

Multi-Jagged: A Scalable Parallel Spatial Partitioning Algorithm

Mehmet Deveci, Sivasankaran Rajamanickam, *Member, IEEE*,
Karen D. Devine, and Ümit V. Çatalyürek, *Senior Member, IEEE*

Abstract—Geometric partitioning is fast and effective for load-balancing dynamic applications, particularly those requiring geometric locality of data (particle methods, crash simulations). We present, to our knowledge, the first parallel implementation of a multidimensional-jagged geometric partitioner. In contrast to the traditional recursive coordinate bisection algorithm (RCB), which recursively bisects subdomains perpendicular to their longest dimension until the desired number of parts is obtained, our algorithm does recursive multi-section with a given number of parts in each dimension. By computing multiple cut lines concurrently and intelligently deciding when to migrate data while computing the partition, we minimize data movement compared to efficient implementations of recursive bisection. We demonstrate the algorithm’s scalability and quality relative to the RCB implementation in Zoltan on both real and synthetic datasets. Our experiments show that the proposed algorithm performs and scales better than RCB in terms of run-time without degrading the load balance. Our implementation partitions 24 billion points into 65,536 parts within a few seconds and exhibits near perfect weak scaling up to 6K cores.

Index Terms—Geometric partitioning, spatial partitioning, recursive bisection, jagged partitioning, load balancing



1 INTRODUCTION

In many areas of science and engineering, problem and dataset sizes are increasing rapidly due to technological advances that enable generation and storage of such data. This growth is fueled by our urge to better understand more complex phenomena. Many of the datasets are irregular, in the sense that no immediately observable or predictable pattern exists among data elements. Lack of such patterns makes it difficult to find simple, effective and efficient mechanisms to partition data for scalable parallel processing, as well as indexing and storage. While some data, like interaction networks (e.g., gene/protein interaction networks, social networks, task dependency networks) are defined only by interactions among entities, a significant amount of data includes spatial coordinates together with rules about how the elements interact based on those spatial coordinates. This paper deals with efficient partitioning of the latter datasets for parallel processing.

With recent increases in computing-system sizes, scalable partitioning for distributing data and tasks to processors has become critical to application performance. Researchers have been developing partitioning techniques for interaction-based datasets, in the form of graph and hypergraph partitioning ([5], [8], [14], [15], [17], [18], [30]) and for spatial datasets ([3], [13], [24], [28], [29]) for more

than four decades. Yet the complexity of the problems (many are NP-hard) makes it challenging to develop both effective and scalable solutions. For many applications for which spatial information is available, spatial partitioning is preferred over interaction-based (also called connectivity-based) methods because of its speed, despite the fact that spatial partitioners do not provide an exact model for the communication cost of most applications. Scalability is crucial especially when dataset sizes and architectural changes require applications to use massively parallel machines to obtain effective solutions. The scalability of the partitioner is an important part of the applications’ scalability, especially in applications needing dynamic load balancing; time to partition should scale at the same rate as the application’s computation time.

Partitioning in parallel is a chicken-and-egg problem. We assume the data is already distributed in the memory of the parallel machine by the application and there is no replication. This initial data distribution directly affects at least the execution time of the partitioner, and in some implementations might affect the the partition quality. If the initial data distribution is not load balanced, some of the processors may have significantly more work than others, leading to load imbalance while computing the partition. Some successful parallel partitioners, such as the recursive coordinate bisection (RCB) [3] implementation in Zoltan, migrate data while partitioning, which improves both load balance and data locality during partitioning. As its name implies, RCB obtains a partition of a dataset by recursively bisecting the domain into two parts with cutting planes orthogonal to a coordinate axis, so that the weight of data on each side of the cutting plane is equal. Migration happens after each bisection, and hence, depending on the initial data distribution, it could result in significant data movement during partitioning.

- M. Deveci is with the Departments of Biomedical Informatics, and Computer Science & Engineering, The Ohio State University, Columbus, OH. E-mail: mdeveci@bmi.osu.edu
- S. Rajamanickam and K.D. Devine are with the Computer Science Research Institute in the Center for Computing Research at Sandia National Laboratories, Albuquerque, NM. E-mail: {srajama,kddevin}@sandia.gov
- Ü.V. Çatalyürek is with the Departments of Biomedical Informatics, Electrical & Computer Engineering and Computer Science & Engineering, The Ohio State University, Columbus, OH. E-mail: umit@bmi.osu.edu

Manuscript received July 2014; revised October 2014 and February 2015.

In this paper, we propose an efficient parallel geometric partitioning algorithm for multi-dimensional datasets. Our algorithm computes partitions by recursively multi-sectioning the dataset. Hence, our algorithm, *multi-jagged* (MJ), can be viewed as generalization of two-dimensional jagged partitions (also called Semi-Generalized Block Distribution) [22], [27], [35] to multiple dimensions. There are many efficient heuristics and optimal, sequential algorithms [22], [27], [29] to compute two-dimensional partitions of dense two-dimensional arrays, where each element represents the load of its respective object in two-dimensional space. We aim to partition objects with real-valued coordinate information given in multi-dimensional space. Our algorithm can be also viewed as a generalization of RCB in which we perform multi-sections instead of bisections in each level of recursion; the number of sections can be given by the user (or computed by us). Hence, one can use our algorithm to perform RCB. However, as shown in [29], by considering multiple cut-lines concurrently, heuristic jagged partitioning generally yields better load balance (especially for workloads with non-uniform weights) than greedy recursive bisection algorithms that assume a good bisection at each level yields a good final partition. As we will show, in a parallel environment, our algorithm also scales better than RCB due to reduced data movement.

We have implemented our algorithm in the Zoltan2 [4] framework, which is a revision of the Zoltan library [10] exploiting advances in compiler technology and software design principles. We have experimentally evaluated our implementation of the new algorithm against Zoltan’s robust RCB implementation, which is used successfully in many large scale applications [31], [32], [11]. The experiments on real and synthetic datasets show that the proposed algorithm performs and scales better than the existing RCB method in terms of run-time without degrading the load balance. We also compare the communication patterns of the resulting partitions, and show that, although multi-sectioning has a larger theoretical upper-bound, the two methods produce similar communication patterns in practice.

Our MJ algorithm is already in use in several scenarios. For example, it repartitions coarsened matrix operators in the multigrid solver MueLu [9], so that coarse-matrix operations are both balanced and scalable. In this context, MJ partitioned 22.3 million coordinates into 20,736 parts on 131K cores as part of a nine billion element simulation of low Mach number fluid flow with unstructured meshes on DOE’s Cielo Cray XE6 supercomputer [19]. In recent studies, Lin et al. used MJ to partition 78.4 million coordinates into 78K parts on 524K cores of DOE’s Sequoia BlueGene/Q supercomputer [20]. They demonstrate better application performance and scaling using MJ in the second-generation Trilinos stack, when compared to RCB in the first-generation Trilinos stack. In addition, our new algorithm is used for architecture-aware placement of MPI ranks onto allocated cores of a supercomputer to reduce network congestion and application communication costs. Our geometric partitioner orders both allocated nodes and applications’ MPI ranks in a consistent way that places interdependent ranks in “nearby” cores. For a structured finite-difference proxy-application [2], mapping with MJ reduced execution time

by 34% on average on 65,536 cores of Cielo [7].

The rest of the paper is organized as follows. Background and related work are presented in Section 2. Section 3 includes the details of our proposed parallel multi-dimensional jagged algorithm. Experimental evaluation is presented in Section 4. We conclude in Section 5.

2 BACKGROUND

Spatial partitioners use only geometric information from the application to compute partitions. They have several advantages over graph- or hypergraph-based methods. Spatial partitioners typically have lower runtimes to compute partitions. They assign physically close objects together within a processor; this feature is important for applications such as contact detection and particle methods for which geometric locality of data is important. And they can be used in applications where connectivity information is unavailable, such as particle methods or visualization.

The input to spatial partitioners includes a set of physical coordinates (e.g., (x, y, z) in 3D) and, optionally, weights associated with each point. Additionally, desired part sizes can be provided; the algorithmic details of handling nonuniform part sizes are straightforward and will not be presented in this paper. The output of spatial partitioners is a list of part numbers to which the input points are assigned.

Formally, given the coordinates $C_{i,j}$ and weights W_j of N data points in d dimensional space ($0 \leq i < d$ and $0 \leq j < N$), and the number of parts κ and maximum allowed imbalance ratio ε , the spatial (coordinate) partitioning problem can be defined as finding κ non-overlapping bounding boxes $B_k = \{b_{1-}, b_{1+}, b_{2-}, b_{2+}, \dots, b_{d-}, b_{d+}\}$ for all parts $1 \leq k \leq \kappa$ that cover all data points. A data point j is said to be in part k , i.e., $\mu_j = k$, if and only if it is inside the bounding box B_k , i.e., $b_{i-} < C_{i,j} < b_{i+}$ for all $0 \leq i < d$. The goal is to achieve balanced parts defined as:

$$\sum_{j, 0 \leq j < N \wedge k = \mu_j} W_j \leq (1 + \varepsilon) W_{avg}$$

for all parts $1 \leq k \leq \kappa$, where ε is a user-specified imbalance tolerance, and W_{avg} is the average weight defined as $W_{avg} = \frac{\sum_j W_j}{\kappa}$. In addition to achieving even load distribution, spatial partitioning aims to reduce the communication overhead of the application. However, without precise connectivity information, spatial partitioning implicitly reduces communication by increasing geometric locality. Therefore, communication can be estimated by metrics such as the length of the bounding boxes (approximating total communication volume), or the number of neighboring boxes (approximating the total number of messages).

