Effective Out-of-Core Parallel Delaunay Mesh Refinement using Off-the-Shelf Software

ANDRIY KOT, College of William and Mary
ANDREY N. CHERNIKOV and NIKOS P. CHRISOCHOIDES, Old Dominion University

We present three related out-of-core parallel mesh generation algorithms and their implementations for small size computational clusters. Computing out-of-core permits to solve larger problems than otherwise possible on the same hardware setup. Also, when using shared computing resources with high demand, a problem can take longer to compute in terms of wall-clock time when using an in-core algorithm on many nodes instead of using an out-of-core algorithm on few nodes. The difference is due to wait-in-queue delays that can grow exponentially to the number of requested nodes. In one specific case, using our best method and only 16 nodes it can take several times less wall-clock time to generate a 2 billion element mesh than to generate the same size mesh in-core with 121 nodes.

Although our best out-of-core method exhibits unavoidable overheads (could be as low as 19% in some cases) over the corresponding in-core method (for mesh sizes that fit completely in-core) this is a modest and expected performance penalty. We evaluated our methods on traditional clusters of workstations as well as presented preliminary performance evaluation on emerging BlueWaters supercomputer.

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and software—*Performance evaluation (efficiency and effectiveness)*; I.1.2 [**Computing Methodologies**]: Algorithms—*Nonalgebraic algorithms*; I.3.5 [**Computing Methodologies**]: Computational Geometry and Object Modeling—*Geometric algorithms, languages, and systems*

General Terms: Algorithms; Performance

Additional Key Words and Phrases: Effective computing, off-the-shelf, Delaunay, wall-clock time

ACM Reference Format:
Kot, A., Chernikov, A. N., Chrisochoides N. P. 2011. Effective Out-of-Core Parallel Delaunay Mesh Refinement using Off-the-Shelf Software. ACM J. Exp. Algor. V, N, Article A (YYYY), 22 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

A parallel Finite Element mesh generation software decomposes the original mesh generation problem into smaller sub-problems that can be solved (meshed) in parallel. The limiting factor for parallel mesh generation is memory. In turn, while increasing the number of Processing Elements (PEs) makes mesh generation time shorter the difference is not critical when compared to other delays such as wait-in-queue time usually associated with using large-scale shared computing resources. According to statistics collected on the SciClone cluster at the College of William and Mary from the last four and a half years (see Fig. 1, right) the average waiting time for 300 PEs is several hours while it takes only several minutes to generate the largest possible mesh (for this configuration) using our presented parallel mesh generation software.

This work is supported in part by NSF grants: CCF-0916526, CCF-0833081, and CSI-719929 and by the John Simon Guggenheim Foundation and Richard T. Cheng Endowment.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

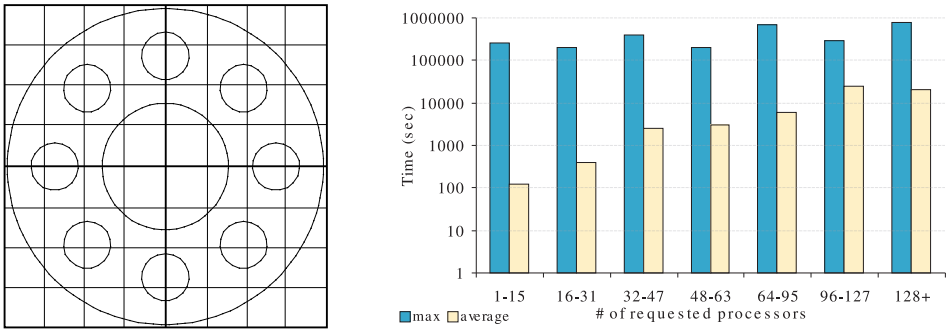

Fig. 1. (Left) The pipe cross-section overlapped by a uniform lattice used by the PDR method. The squares in bold represent refinement blocks that are subdivided into smaller cells; and (right) the wait-in-queue time statistics for parallel jobs collected from the last four and a half years from a 300+ processor cluster at the College of William and Mary.

Our goal is to make possible generating very large meshes on limited memory machines like the current and emerging multi-processor multi-core high-end workstations. The solution is to store on disk most of the mesh i.e., use out-of-core (OoC) methods. We designed a family of OoC parallel mesh generation algorithms using existing parallel in-core mesh generation methods [Chernikov and Chrisochoides 2004; 2006a].

Existing meshing methods first decompose the original problem into smaller sub-problems. The sub-problems can be formulated to be either tightly or partially coupled or even decoupled. The coupling of the sub-problems (i.e., the degree of dependency) determines the intensity of the communication and the synchronization between the sub-problems [Chrisochoides 2005].

Tightly coupled mesh generation methods such as Parallel Optimistic Delaunay Mesh (PODM) generation method [Nave et al. 2002] are not suitable for out-of-core (OoC) computations due to intensive communication among subdomains which would require very frequent (up to 25K per second) disk accesses. Since this requirement cannot be satisfied such methods would be prohibitively slow.

Partially coupled methods are suitable for OoC parallel Delaunay mesh generation because they: (1) require less communication than tightly coupled methods and may employ structured communication patterns, (2) do not rely on domain decomposition and (3) can take advantage of existing off-the-shelf sequential mesh generation libraries. The last point is important due to high complexity and long development time of high-performance sequential industrial strength mesh generation libraries.

Weakly coupled OoC Parallel Constraint Delaunay Meshing [Chernikov and Chrisochoides 2008a] proved suitable for effective OoC computing for 2D geometries. Unfortunately, the method relies on solving the Medial Axis problem which has not yet been solved for 3D. Additionally, it uses a custom mesh generation kernel; extra work would be required to either replace it or extend it to 3D.

The decoupled methods (e.g., Parallel Delaunay Domain Decoupling Method [Linardakis and Chrisochoides 2008b]) need little to no communication and use off-the-shelf state-of-the-art sequential software like Triangle [Shewchuk 1996]. Unfortunately, these methods rely upon the solution of a very difficult Domain Decomposition problem which is open for 3-dimensional (3D) geometries [Linardakis and Chrisochoides 2008a].

In [Chernikov and Chrisochoides 2004; 2006a] we presented an in-core partially coupled Parallel Delaunay Refinement (PDR) method which relies upon a simple block data decomposition (see Figure 1, left). The data decomposition is generated with a

uniform 2D/3D lattice[1] covering the entire domain of $\mathcal{M}$. The block decomposition is used to guide the parallel refinement, so that Steiner points, independently inserted in certain regions of $\mathcal{M}$, are a priori Delaunay-independent. This method is extended to 3D [Chernikov and Chrisochoides 2008b]. In this paper we use the 2D PDR method to develop and analyze the performance of the parallel OoC guaranteed quality Delaunay mesh generation algorithms and their implementation. The 2D method is not as computationally intensive as the 3D PDR method and thus we expect that the performance data we present here will improve even further for the 3D case since longer time spent in computations leads to better overlap with network and disk I/O.

The main contribution of this paper are three OoC algorithms and their implementations on a cluster of workstations (CoW) with both single processor nodes and k-way multi-processor/multi-core nodes. Specifically, in Section 4.1 we present an OoC method for shared memory machines with multiple processing cores, in Section 4.2 we present an OoC method for distributed memory machines with a single processor per node and in Section 4.3 we present a hybrid OoC method for CoWs with multi-core nodes. Our performance data in Section 5 indicate that the total wall-clock time including wait-in-queue delays and total execution time for the hybrid OoC method on 16 processors is 3.3 times shorter than the total wall-clock time for the in-core generation of the same size meshes using more than one hundred processors. Although the OoC methods exhibit 19% to 56% overhead over the corresponding in-core method (for mesh sizes that fit completely in the core of the CoWs) this is a modest performance penalty for savings of many hours in response time. All OoC codes use the fastest to our knowledge off-the-shelf sequential Delaunay mesh generator [Shewchuk 1996]. This helps us leverage the on-going improvements in terms of quality, speed, and functionality of sequential in-core Delaunay mesh generation methods.

2. RELATED WORK

The modeling of physical phenomena in Computational Fluid Dynamics, Solid Mechanics, Biomedical and Material image analysis, and other related areas is based on solving systems of partial differential equations (PDEs). When PDEs are defined over geometrically complex domains, they often do not admit closed form solutions. In these cases, the PDEs are solved approximately using finite element and finite volume methods by considering discretizations of domains into simple elements like triangles in two dimensions, and tetrahedra in three dimensions. These discretizations are called finite element meshes.

Delaunay refinement is a popular technique for generating triangular and tetrahedral meshes for approximation and interpolation in various numeric computing areas. Among the reasons of its popularity is the amenability of the method to rigorous mathematical analysis, which allows to derive guarantees on the quality of the elements in terms of circumradius-to-shortest edge ratio, the gradation of the mesh, and the termination of the algorithm. The problem of parallel Delaunay *triangulation* of a specified point set has been solved by Blelloch et al. [Blelloch et al. 1999]. Delaunay *refinement* algorithms work by inserting additional (so-called Steiner) points into an existing mesh to improve the quality of the elements. In Delaunay mesh refinement, the computation depends on the input geometry and changes as the algorithm progresses. The basic operation is the insertion of a single point which leads to the removal of a poor quality tetrahedron and of several adjacent tetrahedra from the mesh and the insertion of several new tetrahedra. The new tetrahedra may or may

[1]We call the PDR method that employs a uniform lattice for data decomposition the *uniform* PDR. We also developed the PDR method that employs a quadtree (octree in 3D) for data decomposition which we call *non-uniform* PDR. In this paper we always refer to the *uniform* PDR method unless stated otherwise.

not be of poor quality and, hence, may or may not require further point insertions. It is proven that the algorithm eventually terminates after having eliminated all poor quality tetrahedra, and in addition the termination does not depend on the order of processing of poor quality tetrahedra, even though the structure of the final meshes may vary. Traditional Delaunay refinement algorithms insert new points in the centers of the circumscribed circles [Shewchuk 2002; Ruppert 1995], while in our recent work [Chernikov and Chrisochoides 2009; Foteinos et al. 2010] we have shown that any points inside larger regions (called selection disks) can be chosen.

