UCRL-JRNL-216580

Parallel Clustering Algorithms for Structured AMR

Brian T.N. Gunney, Andrew M. Wissink, David A. Hysom

October 27, 2005

Journal of Parallel and Distributed Computing

Parallel Clustering Algorithms for Structured AMR

Brian T. N. Gunney *, Andrew M. Wissink, David A. Hysom

Center for Applied Scientific Computing
Lawrence Livermore National Lab

Abstract

We compare several different parallel implementation approaches for the clustering operations performed during adaptive gridding operations in patch-based structured adaptive mesh refinement (SAMR) applications. Specifically, we target the clustering algorithm of Berger and Rigoutsos (BR91), which is commonly used in many SAMR applications. The baseline for comparison is a simplistic parallel extension of the original algorithm that works well for up to $O(10^2)$ processors. Our goal is a clustering algorithm for machines of up to $O(10^5)$ processors, such as the 64K-processor IBM BlueGene/Light system. We first present an algorithm that avoids the unneeded communications of the simplistic approach to improve the clustering speed by up to an order of magnitude. We then present a new task-parallel implementation to further reduce communication wait time, adding another order of magnitude of improvement. The new algorithms also exhibit more favorable scaling behavior for our test problems. Performance is evaluated on a number of large scale parallel computer systems, including a 16K-processor BlueGene/Light system.

Key words: parallel computing, high-performance computing, task-parallel, asynchronous, clustering, adaptive mesh refinement

1 Introduction

Adaptive mesh refinement (AMR) is an approach for discretizing and solving science and engineering problems on computational meshes. It is useful

* Corresponding author. Lawrence Livermore National Lab, Mail Code L-561, P.O. Box 808, Livermore, CA 94550-0808. Phone (925) 422-4115. Fax (925) 423-6374.
Email addresses: gunneyb@llnl.gov (Brian T. N. Gunney), awissink@llnl.gov (Andrew M. Wissink), hysom@llnl.gov (David A. Hysom).

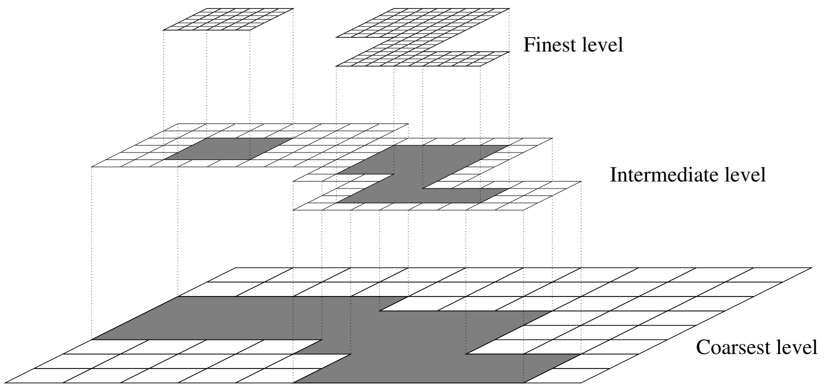

Fig. 1. Simple 2D structured AMR mesh hierarchy with three levels of refinement. Shaded cells are those on which finer grids are overlaid.

for problems with localized fine-scale regions in the computational domain. By placing the mesh points and computational efforts where they are needed most, AMR can require far fewer computational resources than would be required by using uniformly fine meshes. In a dynamic problem where solution features move and appear or disappear, the AMR mesh changes to adapt to the changing features. Grid points can be automatically inserted and removed where needed.

Patch-based structured AMR (SAMR) is an approach originally proposed by Berger, Oliger, and Colella (BO84; BC89) that composes the adaptively refined mesh by overlaying successively finer individual structured grids, known as patches, where higher resolution is needed (Figure 1). The mesh is composed of a sequence of levels, each having greater resolution than the previous. As shown in Figure 1, fine grids lines are aligned with coarse grid lines at fixed intervals. Although other approaches to AMR, such as tree-based block-structured AMR, do not require the clustering algorithm, the algorithm is widely used in patch-based SAMR.

The clustering algorithms described in this paper are used during the dynamic gridding steps in the SAMR applications. SAMR mesh adaptivity involves replacing the current level with an updated one and transferring data to the new level. Refinement involves adding a finer level to the hierarchy, overlaying the finest existing level. Each of these operations builds new levels. Clustering generates the initial set of *boxes* from which to build the new levels. The boxes are logically rectangular, defined by the indices of their lower and upper corners.

To create a new level, the application must determine what regions the new level should cover and generate the structured grids to cover them. A feature detection scheme specific to the problem is commonly used to “tag” cells that the new level should cover, e.g., it finds cells that contain large gradients or numerical error. The clustering algorithm then computes a set of boxes that cover the tagged cells. Each box encloses a cluster of tagged cells. The set of boxes may be further processed (e.g., to enforce size constraints). The grids in the new level are then created from the final set of boxes.

The algorithm proposed by Berger and Rigoutsos (BR91) is widely used for clustering in SAMR operations. Often referred to as the Berger-Rigoutsos algorithm, it is generally quite fast and works well in serial. A simplistic parallel version is sufficiently fast for low and moderate numbers of processors. However, its scaling properties can be poor, and hence it can be expensive for large problems run on many processors (WHH03). This paper describes new algorithms, based on that of (BR91), that have improved scaling properties on large parallel computers.

It is important to note that clustering is only done when a new grid level is generated (during refinement and regridding) so that the frequency of grid adaptation affects the overall clustering cost. For example, a problem with static adaptivity will only perform clustering once to generate the initial refined grid, whereas a fully adaptive problem will cluster whenever the grid changes. While it is possible to reduce the overall clustering costs in a computation by reducing the frequency with which grids are adapted, this approach introduces other overheads. For instance, the refinement region must include extra buffer zones to ensure that dynamic features in the solution do not move beyond the refined region between regridding steps. In this work we focus on improving the efficiency and scalability of the clustering operation itself, independent of how often it is applied.

Clustering is one of several steps typically used in grid adaptation. While the overall speed of an SAMR simulation depends on other steps, numerical algorithms, the problem being solved and other parameters, we focus on the clustering step in this paper.

The algorithms described are implemented in the SAMRAI framework (HK02; SAM04) developed at Lawrence Livermore National Laboratory. The concepts developed should readily apply to other SAMR implementations.

The remainder of the paper is organized as follows. Section 2 reviews the original Berger-Rigoutsos clustering algorithm. Section 3 discusses two single-task parallel implementations for the Berger-Rigoutsos algorithm and their performance. Section 4 introduces a task-parallel approach and two algorithms using this approach. The performance of the task-parallel algorithms are given and compared to the single-task algorithms. Concluding remarks are given in the last section.

