An assumed partition algorithm for determining processor inter-communication [1]

A. H. Baker, R. D. Falgout, U. M. Yang

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Box 808 L-551, Livermore, CA 94551

Abstract

The recent advent of parallel machines with tens of thousands of processors is presenting new challenges for obtaining scalability. A particular challenge for large-scale scientific software is determining the inter-processor communications required by the computation when a global description of the data is unavailable or too costly to store. We present a type of rendezvous algorithm that determines communication partners in a scalable manner by assuming the global distribution of the data. We demonstrate the scaling properties of the algorithm on up to 32,000 processors in the context of determining communication patterns for a matrix-vector multiply in the *hypre* software library. Our algorithm is very general and is applicable to a variety of situations in parallel computing.

Key words: global partition, processor inter-communication, rendezvous algorithm, scalability, distributed directory

1 Introduction

Scalability is a key issue for scientific computing due to the ever-increasing demand to solve large applications on high performance massively parallel computers. We view an application code as scalable if that code can use additional computational resources effectively. In particular, if we increase the problem size and the number of processors in such a way that the local problem size on a single processor stays fixed, the application code execution time

[1] This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Email addresses: `abaker@llnl.gov` (A. H. Baker), `rfalgout@llnl.gov` (R. D. Falgout), `umyang@llnl.gov` (U. M. Yang).

should remain relatively constant. Many factors may affect the scalability of a particular code, such as the machine architecture or various properties of the algorithm (e.g., the convergence rate of a linear solver). Furthermore, scalable implementation choices are particularly important when developing software for scientific computing.

Many algorithms that are relatively scalable for several thousand processors may not scale well on machines with ten times as many processors. In fact, these new machines with tens of thousands of processors, such as BlueGene/L at Lawrence Livermore National Laboratory, are forcing the re-examination of some existing computations in scientific software for parallel computing. As always, minimizing computation and communication costs is critical for good performance. However, with many more processors, minimizing storage also becomes an issue. Data structures that depend on the number of processors (P) may be problematic as P approaches 100,000, particularly if the machine has a relatively small amount of memory per processor. (BlueGene/L has 256 MBytes per processor [2].)

We focus on a situation in parallel computing that is common to much scientific software. In a parallel application code, problem data is distributed across processors. When that application code utilizes a parallel software library, for example, the problem data is most efficiently passed to the library in its distributed form. It is typically the case that for the library to perform its function, some information regarding the global distribution of the data will be needed. Certainly in the case of a linear solver library, each processor cannot continue to work independently. A solver algorithm requires that a processor obtain "nearby" data from other processors in order to complete the solve. While a processor may easily determine what data it needs from other processors, it may not know which processor owns the data it needs. Therefore, processors must determine their communication partners, or neighbors. The problem of determining inter-processor communication (in the absence of a global description of the data) in a scalable manner is the focus of this paper.

We examine algorithms for determining neighbors in the context of the the *hypre* software library [7,6]. The *hypre* software library provides high performance preconditioners and solvers for the solution of large, sparse linear systems on massively parallel computers. For ease of use, these solvers are accessed from the application code via *hypre*'s conceptual interfaces, which allow a variety of natural problem descriptions. For a code utilizing *hypre* to be scalable, the interfaces as well as the solver algorithms must be be scalable.

Our specific goal for determining neighbor communications in a scalable manner is an algorithm that depends on the number of processors logarithmically or better in terms of computation, communication, and storage requirements. Minimizing storage is particularly challenging as the most straightforward approach involves constructing a global partition of the data which requires $O(P)$ data storage.

In this paper, we present an algorithm for determining neighbor data on tens of thousands of processors in a scalable manner. Under reasonable assumptions, our new algorithm achieves scalability by assuming the global distribution of the data. In particular, the new algorithm is a type of parallel rendezvous algorithm that uses the concept of an assumed partition to answer queries about the actual global distribution of the data. This algorithm concept has wide applicability to other situations within *hypre* and to other application codes as well. For example, many situations that require a call to MPI_Allgatherv can be easily restructured to use the assumed partition concept instead, thereby reducing storage and improving scalability.

This paper is organized as follows. In Section 2, we discuss previous neighbor-finding algorithms and related work as well as describe the conceptual interfaces in *hypre*. We introduce our assumed partition algorithm in general terms in Section 3. We then present the details of the assumed partition algorithm in the context of the *hypre* linear-algebraic and structured-grid interfaces in Sections 4 and 5, respectively. Numerical results and cost analyses are also given. In Section 6, we discuss a general algorithm we developed that facilitates implementation of the assumed partition idea by allowing data exchange in a generic way. Finally, we close with some concluding remarks in Section 7.

2 Background

A unique feature of the *hypre* software library is the provision of multiple conceptual views for describing the problem being solved. These *conceptual interfaces* allow the user to access the library by describing the problem in the manner that is most natural. To illustrate the need for determining communication neighbors in a scalable manner, we focus on two of the conceptual interfaces in *hypre*: the linear-algebraic interface (`IJ`) and the structured-grid interface (`Struct`). These two interfaces allow access to different solvers and may produce different data layouts within *hypre*. The `IJ` interface is a standard linear-algebraic interface for applications with sparse linear systems. With this interface, the user describes a linear system in terms of a sparse matrix and a vector, i.e. in terms of row and column indices. The `Struct` interface, on the other hand, is appropriate for applications with logically rectangular grids. This interface allows the user to describe a linear system in terms of grids and stencils and access grid-based solvers such as geometric multigrid. With this interface, the details of forming a linear system are invisible to the user. A detailed discussion of the design and use of all of the conceptual interfaces in *hypre* can be found in [3,4].

We are interested in the situation in which each processor is aware only of its own portion of a global problem. For example, a processor may only know of the rows it owns in a matrix or its particular piece of a global grid. In the case of a linear solver library, for example, consider the need of a solver to perform a matrix-vector multiply. To do so, each processor will need to determine two things: which processors to receive data from and which processors to send data to for the multiply operation. In effect, its communication neighbors must be determined by the library. While a library could require its users to provide relevant communication pattern information, we consider this a non-trivial and unrealistic burden on the user in most cases.

A straightforward approach for determining neighbors is for each processor to construct a global partition of the data. This global partition can be constructed via a global collective communication. Once each processor knows what data every other processor owns, determining which processors to receive data from is straightforward. At this point, a second global communication can convey the receive processor information. This second communication of data allows processors to determine what data they need to send and to whom. Such an approach requires $O(\log(P))$ communication costs and $O(P)$ storage. Computation costs may easily be $O(P)$ as well, depending on the implementation. While such costs are not problematic for a moderate number of processors, they become problematic as P approaches 100,000.

In [5], the current algorithms for determining neighbor communication patterns in *hypre* are discussed and analyzed in detail. These algorithms essentially follow the straightforward approach just described, with a few optimizations. The algorithm cost dependence on P is $O(\log(P))$ for communication and $O(P)$ for storage and computation. Therefore, in this paper we follow up on ideas for a scalable assumed partition algorithm suggested in [5]. In particular, we implement and analyze the ideas in [5] for the linear-algebraic interface, and, more significantly, we develop a method for determining a reasonable assumed partition in the structured-grid interface. The structured-grid interface presents particular difficulties because the assumed partition must be two or three-dimensional, depending on the problem, and the grid may contain "gaps".

We mention that determining inter-processor communications patterns in the case of distributed problem data has been addressed in [9] and [10]. There, the primary concern is avoiding redistributing large amounts of data. In [9], an algorithm that creates a distributed directory is suggested. The directory is then populated via a hash function. This directory is used by a rendezvous algorithm to determine neighbors. Our assumed partition idea is similar except that we do not restrict ourselves to a hash function, but instead we assume the distribution of the data. More importantly, our main focus in this work is to avoid linear dependence on P in all aspects of the algorithm costs, particularly storage costs, and this concern is not an emphasis in [9].

3 The assumed partition algorithm idea

As mentioned in Section 1, to calculate communication neighbors in a scalable manner, the algorithm costs should at most depend on P logarithmically. In this section, we give a general description of the assumed partition algorithm idea. Then specific algorithm details are given in the context of two different interfaces in *hypre* in the following two sections.