There are two basic approaches for the out-of-core computing: implicit, usually involves virtual memory (VM) supported by internal mechanisms of an operating system (OS); and explicit, which often implies algorithm-specific optimizations.

While VM is easy to employ it has limitations. The OS-supported VM is optimized for system throughput and usually cannot exploit access patterns of irregular and adaptive applications. On four processors, our tests indicate that an increase in the problem size from 23.8 million elements to 58.8 million elements (doubling the amount of memory by using disk) resulted in an increase of the execution time from about 7 minutes to over 3 hours (192 minutes). Our out-of-core methods generate meshes of the same size (58.8 millions) in less than 30 minutes on the same four processor workstation. Additionally, the amount of VM may be limited by either computer architecture (32-bit processors can only address 4GB) or by administration of the computing resources (it is common to set VM no more than twice the amount of RAM [2]).

The explicit approach is usually employed to develop algorithm-specific out-of-core methods. This approach has been very effective in linear algebra parallel computations [Toledo and Gustavson 1996; D'Azevedo and Dongarra 2000]. Out-of-core linear algebra libraries use various mapping layouts (depending on the underlying I/O and algorithm specifics) to store out-of-core matrices and employ vendor supplied libraries for asynchronous disk I/O. They rely on high performance in-core subroutines of BLAS [Dongarra et al. 1988], LAPACK [Demmel et al. 1987] and ScaLAPACK [Choi et al. 1992] and a simple non-recursive (in most cases) pipeline to hide latencies associated with disk accesses.

Large amount of work was performed on designing optimal algorithms for parallel multi-level memory model [Aggarwal and Vitter 1988; Vitter and Nodine 1993; Nodine and Vitter 1995; Vitter et al. 1994; Vitter and Shriver 1993] as well as designing methods to map existing in-core algorithms based on batch-synchronous parallel models into efficient out-of-core algorithms [Dehne et al. 1997].

In [Salmon and Warren 1997] authors described an out-of-core N-body parallel method which is irregular and there is no creation or deletion of new bodies during the execution, unlike the parallel mesh refinement computation we focus on in this paper. Salmon et al. extend the virtual memory scheme to store out-of-core pages on the disk. They use an algorithm-specific space-filling curve to arrange data within memory pages. A problem-independent feature [Salmon and Warren 1997] is the page replacement algorithm which is based on the last recently used (LRU) replacement policy. The same policy is used as a basic virtual memory policy for many platforms (e.g., Linux). However, the authors extend it by introducing priorities, different aging speeds for different data types, and explicit page locking.

Etree [Tu and O'Hallaron 2004] is an out-of-core algorithm-specific approach for sequential mesh generation. The novelty of Etree is in the use of a spatial database to store and operate on large octree meshes. Each octant is assigned a unique key using the linear quadtree technique which is stored as a B-tree. There are three steps to generate a mesh with Etree: (1) create an unbalanced octree on disk, (2) balance the

[2]Based on authors' personal experience having access to computational clusters of varying sizes

etree by decomposing further the octants that violate the 2-to-1 constraint (each octant may not have more than two neighbors on each side) and (3) store the element-node relations and node coordinates in two separate databases. Subsequently, all the mesh operations are performed by querying the databases using Etree calls. This method targets octree meshes and it is exceptionally fast, especially after recent new improvements using a two-level bucket sort algorithm [Tu and O'Hallaron 2005]. However, it targets octree-based meshes and is not parallel yet.

3. PARALLEL DELAUNAY REFINEMENT METHOD

The Parallel Delaunay Refinement (PDR) algorithm is based on a theoretical framework for constructing guaranteed quality Delaunay meshes in parallel [Chernikov and Chrisochoides 2004; 2006a]. Sequential guaranteed quality Delaunay Refinement algorithms insert points at the selection disks around circumcenters of triangles [Chernikov and Chrisochoides 2006b] of poor quality or of unacceptable size. Two points are called Delaunay-independent iff they can be inserted concurrently without destroying the conformity and Delaunay properties of the mesh. For 2-dimensional geometries, we presented [Chernikov and Chrisochoides 2004] a sufficient condition of Delaunay-independence which is based on the distance between points: two points are Delaunay-independent if the distance between them is greater than $4\bar{r}$, where $\bar{r}$ is an upper bound on triangle circumradius in the initial mesh. In n-dimensions, to ensure that processors insert only Delaunay-independent points at each step of the algorithm we impose an n-dimensional hypercube lattice[3] over the entire n-dimensional domain.

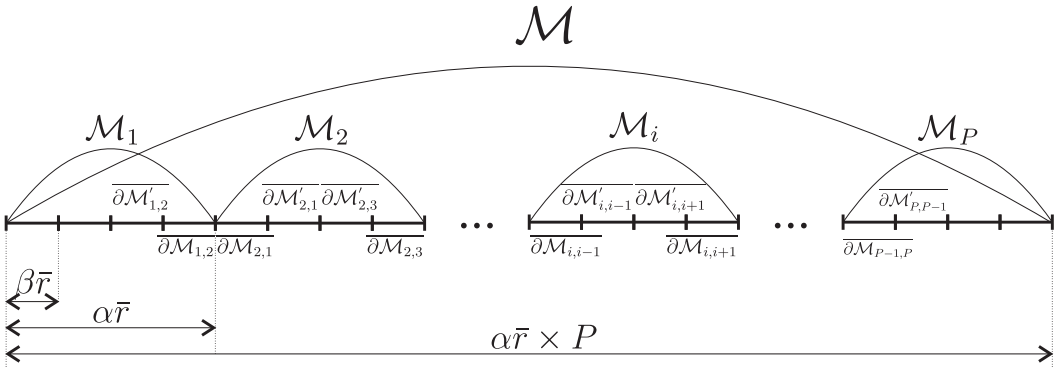

Fig. 2. Subdivision of a mesh $\mathcal{M}$.

For simplicity we begin by presenting the algorithm in one dimension[4]. In one dimension the hypercube lattice is equivalent to a segment subdivided into a number of smaller equal size subsegments (cells). We call the length of the segment (i.e., 1-D lattice) the size of the lattice. Similarly, we call the length of a subsegment (i.e., cell) the size of the cell. Consequently, the length of a segment that consists of several cells is the size of the segment and is equivalent to the sum of the cell sizes.

Given a conforming Delaunay mesh $\mathcal{M}$ and the number of available processors P we compute $\bar{r}$ such that the *size* of the corresponding lattice can be computed as $\alpha\bar{r} \times P$ where α is a constant that depends on implementation and dimensionality of the problem ($\alpha = 16$ for our 2D implementation). Next, $\mathcal{M}$ is distributed among P processors:

[3]The points pattern of the lattice is equivalent to that of an n-dimensional hypercube
[4]In one dimension a "triangulation" of a segment is a discretization of the segment.

let $\mathcal{M}_i$ be the mesh that resides in memory of processor i such that $\mathcal{M} = \bigcup_{i=1}^{P} \mathcal{M}_i$, and the *size* of a lattice segment that corresponds to $\mathcal{M}_i$ is equal to $\alpha\bar{r}$.

We denote bordering segments of $\mathcal{M}_i$ as $\overline{\partial\mathcal{M}_{i,j}}$ where i is the index of the subdomain containing $\mathcal{M}_i$ and j is the index of the respective neighbor, $j \in \{i-1, i+1\}$ (e.g., $\overline{\partial\mathcal{M}_{3,4}}$ would be the rightmost segment of $\mathcal{M}_3$). *Size* of each bordering segment is $\beta\bar{r}$, where β is a constant that depends on implementation and dimensionality of the problem ($\beta = 4$ for our 2D implementation) and $\beta \mid \alpha$. Additionally, we denote segments of equal size of the border $\overline{\partial\mathcal{M}_{i,j}}$ inside $\mathcal{M}_i$ as $\overline{\partial\mathcal{M}'_{i,j}}$. Figure 2 shows the subdivision[5] of $\mathcal{M}$.