2 Previous Clustering Algorithms for SAMR

In 1991, Berger and Rigoutsos (BR91) considered a number of general variations of *bottom-up* and *top-down* clustering algorithms for SAMR. Bottom-up variations start with seed points computed from a tagged-cell pattern and build boxes around the seed points using variances of the k-means partitioning algorithm (And73; Har73). The top-down algorithm places all tagged cells into an initial single box, then splits the initial and subsequent boxes to eventually form the final set of boxes (see Figure 2). These variations are forms of hierarchical clustering. Each hierarchical clustering corresponds to a tree, known as a *dendogram*, with the initial grouping at the root and the final groupings at the leaves (DH73) (see Figure 3).

Berger and Rigoutsos judged their top-down algorithm as best. In this algorithm the criteria for whether and where to split a box are based on ideas from edge detection algorithms (MH80), using *signatures*. Signatures for a d-dimensional box are computed by projecting each tag to the d axes and summing the number of tags at each point on the axes (see Figure 2a). The signatures form one-dimensional descriptions of the tag distribution in higher-dimensional boxes.

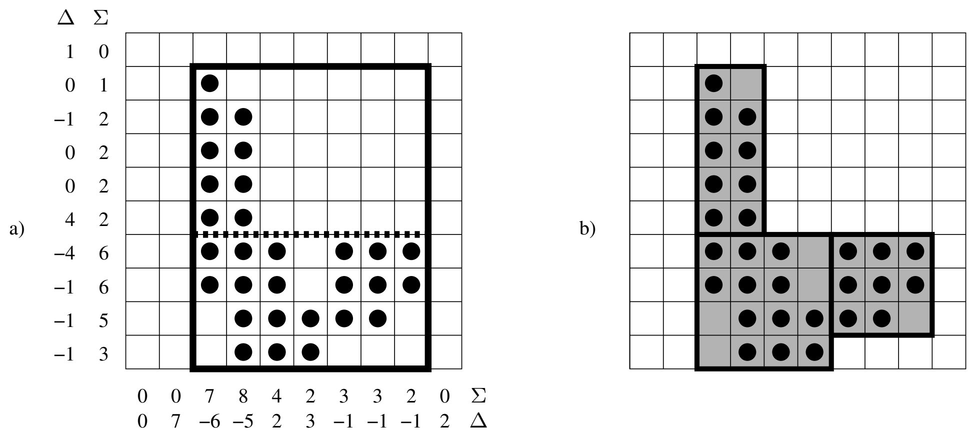

Fig. 2. A top-down hierarchical clustering example using the Berger-Rigoutsos Algorithm (BR91). Tagged cells are marked by dots. a) Signatures, bounding box and cutting plane used by the Berger-Rigoutsos algorithm. Σ is the signature. Δ is the undivided Laplacian of the signature; $\Delta_i = \Sigma_{i-1} - 2\Sigma_i + \Sigma_{i+1}$. Heavy-lined box is the bounding box of the clustered tags. Dashed line is the location of the cutting plane based on the inflection point criterion (see text). b) Resulting box clusters.

Signatures are used to decide whether and where to split a candidate box in the top-down algorithm, according to the following criteria:

(1) A box is split if it does not meet a preset *efficiency threshold*. Efficiency

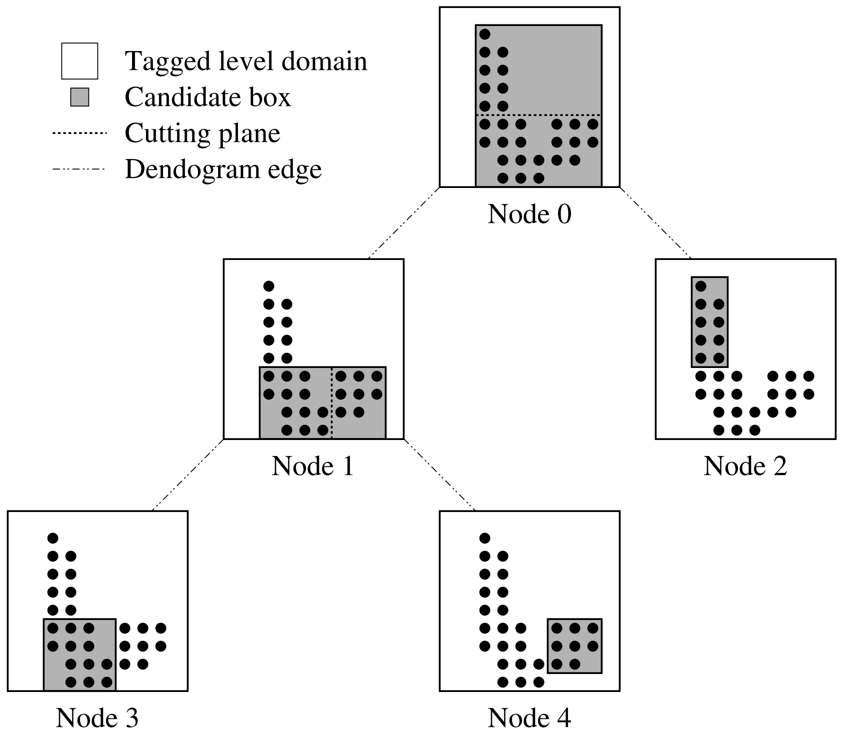

Fig. 3. Dendogram corresponding to the clustering example in Figure 2. The edges in the dendogram connects parents to their children. Node 0 is the initial cluster. Node 2, 3, and 4 constitute the final clusters.

is defined as the ratio of the number of tagged cells in the box to all cells in the box. It controls the degree of extra refinement in untagged regions.

(2) The first preferred location to split a box is at a hole, or zero value, in a signature.

(3) If no hole is found in the signature, the next preferred cutting plane location is at an inflection point (zero-crossing of the second derivative) of a signature. Figure 2a shows the second derivative of the signature approximated by the undivided Laplacian Δ.

In step (1), if the efficiency threshold is set to 1, every new box constructed will contain only tagged cells. While this may seem desirable, in practice it leads to construction of many small boxes, a process that introduces other overheads. It is generally most efficient to set the threshold to something slightly less than 1, which reduces the number of boxes but includes some cells in the refined region that were not originally tagged to be refined.

Rantakokko (Ran03) described a similar top-down algorithm but (optionally) with specific criteria for choosing the next box to split. Whereas Berger-Rigoutsos equivalently chooses the next box from a breadth-first search of the current dendogram leaves, Rantakokko chooses, from all current dendogram leaves, the one that has the most untagged cells. This search introduces a dependency among the branches of the dendogram which would inhibit the task-parallel approach we are presenting (though other concepts we introduce would still be usable). In this work, we focus on the original Berger-Rigoutsos criteria which remains in wide use.