The complexity of spatial partitioning increases with the number of dimensions in the data, the existence of weights for the data, and the data’s density/sparsity. For dense one-dimensional (1D) partitioning (also known as *chains-on-chains* partitioning), Pinar and Aykanat [28] provide an extensive theoretical and experimental comparison of 1D algorithms. For dense two-dimensional partitioning (2D), Saule et al. [29] present a survey of existing methods and introduce new, more effective variants. For example, *rectilinear partitioning* (also known as *general block distribution* [1])

partitions 2D space into a $P \times Q$ mesh with non-uniform mesh spacing such that the load of each part is balanced. This problem is NP-hard [12], and, hence, efficient heuristics are proposed [24]. $P \times Q$ jagged-partitioning [27], [35] relaxes rectilinear partitioning such that space is first partitioned into P row (or column) stripes, and then each stripe is partitioned independently into Q parts. In another variant called m -way Jagged partitioning [29], the number of parts while partitioning P stripes is decided based on the load of each stripe. Octrees [21], [23] and space-filling curves [25], [26], [36] have been used for two- and three-dimensional spatial partitioning; both produce partitions with similar quality. Each point is assigned an octant or a space-filling curve key, respectively, based on its position in space; a 1D traversal of the octree or space-filling curve is then partitioned into equally weighted parts.

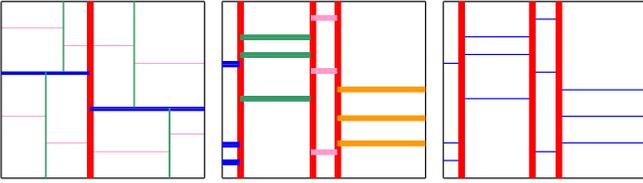


Fig. 1. A partition of size 16 using (left) RCB, (middle) MJ with no migration and a 4×4 configuration, and (right) MJ with migration. The order of the cut computations and the number of processors participating are indicated by colors (red first, then blue, green, pink, and orange) and line thickness (thickest to thinnest for decreasing number of processors), respectively. The two MJ variants produce identical partitions, as they compute the same number of cuts in each dimension; they differ only in the amount of concurrency during partitioning.

The most popular techniques for spatially partitioning dense or sparse data arguably are variants of *recursive bisection*. In Recursive Coordinate Bisection (RCB) [3], a cutting plane perpendicular to a coordinate axis is computed such that the total weight of data on each side of the cutting plane is equal; the resulting two subregions are then recursively divided until the desired number of parts is obtained. (When the number of desired parts is not a power of two, parts can be obtained by simply adjusting split ratios in each bisection.) Figure 1(left) shows an example using RCB to find 16 parts. The thick, red line represents the initial bisection; progressively thinner blue, green, and pink lines represent subsequent cuts in the recursion. Several alternatives for selecting the dimension (x , y , or z) of each cutting plane [29] are possible, such as alternating dimension or longest dimension. In most common implementations, the longest dimension of the bounding box including the data is chosen to be cut. The dimension is selected independently for each bisection, with the intent that subdomains' aspect ratios approach one. Principal axes of the geometric data can be used instead of the coordinate axes; this variant is called Recursive Inertial Bisection [33], [34].

The main kernel of RCB is finding the cutting plane. Zoltan's parallel implementation uses a binary-search approach to finding the median. The minimum and maximum data coordinates in the cut direction are found, and the initial cutting plane is taken as the average of the minimum and maximum. The weight of points to the left and right of the cut is computed via a parallel reduction operation,

and the cut is moved left or right depending on whether the left side's load is heavier or lighter, respectively, than the right's. This process repeats until the left and right side are equally weighted. Information about points closest to the cut is included in the reduction operations to allow small amounts of weight to be shifted between parts and, potentially, avoid additional iterations by satisfying the balance criterion immediately.

Once the median is found, data is reorganized into left and right subgroups for recursion. The algorithm recursively partitions the subgroups each with its own new bounding box and a new cut dimension.

In the serial implementation, points are not moved when the left and right subgroups are formed; instead, an index array points to the data in each subgroup. But in parallel, points are migrated into subgroups based on their positions relative to the cutting plane. The processors are divided into two sets; points to the left of the cut are sent to processors in one set while points to the right are sent to processors in the other. Only coordinate information is sent, not all application data associated with the points. The amount of data sent depends on the input distribution of the data. When doing dynamic partitioning to correct small imbalances or changes in locality, the migration cost is expected to be small. Static partitioning with arbitrarily distributed data, however, can result in significant migration costs.

Once the data is migrated, multiple cuts can be computed independently and in parallel by using different subcommunicators for the processors in different sets. Reduction operations to compute bounding boxes and accumulate weights on each side of a cutting plane during the median-finding routine are performed within the subcommunicators. When the subcommunicator size is one, the implementation reverts to the serial algorithm. There are some similarities between this implementation of RCB and our Multi-Jagged implementation. The next section describes our algorithm and the implementation details.

3 MULTI-JAGGED: MULTI-DIMENSIONAL JAGGED PARTITIONING

The Multi-dimensional Jagged algorithm (MJ) is a geometric partitioning method in which a P_l -way multi-sectioning is applied recursively to partition a domain into $\kappa = \prod_{l=0}^{d'-1} P_l$ parts for a given recursion depth d' . Figure 1(middle) shows a 16-part decomposition of a 2D domain generated by MJ with $d' = 2$ and $P_0 = P_1 = 4$. Red lines show the first four-way multi-section; each of the four subdomains is then divided into four, as shown by the horizontal lines.

The goal for MJ is to improve upon parallel RCB to make a scalable algorithm for very large data sets and for large numbers of parts. MJ has several key differences from RCB.

First, MJ reduces the depth of recursion by doing multi-section instead of bisection. To multi-sect, say, the x -dimension into P_l parts, the initial cut lines are spaced evenly between the minimum and maximum x -coordinates of the input data. The weight of points in the parts between pairs of adjacent cut lines is computed, and the cut lines are moved in order to adjust the weight in the parts. This process iterates until the parts all have the same weight.

Second, to improve scalability, MJ has several options that allow trade-offs between computation and data migration. MJ has the ability to avoid entirely the data migration costs that are inherent in parallel RCB. All processors cooperate to compute cuts at all levels of the recursion; subcommunicators are not used as in RCB. This mode is illustrated in Figure 1(middle), where the horizontal cuts in each subdomain are computed in sequence by all processors. Alternatively, MJ can migrate data to intermediate subdomains as in RCB to allow concurrent computation of cuts during the recursion; in Figure 1(right), the horizontal cuts are computed concurrently in subcommunicators. Each mode has advantages, depending on the distribution of the input. In addition, MJ can perform “smart” migration, switching between modes based on the imbalance of the partitioner, predictions of the number of global reductions needed during partitioning, and estimates of the future communication-computation bounds. For example, migration might be very expensive for datasets in which the points are scattered randomly to the processors, while it is expected to be less expensive for datasets with localized points in processors. MJ favors migration for the latter, while it avoids migration for the former. Details of this “smart” migration heuristic are in Section 3.2.

Third, unlike Zoltan’s RCB which uses only MPI for parallelism, MJ is implemented using a hybrid MPI+OpenMP paradigm. This implementation allows MJ to fully exploit multicore architectures to provide dynamic load balancing to applications running with MPI+threads. While the multithreaded implementation is not the focus of this paper, it is critical for hybrid applications in which Zoltan’s single-threaded RCB implementation would result in idle cores during partitioning. Moreover, since MJ’s hybrid implementation uses fewer processes, it may reduce data movement during partitioning.

3.1 The Basic MJ Algorithm

Algorithm 1 gives the overall description of MJ. The algorithm takes a dataset of n d -dimensional coordinates C , computational weights W , and a vector P of size d' which is an array of the number of parts desired at each recursion level. MJ computes a partition with maximum imbalance ε , by partitioning the coordinates into $\kappa = \prod_{l=0}^{d'-1} P_l$ parts in d' steps. In order to avoid data movement, MJ maintains a permutation array ($Permute$) of length n . Coordinate data is not rearranged during partitioning; instead, $Permute$ stores the indices of the coordinates, and part assignments are made by reordering $Permute$. $xPerm$ maintains the beginning and end indices of each part in $Permute$. Initially, there is a single part with all coordinates. Then, P is traversed, and the dimension on which the partitioning will occur (i) is determined in a round-robin fashion. 1DPART is then called for all available parts (i.e., for κ which is initially 1), and partitioning information is stored in μ . UPDATEPERM (not shown) updates the $Permute$ array, and stores the beginning indices of new parts in $newxPerm$ according to the part assignment information. Once all parts are partitioned along a dimension, the number of parts κ increases by factor of P_l . At this point, the algorithm estimates whether migrating the coordinates is good for the

Algorithm 1: Parallel MULTI-JAGGED Algorithm (MJ)

```

Data:  $d, n, C_{d,n}, W_n, \varepsilon, d', P_{d'}$ 
for  $j$  from 0 to  $n - 1$  do
   $Permute_j \leftarrow j$ ; // Initialize permutation
 $\kappa \leftarrow 1$ ; // All of  $C$  is in one part
 $xPerm_0 \leftarrow 0$ ;
 $xPerm_1 \leftarrow n$ ;
for  $l$  from 0 to  $d' - 1$  do
   $i \leftarrow l \bmod d$ ; // Dimension to partition
  // Compute  $P_l$  parts within each current
  part
  for  $p$  from 0 to  $\kappa - 1$  do
     $pBegin \leftarrow xPerm_p$ ;
     $pEnd \leftarrow xPerm_{p+1}$ ;
     $\mu \leftarrow$ 
    1DPART( $C_{i,*}, W, Permute, pBegin, pEnd, \varepsilon, P_l$ );
     $newxPerm \leftarrow$ 
    UPDATEPERM( $Permute, pBegin, pEnd, \mu$ );
   $\kappa \leftarrow \kappa \times P_l$ ;
  CHECKANDMIGRATE( $n, C_{*,*}, W, \mu, newxPerm, \kappa$ );
   $xPerm \leftarrow newxPerm$ ;

```

execution time or not by using CHECKANDMIGRATE where, if needed, the actual migration operation is also performed and the subcommunicators are created. After this point, the partitioning continues in the next dimension.

The pseudocode of 1DPART is given in algorithm 2. 1DPART finds P_l part assignments with ε imbalance for the given single dimensional coordinates associated with weights W . As the first step, GETINITIAL (not shown) finds the minimum and maximum coordinate together with the total weight of the part ($C_i^{min}, C_i^{max}, W_{tot}$) by using a single parallel reduction REDUCEALL. Initial cut coordinates are assigned as uniform slices between C_i^{min} and C_i^{max} . Moreover initial parts are assigned with following formula:

$$\mu_j = \left\lfloor (C_{i,j} - C_i^{min}) \times \frac{P_l}{C_i^{max} - C_i^{min}} \right\rfloor$$

1DPART maintains two arrays ($hiBounds, loBounds$) to store the upper and lower coordinates bounds of each cut. Flags indicating whether a cut position is finalized or not are saved in the $done$ array. After initialization, the iteration at line A continues until all cut positions are finalized. In every step, the part weights ($pWeights$) are calculated by calling GETPARTWEIGHTS; then according to this part weight information, GETNEWCUTS either finalizes a position of a cut, or makes a new estimate for the new cut position.