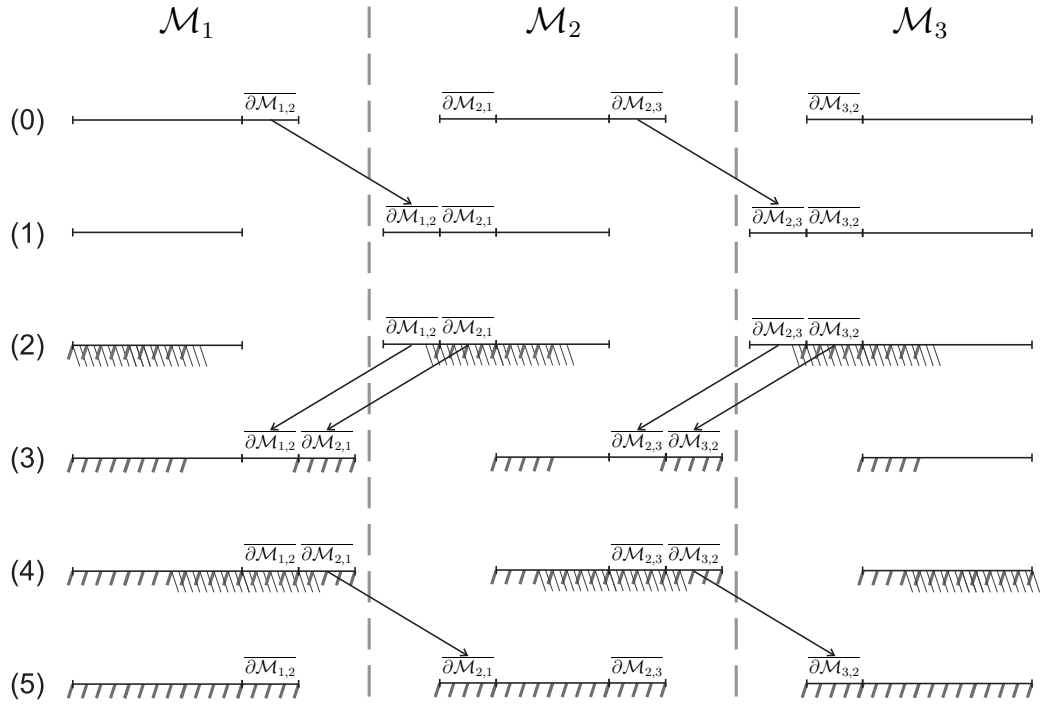

Fig. 3. An example of the PDR algorithm in one dimension. The mesh is comprised of three submeshes $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ (there are three processors), $\overline{\partial\mathcal{M}_{i,j}}$ denote border segments. Stages (0)–(5) correspond to algorithm steps 0–5. Arrows between different steps indicate movements of submeshes between domains (e.g., network send-receive). Right dashed (thin lines) areas show parts that are being modified during refinement, left dashed (thick lines) areas show refined parts.

Below is the outline of the algorithm. First, we define the necessary operations (for simplicity, A and B are abstract variables):

$A \leftarrow B$:	A is assigned a copy of a value in B, this includes transferring the copy to a processor where A is located, if necessary
$A \cup B$:	the result of this operation is a mesh that contains all elements of A and B as a single simply connected mesh, A and B are not modified
$A \setminus B$:	the result of this operation is a mesh that contains all elements in A except those in B, A and B are not modified

[5]According to the figure $\mathcal{M}_i = (\overline{\partial\mathcal{M}_{i,i-1}} \cup \overline{\partial\mathcal{M}'_{i,i-1}} \cup \overline{\partial\mathcal{M}'_{i,i+1}} \cup \overline{\partial\mathcal{M}_{i,i+1}})$ which is true for our 2D implementation but is not required, in fact $\mathcal{M}_i \supseteq (\overline{\partial\mathcal{M}_{i,i-1}} \cup \overline{\partial\mathcal{M}'_{i,i-1}} \cup \overline{\partial\mathcal{M}'_{i,i+1}} \cup \overline{\partial\mathcal{M}_{i,i+1}})$.

refine(A, B): defined only if $A \cup B$ where mesh A is refined as follows: elements in A that belong to $A \cap B$ are refined, additionally refinement may affect elements in A that belong to $A \triangle B$ and are geometrically within γ (γ is an implementation dependent constant, $\gamma \mid \beta$; $\gamma = 2\bar{r}$ for our 2D implementation) from the bounding box of B, resulting in refined mesh stored in A.

The algorithm will perform the following steps:

(0) distribute $\mathcal{M}$: $\mathcal{M} = \bigcup_{i=1}^{P} \mathcal{M}_i$, $\mathcal{M}_i \cap \mathcal{M}_j = \emptyset$, $i, j = 1, 2, \ldots, P$, $i \neq j$
let $I = \{2, 3, \ldots, P-1\}$

(1) $\forall i, i \in I$: $\mathcal{M}_i \leftarrow (\mathcal{M}_i \setminus \overline{\partial\mathcal{M}_{i,i+1}}) \cup \overline{\partial\mathcal{M}_{i-1,i}}$
$\mathcal{M}_1 \leftarrow \mathcal{M}_1 \setminus \overline{\partial\mathcal{M}_{1,2}}$
$\mathcal{M}_P \leftarrow \mathcal{M}_P \cup \overline{\partial\mathcal{M}_{P-1,P}}$

(2) $\forall i, i \in I$: **refine** $\left(\mathcal{M}_i, (\mathcal{M}_i \setminus (\overline{\partial\mathcal{M}'_{i,i+1}} \cup \overline{\partial\mathcal{M}_{i-1,i}}))\right)$
refine $\left(\mathcal{M}_1, (\mathcal{M}_1 \setminus (\overline{\partial\mathcal{M}'_{1,2}} \cup \overline{\partial\mathcal{M}_{1,2}}))\right)$
refine $\left(\mathcal{M}_P, (\overline{\partial\mathcal{M}_{P-1,P}} \cup \overline{\partial\mathcal{M}'_{P,P-1}})\right)$

(3) $\forall i, i \in I$: $\mathcal{M}_i \leftarrow \mathcal{M}_i \cup (\overline{\partial\mathcal{M}_{i,i+1}} \cup \overline{\partial\mathcal{M}_{i+1,i}}) \setminus (\overline{\partial\mathcal{M}_{i-1,i}} \cup \overline{\partial\mathcal{M}_{i,i-1}})$
$\mathcal{M}_1 \leftarrow \mathcal{M}_1 \cup \overline{\partial\mathcal{M}_{1,2}} \cup \overline{\partial\mathcal{M}_{2,1}}$
$\mathcal{M}_P \leftarrow (\mathcal{M}_1 \setminus \overline{\partial\mathcal{M}_{P-1,P}}) \cup \overline{\partial\mathcal{M}_{P,P-1}}$

(4) $\forall i, i \in I$: **refine** $\left(\mathcal{M}_i, (\mathcal{M}_i \setminus (\overline{\partial\mathcal{M}'_{i,i-1}} \cup \overline{\partial\mathcal{M}_{i+1,i}}))\right)$
refine $\left(\mathcal{M}_1, (\overline{\partial\mathcal{M}'_{1,2}} \cup \overline{\partial\mathcal{M}_{1,2}})\right)$
refine $\left(\mathcal{M}_n, (\mathcal{M}_n \setminus \overline{\partial\mathcal{M}'_{P,P-1}})\right)$

(5) $\forall i, i \in I$: $\mathcal{M}_i \leftarrow (\mathcal{M}_i \cup \overline{\partial\mathcal{M}_{i,i-1}}) \setminus \overline{\partial\mathcal{M}_{i+1,i}}$
$\mathcal{M}_1 \leftarrow \mathcal{M}_1 \setminus \overline{\partial\mathcal{M}_{2,1}}$
$\mathcal{M}_P \leftarrow \mathcal{M}_P \cup \overline{\partial\mathcal{M}_{P,P-1}}$

See Figure 3 for an example of algorithm execution with mesh partitioned between three subdomains. In step 0, mesh is subdivided into submeshes and distributed between processors. In step 1, border segments on the right side of each submesh are transferred to neighbors on the right of their respective processors. In step 2, each processor refines its submesh, border segments $\overline{\partial\mathcal{M}'_{i,i+1}}$ and $\overline{\partial\mathcal{M}_{i-1,i}}$ are not refined but changes may propagate into them. In step 3, border segments on the left side of each submesh together with the border segments that were transferred in step 1 are transferred to neighbors on the left of their respective processors. In step 4, each processor refines its submesh, border segments $\overline{\partial\mathcal{M}'_{i,i-1}}$ and $\overline{\partial\mathcal{M}_{i+1,i}}$ are not refined but changes may propagate into them. In step 5, border segments now located on the right of each submesh are transferred to their original locations, on the left side of their respective submeshes. At this point the mesh is refined and the algorithm finishes.

3.1. Shared memory implementation of the PDR

The original implementation of the PDR was for distributed memory computing. However, since multi-core (including support for hardware threads) is becoming increasingly popular we implemented a modified algorithm to take advantage of shared resources and to avoid unnecessary communication:

— due to the location of buffer cells from different domains in the same memory space it is no longer necessary to exchange them using message passing; instead those cells are referenced by different processors
— synchronization is necessary to allow concurrent access to shared data-structures
— consequently, all supportive operations that accompany buffer exchange (i.e., packing/unpacking and merging of submeshes) are no longer needed