The general assumed partition algorithm idea can be described in the following three steps.

```
Algorithm 1: Assumed Partition
1.1. Assume the global distribution of data (define $\mathcal{F}_a$)
1.2. Redistribute the actual partition description to the assumed partition
1.3. Use the assumed partition to determine global information
```

The novelty of this method lies in step 1.1 where a global description of the problem data is assumed by the algorithm. We refer to the global distribution of the data as the *actual partition* and the distribution of the data assumed by our algorithm as the *assumed partition*. The key is to define the assumed partition of the data in a manner that requires minimal storage (or no storage) and can be queried by a function call that costs $O(1)$ operations. We refer to this function as the assumed partition function, or $\mathcal{F}_a$. For example, for the case where problem data consists of a matrix and a vector, a typical query would ask which processor owns a given row, or conversely, what range of rows a particular processor owns.

In step 1.2 of the algorithm, data describing the actual partition is redistributed to the assumed partition. This step in effect creates a distributed directory as each processor learns who actually owns the data that it is assumed to own. Note that in this algorithm the actual problem data does not need to be redistributed: a description of who owns the data is sufficient and requires less storage and data transfer.

In step 1.3, each processor uses the assumed partition to determine global information. This requires a rendezvous algorithm, and the assumed partition defines the rendezvous locations. In the case of determining the required communication patterns for a matrix-vector multiply, each processor uses the assumed partition to learn who to send data to and receive data from. However, one can envision this algorithm more generally being of use when an MPI_Allgatherv is used, but each processor needs only its own particular subset of the global data.

4 The linear-algebraic interface (`IJ`)

To access *hypre*'s linear solvers via the `IJ` interface, the user defines the matrix and right-hand side in terms of row and column entries. As with all the interfaces in *hypre*, the user provides the data in distributed form. Matrices and vectors are distributed across P processors in contiguous blocks of rows.

For example,

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_P \end{pmatrix}, \tag{1}$$

where each submatrix A_k is owned by Processor k, $k = 1, \ldots, P$. We define N as the global number of rows in coefficient matrix A. Clearly, storing a global partition of the data requires an array of size $O(P)$. More detail on this interface may be found in [5] or [3].

In this section, we first describe the three steps of the assumed partition algorithm, Algorithm 1, in terms of the `IJ` interface. We then provide numerical results which illustrate the benefit of using an assumed partition algorithm.

4.1 The `IJ` *assumed partition algorithm*

The assumed partition algorithm proceeds as follows. Recall that each processor only knows the range of rows that it owns and has no information regarding the rows on other processors. Therefore, in step 1.1, we determine an assumed partition of the matrix that is defined by function $\mathcal{F}_a$ and is available to all processors.

For example, a function describing a balanced partition that for a given row number, r, returns a processor id, p, is $p = \mathcal{F}_a(r; P, N) = \lfloor (r \times P)/N \rfloor$, where the global number of rows and processors are parameters. The inverse of that function also needs to be available: for a given processor id p, return the range of rows that Processor p owns. Note that the chosen assumed partition function need not describe the actual partitioning of the data. (This point will be particularly obvious in the context of the `Struct` interface.)

Step 1.2 requires that each processor reconcile its actual and assumed partition data. To be specific, Processor p uses $\mathcal{F}_a$ to determine which processors are *assumed* to own the rows that it *actually* owns. Processor p must contact each of these processors to let them know that it owns their assumed rows. In this way each processor learns the actual owners of all the rows in its assumed row range. Figure 1 illustrates how the assumed and actual partitions may differ with four processors. In this example, horizontal bars represent rows 0 to $N-1$ for the two partitions which are divided into four ranges for four processors numbered 0 to 3. The vertical dashed lines indicate where the actual and assumed partitions of this data differ. The arrow indicates rows owned by processor 0 that are in the assumed partition of processor 1. Therefore processor 0 must contact processor 1 with the range of rows indicated by the arrow. Similarly, processor 1 must contact processor 2, processor 2 must contact processor 3, and processor 3 contacts no one. In this manner, a distributed directory is created such that each processor stores the actual owners of the rows it is assumed to own.

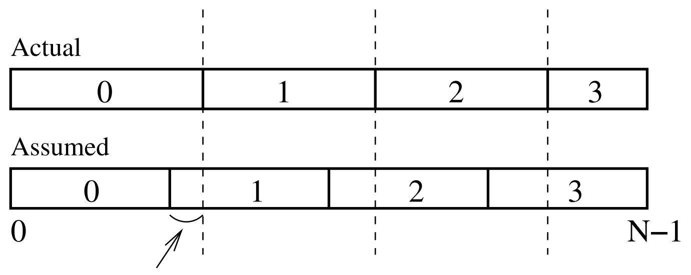

Fig. 1. A graphical representation of an assumed and actual partition of N rows of a matrix among four processors.

In step 1.3, the assumed partition is used to calculate inter-processor communication information. For example, to perform a matrix-vector multiply operation, the communication pattern is determined by the non-zero pattern of the matrix. Each Processor p needs to determine two types of neighboring processors: the *receive processors* and the *send processors*. The receive processors are those processors from which Processor p needs to receive vector data that it does not own to perform the multiply. Conversely, the send processors are those processors to which Processor p must send data. To calculate its receive processors, Processor p uses the assumed partition to determine which processors are assumed to own the data it needs. Processor p then contacts those assumed processors and learns from them who actually owns the data in question. Now Processor p knows all the processors that own the data it needs. Next, Processor p directly contacts these processors, telling them what data it needs. In this manner all processors learn their send processors.

Step 1.3 contains a complexity. Calculating the send and receive processors in the manner described requires point-to-point communications in which processors do not know how many times they will be contacted, by whom they will be contacted, or with how much data. It was suggested in [5] that a distributed termination detection algorithm (e.g., [8]) is appropriate for this situation. Therefore, we use a termination detection procedure that is appropriate for a number of such situations with unknown contact information. We discuss this algorithm in some detail in Section 6.

Because our goal is a scalable algorithm, we are interested in the algorithm costs for storage, computation, and communications in terms of the dependency on P, and we ignore other terms. In step 1.1 of the algorithm is a function that does not require storage or dependence on P. In step 1.2, costs depend on the number of neighbors that a processor has, which is independent of P. No termination detection is needed for this step as processors can probe for messages until they learn who owns each of their assumed rows. Any reasonable assumed partition function will require modest amounts of storage for each processor when the distributed directory is built ($O(1)$ in most cases), provided that the user's actual distribution is reasonable. For example, if the user assigns one row each to $P-1$ processors and the remaining $N-P-1$ rows to a single processor, this is not a reasonable distribution. (We are more precise in defining a reasonable distribution in Section 5.) In step 1.3, determining the send and receive processors requires storage, communications, and computations based on the number of neighbors. However, the termination detection algorithm requires a couple of sweeps of a binary tree of processors, resulting in $O(\log(P))$ costs.

4.2 Experimental results

Here we provide results from runs on BlueGene/L at Livermore National Laboratory. We note that only a subset of the machine is available at this time. When completed, BlueGene/L will have 65,536 nodes. Each node contains two 667 MHz processors and 512 MBytes of memory. (See [2] for more information on the machine architecture.) We compare the new assumed partition algorithm with the previous implementation that stored the global partition in the context of determining communication patterns for a matrix-vector multiplication. The problem that is used for the results in Figure 2 is a matrix derived from finite differences for a three-dimensional Laplace operator with a 27-point stencil on a cube. Each processor has 64,000 rows. Test runs were done using $P = 4^3$, 6^3, 8^3, 10^3,12^3, 14^3, 16^3,18^3,20^3, 22^3, 24^3, 25^3, 28^3, and 32^3 processors.

For small numbers of processors the previous implementation is acceptable. However, the scaling properties of the new method are clearly superior. While the calculation times are relatively small for this interface on this machine, more significantly, the storage requirements are only $O(1)$ with the new algorithm. The timing obtained with 28^3 processors slightly deviates from the linear trend for the old algorithm. We note that the BlueGene/L machine appears to be somewhat sensitive to the number of processors requested and their associated mapping to nodes. For example, numbers of processors that are powers of two tend to be very efficient. We have observed such repeatable anomalies in other codes as well when the executions times are as small as in Figure 2 (and later in Figure 12 as well).

