2011

# Effective Large Scale Computing Software for Parallel Mesh Generation

andriy Kot
*College of William & Mary - Arts & Sciences*

Recommended Citation
Kot, andriy, "Effective Large Scale Computing Software for Parallel Mesh Generation" (2011). *Dissertations, Theses, and Masters Projects.* Paper 1539623585.
https://dx.doi.org/doi:10.21220/s2-hsxy-8829

# Effective Large Scale Computing Software for Parallel Mesh Generation

Andriy Kot
Ternopil, Ukraine

Master of Science, The College of William and Mary, 2004
Master of Science, Ternopil Academy of National Economy, 2003

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
May 2011

# APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

[illegible]

Andriy Kot

Approved by the Committee, May 2011

[illegible]

Committee Chair
Professor Nikos Chrisochoides, Computer Science
Old Dominion University
Adjunct Professor, Computer Science
The College of William and Mary

Weizhen Mao

Associate Professor Weizhen Mao, Computer Science
The College of William and Mary

[illegible]

Associate Professor Qun Li, Computer Science
The College of William and Mary

[illegible]

Assistant Professor Xipeng Shen, Computer Science
The College of William and Mary

[illegible]

Professor Henry Krakauer, Physics
The College of William and Mary

# ABSTRACT PAGE

Scientists commonly turn to supercomputers or Clusters of Workstations with hundreds (even thousands) of nodes to generate meshes for large-scale simulations. Parallel mesh generation software is then used to decompose the original mesh generation problem into smaller sub-problems that can be solved (meshed) in parallel. The size of the final mesh is limited by the amount of aggregate memory of the parallel machine. Also, requesting many compute nodes on a shared computing resource may result in a long waiting, far surpassing the time it takes to solve the problem.

These two problems (i.e., insufficient memory when computing on a small number of nodes, and long waiting times when using many nodes from a shared computing resource) can be addressed by using out-of-core algorithms. These are algorithms that keep most of the dataset out-of-core (i.e., outside of memory, on disk) and load only a portion in-core (i.e., into memory) at a time.

We explored two approaches to out-of-core computing. First, we presented a traditional approach, which is to modify the existing in-core algorithms to enable out-of-core computing. While we achieved good performance with this approach the task is complex and labor intensive. An alternative approach, we presented a runtime system designed to support out-of-core applications. It requires little modification of the existing in-core application code and still produces acceptable results. Evaluation of the runtime system showed little performance degradation while simplifying and shortening the development cycle of out-of-core applications. The overhead from using the runtime system for small problem sizes is between 12% and 41% while the overlap of computation, communication and disk I/O is above 50% and as high as 61% for large problems.

The main contribution of our work is the ability to utilize computing resources more effectively. The user has a choice of either solving larger problems, that otherwise would not be possible, or solving problems of the same size but using fewer computing nodes, thus reducing the waiting time on shared clusters and supercomputers. We demonstrated that the latter could potentially lead to substantially shorter wall-clock time.

# Table of Contents

# ACKNOWLEDGMENTS

I am grateful to my adviser Nikos Chrisochoides for everything he has done.

I thank Andrey Chernikov for his collaboration on numerous projects.

I very much appreciate feedback and support from my committee: Henry Krakauer, Weizhen Mao, Qun Li and Xipeng Shen.

Last but not least, I thank the people who suggested and helped me to get into graduate school, Allen Tucker and his wife Meg.

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

This work was 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.

# List of Tables

# List of Figures

# Effective Large Scale Computing Software for Parallel Mesh Generation

# Chapter 1

# Introduction

Scientists commonly turn to supercomputers or Clusters of Workstations (CoW) with hundreds (even thousands) of nodes to generate meshes for large-scale simulations. A parallel mesh generation software is then used to decompose the original mesh generation problem into smaller sub-problems that can be solved (meshed) in parallel. The limiting factor is memory – it is not possible to generate the mesh if memory is not sufficient.

While mesh generation time shortens with increasing the number of Processing Elements (PEs), however there are other factors that could affect the total wall clock (or completion) time significantly. For instance, it takes only several minutes to generate largest possible mesh (for available memory) using one of our parallel mesh generation applications on SciClone Cluster at the College of William and Mary. At the same time, according to collected statistics from the about four and a half years (see Fig. 1.1) the waiting time is considerably longer than the actual execution time for cases requiring more than 16 PEs.

The only way to decrease the waiting time that is in the power of the user is to ask

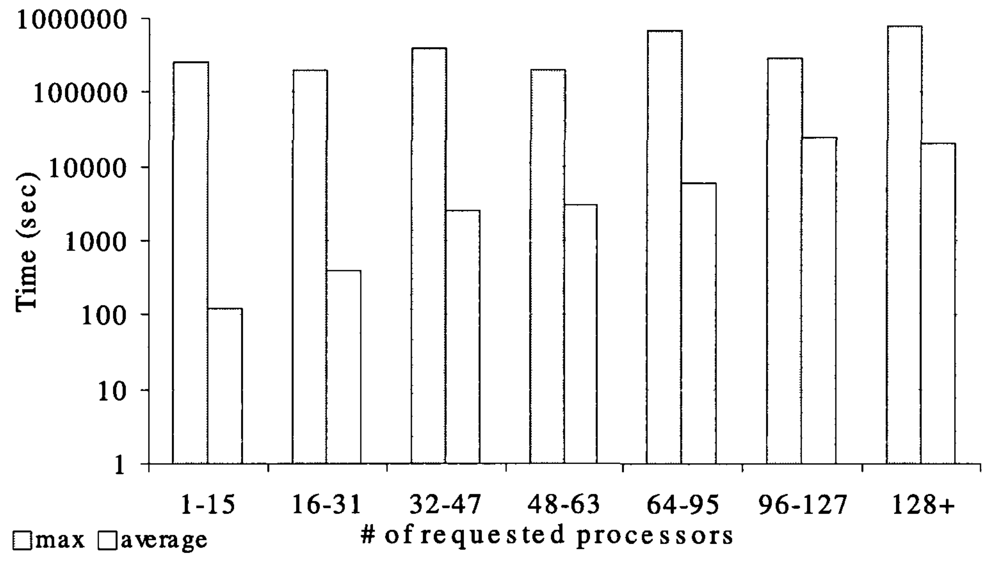


**Figure 1.1**: The wait-in-queue time statistics for parallel jobs collected from the last four and a half years from a 300+ processor cluster at the College of William and Mary.

for fewer nodes. This, in turn, results in less aggregate memory. This thesis is aimed at solving this problem by enabling computing larger problems by using less memory than would be required otherwise. In turn, this can potentially lead to shorter waiting times and even shorter overall times, also known as wall-clock times. Because we use computing resources effectively (i.e., fewer nodes and shorter wall-clock time) we call our approach effective computing.

We will focus on parallel mesh generation since we have access to experts and readily available state of the art software in that area, but our research should be applicable to much broader range of scientific applications. Thus, our goal it to make possible generating large meshes on machines with limited memory, including both shared memory workstations with multiple processors and/or processing cores and small affordable CoWs. Our solution is to store most of the mesh out-of-core (OoC) with only small portion that we work on in memory.

First, we present several out-of-core parallel mesh generation algorithms that are based on our research in parallel in-core mesh generation. These algorithms required substantial amount of time and effort to enable out-of-core computing, as well as to optimize them to an acceptable level of performance. Since algorithms need to be re-structured to for out-of-core computing the bulk of changes are specific to each algorithm and cannot be easily reused.

Next, we designed an out-of-core layer to be used with a run-time system and customized an existing application for this run-time system to support out-of-core computing. While it still required substantial amount of customization to port and optimize the application we could reuse the out-of-core layer.

Finally, we designed and implemented the Multi-layered Runtime System (MRTS), which permits out-of-core computing with any application designed for or ported to this runtime system with virtually no changes to the application code. Note, to achieve optimal performance some modification to the application code may still be required. Nevertheless, the changes will be small compared to adding out-of-core support from scratch.

We have evaluated the out-of-core layer with Out-of-core Parallel Constrained Delaunay Mesh Generation (OPCMD) which is an out-of-core version of the Parallel Constrained Delaunay Mesh Generation (PCDM) [12]. Because OPCDM uses the same programming model, the porting process was simple and few changes had to be made to the application code. The Out-of-core PCDM (OPCDM) is not limited in problem size and its performance is comparable to the original PCDM for small to medium mesh sizes. The OPCDM can be used as an effective alternative to PCDM, for very large meshes that require high number of processors for their aggregate physical memory. In such cases the difference in waiting for larger number of processors can often be quite substantial, and the wait time can be much higher than the additional overhead cost one pays for running PCDM in an out-of-core mode. Also, we evaluated MRTS with Out-of-core Non-Uniform Parallel Delaunay Refinement (ONUPDR) which is an out-of-core version of the Non-Uniform Parallel Delaunay Refinement (NUPDR). NUPDR uses a different programming model than MRTS and therefore the porting was more challenging. However, once the application was ported and able to compute in-core, adding out-of-core support was very straightforward.

In summary, we presented an approach for *effective* computing of large irregular scientific problems such as unstructured mesh generation. We showed that out-of-core computing allows solving larger than otherwise possible problems as well as getting the results faster on shared computing resources. We designed, implemented and evaluated the MRTS, which permits out-of-core computing with many application by simply porting an existing or developing a new applications for the MRTS. While porting and development are greatly simplified performance is not sacrificed.

The main contributions of this thesis are presented in:

- Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. Effective out-of-core parallel Delaunay mesh refinement using off-the-shelf software. ACM Journal of Experimental Algorithmics. In print.

- Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. The evaluation of an effective out-of-core run-time system in the context of parallel mesh generation. In IEEE International Parallel and Distributed Processing Symposium, 2011. To appear.

- Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. Parallel out-of-core Delaunay refinement. In IEEE Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Sophia, Bulgaria, September 2005.

## 1.1 Related work

### 1.1.1 Out-of-core computing

There are two basic approaches for out-of-core computing: implicit, which 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 at no more than twice the amount of RAM[1]).

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

In contrast, 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 [20, 42]. 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 to employ vendor-supplied libraries for asynchronous disk I/O. They rely on high performance in-core subroutines of BLAS [23], LAPACK [22] and ScaLAPACK [19] and a simple non-recursive (in most cases) pipeline to hide latencies associated with disk accesses.

Extensive research was performed on designing optimal algorithms for a parallel multi-level memory model [1, 39, 45–47] as well as designing methods to map existing in-core algorithms, based on batch-synchronous parallel models, into efficient out-of-core algorithms [21].

Salmon et al. described [40] an out-of-core N-body parallel method which is irregular and does not involve creation or deletion of new bodies during the execution, unlike the parallel mesh refinement computation. The authors 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 [40] 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 [43] is an out-of-core algorithm-specific approach for sequential mesh generation. The novelty of Etree is in its 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 associated with generating 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 is exceptionally fast, especially after additional improvements using a two-level bucket sort algorithm [44]. However, it targets octree-based meshes and is not yet parallel.

### 1.1.2 Run-time systems

Charm++[30] support global address space by providing directory service, message deliveries and migration of *chares*. A chare is a collection of data, similar to an object from object-oriented programming. Each chare can have a number of entry methods (again, similar to class methods). To invoke a method on a chare a message is sent from another chare. A program consists of a collection of chares and progresses by the exchange of messages between chares (with subsequent methods executions), by the creation of new chares and by the destruction of existing ones. It should be noted that the asynchronous entry method invocation is the only method of communication for a Charm++ application. The system is load-balanced by “off-loading” chares with the highest loads to the least loaded processors.

Chaos++ [9] is a runtime library extending functionality of Chaos by supporting distributed arrays and distributed pointer-addressable data structures. Two base classes are provided: mobile object and globally addressable object. Contrary to globally addressable objects, the content of mobile objects is accessible by remote processors. Every object is owned by a single processor, and shadow copies are maintained by all other processors accessing remotely. Similar to Chaos++ and independently developed ABC++[4] supports mobile objects and allows for migration of objects between nodes, and communication with a *home node* is required to find the object once it has migrated.

There are also languages that are designed to achieve the same goal. Emerald [29] is a specialized language that relies on its own compiler and preprocessor to allow for transparent accesses to remote objects. Amber [10] is a dialect of C++ where each object is assigned globally unique address space which permits seamless migration between nodes.