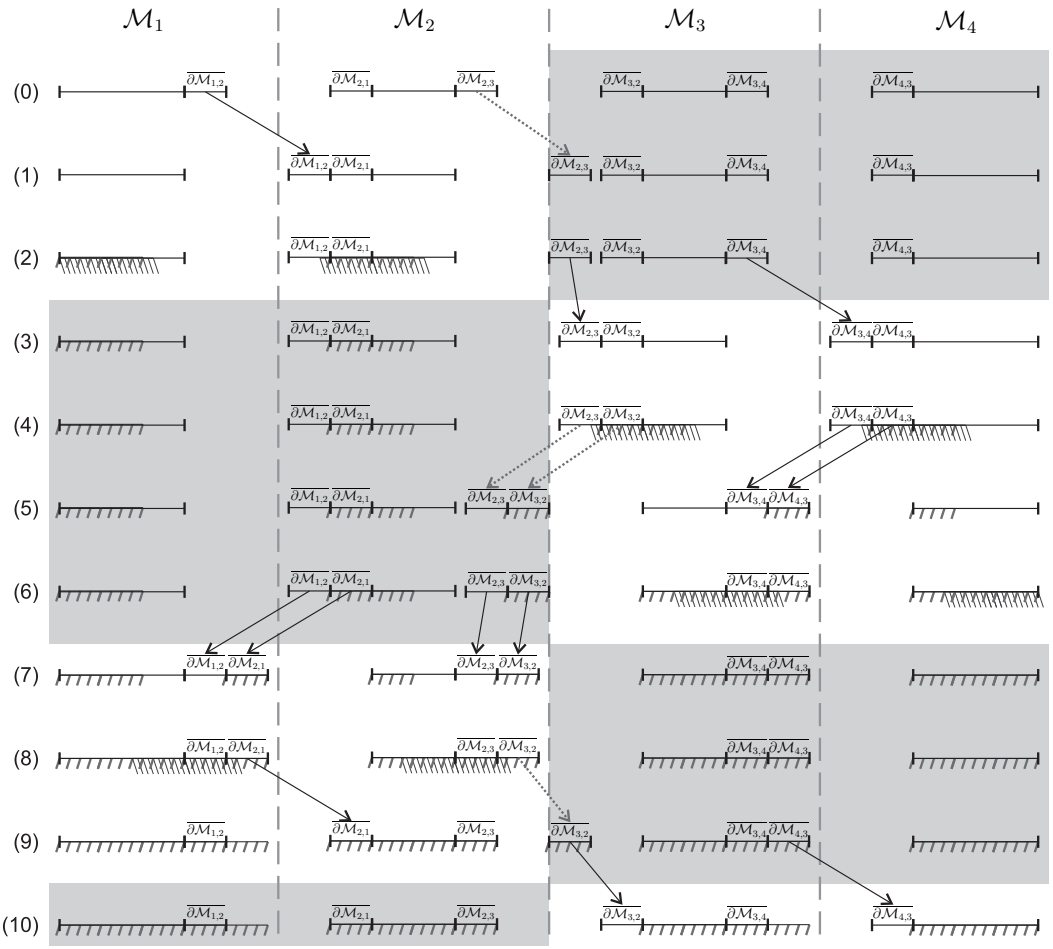

Fig. 4. An example of out-of-core PDR algorithm in one dimension. Mesh is comprised of four submeshes $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, $\mathcal{M}_4$ (there are two processors, RAM is limited so only one submesh can be loaded per processor), $\overline{\partial\mathcal{M}_{i,j}}$ denote border segments. Solid arrows between different steps indicate movements of submeshes between subdomains, dashed arrows indicate that a submesh will be stored on disk until it is required. Right dashed (thin lines) areas show parts that are being modified during refinement, left dashed (thick lines) areas show refined parts. Large gray-shaded areas show data that currently reside on disk.

Our evaluation showed [Kot et al. 2005] that performance of the Shared memory PDR (SPDR) is better than the original method when used on the same hardware platform. However, the difference is very small and the problem size is limited by the total memory of an SMP/SMT node. Nevertheless, this work was used to implement an advanced version of the out-of-core algorithm giving more of a performance boost (see Section 5).

4. OUT-OF-CORE PDR

See Figure 4 for an example of the algorithm execution with the mesh partitioned into four subdomains with only room for two in memory.

4.1. Shared Memory Out-of-Core PDR

The Out-of-Core Shared memory PDR (OSPDR) algorithm is designed to create large meshes in parallel, using only one node of the supercomputer with the hard disk complementing the memory. The following assumptions were made for the design of the OSPDR algorithm: (1) parts of the mesh stored on disk can be accessed by any processor that needs them but synchronization is necessary to handle collisions; (2) only a small fraction of the mesh can be loaded into the system memory, and (3) disk accesses have a very high latency. Therefore, our goal in OSPDR is to minimize the number of accesses and overlap them with computation whenever possible.

The mesh is stored on disk as a collection of subdomains. The subdomains are generated from the block decomposition using an auxiliary lattice we used for the PDR method. All processors can access all subdomains therefore no specific data distribution is required. The subdomains are stored as a sequence of separate entities, that is each subdomain is an atomic block and can be loaded/stored independently of the others, yet it must be loaded/stored as a whole. Only one subdomain can be loaded into processor memory at any time. Throughout the paper for simplicity of presentation we assume that subdomains send and receive data (e.g., if subdomain i is loaded into the memory of processor m and subdomain j is loaded into the memory of processor n and processor m sends data to processor n we say subdomain i sends data to subdomain j).

There are four main steps (we call them *phases*) in the PDR algorithm, each consists of a refinement step and a data exchange called *shift*. Since we only have enough memory to hold a portion of the mesh in-core it is impossible to perform a phase simultaneously for all subdomains as in the PDR. In OSPDR, we break each phase into several steps. At each step we load a portion of the mesh, refine it, exchange data between in-core subdomains and store the updated portion of the mesh. We call the data exchanges between in-core subdomains a *shift*, in consistence with the PDR.

During a shift each subdomain[6] receives data from one of its neighbors and sends data to another. We define a *direction* of a shift as a relative geometric position of the subdomain that receives data with regards to the position of the subdomain that sends data. All shifts in a phase share the same direction which is the direction of the phase.

There are two distinct types of phases based on their direction: *parallel* (up, right, down, left) and *diagonal* (up-right, down-right, down-left, up-left). We only need to explain one of each, the rest can be understood by analogy. In particular, we describe the phase with right shift and the phase with down-right shift. $PDR_{\text{refinement}}$ refines a portion of the mesh using external mesh library (Triangle). PDR_{shifts} integrates triangles in the border subdomain into the mesh. For more detailed description of $PDR_{\text{refinement}}$ and PDR_{shifts} see the in-core algorithm [Chernikov and Chrisochoides 2004].

[6]With the exception of the boundary subdomains.

A phase with parallel direction is rather straightforward, the order of refinement (geometrical direction in which blocks are loaded, refined and stored back to disk) coincides with the direction of the shift:

OSPDR.HORIZONTALSHIFT($\mathcal{M}$, $\mathcal{X}$, $\bar{\Delta}$, $\bar{\rho}$, P, p, N)
Input: $\mathcal{M}$ is a Delaunay mesh computed in previous phase(s)
$\mathcal{X}$ is a planar straight line graph which defines the domain of $\mathcal{M}$
$\bar{\Delta}$ and $\bar{\rho}$ are desired upper bounds on triangle area
and circumradius-to-shortest edge ratio, respectively
P is the total number of processors ($\sqrt{P}$ is integer)
p is the index of the current processor, $1 \leq p \leq P$
N^2 is the total number of subdomains ($N/\sqrt{P}$ is integer)
Output: a (partially) refined Delaunay mesh $\mathcal{M}_p$ which conforms to $\mathcal{X}$
and respects (in certain regions) $\bar{\Delta}$ and $\bar{\rho}$

```
0   Calculate row(p) and col(p) of the current processor
      // 1 ≤ row(p), col(p) ≤ √P
1   for m = 1, ..., N
2     for n = 1, ..., N
3       Load block p of subdomain (m − 1) × N + n as local mesh M_p
4       if n ≠ 0 and col(p) = 1
5         Reference cells {c_{i,1} | 1 ≤ i ≤ 4} of local mesh M_p
6       endif
7       M_p ← PDR_refinement(M_p, Δ̄, ρ̄, P, p)
8       M_p ← PDR_shifts(M_p, Δ̄, ρ̄, P, p)
9       if col(p) = √P and n ≠ N
10        Assign cells {c_{i,4} | 1 ≤ i ≤ 4} to processor in (row(p), 1)
11      endif
12      Store local mesh M_p as block p of subdomain (m − 1) × N + n
13    endfor
14  endfor
15  return M_p
```

A phase with diagonal direction is more complex, because the corner cell shifts both horizontally and vertically and both groups of side cells shift into their respective directions:

OSPDR.DIAGONALSHIFT($\mathcal{M}$, $\mathcal{X}$, $\bar{\Delta}$, $\bar{\rho}$, P, p, N)
Input: same as in OSPDR.HorizontalShift
Output: a (partially) refined Delaunay mesh $\mathcal{M}_p$ which conforms to $\mathcal{X}$

```
0   Calculate row(p) and col(p) of the current processor
      // 1 ≤ row(i), col(i) ≤ √P
1   for m = 1, ..., N
2     for n = 1, ..., N
3       Load block p of subdomain (m − 1) × N + n as local mesh M_p
4       if n ≠ 0 and col(p) = 1
5         Reference cells {c_{i,1} | 1 ≤ i ≤ 3} of local mesh M_p
6       endif
7       M_p ← PDR_refinement(M_p, Δ̄, ρ̄, P, p)
8       M_p ← PDR_shifts(M_p, Δ̄, ρ̄, P, p)
9       if col(p) = √P and n ≠ N
10        Assign cells {c_{i,4} | 1 ≤ i ≤ 3} to processor in (row(p), 1)
11      endif
12      if row(p) = √P and m ≠ N
13        Assign cells {c_{4,i} | 1 ≤ i ≤ 3} to processor in (1, col(p))
14      endif
15      if p = P and n ≠ N and m ≠ N
16        Assign cell c_{4,4} to processor in (1, 1)
```

```
17            endif
18            if row(p) = 1 and m < N
19                Reference cells {b_{1,i} | 1 ≤ i ≤ 3} of local buffer B
20                Overwrite cells {c_{1,i} | 1 ≤ i ≤ 3} of block p in subdomain m × N + n with B content
21            endif
22            if n < N and m < N
23                Reference cell b_{1,1} of local buffer B
24                Overwrite cell c_{1,1} of block p in subdomain m × N + n + 1 with B content
25            endif
26            Store local mesh M_p as block p of subdomain (m − 1) × N + n
27        endfor
28    endfor
29    return M_p
```

4.2. Out-of-core Distributed Memory PDR

The Out-of-core Distributed memory PDR (ODPDR) algorithm is designed to create very large meshes in parallel, using the aggregate and concurrent access of disk space through multiple nodes of a CoW. The following assumptions were made for the design of the ODPDR algorithm: (1) parts of the mesh stored on disk can only be accessed by the processor that the disk is directly attached to; (2) only a small fraction of the mesh can be loaded into the system memory, and (3) network and disk accesses have a very high latency. Therefore our goal in ODPDR is to minimize the number of accesses and overlap them with computation whenever possible.