5 The structured-grid interface (`Struct`)

The `Struct` interface is appropriate for problems that may be described in terms of a structured-grid and a fixed stencil pattern of non-zeros at each grid point. Consider a global d-dimensional index space. Boxes are described in terms of "lower" and "upper" corner indices as in Figure 3. Each processor owns some number of boxes in the index space, and these boxes collectively describe the grid (the non-empty space in the index space). The subset of the index space that we are concerned with is the bounding box for all of the grid boxes. Note that the grid boxes are non-overlapping. Therefore, instead of providing a matrix, the user provides the grid in distributed form such that each processor knows only about the grid boxes that it owns. Each processor owns at least one box, and the box indices indicate how data is related spatially. More detail on this interface may be found in [5] or [3].

For this interface, neighbor information needed by the structured-solver algo-

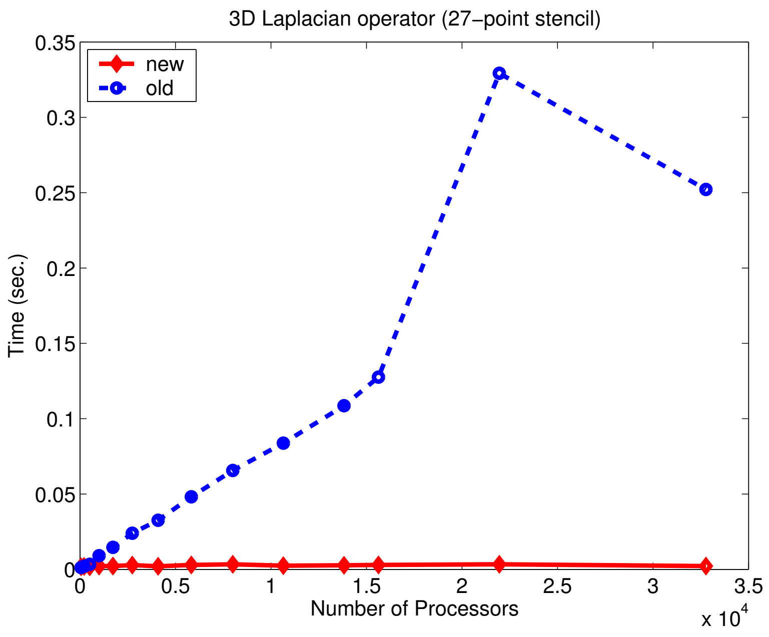

Fig. 2. A comparison of the new and old implementations for determining the inter-processor communications for a matrix-vector multiply with the IJ interface. The test matrix is the 3D Laplacian operator with a 27-point stencil.

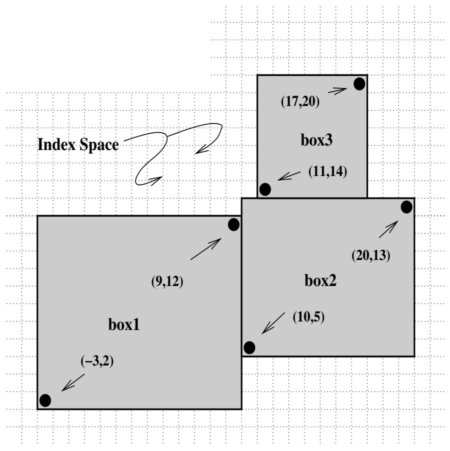

Fig. 3. A box is a collection of abstract cell-centered indices and is described by its lower and upper indices.

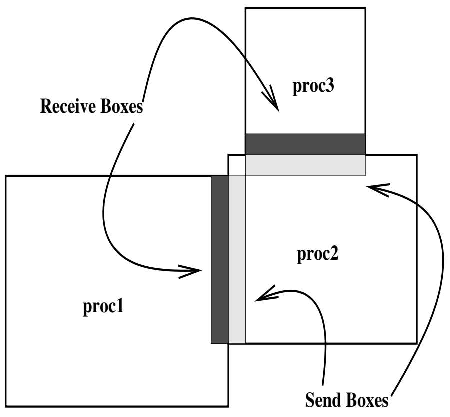

Fig. 4. Processor 2's send boxes are values owned by Processor 2 needed by other processors and its receive boxes values owned by other processors needed by Processor 2.

rithms depends on the spatial location of the grid boxes. Generally, processors need to be aware of boxes owned by other processors that are spatially close to their own boxes. Determining neighbors by constructing the actual partition is quite costly for this interface. To collect information about the global grid, each processor must send the extents of its boxes to all other processors. This can be done with a collective MPI_Allgatherv with $O(\log(P))$ operations. Unfortunately, the storage required is $O(B)$, where B denotes the total

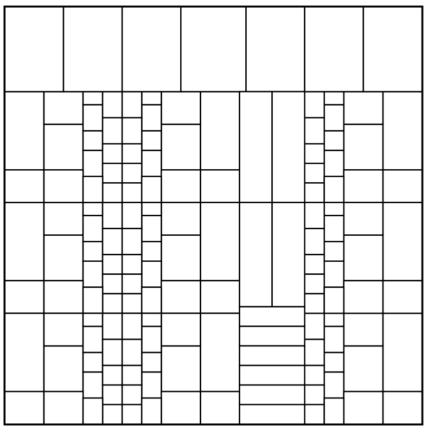

Fig. 5. An example of a two-dimensional structured-grid with no gaps in the bounding box.

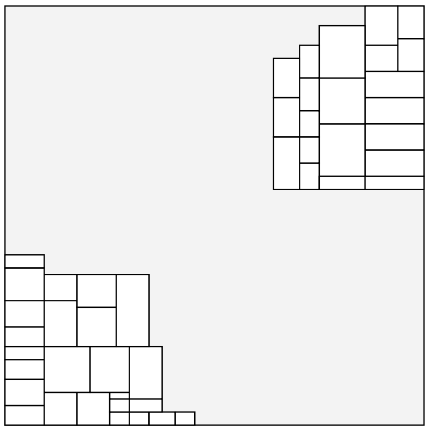

Fig. 6. An example of a two-dimensional structured-grid with gaps in the bounding box.

number of boxes in the grid and $B \geq P$. This can be an excessive amount of storage if P is large and each processor has even a modest number of boxes. In addition, in the case of a multigrid solver, each grid may require a partition. Next, neighbor data that needs to be sent and received to perform a matrix-vector product, for example, is determined based on spatial locality. In Figure 3, assume that each box is owned by a processor of the same number. Processor 2 needs to learn about boxes 1 and 3. If the given stencil requires the knowledge of boxes that are d grid points away, Processor 2 enlarges its box by d grid points in each direction and determines neighbors via a box-by-box comparison with the actual partition. This step involves $O(B)$ operations. In the case of a matrix with a 5-point stencil, for example, the data to be sent and received by Processor 2 is shown in Figure 4. It would be possible to reduce the $O(B)$ operations required for the box-by-box comparison by storing the actual partition in a data structure that indicates spatial position (such as an octree, for example). However, these types of data structures also have $O(B)$ storage requirements.

For this interface, a scalable algorithm for determining neighbors is particularly important. Recall that the `IJ` interface requires partitioning a one-dimensional index space where each index represents a row. In contrast, the `Struct` interface requires partitioning a two or three-dimensional bounding box that may or may not have gaps (i.e., areas with no grid boxes), as shown in Figures 5 and 6. Therefore, a more sophisticated assumed partition function is required to achieve good performance.

Next, we discuss an implementation of the assumed partition idea for the `Struct` interface. Because this algorithm is more complex than for the `IJ` interface, we provide more detail. We then analyze the cost requirements and present numerical experiments to demonstrate the benefit of the new algorithm.

5.1 The `Struct` assumed partition algorithm

We describe an assumed partition algorithm for the `Struct` interface in the context of the three steps given in Algorithm 1. We focus on the example grid in Figure 5. Then in Section 5.2, we discuss modifications required for large gaps in the grid as in Figure 6.

5.1.1 Step 1.1 of Algorithm 1 (constructing the assumed partition)

Consider the example two-dimensional grid shown in Figure 5 for sixteen processors. In this example, the bounding box is completely filled with grid boxes. We determine an assumed partition of the data by assigning each processor a subset of the bounding box such that each processor's individual assumed partition area is roughly equal. For example, in Figure 7, the dashed lines divide the bounding box equally into sixteen areas.