Algorithm 2.1 is a recursive version of the Berger-Rigoutsos algorithm used in SAMRAI. We denote it as a *single-task* algorithm to differentiate it from others that we present later. Each recursive call takes a single candidate box and builds up a list of non-overlapping boxes that, collectively, cover the tags in the candidate box. If the candidate box is split the algorithm is called recursively for the two child boxes. We add at line (iii) an additional step (not present in the Berger-Rigoutsos algorithm) of checking the *combined efficiency* of the child boxes. The combined efficiency is defined as the ratio of tagged cells summed in the left and right[1] boxes to the sum total of cells in those boxes. This ratio is defined only if the left and right boxes are not split. If the combined efficiency does not improve by the factor β over the parent's efficiency, the parent's box is not split. In practice, we use $\beta \approx 1.2$. The goal of this step is to avoid generating two smaller boxes if the potential gain in efficiency is minimal. The recursive structure of Algorithm 2.1 provides the parent-children and sibling relationships needed for the check on combined efficiency.

[1] We use the terms *left* and *right* in an imagistic sense only, to aid comprehension. In 3D boxes may be split into two parts with respect to a plane orthogonal to either the x, y, or z axes.

Algorithm 2.1: SingleTaskBergerRigoutsos(C, b)

```
comment: Compute a set of boxes $C$ (the cluster), starting with candidate box $b$

compute signatures of $b$                                        (i)
$b \leftarrow$ bounding box of tags in $b$
if efficiency($b$) $\geq$ threshold
  then $C \leftarrow \{b\}$
  else
    split $b$ in two to create boxes $b_L$ and $b_R$             (ii)
    SingleTaskBergerRigoutsos($C_L, b_L$)
    SingleTaskBergerRigoutsos($C_R, b_R$)
    if length($C_L$) $> 1$ or
       length($C_R$) $> 1$ or                                     (iii)
       combined efficiency $> \beta$ efficiency($b$)
      then $C \leftarrow \{C_L, C_R\}$
      else $C \leftarrow \{b\}$
```

We refer to the original Berger-Rigoutsos clustering method as a *single-task* algorithm since operationally it consists of a set of tasks that are executed sequentially (i.e., one at a time). With respect to Figure 3, the tasks are executed in the order: 0, 1, 3, 4, 2. Later, in Section 4, we will introduce *task-parallel* methods that permit some of these tasks to be performed simultaneously.

3 Single-task Clustering Algorithms

Clustering involves accumulating and operating on tagged cells from the overall problem domain. That is, the clustering operation initially considers the tagged cells over the entire level domain before breaking the domain up through recursive subdivisions. This top-down approach causes its cost to grow with problem size. Figure 4 demonstrates the typical growth in the clustering costs for a scaled problem. This problem advects a geometrically complex discontinuity across a domain with grid refinement following the discontinuity. It is desirable to keep adaptive gridding overheads to a minimum so that most of the computational work is concentrated on the numerical physics. The figure shows the costs of the numerical physics and the overheads introduced by clustering during the adaptive gridding operations for the application run on a production computer at LLNL. Note that on a small number of processors, the cost of clustering is more than two orders of magnitude smaller than the physics calculations, but on 1024 processors the clustering cost is several times more costly than the physics. This demonstrates how poor scaling of the clustering operations can severely impact the performance of fully adaptive SAMR applications on large parallel systems. Although the clustering cost is clearly problematic, it may not be the only slow algorithm. Overall performance of the SAMR application also depends on other steps that are outside the scope of this paper.

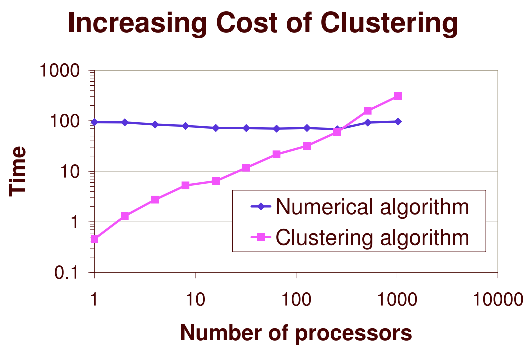

Fig. 4. Clustering vs. numerics costs for a weakly scaled adaptive advecting front problem (global problem size is proportional to the number of processors) . Poor scaling of the clustering algorithm causes it to become dominant on large numbers of processors. The clustering algorithm used was a simplistic extension of the serial algorithm, described in the text as the GlobalSumCluster algorithm.

Like most SAMR implementations (CGL+03; BB87; KB96; RBL+00), SAMRAI uses a domain decomposition single-program multiple-data (SPMD) parallelization approach. Individual structured grids that form a level are distributed to processors in such a way that the load is balanced. All data on a grid resides on the processor that owns the grid. However, each processor maintains an identical copy of the global list of boxes. When clustering with this distributed data model, communication is required for the signature computation. The simplest approach for this operation is to use standard global sum operations. Partial signatures are computed using the data local to each processor, then the global signature is formed by summing the contributions from all processors. Each processor then independently performs the identical computations as would occur if all data resided on a single processor.

We refer to this algorithm as GlobalSumCluster. This algorithm is identical to Algorithm 2.1 except that line (i) has been augmented with the communications described in the previous paragraph. (Although we have not found a description of GlobalSumCluster in the literature, we believe that others have used this algorithm in their AMR codes and therefore we make no claim as to its novelty.)

The GlobalSumCluster algorithm performs well as long as the number of processors is reasonably small (typically < 100). However, as problems are scaled up to run on large numbers of processors, performance degrades for two reasons. First the length of the signatures increases with problem size; that is, more data is exchanged through the global sums. Second, the number of global sums increases as we consider more box regions; in practice, the number of global sums increases roughly proportional to the problem size. The cost of each global reduction is theoretically[2] $O(NP\log(P))$, where N is the amount of signature data and P is the number of processors. This is the reason behind the rapid increase in costs demonstrated in Fig 4.

Analysis reveals that, as the recursion level in the Algorithm 2.1 increases, the box sizes and hence the number of tags evaluated by the algorithm quickly decrease. Table 1 shows how the number of participating processors changes with each recursion level in a sample problem using 128 processors. We say a processor *participates* if it owns data that contributes to the global sum that is needed to evaluate the candidate box. The table shows that the relevant tag data for most calls resides on a small subset of processors. In this example, we find that 74.5% of all global sum operations require data from 18 or fewer processors. Thus in the global sums, 110 of the 128 processors are performing needless communication.

These findings indicate that global sums are unnecessary for most signature construction operations. To achieve a more scalable approach, we redesigned the algorithm by replacing global sums with sums within each group of participating processors. We refer to the sum reduction within the group of participant as group-summing.