The mesh is stored on disk as a collection of subdomains. The subdomains are generated from the block decomposition (using an auxiliary lattice) we used for the PDR method. The ODPDR uses different from PDR assignment of the cells to processors, but relies on the PDR (in-core) parallel Delaunay meshing and refinement code.

Optimal data distribution reduces the amount of communication to a necessary minimum and consequently lowers associated latencies. We propose an interleaving block partitioning (see Figure 5, left). That is the domain is partitioned into N^2 subdomains, where N is a number related to the size of the mesh and the amount of available RAM. Each subdomain is further partitioned into P *blocks*, where P is the total number of processors. Since P is a constant for every configuration, N is chosen such that the memory requirements of any single *block* is small enough to fully fit into RAM of a single node. The total number of blocks in the domain is $P \times N^2$; each processor stores (on local disk) one block from each subdomain, total of N^2 blocks. This scattered decomposition helps to implicitly improve workload imbalances. Similarly to OSPDR we will only explain one horizontal and one diagonal shifts.

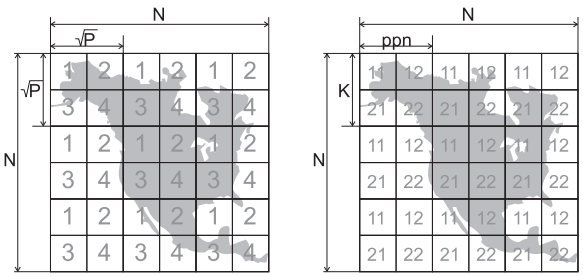

Fig. 5. An example of domain partitioning for the ODPDR (left) and the OHPDR (right) methods. P is the number of processors in 1 processor/core per node scenario, ppn is the number of processors per node, K is the number of nodes. N is derived empirically and depends on amount of memory and disk space (N^2 is the total number of subdomains).

The horizontal/vertical type of top-level shift is rather straightforward, the order of refinement coincides with the direction of the shift (see Figure 6):

ODPDR.HORIZONTALSHIFT($\mathcal{M}$, $\mathcal{X}$, $\bar{\Delta}$, $\bar{\rho}$, P, p, N)
Input: $\mathcal{M}$ is a Delaunay mesh computed in previous phase(s)
$\mathcal{X}$ is a planar straight line graph which defines the domain of $\mathcal{M}$
$\bar{\Delta}$ and $\bar{\rho}$ are desired upper bounds on triangle area
and circumradius-to-shortest edge ratio, respectively
P is the total number of processors ($\sqrt{P}$ is integer)
p is the index of the current processor, $1 \le p \le P$
N^2 is the total number of subdomains ($N/\sqrt{P}$ is integer)
Output: a (partially) refined Delaunay mesh $\mathcal{M}_p$ which conforms to $\mathcal{X}$
and respects (in certain regions) $\bar{\Delta}$ and $\bar{\rho}$

```
0   Calculate row(p) and col(p) of the current processor
        // 1 ≤ row(i), col(i) ≤ √P,  1 ≤ i ≤ P
1   for m = 1, ..., N
2       for n = 1, ..., N
3           Load block p of subdomain (m − 1) × N + n as local mesh M_p
4           if n ≠ 0 and col(p) = 1
5               Receive cells {c_{i,1} | 1 ≤ i ≤ 4} of local mesh M_p
6           endif
7           M_p ← PDR_refinement(M_p, Δ̄, ρ̄, P, p)
8           M_p ← PDR_shifts(M_p, Δ̄, ρ̄, P, p)
9           if col(p) = √P and n ≠ N
10              Send cells {c_{i,4} | 1 ≤ i ≤ 4} to processor in (row(p), 1)
11          endif
12          Store local mesh M_p as block p of subdomain (m − 1) × N + n
13      endfor
14  endfor
15  return M_p
```

The diagonal shift is more complex, because the corner cell shifts both horizontally and vertically and both groups of side cells shift into their respective directions (see Figure 6):

ODPDR.DIAGONALSHIFT($\mathcal{M}$, $\mathcal{X}$, $\bar{\Delta}$, $\bar{\rho}$, P, p, N)
Input: same as in ODPDR.HorizontalShift
Output: a (partially) refined Delaunay mesh $\mathcal{M}_p$ which conforms to $\mathcal{X}$

```
0   Calculate row(p) and col(p) of the current processor
        // 1 ≤ row(i), col(i) ≤ √P,  1 ≤ i ≤ P
1   for m = 1, ..., N
2       for n = 1, ..., N
3           Load block p of subdomain (m − 1) × N + n as local mesh M_p
4           if n ≠ 0 and col(p) = 1
5               Receive cells {c_{i,1} | 1 ≤ i ≤ 3} of local mesh M_p
6           endif
7           M_p ← PDR_refinement(M_p, Δ̄, ρ̄, P, p)
8           M_p ← PDR_shifts(M_p, Δ̄, ρ̄, P, p)
9           if col(p) = √P and n ≠ N
10              Send cells {c_{i,4} | 1 ≤ i ≤ 3} to processor in (row(p), 1)
11          endif
12          if row(p) = √P and m ≠ N
13              Send cells {c_{4,i} | 1 ≤ i ≤ 3} to processor in (1, col(p))
14          endif
15          if p = P and n ≠ N and m ≠ N
16              Send cell c_{4,4} to processor in (1, 1)
17          endif
```

```
18              if row(p) = 1 and m < N
19                  Receive cells {b_{1,i} | 1 ≤ i ≤ 3} of local buffer B
20                  Overwrite cells {c_{1,i} | 1 ≤ i ≤ 3} of block p in subdomain m × N + n with B content
21              endif
22              if n < N and m < N
23                  Receive cell b_{1,1} of local buffer B
24                  Overwrite cell c_{1,1} of block p in subdomain m × N + n + 1 with B content
25              endif
26              Store local mesh M_p as block p of subdomain (m − 1) × N + n
27          endfor
28      endfor
29      return M_p
```

4.3. Out-of-core Hybrid Memory PDR

To take full advantage of current hardware trend of having multiple processors / cores per node we designed and implemented the Out-of-core Hybrid memory PDR (OHPDR). Indeed, our experimental study (see Section 5) showed that the OHPDR method is faster than the ODPDR on nodes with more than one processor / core. We made the same design assumptions as in the case of the ODPDR, additionally, processors of the same node have equal access time to its local disk.

The mesh is stored on disks as a collection of subdomains generated from the block decomposition (using the auxiliary lattice). Part of the code responsible for meshing is taken from the OSPDR, but the assignment of cells to processors is different. We use an interleaving partition similar to the one used in the ODPDR (see Figure 5, right). The mesh is divided into N^2 subdomains, where N is a number related to the size of the mesh and the amount of available RAM. Each subdomain is then subdivided into $ppn \times K$ *blocks*, where K is the number of SMP nodes and ppn is the number of processors per node. The value of N is chosen in the same way we chose the number of subdomains for the ODPDR method.

The OHPDR also (as the ODPDR) uses the same two levels of data movements. However, a shift can be either *shared* (between processors of an SMP) or *distributed*, over the network (between nodes). Similarly, there are two distinct types of top-level shifts: horizontal/vertical and diagonal. We will only focus on the horizontal shift to the right and the diagonal shift to the right and down (the rest is done by analogy).

A top-level horizontal shift is performed in the following steps (see Figure 7):

```
OHPDR.HorizontalShift(M, X, Δ̄, ρ̄, K, ppn, p, N)
Input: ppn is the number of processors per node (the same number of
           processors on all nodes)
       K is the number of nodes (we assume √(K * ppn) is integer and,
       for simplicity of the presentation, K = ppn)
       p is the index of the current processor, 1 ≤ p ≤ ppn × K
       M, X, Δ̄, ρ̄ and N are the same as in ODPDR.HorizontalShift
Output: a (partially) refined Delaunay mesh M_p which conforms to X
            and respects (in certain regions) Δ̄ and ρ̄
0   Calculate node(p) and proc(p) of the current processor
        // 1 ≤ node(i) ≤ K, 1 ≤ proc(i) ≤ ppn, 1 ≤ i ≤ ppn × K
1   for m = 1, ..., N
2       for n = 1, ..., N
3           Load block p of subdomain (m − 1) × N + n as local mesh M_p
4           if n ≠ 0 and proc(p) = 1
5               Read cells {c_{i,1} | 1 ≤ i ≤ 4} of local mesh M_p from shared-memory buffer
6           endif
7           M_p ← SPDR_refinement(M_p, Δ̄, ρ̄, ppn, K, p)
```

```
 8            M_p ← SPDR_shifts(M_p, Δ̄, ρ̄, ppn, K p)
 9            if proc(p) = ppn and n ≠ N
10                Write cells {c_{i,4} | 1 ≤ i ≤ 4} into shared-memory buffer
11            endif
12            Store local mesh M_p as block p of subdomain (m − 1) × N + n
13        endfor
14    endfor
15    return M_p
```