The most computationally expensive portion of our algorithm is deciding the part to which all coordinates belong. This operation is $O(n)$ in RCB. However, it is $O(n \log(P_l))$ when using multi-section, as each point has to be placed in one of the P_l parts, which can be done in a binary-search-like fashion. Although computational cost is slightly increased, finding multiple cuts in each iteration reduces the needed communication cost, and, hence, yields faster execution.

Algorithm 3 outlines GETPARTWEIGHTS. GETPARTWEIGHTS initializes the part weights of non-finalized parts to 0. It computes the part assignment for all coordinates by traversing the coordinates and comparing them to the cut lines. To reduce the computational cost, part information from the previous iteration is used as initial

Algorithm 2: 1DPART

```

Data:  $C_i, W, Permute, pBegin, pEnd, \varepsilon, P_l$ 
// Get initial weights, min, max across
  all parts;
// requires one REDUCEALL operation
( $cuts, \mu, W_{tot}, C_i^{min}, C_i^{max}$ )  $\leftarrow$ 
GETINITIAL( $C_i, W, Permute, pBegin, pEnd, P_l$ );
 $cutCnt \leftarrow P_l - 1$ ;
for  $p$  from 0 to  $cutCnt$  do
   $hiBounds_p \leftarrow C_i^{max}$ ;
   $loBounds_p \leftarrow C_i^{min}$ ;
   $done_p \leftarrow False$ ;
 $leftCutCnt \leftarrow cutCnt$ ;
// Find  $P_l - 1$  cut positions
A while  $leftCutCnt > 0$  do
   $pWeights \leftarrow$  GETPARTWEIGHTS( $C_i, W, Permute,$ 
     $pBegin, pEnd, cuts, cutCnt, \mu$ );
  ( $cuts, cutReduction$ )  $\leftarrow$ 
  GETNEW CUTS( $W_{tot}, pWeights, \varepsilon, cuts, cutCnt,$ 
     $hiBounds, loBounds, done$ );
   $leftCutCnt \leftarrow leftCutCnt - cutReduction$ ;
return  $\mu$ ;

```

part estimates for the binary search. Once a coordinate's part is found, its part assignment and the part's weight are updated accordingly. If the left and right cut lines of the part in which the coordinate lies are finalized, no further calculation is done for the coordinate. During these iterations, we also store the left and right closest coordinates to each cut line. This detail is omitted in the pseudocode for simplicity. This information is used later in GETNEW CUTS to be able to skip any huge holes in the input domain. Once the part weights are known in each process, a prefix-sum operation is performed on the $pWeights$, and a single REDUCEALL operation computes the global part weights. MJ can then evaluate each cut position independently of other cut positions; each cut can be evaluated by comparing the weight on its left and right.

Once the global part weights are known, the GETNEW CUTS function (Algorithm 4) determines whether the cut positions are final. If the cuts are to be moved, it makes a new guess for the new cut coordinates. GETNEW CUTS traverses all the cuts and skips the ones that are already finalized. For each cut that is not finalized, it computes the expected weight (ew) and the imbalances to the left and right of the cut (li, ri). If they are both smaller than ε , the position of the cut is finalized. Otherwise, a better upper or lower bound is computed and the new cut position is estimated using them.

In order to compute a better upper or lower bound, let us assume without loss of generality that the weight on the left side of a cut is less than the expected weight; the operations are simply reversed when the weight on the right side is less than its expected weight. When the weight on the left side of a cut is less than the expected weight, the lower bound for where the cut could be is set to the cut's position, as that position is our best estimate so far. In order to compute a better position for the cut, the cuts on the right side of the current cut are traversed in TIGHTENBOUNDS (not shown). If a right-side cut ($cuts_{p'}$) with an equal weight to the expected weight is found, the new cut coordinate

Algorithm 3: GETPARTWEIGHTS

```

Data:  $C_i, W, Permute, pBegin, pEnd, cuts,$ 
   $cutCnt, done, \mu$ 
for  $p$  from 0 to  $cutCnt$  do
  if not ( $done_p$  and  $done_{p-1}$ ) then
     $pWeights_p \leftarrow 0$ ;
for  $j$  from  $pBegin$  to  $pEnd - 1$  do
   $j \leftarrow Permute_j$ ;
   $p \leftarrow \mu_j$ ;
  // If the cuts on left and right are
  // finalized, skip part search;
  if ( $done_p$  and  $done_{p-1}$ ) then
    continue;
  // Perform binary search starting from
  // the previously assigned part index;
   $p \leftarrow$  BINARYSEARCH( $p, cuts, C_{i,j}$ );
   $pWeights_p \leftarrow pWeights_p + W_j$ ;
   $\mu_j \leftarrow p$ ;
// Prefix-sum;
for  $p$  from 1 to  $cutCnt$  do
   $pWeights_p \leftarrow pWeights_{p-1} + pWeights_p$ ;
REDUCEALL( $pWeights$ );
return  $pWeights$ ;

```

is set to $cuts_p$. If the weight of a right-side cut is greater than the expected weight and is less than the current upper bound, the upper bound is tightened to this position. When the weight of a right-side cut is lower than the expected weight, it is a better lower bound than before.

After the new upper and lower bounds are determined, the function estimates a new position for the cut. While doing that, the weights of the upper and lower bounds are used (omitted in the algorithm for simplicity), and an estimate is computed by assuming uniform distribution of weights between the upper and lower bounds. If the new estimated cut position is the same as the previous cut position, the imbalance tolerance cannot be achieved; therefore, the cut coordinate is finalized. This situation occurs when there are coordinates with a total weight more than unit weight exactly on the cut line position. To achieve a balanced partition, some of the coordinates must be placed on the left side of the cut, while the rest of the coordinates must be placed on the right side.

3.2 Migration of the Coordinates

After a partition along a dimension is completed, the coordinates in each part can be localized in a smaller number of processors by performing migration. However, the volume of the migration operation is linear with the total number of coordinates; hence, migration might become a bottleneck for the partitioning algorithm. Therefore, MJ tries to avoid migration. When MJ does not migrate data, all MPI processes participate in computing the position of each cut. This requirement might reveal two problems.

Load Imbalance: MPI processes might have different numbers of coordinates in parts computed during the execution of MJ. Therefore, the MPI processes will have different loads in further partitioning of these parts. The worst case occurs when an MPI process does not have any points in a part. Even though the process does not have any points, it will

Algorithm 4: GETNEWCUTS

Data: $W_{tot}, pWeights, \varepsilon, cuts, cutCnt,$
 $hiBounds, loBounds, done$
 $cutReduction \leftarrow 0;$
for p **from** 0 **to** $cutCnt - 1$ **do**
 if $done_p$ **then**
 continue;
 $ew \leftarrow (p + 1) * W_{tot} / (cutCnt + 1);$
 $(li, ri) \leftarrow \text{IMBALANCEOF}(pWeights_p, p, W_{tot});$
 if $li < \varepsilon$ **and** $ri < \varepsilon$ **then**
 $done_p \leftarrow \text{True};$
 $cutReduction \leftarrow cutReduction + 1;$
 continue;
 else
 if $pWeights_p < ew$ **then**
 $loBounds_p \leftarrow cuts_p;$
 $(loBounds_p, hiBounds_p) \leftarrow$
 TIGHTENBOUNDS ($p + 1, cutCnt$);
 else
 $hiBounds_p \leftarrow cuts_p;$
 $(loBounds_p, hiBounds_p) \leftarrow$
 TIGHTENBOUNDS (0, p);
 $newCuts_p \leftarrow \text{GETNEWCUT}(loBounds_p, hiBounds_p);$

 if $newCuts_p = cuts_p$ **then**
 $done_p \leftarrow \text{True};$
 $cutReduction \leftarrow cutReduction + 1;$

 return ($newCuts, cutReduction$);

participate in the computation and communication. Taking such considerations into account, CHECKANDMIGRATE calculates the imbalance of the processors in each part. The function performs a single REDUCEALL to find the total number of coordinates in each part ($N_{*,*}$), and another REDUCEALL to calculate the imbalance which is defined as:

$$imbalance = \frac{\sum_{i=0}^{\rho} \sum_{j=0}^{\kappa} \frac{|N_{avg,j} - N_{i,j}|}{N_{avg,j}}}{\rho \times \kappa}$$

where ρ is the number of processors, $N_{avg,j} = \frac{N_{*,j}}{\rho}$ is the average number of coordinates over the processors in part j , and $N_{i,j}$ is the number of local points of processor i in part j . If MJ does not perform migration, all subsequent partitions are computed one after the other, with each taking less time if the work is balanced. As the imbalance increases, the cost of subsequent partitioning also increases. On the other hand, as imbalance increases, migration becomes cheaper, as imbalance suggests that the data is already localized, which reduces the volume of the migration. Therefore, MJ performs migration when this imbalance value is greater than a specified threshold. We choose this threshold empirically as 30% as a result of our initial experiments.

High Number of Global Messages: In a single call of the 1DPART function, the number of global messages is $m = 1 + it$, where it is the iteration count of the loop in line A of Algorithm 2. Assuming that it' is the average iteration count, and t is the total number of calls to 1DPART, the total number of messages becomes $t \times (it' + 1)$, in which t can be

computed as follows:

$$t = 1 + \sum_{l=0}^{d'-2} \prod_{l'=0}^l P_{l'}$$

For example, with $32 \times 32 \times 32 = 32,768$ parts, $t = 1057$. Although this t is small, the number of global messages might become problematic as the number of processors and partitioning dimensions d' increases. Therefore, throughout its execution, MJ limits the number of future REDUCEALL operations red_{opt} , and performs migration when $red_{opt} * \rho$ is greater than a specified value. We choose this threshold empirically as $1.5M$ as a result of initial experiments.