The Portable Runtime Environment for Mobile Applications (PREMA) [7] framework has been created to support development of adaptive and irregular applications like parallel mesh refinement. It has been demonstrated that PREMA has a number of advantages over similar systems [7] while simplifying application development. PREMA consists of a communication layer and the Implicit Load Balancing Library (ILB) [7]. There were several implementations of the communication layer; the latest and current version is Clam [25]. Clam is a runtime system designed to be used for development of irregular and adaptive applications. It provides one-sided communication and Remote Service Request (RSR) functionality as well as global address space and management of so called "mobile objects". A Clam mobile object is a user-defined data structure that is referenced by a *mobile pointer* anywhere in the system regardless of its location.

A mobile object can *migrate* to another processor without the necessity of updating mobile pointers that point to that object[2]. Additionally, RSR handlers called *mobile object handlers* can be invoked on a node where the mobile object is currently located (a local pointer to the object is passed to a handler upon execution). The application does not need to know where the object is located to post a RSR; Clam routes it to the processor where the object is located and can postpone it if the object is in the process of migration.

The ILB uses the mobile object concept provided by Clam to implement schedulable objects. These are the smallest units of work managed by the ILB and can be moved between processors to counter the imbalance.

In contrast to existing runtime systems the MRTS presented in this thesis provides support for out-of-core computing and interfaces to both fine- and coarse-grain parallelism. Enabling out-of-core computing with the MRTS is relatively straightforward and for many applications only minor changes are required. Similarly to the systems presented above the MRTS provides one-sided communication and RSR functionality, global address space and management of user defined mobile objects.

[2]The application is responsible for the actual movement of the data-structure, Clam only provides procedures to uninstall the mobile object on one processor and then install it on another.

# Chapter 2

# Out-of-core Parallel Delaunay Refinement

## 2.1 Parallel Delaunay Refinement Method

The Parallel Delaunay Refinement (PDR) algorithm is based on a theoretical framework for constructing guaranteed quality Delaunay meshes in parallel [15, 16]. Sequential guaranteed quality Delaunay Refinement algorithms insert points at the selection disks around circumcenters of triangles [14] 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, the authors presented in [15] 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 they impose a $n$-dimensional hypercube lattice[1] over the entire $n$-dimensional domain.

In this thesis we present an out-of-core version of the algorithm which we published in [31]. For simplicity we begin by presenting the algorithm in one dimension[2]. 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: 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 $\beta$ 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 $\overline{\partial\mathcal{M}_{i,j}}$ inside $\mathcal{M}_i$ as $\overline{\partial\mathcal{M}'_{i,j}}$. Figure 2.1 shows the subdivision[3] of $\mathcal{M}$.

[1]The points pattern of the lattice is equivalent to that of a $n$-dimensional hypercube vertices
[2]In one dimension a "triangulation" of a segment is a discretization of the segment.
[3]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 $(\overline{\partial\mathcal{M}_{i,i-1}} \cup \overline{\partial\mathcal{M}'_{i,i-1}} \cup \overline{\partial\mathcal{M}'_{i,i+1}} \cup \overline{\partial\mathcal{M}_{i,i+1}}) \subseteq \mathcal{M}_i$.

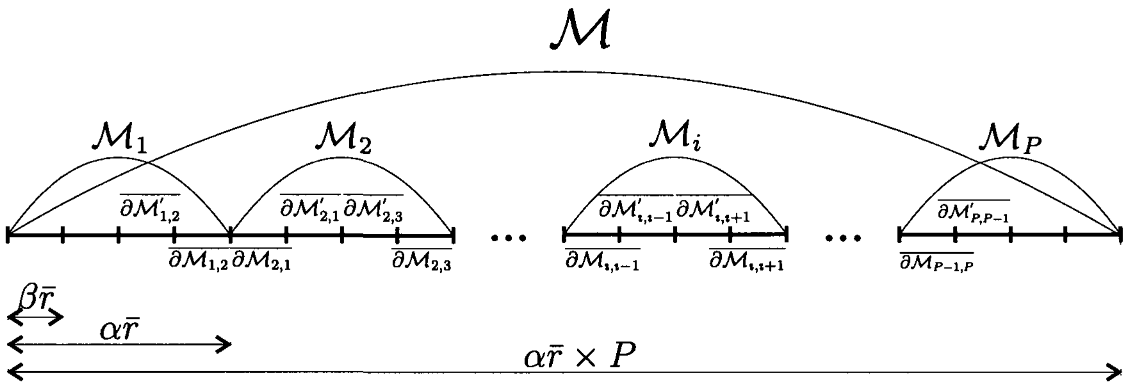

$\mathcal{M}$
$\mathcal{M}_1$
$\mathcal{M}_2$
$\mathcal{M}_i$
$\mathcal{M}_P$
$\overline{\partial\mathcal{M}'_{1,2}}$
$\overline{\partial\mathcal{M}'_{2,1}}\overline{\partial\mathcal{M}'_{2,3}}$
$\overline{\partial\mathcal{M}'_{i,i-1}}\overline{\partial\mathcal{M}'_{i,i+1}}$
$\overline{\partial\mathcal{M}'_{P,P-1}}$
$\overline{\partial\mathcal{M}_{1,2}}\overline{\partial\mathcal{M}_{2,1}}$
$\overline{\partial\mathcal{M}_{2,3}}$
$\overline{\partial\mathcal{M}_{i,i-1}}$
$\overline{\partial\mathcal{M}_{i,i+1}}$
$\overline{\partial\mathcal{M}_{P-1,P}}$
$\beta\bar{r}$
$\alpha\bar{r}$
$\alpha\bar{r} \times P$

**Figure 2.1**: Subdivision of a mesh $\mathcal{M}$.

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

**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 $A \bigtriangleup B$ and are geometrically within $\gamma$ ($\gamma$ 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** $\Big(\mathcal{M}_i, (\mathcal{M}_i \setminus (\overline{\partial\mathcal{M}'_{i,i+1}} \cup \overline{\partial\mathcal{M}_{i-1,i}}))\Big)$

   **refine** $\Big(\mathcal{M}_1, (\mathcal{M}_1 \setminus (\overline{\partial\mathcal{M}'_{1,2}} \cup \overline{\partial\mathcal{M}_{1,2}}))\Big)$

   **refine** $\Big(\mathcal{M}_P, (\overline{\partial\mathcal{M}_{P-1,P}} \cup \overline{\partial\mathcal{M}'_{P,P-1}})\Big)$

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** $\Big(\mathcal{M}_i, (\mathcal{M}_i \setminus (\overline{\partial\mathcal{M}'_{i,i-1}} \cup \overline{\partial\mathcal{M}_{i+1,i}}))\Big)$

   **refine** $\Big(\mathcal{M}_1, (\overline{\partial\mathcal{M}'_{1,2}} \cup \overline{\partial\mathcal{M}_{1,2}})\Big)$

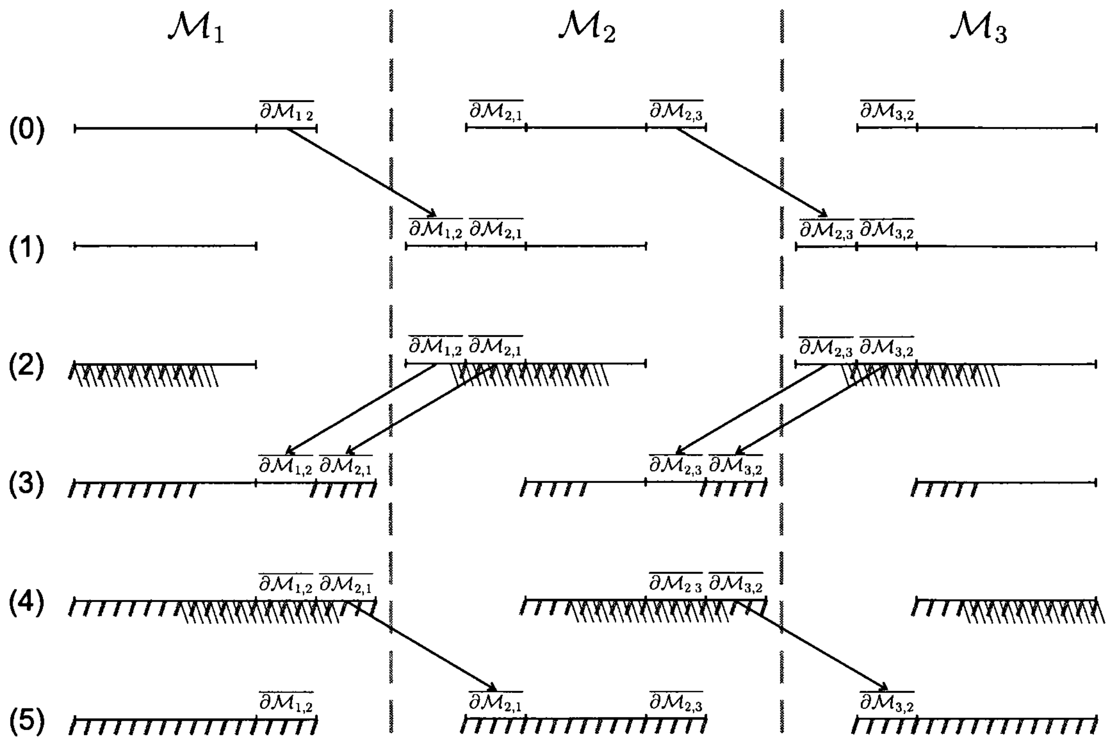
   **refine** $\Big(\mathcal{M}_n, (\mathcal{M}_n \setminus \overline{\partial\mathcal{M}'_{P,P-1}})\Big)$

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 2.2 for an example of algorithm execution with the mesh partitioned between three subdomains. In step 0, the 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



$\mathcal{M}_1$
$\mathcal{M}_2$
$\mathcal{M}_3$
(0)
(1)
(2)
(3)
(4)
(5)
$\overline{\partial\mathcal{M}_{1,2}}$
$\overline{\partial\mathcal{M}_{2,1}}$
$\overline{\partial\mathcal{M}_{2,3}}$
$\overline{\partial\mathcal{M}_{3,2}}$

**Figure 2.2**: 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.

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.

### 2.1.1 Shared memory implementation of the PDR

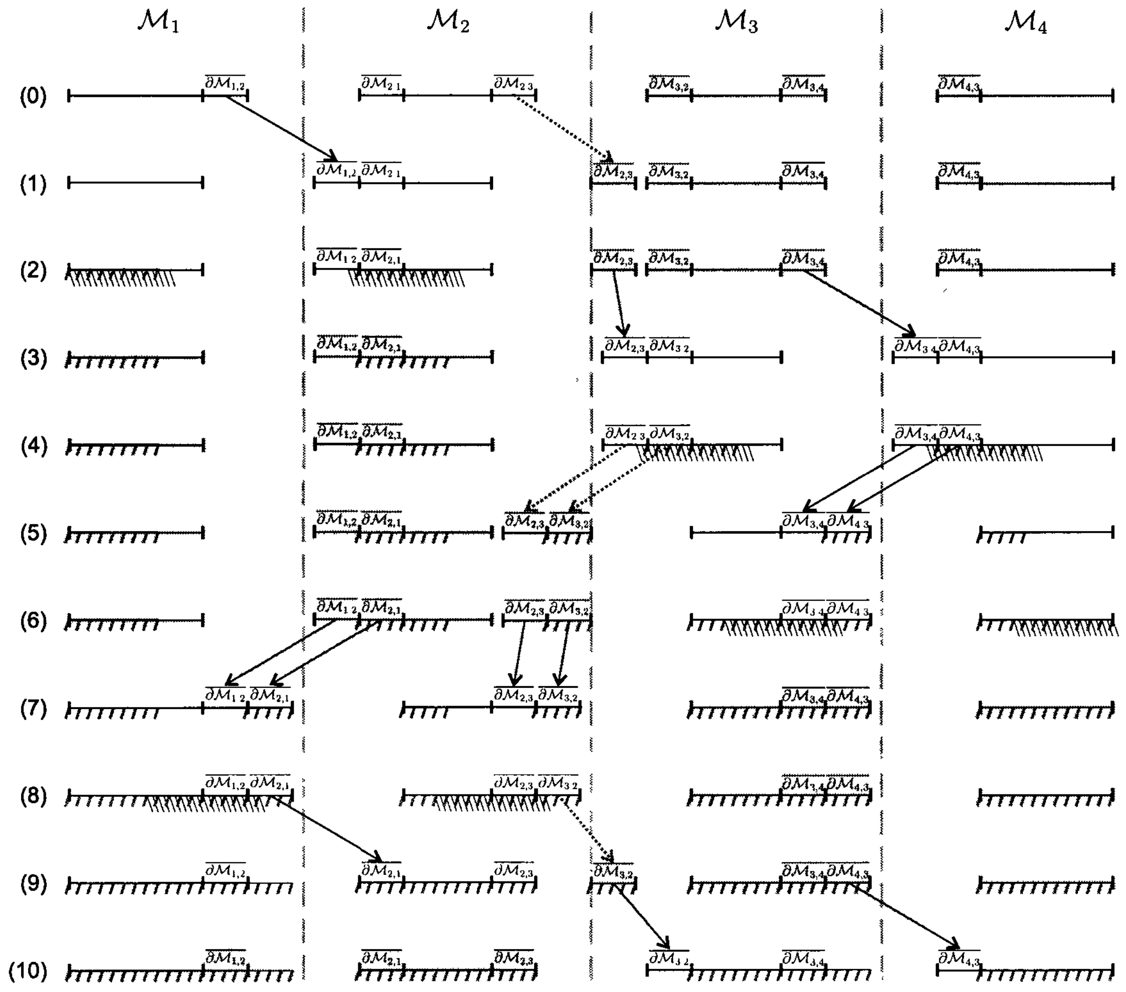
The original implementation of the PDR was for distributed memory computing. However, since multi-core (including support for hardware threads) is becoming increasingly popular we implemented a modified algorithm to take advantage of shared resources and to avoid unnecessary communication:

- 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

Our evaluation showed [32] 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 2.2.3).

## 2.2 Out-of-core PDR

See Figure 2.3 for an example of the algorithm execution with the mesh partitioned into four subdomains with only room for two in memory.



$\mathcal{M}_1$
$\mathcal{M}_2$
$\mathcal{M}_3$
$\mathcal{M}_4$
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
$\overline{\partial\mathcal{M}_{1,2}}$
$\overline{\partial\mathcal{M}_{2,1}}$
$\overline{\partial\mathcal{M}_{2,3}}$
$\overline{\partial\mathcal{M}_{3,2}}$
$\overline{\partial\mathcal{M}_{3,4}}$
$\overline{\partial\mathcal{M}_{4,3}}$

**Figure 2.3**: 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.

### 2.2.1 Out-of-Core Shared memory PDR

The Out-of-Core Shared memory PDR (OSPDR) algorithm is designed to create large meshes in parallel, using only one workstation or a single node of a cluster 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 similar to the one utilized in the PDR [15]. 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., when we say subdomain $i$ sends data to subdomain $j$, and subdomain $i$ is loaded into the memory of processor $m$ while subdomain $j$ is loaded into the memory of processor $n$, if fact processor $m$ sends data to processor $n$).

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[4] 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 [15].

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($\mathcal{M}$, $\bar{\Delta}$, $\bar{\rho}$, $P$, $p$, $N$)
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
       // $1 \le row(p), col(p) \le \sqrt{P}$
1  for $m = 1, \quad , N$
2      for $n = 1, \quad , N$
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 $\{c_{i,1} \mid 1 \le i \le 4\}$ of local mesh $\mathcal{M}_p$
6          endif
7          $\mathcal{M}_p \leftarrow PDR_{refinement}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, P, p)$
```

[4]With the exception of the boundary subdomains

```
8           $\mathcal{M}_p \leftarrow PDR_{shifts}(\mathcal{M}_p,\ \bar{\Delta},\ \bar{\rho},\ P,\ p)$
9           if $col(p) = \sqrt{P}$ and $n \neq N$
10              Assign cells $\{c_{i,4} \mid 1 \leq i \leq 4\}$ 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($\mathcal{M},\ \bar{\Delta},\ \bar{\rho},\ P,\ p,\ N$)
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
        // $1 \leq row(i), col(i) \leq \sqrt{P}$
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 $\{c_{i,1} \mid 1 \leq i \leq 3\}$ of local mesh $\mathcal{M}_p$
6           endif
7           $\mathcal{M}_p \leftarrow PDR_{refinement}(\mathcal{M}_p,\ \bar{\Delta},\ \bar{\rho},\ P,\ p)$
8           $\mathcal{M}_p \leftarrow PDR_{shifts}(\mathcal{M}_p,\ \bar{\Delta},\ \bar{\rho},\ P,\ p)$
9           if $col(p) = \sqrt{P}$ and $n \neq N$
10              Assign cells $\{c_{i,4} \mid 1 \leq i \leq 3\}$ to processor in $(row(p), 1)$
11          endif
12          if $row(p) = \sqrt{P}$ and $m \neq N$
13              Assign cells $\{c_{4,i} \mid 1 \leq i \leq 3\}$ 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 $(1, 1)$
17          endif
18          if $row(p) = 1$ and $m < N$
19              Reference cells $\{b_{1,i} \mid 1 \leq i \leq 3\}$ of local buffer $B$
20              Overwrite cells $\{c_{1,i} \mid 1 \leq i \leq 3\}$ of block $p$
                    in subdomain $m \times N + n$ with the content of $B$
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 $m \times N + n + 1$ with the content of $B$
25          endif
26          Store local mesh $\mathcal{M}_p$ as block $p$ of subdomain $(m-1) \times N + n$
27      endfor
28  endfor
29  return $\mathcal{M}_p$
```

## 2.2.2 Out-of-core Distributed Memory PDR

Similarly to the OSPDR the Out-of-core Distributed memory PDR (ODPDR) algorithm is designed to create very large meshes in parallel. Unlike the OSPDR the ODPDR is designed to use multiple nodes of a CoW and exploit the aggregate and concurrent access to disk space. 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 the same as in the OSPDR: to minimize the number of accesses and overlap them with computation whenever possible.

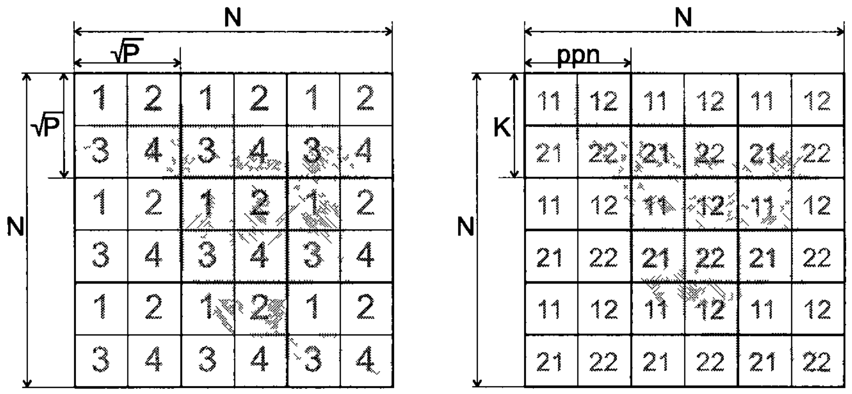


**Figure 2.4**: An example of domain partitioning for the ODPDR (left) and the OHPDR (right) methods. $P$ is the number of processors in 1 processor/core per node scenario, $ppn$ is the number of processors per node, $K$ is the number of nodes. $N$ is derived empirically and depends on amount of memory and disk space ($N^2$ is the total number of subdomains).

Again, 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 relies on the PDR (in-core) parallel Delaunay meshing and refinement code, but uses a different assignment of cells to processors than the PDR.

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 2.4, 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 the RAM of a single node. The total number of blocks in the domain is $P \times N^2$; each processor stores (on the local disk) one block from each subdomain, having a total of $N^2$ blocks. This scattered decomposition helps to implicitly improve workload imbalances. Similarly to the way we described OSPDR we will only explain one horizontal and one diagonal shift.

The horizontal/vertical type of top-level shift is rather straightforward: the order of refinement coincides with the direction of the shift (see Figure 2.5):

```
ODPDR.HorizontalShift($\mathcal{M}$, $\bar{\Delta}$, $\bar{\rho}$, $P$, $p$, $N$)
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
        // $1 \le row(i), col(i) \le \sqrt{P},\ 1 \le i \le P$
1   for $m = 1, \dots, N$
2       for $n = 1, \dots, N$
3           Load block $p$ of subdomain $(m-1) \times N + n$ as local mesh $\mathcal{M}_p$
4           if $n \ne 0$ and $col(p) = 1$
5               Receive cells $\{c_{i,1} \mid 1 \le i \le 4\}$ of local mesh $\mathcal{M}_p$
6           endif
7           $\mathcal{M}_p \leftarrow PDR_{refinement}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, P, p)$
```

```
8            $\mathcal{M}_p \leftarrow PDR_{shifts}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, P, p)$
9            if $col(p) = \sqrt{P}$ and $n \neq N$
10               Send cells $\{c_{i,4} \mid 1 \leq i \leq 4\}$ 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 $\mathcal{M}_p$
```

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 2.5):