The GroupSumCluster algorithm is similar to Algorithm 2.1 but uses group-summing. If a processor is not the root processor (processor 0) and owns no patch that intersects box b (i.e., if it is not a participating processor), the routine returns immediately. The processors that remain then communicate amongst themselves to compute the signature for b. The signatures are assembled on the root processor using an all-to-one reduction operation (where *all* refers to the participating processors). The root processor performs the operations involving the location of box cut points and cutting boxes. In the box splitting step, line (ii) of Algorithm 2.1, the root processor broadcasts b_L and b_R to all processors participating in the current recursion level. Then, recursive calls are made on the participating processors only. Since the root processor participates in all recursive calls, when the recursion ends, it has the complete list of boxes.

[2] In practice, the performance may not conform to this model. Actual performance depends on software and hardware communications design.

Table 1
Breakdown of processor participation for an adaptive Sedov computation on 128 processors, for which the BR clustering algorithm was called over 500 times. The number of participating processors decrease rapidly with the depth of recursion. For example, 74.5% of the function calls require 18 or fewer processors.

Recursion level	Max number of participating processors at recursion level	Percent of function calls at or below recursion level
0	128	100
1	100	98
2	80	95.6
3	80	91.5
4	32	84.7
5	18	74.5
6	8	61.9
7	8	46.3
8	8	30.6
9	8	17
10	4	7.1
11	2	1.4

We tested the above approach using MPI communicators for the all-to-one reductions but found that hand-coded point-to-point MPI operations performed significantly better. This may arise from the need to synchronize processors each time an MPI communicator is formed, but we did not explore the issue in depth.

GroupSumCluster remains a single-task algorithm. The tasks are executed in the same order as before, but not all tasks are executed by all processors. As we will see in the next subsection, the resulting savings in communication cost has significant performance benefits.

3.1 Single-task Algorithm Timing Results

We evaluated the performance of our two single-task algorithms, GlobalSumCluster and GroupSumCluster, for a representative moving-grid configuration. Calculations were performed on three different parallel computer systems, with up to 16K processors.

Performance is measured for adapting to a 3D sinusoidal front advecting through the domain (see Figure 5). Although the physics calculation in this problem is absent, the moving grid is representative of a geometrically complex shock wave moving through a domain. We leave out the physics to isolate the performance of the clustering operation. The grid domain size is 24x16x16 cells on the coarsest level. In physical space, the domain is a right hexahedron with a corner at (0,0,0) and the opposite corner at (3,2,2). The front is initially centered at (0.5, 0, 0) and moves (0.02, 0.005, 0.005) each time step. The SAMR hierarchy has four levels, with a refinement ratio of two for each level. The patches in the hierarchy are distributed in parallel without regard to their spatial relationships, thus adjacent patches are *not* more likely to be on the same processor than separated patches. The sinusoidal front moves through the domain and the grids around it are re-generated from tagged cells five times. The global problem size remains the same as the number of processors increases (strong scaling). Where needed, three timings are done on each processor partition, and outlying dat (presumably caused by unrelated applications using the high-speed network at the same time) are omitted. The results presented are average times taken from the remainder.

Table 2
Computers used to evaluate performance.

Machine name	Model	Processor type	Max number of processors	CPUs per computing node	Network type
MCR	Linux cluster	2.4 Ghz Xeon	1024	2	Quadrics QsNet
Thunder	Linux cluster	1.4 Ghz Itanium 2	2048	4	Quadrics QsNet
BG/L	Linux cluster	700 Mhz PPC 400	16K	2	3D Torus

We present the results on two current LLNL production parallel platforms, MCR and Thunder, and the new BlueGene/Light (BG/L) system. All are distributed memory parallel, with modest shared memory parallelism. Table 2 shows the characteristics of the platforms. The "Max number of processors" shown in Table 2 is the number available for the experiments, not the absolute maximum on the platforms. On BG/L, there are 32K processors, but half of them are used as communication co-processors (the standard configuration). Although the BG/L system will eventually have 64K processors (and 64K communicaiton co-processors), the full machine was not yet fully available at

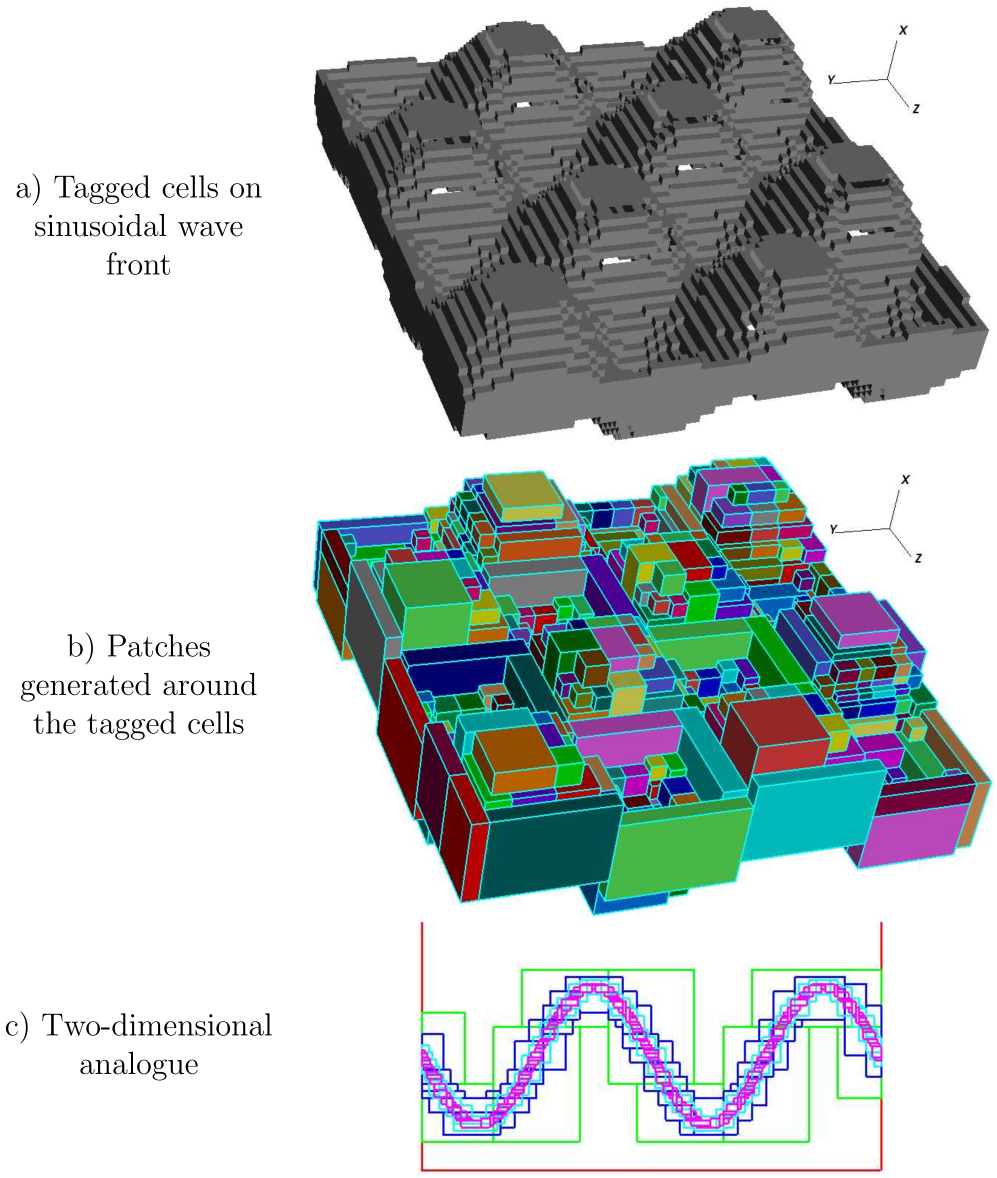
a) Tagged cells on sinusoidal wave front
x
Y
z
b) Patches generated around the tagged cells
x
Y
z
c) Two-dimensional analogue