Recall that this division of the bounding box must be done virtually via an $O(1)$ function, $\mathcal{F}_a$, that answers queries about which processor owns which piece of the bounding box. To construct such a function, the following steps are performed:

```
Algorithm 2: Step 1.1 of Algorithm 1
2.1.  Determine the extents of the bounding box.
2.2.  Determine the number of assumed divisions in each dimension.
```

As each processor is only aware of its own grid boxes, the extents of the bounding box are obtained in step 2.1 via a collective call to MPI_Allreduce in which each processor contributes its minimum and maximum extents. Step 2.2 requires determining the number of necessary divisions in each dimension. For the example in Figure 7, three divisions are needed in each of the x and y directions to obtain sixteen areas of equal size. In practice, the length of the bounding box in the x and y directions may not be equal. Therefore, the number of divisions assigned to each direction is determined such that the assumed areas are as close to square as possible. For example, if the bounding box is twice as wide in the x direction as in the y, then we assign twice as many divisions to the x direction. In addition, the number of processors may not be a perfect square or may not factor into appropriate choices for the number of divisions. Therefore, we allow the number of areas in the assumed partition to exceed the number of processors, with the restriction of at most two areas per processor. Typically a very small number of processors will own two areas and the remainder own one area. For example, suppose we had fifteen processors for the grid in Figure 5. We could also subdivide the bounding box as shown in Figure 7 and assign one of the processors to two areas.

The bounding box extents and the number of divisions in each direction are stored by all processors. This storage is all that is required by the assumed

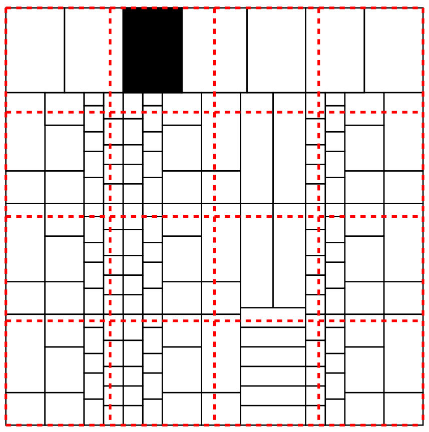
Fig. 7. The assumed partition for sixteen processors.

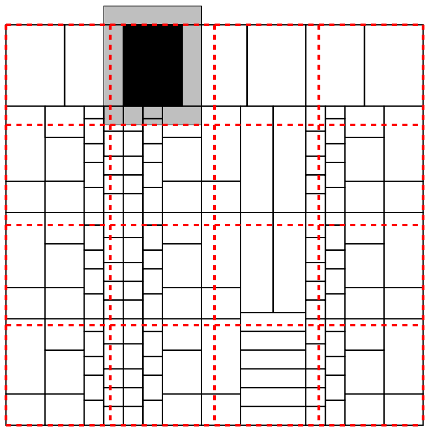
Fig. 8. Using the assumed partition to determine neighbor boxes.

partition. An $O(1)$ function is then easily constructed to answer queries about which processor owns which piece of the bounding box. For example, for the grid in Figure 7, given a processor id p, $\mathcal{F}_a$ returns the corresponding dimensions of the area assigned to Processor p in the assumed partition. Our implementation implicitly assigns processors to areas from left to right and from bottom to top starting with processor 0. Therefore, for processor id $p = 0$, the dimensions of the lower left area are returned by the function. Conversely, given any grid box extents, $\mathcal{F}_a$ returns the corresponding owner(s) of that box in the assumed partition. For example, given the extents of the black grid box, $\mathcal{F}_a$ returns the processor id $p = 13$.

5.1.2 Step 1.2 of Algorithm 1 (redistributing data to the assumed processors)

In step 1.2 of Algorithm 1, the assumed and actual partitions are reconciled as in the `IJ` case. Consider the black grid box in Figure 7. The processor that actually owns this black box must contact the corresponding owner in the assumed partition. In this manner, each processor learns of all the boxes that intersect its area in the assumed partition and builds its subset of the distributed directory. Our algorithm for this step consists of three parts:

```
Algorithm 3: Step 1.2 of Algorithm 1
3.1. For each local box, use $\mathcal{F}_a$ to determine the corresponding proces-
     sor id in the assumed partition.
3.2. Send local box descriptions to the appropriate processors.
3.3. Receive box descriptions from other processors that intersect my
     area in the assumed partition. Store these box descriptions.
```

Step 3.1 of the algorithm above requires calling the assumed partition function for each local box. Because a grid box may intersect more than one area in the assumed partition, the assumed partition function may return more than one processor id. Regarding step 3.2, we clarify that only the grid box extents are included in the distributed directory, and corresponding stencil data is not transferred. Additionally, this algorithm requires an unknown number of contacts from unidentified processors, and, therefore, a termination detection routine is required.

5.1.3 Step 1.3 of Algorithm 1 (determining neighbors with the assumed partition)

In step 1.3 of Algorithm 1, processors use the assumed partition to determine data that needs to be sent to and received from other processors. The assumed partition facilitates finding neighbors by supplying processors with information on potential neighbors. To be specific, consider again the same example in Figure 7. Processor k owns the black box and virtually enlarges it by the distance for which it wants neighbor information as shown in Figure 8. Then Processor k uses the assumed partition function to learn which processors' assumed partition areas intersect the enlarged black box. In this example, two areas in the assumed partition are intersected. Processor k then contacts the corresponding two processors and requests the list of grid boxes in their area of the assumed partition. Now Processor k has a list of potential neighbor boxes and can determine actual neighbors for the black box via box-by-box comparison.

Below we summarize this algorithm for finding neighbors:

```
Algorithm 4: Step 1.3 of Algorithm 1
4.1. Determine which processors' areas in the assumed partition could
     contain neighbors for my local boxes.
4.2. Contact these processors and receive the list of boxes in their areas
     of the assumed partition (the potential neighbor boxes).
4.3. Compare my local boxes to the potential neighbors boxes to de-
     termine actual neighbor boxes.
```

This algorithm also requires a termination detection routine as the number of contacts in unknown. In addition, note that this algorithm does not require or assume a resemblance between the actual partition and the assumed partition. Although the two distributions are likely to be similar in the `IJ` interface, no such resemblance is likely for the `Struct` interface. The distribution of data is further analyzed when we discuss algorithm costs in Section 5.3.

5.2 Gaps in the bounding box

Consider the example in Figure 6 where large portions of the bounding box do not contain grid boxes. When we construct an assumed partition for this example in the manner described in the previous section, the resulting partition is shown in Figure 9. For this example, only half of the processors have grid

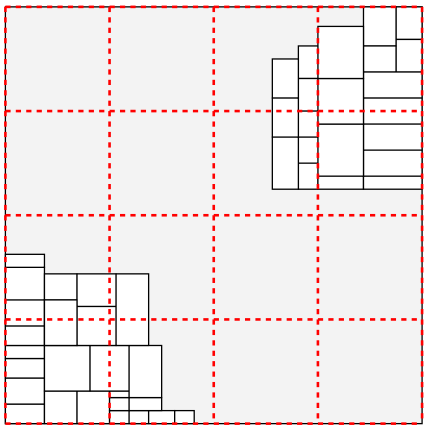
Fig. 9. The assumed partition regions for sixteen processors.

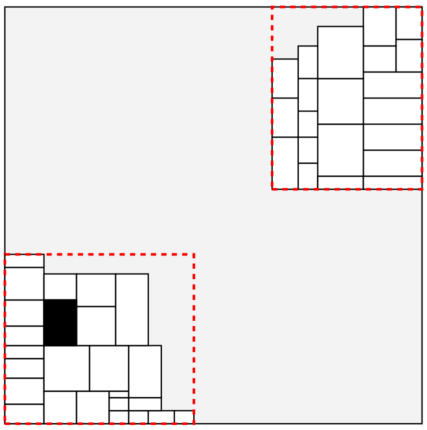
Fig. 10. Two assumed partition covering regions.