```
ODPDR.DiagonalShift($\mathcal{M}$, $\bar{\Delta}$, $\bar{\rho}$, $P$, $p$, $N$)
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
         // $1 \leq row(i), col(i) \leq \sqrt{P},\ 1 \leq i \leq P$
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                Receive cells $\{c_{i,1} \mid 1 \leq i \leq 3\}$ of local mesh $\mathcal{M}_p$
6            endif
7            $\mathcal{M}_p \leftarrow PDR_{refinement}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, P, p)$
8            $\mathcal{M}_p \leftarrow PDR_{shifts}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, P, p)$
9            if $col(p) = \sqrt{P}$ and $n \neq N$
10               Send cells $\{c_{i,4} \mid 1 \leq i \leq 3\}$ to processor in $(row(p), 1)$
11           endif
12           if $row(p) = \sqrt{P}$ and $m \neq N$
13               Send cells $\{c_{4,i} \mid 1 \leq i \leq 3\}$ to processor in $(1, col(p))$
14           endif
15           if $p = P$ and $n \neq N$ and $m \neq N$
16               Send cell $c_{4,4}$ to processor in $(1, 1)$
17           endif
18           if $row(p) = 1$ and $m < N$
19               Receive cells $\{b_{1,i} \mid 1 \leq i \leq 3\}$ of local buffer $B$
20               Overwrite cells $\{c_{1,i} \mid 1 \leq i \leq 3\}$ of block $p$ in
                     in subdomain $m \times N + n$ with the content of $B$
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 the content of $B$
25           endif
26           Store local mesh $\mathcal{M}_p$ as block $p$ of subdomain $(m-1) \times N + n$
27       endfor
28   endfor
29   return $\mathcal{M}_p$
```

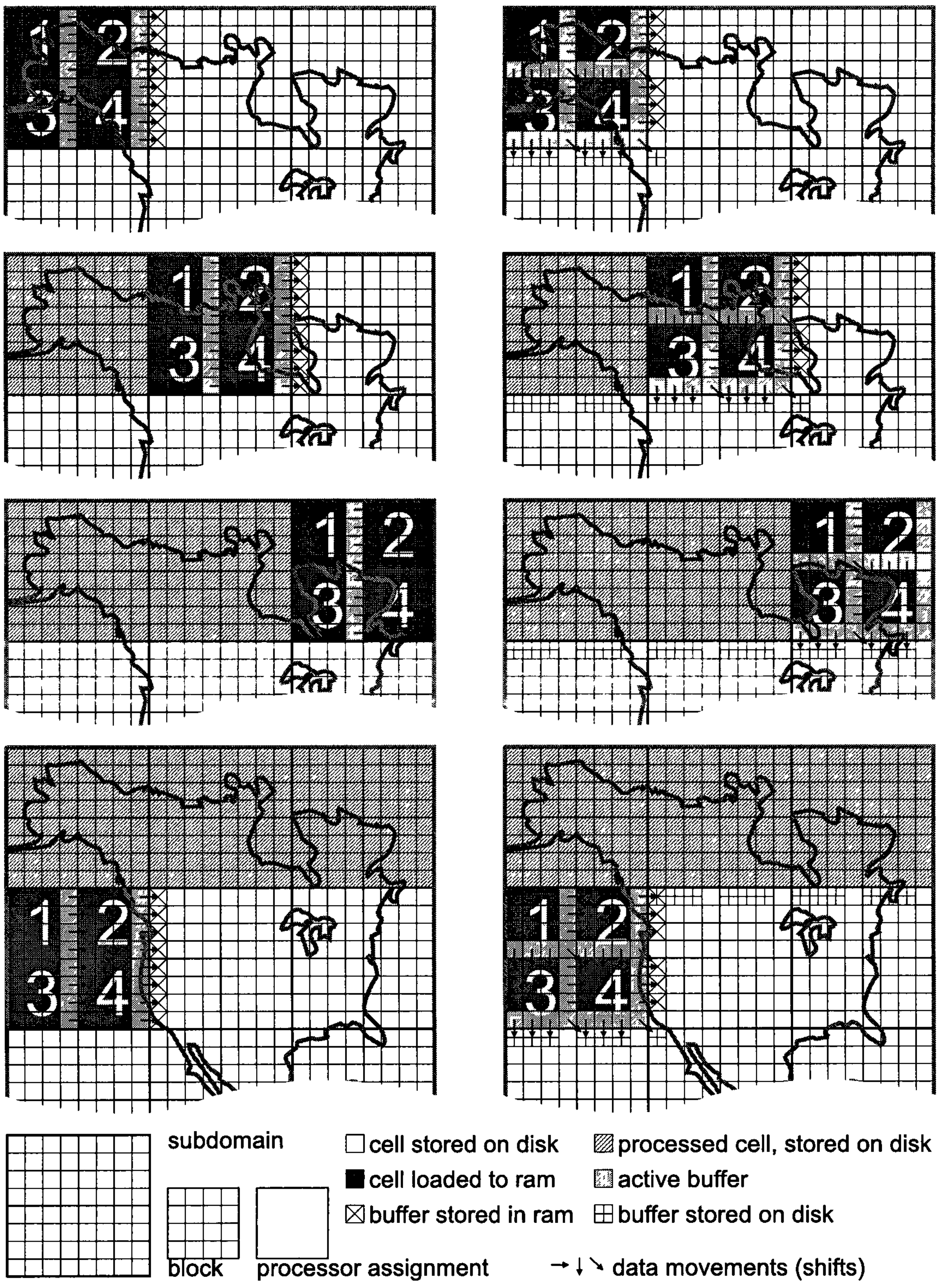


**Figure 2.5**: Out-of-core schemes of top-level shifts for ODPDR: along axis (left) and diagonal (right). Input geometry is the outline of North American continent. Setup: 4 processors, 9 subdomains, distributed memory and disk storage.

### 2.2.3 Out-of-core Hybrid Memory PDR

To take full advantage of the current hardware trend of having multiple processors / cores per node, we designed and implemented the Out-of-core Hybrid memory PDR (OH-PDR). Indeed, our experimental study (see Section 5.2) 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.

Again, 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 2.4, 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 2.6):

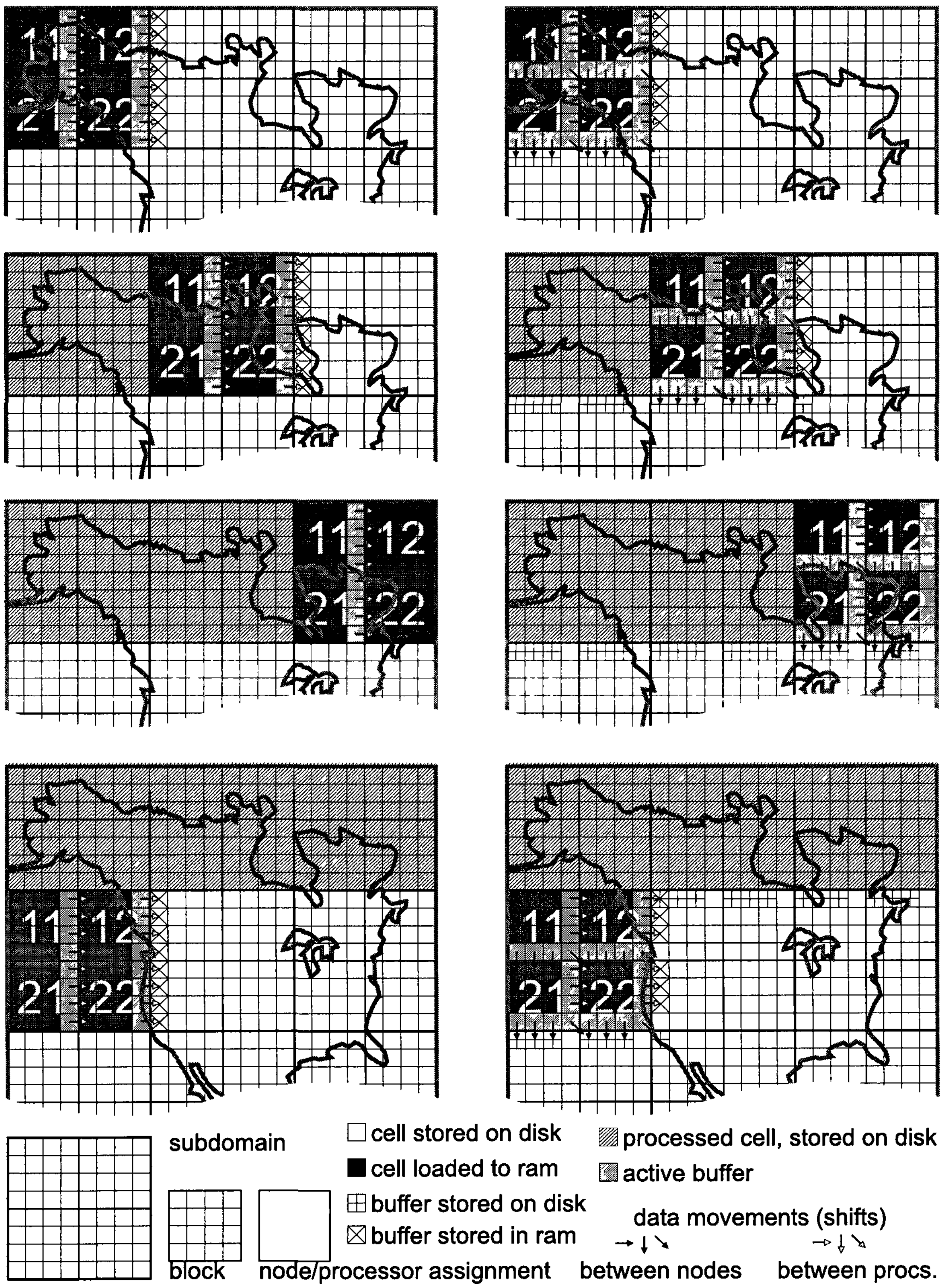


**Figure 2.6**: Out-of-core schemes of top-level shifts for OHPDR: along axis (left) and diagonal (right). Input geometry is the outline of North American continent. Setup: 2 nodes, 2 processors with shared memory per node, 9 subdomains, disk storage.

```
OHPDR.HorizontalShift($\mathcal{M}$, $\bar{\Delta}$, $\bar{\rho}$, $K$, $ppn$, $p$, $N$)
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 $\sqrt{K * ppn}$ is integer and,
        for simplicity of the presentation, $K = ppn$)
        $p$ is the index of the current processor, $1 \le p \le ppn \times K$
        $\mathcal{M}$, $\mathcal{X}$, $\bar{\Delta}$, $\bar{\rho}$ and $N$ are the same as in ODPDR.HorizontalShift
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 $node(p)$ and $proc(p)$ of the current processor
        // $1 \le node(i) \le K$, $1 \le proc(i) \le ppn$, $1 \le i \le ppn \times K$
 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 $proc(p) = 1$
 5              Read cells $\{c_{i,1} \mid 1 \le i \le 4\}$ of local mesh $\mathcal{M}_p$ from shared-memory buffer
 6          endif
 7          $\mathcal{M}_p \leftarrow SPDR_{refinement}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, ppn, K, p)$
 8          $\mathcal{M}_p \leftarrow SPDR_{shifts}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, ppn, K\ p)$
 9          if $proc(p) = ppn$ and $n \neq N$
10              Write cells $\{c_{i,4} \mid 1 \le i \le 4\}$ into shared-memory buffer
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 $\mathcal{M}_p$
```

The top-level diagonal shift to the right and down is performed in the following steps (see Figure 2.6):

```
OHPDR.DiagonalShift($\mathcal{M}$, $\bar{\Delta}$, $\bar{\rho}$, $K$, $ppn$, $p$, $N$)
Input: same as in OHPDR.HorizontalShift
Output: a (partially) refined Delaunay mesh $\mathcal{M}_p$ which conforms to $\mathcal{X}$
 0  Calculate $node(p)$ and $proc(p)$ of the current processor
        // $1 \le node(i) \le K$, $1 \le proc(i) \le ppn$, $1 \le i \le ppn \times K$
 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 $proc(p) = 1$
 5              Read cells $\{c_{i,1} \mid 1 \le i \le 3\}$ of local mesh $\mathcal{M}_p$ from shared-memory buffer
 6          endif
 7          $\mathcal{M}_p \leftarrow SPDR_{refinement}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, ppn, K, p)$
 8          $\mathcal{M}_p \leftarrow SPDR_{shifts}(\mathcal{M}_p, \bar{\Delta}, \bar{\rho}, ppn, K, p)$
 9          if $proc(p) = ppn$ and $n \neq N$
10              Write cells $\{c_{i,4} \mid 1 \le i \le 3\}$ into shared-memory buffer
11          endif
12          if $node(p) = K$ and $m \neq N$
13              Send cells $\{c_{4,i} \mid 1 \le i \le 3\}$ to node $node(p)$
14          endif
15          if $proc(p) = ppn$ and $node(p) = K$ and $n \neq N$ and $m \neq N$
16              Send cell $c_{4,4}$ to node 1
```