Communication-Bounded Last-Dimension Partitioning: Even if the processor imbalances are very low, migration may improve the partitioner's performance, since computation of cuts in the last dimensions may be communication bounded. There are $\prod_{l=0}^{d'-2} P_l$ parts at the beginning of the last step. In the best case (when $imbalance = 0$), each processor owns $N' = \frac{N}{\prod_{l=0}^{d'-2} P_l}$ in each part. For all parts, processors perform a REDUCEALL operation after comparing N' coordinates against $P_{d'-1} - 1$ cuts. If N' is small, the REDUCEALL among all processors dominates the execution time. For example, when $N = 500K$ and $\kappa = 32 \times 32 \times 32$, each processor owns only ~ 500 coordinates in its parts at the beginning of last dimension. Therefore, MJ forces migration of coordinates if $N' < 1000$.

The CHECKANDMIGRATE function checks the above three conditions, and decides whether or not to do migration. When doing the migration, two cases might occur. When $\rho \leq \kappa$, each processor can be given one or more parts. To minimize migration volume, a part is greedily assigned to the available processor with the largest number of coordinates in that part. When $\rho > \kappa$, several processors will be assigned to a single part. As before, the processors are greedily chosen according to their availability and the number of coordinates they have in that part. Since there are several processors assigned to a single part, the processors will perform communication operations in the further iterations. Therefore, a second idea would be to assign a part to the set of processors that are close to each other. Although this assignment might increase the volume of the migration, it might decrease the cost of future communication operations. However, the performance of these variants is architecture-dependent. Although MJ supports both of these processor-assignment heuristics, we use only the first one in our experiments, since there was no significant execution-time difference on the architecture we used.

After each processor determines to which processor it should send its data, migration is performed. A part might be assigned to a single processor, and the processors might own more than one part. In this case, further partitions of these parts are computed sequentially by the owner processor. On the other hand, if a part is assigned to a set of processors, a smaller subcommunicator is created, and further partitioning of that part is performed in this smaller subset of processors. Depending on the number of parts assigned to the processor, CHECKANDMIGRATE updates $newxPerm, xPerm$, and κ .

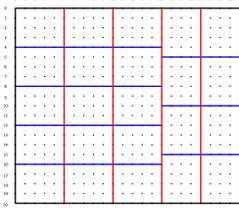


Fig. 2. MJ partition with part count $\kappa = 23$ and recursion depth $d' = 2$.

3.3 Partitioning into Arbitrary Numbers of Parts

For simplicity, section 3.1 explains a limited partitioning into $\kappa = P_0 \times P_1 \times \dots \times P_{d'-1}$ parts. However, MJ does not require κ to be a product of integers P_i . Instead, κ can be any number, even a prime number. Therefore, MJ can be defined in a more general way: MJ partitions the set of coordinates into a desired number of parts κ in a given number of steps (recursion depth d'). For example, Figure 1(left) can be viewed as MJ partitioning into $\kappa = 16$ in $d' = 4$ steps, while its other two partitions use $d' = 2$ steps.

When d' and a prime κ are given, MJ tries to choose multisection counts in each dimension that are as close as possible. MJ partitions the data into $P_0 = \lceil \kappa^{1/d'} \rceil$ along the first dimension. Then, for each of the parts obtained at the first level, its future number of parts is calculated and used as the part's expected weight. Figure 2 shows an example of MJ used to partition a dataset (with total weight 460 equal to its area) into $\kappa = 23$ parts in $d' = 2$ steps. Initially, MJ determines the number of sections that will partition the data along the first dimension; that is, $partArray_0 = \lceil 23^{1/2} \rceil = 5$. Then, MJ determines how many future parts will be obtained from each part. In this case, the first three parts will be partitioned into 5, while the last two parts will be partitioned into 4. The weights for the parts are adjusted according to the future number of parts, and MJ multisections the data that yields a partitioning into 5 parts with weights 100, 100, 100, 80 and 80, respectively. The first three parts and the last two are partitioned into 5 and 4 parts, respectively, yielding 23 parts, each with weight 20.

3.4 Properties of partitions from Multi-Jagged

In this subsection, we use some definitions used to analyze the properties of RCB [3]. The definitions are restated here for clarity. Each partition line in Figure 1 is subdivided into a number of *segments* by the incidence of other partition lines. For example, the first (red) cut line using RCB in Figure 1(left) is subdivided into seven segments. The *graph of a partition* is the dual graph obtained by representing each part by a vertex, and adding an edge between two nodes if and only if the corresponding regions share a segment.

For the analysis of spatial partitions, assume that each part is assigned to different processors. As a result, there are two metrics to analyze the quality of the partition: the maximum degree of a vertex in the partition graph and the total number of edges in the partition graph. The analysis in this section is restricted to partitioning in two dimensions $p \times q$, even though the multi-jagged algorithm works for arbitrary number of dimensions. To simplify the discussion, we say *stripes* are the regions corresponding to parts in the first dimension. For example, Figure 1(middle) has four

vertical stripes. Each stripe is further partitioned into q parts with $q - 1$ cut lines.

The upper bound for the maximum degree of a vertex in the partition graph is $2 \times q + 2$. This bound arises when any one part in stripe i ($1 < i < p$) shares its perimeter with all the q parts of each of the two neighboring stripes $i - 1$ and $i + 1$. This upper bound was observed previously in Manne [22]. If $p = q = 2^{k/2}$ for computing 2^k parts, the upper bound for the maximum degree of a vertex in the partition graph is $2 \times 2^{k/2} + 2$; this bound is worse than the upper bound for RCB which is $2^{k/2} + 2^{k/2-1} + 3$ (when k is even and $k \geq 4$) [3]. The difference between the two upper bounds is $2^{k/2-1} - 1$; however, we will show that MJ and RCB behave similarly in practice. When k is odd, MJ can reach the same upper bound as RCB by using $p = 2^{\lfloor k/2 \rfloor + 1}$ and $q = 2^{\lfloor k/2 \rfloor}$. Instead of using $p = q$, we can increase p to obtain a better bound in practice.

The total number of edges in the partition graph can be derived using a simple constructive argument. Within any one stripe i , there are q regions (or q vertices in the partition graph) and $q - 1$ cut lines (or $q - 1$ edges in the partition graph). There are p such stripes, resulting in $p \times (q - 1)$ edges corresponding to the cut lines in the second dimension.

The number of edges corresponding to the cut lines in the first dimension is the number of segments that could be created by joining p stripes. Consider the case with two stripes ($p = 2$). The least number of segments (q) on the cut line between the two stripes is obtained when for any j , $1 \leq j \leq q$, the j^{th} cut lines inside both stripes are collinear. Moving any one of the cut lines in the second dimension up/down increases the number of segments by at most two. There are $q - 1$ such cut lines, which gives us $2 \times (q - 1)$ as the maximum number of segments between two stripes. There are $p - 1$ such cut lines (or p stripes) giving us $(p - 1) \times (2 \times q - 1)$ segments. Adding the number of segments from the second dimension, the upper bound for the total number of edges in the partition graph is $p \times (q - 1) + (p - 1) \times (2 \times q - 1)$. When $p = q = 2^{k/2}$, the upper bound is $3 \times 2^k - 2^{k/2+2} + 1$, the same as the upper bound for RCB (when k is even).

4 EXPERIMENTAL RESULTS

We evaluated the performance of MJ on various real and synthetically generated datasets. We compared MJ's runtime and partition quality with those of RCB. All experiments were run on the Hopper supercomputer, a Cray XE6 at the National Energy Research Scientific Computing Center. Each compute node on Hopper has two twelve-core AMD "MagnyCours" processors running at 2.1GHz, with 32GBs of memory (slightly more than 1GB of memory per core). Hopper has a Cray "Gemini" interconnect for internode communication.

We used Zoltan and Zoltan2 [10] to test RCB and MJ, respectively. Zoltan (hence, RCB) is implemented in C; Zoltan2 (hence, MJ) is in C++. We used the gcc compiler (version 4.7.1) with -O3 optimization flag, and the OpenMPI library (version 1.4.5). For fair comparisons with RCB (which does not have multithreading support), we ran all experiments with one MPI rank per core.

Experiments were run on four synthetic and three real datasets, whose properties are shown in Table 1. Uniform

and `Normal` are 2D datasets whose (x, y) coordinates are randomly distributed with uniform and normal distributions, respectively. `2DANorm` is generated by taking the absolute values of (x, y) drawn from normal distributions; a circular hole in the domain creates a large empty region (see Figure 3), making this dataset challenging to find cut positions with linear prediction methods. `3DANorm` is the three-dimensional analogue of `2DANorm`. Two real datasets (`huge-bubbles` and `europa_osm`) are from the University of Florida sparse matrix collection [6]. The `tetraMesh` dataset is a 3D mesh of the Greenland ice sheet [16].

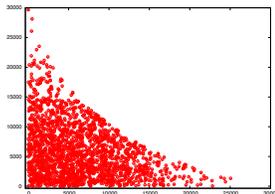


Fig. 3. An example of a 2DANorm dataset

TABLE 1
Properties of datasets for the experiments

Type	Name	Number of Points	d	Sequential RCB Time (seconds)
Weak Scaling				
Synthetic	Uniform	4M/proc	2	6.39
	Normal	4M/proc	2	7.23
	2DANorm	4M/proc	2	7.18
	3DANorm	4M/proc	3	7.68
Strong Scaling				
Synthetic	Uniform	50M	2	96.0
	Normal	50M	2	114.9
	2DANorm	50M	2	114.3
	3DANorm	50M	3	120.2
Real	hugebubbles-00020	21.2M	2	35.7
	europa_osm	50.9M	2	96.8
	tetraMesh	94.7M	3	102.0

The execution time of the partitioning algorithm depends highly on the initial distribution of the data. Thus, we studied scaling on datasets with different initial distributions of the coordinates among the processors. For the synthetic datasets, each processor generated points across the entire domain according to the specified distribution. Thus, in this default initial distribution, each processor had points from across the entire domain. After partitioning along a dimension, if further partitioning was localized via migration, migration volume would be very large. On the other hand, if localization was avoided and further partitioning was done by all processors, the workloads of the processors would be quite balanced, as all processors would have roughly equal numbers of points in all regions of the domain and, thus, in each part.

