CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2011; **23**:1522–1537
Published online 25 March 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1719

Scalable parallel regridding algorithms for block-structured adaptive mesh refinement

J. Luitjens*, † and M. Berzins

School of Computing, 50 S Central Campus Dr. Rm. 3190, Salt Lake City, UT 84112, U.S.A.

SUMMARY

Block-structured adaptive mesh refinement (BSAMR) is widely used within simulation software because it improves the utilization of computing resources by refining the mesh only where necessary. For BSAMR to scale onto existing petascale and eventually exascale computers all portions of the simulation need to weak scale ideally. Any portions of the simulation that do not will become a bottleneck at larger numbers of cores. The challenge is to design algorithms that will make it possible to avoid these bottlenecks on exascale computers. One step of existing BSAMR algorithms involves determining where to create new patches of refinement. The Berger–Rigoutsos algorithm is commonly used to perform this task. This paper provides a detailed analysis of the performance of two existing parallel implementations of the Berger–Rigoutsos algorithm and develops a new parallel implementation of the Berger–Rigoutsos algorithm and a tiled algorithm that exhibits ideal scalability. The analysis and computational results up to 98 304 cores are used to design performance models which are then used to predict how these algorithms will perform on 100 M cores. Copyright © 2011 John Wiley & Sons, Ltd.

Received 21 June 2010; Revised 18 November 2010; Accepted 29 January 2011

KEY WORDS: adaptive mesh refinement; regridding; remeshing; scalability

1. INTRODUCTION

Large-scale adaptive mesh refined (AMR) simulations are increasingly being used throughout the scientific community. AMR simulations provide an advantage over fixed mesh simulations by refining the computational mesh in portions of the domain that have significant error and thereby reducing the error in those regions. This allows simulations to produce results that are as accurate as fine-scale fixed mesh computations but at a greatly reduced computational cost. The meshes used within AMR algorithms can either be structured (SAMR) or unstructured (UAMR). Both methods have their advantages and disadvantages as discussed in [1].

A method for block-structured AMR (BSAMR) was originally presented by Berger and Oliger in [2] and then by Berger and Colella in [3]. This algorithm used various approaches to flag portions of the domain that require more refinement. The algorithm then adds patches of refinement on top of the refinement flags through a process commonly referred to as remeshing or regridding. The process is repeated until a desired resolution is reached producing a dynamic multi-level grid where each finer level is nested within the coarser level as shown in Figure 1.

The next advance was a commonly used regridding method which is known as the Berger–Rigoutsos algorithm [4]. This algorithm uses edge detection algorithms from image processing to generate a tight-fitting axis-aligned patch-set with relatively few patches. This algorithm was later

*Correspondence to: J. Luitjens, School of Computing, 50 S Central Campus Dr. Rm. 3190, Salt Lake City, UT 84112, U.S.A.
†E-mail: luitjens@cs.utah.edu

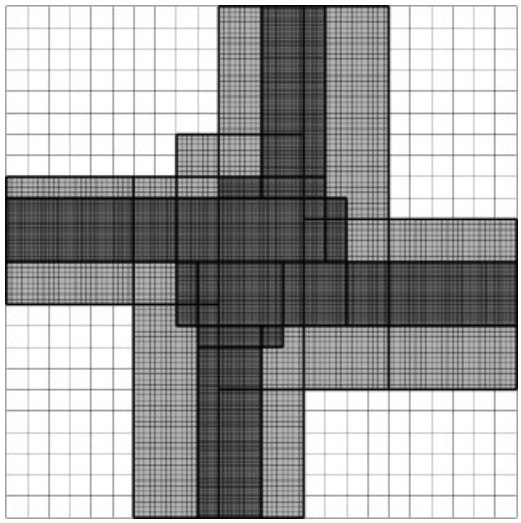

Figure 1. A multi-level grid of nested levels used within BSAMR.

parallelized in [5, 6] but both relied substantially on global communication which limited scalability on large numbers of cores [7]. Improvements to this algorithm which focused on reducing the amount of global communication were later presented in [7] and showed scalability to 16 K cores.

To change the grid in response to a solution evolving in time, a number of steps must occur that would not occur in a fixed mesh calculation. These steps generally include the creation of the new grid (regridding), the assignment of the new grid to cores (load balancing), and updating the computation and communication algorithms (scheduling). In many adaptive simulations these changes can occur as often as every few timesteps. This necessitates that these algorithms run efficiently in parallel. Poor performance within any of these algorithms can lead to performance problems at large scales [6, 8, 9].

Octree-structured AMR (OSAMR) is an alterative to BSAMR that builds the mesh using an octree. This method combines elements of UAMR with SAMR and has shown promising results [10–12], by scaling to over 200 K cores. This method offers a fundamentally different approach to that considered here. The challenge considered here is to analyze and extend the BSAMR method to similar scalability and beyond.

Achieving large-scale parallelism, especially for AMR applications, is a challenging task. This has led to the development BSAMR frameworks like Uintah [9, 13], SAMRAI [8], Chombo [14], and AMROC [15] along with OSAMR frameworks such as Paramesh [16], Flash [17], and ALPS [18].

These frameworks simplify the development of large-scale adaptive simulations by compartmentalizing the AMR algorithms allowing them to be developed and optimized separately [13, 19] and also allow for code reuse. Scalability onto today's largest machines has been shown for some of these frameworks [10–12, 14, 20].

The first exascale machine is expected to arrive within the next decade and it is estimated that it may contain on the order of 100M cores. Achieving scalability onto machines of this size will be a challenging task. As we move toward exascale computers, the scalability of these frameworks and the algorithms they use will need to be improved. This paper analyzes the performance of two existing parallelizations of the Berger–Rigoutsos algorithm, develops a new parallelization of the Berger–Rigoutsos algorithm, and develops an alternative regridding algorithm that exhibits ideal scalability. The scalability characteristics of this algorithm make it a candidate for exascale computers at some point in the future.

The situation is summarized by Solomonik and Kale [21] in the context of quite different algorithms but similar architectures: 'However newer and much larger architectures have changed the problem statement further. Therefore traditional approaches…require re-evaluation and

DOI: 10.1002/cpe

improvement' in undertaking an investigation of this type, but for AMR, the regridding algorithms are described in Section 2, the characteristics of patch-sets generated is analyzed in Section 3, Section 4 uses the analysis along with experimental results using up to 98 304 cores to derive and validate performance models, and finally Section 5 predicts the performance of these regridding algorithms on 100 M cores. Throughout this paper we will use the term core to refer to a single processing unit on which distributed memory processes are executed.