```
        endif
        if $node(p) = 1$ and $m < N$
            Receive cells $\{b_{1,i} \mid 1 \leq i \leq 3\}$ of local buffer $B$
            Overwrite cells $\{c_{1,i} \mid 1 \leq i \leq 3\}$ of block $p$ in
                in subdomain $m \times N + n$ with the content of $B$
        endif
        if $n < N$ and $m < N$
            Receive cell $b_{1,1}$ of local buffer $B$
            Overwrite cell $c_{1,1}$ of block $p$ in subdomain $m \times N + n + 1$ with the content of $B$
        endif
        Store local mesh $\mathcal{M}_p$ as block $p$ of subdomain $(m - 1) \times N + n$
    endfor
endfor
return $\mathcal{M}_p$
```

# Chapter 3

# Out-of-core Parallel Constraint Delaunay Meshing

## 3.1 Parallel Constrained Delaunay Meshing

In this thesis we present an out-of-core version of the Parallel Constrained Delaunay Meshing (PCDM) [12] which we presented in [33].

The mesh generation procedure starts with constructing an initial mesh which conforms to the input vertices and segments, and then refines this mesh until the constraints on triangle quality and size are met. The general idea of the Delaunay refinement is to insert points inside the circumscribed circles[1] of triangles that violate the required bounds, until there are no such triangles left. To update the triangulation, the Bowyer/Watson algorithm [8, 50] is used, which is based on deleting the triangles that are no longer Delaunay and inserting new triangles that satisfy the Delaunay property.

[1]Traditionally circumcenters are used. However, as the authors have shown [11] there exist entire regions for the selection of new points.

The set of triangles in the mesh whose circumcircles include the newly inserted point $p_i$ is called a *cavity* [26], and is denoted as $\mathcal{C}(p_i)$. Also, the symbol $\partial\mathcal{C}(p_i)$ stands for the set of edges which belong to only one triangle in $\mathcal{C}(p_i)$, i.e., external edges.

In the absence of external boundaries, the algorithm maintains a Delaunay mesh $\mathcal{M}$, and at every iteration performs the following steps:

1. Select a triangle from the queue of unsatisfactory triangles.
2. Compute the circumcenter $p_i$ of this triangle.
3. Find $\mathcal{C}(p_i)$ and $\partial\mathcal{C}(p_i)$.
4. Delete all triangles in $\mathcal{C}(p_i)$ from $\mathcal{M}$.
5. Add triangles obtained by connecting $p_i$ with every edge in $\partial\mathcal{C}(p_i)$ to $\mathcal{M}$.

The case when the new point happens to be close to a constrained edge is treated separately. Following Shewchuk [41], diametral lenses are used to detect if a segment is encroached upon. The *diametral lenses* of a segment is the intersection of two disks, whose centers lie on the opposite sides of the segment on each others boundaries, and whose boundaries intersect in the endpoints of the segment. A segment is said to be *encroached upon* by point $p_i$ if $p_i$ lies inside its diametral lenses. When a point selected for insertion is found out to encroach upon a segment, another point is inserted in the middle of the segment instead.

To refine the mesh in parallel, a coarse grained domain decomposition obtained with the meDDec [36] library is used. The goal is to distribute the subdomains among processors, so that the sums of the weights of the subdomains on each processors are approximately equal, and the total length of the subdomain boundaries which are shared between processors is minimized. Figure 3.1[2] shows an example of a human brain domain decomposition. During runtime, the Load Balancing Library [5] maintains the equidistribution and small edgecut conditions by moving the subdomains among the processors in response to dynamically changing work load imbalance.

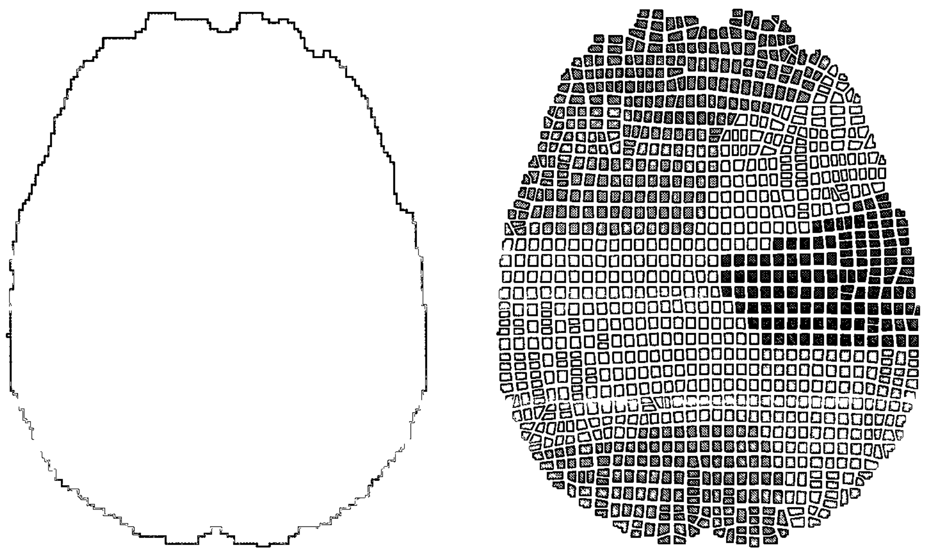


**Figure 3.1**: Decomposition of a human brain into 1024 subdomains mapped onto 8 processors.

The domain decomposition procedure described above creates $N$ subdomains, each of which is bounded by edges of the initial domain decomposition. The edges and their endpoints that are shared between two subdomains are duplicated. The boundary edges are treated as constrained segments, and whenever they are split due to encroachment on one processor, an active message [25, 49] is sent to the mobile object holding the adjacent subdomain, so that the duplicate of the boundary edge is also split, and the mesh is globally consistent (see Figure 3.2).

[2]Courtesy of Andrey Chernikov.

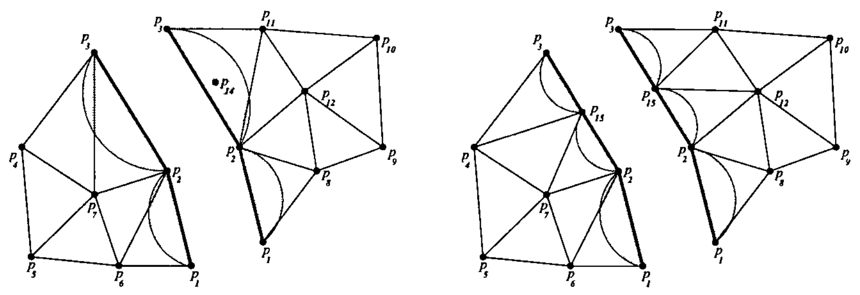


**Figure 3.2**: Splitting an edge.

## 3.2 Programming model

First step to enable out-of-core computing for PCDM is to port it to Portable Runtime Environment for Mobile Applications (PREMA) [7] framework. PREMA has been created to support development of adaptive and irregular applications like parallel mesh refinement. It has been demonstrated that PREMA has a number of advantages over similar systems while simplifying application development.

The PREMA programming model is centered around a *mobile object* concept. A mobile object is an application user defined structure and it is not restricted to continuous memory. A mobile object can be referenced by any processor via a unique global *mobile pointer*. PREMA is designed for data centric computation where most of communication happens between mobile objects rather than between processors. The communication is handled by *message* operation. A message is an asynchronous remote request call that is routed to the location of the target mobile object. Since a mobile object can move between processors the request is sent to the last known location and then routed to the correct destination. Upon arrival a user defined handler function is called passing target mobile object and arguments from the message.

PREMA encourages *overdecomposition* that is the problem is broken into $N$ subproblems and $N \gg P$, where $P$ is the number of processors. Originally, high degree of overdecomposition was required to allow greater flexibility to the Implicit Load Balancing (ILB) library. However, it is also important in out-of-core computing – the more subproblems can fit in-core the greater is the flexibility of paging.

Thus the computation of the usual PREMA application consists of operations on mobile objects. When communication is necessary message operation is used regardless of the location of the target mobile object. Common iterative accesses in this model should be replaced by sending messages to corresponding mobile objects.

## 3.3 Out-of-core subsystem for the PCDM

The smallest chunk of data that can be stored out-of-core is a mobile object. We developed a light-weight out-of-core layer to facilitate loading and storing objects as well as determining the status (in-core or out-of-core) and to assist the algorithm in making decisions on what needs to be loaded/stored at any given time.

This layer provides functionality of a high level cache to local disk. For every node, it maintains a directory of local objects currently in memory or stored on disk as well as history of accesses. When on disk the objects can be stored as a collection of files (one object per file) or as a single file. Storing all objects in one file eliminates the overhead of the file system[3] but requires extra data structures and code to index the objects. In the single file case there is an option to pad to accommodate growing objects. We also support raw access to a disk partition but this solution is not very practical due to

[3]Unless we can directly write to disk device (rarely an option on public resources) the overhead savings are insignificant.

access limitation on public resources.

Upon request to load an object it is loaded into a buffer, callbacks to unpack the object are executed and its pointer returned. Often, the object will be in buffer, then no loading is necessary. Regardless whether the object was in buffer or not, the request is blocked until the object is ready.

When there is a request to store an object, the state of the buffer is evaluated. Then, based on the amount of available space in the buffer, access statistics, and priorities, the object might be unloaded to disk or kept in the buffer. The priority of the object is lowered in the latter case. The requests to store an object are always non-blocking.

Additionally, on every object store request, a state of cache is evaluated and, based on the access statistics, some objects may be scheduled to move to disk or prefetched. For these decisions, we use prioritized least recently used page replacement policy with priority information passed from higher layers.

## 3.4 Implementation

Implementation is done in two steps: first, we port the PCDM to PREMA without adding any new functionality; second, we make changes and optimize the application to enable and improve its out-of-core performance.

We start by registering the subdomains as mobile objects, then we replace all communication between processors with communication between subdomains. We also remove some of the code that keeps track of the remote subdomain and decides which subdomain should receive a split request when it arrives. With the exception of few minor changes, this constitutes the porting process.

We modify the main loop to access local mobile object through out-of-core subsystem rather than directly. This ensures that objects are in fact in-core when the application tries to access them. In case an object is out-of-core, the application call will be blocked until the object is loaded. At this stage the application already can run out-of-core but the performance is not very good. The main reason for poor performance is the order in which each processor refines the subdomains it owns. Since the original PCDM does not differentiate between in-core and out-of-core subdomains, it often tries to refine out-of-core subdomains before it refines all subdomains that are currently in-core. This adds extra overhead to migrate unrefined subdomains to disk and back.

We introduced the following changes (see Figures 3.3 and 3.4). First, before processing a subdomain in the main loop, we check whether the next subdomain in queue is in-core and: mark it as sticky if it is in-core or post a non-blocking load request for that subdomain if it is not. Second, after all bad triangles were processed for a subdomain, we check whether the next subdomain in queue is in-core. If it is not, we move it to the end of the queue and examine the next. If we cannot find an in-core subdomain we load the next subdomain in queue with a blocking call.

It should be noted that the RTS will mark subdomains with multiple incoming messages as sticky and may attempt to prefetch them. Additionally, when processing incoming messages (application is polling) the RTS first executes messages addressed to in-core subdomains regardless of the order in which messages were received (order of the messages sent to the same subdomain is preserved).

```
OPCDM($\{(\mathcal{X}_i, \mathcal{M}_i) \mid i = 1, \ldots, N\}, \bar{\Delta}, \bar{\rho}, P, p$)
Input:  $\mathcal{X}_i$ are PSLGs that define the subdomains $\Omega_i$
        $\mathcal{M}_i$ are initial coarse meshes of $\Omega_i$
        $\bar{\Delta}$ is the upper bound on triangle area
        $\bar{\rho}$ is the upper bound on triangle circumradius-to-shortest edge ratio
        $P$ is the total number of processes
        $p$ is the index of the current process
Output: Modified Delaunay meshes $\{\mathcal{M}_i\}$ which respect the bounds $\bar{\Delta}$ and $\bar{\rho}$
 1  Compute the mapping $\kappa : \{1, \ldots, N\} \rightarrow \{1, \ldots, P\}$
        of subdomains to processes
 2  Distribute subdomains to processes
 3  Let $\{\Omega_{i_1}, \ldots, \Omega_{i_{N_p}}\}$ be the set of local subdomains
*4  Let $Q$ be the set of unrefined subdomains
*5  $Q \leftarrow \{\Omega_{i_j} \mid j = 1, \ldots, N_p\}$
*6  while $Q \neq \emptyset$
*7      $\Omega_{i_j} \leftarrow$ SCHEDULE($Q$)
 8      DELAUNAYREFINEMENT($\mathcal{X}_{i_j}, \mathcal{M}_{i_j}, \bar{\Delta}, \bar{\rho}, \kappa$)
*9  endwhile
10  TERMINATE()

DELAUNAYREFINEMENT($\mathcal{X}$, $\mathcal{M}$, $\bar{\Delta}$, $\bar{\rho}$, $\kappa$)
11  $Q \leftarrow \{t \in \mathcal{M} \mid (\rho(t) \geq \bar{\rho}) \vee (\Delta(t) \geq \bar{\Delta})\}$
12  while $Q \neq \emptyset$
13      Let $t \in Q$
14      BADTRIANGLEELIMINATION($\mathcal{X}$, $\mathcal{M}$, $t$, $\kappa$)
15      Update $Q$
16  endwhile

BADTRIANGLEELIMINATION($\mathcal{X}$, $\mathcal{M}$, $t$, $\kappa$)
17  $p_i \leftarrow$ CIRCUMCENTER($t$)
18  if $p_i$ encroaches upon a segment $s \in \mathcal{X}$
19      $p_i \leftarrow$ MIDPOINT($s$)
20      REMOTESPLITREQUEST($\kappa$(NEIGHBOR($s$)), $p_i$)
21  endif
22  $\mathcal{C}(p_i) = \{t \in \mathcal{M} \mid p_i \in t\}$
23  $\mathcal{M} \leftarrow \mathcal{M} \setminus \mathcal{C}(p_i) \cup \{\triangle(p_i p_m p_n) \mid e(p_m p_n) \in \partial\mathcal{C}(p_i)\}$
```

**Figure 3.3**: A high level description of the OPCDM. This figure is based on the original PCDM algorithm figure [12] with augmented lines marked with $*$. Function SCHEDULE is explained in Figure 3.4.