In our experiments, we compared this default initial distribution with data that was localized via pre-partitioning. As before, the processors generated their coordinates randomly. Then RCB was used to partition the datasets into the given number of *processors*. All of the coordinates were then migrated to their assigned processor in this pre-partition,

and the resulting localized data distribution was used as the initial distribution for MJ and RCB. In this scenario, MJ and RCB might still need to migrate data. For example, MJ might require migration since pre-partitioning was done with RCB, while RCB might require migration since the pre-partition had ρ parts and the target partition had κ parts.

4.1 Partition Quality

We used two metrics to evaluate the quality of a partition: load balance and communication costs for the application.

Load balance is the goal of both MJ and RCB, and both did well in this metric. As the data used in the experiments in this paper have unit weights, both algorithms found perfectly balanced partitions. Therefore, their partition quality in terms of load balance was the same.

Neither RCB nor MJ minimize the application communication costs metric explicitly; instead, enforcing geometric proximity of data within processors is meant to implicitly keep communication costs low. To estimate application communication costs, we computed the maximum and total number of messages induced by computed partitions. Since each neighbor represents a communication message, we counted the number of neighbors of each part and reported the maximum number of messages per part, and the total number of messages over the entire partition. In this experiment, we ran MJ with $d' = d$ (denoted by MJ) and with $d' = 2d$ (denoted by MJ'). 2D datasets are partitioned into 256 (16×16 and $4 \times 4 \times 4 \times 4$ for MJ and MJ'), respectively), 4,096 (64×64 and $8 \times 8 \times 8 \times 8$) and 65,536 (256×256 and $16 \times 16 \times 16 \times 16$), while 3D datasets are partitioned into 512 ($8 \times 8 \times 8$ and $2 \times 2 \times 2 \times 4 \times 4 \times 4$), 4,096 ($16 \times 16 \times 16$ and $4 \times 4 \times 4 \times 4 \times 4 \times 4$) and 32,768 ($32 \times 32 \times 32$ and $4 \times 4 \times 4 \times 8 \times 8 \times 8$).

Figure 4 and Figure 5 give the results for 2D and 3D datasets, respectively. As seen in the figures, the total number of messages is similar for RCB and MJ variants. This result is supported by the theory in Section 3.4. Even though the maximum number of messages is worse with MJ than with RCB, it is much lower than the theoretical worst-case bounds. For example, for 65,536 parts with 256×256 partitioning, the theoretical worst-case maximum number of messages with MJ is $2 \times 256 + 2 = 514$, while the actual number is 109 or less in all our experiments. Moreover, in most of the cases, the maximum number of neighbors can be reduced by increasing the recursion depth, as MJ' generally has lower maximum numbers of neighbors than MJ.

4.2 Weak Scaling

Weak scaling experiments were performed using the synthetic datasets. Every processor was given 4M coordinates, and the number of processors ranged from 1 to 6144. Since the number of points per processor, and hence, the work, was kept constant, the ideal result would be constant execution time as the number of processors increased. In practice, however, we expect small increases in the execution time as we increase the processor count because of increased communication costs as the number of processors increases. In this experiment, 2D datasets were partitioned into 65,536

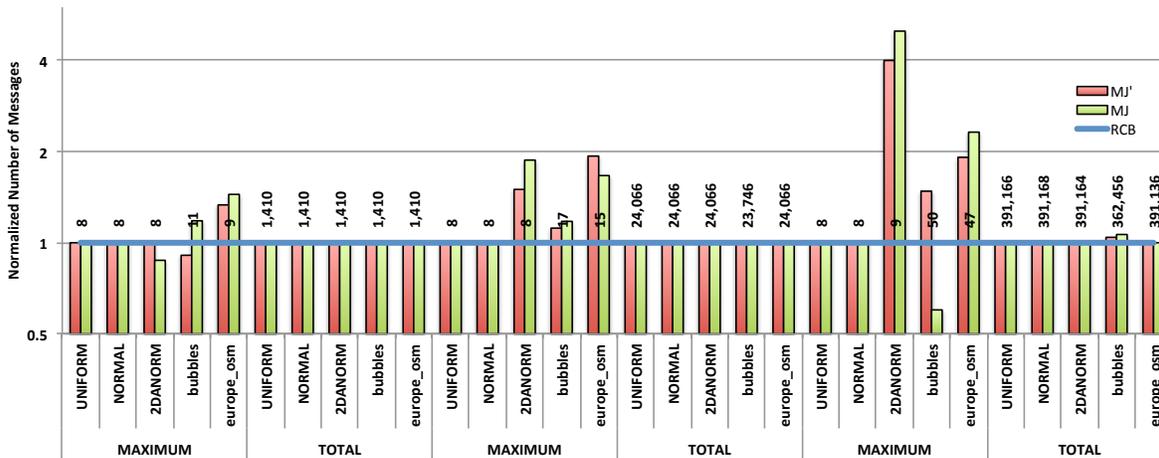


Fig. 4. Normalized maximum and total number of messages in MJ, MJ' and RCB partitions of 2D datasets. For each instance, the results with MJ and MJ' are normalized with respect to RCB's results, which are listed next to the corresponding figure bars. (Lower values are better.)

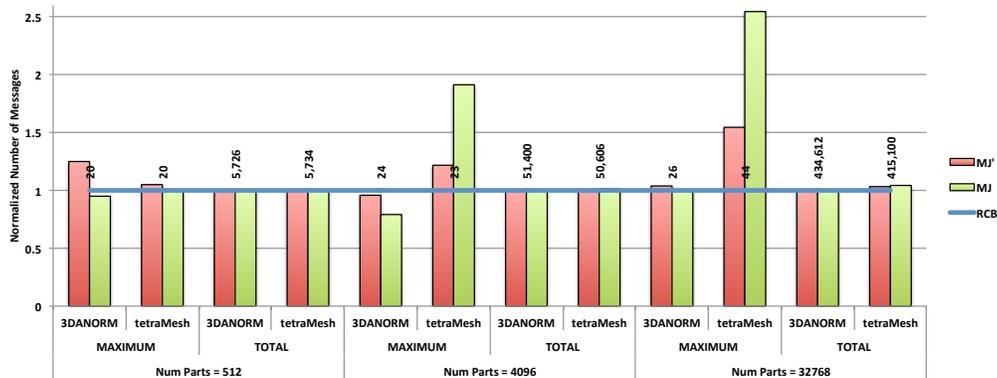


Fig. 5. Normalized maximum and total number of messages in MJ, MJ' and RCB partitions of 3D datasets. For each instance, the results with MJ and MJ' are normalized with respect to RCB's results, which are listed next to the corresponding figure bars. (Lower values are better.)

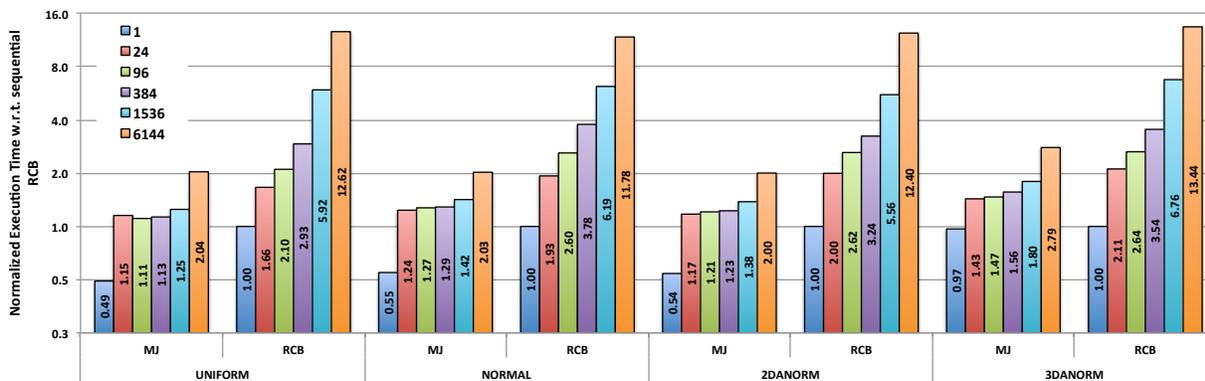


Fig. 6. Weak-scaling execution times of MJ and RCB (normalized with respect to sequential RCB's time) on synthetic datasets with 4M points per processor. For all processors, the points are randomly generated and scattered over the entire domain. (Lower values are better.)

parts; the 3D dataset was partitioned into 32,768. MJ partitioned the space into 256×256 and $32 \times 32 \times 32$ parts for the 2D and 3D datasets, respectively.

Figure 6 shows the weak-scaling execution times of MJ and RCB, normalized with respect to the sequential run-time of RCB on the corresponding dataset. In Figure 6, the default initial distribution was used, with each processor owning coordinates across the entire domain. We see in the

figure that the execution time of RCB increased with the number of processors due to the cost of migration. Since the points owned by a processor can be from any region of the domain, a significant portion of the data (roughly half) needed to be migrated between subsets of processors after each bisection. On the other hand, in most instances, MJ avoided migration as it was predicted to be expensive. As seen in the figure, MJ did not have perfect weak scaling,

but the increase in MJ’s parallel execution time was much smaller than RCB’s. Even though the migration was expensive in these experiments, MJ did perform migration starting from 6144 (1536) processors to avoid $256 \times it'$ ($1024 \times it'$) REDUCEALL operations among large numbers of processors for the 2D (3D) datasets. Although performing migration in these cases reduced the performance of MJ, the cost of the migration was lower than in RCB, since it was performed only once. Even in the 3D datasets, migration was performed only after partitioning along the first dimension, which reduced the migration cost relative to RCB.

Figure 7 shows the weak-scaling execution times (again, normalized to sequential RCB) using MJ and RCB with the same datasets, but now with initial distributions localized by RCB pre-partitioning. In this experiment, we also studied the effect of perturbing the input after the pre-partitioning step. Perturbing the data changes the execution time by changing the migration volume and/or the number of iterations needed to compute cut positions. In the figure, the *Noise* value refers to the fraction of points in each processor whose coordinates have been regenerated using a uniform distribution after pre-partitioning. Each processor chose the interval for its uniform distribution by doubling the minimum and maximum coordinate distance to the processor centroid. In Figure 7, *Noise* = 0 represents the normalized execution time of MJ and RCB with pre-partitioning and no perturbed points, while *Noise* = 0.05 represents the increase in execution time when perturbations were applied to 5% of the points.