2. REGRIDDING ALGORITHMS

In this section, existing BSAMR algorithms are evaluated with regard to their parallel performance. The Berger–Rigoutsos regridding algorithm was originally designed in serial and focused on generating a patch-set that minimized the total amount of refinement while also minimizing the total number of patches [4].

This algorithm has worked well leading to its widespread use throughout BSAMR community. Parallel versions of this algorithm were presented in [5, 6] and improved in [7]. These parallelizations computed portions of the algorithm in parallel and combined the results across cores using all-reduces. An all-reduce is a process whereby data on each core is combined with data from each other core. For example, an all-reduction may compute the maximum value across cores or the sum across cores. This algorithm has been implemented within SAMRAI and Uintah and within these frameworks it has been observed that the reduction operations limited the parallel performance of the algorithm at large scales [7, 22]. These algorithms will be referred to as global Berger–Rigoutsos (GBRv1 and GBRv2, respectively). An alternative approach to GBR is to run Berger–Rigoutsos locally on a subset of the domain. This approach is similar to the approach taken in AMROC [23]. This algorithm will be referred to as LBR. Finally, within Uintah, a tiled algorithm has been implemented which initially has shown good performance and will be examined throughout this paper.

The remainder of this section will describe each of these algorithms in more detail and provide complexity analysis of their performance. This analysis will assume that the work per core is load balanced well, making the time for load imbalance insignificant and that all logarithms are base two.

Within this section the following symbols will be used:

$$C = \text{Number of mesh cells in the domain,}$$

$$F = \text{Number of refinement flags in the domain,}$$

$$B = \text{Number of patches (blocks) in the domain,}$$

$$P = \text{Number of processing cores.}$$

The Berger–Rigoutsos algorithm operates on a list of refinement flags which is a sparse data structure. However, in many frameworks, the refinement flags are stored per-cell which is a dense data structure. These frameworks often require an extra step that computes the list of refinement flags. This step can be done by iterating over the dense data structure and adding all refinement flags to a list, which would have a parallel complexity of $O(C/P)$.

It is worth noting that GBRv2, LBR, and the tiled algorithms generate a local patch-set, meaning that each core only has knowledge of a subset of the patches. Generally today's frameworks require each core to be aware of the global patch set. These frameworks would require an all-gather on the local patch-sets to generate the global patch-set. An all-gather is a process that combines local arrays on each core into a global array on each core and it is commonly done through the MPI_Allgather function. The analysis below does not include the time for creating the sparse flag set or for performing the all-gather which may be significant at large numbers of cores. The time for these operations will be revisited in Section 4.

DOI: 10.1002/cpe

2.1. Serial Berger–Rigoutsos

The following presents a high-level description of the serial Berger–Rigoutsos algorithm (SBR). For brevity, a complete description of this algorithm has been omitted but can be found in [4]. The Berger–Rigoutsos algorithm can be defined recursively as shown in Algorithm 1 below.

The first step of the algorithm is to compute the bounding box of the refinement flags list. This can be done in $O(F)$ time. The second step is to check the termination condition by comparing the number of cells in the bounding box to the number of flags. The time for this is insignificant. The third step is to compute a histogram of the spatial location for each refinement flag. A histogram is created for each dimension yielding X, Y, and Z histograms. This process touches each flag after making the complexity for this operation $O(F)$. Next each

Algorithm 1 The serial Berger–Rigoutsos algorithm.

```
BRSplit(INPUT: flags, OUTPUT: patches)
   //compute the bounding box
   BB=computeBoundingBox(flags)

   //check the termination condition
   IF flags.size/BB.size > tolerance
      patches.add(BB)
      RETURN
   END IF

   //compute the histogram in each dimension
   histogram=computeHistogram(flags)

   //split the domain into left and right halves
   [left,right]=split(flags,histogram)

   //recursively repeat this process
   BRSplit(left,patches)
   BRSplit(right,patches)
```

histogram is used to determine the best location to split the domain and the flags are divided into left and right halves. This step has a complexity of $O(F)$. Finally, the algorithm recursively repeats on each half. The recursion implicitly defines a recursive tree in which parent nodes are split into two children nodes. In the best case, the recursive tree is a complete tree in which every parent has two children except perhaps the lowest level of the tree. In this case, the recursion would repeat $O(\log B)$ times. If the tree were largely unbalanced, then in the worst case the algorithm could repeat up to $O(B)$ times. In practice, the algorithm generally behaves similarly to the best case which is

$$O(\text{SBR}) = F \log B.$$

The left image of Figure 2 shows an example of a patch-set generated by the SBR algorithm. The patches fit the refinement flags tightly and are irregular in size.

2.2. Parallel global Berger–Rigoutsos (GBR)

The GBR algorithm is implemented in parallel by performing the bounding of the refinement flags and the generation of the histograms in parallel [6, 7]. To do this each core computes the bounds and the histogram on a subset of the refinement flags and then combines the results with every other process through an all-reduce operation. In this analysis, the time to perform the all-reduce will be referred to as the communication time and the time for the rest of the algorithm will be referred to as the computation time.

DOI: 10.1002/cpe

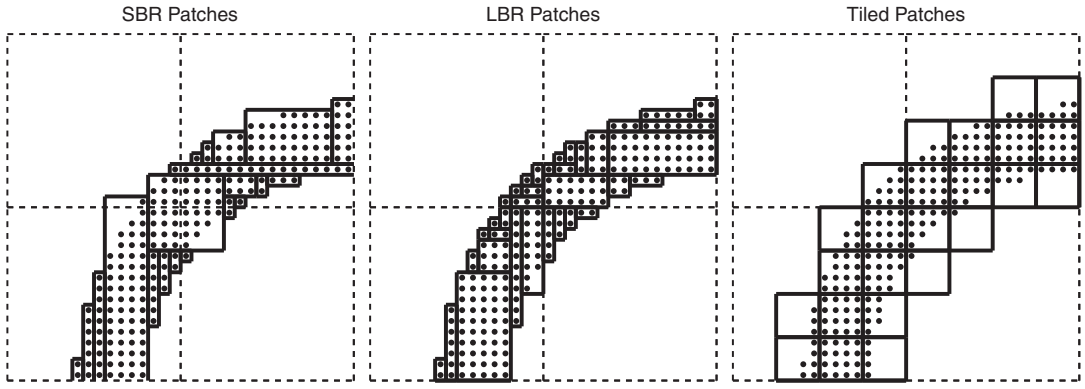

Figure 2. Patch-sets generated by the SBR (left), LBR (middle), and tiled (right) algorithms. The coarse-level patches are drawn using dashed lines and the fine-level patches are drawn using solid lines. The SBR and LBR algorithms use less patches but generate irregular patch-sets. In the LBR algorithm boundaries of the coarse level also exist on the fine level. The tiled algorithm generates regular patches.