```
SCHEDULE($Q$)
**Input:** $Q$ is a set of local subdomains (in-core and out-of-core)
**Output:** A subdomain $\Omega_{i_j}$ that is guaranteed to be in-core; also start preloading
$\Omega_{i_j} \leftarrow$ FINDINCORE($Q$)
**if** $\Omega_{i_j}$ = **null**
    $\Omega_{i_j} \leftarrow$ **pop** ($Q$)
    Blocking load $\Omega_{i_j}$
**endif**
$\Omega_{i_j+1} \leftarrow$ FINDINCORE($Q$)
**if** $\Omega_{i_j+1}$ = **null**
    $\Omega_{i_j+1} \leftarrow$ **pop** ($Q$)
    Non-blocking load $\Omega_{i_j+1}$
**else**
    Mark $\Omega_{i_j+1}$ as *sticky*
**endif**
**push** ($Q$, $\Omega_{i_j+1}$)
**return** $\Omega_{i_j}$
```

**Figure 3.4**: A high level description of function SCHEDULE, part of the OPCDM algorithm. FINDINCORE returns a subdomain that is currently in-core (if any).

# Chapter 4

# Multi-layered Run-Time System

To simplify and streamline the process of enabling existing codes to compute out-of-core as well as developing new out-of-core applications we designed and implemented the Multi-layered Run-Time System (MRTS) [35], a practical out-of-core runtime system that supports the execution of large scale parallel applications on a fraction of the nodes that otherwise would be normally required.

## 4.1 Requirements

The mesh generation methods we described earlier have the following common characteristics:

1. spatial locality — each PE works with a subset of mesh elements that cover a certain geometrically defined area, and most of the computation is performed on data that does not have outside dependencies;

2. although the communication patterns vary among the methods, the common property is that the amount of the data that the PEs need to exchange is substantially

smaller than the sizes of the subdomains, i.e., mesh sizes of subdomains;

3. local synchronization — changes in a subdomain usually affect only neighbors of that subdomain and global synchronization is not required;

4. irregular access pattern — it is not possible to predict the exact mesh elements and memory locations that are accessed;

5. SPMD data model — a single program is used to process portions of the dataset in parallel;

6. interoperability — to simplify the porting process we should not obstruct the MPI or any other form of communication used by the rest of the application (i.e., FE solver).

## 4.2 Background

We adopted the *mobile object* which is defined in [7] as a location-independent container implemented by the run-time system to store application data. The decision to define mobile objects is left to an application programmer, but we recommend using it for representing larger semi-isolated fragments of a dataset (e.g., subdomains). A mobile object can be freely moved by the run-time system between nodes and is globally addressable.

A *message* is an amalgamation of data transfer and a remote procedure call [48]. It is one-sided, which means the receiving node does not have to post an explicit receive and is not interrupted when a message arrives.

A *message handler* is a function defined by an application and registered with a mobile object. A message is delivered to a mobile object by invocation of a corresponding message handler on a node where the mobile object is located. Message handlers, messages and mobile objects allow encapsulation of data represented by mobile objects.

A *mobile pointer* is a global identifier and is used to reference a mobile object. Because a mobile object is not restricted to any specific node a message is addressed to the mobile pointer and the run-time system routes the message appropriately. Order of messages is preserved only between two endpoints.

In the course of out-of-core computing mobile objects can be unloaded to and reloaded from the disk. Mobile objects support *serialization*[1] by implementing serialization interfaces provided by the run-time system. A more detailed description of out-of-core objects behavior and requirements is provided in the following sections.

## 4.3 Programming Model

The programming model is centered around the mobile object concept. The run-time system is designed for data-centric computation where most communication happens between mobile objects rather than between processors. Parallelism is achieved by executing message handlers simultaneously on multiple nodes and multiple tasks within each message handler. The MRTS tries to achieve maximum utilization by executing as many tasks as are available while not oversubscribing the PEs which can lead to unnecessary context switches and performance degradation.

The usual application for the run-time system has its dataset broken into a collection of mobile objects. We encourage *overdecomposition*, that is breaking the problem into

[1]Serialization is the process of transforming the memory representation of an object to a data format suitable for storage or transmission.

$N$ subproblems and $N \gg P$, where $P$ is the number of PEs. Overdecomposition allows greater flexibility for dynamic load balancing[6] and is even more important for out-of-core computing where the number of objects simultaneously allowed in memory is limited by available physical memory.

At the beginning, an application performs initial preprocessing (if necessary), creates mobile objects, defines serialization interfaces, registers message handlers, distributes the mobile objects between nodes (optional), initiates the parallel phase by posting the initial messages (e.g., main/driver function) and then passes control to the run-time system.

The execution progresses by executing messages handlers, posting messages and dynamically creating new mobile objects. A message is posted to perform an operation on the data of a particular mobile object. Messages can be addressed to local (including self), out-of-core and remote mobile objects. In fact, we strongly recommend using messages rather than function calls or other means of communication outside the context of the mobile object. Otherwise, the application is responsible for load balancing and for checking and ensuring availability of the data it tries to access.

A message addressed to a local mobile object is inserted into its message queue. If the object is local but out-of-core, the message is queued and the object is scheduled to be loaded in-core. If the object is remote, the message is routed to the corresponding node and processed there. The processing of a message from a remote node is the same as for a local message.

The bulk of parallel computations are performed inside message handlers. When no message handlers are executing and no messages are being delivered, the run-time system detects a termination condition. At this point the control is passed back to the application. Usually, at this point the application performs post-processing (if necessary) and terminates, although it is possible to start another phase of computing with the run-time system.

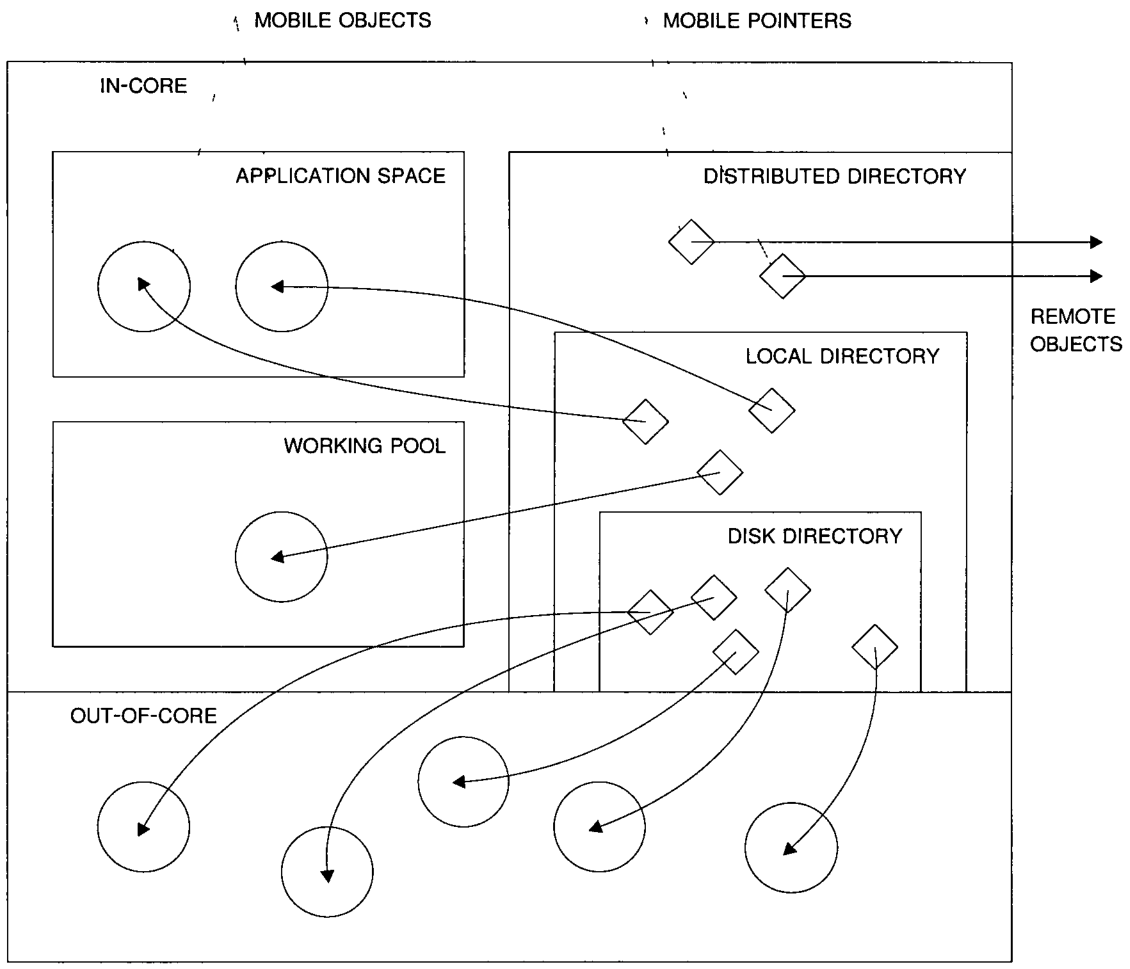


**Figure 4.1**: Memory organization and global addressing of the MRTS

## 4.4 Organization

The run-time system is organized into layers according to the principle of separation of concerns (see Fig. 4.1). Parallelism is exploited via multi-threading on a node level and via message passing between nodes. The memory space available to an application consists of local, disk and remote memory. Hence, we call our run-time system the Multi-layered Run-Time System). The MRTS is organized into the following layers: the storage layer, the out-of-core layer, the control layer and the computing layer.

The *storage layer* is used for managing mobile objects stored out-of-core. The underlying storage facility is hidden from the application and can utilize regular files, block devices and databases[2]. Blocking and non-blocking operations for loading and storing a mobile object are provided. This functionality is primarily used by the MRTS internally and is not exposed to an application.

The *out-of-core layer* is responsible for keeping track of mobile objects and controlling swapping (i.e., determining when and which objects should be un-/loaded from and to memory). The out-of-core layer also maintains a cache to prefetch mobile objects depending on swapping scheme and input from application.

The *control layer* is responsible for delivering messages either locally or remotely and for controlling migration of objects between nodes. Object location is determined by querying the mobile object distributed directory. Depending on the location of the object the message can be routed to a remote node or queued for local execution. The control layer decides the order in which message queues of local mobile objects are processed. The input from the control layer influences the swapping decisions of the out-of-core layer. In addition, the control layer provides memory management primitives to an application [2].

[2]The evaluation of different storage subsystems is out of scope of this paper and will be submitted elsewhere. Out-of-core objects are stored in a single large file and meta-data is kept in memory at all times for all experiments presented in this paper.

The *computing layer* is used to provide a uniform interface to various multi-threading technologies employed in the MRTS. We encourage the use of *tasks* – fragments of code that can run in parallel and are expected to complete without blocking. Each message handler function viewed as a task once it is scheduled to be executed and can spawn new tasks during the execution. Unlike messages tasks can only access data of the corresponding mobile object. However, tasks are lightweight and can be used to exploit fine-grain parallelism without much overhead. The computing layer manages the execution of message handlers and tasks and is responsible for memory allocation, synchronization and load balancing of the tasks between PEs (i.e., cores, nodes, racks).

## 4.5 Implementation

### 4.5.1 Software layers

The storage layer implements several swapping schemes which are based on popular cache algorithms. In addition to the least recently used (LRU) scheme, we implemented the least frequently used (LFU), the most recently used (MRU), the most used (MU) and the least used (LU) schemes. While the LRU scheme enjoys highest performance most of the time, for some applications (e.g., PCDM) the LFU can be up to 7% faster.

A set of swapping thresholds is used to influence swapping in normal cases as well as to force swapping in extreme cases. The hard swapping threshold is defined as a multiple of the size of the largest mobile object currently stored on disk. The actual value can be set at the initialization of the MRTS; the default is 2. This threshold is checked whenever the application wants to allocate additional memory. If the amount of memory after allocation is less than the threshold, unused objects are forcefully unloaded to free memory. The soft swapping threshold is defined as a fraction of the total available memory and is used to influence caching of the out-of-core mobile objects. When the amount of free memory drops below the soft threshold the storage layer is "advised" to start swapping. The soft threshold can be set at the initialization of the MRTS; the default is 0.5.

Additionally, the out-of-core layer provides an API to assign swapping priorities to mobile objects[3] and to directly lock/unlock mobile objects. The locking is straightforward: a locked object cannot be unloaded from memory before it is unlocked. The priorities are used to provide hints to the run-time system regarding the importance of keeping an object "in-core" while still allowing it to make final decisions.

The control layer uses preemptive communication internally. When such a message arrives it interrupts one of the currently running threads and gives control to the message handler. The control is returned back to the interrupted thread after message handler competed its execution. Executing potentially long running mobile messages can lead to high overheads. Therefore, application messages are queued upon arrival and executed when appropriate. When a message is removed from the queue it is "delivered" by executing its respective message handler. When the message handler terminates, the control layer makes a decision whether to continue to process the message queue of the current object, to switch to another object or to serve systems aspects like information dissemination and/or decision making for load-balancing or swapping. The control layer keeps track of all messages, including the messages of out-of-core mobile objects, and

[3]The swapping priority assigned to a mobile object is stored inside the corresponding mobile pointer data-structure.

assigns swapping priorities depending on the number of messages and the order in which they were delivered. Depending on the amount of work (i.e., number of messages) in-core, the control layer can "advise" the out-of-core layer to initiate swapping.