The top-level diagonal shift to the right and down is performed in the following steps (see Figure 7):

```
OHPDR.DiagonalShift(M, X, Δ̄, ρ̄, K, ppn, p, N)
Input: same as in OHPDR.HorizontalShift
Output: a (partially) refined Delaunay mesh M_p which conforms to X
 0    Calculate node(p) and proc(p) of the current processor
          // 1 ≤ node(i) ≤ K, 1 ≤ proc(i) ≤ ppn, 1 ≤ i ≤ ppn × K
 1    for m = 1, ..., N
 2        for n = 1, ..., N
 3            Load block p of subdomain (m − 1) × N + n as local mesh M_p
 4            if n ≠ 0 and proc(p) = 1
 5                Read cells {c_{i,1} | 1 ≤ i ≤ 3} of local mesh M_p from shared-memory buffer
 6            endif
 7            M_p ← SPDR_refinement(M_p, Δ̄, ρ̄, ppn, K, p)
 8            M_p ← SPDR_shifts(M_p, Δ̄, ρ̄, ppn, K, p)
 9            if proc(p) = ppn and n ≠ N
10                Write cells {c_{i,4} | 1 ≤ i ≤ 3} into shared-memory buffer
11            endif
12            if node(p) = K and m ≠ N
13                Send cells {c_{4,i} | 1 ≤ i ≤ 3} to node node(p)
14            endif
15            if proc(p) = ppn and node(p) = K and n ≠ N and m ≠ N
16                Send cell c_{4,4} to node 1
17            endif
18            if node(p) = 1 and m < N
19                Receive cells {b_{1,i} | 1 ≤ i ≤ 3} of local buffer B
20                Overwrite cells {c_{1,i} | 1 ≤ i ≤ 3} of block p in subdomain m × N + n with B content
21            endif
22            if n < N and m < N
23                Receive cell b_{1,1} of local buffer B
24                Overwrite cell c_{1,1} of block p in subdomain m × N + n + 1 with B content
25            endif
26            Store local mesh M_p as block p of subdomain (m − 1) × N + n
27        endfor
28    endfor
29    return M_p
```

5. PERFORMANCE EVALUATION

For the evaluation of all PDR algorithms we used the cluster of Center for Real-time Computing[7] at the College of William and Mary. The cluster consists of four, four-way SMP IBM OpenPower720 compute nodes, with IBM Power5 processors clocked at 1.62 GHz and 8 GB of physical memory on every node. The IBM Power5 is a dual-core processor, and each one of its cores is organized as a simultaneous multi-threading execution engine, running two concurrent threads of control from the same or different address spaces. The processor has a large L2 cache (1.9 MB organized in three banks) which is shared between the cores via a crossbar switch, and a very large (36 MB) dedicated L3 cache, which is also shared between the processor's cores and threads. The

[7]http://crtc.wm.edu/

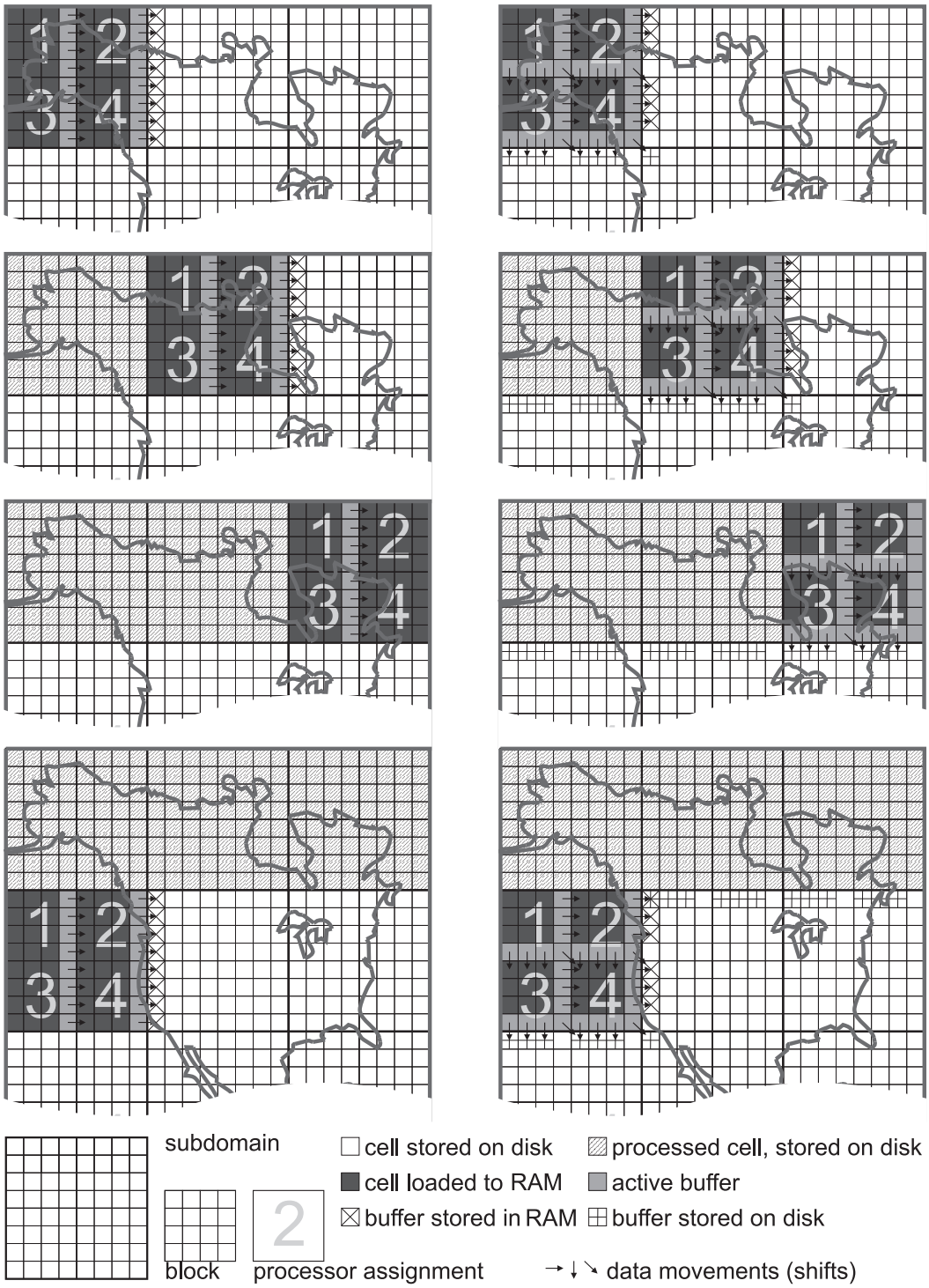

Fig. 6. Out-of-core schemes of top-level shifts for ODPDR: along axis (left) and diagonal (right). Setup: 4 processors, 9 subdomains, distributed memory and disk storage.

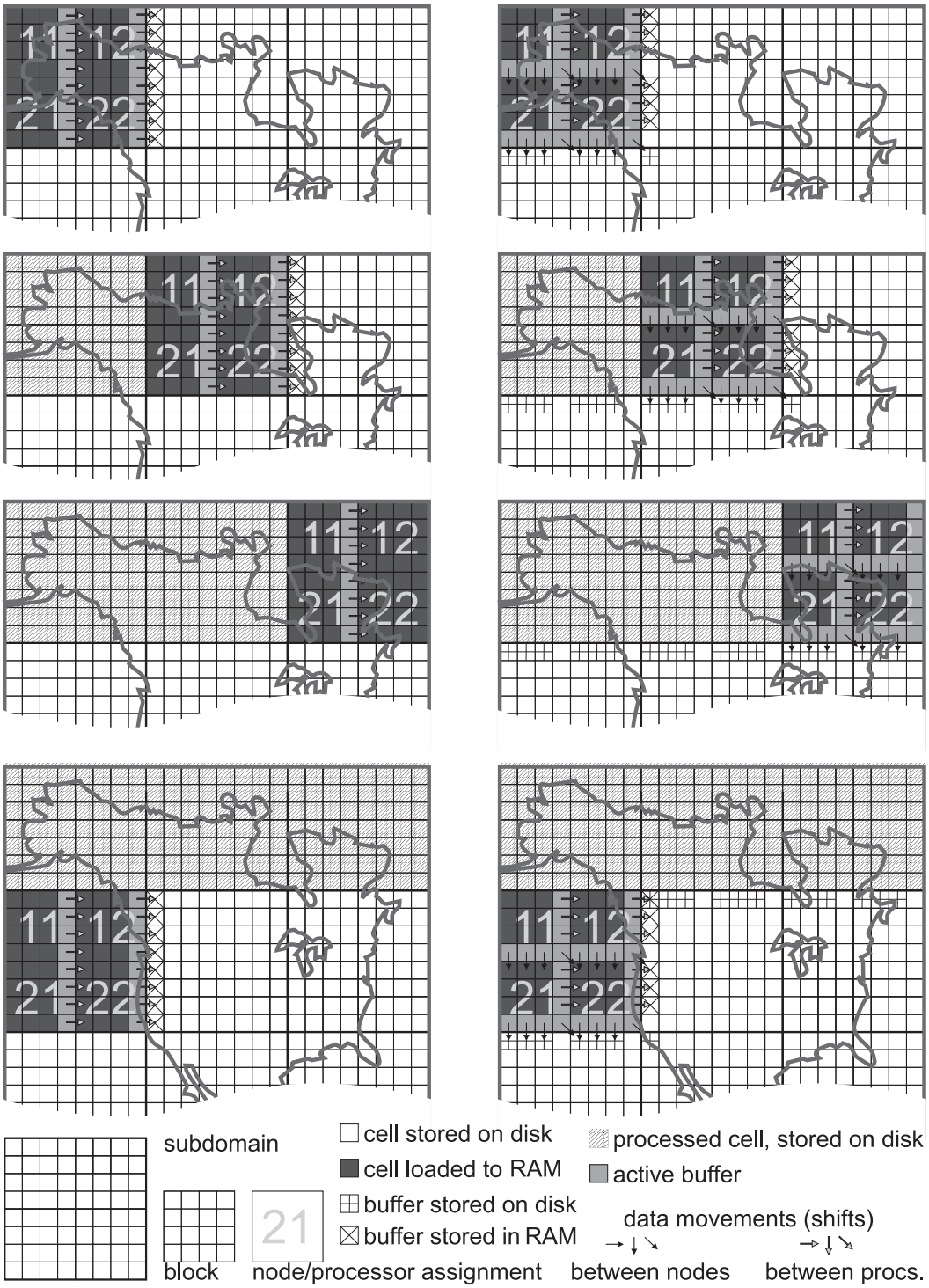

Fig. 7. Out-of-core schemes of top-level shifts for OHPDR: along axis (left) and diagonal (right). Setup: 2 nodes, 2 processors with shared memory per node, 9 subdomains, disk storage.