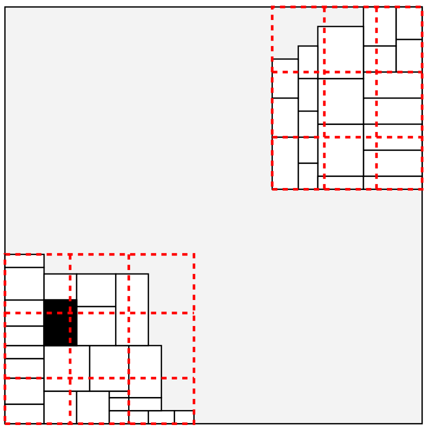
Fig. 11. The assumed partition regions for sixteen processors.

boxes in their area of the assumed partition. If the assumed partition does not properly represent where the grid boxes are located, then load balancing problems may occur when the distributed directory is built in step 1.2. For this reason, a more sophisticated assumed partition function is required when large gaps exist in the bounding box.

In the presence of gaps, we still assign processors to areas in the assumed partition as described in Section 5.1.1. However, instead of the assumed partition consisting of the entire bounding box whose area is denoted by V_b (in this discussion, the term "area" could be replaced by "volume" in the case of a three-dimensional grid), and dividing that area into $O(P)$ regions as in Figure 9, we first determine an assumed partition that more closely fits the grid boxes. In particular, we construct an assumed partition with total area V_c that consists of some small number of covering regions that contain the grid boxes. Let V_g denote the total area of all of the grid boxes. We construct the assumed partition such that $\frac{V_g}{V_c} \geq \gamma$, where γ is independent of P, and $V_g \leq V_c \leq V_b$. This reduces the gaps in the assumed partition area, thereby improving load balancing. For example, Figure 10 shows an assumed partition that consists of two covering regions that more closely fit the grid data. We first describe an algorithm to determine the covering regions and then explain how each processor is assigned to a covering region and ownership is assumed in that region via an $O(1)$ function.

Consider the following algorithm for determining the covering regions. We equate the covering regions to leaves in a quadtree (for problem dimension $d = 2$) or octree (for $d = 3$) data structure. A quadtree/octree is a tree-based data structure that relays spatial information. For example, to create a quadtree graph, a two-dimensional box is recursively divided into four quadrants. Each level of recursion corresponds to a level in the tree, and we denote the levels of the tree by m. Therefore, the number of leaves on each level m is $n_l = (2^d)^m$. We begin with level $m = 0$ which corresponds to one leaf representing the entire bounding box and continue until $\frac{V_g}{V_c} > \gamma$ is satisfied.

```
Algorithm 5: *Determining covering regions (Step 2.1 of Algorithm 2)*
 For $m = 1,\ 2,\ \ldots$
5.1. Determine the area (volume) of my local grid boxes in each leaf
     region.
5.2. Participate in an MPI_Allreduce to determine the global area
     (volume) of grid boxes in each leaf region.
5.3. Divide any leaf regions that do not contain at least $\gamma$ fraction of
     grid boxes into $2^d$ new leaf regions. If all leaves are sufficiently full,
     then stop iterating.
5.4. Determine the maximum and minimum extents of my local grid
     boxes in each leaf region.
5.5. Participate in an MPI_Allreduce to determine the global maxi-
     mum and minimum grid box extents and the number of grid boxes
     in each leaf region.
5.6. Eliminate all leaf regions that do not contain any grid boxes.
5.7. Shrink the remaining leaf regions according to the maximum and
     minimum grid box extents (now the leaves are bounding boxes).
```

When the above algorithm is applied to the grid in Figure 5, an assumed partition consisting of the entire bounding box is determined by the third step of the first iteration regardless of the value of γ. For the grid in Figure 6, we chose $\gamma = 0.6$. During the first iteration, the bounding box is split into four quadrants. Two of these quadrants contain no grid boxes (the upper left and lower right) and are eliminated. The extents of the remaining two quadrants are modified according to the bounding boxes. The two covering regions remaining at the end of the first iteration are shown in Figure 10. No further subdividing is needed in the second iteration as each region satisfies $\frac{V_g}{V_c} > \gamma$. Note that we do not need to store an actual quadtree data structure as storing only the non-empty leaves is sufficient.

After determining the covering regions that define the assumed partition, the total area is divided in an assumed manner among all the processors. Let n_c denote the number of covering regions. Each processor knows the dimensions of the covering regions as well as the number of grid boxes in each. Processors are assigned to the covering regions in proportion to the fraction of the total number of grid boxes contained by each region. In other words, regions containing many grid boxes are assigned more processors than regions containing few grid boxes. Assignment is done in this manner to load balance the number of boxes in each processor's area in the assumed partition, and therefore the number of times a processor is contacted in step 1.2 (described in Section 5.1.2). Note that in general there are many fewer covering regions than processors ($n_c \ll P$), and each processor is assigned ownership in only one covering region. Within each covering region, each processor's area of the assumed partition is roughly equal and determined by an $O(1)$ function in the exact manner described in Section 5.1.1. Therefore, the data required for the global assumed partition for each of the n_c covering regions are the covering region's extents, the range of processor ids assigned to that region, and the number of divisions in the x, y, and z dimensions.

Figure 11 demonstrates the divisions in the assumed partition for sixteen processors. In our implementation, processors 0-7 are assigned to the covering region in the lower left corner and processors 8-15 are assigned to the region in the top right. Note that each of the regions is divided into nine areas, even though only eight processors are assigned to each covering region. As a result processor 0 owns two areas in the left region and processor 8 owns two areas in the right region. As mentioned in Section 5.1.1, the assignment of areas is determined according to the number of divisions and number of processors.

5.3 Costs

As we have seen, the implementation of Algorithm 1 for the `Struct` interface (Algorithms 2, 3, and 4) is more complicated than for the `IJ` interface. Similarly, the costs for the algorithm are considerably more difficult to analyze. To do this, we first need to characterize the grid and its distribution in a way that is both easy to understand yet helpful in the analysis. First, we make some simplifying assumptions.

> *Assumptions:* Assume that the d-dimensional grid bounding box has sides of equal length, $V_b^{1/d}$. Assume that the covering regions are perfectly load balanced so that each process owns exactly the same volume, V_c/P, of the assumed partition. Finally, we assume without loss of generality that
>
> $$\gamma \geq \frac{V_g}{V_b}, \tag{2}$$
>
> since Algorithm 5 will determine the same covering regions for all values of γ smaller than this.

There are two main aspects of the grid that are important to characterize: its overall "shape" and its underlying decomposition.

Grid shape (cover mesh): Consider a uniform mesh over the bounding box with mesh spacing

$$h = \frac{V_b^{1/d}}{2^L}, \tag{3}$$

where $L \geq 1$ is an integer specified below. Define the cover mesh to be the set of all mesh cells that intersect the grid, and define V_h to be the volume of the cover mesh. As an example, the cover mesh in Figure 9 for $L = 2$ consists of the two 2×2 blocks of cells in the lower-left and upper-right corners. It is clear that $V_h \to V_g$ as $h \to 1$ (as L increases, 2^L approaches the number of mesh cells). In this section, we define L such that the mesh spacing h is maximized and the following is satisfied:

$$\frac{V_g}{V_h} \geq \gamma. \tag{4}$$

In other words, we define the cover mesh to be the coarsest uniformly-spaced mesh that covers the grid with accuracy prescribed by γ. This cover mesh provides a characterization of the overall shape of the grid that is both simple to understand and easy to relate to the actual cover regions in the algorithm. We will comment more on this particular characterization at the end of this section.

Grid decomposition: Define $w_{\min}$ to be the minimum box width over the boxes in the grid. Also, define $v_{\max}$ to be the maximum volume over the local grids stored on each processor. In a well load-balanced setting, all boxes are equal-sized cubes with sides of length $w_{\min}$, and each process "owns" the same amount of the grid given by $v_{\max}$.