### 4.5.2 Mobile Objects and Threads

The mobile object directory that stores mobile pointers is a distributed directory with lazy updates [24]; for a mobile object that resides on a remote node, its last known location is stored. When a message is sent to that location it is not guaranteed that the destination mobile object will be there. If not, the message is forwarded to the last known location of the object on that node. When the message finally arrives at the object's current location, an update service message is sent back to all nodes through which the message was routed. In [24] experimental evaluation using different location management policies shows that lazy updates provide good compromise between accuracy and update overhead.

The computing layer provides a lightweight mostly-wrapper interface to multi-threading libraries. We encourage and support multi-threading within a message handler. Each message handler is a task and can be further broken into child tasks, and some of those tasks can be executed in parallel. We utilize two different but similar industrial-strength multi-threading programming technologies (only one can be active). (1) Intel Threading Building Blocks (TBB) [27] is a C++ template library designed to simplify and streamline parallel programming for C++ developers. It provides high-level abstraction, is based on generic programming, and is designed to hide low level details of managing threads and supports nested parallelism. (2) Grand Central Dispatch (GCD) [3] is an Apple technology used to optimize application support for systems with multiple and/or multi-core processors. GCD implements task parallelism based on the thread pool pattern. In both cases we use provided functionality to achieve task level parallelism within a message handler, a task can be implemented as a *block* in the case of GCD or as a method of the *task* class or a lambda function [28] in the case of TBB.

A user defined mobile object must implement initialization, un-/registration and de-/serialization methods. Initialization is performed when the object is first created; the object is unregistered when it has to be moved to another node and is registered when it is installed on a new node; the object is de-/serialized when it is transferred from/to disk.

Whenever a mobile object is created a mobile pointer is generated. Each mobile pointer contains either a reference to its object if that object is local and in-core, or its location otherwise. Additionally, a mobile pointer of a local mobile object is associated with a queue of messages that were delivered to the mobile object. When an object is loaded in-core the message queue is processed. The size of a message queue influences scheduling and swapping.

### 4.5.3 Message Passing

A message is composed of a destination mobile pointer, a message handler and optional arguments. A message handler is implemented as a function. When it is called, it is provided with a reference to the corresponding mobile object (not the mobile pointer) and optional arguments. Messages that are delivered to their destination nodes are stored together with the respective mobile objects. This means that if an object is out-of-core, its messages are also stored out-of-core. The number of messages in a message queue is stored in the respective mobile pointer.

To send a message to a mobile object the following should be supplied: a mobile pointer that identifies the destination mobile object, a message handler, and optional arguments. In case of a local mobile object the message is queued in the respective queue. Alternatively, the message is delivered through a one-sided communication mechanism to a last known node where the object might be located. A remote procedure call is performed to both deliver the message as well as to notify remote node of the delivery. We are using the Aggregate Remote Memory Copy Interface (ARMCI)[37] library for such low-level inter-node communications. The ARMCI is a portable one-sided communication library that can be used in MPI applications and offers an extensive set of functionality in the area of RMA communication: (1) data transfer operations, (2) atomic operations, (3) memory management and synchronization operations, and (4) locks. Additionally, the ARMCI library is part of the Global Arrays [38] which is popular in scientific computing and widely supported on existing and upcoming supercomputers, which, in turn, ensures the MRTS portability.

### 4.5.4 Object Migration

When an object is to be migrated to another node or stored out-of-core it must be appropriately serialized, i.e., packed. Then again, when an object is installed on a node or is loaded in-core it has to be de-serialized, i.e., unpacked. Due to the potentially complex internal structure of a mobile object, the serialization operation must be defined by the application. Not all mobile objects designated as out-of-core are actually unloaded to disk, some are cached in memory. To allow a high degree of flexibility for the out-of-core computing we provide several instruments of control. An application can choose not to influence the system altogether; in such a case the decision to load/store mobile objects is made based on their access pattern (i.e., message pattern). Alternatively, an application can assign priorities which requires high priority objects to be cached more often. Finally, an application can force loading an object as well as locking an object which means the object is loaded or stays in memory regardless of its access pattern and priority respectively. Note, an application should be very careful with locking too many objects since it can result in running out of memory.

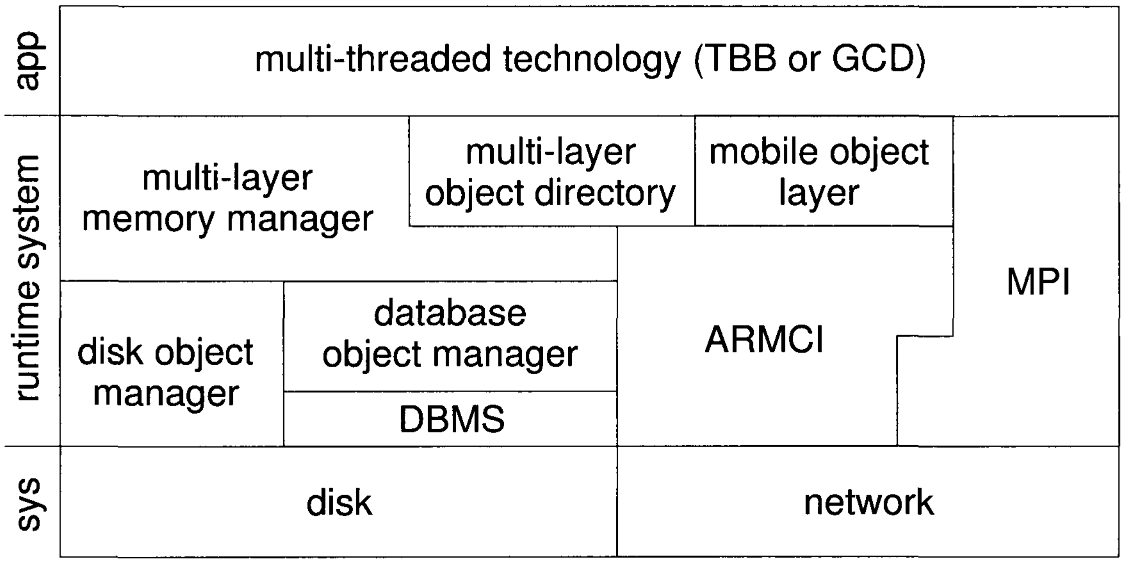


**Figure 4.2**: Software organization of the MRTS.

Figure 4.2 shows the software organization of the MRTS.

## 4.6 Out-of-core Non-Uniform Parallel Delaunay Refinement

In this section, we describe in more detail the out-of-core NUPDR method and its implementation with MRTS. Its in-core versions appeared in [16] for 2D and in [13] for 3D. We presented the out-of-core version in [35].

The NUPDR uses a master-worker model. The master starts by constructing a quadtree which initially contains a single leaf enclosing the entire geometry and an initial triangulation. Next, a queue of leaves containing poor quality triangles is generated (we will refer to it as a refinement queue). At this point the master enters a loop which will only terminate when the refinement queue is empty and no workers are computing. Termination of the loop indicates that the mesh is refined and the algorithm terminates.

Inside the loop, if the refinement queue is not empty and there is an available worker, a leaf is removed from the queue. Additionally, a buffer zone BUF of the leaf, which is defined as the collection of all neighboring leaves, is also removed from the queue. A leaf is then passed to an available worker for refinement.

If the queue is empty or no workers are available, the master waits for a worker to finish refining. When this happens, the leaves that compose the buffer BUF of the refined leaf are checked for poor quality triangles. All leaves that have bad triangles are reinserted into the refinement queue.

Poor quality triangles are stored as several structures based on a ratio between the side length of the enclosing leaf and their circumradius. A worker refines a leaf by processing poor quality triangle structures in a loop starting with the lowest ratio (largest triangles). In that loop a queue of poor triangles with a specific ratio is processed until it is empty.

For each poor triangle, a point is computed using a deterministic function and is inserted into the mesh. Then the mesh is updated which could lead to a propagation of changes into buffer leaves BUF and the creation of poor triangles for the current leaf and for the buffer leaves. As a result, the poor quality triangles are inserted into the corresponding data structures.

When both loops complete, the leaf is recursively split while a relation for constructing the quad-tree holds [13]. The locally refined mesh and quad-tree leaf are returned to the master.

### 4.6.1 Implementation

The MRTS programming model does not support master-worker pattern directly and as such, some restructuring of the algorithm is required. First, for each leaf of the quad-tree we create a mobile object which holds a portion of the mesh that is enclosed by this leaf. The refinement queue is also a mobile object. Additionally, the refinement queue mobile object holds and updates the quad-tree structure internally.

At the start a single thread creates the first top leaf mobile object and generates the initial mesh. In the process of mesh generation, the top leaf could be split and in such cases new mobile objects are constructed. Each leaf stores its list of poor quality triangles independently of the rest.

Next, a list of leaves that contain poor triangles is generated. A message designated `update` is sent to the refinement queue mobile object and the control is passed to the

MRTS. When the control is returned to the application the mesh is fully refined.

The `update` message takes the following arguments: a list of changes to the quad-tree, which is a list of mobile pointers to the newly created leaves and their relation to the existing leaves; a list of mobile pointers of the leafs that have bad triangles.

When an `update` message is received by the refinement queue mobile object, its handler performs the following. The quad-tree and the refinement queue are updated with the new leaves. If the refinement queue is empty (a list of leaves with bad triangles could be empty) the message handler exits. If not, a leaf is removed from the queue, its buffer BUF is computed, and the respective leaves are also removed from the queue. A message designated as `construct buffer` is sent to the leaf and its BUF buffer. The only arguments of the message are the mobile pointer of the leaf and the number of leaves in the buffer.

The message handler of `construct buffer` will do the following depending on the receiver. If the message is received by the leaf object, a counter is created with the number of leaves in the buffer. If the message is received by one of the leaves in the buffer, it sends the message `add to buffer` to the leaf being refined and frees the memory it used for storing its portion of the mesh.

The `add to buffer` message is used to deliver a portion of the mesh to another leaf. When an `add to buffer` message is received by a leaf, the counter of the buffer leaves is decremented and the argument mesh is integrated into the mesh of the receiving mobile object. When the counter reaches zero, a message designated as `refine` is sent to the leaf object (i.e., itself). The `refine` message takes no arguments.

The message handler of a `refine` message performs the same step as a worker in the

NUPDR algorithm. The only difference is the following. Instead of updating a global list of leaves with poor triangles, a local structure is created and updated through the refinement. After the refinement completes, an `update` message is sent to the refinement queue object. The local list of leaves with poor triangles as well as any changes made to quad-tree are passed as arguments to the `update` message. Then, new mobile objects are created as needed (for every new leaf) and the corresponding portions of the mesh are distributed among them. Finally, the portions of the mesh that correspond to the leaves other than the current leaf are returned to their owners via `recreate` messages.

In the end, when no message handlers are executing and no messages are traveling, we reach the termination condition. At this point the control is returned to the application and the algorithm completes.

### 4.6.2 Optimization

While the algorithm described above works correctly, it is not as efficient as it could be. Following are the number of changes we introduced to considerably improve the performance.

The refinement queue object is relatively small and receives and sends many messages. Therefore, we locked it in memory meaning it will never be unloaded out-of-core.

Since we operate in a shared memory environment, we try to minimize the use of `add to buffer` messages. We check whether the receiving leaf object is in-core, and in such a case call the message handler directly. When the handler is called directly the sender's mesh fragment is made available to the receiver and does not have to be copied. Consequently, the memory occupied by the mesh fragment is not freed and a `recreate` message is unnecessary.