Fig. 5. Sinusoidal front test problem. A sinusoidal front is advected primarily in the x-direction. The upper image shows the cells near the front–these are tagged for clustering. The middle image shows a typical set of boxes output by the clustering algorithm. Only one level in the AMR hierarchy is shown, with each box having a different color. To aid in visualizing the 3D problem, the lower image shows a 2D analogue showing the box outlines of 5 levels, each in a different color.

the time we conducted the experiments.

Figure 6 shows the clustering times for the two algorithms. Again, note that both algorithms produce identical results; they only differ in the way the communications are organized. The GroupSumCluster algorithm is significantly faster than the GlobalSumCluster algorithm. In the upper half of the range of number of processors shown, the line for the GroupSum-

Cluster algorithm rises less steeply, indicating that it is scaling better in the ranges shown.

Table 3 shows the actual timing results of the two approaches. The GroupSumCluster implementation is faster than the GlobalSumCluster implementation by 4.2x on 1K processors of MCR, by 13x on 2K processors of Thunder and by 7.4x on 16K processors of BG/L.

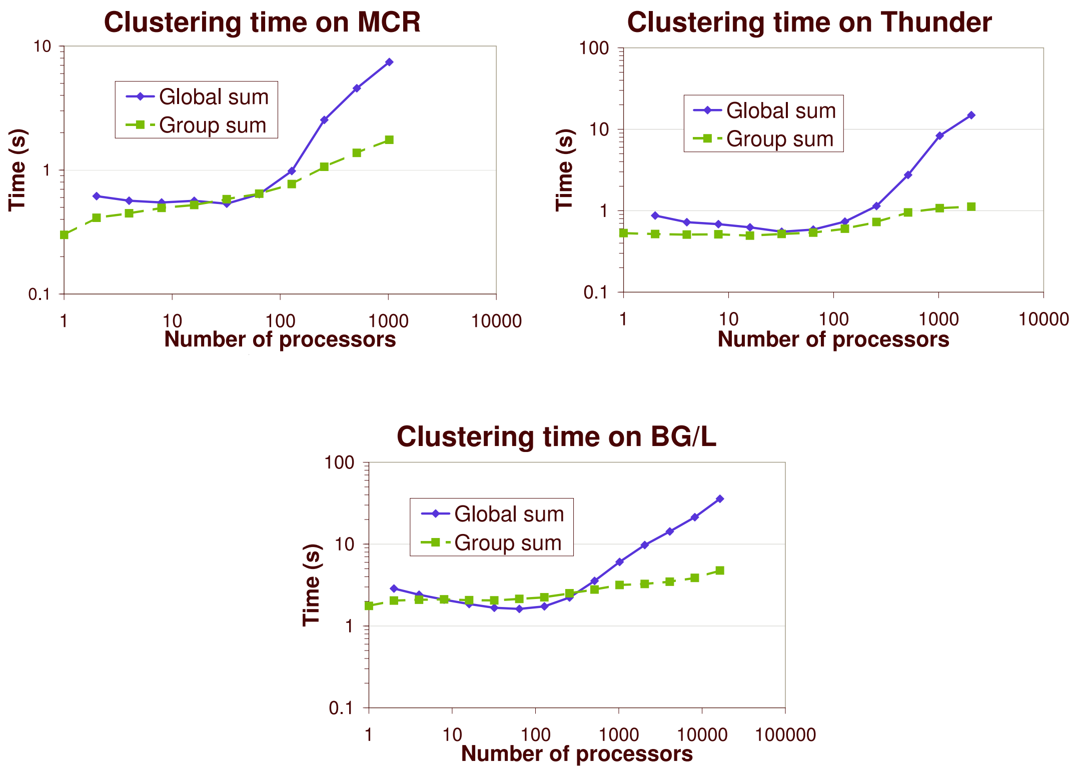

Fig. 6. Comparison of the clustering cost for an advecting sinusoidal front problem using the two single-task implementations, GlobalSumCluster and GroupSumCluster .

Table 3
Timing comparison of two single-task clustering algorithms. Algorithm GlobalSumCluster uses a simplistic parallelization (which accumulates the signatures using global sums). Algorithm GroupSumCluster uses the new group-suming approach.

Algorithm	Timings (seconds)		
	1K MCR	2K Thunder	16K BG/L
GlobalSumCluster	7.4	15	36
GroupSumCluster	1.8	1.2	4.8

4 Task-parallel Clustering Algorithms

Despite the performance gains achieved by using a better communication scheme in the single-task algorithm, scalability is still limited by two factors. First, communication costs, which dominate the overall time, worsen as the number of processors increases. Second, the algorithm relies on a single manager processor that coordinates contributions from multiple worker processors. Clearly, this will not be efficient on systems with $O(10^5)$ processors.

Fortunately, the Berger-Rigoutsos top-down hierarchical approach *does* present an opportunity to integrate task-parallelism into the logic. After a task's box is split, the left and right branches form tasks that are mutually independent, meaning that decisions for one task do not affect the decisions for the other. In Figure 3, for instance, tasks 1 and 2 are mutually independent, as are tasks 3 and 4. If task 1 is split before task 2 is completed, then tasks 2, 3 and 4 are all mutually independent. It is possible to work on the independent tasks concurrently rather than sequentially.

