trivially in the top most level
  for i = nlevels-1, 1, -1 do
    parent = interval containing q in level i+1
    left = left end point of parent;
    ld = descent pointer to level i from left
    j = ld
    while (coordinate of point j in level i ¡ q)
      j = next point at level i
    endwhile
    store [j..j+1] as the interval containing q at level i
  endfor
}
```

Figure 10: Procedure to determine the intervals of skip list containing a query point

5.1.1 Randomized skip list

The first step is to pre-process the input line segments into a data structure that can be queried efficiently. The particular data structure we choose is called the randomized skip list. Let M be any given set of m points on the real line R. Given a fair coin, a randomized search structure is associated with M. Starting with M, a sequence of sets $M = M_1 \supseteq M_2 \supseteq M_2 \supseteq M_{r-1} \supseteq M_r = \phi$, where set M_{j+1} is obtained from set M_j by tossing the fair coin for each point in M_j and including only those points in M_{j+1} for which the toss results in a head. This sequence of sets is known as a *gradation*. The expected number of levels in the gradation is $O(log\ m)$. Each level of the partition is stored as a linked list of the points in that level in ascending order. In addition, a *descent pointer* is maintained from each point in a level to the point in the immediately lower level with the same coordinate. The topmost level of the gradation (which is empty) maintains a single pointer to the first point in the next lowest level. This storage mechanism for the gradation is known as a *skip list* and is shown in Figure 8. It may also be noted that the points in every level of the skip list break up the real line into a set of intervals. $\mathbf{H}(M_i)$ denotes the set of intervals generated by level i. Figure 9 shows the procedure for constructing a skip list from an input set of points. A skiplist can be used for searching using the code in Figure 10. The expected cost of the search procedure is equal to the number of children of intervals in the various levels of the skip list that contain the query point. It is straightforward to show this to be equal to $O(log\ m)$.

5.1.2 Augmented skip list

To use the skip list to answer queries about segment intersections, certain augmentation must be performed. Given s input segments in set S, the set M consists of $2 * s$ elements and is formed by taking all the endpoints of all the segments of S. A skip list is formed using the procedure described in section 5.1.1. An interval of a skip list is defined to be an interval of the real line that belongs to $\mathbf{H}(M, i)$ for some i. The parent of an interval I at level i is defined to be the interval J at

Procedure genRecv(b, D, r)

```
{
    b is a block of data of array A
    D is A's distribution descriptor, r is a reference
    Let E be the set of segments in D
    M is the skip list on endpoints of segments in E
    R = E ∩ vB(r, MYPROC, b).
    for (all elements r of R do)
        recv-section = r ∩ vB(r, MYPROC, b)
        mpi-recv(owner(r), recv-section)
    endfor
}
```

Figure 11: Code a processor executes to determine recv calls it must insert for a given block

level $i+1$ that contains I. An input segment S_i of S covers an interval I of M if S_i contains I. S_i is defined to cover I *canonically* if S_i covers I and the parent of I covers S_i. The following lemma holds when S_i covers I canonically.

Lemma 1 *An interval S_i covers an interval I of a skip list* canonically *iff the parent of I contains an endpoint of S_i and S_i covers I.*

Thus, the intervals of a skip list covered canonically by a given input segment S_i can be determined by testing the children of the intervals of the skip list containing an end point of S_i

An *augmented skip list* is formed by storing with each interval I of a skip list, the list of input segments that cover I canonically.

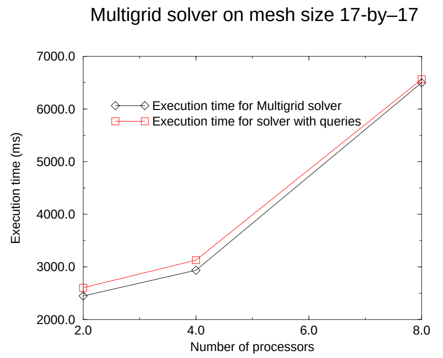
We state the following theorem without proof.

Theorem 1 *A skiplist built on an a set of s intervals of set S can be augmented in $O(slog\ s)$ time and the augmentation requires $O(slog\ s)$ space.*

5.1.3 Answering intersection queries and generating send/recv calls

How do we answer queries regarding the intersection of a query segment with the set of input segments? It is easy to see that two segments intersect if one of them contains an endpoint of the other. When presented with a query segment, we first report all the input segments that canonically cover all intervals of the skiplist containing an end point of the query segment. We also need to report all the input intervals completely contained inside the query segment. This is done by locating the left endpoint of the query segment in the lowest level of the skip list, and traversing the level starting from there and reporting all segments for which we find both points within the query range. A more detailed discussion of this procedure and a generalization to d-dimensions is presented in [13]

How does this relate to doing sends and receives? We show the relationship in one-dimension. The relationship in higher dimensions is similar. Figure 11 shows how the results of the range search problem can be used to generate recv calls for a single block of data. The procedure for generating the send calls is analogous.