The time for the computation of GBR is equivalent to the serial algorithm performed on a subset of the flags. Consequently, the complexity for the computational portion of the GBR algorithm $O((F/P)\log B)$.

The communication step in GBRv1 involves performing two all-reduces at each node and leaf of the recursive tree. The parallel complexity for an all-reduce operation is not straightforward to define as it varies depending on the network topology and the MPI library implementation. Within some MPI libraries the algorithm used may vary according to the size of the reduction and the number of cores. In most cases the time for an all-reduce is dominated by message latency and thus the time for an all-reduce is proportional to the number of pairwise communications required to reduce the data, which requires a minimum of $\log P$ pairwise messages.

In the best case, the recursive tree of the Berger–Rigoutsos algorithm is completely balanced and the number of nodes is equal to $\sum_{k=0}^{\log B} 2^k$ which is the sum of a geometric series and is equal to $2B-1$ making the number of messages

$$M(\text{GBRv1}) = (2B-1)\log P.$$

Thus, the complexity for GBRv1 is

$$O(\text{GBRv1}) = \frac{F}{P}\log B + B\log P.$$

The improvements to this algorithm presented in [7] involve reducing the number of cores that contribute to reductions at each level of the recursion. At the first level, every core contributes and at each successive level the number of contributing cores decreases until the algorithm terminates or only a single core is contributing to each reduction, at which point the algorithm completes in serial. In the best case, the number of contributing cores at each node would be half the number of contributing cores of the parent node. This leads to a total number of messages of

$$\sum_{k=0}^{\min(\log B, \log P-1)} \left(2^k \log \frac{P}{2^k}\right).$$

Refactoring the equation above yields

$$\sum_{k=0}^{\min(\log B, \log P-1)} 2^k \log P - \sum_{k=0}^{\min(\log B, \log P-1)} k2^k.$$

DOI: 10.1002/cpe

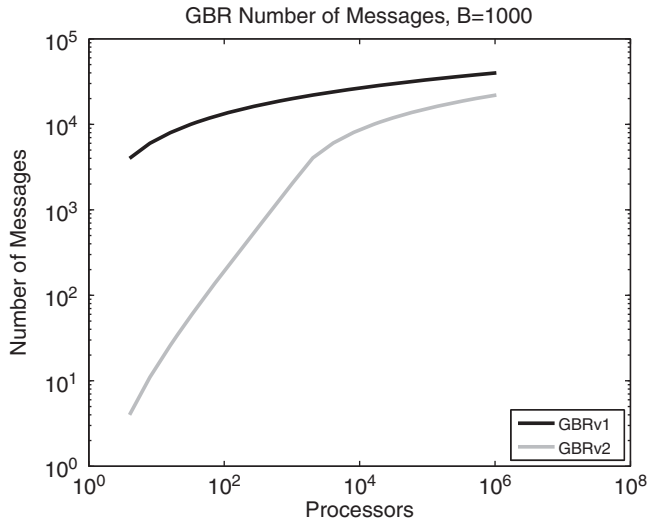

Figure 3. The number of messages required to execute GBR.

By substituting these summations with their known solutions found in [24] and performing algebraic simplification we get the number of messages is equal to

$$M(\text{GBRv2}) = \begin{cases} 2P - \log P - 2 & : P < B, \\ (2B-1)\log P - 2B\log B + 2B - 2 & : P \geq B. \end{cases}$$

Figure 3 shows the number of messages generated for both GBR algorithms when generating 1000 patches. This figure shows that GBRv2 has a significant improvement over GBRv1 when the number of patches is less than the number of processors. However, when this is not the case the number of messages generated by the two algorithms are similar.

Thus GBRv2 has an overall complexity of

$$O(\text{GBRv2}) = \begin{cases} \dfrac{F}{P}\log B + P & : P < B, \\ \dfrac{F}{P}\log B + B\log P & : P \geq B. \end{cases}$$

From this we can see that when $P < B$ the communication complexity is $O(P)$, however, when $P \geq B$ the communication complexity is $O(B \log P)$ which is the same as GBRv1. The GBRv2 algorithm replaces the all-reductions in the GBRv1 algorithm with pairwise asynchronous communication performed along the edges of a hypercube. In addition, the algorithm is decomposed into tasks, where each task represents a node of the recursive tree. The tasks are designed such that they are restartable. When a task blocks for communication it cedes control of the core so that the next scheduled tasks can execute. Once communication is completed that task is once again scheduled for execution and will restart where it left off. When there are enough tasks the communication can be overlapped with computation leading to potentially further increases in performance. These optimizations within the GBRv2 algorithm are complex and can be difficult to implement correctly.

The analysis above did not consider the case where communication is overlapped with computation and thus should be considered an upper bound. Details of this algorithm can be found in [7].

2.3. *Local Berger–Rigoutsos* (*LBR*)

The LBR algorithm performs SBR locally on each coarse patch to generate finer patches that are nested within the coarse patch. This provides an advantage over GBR in that no communication is required to bound the refinement flags or generate the histogram. However, since this algorithm

DOI: 10.1002/cpe

is running locally on each patch, the patches on the finer level cannot span across a coarse patch. This causes the boundary of the coarse patches to project onto finer patches thereby increasing the total number of patches. This is similar to an algorithm implemented within AMROC's [23], however, that algorithm runs SBR locally on the entire local patch-set and then intersects the resulting patch-set with the original patches. This generates more patches than the SBR algorithm but less patches than the LBR algorithm. In addition, this results in patch-sets that change as the number of cores changes, which may be an undesirable property.

The complexity for LBR (assuming that patches are fairly evenly sized) is defined as follows. Let B_c be the number of coarse patches. Then the flags per coarse patch is F/B_c, the number of coarse patches per core is B_c/P, and the number of fine patches per coarse patch is B/B_c. Thus the complexity for this operation is

$$O(\text{LBR}) = \frac{B_c}{P}\frac{F}{B_c}\log\frac{B}{B_c} = O\left(\frac{F}{P}\log\frac{B}{B_c}\right).$$

AMROC's implementation of this algorithm has a similar complexity. The middle image in Figure 2 shows the patch-set generated using the LBR algorithm. This figure shows the patch-set is similar to SBR and that the boundaries of the coarse patches exist on the finer level.