We now introduce a few additional quantities that are useful in the analysis. Note that we can relate all of these quantities back to the above grid characterization. We let n_c denote the number of covering regions and n_h denote the number of cells in the cover mesh. Recall that both the cover mesh and the cover regions are constructed by recursively subdividing the grid bounding box by two in each dimension. However, because we only subdivide leaves when necessary during the construction of the covering regions (line 5.3), and because we also shrink the leaves (line 5.7), we have that $n_c \leq n_h$ and $V_c \leq V_h$. From (3), (4), and (2) we have

$$n_c \leq n_h = \frac{V_h}{h^d} = 2^{Ld}\left(\frac{V_h}{V_b}\right) \leq \left(\frac{2^{Ld}}{\gamma}\right)\left(\frac{V_g}{V_b}\right) \leq 2^{Ld}. \tag{5}$$

We let $b_{\max}$ denote the upper bound for the maximum number of boxes on a processor given by

$$b_{\max} := \frac{v_{\max}}{w_{\min}^d}. \tag{6}$$

In what follows, we will analyze the costs for Algorithm 1 for the `Struct` interface. The costs will be written in terms of the quantities n_c, $b_{\max}$, and L, which are either directly related to the grid characterization or can be related through (5) and (6). The quantities γ, P and d will also appear in the analysis.

Cost for Algorithm 2 (and Algorithm 5)

Algorithm 2 for constructing the assumed partition consists of two lines. Line 2.1 requires either determining the extents of the bounding box (via MPI_Allreduce) which costs $O(\log(P))$ communications, or determining covering regions via Algorithm 5. Recall that the covering regions result from recursively subdividing the bounding box and retaining non-empty regions until $\frac{V_g}{V_c} \geq \gamma$ is satisfied. The lines in the algorithm that call MPI_Allreduce (lines 5.2 and 5.5) each require $O(\log(P))$ communications with at most $O(n_c)$ data. The remaining lines all require gathering statistics on local boxes for each covering region at a cost bounded by $O(n_c b_{\max})$ computations. The total number of iterations is given by the depth of the tree, which is bounded by L. Hence, the total computational cost of Algorithm 5 is at most $O(L\log(P))$ communications with a total of $O(Ln_c)$ data and $O(Ln_c b_{\max})$ floating-point operations. Storage costs are at most $O(n_c)$. Line 2.2 assigns ownership of processors to covering regions and determines the number of divisions in each dimension for each covering region. The computational cost for this line is $O(n_c)$.

Cost for Algorithm 3

Here we examine the cost to distribute the grid boxes to the correct assumed partition processors via Algorithm 3. Each call of the assumed partition function requires $O(n_c)$ computations. Therefore, $O(n_c b_{\max})$ computations are required for line 3.1. In line 3.2, the number of processors that are contacted is at most $b_{\max}$ and is fewer if the local grid boxes are spatially clustered. To analyze line 3.3, temporarily denote the volume of each processor's assumed partition by v_c. From the assumptions and (4),

$$v_c = \frac{V_c}{P} \leq \frac{V_h}{P} \leq \frac{V_g}{\gamma P} \leq \frac{v_{\max}}{\gamma}.$$

From this and the fact that $w_{\min} \leq v_c^{1/d}$, we see that the number of boxes that intersect a processor's assumed partition is bounded by

$$\left(\frac{v_c^{1/d}}{w_{\min}} + 1\right)^d \leq \left(2\frac{v_c^{1/d}}{w_{\min}}\right)^d \leq \left(\frac{2^d}{\gamma}\right)\left(\frac{v_{\max}}{w_{\min}^d}\right) = \left(\frac{2^d}{\gamma}\right) b_{\max}. \qquad (7)$$

Hence, the number of receives and storage in line 3.3 is at most $O((2^d/\gamma)b_{\max})$. In addition, as previously mentioned, a termination detection mechanism is also required for lines 3.2 and 3.3, and this mechanism adds an additional $O(\log(P))$ cost for communication.

Cost for Algorithm 4

We now examine the cost of calculating neighbors in Algorithm 4 using the distributed directory. We denote the maximum number of neighbors a processor may have by q (and assume that q is independent of P). Line 4.1 involves calls to the assumed partition function for each local box, which costs $O(n_c b_{\max})$ computations. Line 4.2 costs $O(q) + O(\log(P))$ communications because a termination detection routine is required. Storage requirements depend on q and the amount of data in the distributed directory. In the worst case, if a processor's local boxes are not spatially clustered, the number of potential neighbor boxes is $O(q\ b_{\max})$. Then the comparison in line 4.3 requires $O(q\ b_{\max}^2)$ computations.

Overall Costs

If we assume that the quantities n_c, $b_{\max}$, and L are all independent of P, then the above analysis shows that Algorithm 1 for the `Struct` interface has communications and computations costs that depend only logarithmically on P. Furthermore, storage costs have no dependence on the number or processors. In Section 5.4, we give experimental results that support these estimates.

We note that the assumption of perfect load balancing for the covering regions is reasonable because the construction of the covering regions is wholly determined by the assumed partition algorithm. For the particular algorithm described in the previous section, a processor may have a covering region that is at most twice the size of that of another processor. This factor of two enters the cost analysis in (7) and does not affect the overall cost estimate.

It is clear that the grid characterization plays a crucial role in the above analysis. In particular, it is important that L is not too big and that the grid is well load-balanced. We can somewhat control the size of L by choosing a γ that is not too close to one. However, from the analysis (e.g., see (7)), we also see that we should not choose γ too close to zero.

Because the grid characterization uses fixed-size mesh cells in its cover mesh, it does not take into account the shrinking feature in line 5.7 of Algorithm 5. As a result, the constants in the analysis can be much larger than they are in practice. For example, consider the 2d grid composed of two small boxes, one in the lower-left corner of the grid bounding box and the other in the upper-right. Here, L can be made arbitrarily large by increasing the distance between the two grid boxes. However, for all of these grids, Algorithm 5 would find at most two cover regions with a tree depth of at most two. To improve the analysis, we might try to define a different grid shape characterization (remembering that it should be both simple to understand and easy to relate to the cover region), but this does not seem to be a straightforward task.

Even though the above analysis is sometimes a crude upper bound on the performance of Algorithm 5 (it can also be sharp), it helps to illustrate possible algorithmic improvements that might be made. Note that the only way to reduce V_c is for leaves to either shrink as a result of line 5.7 or be eliminated altogether in line 5.6. The only way to eliminate leaves is for the midpoint of its parent leaf to land outside of the grid. If the midpoint of the parent leaf lands inside of the grid, then all of its 2^d children will have non-empty grid intersections (note that it might still be possible to shrink some of these child leaves). Because of this, it is easy to construct grids for which Algorithm 5 will perform poorly.

For example, consider the 1d grid constructed by taking B equal-sized grid boxes laid out next to each other, then removing the second one from the left. This results in a grid bounding box that is full except for a small gap near the left side. If γ is large enough, Algorithm 5 will produce $n_c = O(\log(B))$ cover regions with a tree of depth n_c. The optimal choice for this example would be two cover regions. Modifying the algorithm to do a better job with this particular example is possible, but modifying it to handle any grid in a manner that is independent of B or P is not possible.

There are some algorithmic modifications that would improve the likelihood of better performance. For example, we could change the midpoints used in Algorithm 5 to divide leaves into children. To do this, in line 5.1, each processor could also determine a "closest midpoint" for each leaf region intersected by its local grid boxes. These midpoints must land outside of the local grid boxes. In line 5.1, this information would be compared to closest midpoint data from other processors in the MPI_Allreduce to determine global closest midpoints to use for dividing the leaves. Depending on the distribution of boxes in the 1d example above, this may reduce the depth of the tree, even to the optimal depth of two.

As illustrated in the simple example above, the performance of Algorithm 5 depends heavily on the grid layout, and for some (obscure) grids, its performance can even depend linearly on P. For most grids, this will not be the case (see the experiments in Section 5.4), but we mention one case that may occur in practice. The grids supported by the `Struct` interface often represent grid levels in structured adaptive mesh refinement codes. In the case of a boundary layer problem or a problem with shocks, the fine grid levels that capture these features are long and thin, and get thinner as the mesh spacing is refined. Capturing these features with fixed accuracy γ may require more and more cover regions as the mesh is refined. Since P is related to the mesh spacing (larger problems require more processors), the performance of the algorithm could also depend on P, in the worst case, linearly.