Since the processors had localized data in this experiment, we expected the total migration volume to be lower than in the previous experiment. Also because of the localization, the workloads of the processors among the parts after the first partitioning in MJ were uneven. Therefore, MJ chose to perform migration in all instances in this experiment, including the ones with noise. As shown in Figure 7, MJ performed better than RCB in most instances, even though the pre-partitioning was done with RCB and MJ’s migration volume was expected to be larger than that of RCB. This result was mainly because MJ performed migration less frequently than RCB. For example, with the 2D datasets, MJ migrated data only once after partitioning the dataset into 256 parts, regardless of the number processors used. The migration cost of MJ increased with the number of processors, because the migration volume and the number of processors participating in the migration increased. When we added noise to the datasets, we observed that greater migration volume more significantly increased the execution time at large core counts for RCB than for MJ. For example, when 5% noise was added to the 3DANorm dataset on 6144 processors, the execution time of RCB increased to 92.08 seconds from 57.11 seconds, while the execution time of MJ increased to 17.50 seconds from 13.42 seconds.

A surprising result in Figure 7 is that MJ’s execution time decreased above 24 processors. On 24 processors, processor workloads were imbalanced after the 2D (3D) datasets were partitioned into 256 (32) along the first dimension. During migration, 16 (8) processors received 11 (2) of the resulting parts each, while 8 (16) processors received 10 (1) parts each. Therefore, after partitioning along the first dimension, the processors had imbalanced workloads, resulting in higher

execution times. For higher processor counts, processor workloads were balanced after the first partition; if workloads became imbalanced, they did so at later steps.

In the weak scaling experiments, we observe that the execution time of both MJ and RCB were smaller when the data was pre-partitioned for almost all instances compared to their counterparts in Figure 6. Localization via migration makes the algorithms faster for subsequent steps, and with pre-partitioning, the cost of each migration is lower. However, there were a few instances for which the algorithms ran more slowly on the data with 5% noise than without pre-partitioning (e.g., 6144-processor experiments with MJ on 2DANorm and RCB on Normal). Even though the migration costs were lower with pre-partitioning, execution time increased because, with noise, the global intervals in which cuts were considered changed, resulting in increased time to find cut positions.

4.3 Strong scaling

Strong scaling experiments were run up to 1536 processors for all synthetic and real datasets in Table 1. The synthetic dataset size was 50M points. As in the weak-scaling experiments, 2D datasets were partitioned into 65,536 (256×256 for MJ) parts and 3D datasets were partitioned into 32,768 ($32 \times 32 \times 32$ for MJ). Figure 8 shows the strong-scaling speedup with respect to sequential RCB for the generated datasets with the default initial distribution (i.e., each processor owning points from across the entire domain).

In all instances, MJ obtained higher speedups than RCB. As in our weak-scaling experiments, this result was due to differing migration costs in the two methods. In all datasets, RCB scaled well up to 384 processors; starting at 1536 processors, RCB was communication bounded. With the 2D datasets, MJ showed increasing speedups up to 96 processors; then the speedup dropped slightly for 384 processors, and increased again for 1536 processors. Increasing the number of processors reduced the work of each processor and made the partitioning more communication bounded, causing MJ to switch from no-migration mode to migration mode. For example, when partitioning Uniform with 384 processors, each processors initially owned $\sim 130K$ coordinates. After partitioning into 256 parts along the first dimension, each part in each processor had ~ 500 coordinates, less than the threshold (1000) from Section 3.2. The added migration lowered the speedup on 384 processors, but the speedup was recovered at 1536 processors. The switch between MJ’s no-migration and migration modes occurred at 96 processors for 3D datasets.

We observed higher speedups for 3DANorm than for the 2D datasets. Since each recursion level found fewer parts for 3DANorm compared to the 2D datasets, less time was spent in steps that were communication bounded. For example, with 1536 processors, each processor initially owned only $\sim 32.5K$ coordinates, so partitioning was communication bounded in the first recursion level. In this level, MJ performed more operations by finding 255 cuts in 2D datasets, while for 3D datasets, MJ sought only 31 cuts. Data migration reduces this problem in subsequent recursion levels.

Figure 9 shows the strong scaling speedups with the pre-partitioned initial distributions. As before, we add an

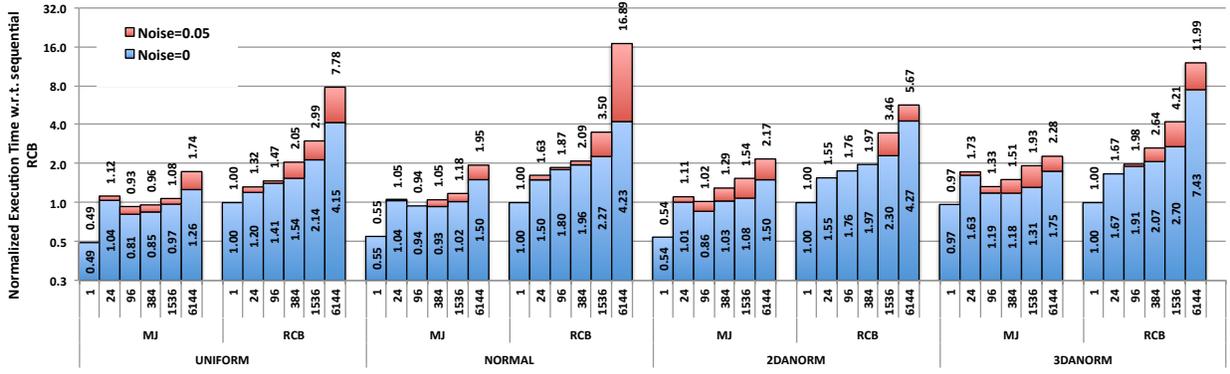


Fig. 7. Weak-scaling execution times of MJ and RCB (normalized by serial RCB's time with Noise=0) on synthetic datasets with 4M points per process. Points are randomly generated and prepartitioned using RCB so that each processor owns a unique region. (Lower values are better.)

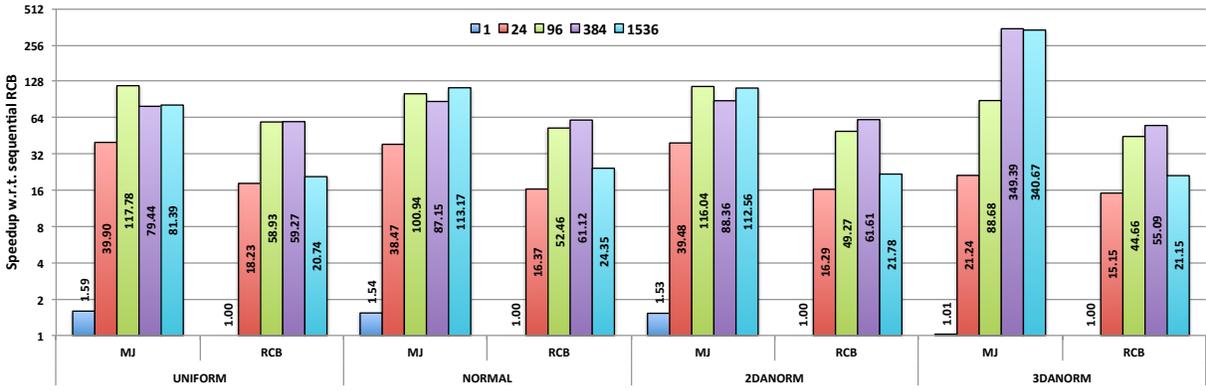


Fig. 8. Strong-scaling speedups of MJ and RCB relative to serial RCB on synthetic data sets with 50M points. Each processor's initial points are randomly generated and scattered over the entire space. (Higher values are better.)

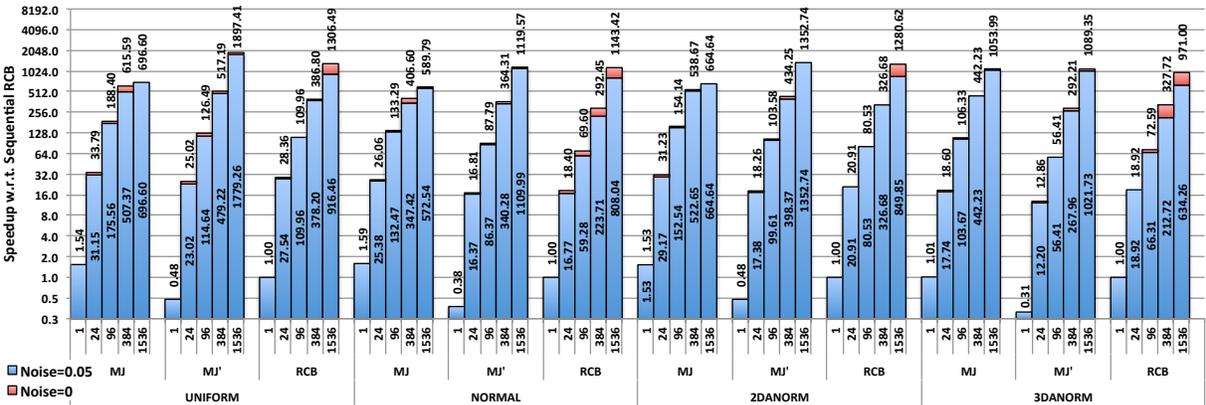


Fig. 9. Strong-scaling speedups of MJ and RCB relative to serial RCB on synthetic datasets with 50M points. Points are randomly generated and then prepartitioned using RCB so that each processor owns a unique area in the space. (Higher values are better.)

MJ variant MJ' to the comparisons. In MJ', the recursion depth is doubled; that is, MJ' partitions the 2D datasets into $16 \times 16 \times 16 \times 16$, and 3D datasets into $4 \times 4 \times 4 \times 8 \times 8 \times 8$ parts. Similar to the weak-scaling experiments in Figure 7, we also study how speedups are affected by the addition of 5% noise. In Figure 9, Noise=0.05 corresponds to the speedups obtained for the datasets perturbed with 5% noise. Noise=0 corresponds to the reduction in speedup when noise is added. Thus, the speedup without noise is the sum of the Noise = 0.05 and Noise = 0 values, which is printed on the top of each bar in the chart.