2.4. Tiled

The tiled algorithm was developed within Uintah to eliminate the performance problems associated with the GBR algorithms. A similar grid algorithm was utilized in [25] for non-hierarchical grids. This algorithm is defined in Algorithm 2.

Algorithm 2 The tiled regridding algorithm.

```
FOR each local tile
    FOR each cell in tile
        IF cell has refinement flag
          patches.add(tile)
          BREAK
        END IF
    END FOR
END FOR
```

This algorithm defines a set of tiles using a lattice where each lattice cell corresponds to a possible patch. To compute the patch-set each core searches a subset of the tiles for refinement flags, when a refinement flag is found the tile is added to the patch-set and the next tile is searched for refinement flags. Like the LBR algorithm this algorithm requires no communication. This algorithm is an embarrassingly parallel algorithm and has a complexity of

$$O(\text{Tiled}) = \frac{C}{P}.$$

The right image in Figure 2 shows an example of a patch-set generated by the tiled regridder. In this case, the algorithm uses more patches than the Berger–Rigoutsos algorithms, however, the patches generated are regular. If the algorithm is generating too many patches, which are causing inefficiencies elsewhere in the code, then the tiled size could be increased, which would thereby decrease the number of patches. The grids generated by the tiled algorithm look similar to the grids produced using OSAMR, however, it is not the case that they are the same. For example, OSAMR patches are constrained to a power of two in size whereas patches produced by the tiled algorithm are not.

DOI: 10.1002/cpe

3. PATCH-SET CHARACTERISTICS

The characteristics of the patch-sets generated by these algorithms is important. The following will describe and analyze the differences between the patch-sets generated by the SBR, LBR, and tiled algorithms.

The original Berger–Rigoutsos algorithm generates patch-sets of varying sizes. Using this method it is not uncommon to have patches that span large portions of the domain next to patches that only span a few cells which can complicate the load balancing operation. The irregularity of the patches necessitates a general framework. Some frameworks restrict the types of patch-sets to simplify other portions of the framework [22]. These restrictions require a cleanup phase after regridding to ensure that the grid constraints are met [22].

The GBRv1 and GBRv2 algorithms generate patch-sets that are equivalent to the SBR algorithm. Like the SBR algorithm the LBR algorithm generates irregular patches; however, it will generate more patches than the SBR algorithm. Finally, the tiled algorithm may generate more patches but the patches are regular. This regularity can be exploited to provide further increases in performance. The following will discuss the advantages and disadvantages of the number of patches, tightness of fit, and regularity of the patch-sets.

Each of the regridders above were implemented as a stand-alone program as described in their original papers and the patch-sets generated by each regridder were tested with a benchmarking problem. This problem defined a 3D domain in the range of [0,1]. Within this domain refinement flags were added in the region between two concentric spheres centered at [0.5,0.5] with radii 0.3 and 0.4. The resolution of the grid was increased and the patch-sets generated were recorded. The coarse level was split into square patches of 16^3 cells and the tile size was set to 16^3 for the tiled regridder. The Berger–Rigoutsos based algorithms used a tolerance parameter of 0.85, meaning that the recursion terminated when 85% of the patch contained refinement flags. This problem was chosen as it closely resembles an expanding blast-wave that is commonly simulated within Uintah [9, 22]. Figure 2 shows the corner of a 2D version of this test problem.

3.1. Number of patches

Increasing the number of patches also decreases the average size of a patch for a fixed size mesh. Depending on the underlying framework there may be a significant overhead per patch. If the overhead is significant then increasing the number of patches may hinder performance as shown in [26]. In addition, some data structures may depend on the total number of patches. For example, Uintah and SAMRAI utilize tree-based data structures for neighbor finding [6]. As the number of patches increases the size of these data structures and the time to query them will also increase.

While increasing the number of patches may decrease performance, it can also potentially increase parallel performance. Decreasing the average size of a patch often decreases the load imbalance which leads to an increase in performance. In addition, having more patches on each processor may allow the simulation to proceed in a more asynchronous manner reducing the synchronization and thereby overall simulation time [27]. Finally, decreasing the patch size may also improve cache performance. Whether or not increasing the number of patches will increase or decrease performance depends on the framework. Within Uintah, we have found that increasing the number of patches often increases the overall performance. However, the performance begins to decrease once patches become too small [27].

Figure 4 shows the number of patches generated by each regridder for the benchmark problem. In this graph the points are the recorded data and the lines are a linear least squares fits to the data for the curve C^m. The slope of these lines correspond to the rate at which the number of patches grows relative to the number of cells. This graph shows that the number of patches using the tiled algorithm is directly proportional to the number of cells. The SBR algorithm grows at $\approx C^{0.5}$ and the LBR grows at $\approx C^{0.7}$. The tiled algorithm uses a factor of 10 more patches at the largest scales.

For scalability at large numbers of processors it is desirable that the number of patches scale proportionally to the number of cells. Since the SBR and LBR algorithms do not exhibit this

DOI: 10.1002/cpe

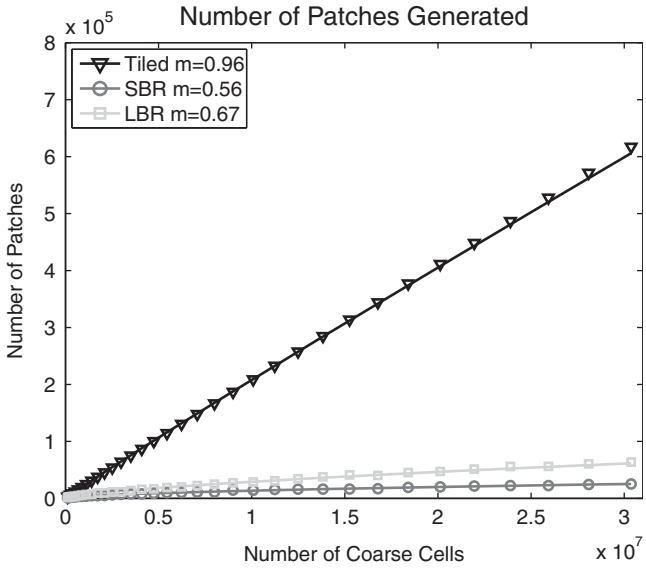

Figure 4. The number of patches generated by the regridding algorithms. The points are the recorded data and the lines are a linear least squares fit of that data within a loglog plot.

property they will require an extra step for large problems at large numbers of processors. This step will need to split the largest patches until every processor has at least one patch. One efficient way to do this would be to split every patch that was larger than some percent of the total volume of the new grid. This would take $O((B/P)+\log P)$ time to compute the total volume using an all-reduce and $O(B/P)$ time to split the patch-set leading to an overall complexity of

$$O(\text{Split}) = \frac{B}{P} + \log P. \tag{1}$$

The tiled algorithm would not require this step. The time for this step is investigated in Section 4.