Table I. Parallel Delaunay refinement for a mesh of a unit square using the IBM cluster. The OSPDR, the ODPDR and the OHPDR use 4 processors; the PDR uses 4, 9, 16 and 25 processors.

Mesh size, # elements $\times 10^6$	PDR	OSPDR	ODPDR	OHPDR
	execution time, sec			
23.8	121(4)	249	276	264
58.8	105(9)	438	486	444
109.3	116(16)	631	639	578
175.4	114(25)	1136	1236	1257

Table II. Parallel Delaunay refinement for a mesh of a unit square using the IBM cluster. The OSPDR, the ODPDR and the OHPDR use 4 processors; the PDR uses 4, 9, 16 and 25 processors.

Mesh size, # elements $\times 10^6$	PDR	OSPDR	ODPDR	OHPDR
	normalized speed ($\times 10^3$ triangles per sec per proc)			
23.8	49.10(4)	23.87	21.52	22.53
58.8	62.01(9)	33.56	30.24	33.12
109.3	58.67(16)	43.28	42.76	47.26
175.4	61.67(25)	38.61	35.47	34.89

nodes are interconnected with Gigabit Ethernet and the cluster is accessible from the outside world via Gigabit lines as well. The main 16-processor, 32-core, 64-thread compute infrastructure, is stored in one rack along with one dual-processor OpenPower720 storage server and one dual-processor OpenPower720 management and software development node.

All algorithms are independent of the geometry of the domain, however, for our performance evaluation we used a square geometry to eliminate other parameters like work-load imbalance. This and other issues of the in-core algorithm are addressed in non-uniform Parallel Delaunay Refinement algorithm [Chernikov and Chrisochoides 2004] and are out of scope of this paper. However, is should be noted that over-decomposition introduced by out-of-core algorithms somewhat improves the work-load imbalance. We tested it with a mesh of a cross section of a pipe model that is part of a rocket fuel system (see Figure 1, left). This test geometry shows that the impact of load imbalances is much less severe to the out-of-core PDR algorithms.

In order to compare the performance of the in-core and the OoC PDR methods which run on differing number of nodes, we use *normalized speed*. This measure computes the number of elements generated by a single processor over a unit time period, and it is given by $V = \frac{N}{T \times P}$, N is the number of elements generated, P is the number of processors in the configuration and T is the total execution time.

Tables I and II shows the performance of all three out-of-core methods on a single 4-way SMP node from the workstation. The PDR performance is also included for comparison. However, the PDR has to use 9, 16 and 25 processors, respectively from the second problem and on since they would not fit in the aggregate memory of fewer processors. As expected, OSPDR and OHPDR show the best performance (not including the PDR). The ODPDR does not take advantage of shared memory and thus is slower. These data show that for some cases, the OHPDR method is only 19% slower, usually, it is about twice as slow (which is acceptable) as its counterpart in-core PDR method for the mesh sizes that fit completely in the core of the CoWs.

Table III. Parallel Delaunay refinement for the unit square. The ODPDR and the OHPDR1 use 16 processors (4 nodes, 4 CPU per node); the OHPDR2 uses 16 processors (2 nodes, 8 CPUs per node) of the IBM cluster; the PDR uses up to 121 processors of the SciClone cluster. In parentheses on the PDR column are the corresponding values from running the in-core PDR on up to 32 processors of the IBM cluster (there are only 32 computing processors total in IBM cluster). Wait-in-queue time is included when computing normalized speed for the in-core algorithm.

Mesh size, # elements $\times 10^6$	PDR		ODPDR	OHPDR1	OHPDR2
	normalized speed ($\times 10^3$ triangles per sec per proc)				
109.3	23.24	(98.1)	53.33	53.01	45.23
175.4	23.78	(96.74)	52.24	47.61	43.24
255.0	24.01	(98.32)	42.12	48.54	44.51
352.6	24.23	(97.84)	39.8	40.9	38.45
470.7	25.1	(100.2)	52.1	46.24	43.18
587.8	24.6		49.8	50.23	47.12
738.9	24.63		47.27	50.43	46.88
873.5	24.55		51.2	49.67	45.81
1284.1	23.11		50.6	48.72	44.14
1967.2	24.23		49.82	50.01	46.12

Table III shows the performance of distributed memory out-of-core PDR methods along with the in-core PDR using up to 121 processors. The unit square is used as a test case. The OHPDR is tested on two slightly different configurations: (1) using 16 nodes with a single processor per node, listed as OHPDR1 and (2) using 8 nodes with two processors per node, listed as OHPDR2. The OSPDR being designed solely for shared memory cannot run on these configurations.

The performance of both OoC methods is similar on the same configuration which is expected since the OHPDR does not take advantage of shared-memory. On SMP nodes the OHPDR (listed as OHPDR2) performs slightly worse. This is the opposite of the results we have seen on another system [Kot et al. 2006]. It is likely due to smaller cache (per core) and/or different implementations of MPI and OpenMP.

The normalized speed of the parallel OoC methods is approximately constant for all large problem sizes we ran. This suggests that the parallel OoC methods scale very well with respect to the problem size.

The total execution time for just under 2 billion elements is a little over one hour and a half (one hour and 37 minutes) using parallel OoC methods and 16 processors. However, the wait-in-queue delays for parallel jobs with more than 100 processors (they are required to generate the same size mesh using the in-core PDR) in our cluster is on average about five hours. However, on the same cluster the waiting time for 16 processors is less than half an hour. This makes the OHPDR2 response time 3.3 times shorter than the response time of the in-core PDR, for mesh sizes close to a billion elements.

Moreover, many scientific computing groups can afford to own a dedicated 8 to 16 processor cluster which means zero waiting time. Thus, the parallel OoC methods are much more effective and even faster if one uses the total "wall-clock" time.

Table IV shows the performance of distributed and shared memory OoC methods along with the PDR on large configurations for an irregular geometry, the pipe model. The uniform block data decomposition we used for the pipe model results in an uneven distribution of work to processors. This load imbalance on average reduces the speed for both the in-core method (by 61%) and the OoC method (by 27%). In the case of OoC methods, at every point of time processors refine only a portion of over-

Table IV. Parallel Delaunay refinement for a mesh of the pipe model. The ODPDR and the OHPDR use 16 processors (4 nodes, 4 CPUs per node); the PDR uses varying number of processors (16-121). Wait-in-queue time is included when computing normalized speed for the in-core algorithm.

Mesh size, # elements $\times 10^6$	PDR	ODPDR	OHPDR
	normalized speed ($\times 10^3$ triangles per sec per proc)		
58.3	16.12(16)	36.38	36.96
91.1	15.18(25)	35.21	35.85
131.2	14.29(36)	36.12	37.02
178.6	14.35(49)	35.78	36.65
233.3	13.3(64)	36.35	36.88
295.3	14.08(81)	35.10	36.03
364.6	15.72(100)	35.61	36.83
441.1	17.2(121)	35.89	37.12

Table V. I/O rates for reading and writing shown as percentages of maximum sustained I/O rates; Parallel Delaunay refinement for a mesh of a unit square using 4 nodes of the IBM cluster.

Method / operation	IO rate for large problem sizes ($\times 10^6$ elements) 587.8	738.9	873.5	1284.1	1967.2
ODPDR read	21.56	21.97	22.11	22.1	22.06
ODPDR write	18.02	17.72	18.03	17.98	18.22
OHPDR read	23.21	23.07	23.54	23.48	23.76
OHPDR write	21.06	20.92	21.48	21.44	21.34

decomposed [Barker et al. 2004] mesh, with all processor working in close proximity of each other. As a result, the workload is implicitly balanced because by far all processors have to perform approximately the same amount of computation.

Table V shows the I/O rates we achieved for large problem sizes. We computed these values by measuring the amount of data read or written and then subdividing it by the total execution time. We list the rates as percentages of the maximum sustained I/O rates on this machine. These rates are taken as sustained throughput of local disks (measured by reading/writing large files with Unix `dd` command), and are equal to 25.51 MB/s for reading and 18.02 MB/s for writing.

6. SUMMARY

We presented three OoC methods for parallel guaranteed quality Delaunay mesh generation. The OoC Shared memory PDR (OSPDR) method allows to generate comparatively large meshes on a single computing node with limited memory. The method takes advantage of multiple processors of an SMT/SMP but is only limited to a single node. The Out-of-core Distributed memory PDR (ODPDR) method allows to generate much larger meshes. Alternatively, it is faster to generate the same size mesh with the ODPDR if more processors are available. Finally, a combination of the OoC shared and distributed memory PDR (OHPDR) method is efficient for CoWs with k-way SMP nodes. The OoC methods are cost-effective in terms of response time. The total wall-clock time including wait-in-queue delays and total execution time for the OoC methods is 3.3 times shorter than the total wall-clock time for the in-core generation of the same meshes using more than one hundred processors. Our out-of-core methods are 19% to 56% slower than its counterpart in-core method for mesh sizes that fit completely in the core of the CoWs. This is a modest performance penalty for savings of

Table VI. Preliminary evaluation of PDR on BlueDrop using pipe model. SMT1 shows execution time with 1 threads enable per core and SMT4 shows execution time with 4 threads per core (consequently 4 times more computing threads).