An algorithm taking advantage of the task independence can be created by adopting a task-parallel approach, where the operations on each independent box form a *task*. Each task is data-parallel, requiring communication within the group of processors participating in the task. Dependencies of the tasks can be determined by a parent-child relationship. Parents and their children are not independent because children cannot be created until a parent's box is split, and parent tasks depend on the results of its children so it can perform the combined efficiency check. However, different child tasks *are* independent of their siblings (and consequently of their cousins, nephews, etc.). Thus, tasks that have no children, and whose children have all completed, are independent of one another and can proceed concurrently.

There are two complicating factors in designing a task-parallel algorithm. First, although two tasks may be independent in the sense described above, they may be interdependent in that they require data from the same processor. Second, in a task-parallel implementation much of the time is actually consumed waiting for dependent tasks to finish. For instance, a parent task may launch some child tasks and must then wait for its children to complete.

Both of these factors can be dealt with by exiting the current task at some point and starting (or restarting) a different task. Details of the algorithm used at each task, and the task-switching mechanism, are discussed in the next section.

The task routine used in the task-parallel implementation is shown in Algorithm 4.1. Functions attributable to a task use the internal data and are written using the task's dot (.) operator borrowed from C++ syntax. For example, *task*.Box() returns the box of the task, and *task*.Overlap() returns the amount of overlap between the candidate box and local grids on the tagged level.

Algorithm 4.1 largely follows the steps in the original single-task algorithm 2.1 to evaluate a candidate box. The processors compute local signatures and sum the signatures on the root processor. The root processor decides whether to accept or split the candidate box and broadcasts the decision to the participating group of processors (along with candidate boxes for child tasks, if any). If the box is does not meeth the efficiency criteria, line (i), the task routine creates child tasks and computes the associated group of processors holding data for the task. The child tasks are then placed in the task manager (details of which are given in the next section).

Once a task has put its children in the task manager it waits for them to finish. The function WaitForChildren called at line (ii) was a wait similar but not identical to those caused by communications. The operational difference is that the task is not immediately put in the task manager; it is put there only after its last child finishes, at line (iii). Differentiating "communication waits" from "child waits" prevents a parent task from being unnecessarily checked until all its children are completed. When the parent continues, the CheckCombinedEfficiency method performs a combined efficiency check (like what is done in the single-task Algorithm 2.1) to determine if the efficiency of the children warrants keeping them over their parent (i.e. if the combination of the children boxes are within a certain tolerance of the parent box).

Algorithm 4.1: TASKROUTINE(*task*)

```
if task.OVERLAP() > 0
  then {
    task.COMPUTELOCALSIGNATURES()
    task.SUMREDUCESIGNATURES()
    if local process rank = 0
      then {
        task.BOX() ← bounding box of tags
        task.ACCEPTORSPLIT()
      }
    task.BROADCASTACCEPTABILITY()
    if task.ACCEPTABILITY() = false                      (i)
      then {
        task.CREATECHILDREN()
        task.FORMCHILDGROUPS()
        PUTINTASKMANAGER(task.LEFTCHILD())
        PUTINTASKMANAGER(task.RIGHTCHILD())
        task.WAITFORCHILDREN()                           (ii)
        task.CHECKCOMBINEDEFFICIENCY()
      }
  }
comment: After last child completes, parent may continue.
if task.SIBLING() is completed
  then PUTINTASKMANAGER(task.PARENT())                   (iii)
```

To exit a task before it completes, we implemented a *self-suspending node routine* by building in logic for suspending the routine and returning to where it left off. The node routine suspends itself by storing its state in a data structure and exiting. When it is restarted, it jumps to the point where it was suspended, using the stored information. The node routine suspends itself at points where the processor has to wait and does no useful work. The node routine may be viewed as a sequence of alternating local computation and wait phases, much like other SPMD applications. A node might wait for its communications or for its children to complete. (Communications must be initiated by non-blocking calls so that the task is not forced to wait for the communication to finish.) Algorithm 4.2 illustrates the general logic for implementing the self-suspending node routine.

Algorithm 4.2: SelfSuspendingNodeRoutine(*node*)

```
comment: node is a data structure containing data associated
         with a particular dendogram node. This data includes
         the state of the node routine when it was suspended.
         The "switch" statement structure is borrowed from
         the C language.

switch ( state of task )                              (i)
case starting routine:                                (ii)
compute local signatures for node
initiate sum-reduce operation for signatures
case reducing signature:                              (iii)
if sum-reduce is incomplete
  then { save state of task                           (iv)
         PutInTaskManager(node)
         return
...
case ...:
...
endswitch
```

The "switch" statement (i) directs the routine to the appropriate "case statements" (ii, iii, etc.). At statement (iv), to where the node routine would return before it completes, it marks its state and places itself back in the task manager for restarting later.

4.2 Task Manager Algorithm

We investigated two approaches for a task manager algorithm. The first is a simple queue-based method wherein tasks are pulled from the queue and started (or restarted). The task either completes, or the self-suspending routine re-enters the task at the tail of the queue. In either case the manager continues to the next task at the head of the queue. Described in Algorithm 4.3, this simple queue task manager was found to be inefficient because it looped through an enormous number of cycles before finding a task with completed communications, consuming CPU cycles without making progress toward completion.

```
Algorithm 4.3: QueueTaskManager($Q$)

 comment: $Q$ is a queue (list) of tasks.

 while $length(Q) > 0$
     do {
         $task \leftarrow$ Dequeue($Q$)
         SelfSuspendingTaskRoutine($task$)
         comment: If $task$ did not complete, it is put back in the
                  queue by the self-suspending task routine of
                  Algorithm 4.2.
     }
```

To get around this problem we needed a mechanism to more quickly find those tasks that can make immediate progress. This led to our second approach, which was ultimately adopted into our code since it yielded better performance. This approach utilizes a two-stage method where one stage treats tasks that can make immediate progress and another stage treats tasks that are waiting on communications. In the two-stage approach, the queue mechanism only accepts tasks that can make immediate progress, which include new tasks and tasks whose children have just completed. Tasks that were waiting for communication are processed by a second stage that is entered when the queue is empty. This algorithm, which involves two stages, is referred to as the two-stage task manager.

```
Algorithm 4.4: TwoStageTaskManager()

 while {
     There are tasks in the queue  or
     there are incomplete communications
 }
     do {
         Restart all tasks in the queue until the queue is empty
         Wait for some communication calls to finish
         Restart tasks corresponding to completed communications
     }
```

In the two-stage task manager, tasks waiting for communication are referenced through their `MPI_Request` objects through a mapping of the request back to the task. (`MPI_Request` objects are handles, returned by MPI, that refer to specific outstanding communication requests.) Communication waits are handled by MPI_WAITSOME and invoked only when there are no more tasks that can make immediate progress, which in practice happens when the queue is empty. Parent tasks waiting for their children to complete are not entered into the two-stage task manager; children tasks put their parent back into the manager once they have completed (see Algorithm 4.1). The outermost loop in the two-stage algorithm is required because more tasks might be added to the queue as it executes, and more communication operations might be launched.