5.4 Experimental results

We compare the new assumed partition method with the the old method that stores the actual partition, as described at the start of Section 5. In particular, we report the time to determine the neighbor boxes and communications required for a matrix-vector multiply in the `Struct` interface for each method. Again we provide results from runs on BlueGene/L for $P = 4^3$, 6^3, 8^3, 10^3,12^3, 14^3, 16^3,18^3,20^3, 22^3, 24^3, 25^3, 28^3, and 32^3 processors. We contrast the performance of the two methods on four different problems whose grids vary in degree of difficulty for the assumed partition method.

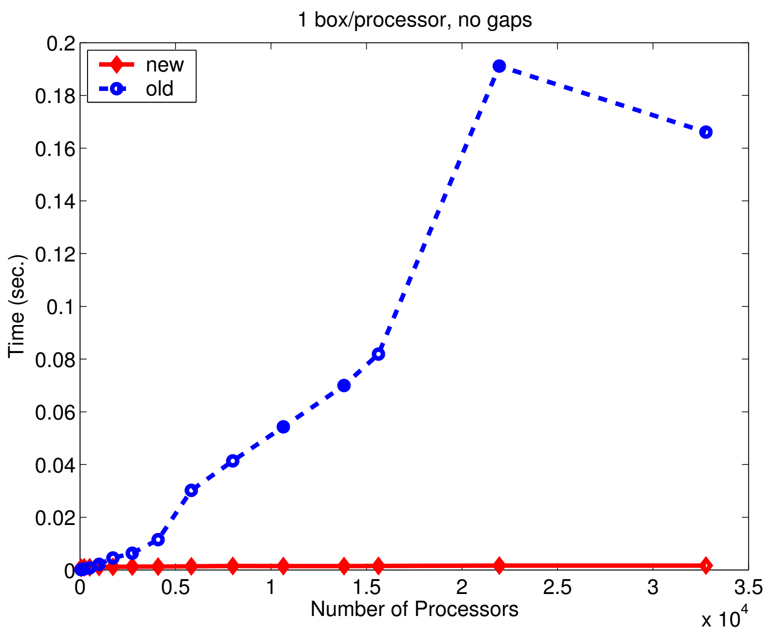

Fig. 12. A comparison of the new and old implementations for determining the neighbor communications for the structured-grid interface. Here each processor owns one large grid box and the bounding box has no empty space.

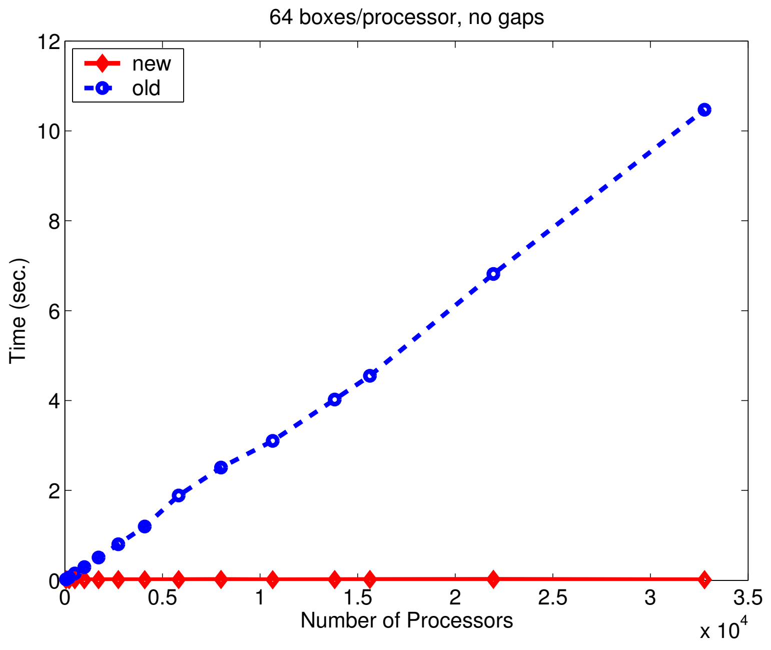

Fig. 13. A comparison of the new and old implementations for determining the neighbor communications for the structured-grid interface. Here each processor owns 64 grid boxes and the bounding box has no empty space.

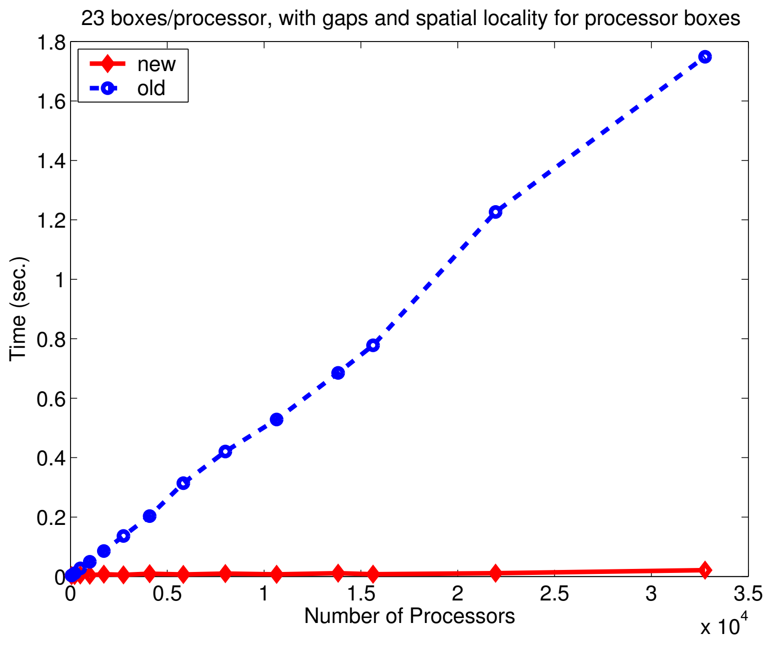

Fig. 14. A comparison of the new and old implementations for determining the neighbor communications for the structured-grid interface. Here each processor owns 23 grid boxes and the bounding box is not completely full of grid boxes.

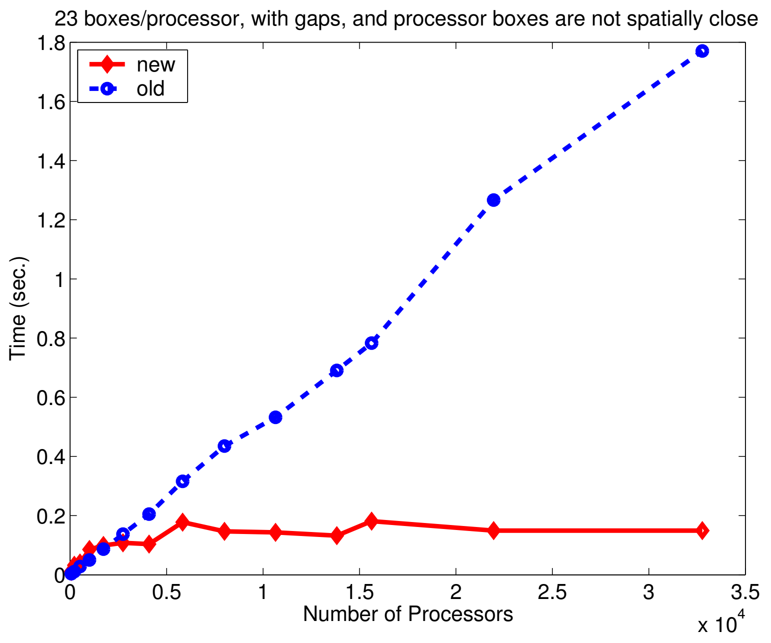

Fig. 15. A comparison of the new and old implementations for determining the neighbor communications for the structured-grid interface. Here each processor owns 23 grid boxes and the bounding box is not completely full of grid boxes. In addition, grid boxes belonging to a single processor are not adjacent.

For the grid in the first problem, each processor owns one grid box and the entire bounding box is covered with grid boxes such that there are no gaps in the bounding box. Results for this problem are given in Figure 12. Because of the absence of gaps and the small number of grid boxes per processor, this problem is the easiest of the four problems for both implementations. The new assumed partition algorithm scales very nicely for this problem.

The second problem in Figure 13 is more challenging in terms of computation and storage as each processor own 64 boxes. Again, this problem has no empty space in the bounding box. Note that the times for both methods on this problem are slower due to the larger number of boxes per processor. In this case, the storage savings for the assumed partition method is also particularly important.

