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:

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 1084-6654/YYYY-ARTA $10.00
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 subproblems. 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 \(M\). The block decomposition is used to guide the parallel refinement, so that Steiner points, independently inserted in certain regions of \(M\), 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 \textit{uniform} PDR. We also developed the PDR method that employs a quadtree (octree in 3D) for data decomposition which we call \textit{non-uniform} PDR. In this paper we always refer to the \textit{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).

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
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 if 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 over the entire $n$-dimensional domain.

![Fig. 2. Subdivision of a mesh $\mathcal{M}$.](image)

For simplicity we begin by presenting the algorithm in one dimension. 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 $\alpha$ 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 $\partial \mathcal{M}_{i,j}$ 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., $\partial \mathcal{M}_{3,4}$ would be the rightmost segment of $\mathcal{M}_4$). Size of each bordering segment is $\beta \bar{r}$, where $\beta$ 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 $\partial \mathcal{M}_{i,j}$ inside $\mathcal{M}_i$ as $\partial \mathcal{M}'_{i,j}$. Figure 2 shows the subdivision\(^5\) of $\mathcal{M}$.

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

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

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

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 \(\gamma\) (\(\gamma\) is an implementation dependent constant, \(\gamma \leq \beta\); \(\gamma = 2\beta\) for our 2D implementation) from the bounding box of \(B\), resulting in refined mesh stored in \(A\).

The algorithm will perform the following steps:

\begin{enumerate}
\item \(\forall i, i \in I: M_i \leftarrow (M_i \setminus \partial M_{i+1} \cup \partial M_{i-1})\)
\item \(\forall i, i \in I: \text{refine}\left(M_i, (M_i \setminus (\partial M_{i+1} \cup \partial M_{i-1}))\right)\)
\item \(\forall i, i \in I: M_i \leftarrow M_i \cup (\partial M_{i+1} \cup \partial M_{i-1}) \setminus (\partial M_{i-1} \cup \partial M_{i+1})\)
\item \(\forall i, i \in I: \text{refine}\left(M_i, (M_i \setminus (\partial M_{i-1} \cup \partial M_{i+1}))\right)\)
\item \(\forall i, i \in I: M_i \leftarrow (M_i \setminus \partial M_{i+1} \cup \partial M_{i-1}) \setminus \partial M_{i+1}\)
\end{enumerate}

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 \(\partial M'_{i+1}\) and \(\partial M_{i-1}\) 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 \(\partial M_{i-1}\) and \(\partial M_{i+1}\) 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

![Diagram showing out-of-core PDR algorithm in one dimension](image)

**Fig. 4.** An example of out-of-core PDR algorithm in one dimension. Mesh is comprised of four submeshes $M_1, M_2, M_3, M_4$ (there are two processors, RAM is limited so only one submesh can be loaded per processor). $\partial M_{i,j}$ 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. $PDR_{refinement}$ 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 $PDR_{refinement}$ 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.HORIZONatalShift($\mathcal{M}, X, \Delta, \bar{\rho}, P, p, N$)

**Input:** $\mathcal{M}$ is a Delaunay mesh computed in previous phase(s)
- $X$ is a planar straight line graph which defines the domain of $\mathcal{M}$
- $\Delta$ 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 $X$
- and respects (in certain regions) $\Delta$ and $\bar{\rho}$

0 Calculate $row(p)$ and $col(p)$ of the current processor
\[ 1 \leq row(p), col(p) \leq \sqrt{P} \]
1 for $m = 1, \ldots, N$
2 for $n = 1, \ldots, N$
3 Load block $p$ of subdomain $(m - 1) \times N + n$ as local mesh $\mathcal{M}_p$
4 if $n \neq 0$ and $col(p) = 1$
5 Reference cells \{ $c_{i,1} | 1 \leq i \leq 4$ \} of local mesh $\mathcal{M}_p$
6 endif
7 $\mathcal{M}_p \leftarrow \text{PDRshift}(\mathcal{M}_p, \Delta, \bar{\rho}, P, p)$
8 $\mathcal{M}_p \leftarrow \text{PDRrefinement}(\mathcal{M}_p, \Delta, \bar{\rho}, P, p)$
9 if $col(p) = \sqrt{P}$ and $n \neq N$
10 Assign cells \{ $c_{i,4} | 1 \leq i \leq 4$ \} to processor in $(row(p), 1)$
11 endif
12 Store local mesh $\mathcal{M}_p$ as block $p$ of subdomain $(m - 1) \times N + n$
13 endfor
14 endfor
15 return $\mathcal{M}_p$