3.2. Tightness of fit

Having a tight-fitting patch-set ensures a minimal amount of unnecessary refinement, thereby reducing the overall runtime. However, in applications that regrid often a tight patch-set may actually hinder performance. For these applications, the cost of regridding, load balancing, and scheduling can become significant. A loose patch-set can reduce the frequency of these operations. Recognizing this, Uintah provides an option for the user to increase the looseness of the patch-set, which has provided large increases in performance problems involving rapidly moving fronts [22].

Figure 5 shows the over-refinement as a percentage of the number of refinement flags for the regridding algorithms. This graph shows that for the SBR algorithm over-refinement increases with the problem size. This is due to large patches that cover a large number of refinement flags but also cover a large number of cells without refinement flags. This can be seen in the left image in Figure 2. This figure shows a large amount of over-refinement within the large vertical patch. The increase in over-refinement should level off at or below the tolerance parameter (15% in this case). The same increase was not seen with the LBR regridder. This is because the sizes of the fine patches are limited by the size of the coarse patches. The over-refinement of both the LBR and tiled algorithms decreases as the number of coarse cells increases. This reduction is due to a reduction in the size of a patch relative to the size of the features being tracked. As this size is decreased the patches more tightly cover the tracked feature thus reducing the over-refinement. The SBR algorithm generates the most over-refinement at larger problem sizes followed by the tiled algorithm and then the LBR algorithms.

DOI: 10.1002/cpe

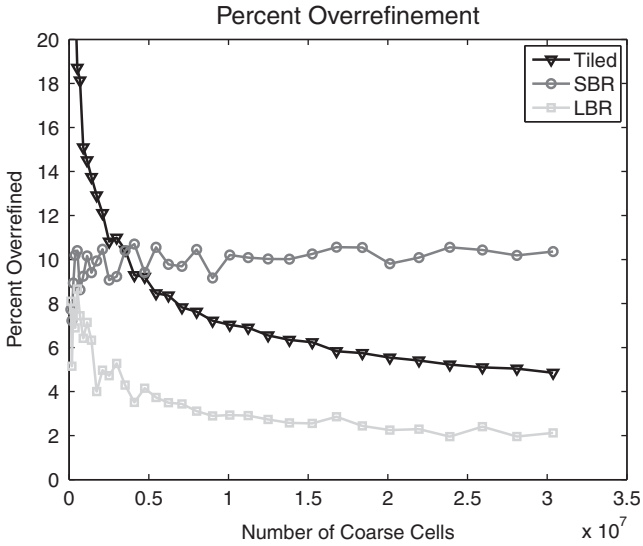

Figure 5. The over-refinement as a percentage of the number of refinement flags for the regridding algorithms.

Table I. Statistics on the number of cells per patch generated by the regridding algorithms.

	Mean	Min	Max	Stdev
SBR	1.1×10^4	64	3.1×10^6	9.0×10^4
LBR	4.2×10^3	64	3.3×10^4	1.0×10^4
Tiled	512	512	512	0

3.3. Regularity

Regularity of patch size within patch-sets is a desirable quality as irregular patches can cause load imbalance. In particular, having a large variance in the size of the patches is potentially problematic as load balancing regular sized patches is much more straightforward. In addition, having long slender patches next to many small patches produces load imbalance in the communication as the large patches must communicate to many neighbors but the small patches only have to communicate with a few neighbors.

In addition, if patches are completely regular the regularity can be exploited to simplify other portions of the algorithm. For example, neighbor finding on irregular patch-sets generally utilizes a tree algorithm, like a bounding volume hierarchy (BVH) [28], which has an $O(\log B)$ query time. When the patches are regular the time for this query can be reduced to $O(1)$ through the use of a hash table [24]. Table I shows the statistics on the number of cells within patches generated by the benchmark problem. This table shows that the SBR algorithm has the greatest variance followed closely by the LBR algorithm. The tiled algorithm has no variance. The SBR algorithm produces the largest patches followed closely by the LBR algorithm. The SBR and LBR algorithms also produce the smallest patches.

3.4. Summary of patch-set characteristics

Figures 4 and 5 along with Table I show that the SBR algorithm generates the least number of patches, this however comes at the cost of some over-refinement and irregular patch sizes. The LBR generates slightly more patches than the SBR but has significantly less over-refinement. The tiled regridder generates the most patches by far and has less over-refinement than the SBR algorithm. The SBR and LBR algorithms may not generate enough patches for large numbers of processors

DOI: 10.1002/cpe

and will thus require an extra step that splits large patches. The time for this operation may become significant at large numbers of processors. The tiled regridder will not have this problem since the number of patches scales linearly with the size of the coarse grid.

It is not clear if any type of patch-set is the best overall. Instead, it appears that the best type of patch-set depends on the framework and the application. If the framework has a low per-patch overhead and can exploit regularity it would appear that the patch-sets provided by the tiled regridder would be the best. However, if a large patch overhead exists and load imbalance is not a problem then the patch-sets generated by the LBR algorithm may be the best.

4. PARALLEL PERFORMANCE

This section evaluates the parallel performance of the regridding algorithms using performance models derived from the analysis above and experiments on Kraken[‡] up to 98 304 cores. The test problem was the same problem as described in Section 3. For these results each algorithm was tested 500 times and the average time was reported. The GBR algorithms were each executed only five times due to the amount of time required to run these algorithms and constraints on our machine usage. Finally, the GBR algorithms were not run at the largest numbers of cores due to this same constraint.