Implementation of the two-stage task manager required storing a queue of tasks, an array of `MPI_Request` objects and a mapping from the request objects to the tasks waiting for them. The array of request objects was input to MPI_WAITSOME to get the next set of MPI communications that have completed.

The task manager acts as a user-space thread controller analogous to the multi-thread approach. We chose not to use multi-thread libraries for this in order to have more flexibility to experiment with the implementation. For example, the task manager used information not available to the threading library to choose which task to start.

4.3 Multiple Owners

The task-parallel approach uses the more effecient group-sum approach of algorithm GROUPSUMCLUSTER, which relies on the root processor to receive the signature, evaluate the candidate box and broadcast the results. However, with many tasks occuring simultaneously, the possibility of the root processor becoming overloaded arises. We alleviate this problem by selecting an "owner" for each participating group from among the group members. Each group uses its ownwer as the "root" for group communication rather than all using the same root processor. We give the name TASKPARALLELCLUSTER to the simpler task-parallel algorithm that uses a single root processor for all tasks. The multi-owner task-parallel algorithm is named MULTIOWNERTASKPARALLEL.

For the multi-owner algorithm, the owner processor was selected as the one having the greatest overlap with the candidate box. We experimented with other selection criteria, such as choosing the processor owning or participating in the fewest number of tasks, but the differences to clustering performance were minimal.

4.4 Task Parallel Performance

We evaluated the performance of our two task-parallel algorithms, TaskParallelCluster and MultiOwnerTaskParallel using the moving-grid configuration with which we tested the single-task algorithms with earlier (see Section 3.1). Figure 7 shows timing results of the two algorithms; for comparison, timings of the single-task algorithms are also shown. Both task-parallel algorithms outperformed the single-task algorithms. The improvement of MultiOwnerTaskParallel over GroupSumCluster is almost as much as that of GroupSumCluster over GlobalSumCluster. For small and moderate numbers of processors, where both single-task algorithms were comparable, the MultiOwnerTaskParallel algorithm was several times faster. The task-parallel algorithms also exhibited improved scaling characteristics seen by the relatively mild rise in run times as the number of processors is increased.

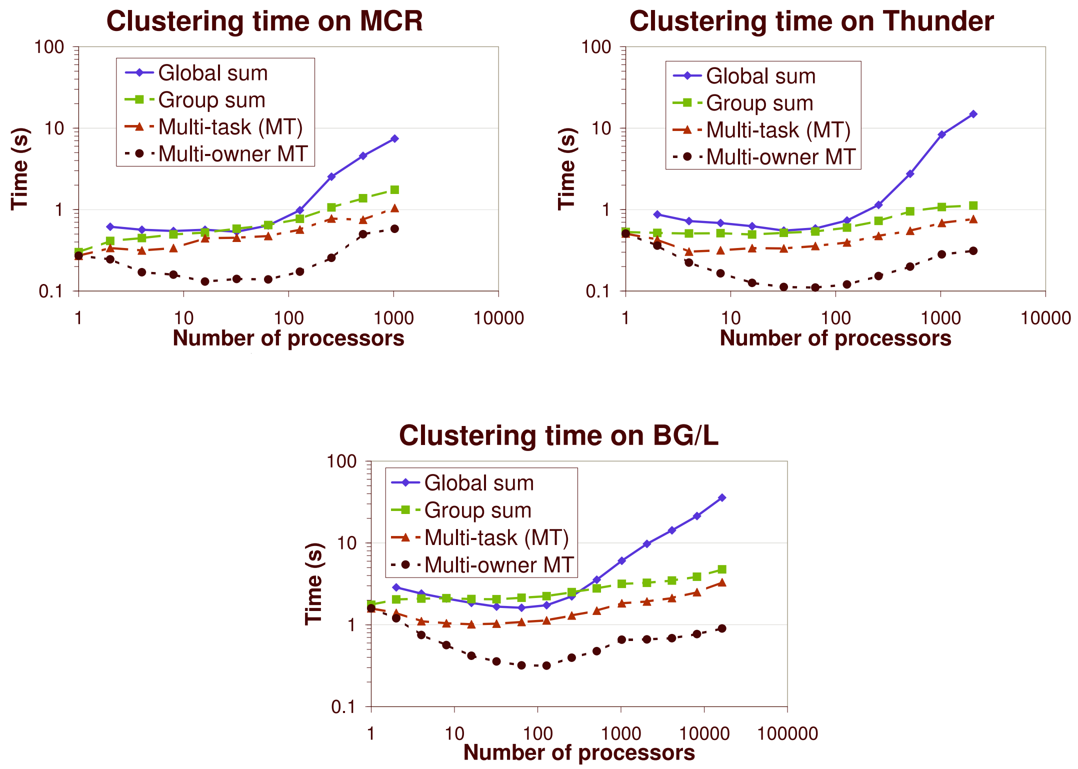

Fig. 7. Comparison of the clustering cost for an advecting sinusoidal front problem, using the task-parallel algorithms. For comparison, the results from the single-task algorithms are also shown.

Table 4 shows the speed-up factors of the different clustering algorithms on the largest processor partitions tested on each machine. Both task parallel implementations show significantly higher speed-ups compared to the best single-task algorithm. The MultiOwnerTaskParallel algorithm showed roughly a 2x better speed-up over the TaskParallelCluster algorithm on MCR and Thunder, and a 40x speed-up on BG/L. Better load balance in the multi-owner implementation accounted for the improved performance. In the best case, algorithm MultiOwnerTaskParallel achieved a 48-fold increase in speed over the GlobalSumCluster algorithm.

Table 4
Speed-up factors on different machines/numbers of processors for the single-task and task-parallel clustering algorithm on the largest number of processors of each platform.

Algorithm	Speed-up over GlobalSumCluster algorithm		
	1K MCR	2K Thunder	16K BG/L
GroupSumCluster	4.2	13	7.4
TaskParallelCluster	7.1	20	11
MultiOwnerTaskParallel	13	48	40

To investigate the effect of increasing the problem size, we also evaluated the performance of the task parallel algorithms on a larger problem, using the same experiment described earlier but with six levels of refinement rather than four. The resulting six-level configuration had two orders of magnitude more boxes than the four-level configuration. Figure 8 shows the timing results for this larger problem. Improvements similar to those in the four-level problem can be seen. The relative ranks of the algorithm speeds are largely the same. The biggest difference is an increase in the gap between the MultiOwnerTaskParallel algorithm and the rest. This was probably due to the algorithm's better load balancing, which became more important as the the problem size increased. The figure shows the MultiOwnerTaskParallel algorithm to be about 10x faster than the next fastest implementation on all three platforms. Interestingly, the TaskParallelCluster algorithm suddenly performs worse than the GroupSumCluster algorithm after eight processors on Thunder. This may be due to the numerous non-blocking communications managed by the root processor, which could cause sub-optimal performance of the underlying communication.