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($\mathcal{M}, X, \Delta, \bar{\rho}, P, p, N$)

**Input:** same as in OSPDR.HorizontalShift
*Output:* a (partially) refined Delaunay mesh $\mathcal{M}_p$, which conforms to $X$

0 Calculate $row(p)$ and $col(p)$ of the current processor
\[ 1 \leq row(p), col(p) \leq \sqrt{P} \]
1 for $m = 1, \ldots, N$
2 for $n = 1, \ldots, N$
3 Load block $p$ of subdomain $(m - 1) \times N + n$ as local mesh $\mathcal{M}_p$
4 if $n \neq 0$ and $col(p) = 1$
5 Reference cells \{ $c_{i,1} | 1 \leq i \leq 4$ \} of local mesh $\mathcal{M}_p$
6 endif
7 $\mathcal{M}_p \leftarrow \text{PDRrefinement}(\mathcal{M}_p, \Delta, \bar{\rho}, P, p)$
8 $\mathcal{M}_p \leftarrow \text{PDRshift}(\mathcal{M}_p, \Delta, \bar{\rho}, P, p)$
9 if $col(p) = \sqrt{P}$ and $n \neq N$
10 Assign cells \{ $c_{i,4} | 1 \leq i \leq 4$ \} to processor in $(row(p), 1)$
11 endif
12 if $row(p) = \sqrt{P}$ and $m \neq N$
13 Assign cells \{ $c_{1,i} | 1 \leq i \leq 4$ \} to processor in $(1, col(p))$
14 endif
15 if $p = P$ and $n \neq N$ and $m \neq N$
16 Assign cell $c_{4,4}$ to processor in $(1, 1)$

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.](image)

$P$ 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):

\[
\text{ODPDR HorizntalShift}(M, X, \Delta, \bar{\rho}, P, p, N)
\]

**Input**: \(M\) is a Delaunay mesh computed in previous phase(s)
- \(X\) is a planar straight line graph which defines the domain of \(M\)
- \(\Delta\) 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 \(M_p\), which conforms to \(X\) and respects (in certain regions) \(\Delta\) and \(\bar{\rho}\)

\[
\begin{align*}
0 & \quad \text{Calculate row}(p) \text{ and col}(p) \text{ of the current processor} \\
1 & \quad \text{// } 1 \leq \text{row}(i), \text{col}(i) \leq \sqrt{P}, \quad 1 \leq i \leq P \\
2 & \quad \text{for } m = 1, \ldots, N \\
3 & \quad \quad \text{for } n = 1, \ldots, N \\
4 & \quad \quad \quad \text{Load block } p \text{ of subdomain } (m-1) \times N + n \text{ as local mesh } M_p \\
5 & \quad \quad \quad \text{if } n \neq 0 \text{ and col}(p) = 1 \\
6 & \quad \quad \quad \quad \text{Receive cells } \{c_{i,1} | 1 \leq i \leq 4\} \text{ of local mesh } M_p \\
7 & \quad \quad \quad \text{endif} \\
8 & \quad \quad \quad M_p \leftarrow \text{PDRrefinemnt}(M_p, \Delta, \bar{\rho}, P, p) \\
9 & \quad \quad \quad M_p \leftarrow \text{PDRshift}(M_p, \Delta, \bar{\rho}, P, p) \\
10 & \quad \quad \quad \text{if col}(p) = \sqrt{P} \text{ and } n \neq N \\
11 & \quad \quad \quad \quad \text{Send cells } \{c_{i,4} | 1 \leq i \leq 4\} \text{ to processor in } (\text{row}(p), 1) \\
12 & \quad \quad \quad \text{endif} \\
13 & \quad \quad \text{Store local mesh } M_p \text{ as block } p \text{ of subdomain } (m-1) \times N + n \\
14 & \quad \text{endfor} \\
15 & \quad \text{endfor} \\
16 & \quad \text{return } M_p \\
\end{align*}
\]

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):

\[
\text{ODPDR DiagonalShift}(M, X, \Delta, \bar{\rho}, P, p, N)
\]