4.1. Performance models

Using the analysis in Section 2 the following performance models are defined:

$$T(\text{GBRv1}) = c_1 \frac{F}{P} \log(B) + c_2 M(\text{GBRv1}),$$

$$T(\text{GBRv2}) = c_1 \frac{F}{P} \log(B) + c_3 M(\text{GBRv2}),$$

$$T(\text{LBR}) = c_4 \frac{F}{P} \log\left(\frac{B}{B_c}\right),$$

$$T(\text{Tiled}) = c_5 \frac{C}{P},$$

$$T(\text{Split}) = c_6 \frac{B}{P} + c_7 \log(P),$$

where $c_1 = 10^{-6}$, $c_2 = 2 \times 10^{-4}$, $c_3 = 4 \times 10^{-5}$, $c_4 = 2.5 \times 10^{-8}$, $c_5 = 10^{-8}$, $c_6 = 4 \times 10^{-8}$, and $c_7 = 2 \times 10^{-5}$ were chosen to match experimental data.

In addition, the following models for F and B were used:

$$B = C^m,$$

$$F = 0.154\,C,$$

where m is the slope defined in Figure 4.

4.2. Parallel scalability

To run on large-scale machines applications must exhibit scalability on those machines. Scalability is a measure of runtime as the problem size and number of cores varies. Strong and weak scaling are two commonly used metrics for scalability. These metrics are defined below.

[‡]Kraken is a supercomputer located at the University of Tennessee with 99 072 cores. More information is available at http://www.nics.tennessee.edu/computing-resources/kraken.

DOI: 10.1002/cpe

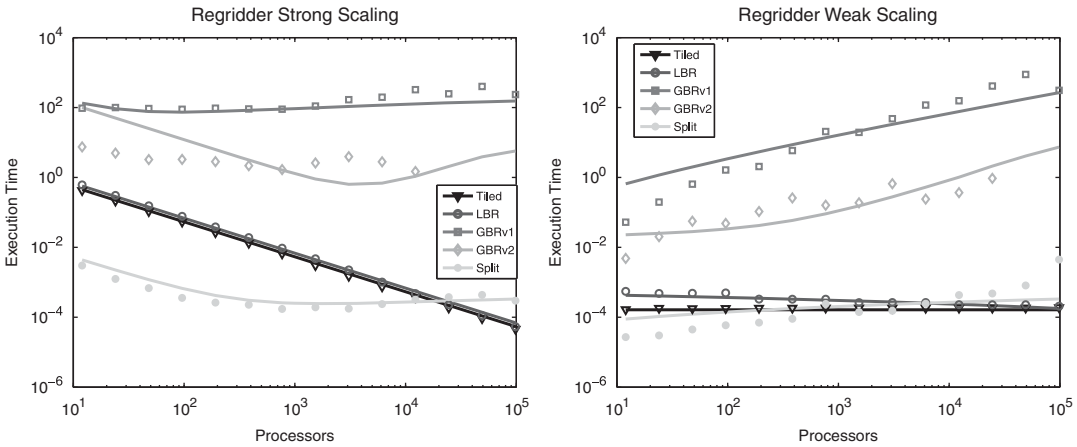

Figure 6. The strong and weak scaling for the regridders. The points represent the experimental data and the line represents the model. Ideal strong scaling is shown by a line with a slope of -1 and ideal weak scaling is shown by a horizontal line.

Let $T(N, P)$ be the runtime given the problem size (N) and the number of cores (P). Then *strong scaling* is defined as $T(N, P)$ for N fixed as P varies. Ideally the runtime should be inversely proportional to the number of cores. However, ideal strong scaling will eventually fail for all applications as more cores are added. This is because all applications, even embarrassingly parallel ones, contain some overhead that cannot be parallelized (for example the time to launch the application on each core). As the number of cores is increased the time for the parallel portion of the application is reduced while the serial time remains constant eventually leading to a failure in scalability [29].

Figure 6 (left) shows the theoretical and experimental strong scalability for the various regridding algorithms along with the time to split for the LBR algorithm. For this graph, $B_c = 131\,072$, $C = 4096 B_c$ and for the splitting algorithm $m = 0.7$. These parameters correspond to having around one 16^3 coarse patch per core at 100 000 cores. In this figure, the dashed line is the theoretical performance and the points are the experimental data. This graph shows that the models are highly accurate. The GBRv2 model is not as accurate as the others because the model did not take into account the possible overlap of communication and computation. Ideal strong scaling is shown as a diagonal line with a slope of -1 which is shown by both the LBR and tiled algorithms. The GBRv1 algorithm scales poorly. The GBRv2 algorithm strong scales well until the communication dominates the runtime, at which point the algorithm fails to scale. The splitting algorithm scales but eventually stops scaling due to the cost of the reduction. The time to split eventually exceeds the time to execute the LBR algorithm which will limit the total scalability of the LBR algorithm.

While exhibiting strong scaling is important, it does not measure the way in which large-scale machines are typically used. Weak scaling is a metric that more accurately measures scalability in terms of how the machines are utilized. *Weak scaling* is defined as $T(N, P)$ for N/P fixed. That is, the problem size per core is fixed and as more cores are added the problem size is proportionally increased. The weak scalability for the regridders is shown in Figure 6 (right).

For this graph $C = 16\,384 P$, $B_c = C/4096$ and for the splitting algorithm $m = 0.7$, which corresponds to each core having four 16^3 coarse patches. Ideal weak scaling is shown as a horizontal line. The models are highly accurate for all cases. This graph shows GBRv1 and GBRv2 again scale poorly. The tiled regridder shows ideal weak scaling and the LBR regridder shows better than ideal weak scaling. This is because the ratio B_c/B is decreasing as the problem size gets larger. The splitting algorithm weak scales according to $\log P$ and eventually takes longer than the LBR algorithm. The model for the splitting algorithm is not as accurate when weak scaling. The splitting algorithm is dominated by the time for the all-reduce, which is assumed to be $O(\log P)$. This assumption is not true for all problem sizes as the MPI library may choose a different reduction algorithm.