We observe that all algorithms scaled up to 1536 pro-

cessors, and speedups were always higher than their counterparts in Figure 8, thanks to the pre-partitioned data. The addition of the noise does not affect performance as significantly as in the weak-scaling experiments, since the number of coordinates was fixed at 50M, resulting in fewer perturbations and smaller effects on migration cost.

The speedups obtained by MJ are larger than RCB and MJ' up to 384 processors in all instances. However, when the number of processors increases to 1536, MJ is outperformed by both RCB and MJ'. This result is, again, because MJ seeks a larger number of cuts when the execution is communication bounded. As in the experiments of Figure 8,

MJ computes 255 cuts in the first communication-bounded recursion level, while RCB and MJ' compute only 1 and 15 cuts, respectively. However, MJ' outperformed RCB in most instances with 1536 processors. These results suggest that when the processor workloads are low, scalability is enhanced by increasing the recursion depth slightly, but multi-sectioning still has performance benefits over bisection.

Figure 10 shows the strong scaling results for the real datasets using MJ, MJ' and RCB, with the same number of parts and recursion depths as for the synthetic 2D and 3D experiments. The real datasets were read from their input files in the order they were stored. Each processor owned a chunk of consecutive coordinates from the file. Because of the order in which the coordinates were listed in the files, the initial distribution of coordinates to processors was somewhat localized. MJ and MJ' performed migration for all of the real datasets after the first step. However, when the recursion depth was larger than two, some processor groups performed migration in subsequent steps based on the imbalance calculations, while others avoided migration.

Huge-bubbles is the smallest real dataset in this experiment. None of the algorithms scaled after 96 processors. In all instances with *Huge-bubbles*, MJ and MJ' outperformed RCB. As expected, MJ' outperformed MJ on 96 and 384 processors. However, MJ was faster than MJ' on 1536, likely because MJ' needed to perform more migration because of its higher recursion depth. Similarly, for *europa*, MJ and MJ' outperformed RCB; MJ' is faster than MJ only on 384 processors. For *tetraMesh*, the largest real dataset, all of the algorithms scaled up to 1536 processors, and MJ obtained the best performance in all instances.

4.4 Dynamic Partitioning Simulation

In this experiment, we simulated the behavior of the algorithms for dynamic partitioning. We used the synthetic datasets with 4M coordinates per processor. As opposed to the weak-scaling experiments where the target number of parts was fixed, in these experiments, the target number of parts is the number of processors used (i.e., $\kappa = \rho$). Thus, although the number of points per processor is constant, the target number of parts (and, therefore, the number of cuts to be computed) increases. In the experiments, MJ was given only the target number of parts and the recursion depth; it determined the number of the cuts along the dimensions automatically, as described in Section 3.3. To simulate pre-distributed data for dynamic partitioning, we generated each dataset and then pre-partitioned the data using MJ or RCB, depending on which algorithm was to be tested. Since data was pre-partitioned using the algorithms themselves, migration volume was zero when no noise was added. However, with noise, the migration volume increased.

Figure 11 shows the execution time of the algorithms, normalized with respect to the execution time of RCB on 24 processors. Although the number of parts computed increased with the number of processors, MJ's execution time was nearly unchanged when there was no noise; this result was largely because the recursion depth was fixed to two or three for 2D and 3D datasets, respectively, regardless of the number of parts computed. In contrast, RCB's recursion depth (and, thus, execution time) grew with the number

of parts. For all instances, MJ was faster than RCB, and its performance was affected by noise much less than RCB's.

4.5 Multithreading

Almost all of today's parallel systems contain CPUs with multiple cores. To take advantage of these architectures, many applications use hybrid programming models such as MPI+OpenMP. Many studies show that using one MPI process per NUMA node (usually one per socket), and one thread per core of the NUMA node yields good performance. When such a hybrid application calls a partitioning library, the library has two options: either run an MPI-only implementation of the partitioning algorithm, matching the number of MPI processes in the application, or run a hybrid implementation of the algorithm, again with same number of MPI processes but with more flexibility in the number of threads. Clearly, in the former case, the library does not take advantage of all of the cores. But if only MPI-only partitioners are available, this option is the only one that can be used. To address this practical problem, we developed MJ using a hybrid MPI+OpenMP programming model.

All results presented up to this point used one thread per MPI process to fairly compare with Zoltan's RCB. Here, we run an experiment using the *huge-bubbles* dataset, assuming that the application calling the partitioning library is hybrid. We present results on 1-64 nodes, using four MPI processes per node (since each node in our test environment has four NUMA nodes with six cores each). The partitioner is run using 4-256 MPI processes, and the dataset is partitioned into the same number of parts as MPI processes. The results are given in Figure 12(a). In this figure, U-RCB and U-MJ correspond to RCB's and MJ's under-utilized MPI-only versions; H-MJ represents MJ's hybrid version. As seen in figure, H-MJ obtains speed-ups between 2 and 3 with respect to U-MJ, which clearly highlights the importance of having a partitioning technique that can match the environment in which the application is running.

On the other hand, in Figure 12(b), we compare H-MJ with fully utilized versions of RCB and MJ for the same problem. That is, RCB and MJ are run in the MPI-only mode but using all the underlying cores (24-1536 MPI processes); the data is still partitioned into 4-256 parts. Although H-MJ is faster than RCB in most cases, MJ's MPI-only version is often faster than the hybrid version. This result occurs because the MPI-only implementation has more localized data storage; therefore, it is more cache-friendly than the hybrid version. Hence, we envision H-MJ is needed only for applications that use fewer MPI ranks than available cores. For traditional applications, we recommend MPI-based MJ.

5 CONCLUSIONS

We have proposed a parallel multi-jagged coordinate partitioning algorithm, MJ, that differs from RCB in its ability to partition into multiple parts at once and its handling of data during partitioning. We have presented experiments on various datasets, and compared the performance of MJ against Zoltan's RCB. Our weak scaling experiments demonstrated that our MJ algorithm performs better than Zoltan's RCB, and scales up to 6144 processors. MJ scaled well up to 384

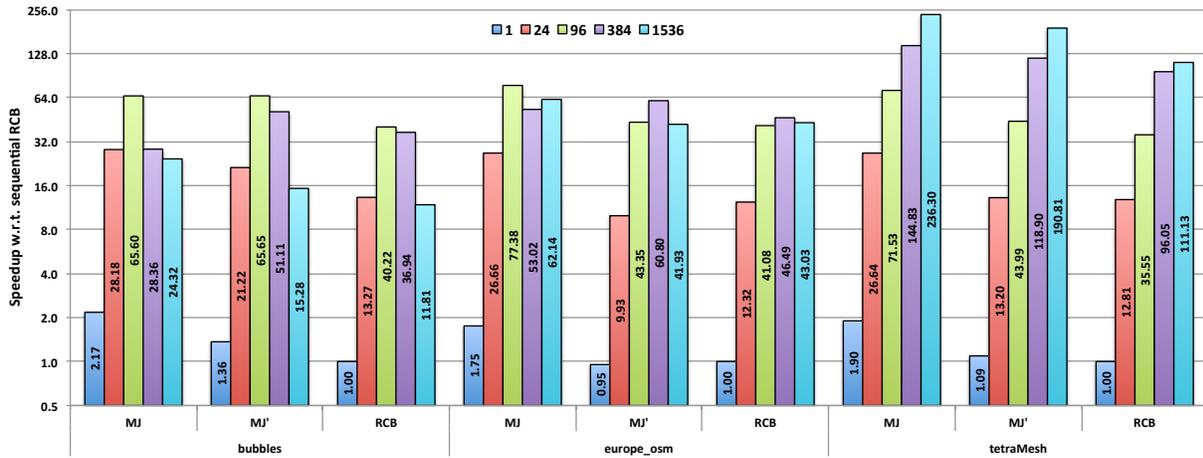


Fig. 10. Strong-scaling speedups of MJ and RCB relative to serial RCB on real datasets. (Higher values are better.)

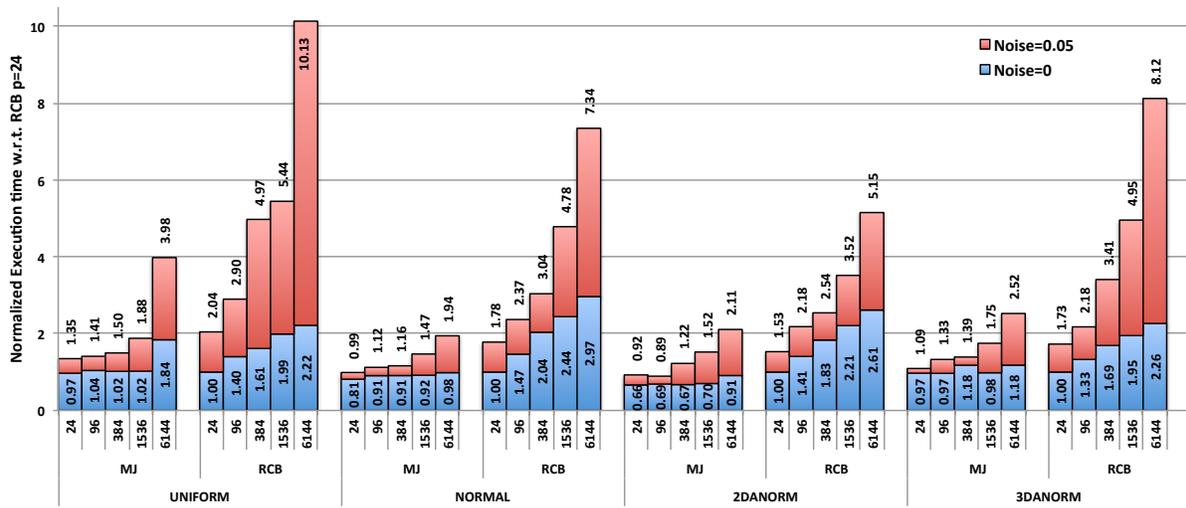


Fig. 11. Execution times for MJ and RCB (normalized with respect to RCB's time with Noise=0 on 24 processors) using synthetic data sets in a dynamic partitioning scenario. (Lower values are better.)