**Input**: same as in ODPDR HorizontalShift

**Output**: a (partially) refined Delaunay mesh \(M_p\) which conforms to \(X\)

\[
\begin{align*}
0 & \quad \text{Calculate row}(p) \text{ and col}(p) \text{ of the current processor} \\
1 & \quad \text{// } 1 \leq \text{row}(i), \text{col}(i) \leq \sqrt{P}, \quad 1 \leq i \leq P \\
2 & \quad \text{for } m = 1, \ldots, N \\
3 & \quad \quad \text{for } n = 1, \ldots, N \\
4 & \quad \quad \quad \text{Load block } p \text{ of subdomain } (m-1) \times N + n \text{ as local mesh } M_p \\
5 & \quad \quad \quad \text{if } n \neq 0 \text{ and col}(p) = 1 \\
6 & \quad \quad \quad \quad \text{Receive cells } \{c_{i,1} | 1 \leq i \leq 3\} \text{ of local mesh } M_p \\
7 & \quad \quad \quad \text{endif} \\
8 & \quad \quad \quad M_p \leftarrow \text{PDRrefinemnt}(M_p, \Delta, \bar{\rho}, P, p) \\
9 & \quad \quad \quad M_p \leftarrow \text{PDRshift}(M_p, \Delta, \bar{\rho}, P, p) \\
10 & \quad \quad \quad \text{if col}(p) = \sqrt{P} \text{ and } n \neq N \\
11 & \quad \quad \quad \quad \text{Send cells } \{c_{i,4} | 1 \leq i \leq 3\} \text{ to processor in } (\text{row}(p), 1) \\
12 & \quad \quad \quad \text{endif} \\
13 & \quad \quad \quad \text{if } \text{row}(p) = \sqrt{P} \text{ and } m \neq N \\
14 & \quad \quad \quad \quad \text{Send cells } \{c_{4,i} | 1 \leq i \leq 3\} \text{ to processor in } (1, \text{col}(p)) \\
15 & \quad \quad \quad \text{endif} \\
16 & \quad \quad \text{if } p = P \text{ and } n \neq N \text{ and } m \neq N \\
17 & \quad \quad \quad \text{Send cell } c_{4,4} \text{ to processor in } (1, 1) \\
18 & \quad \text{endif} \\
\end{align*}
\]
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 \( \text{ppn} \times K \) blocks, where \( K \) is the number of SMP nodes and \( \text{ppn} \) 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):

```
OH HDR . H ORI Z EN TA L S H IF T ( M, X, Δ, ̃ρ, K, pp n, p, N)

Input: pp n is the number of processors per node (the same number of processors on all nodes)
  \( K \) is the number of nodes (we assume \( \sqrt{K \times \text{ppn}} \) is integer and, for simplicity of the presentation, \( K = \text{ppn} \))
  \( p \) is the index of the current processor, \( 1 \leq p \leq \text{ppn} \times K \)
  \( M, X, Δ, ̃ρ \) and \( N \) are the same as in ODPDR.HorizontalShift

Output: a (partially) refined Delaunay mesh \( M_ρ \), which conforms to \( X \) and respects (in certain regions) \( Δ \) and \( ̃ρ \)

0. Calculate \( \text{node}(p) \) and \( \text{proc}(p) \) of the current processor

\( \begin{align*}
1 \leq \text{node}(i) & \leq K, 1 \leq \text{proc}(i) \leq \text{ppn}, 1 \leq i \leq \text{ppn} \times K \\
\end{align*} \)

1. for \( m = 1, \ldots, N \)

2. for \( n = 1, \ldots, N \)

3. Load block \( p \) of subdomain \( (m - 1) \times N + n \) as local mesh \( M_p \)

4. if \( n \neq 0 \) and \( \text{proc}(p) = 1 \)

5. Read cells \( \{c_{1,i} | 1 \leq i \leq 4 \} \) of local mesh \( M_p \) from shared-memory buffer

6. endif

7. \( M_p \leftarrow SPDR\text{refinement}(M_p, Δ, ̃ρ, \text{ppn}, K, p) \)
```
The top-level diagonal shift to the right and down is performed in the following steps (see Figure 7):

\[ OHPDR.DIAgonALShift(M, X, \Delta, \hat{\rho}, K, ppm, p, N) \]