The grids for the two problems in Figures 14 and 15 both have a lot of empty space in the bounding box and 23 grid boxes per processor. As mentioned previously, problems such as these with gaps are challenging in terms of load-balancing. For the problem in Figure 14, the gaps are spread throughout the domain, but grid boxes belonging to a single processor are spatially close to each other. However, for the problem Figure 15, the grid boxes belonging to each processor are not spatially local. In fact, in this example, the grid boxes are arranged in 23 blocks across the diagonal of the bounding box (a cube), and each processor owns one grid box in each of the 23 diagonal blocks. This problem is very challenging for the assumed partition algorithm, as evident in the results in Figure 15. For this problem, the crossover point at which the assumed partition method is faster than the old method occurs at a higher number of processors than with the previous three example problems.

The results for all four example problems indicate that the new assumed partition method has better scaling properties than the old method for even "difficult" grids. In fact, we expect the benefit of the assumed partition method to be even greater for larger numbers of processors. Furthermore, the savings in storage is significant, particularly because multigrid solvers require multiple grids for a single problem.

6 Exchanging data between processors

When implementing the assumed partition algorithm for both the `Struct` and `IJ` interface, it is necessary for processors to exchange data with each other in a way that the communication patterns are not known. In other words, processors know which processors they need to contact and what type of information they want to receive from those processors, but do not know how much data they will actually receive. In addition, processors do not know whether they will be contacted by any other processors requesting information or how much information will be requested of them. The need for this type of communication may arise frequently in parallel algorithms. As described in [5] in the context of possible improvements to the `IJ` interface algorithms, the use of MPI_IPROBE is typically required when processors do not know who they are communicating with. A termination detection routine is then required to determine when to stop probing for messages.

We created a function that allows for such undetermined communication patterns. We refer to the function as `ExchangeData()`. This function allows a processor (i) to "contact" a list of processors with a message containing a variable amount of data. The processors in the list do not know that they will be contacted by Processor i. These processors must then send a "response" message back to Processor i, and the response message can also contain any amount or type of data. Essentially, the user gives `ExchangeData()` a list of processors and corresponding lists of data (of any type) to send to each processor. The function returns a list of data (the "responses") from each of the contacted processors. The user must also supply a function that dictates how to create an appropriate response message based on the contact message (called the `FillResponse()` function).

For example, consider Algorithm 4 for a particular Processor p. In line 4.1, Processor p determines which processors may know about its neighbors, and creates a list of these processors to contact. Then in line 4.2, the `ExchangeData()` function is called. Processor p contacts the processors determined in line 4.1, and among them is Processor k. The contact messages do not need to contain any data in this example. When Processor k is contacted by Processor p, it must send back a response message to Processor p. The `FillResponse()` function is responsible for taking the contact message from Processor p and creating the appropriate response message. In this case, the response message contains all of the boxes in Processor k's assumed partition. Observe that Processor k does not know that it will be contacted by p, and Processor p does not know how much data it will receive from Processor k. In addition Processors p and k may each be contacted by any number of other processors requesting information.

Because of the general applicability of such an algorithm, we provide pseudocode for the `ExchangeData()` function in Figure 16. The algorithm begins by each processor determining its location in a virtual spanning tree of processors. A processor then posts non-blocking sends to each processor it needs to contact and post non-blocking receives for the responses it expects to get. The processor may not know the size of the response, so a response size limit is chosen and memory is allocated for responses of that size. This size limit does not need to be an absolute maximum. If a response needs to be sent that exceeds the limit, the message is broken into two messages. The second message can be completed after termination has been detected since the size of the data in the second message will be known at that time (it is prepended to the first response message).

Essentially a processor continually probes for contact messages until it is told by its parent processor to terminate. When a processor receives a contact message, it calls `FillResponse()` to create an appropriate response for that contact message. It then sends the response in one or two messages depending on the size of the response. If no contact messages are found when probing,

```
1.  determine parent and children processors
2.  send non-blocking contact messages
3.  post non-blocking receives for the response messages
4.  while (not time to terminate){
5.      check for contact messages
6.      while (contact message is waiting) {
7.          receive contact message
8.          call user-defined FillResponse()
9.          send response buffer back to the contacting processor (prepend the size)
10.         if (size of response buffer exceeds predefined limit){
11.             send a second message with the remainder of the response data
12.         }
13.         check for contact messages
14.     }
15.     if (all response messages have been received) {
16.         responses_received = TRUE
17.     }
18.     if (termination messages have been received from all children) {
19.         children_terminate = TRUE
20.     }
21.     if (responses_received AND children_terminate){
22.         send a termination message to parent
23.     }
24.     if (received a termination message from parent) {
25.         send termination messages to children (blocking)
26.         break;
27.     }
28. }
29. Receive the remainder of any messages that exceeded the response size limit
```

Fig. 16. Pseudo-code for `ExchangeData()` function

the processor checks to see if it has received responses to all of its contacts. If so, and if the processor has no children in the tree (i.e., it is a leaf processor), it sends an upward termination message to its parent in the tree. A non-leaf processor must wait for termination messages from all of its children before sending a termination message to its parent. When the root node receives termination messages from all of its children (and has itself received all of its response messages), it then starts a downward termination sweep by sending notice to terminate to its children. When a processor receives a downward termination message, it stops probing for contact messages and sends termination messages to its children. After a processor ceases probing for messages, it receives any second response messages that occurred due to data that exceeded the predefined response size limit.

Note that because we require processors to send response messages back to each processor that contact it, the termination procedure used in this context is simpler than is necessary to detect the termination of a general distributed computation. Without this requirement, we would need at least four traversals of the spanning tree instead of the two required for this algorithm (for example, see [1]). The cost of this algorithm in terms of P is $O(\log(P))$ for the two sweeps of the virtual spanning tree.

7 Concluding remarks

We explored determining inter-processor communications required by a computation in a manner that does not require storing a global partition of the data. Our experiments indicate that our assumed partition strategy is effective in the context of the *hypre* software package on a large number of processors. We are optimistic that our algorithm will scale up to 100,000 processors. This assumed partition idea has general applicability to many situations in parallel computing that utilize MPI_ALLGATHERV.

In addition, we expect that some of the challenges of determining the assumed partition function for the structured-grid interface and determining an algorithm to accommodate unknown communication patterns are common to other applications on large numbers of processors as well.

Acknowledgments

We appreciate the contribution of all of the *hypre* library developers, and we thank Barry Lee for his input regarding the structured-grid interface algorithm.

References

[1] A. Baker, S. Crivelli, and E.R. Jessup. An efficient parallel termination detection algorithm. Technical Report CU-CS-915-01, Department of Computer Science, University of Colorado at Boulder, 2001.

[2] Bluegene/L. http://www.llnl.gov/asci/platforms/bluegenel/, 2004.

[3] R. Falgout, J. Jones, and U.M. Yang. Conceptual interfaces in *hypre*. *Future Generation Computer Systems, to appear*, 2005. Also available as LLNL technical report UCRL-JC-148957.

[4] R.D. Falgout, J.E. Jones, and U.M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A.M. Bruaset, P. Bjørstad, and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers, to appear*. Springer-Verlag, 2005. Also available as LLNL Technical Report UCRL-JRNL-20545.

[5] R.D. Falgout, J.E. Jones, and U.M. Yang. Pursuing scalability for hypre's conceptual interfaces. *ACM Transactions on Mathematical Software*, 31(3):326–350, 2005.

[6] Robert D. Falgout and Ulrike Meier Yang. *hypre*: a library of high performance preconditioners. In P.M.A. Sloot, C.J.K. Tan., J.J. Dongarra, and A.G. Hoekstra, editors, *Computational Science - ICCS 2002 Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641. Springer–Verlag, 2002.

[7] hypre. High performance preconditioners, 2005.
http://www.llnl.gov/CASC/linear_solvers/.

[8] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43:207–221, 1998.

[9] A. Pinar and B. Hendrickson. Communication support for adaptive communication. In *Proceedings of the 10th SIAM Conference on Parallel Processing*, 2001.

[10] A. Pinar and B. Hendrickson. Interprocessor communication with limited memory. *IEEE Transactions on Parallel and Distributed Systems*, 15:606–616, 2004.