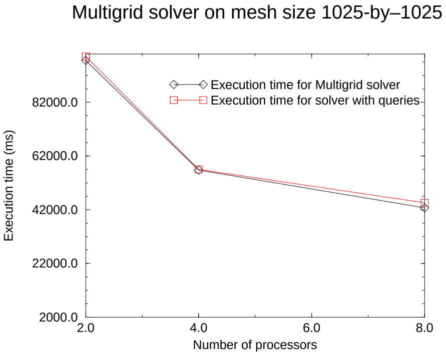


Figure 12: Performance numbers for multigrid solver: (a) small problem, (b) large problem

5.2 Dynamization

The skip list allows for efficient dynamization. Segments can be deleted and inserted into the skip list very efficiently. This allows for incremental maintenance of distribution information in the skip list when data distribution changes at run time. The following theorem (stated without proof) is crucial to the efficient dynamic behavior of the skip list [13].

Theorem 2 *A single point can be inserted into or deleted from a skip list in time $O(log\ m)$, where m is the number of points in the skip list.*

6 Performance

In this section, we present some preliminary performance measurements, shown in Figure 12. We implemented a 2-D multigrid solver using the DAGH library with published code [4]. This is the line labeled “execution time for multigrid solver” in Figure 12. Next, we created a version of this solver which was augmented with the actual intersection queries the compiler would insert to determine the communication sets at run time. This is the line labeled “execution time for solver with queries” in Figure 12. Since the compiler does not have any knowledge of specific data distribution policies (which the library does) and the distribution is unknown at compile time, the compiler-generated code must execute these queries. As can be seen, the resulting overhead is minimal. We ran both versions of the solver on two different problem sizes: a small mesh of 17-by-17 points and a large mesh of 1025-by-1025 points. Figure 12(a) actually exhibits a slowdown because of the small problem size, but we show this case to verify that the absolute overhead introduced by our approach is small. Figure 12(b) is more realistic. Overall, the overhead we introduce is within 7% for the small example and less than 1% for the large problem. We are currently working on obtaining similar numbers for the 2-D wave equation. We have also quantified the performance benefit of using an asymptotically optimal algorithm rather than the naive algorithm. When the number of blocks of data is less than 60, the naive search algorithm is faster. For larger numbers of blocks, the asymptotic algorithm is faster, but there is the penalty of preprocessing. When there were 400 blocks for example, an average query using the naive algorithm takes approximately $13\mu s$, while the optimal algorithm takes approximately $7\mu s$. The preprocessing time in this case takes 2.34 milliseconds, which means that at least 390 queries must be made for the optimal algorithm to be useful. For the sake of comparison, in our multigrid implementation, a total of 1710 queries were made for the small problem size and 4290 queries were made for the large problem size. We conclude that while the performance advantage of the incremental algorithm is small, it is easy to code and worth implementing.

7 Conclusions

We have presented a parallelization framework for "fully general block distributions" which enables us to extend current compiler technology to meet some of the needs of semi-structured applications.

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. On efficient runtime support for multiblock and multigrid applications: Regular section analysis. Technical Report CS-TR-3140 and UMIACS-TR-93-92, University of Maryland, Department of Computer Science and UMIACS, October 1993.

[2] Scott B. Baden, Scott R. Kohn, and Stephen J. Fink. Programming with lparx. Technical Report CS94-377, Department of Computer Science, University of San Diego, 1994.

[3] D Balsara, M Lemke, and D Quinlan. AMR++, a c++ object oriented class library for parallel adaptive refinement fluid dynamics applications. In *American Society of Mechanical Engineers, Winter Annual Meeting, Anaheim, CA*, volume 157, pages 413–433, 1992.

[4] Matthew W. Choptuik, Scott Klasky, and Manish Parashar. Parallel multigrid with the dagh package: Specifications and applications. Draft, September 1995.

[5] Nikos Chrisochoides, Animesh Chatterjee, Rajani Vaidyanathan, and Geoffery Fox. An evaluation of data mapping approaches for parallel multi-block euler solvers. Accepted for publication in Parallel CFD 1994.

[6] Greg Cook, Steve Brandt, Joan Masso, John Shalf, Paul Walker, Matt Choptuik, Mijan Huq, and Manish Parashar. 2d wave equation application.

[7] High Performance Fortran Forum. High performance fortran language specification vertion 1.0, 1993.

[8] High Performance Fortran Forum. High performance fortran language specification, version 2, specification. http://www.crpc.rice.edu/HPFF/hpf2/index.html, October 1996.

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

[10] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessor Systems*. PhD thesis, Universitát Bonn, 1990.

[11] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Unified framework for sparse and dense smpd code generation. Technical Report TR97-1627, Cornell University, 1997.

[12] Ketan Mulmuley. *Computation Geometry - An Introduction Through Randomized Algorithms*, chapter 2. Prentice-Hall.

[13] Ketan Mulmuley. *Computational Geometry - An Introduction Through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[14] Manish Parashar and James C. Browne. Distributed dynamic data-structures for parallel adaptive mesh refinement. In *HiPC*, 1995.

[15] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *SIGPLAN89 conference on Programming Languages, Design and Implementation*, Jun 1989.

[16] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

[17] H. Zima, H. Bast, and M. gerndt. Superb, a tool for semiautomatic mimd/simd parallelization. In *Parallel Comput.*, pages 1–18, 1986.