**Input:** same as in OHPDR.HorizontalShift

**Output:** a (partially) refined Delaunay mesh \( M_p \) which conforms to \( X \)

1. Calculate \( node(p) \) and \( proc(p) \) of the current processor
2. if \( node(i) = K \) and \( i \neq ppm \) and \( m \neq ppm \) and \( n \neq ppm \)
3. Load block \( p \) of subdomain \( (m - 1) \times N + n \) as local mesh \( M_p \)
4. Read cells \( \{c_{i,4} | 1 \leq i \leq 3\} \) of local mesh \( M_p \) from shared-memory buffer
5. \( M_p \leftarrow SPDR_{\text{block}}(M_p, \Delta, \hat{\rho}, ppm, K, p) \)
6. endif
7. Write cells \( \{c_{i,4} | 1 \leq i \leq 3\} \) into shared-memory buffer
8. \( M_p \leftarrow SPDR_{\text{shift}}(M_p, \Delta, \hat{\rho}, ppm, K, p) \)
9. if \( proc(p) = ppm \) and \( n \neq N \)
10. Write cells \( \{c_{i,4} | 1 \leq i \leq 3\} \) into shared-memory buffer
11. endif
12. if \( node(p) = K \) and \( m \neq N \)
13. Send cells \( \{c_{4,1} | 1 \leq i \leq 3\} \) to \( node(p) \)
14. endif
15. if \( proc(p) = ppm \) and \( node(p) = K \) and \( n \neq N \) and \( m \neq N \)
16. Send cell \( c_{4,1} \) to node 1
17. endif
18. if \( node(p) = 1 \) and \( m < N \)
19. Receive cells \( \{b_{1,i} | 1 \leq i \leq 3\} \) of local buffer \( B \)
20. Overwrite cells \( \{c_{1,4} | 1 \leq i \leq 3\} \) of block \( p \) in subdomain \( m \times N + n \) 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 \( m \times N + n + 1 \) with \( B \) content
25. endif
26. Store local mesh \( M_p \) as block \( p \) of subdomain \( (m - 1) \times N + n \)
27. endif
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.

<table>
<thead>
<tr>
<th>Mesh size, # elements ×10^6</th>
<th>PDR</th>
<th>OSPDR</th>
<th>ODPDR</th>
<th>OHPDR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>execution time, sec</td>
<td>23.8</td>
<td>121(4)</td>
<td>249</td>
</tr>
<tr>
<td></td>
<td>58.8</td>
<td>105(9)</td>
<td>486</td>
<td>438</td>
</tr>
<tr>
<td></td>
<td>109.3</td>
<td>116(16)</td>
<td>639</td>
<td>631</td>
</tr>
<tr>
<td></td>
<td>175.4</td>
<td>114(25)</td>
<td>1236</td>
<td>1136</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Mesh size, # elements ×10^6</th>
<th>PDR</th>
<th>OSPDR</th>
<th>ODPDR</th>
<th>OHPDR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>normalized speed (×10^3 triangles per sec per proc)</td>
<td>23.8</td>
<td>49.10(4)</td>
<td>23.87</td>
</tr>
<tr>
<td></td>
<td>58.8</td>
<td>62.01(9)</td>
<td>30.24</td>
<td>33.56</td>
</tr>
<tr>
<td></td>
<td>109.3</td>
<td>58.67(16)</td>
<td>47.26</td>
<td>43.28</td>
</tr>
<tr>
<td></td>
<td>175.4</td>
<td>61.67(25)</td>
<td>34.89</td>
<td>38.61</td>
</tr>
</tbody>
</table>

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.

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

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-
Effective OoC Parallel Delaunay Mesh Refinement

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.

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

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.

<table>
<thead>
<tr>
<th>Method / operation</th>
<th>IO rate for large problem sizes (x 10^6 elements)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>587.8</td>
</tr>
<tr>
<td>ODPDR read</td>
<td>21.56</td>
</tr>
<tr>
<td>ODPDR write</td>
<td>18.02</td>
</tr>
<tr>
<td>OHPDR read</td>
<td>23.21</td>
</tr>
<tr>
<td>OHPDR write</td>
<td>21.06</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Mesh size, ( \times 10^6 ) elems</th>
<th>Cores / Threads</th>
<th>Time, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SMT1</td>
<td>SMT2</td>
</tr>
<tr>
<td>14.6</td>
<td>101</td>
<td>30</td>
</tr>
<tr>
<td>233.3</td>
<td>238</td>
<td>110</td>
</tr>
<tr>
<td>441.1</td>
<td>409</td>
<td>207</td>
</tr>
</tbody>
</table>

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 \( Re^{3/4} \) [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 Scilone.

REFERENCES


Received Month Year; revised Month Year; accepted Month Year