DOI: 10.1002/cpe

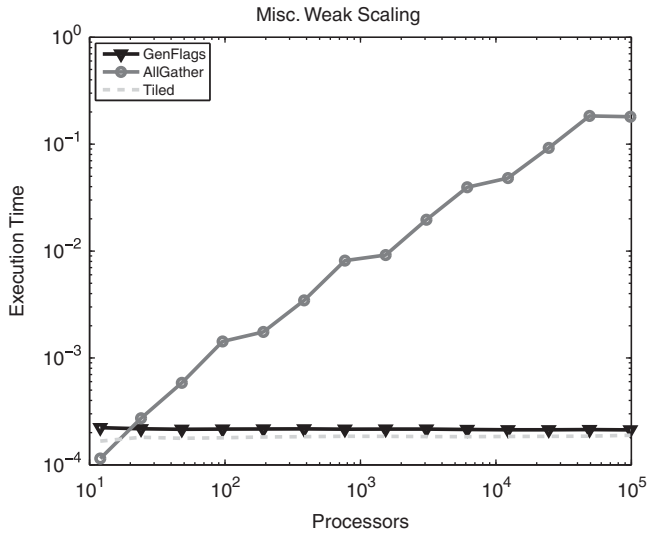

Figure 7. The weak scaling for miscellaneous operations that may be required by the framework. Generating the sparse flags list scales well, however, all-gathering the patch-set does not.

To scale onto petascale and eventually exascale machines applications must exhibit good weak scaling. This analysis has shown that both GBR algorithms do not perform well at large numbers of cores and will limit the scalability of the application. This is due to the cost of combining global data that is inherent in those algorithms. Both the tiled and LBR algorithms avoid this bottleneck by only operating on local data.

Figure 7 shows the experimental time for generating the flags list and performing an all-gather on the patch-sets. The experimental times for the tiled algorithm have also been included on this graph to provide a reference point.

This graph shows that the time to create the sparse flags scales well but is slightly slower than the tiled algorithm. In addition, the cost to all-gather the final patch-set is substantially higher than the cost of regridding. This shows that maintaining global metadata for the patch-sets will become a bottleneck at sufficiently large numbers of cores. Thus frameworks that currently utilize this global metadata will eventually need to find ways to eliminate this requirement by moving to local algorithms.

5. EXASCALE PERFORMANCE

It is expected that exascale computers will start appearing within the next decade. It is uncertain what architecture the first exascale machines will utilize. Though it is not unreasonable to assume that such machines may include upwards of 100 M cores. Using the models presented earlier we will now extend the performance analysis to include a machine like Kraken with 100 M cores. The time to split the patches for the GBR and LBR algorithms has been included in the projected times.

Figure 8 shows the theoretical strong scaling performance up to 100 M cores. In this graph $B_c = 1\,073\,741\,824$, $C = 4096 B_c$ and which corresponds to having around one 16^3 coarse patch per core at 100 000 000 cores. This graph shows that the LBR algorithm scales well but tails off as the number of patches per processor becomes low due to the cost of splitting. Since splitting does not scale ideally it eventually becomes a bottleneck thereby limiting strong scaling. The tiled algorithm continues to show ideal strong scaling. The GBR algorithms scale well at smaller numbers of processors but do not scale well at the largest numbers of processors. The GBR algorithms also take significantly longer than the LBR and tiled algorithms.

Figure 8 shows the theoretical weak scaling performance out to 100 M cores. For this graph $C = 16\,384\,P$ and $B_c = C/4096$, which corresponds to each core having four 16^3 coarse patches. This graph shows both the LBR and the tiled algorithms weak scale well. In this case, LBR scales

DOI: 10.1002/cpe

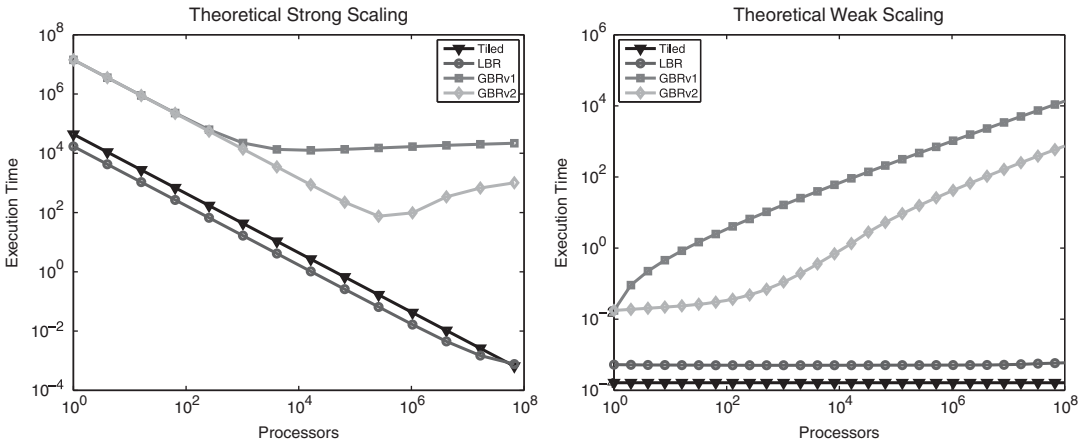

Figure 8. The theoretical strong and weak scaling to 100 M cores. The GBR algorithms do not strong or weak scale well, the LBR and the tiled algorithms both strong and weak scale well.

well because the $\log(P)$ in the splitting algorithm is fairly flat at large numbers of cores. The GBR algorithms do not scale well.

6. SUMMARY AND CONCLUSIONS

To move onto exascale machines the algorithms we use will have to exhibit nearly ideal weak scaling. We have shown that algorithms that utilize global metadata do not scale well in the strong or weak sense and are not feasible for large-scale machines. We have presented two new local algorithms which both show promising weak and strong scalability up to 98 304 cores. In addition, we have presented the analysis of these algorithms which shows they should weak scale up to 100 M cores.

It is commonly assumed that less patches is better. However, when weak scaling we have shown that if the number of patches does not scale linearly with the volume of the domain then a splitting algorithm will be required to ensure that there is at least one patch per core. This is the case with the Berger–Rigoutsos algorithms. This implies that the extra patches generated using the LBR algorithm over the GBR algorithms should not be a problem. In addition, the number of patches produced by the tiled algorithm scales linearly with the size of the domain and thus will not require the splitting algorithm.

The irregularity of the patch-sets produced by the GBR and LBR algorithms may be problematic, especially in terms of load imbalance. All the algorithms generate tight-fitting patch-sets thereby limiting over-refinement. The tiled algorithm generates a completely regular patch-set and that regularity could be exploited. We believe that exploiting this regularity could provide significant simplifications to frameworks along with large increases in performance and may be necessary to achieve good performance on exascale machines.

As we move toward exascale machines we will need to move away from inherently global algorithms and toward more local algorithms. Algorithms that operate on global metadata will eventually be limited by the cost to compute and communicate the global metadata. We have shown that the grid creation for BSAMR can be done completely locally, however, most frameworks still depend on the construction of global metadata. These frameworks should focus on eliminating this dependency in preparation for future petascale and exascale machines.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under the grant numbers OCI-0721659 and OCI-0905068.