Mesh size, $\times 10^6$ elems	Cores / Threads	Time, sec SMT1	SMT2
14.6	4 / 16	101	30
233.3	16 / 64	238	110
441.1	16 / 64	409	207

many hours in response time. Moreover, all three OoC codes use the fastest to our knowledge off-the-shelf sequential Delaunay mesh generator and thus leverage from on-going improvements in terms of quality, speed, and functionality of the sequential in-core Delaunay mesh generation methods.

Although the data we presented are from 2D geometries, the contribution of this paper is still important for two reasons: (1) the memory movement and communication patterns for 3D remains the same and thus the overheads for the 3D geometries will be much smaller since the 3D sequential meshers (e.g., Pyramid [Shewchuk 1997]) are more computationally intensive than their 2D counterpart (Triangle) which is the in-core mesh generation kernel we use and (2) 2D mesh generation is still important for some 3D simulations like direct numerical simulations of turbulence in cylinder flows ("drag" crisis simulations) with very large Reynolds numbers [Dong et al. 2004] and coastal ocean modeling for predicting storm surge and beach erosion in real-time [Walters 2005]. In both cases, 2D mesh generation is taking place in the xy-plane and it is replicated in the z-direction in the case of cylinder flows or using bathymetric contours in the case of coastal ocean modeling applications. With the increase of the Reynolds number (Re), the size of the mesh (in drag crisis simulations) grows in the order of $Re^{9/4}$ [Karniadakis and Orszag 1993], which motivates the use of parallel out-of-core mesh generation algorithms. Similarly, we have seen the few inches difference in damages (in the z-direction) made in two recent hurricanes in the Gulf Coast, this suggests very high resolution (and thus generation of very large meshes) for predicting storm surge and beach erosion.

We had a chance to evaluate PDR performance on BlueDrop, a representation of a single node of the emerging BlueWaters supercomputer (see Table VI). It shows very good utilization of hardware threads (speedup of 2 with 4 threads per core) which means the limiting resource for PDR will continue to be memory. We will look into using BlueWaters parallel disk storage (unfortunately it has not been made available yet) as well as partitioning a set of nodes into computing and storage. The storage nodes would only hold out-of-core data and in case of a complex application which uses PDR as a building block can be utilized to perform additional computations.

7. FUTURE WORK

Our primary focus is a PDR-based OoC method for 3D geometries now that the in-core method is available[Chernikov and Chrisochoides 2008b]. Additionally, we will research the possibility of using external memory libraries as "smart" storage systems for out-of-core data to relieve the programmer from the burden of developing a custom I/O subsystem. Another problem we want to solve is high level of load imbalance inherent to the uniform PDR methods. One possible solution is to adapt existing in-core non-uniform PDR method[Chernikov and Chrisochoides 2004] which does not suffer from this problem by design. Alternatively, we are developing the Multi-layered Runtime System (MRTS) that is designed to provide implicit out-of-core support as well as implicit load-balancing on multiple layers: inside a process between lightweight

threads, inside a multiprocessor node between processes and between nodes. The codes will need to be migrated to the MRTS which should be straightforward.

ACKNOWLEDGMENTS

The experimental work was performed using computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the NSF, and Virginia's Commonwealth Technology Research Fund.

We are grateful to Tom Crockett for making available to us the wait-in-queue time statistics from Sciclone.

REFERENCES

AGGARWAL, A. AND VITTER, JEFFREY, S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM 31,* 9, 1116–1127.

BARKER, K., CHERNIKOV, A., CHRISOCHOIDES, N., AND PINGALI, K. 2004. A load balancing framework for adaptive and asynchronous applications. *IEEE Transactions on Parallel and Distributed Systems 15,* 2, 183–192.

BLELLOCH, G. E., HARDWICK, J., MILLER, G. L., AND TALMOR, D. 1999. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica 24*, 243–269.

CHERNIKOV, A. AND CHRISOCHOIDES, N. 2006a. Generalized delaunay mesh refinement: From scalar to parallel. In *15th International Meshing Roundtable*. Birmingham, AL, 563–580.

CHERNIKOV, A. AND CHRISOCHOIDES, N. 2006b. Parallel guaranteed quality delaunay uniform mesh refinement. *SIAM Journal on Scientific Computing 28*, 1907–1926.

CHERNIKOV, A. AND CHRISOCHOIDES, N. 2008a. Algorithm 872: Parallel 2d constrained delaunay mesh generation. *ACM Transactions on Mathematical Software 34*, 6–25.

CHERNIKOV, A. AND CHRISOCHOIDES, N. 2008b. Three-dimensional delaunay refinement for multi-core processors. In *22nd ACM International Conference on Supercomputing*. Island of Kos, Greece, 214–224.

CHERNIKOV, A. N. AND CHRISOCHOIDES, N. P. 2004. Practical and efficient point insertion scheduling method for parallel guaranteed quality Delaunay refinement. In *Proceedings of the 18th International Conference on Supercomputing*. ACM Press, 48–57.

CHERNIKOV, A. N. AND CHRISOCHOIDES, N. P. 2009. Generalized two-dimensional Delaunay mesh refinement. *SIAM Journal on Scientific Computing 31,* 5, 3387–3403.

CHOI, J., DONGARRA, J., POZO, R., AND WALKER, D. 1992. Scalapack: A scalable linear algebra for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 120–127.

CHRISOCHOIDES, N. 2005. Parallel mesh generation. *Numerical Solution of Partial Differential Equations on Parallel Computers 51*. Eds. Are Magnus Bruaset, Petter Bjorstad, Aslak Tveito.

D'AZEVEDO, E. F. AND DONGARRA, J. 2000. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. *Concurrency - Practice and Experience 12,* 15, 1481–1493.

DEHNE, F., DITTRICH, W., AND HUTCHINSON, D. 1997. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *In Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*. 106–115.

DEMMEL, J., DONGARRA, J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., AND SORENSEN, D. 1987. Prospectus for the development of a linear algebra library for high-performance computers. Tech. Rep. ANL/MCS-TM-97, 9700 South Cass Avenue, Argonne, IL 60439-4801, USA.

DONG, S., LUCOR, D., AND KARNIADAKIS, G. E. 2004. Flow past a stationary and moving cylinder: DNS at Re=10,000. In *2004 Users Group Conference (DOD_UGC'04)*. 88–95.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. J. 1988. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 14,* 1, 1–17.

FOTEINOS, P. A., CHERNIKOV, A. N., AND CHRISOCHOIDES, N. P. 2010. Fully generalized two-dimensional constrained Delaunay mesh refinement. *SIAM Journal on Scientific Computing 32,* 5, 2659–2686.

KARNIADAKIS, G. AND ORSZAG, S. 1993. Nodes, modes, and flow codes. *Physics Today 46*, 34–42.

KOT, A., CHERNIKOV, A., AND CHRISOCHOIDES, N. 2005. Out-of-core parallel delaunay mesh generation for shared memory machines. In *Proceedings of the 17th IMACS World Congress Scientific Computation, Applied Mathematics and Simulation*.

KOT, A., CHERNIKOV, A., AND CHRISOCHOIDES, N. 2006. Effective out-of-core parallel delaunay mesh refinement using off-the-shelf software. In *20th IEEE International Parallel and Distributed Processing Symposium*. Rhodes Island, Greece, 104–114.

LINARDAKIS, L. AND CHRISOCHOIDES, N. 2008a. Algorithm 870: A static geometric medial axis domain decomposition in 2d euclidean space. *ACM Transactions on Mathematical Software 34*, 4:1–4:28.

LINARDAKIS, L. AND CHRISOCHOIDES, N. 2008b. Graded delaunay decoupling method for parallel guaranteed quality planar mesh generation. *SIAM Journal on Scientific Computing 30*, 1875–1891.

NAVE, D., CHRISOCHOIDES, N., AND CHEW, L. P. 2002. Guaranteed: quality parallel Delaunay refinement for restricted polyhedral domains. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*. ACM Press, 135–144.

NODINE, M. H. AND VITTER, J. S. 1995. Greed sort: optimal deterministic sorting on parallel disks. *J. ACM 42,* 4, 919–933.

RUPPERT, J. 1995. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms 18(3)*, 548–585.

SALMON, J. AND WARREN, M. 1997. Parallel out-of-core methods for N-body simulation. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds. Lecture Notes in Computer Science Series, vol. 1148. Springer-Verlag, 203–222. From the First ACM Workshop on Applied Computational Geometry.

SHEWCHUK, J. R. 1997. Delaunay refinement mesh generation. Ph.D. thesis, Carnegie Mellon University.

SHEWCHUK, J. R. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications 22,* 1–3, 21–74.

TOLEDO, S. AND GUSTAVSON, F. 1996. The design and implementation of solar, a portable library for scalable out-of-core linear algebra computations. In *4th Annual Workshop on I/O in Parallel and Distributed Systems*. 28–40.

TU, T. AND O'HALLARON, D. R. 2004. A computational database system for generating unstructured hexahedral meshes with billions of elements. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, Washington, DC, USA, 25–39.

TU, T. AND O'HALLARON, D. R. 2005. Extracting hexahedral mesh structures from balanced linear octrees. In *Proceedings of the 13th International Meshing Roundtable*. Williamsburg, VA, USA, 191–200.

VITTER, J. S. AND NODINE, M. H. 1993. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing 17*, 107–114.

VITTER, J. S. AND SHRIVER, E. A. M. 1993. Algorithms for parallel memory ii: Hierarchical multilevel memories. *ALGORITHMICA 12*, 148–169.

VITTER, J. S., SHRIVER, E. A. M., AND Z, E. A. M. S. 1994. Algorithms for parallel memory i: Two-level memories. *Algorithmica 12*, 110–147.

WALTERS, R. A. 2005. Coastal Ocean Models: Two useful finite element methods. *Recent Developments in Physical Oceanographic Modelling: Part II 25*, 775–793.

Received Month Year; revised Month Year; accepted Month Year