5 Summary

We discussed several methods for parallelizing the clustering algorithm of Berger and Rigoutsos (BR91). First, we focused on the performance using

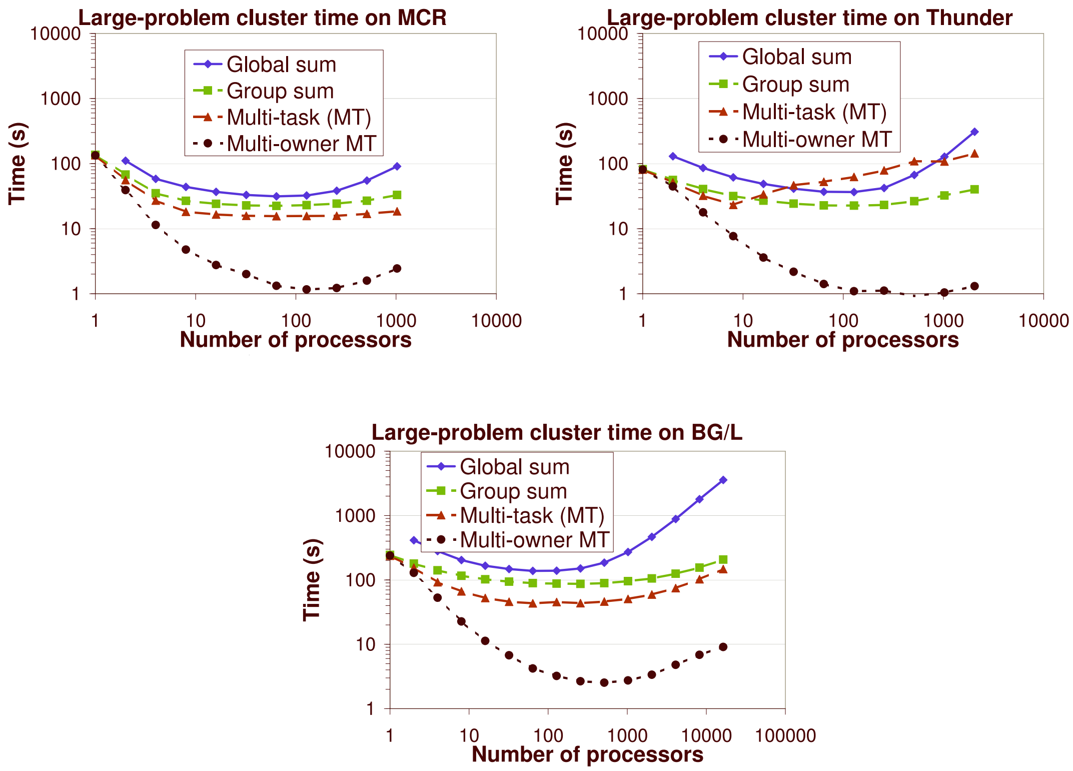

Fig. 8. Clustering cost for a larger problem.

two single-task parallel implementations of the original serial algorithm. The first was a simplistic extension of the original algorithm. The second eliminated unneeded communications by limiting collective communications to smaller groups of processors. We then introduced a task-parallel approach in which we divided the work into independent tasks that ran concurrently, to better utilize the time spent waiting on communications. We assessed the task-parallel algorithm with two variations, one that uses a single root processor for all communicaitons, and another which selects the root processor for communications from among the group members. We compared performance of the different algorithms for a model structured adaptive mesh refinement (SAMR) problem on several current parallel platforms, including up to 16K processors of the BlueGene/Light system.

Our experiments showed that the best single-task implementation was one that limited collective communications to the groups of processors that held relevant data (rather than using global collective communications). It is between 4x and 13x faster than the simplistically parallelized algorithm, depending on the computing platform and number of processors.

Adding task-parallelism to the new communication approach further sped up the clustering process, as did using multiple root processors in the collective communications. The fastest algorithm utilized all three concepts: limitting communication groups, task-parallelism and multiple root processors in the collective communications. It was up to 5.4x faster than the best single-task algorithm and up to 40x faster than the algorithm using global communications. On a larger test problem, this algorithm showed even greater gains over the other algorithms.

The new algorithms showed much better scaling qualities that is very advantageous for large processor systems like BG/L. Their performance remained largely consistent across the three platform used in our experiments.

6 Acknowledgements

The authors gratefully acknowledge many useful discussions with the SAMRAI development team.

This work was performed under the auspices of the US Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Report number UCRL-JRNL-216580.

References

[And73] Michael R. Anderberg. *Cluster Analysis for Applications.* Academic Press, New York, 1973.

[BB87] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, 1987.

[BC89] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:65–84, 1989.

[BO84] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[BR91] M. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man and Cybernetics*, 21(5):1278–1286, September/October 1991.

[CGL$^+$03] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini, and B. Van Straalen. *Chombo Software Package for AMR Applications Design Document.* Applied Numerical Algorithms Group, NERSC Division, September 2003. http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf.

[DH73] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, 1973.

[Har73] John A. Hartigan. *Clustering algorithms.* Wiley, New York, 1973.

[HK02] Richard D. Hornung and Scott R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002.

[KB96] Scott R. Kohn and Scott B. Baden. Irregular coarse-grain data parallelism under LPARX. *Scientific Programming*, 5(3):185–201, 1996.

[MH80] D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London*, 207, 1980.

[Ran03] Jarmo Rantakokko. An integrated decomposition and partitioning approach for irregular block-structured applications. In J. Rolim et. al., editor, *Parallel and Distributed Processing, 15 IPDPS 2000 Workshops, Cancun, Mexico, May 1-5, 2000, Proceedings*, pages 336–347, Berlin, June 2003. Springer-Verlag.

[RBL$^+$00] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield, and John B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3(3):147–157, 2000.

[SAM04] SAMRAI web site, 2004. http://www.llnl.gov/CASC/SAMRAI/.

[WHH03] Andrew M. Wissink, David Hysom, and Richard D. Hornung. Enhancing scalability of parallel structured AMR calculations. In *Proceedings of the 17th ACM International Conference on Supercomputing (ICS03)*, pages 336–347, San Francisco, June 2003.