DOI: 10.1002/cpe

REFERENCES

1. Diachin LF, Hornung R, Plassmann P, Wissink A. Parallel adaptive mesh refinement. *Parallel Processing for Scientific Computing*, Heroux MA, Raghavan P, Simon HD (eds.), ch. 8. SIAM: Philadelphia, PA, U.S.A., 2006.
2. Berger M, Oliger J. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics* 1984; **53**:484–512.
3. Berger MJ, Colella P. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics* 1989; **82**(1):64–84. DOI: http://dx.doi.org/10.1016/0021-9991(89)90035-1.
4. Berger MJ, Rigoutsos I. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man, and Cybernetics* 1991; **21**:1278–1286.
5. Berger MJ, Bokhari SH. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers* 1987; **36**(5):570–580. DOI: http://dx.doi.org/10.1109/TC.1987.1676942.
6. Wissink AM, Hysom D, Hornung RD. Enhancing scalability of parallel structured AMR calculations. *ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing*. ACM: New York, NY, U.S.A., 2003; 336–347. DOI: http://doi.acm.org/10.1145/782814.782861.
7. Gunney BTN, Wissink AM, Hysom DA. Parallel clustering algorithms for structured AMR. *Journal of Parallel and Distributed Computing* 2006; **66**(11):1419–1430. DOI: http://dx.doi.org/10.1016/j.jpdc.2006.03.011.
8. Wissink AM, Hornung RD, Kohn SR, Smith SS, Elliott N. Large scale parallel structured AMR calculations using the samrai framework. *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. ACM: New York, NY, U.S.A., 2001; 6. DOI: http://doi.acm.org/10.1145/582034.582040.
9. Luitjens J, Guilkey J, Harman T, Worthen B, Parker S. Adaptive computations in the Uintah framework. *Advanced Computational Infrastructures for Parallel/Distributed Adaptive Applications*, Parashar MLX (ed.). Wiley: Chichester, U.K., 2009; 171–199.
10. Burstedde C, Ghattas O, Stadler G, Tu T, Wilcox LC. Towards adaptive mesh PDE simulations on petascale computers. *TeraGrid'08*. ACM: New York, NY, U.S.A., 2008.
11. Burstedde C, Ghattas O, Gurnis M, Stadler G, Tan E, Tu T, Wilcox LC, Zhong S. Scalable adaptive mantle convection simulation on petascale supercomputers. *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press: Piscataway, NJ, U.S.A., 2008; 1–15.
12. Burstedde C, Ghattas O, Gurnis M, Isaac T, Stadler G, Warburton T, Wilcox LC. Extreme-scale AMR. *SC '10: Proceedings of the 2010 ACM/IEEE Conference on Supercomputing*. IEEE Press: Piscataway, NJ, U.S.A., 2010.
13. Parker SG, Guilkey J, Harman T. A component-based parallel infrastructure for the simulation of fluid structure interaction. *Engineering with Computers* 2006; **22**(3–4):277–292.
14. Straalen BV, Shalf J, Ligocki T, Keen N, Yang WS. Scalability challenges for massively parallel AMR applications. *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. IEEE Computer Society: Washington, DC, U.S.A., 2009; 1–12. DOI: http://dx.doi.org/10.1109/IPDPS.2009.5161014.
15. California RD, Deiterding R. Detonation structure simulation with AMROC. *International Performance Computing and Communications Conference*. Springer: London, U.K., 2005; 916–927.
16. MacNeice P, Olson KM, Mobarry C, deFainchtein R, Packer C. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications* 2000; **126**:330–354.
17. Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, MacNeice P, Rosner R, Truran JW, Tufo H. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series* 2000; **131**(1):273.
18. Burstedde C, Burtscher M, Ghattas O, Stadler G, Tu T, Wilcox LC. ALPS: A framework for parallel adaptive PDE solution. *Journal of Physics: Conference Series* 2009; **180**:012 009. DOI: 10.1088/1742-6596/180/1/012009.
19. Parker SG. A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Computer Systems* 2006; **22**(1):204–216. DOI: http://dx.doi.org/10.1016/j.future.2005.04.001.
20. Luitjens J, Berzins M. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. *IPDPS 2010 the 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society: Washington, DC, U.S.A., 2010.
21. Solomonik E, Kale LV. Highly scalable parallel sorting. *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing*. IEEE Computer Society: Washington, DC, U.S.A., 2010.
22. Luitjens J, Worthen B, Berzins M, Henderson T. Scalable parallel AMR for the Uintah multiphysics code. *Petascale Computing Algorithms and Applications*, Bader DA (ed.). Chapman and Hall/CRC: New York, NY, U.S.A., 2007; 67–82.
23. Deiterding R. Construction and application of an AMR algorithm for distributed memory computers. *Adaptive Mesh Refinement—Theory and Applications* (*Lecture Notes in Computational Science and Engineering*, vol. 41), Plewa T, Linde T, Weirs VG (eds.). Springer: New York, NY, U.S.A., 2005; 361–372.
24. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms* (2nd edn). MIT Press: Cambridge, MA, U.S.A., 2001.
25. Lötstedt P, Söderberg S, Ramage A, Hemmingsson-Frändén L. Implicit solution of hyperbolic equations with space–time adaptivity. *BIT Numerical Mathematics* 2002; **42**:134–158.
26. Berzins M, Luitjens J, Meng Q, Harman T, Wight C, Peterson J. Uintah a scalable framework for hazard analysis. *TeraGrid'10*. ACM: New York, NY, U.S.A., 2010.
27. Meng Q, Luitjens J, Berzins M. Dynamic task scheduling for the Uintah framework. *Proceedings of the Third IEEE Workshop on Many-task Computing on Grids and Supercomputers* (*MTAGS10*). IEEE Computer Society: Washington, DC, U.S.A., 2010.

28. Klosowski JT, Held M, Mitchell JSB, Sowizral H, Zikan K. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics* 1998; **4**:21–36.
29. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS '67* (*Spring*): *Proceedings of the 18–20 April 1967*, *Spring Joint Computer Conference*. ACM: New York, NY, U.S.A., 1967; 483–485. DOI: http://doi.acm.org/10.1145/1465482.1465560.

DOI: 10.1002/cpe