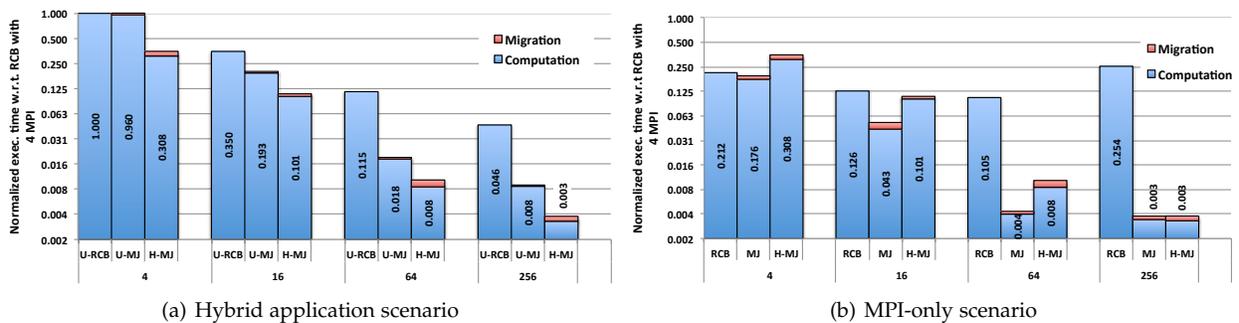


Fig. 12. Execution times of partitioning algorithms normalized with respect to U-RCB with 4 MPI processes. (Lower values are better.) In (a), U-RCB and U-MJ are the under-utilized variants of MJ and RCB, while H-MJ is the hybrid (MPI+OpenMP) version of MJ. In (b), H-MJ is compared to the fully utilized MPI-only versions of MJ and RCB.

processors in most of our strong scaling experiments. We have also evaluated the quality of the RCB and MJ partitions in terms of communication.

A significant difference between MJ and Zoltan's RCB is that RCB migrates the coordinates and weights of the objects after each bisection. Migration allows RCB (and MJ) to work with fewer processors at each recursion and

allows execution of subsequent bisections in parallel. Even though the amount of data moved decreases with increasing levels of recursion, all processors perform migration at each recursion level. The worst case occurs when the data is randomly distributed to all processors. In such a case, half of the data will be moved at each recursion. In the ideal case, the initial data is already partitioned, and RCB will not

migrate any data.

In contrast, at each recursion, MJ decides whether or not to migrate based on three factors: the expected load imbalance in subsequent partitioning steps, the estimated cost of global reduction operations if no migration occurs, and a threshold on the minimum amount of work allowed per processor. We have shown that MJ makes reasonably good migration decisions; however, the interplay of the data size and system size to decide when migration will be most beneficial can be investigated further. The effect of migration is visible from our experiments. In weak scaling tests, RCB's performance can be attributed to communication costs due to migration. In the strong scaling results for real datasets, the data was already well-localized in processors, and hence, provided a more ideal case for RCB.

ACKNOWLEDGMENT

We thank Erik Boman for helpful discussions. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Dept. of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work was supported in part by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program (FASTMath and CSCAPES Institutes) and by National Science Foundation grant OCI-0904809. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Dept. of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

- [1] B. Aspvall, M. M. Halldórsson, and F. Manne, "Approximations for the general block distribution of a matrix," *Theor. Comput. Sci.*, vol. 262, no. 1-2, pp. 145–160, 2001.
- [2] R. F. Barrett, C. T. Vaughan, and M. A. Heroux, "MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing," Sandia National Labs., Albuquerque, NM, Tech. Rep. SAND2012-10431, 2012.
- [3] M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Trans. Comput.*, vol. C36, no. 5, pp. 570–580, 1987.
- [4] E. Boman, K. Devine, V. Leung, S. Rajamanickam, L. Riesen, M. Deveci, and Ü. Çatalyürek, "Zoltan2: Next-generation combinatorial toolkit." Sandia National Labs., Tech. Rep., 2012.
- [5] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, 1999.
- [6] T. A. Davis and Y. Hu, "The University of Florida collection," *ACM Trans. Math. Software*, vol. 38, no. 1, pp. 1:1–1:25, 2011.
- [7] M. Deveci, S. Rajamanickam, V. Leung, K. Pedretti, S. Olivier, D. Bunde, Ü. V. Çatalyürek, and K. Devine, "Exploiting geometric partitioning in task mapping for parallel computers," in *IEEE Int. Parallel Distrib. Proc. Symp.*, 2014.
- [8] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek, "Parallel hypergraph partitioning for scientific computing," in *IEEE Int. Parallel Distrib. Proc. Symp.*, 2006.
- [9] J. Gaidamour, J. Hu, C. Siefert, and R. Tuminaro, "Design considerations for a flexible multigrid preconditioning library," *Sci. Program.*, vol. 20, no. 3, pp. 223–239, Jul. 2012.
- [10] E. G. Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine, "The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring," *Sci. Program.*, vol. 20, no. 2, pp. 129–150, 2012.
- [11] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala, "ML 5.0 smoothed aggregation user's guide," Sandia National Labs., Tech. Rep. SAND2006-2649, 2006.
- [12] M. Grigni and F. Manne, "On the complexity of the generalized block distribution," in *IRREGULAR*, 1996.
- [13] Y. Han, B. Narahari, and H.-A. Choi, "Mapping a chain task to chained processors," *Inform. Process Lett.*, vol. 44, pp. 141–148, 1992.
- [14] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Supercomputing*, 1995.
- [15] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Comput.*, vol. 26, pp. 1519–1534, 2000.
- [16] I. Kalashnikova, M. Perego, A. G. Salinger, R. S. Tuminaro, and S. F. Price, "Albany/FELIX: a parallel, scalable and robust, finite element, first-order Stokes approximation ice sheet solver built for advanced analysis," *Geosci. Model Develop. Discuss.*, vol. 7, no. 6, pp. 8079–8149, 2014.
- [17] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.
- [18] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. Jnl.*, vol. 49, no. 2, pp. 291–307, 1970.
- [19] P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, E. Cyr, and S. Kenyon, "Towards extreme-scale simulations with next-generation Trilinos: a low mach fluid application case study," in *Wksh. Large-Scale Parallel Processing, IEEE Int. Parallel Distrib. Proc. Symp.*, 2014.
- [20] P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, and S. Kenyon, "Towards extreme-scale simulations with second-generation Trilinos," *Parallel Processing Letters*, vol. 24, no. 04, p. 1442005, 2014.
- [21] R. M. Loy, "Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws," Ph.D. dissertation, Rensselaer Polytechnic Inst., 1998.
- [22] F. Manne and T. Sorevik, "Partitioning an array onto a mesh of processors," in *PARA*, 1996.
- [23] T. Minyard and Y. Kallinderis, "Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations," *Int. J. Numer. Meth. Fluids*, vol. 26, pp. 57–78, 1998.
- [24] D. Nicol, "Rectilinear partitioning of irregular data parallel computations," *J. Parallel Distrib. Comput.*, vol. 23, pp. 119–134, 1994.
- [25] A. Patra and J. T. Oden, "Problem decomposition for adaptive hp finite element methods," *J. Comput. Syst. Engg.*, vol. 6, no. 2, 1995.
- [26] J. R. Pilkington and S. B. Baden, "Partitioning with spacefilling curves," U. of California, CSE Technical Report CS94-349, 1994.
- [27] A. Pinar and C. Aykanat, "Sparse matrix decomposition with optimal load balancing," in *HiPC*, 1997.
- [28] —, "Fast optimal load balancing algorithms for 1D partitioning," *J. Parallel Distrib. Comput.*, vol. 64, pp. 974–996, 2004.
- [29] E. Saule, E. Ö. Baş, and Ü. V. Çatalyürek, "Load-balancing spatially located computations using rectangular partitions," *J. Parallel Distrib. Comput.*, vol. 72, no. 10, pp. 1201 – 1214, 2012.
- [30] D. G. Schweikert and B. W. Kernighan, "A proper model for the partitioning of electrical circuits," in *Design Automation Conf.*, 1972.
- [31] SIERRA Solid Mechanics Team, "Presto 4.16 user's guide," Sandia National Labs., Tech. Rep. SAND2010-3112, 2010.
- [32] —, "SIERRA/Solid mechanics 4.22 user's guide," Sandia National Labs., Tech. Rep. SAND2011-7597, 2011.
- [33] H. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering*, vol. 2, no. 2/3, pp. 135 – 148, 1991.
- [34] V. E. Taylor and B. Nour-Omid, "A study of the factorization fill-in for a parallel implementation of the finite element method," *Int. J. Numer. Meth. Engng.*, vol. 37, pp. 3809–3823, 1994.
- [35] M. Ujaldon, S. Sharma, E. Zapata, and J. Saltz, "Experimental evaluation of efficient sparse matrix distributions," in *Supercomputing*, 1996.
- [36] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Supercomputing*, 1993.

Mehmet Deveci is a PhD Student in the Department of Computer Science & Engineering at Ohio State University. He received his B.S. in Computer Engineering from Middle East Technical University, Turkey in 2010.

Dr. Sivasankaran Rajamanickam (M'14) is a Senior Member of Technical Staff in the Computer Science Research Institute in the Center for Computing Research at Sandia National Laboratories. He earned his B.E. from Madurai Kamaraj University, India, and his Ph.D. in Computer Engineering from University of Florida.

Dr. Karen Devine is a Principal Member of the Technical Staff in the Computer Science Research Institute in the Center for Computing Research at Sandia National Laboratories. She earned her B.S. from Wilkes College, and her M.S. and Ph.D. in Computer Science from Rensselaer Polytechnic Institute.

Dr. Ümit V. Çatalyürek (M'09-SM'10) is a Professor in the Depts. of Biomedical Informatics, Electrical & Computer Engineering, and Computer Science & Engineering at the Ohio State University. He received his Ph.D., M.S. and B.S. in Computer Engineering and Information Science from Bilkent University, Turkey, in 2000, 1994 and 1992, respectively.