The leaves that are part of the buffer are locked in memory after they send the `add to buffer` messages or call the respective handlers directly. They do not occupy a significant amount of memory at this point and do not require a `recreate` message anymore. Instead, a `recreate` message handler is called directly and afterwords the objects are unlocked (i.e., can be unloaded from memory). Similarly, we call the `refine` message handler directly, thus eliminating the possibility it will be forced out of memory before the message is delivered.

We change the order of the leaves in the refinement queue based on how many leaves are in their buffers. This way we try to have as many leaves as possible present together in-core and available for refining. We also check which leaves are in-core and try to refine the leaves with the most buffer leaves loaded.

Additional improvements come from managing the priorities of the out-of-core subsystem. When we remove a leaf from the refinement queue we check if it is currently loaded. If it is, we assign it a very high priority to minimize the possibility it will be unloaded before a `construct buffer` message arrives. Also, we assign different priorities to the leaves of the buffers depending on the order they were removed from the refinement queue.

### 4.6.3 Findings

The NUPDR algorithm requires access to several leaves of the quad-tree to refine a single leaf. To accommodate this we either have to collect all leaves in one mobile object dynamically on demand or store a single leaf in each object but then ensure that when the message is delivered all related objects are local and in-core. Since the MRTS discourages direct control over mobile objects we used the first approach. With optimization the ONUPDR using this approach performs similarly to the NUPDR. However, this discovery lead us to believe that the ability to collect several mobile objects during the execution of a mobile message can simplify the development and provide additional space for optimization.

We introduced a multicast mobile message to the MRTS. A multicast mobile message is similar to a mobile message except it can be sent to multiple mobile objects and ensure that specific mobile objects are loaded into memory when the message is delivered. Note that this is still experimental and requires further research and evaluation.

Instead of a destination mobile pointer, a vector of mobile pointers is supplied. Additionally, a counter specifies which objects will receive the message (first $n$ objects will receive the message, where $n$ is the counter). In the example of the ONUPDR, we would provide a vector containing mobile pointers of a leaf and its buffer as the first argument and 1 as the second argument, meaning the message should be delivered only to the leaf mobile object.

Internally, the MRTS must first collect all mobile objects from the vector on the same node and in-core, and only after that the mobile message is delivered. The message is then delivered to one or more mobile objects in the vector (depending on the second argument), order is not important, can be simultaneously.

# Chapter 5

# Performance Evaluation

## 5.1 Experimental Setup

For the evaluation we used the CRTC cluster which is part of the Center for Real-time Computing[1] 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 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

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

server and one dual-processor OpenPower720 management and software development node. We also employed some nodes of the SciClone cluster at the College of William and Mary[2] (64 single-cpu Sun Fire V120 servers at 650 MHz with 1 GB memory and 32 dual-cpu Sun Fire 280R servers at 900 MHz with 2 GB memory).

## 5.2 Out-of-core Parallel Delaunay Refinement

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 [15] and are out of scope of this thesis. However, it 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 5.1, left). This test geometry shows that the impact of load imbalances is much less severe to the out-of-core PDR algorithms.

To date, there are no agreed standards to evaluate the performance of out-of-core algorithms and existing metrics suitable for in-core parallel algorithms are not sufficient for this task. Usually, we expect an out-of-core algorithm to have the following qualities:

- for small problems that can fit in-core, the execution time should be as close as possible to that of an in-core counterpart algorithm
- for large problems that do not fit in-core it is acceptable to have lower performance

[2]http://compsci.wm.edu/SciClone

yet it should be comparable to the performance of an in-core algorithm for the same number of processors

- for the same hardware setup it should be possible to solve much larger problems with an out-of-core algorithm than with its in-core counterpart, however the execution time will be longer

- ideally, *per processor* performance should not degrade as problem size increases while the average number of processors and physical memory stays constant

Therefore, to evaluate the performance of the out-of-core methods, we compare in-core methods and out-of-core methods using the notion of *normalized speed* that we introduced earlier [34]. This measure computes the number of elements generated by a single processor over a unit time period, and it is given by $V = \frac{N}{T \times P}$, where $N$ is the number of elements generated, $P$ is the number of processors in the configuration and $T$ is the total execution time.

In order to compare the performance of the in-core and the out-of-core 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 $V = \frac{N}{T \times P}$, where $N$ is the number of elements generated, $P$ is the number of processors in the configuration and $T$ is the total execution time.

Tables 5.2 and 5.1 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 the OHPDR method can be as low as 19% slower (and no method is more than about two times slower) than its counterpart in-core PDR method for the mesh sizes that fit completely in the core of the CoWs.

| 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 5.1**: 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 |

**Table 5.2**: 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 5.3 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.

| 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 5.3**: 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. Wait-in-queue time is included when computing normalized speed for the in-core algorithm.

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 [34]. 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.

| 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 5.4**: 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 5.4 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-decomposed [6] 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.

## 5.3 Out-of-Core Parallel Constrained Delaunay Meshing

The mesh generation time in practice is linear with respect to the number of the resulting triangles. The number of elements is roughly inversely proportional to the required triangle area bound, and can be controlled by selecting the area bound correspondingly. The estimation is not an exact prediction of the size of the final mesh but it works well in our experiments. We used several geometries (see Fig. 5.1) for our evaluation with the same triangle shape constraint (20° minimal angle).

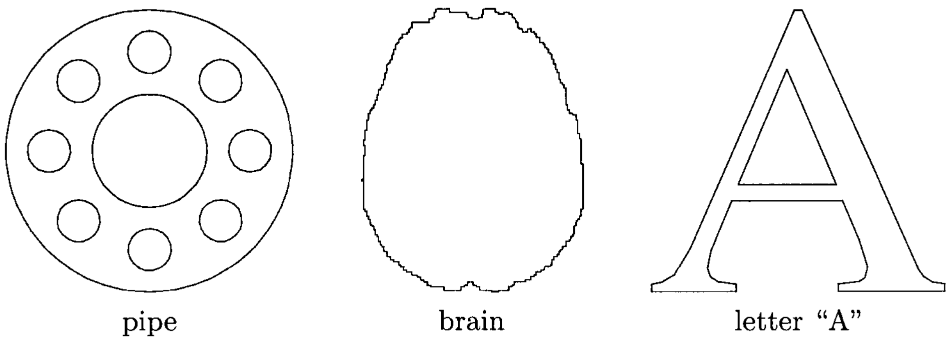


**Figure 5.1**: Geometries used for evaluation: pipe cross-section, brain cross-section and letter A.

Additionally, we compare the effectiveness of the two different object managers: disk and database. Below we will use OPCDM(d) and OPCDM(b) to refer to the experiments performed with the disk object manager and the database object manager respectively.

Table 5.5 shows the average sustained speeds of local disks. These speeds are upper bounds for any disk operations. We will use these to determine the utilization of the disks in our evaluation. We measure the average disk read and write speeds for our applications and present them as disk utilization below as fractions of sustained speeds.

Table 5.6 shows the normalized speed for problem sizes that fit completely in-core on varying number of processors. The problem sizes are experimentally chosen to be

**Table 5.5**: Average sustained speed of local disks for nodes 1 through 4.

| Disk | Sustained speed (MB/sec) for node: | | | |
|---|---|---|---|---|
| operation | 1 | 2 | 3 | 4 |
| read | 25.01 | 25.53 | 24.61 | 25.34 |
| write | 18.58 | 17.33 | 17.95 | 18.87 |

as large as possible and still fit in memory when computed with the PCDM. We can see that the performance of the OPCDM is very close to that of the PCDM yet it is slightly slower due to the overheads and it has slightly larger memory footprint (some of the OPCDM data may be stored out-of-core). There is no difference between using the disk object manager or the database object manager. Since the problem that fits completely in memory would not trigger the use of virtual memory we do not have a separate column for the case when virtual memory is used.

**Table 5.6**: Normalized speed of the PCDM and the OPCDM for problems that fit completely in-core. OPCDM(d) and OPCDM(b) denote respectively OPCDM with disk and database OoC subsystems. Pipe geometry.

| Mesh size, $\times 10^6$ triangles | number of PEs (nodes) | Normalized speed, $\times 10^3$ triangles per second | | |
|---|---|---|---|---|
| | | PCDM | OPCDM(d) | OPCDM(b) |
| 19.79 | 2(1) | 68.61 | 60.01 | 58.91 |
| 39.55 | 4(1) | 70.10 | 62.06 | 61.35 |
| 79.11 | 8(1) | 72.11 | 64.07 | 64.91 |
| 158.25 | 16(2) | 70.18 | 64.01 | 63.86 |
| 316.50 | 32(4) | 69.95 | 62.25 | 61.91 |

Table 5.7 compares the out-of-core performance of the PCDM with virtual memory to that of the OPCDM for problem sizes that have the memory footprint twice as large as the available physical memory. We use half the PEs for the same problem sizes when compared to Table 5.6.

Tables 5.8,5.10 and 5.12 demonstrate the effectiveness of the out-of-core approach

**Table 5.7**: *Normalized speed of the PCDM with virtual memory and the OPCDM for problems that have memory footprint twice as large as the available physical memory.* OPCDM(d) and OPCDM(b) denote respectively OPCDM with disk and database OoC subsystems. Pipe geometry.

| Mesh size, $\times 10^6$ triangles | number of PEs (nodes) | Normalized speed, $\times 10^3$ triangles per second | | |
|---|---|---|---|---|
| | | PCDM | OPCDM(d) | OPCDM(b) |
| 158.25 | 8(1) | 29.23 | 55.59 | 54.24 |
| 316.50 | 16(2) | 29.00 | 57.31 | 55.63 |
| 633.07 | 32(4) | 28.91 | 55.93 | 54.75 |

for computing very large problems in real-life environment. Because such problems will not fit into physical memory of a small cluster like ours one has to run the application on a larger cluster. Therefore, one must factor in the time that is spent in queue waiting for the job to schedule. We used the wall-clock time instead of the total execution time to compute the normalized speed shown in the last table. The wall-clock time is the sum of the wait-in-queue time and the total execution time. We used the average wait-in-queue time for a given number of processors from the statistical data gathered on the SciClone[3] cluster during several years (see Fig. 1.1).

Tables 5.9, 5.11, and 5.13 show utilization of the disks and overlap of computation and disk I/O. Disk utilization is roughly a quarter of the peak possible which is acceptable considering commercial tools achieve about one third of the peak for sparse data access. Overlap is also quite high (up to 68%) when taking into account complexity of our application.

We see that the normalized speed for the OPCDM does not change much as we increase the problem size or the geometry. Disk utilization and overlap also do not change much. At the same time, the wait-in-queue time dominates the wall-clock time

[3]http://www.compsci.wm.edu/SciClone/

Table 5.8: Normalized speed of the PCDM(estimated) and the OPCDM for large problem sizes. The normalized speed for the PCDM is estimated using statistical data for wait-in-queue time and average per processor performance demonstrated on smaller in-core problems. The normalized speed for the OPCDM is computed from the actual total execution time using 16 PE (2 nodes) with total physical memory of 16 GB on varying problem sizes (there is no wait-in-queue time for the OPCDM). OPCDM(d) and OPCDM(b) denote respectively OPCDM with disk and database OoC subsystems. Pipe geometry.

| Mesh size, $\times 10^6$ triangles | number of PEs (est.) | Normalized speed, $\times 10^3$ triangles per second | | |
|---|---|---|---|---|
| | | PCDM(est.) | OPCDM(d) | OPCDM(b) |
| 949.47 | 48 | 7.65 | 54.56 | 54.09 |
| 1265.96 | 64 | 3.38 | 53.79 | 52.90 |
| 1582.44 | 80 | 3.24 | 57.79 | 57.74 |
| 2531.91 | 128 | 1.57 | 52.80 | 53.84 |
| 3164.89 | 160 | 2.15 | 48.75 | 47.30 |
| 3956.11 | 200 | 1.76 | 51.85 | 53.23 |

Table 5.9: OPCDM disk utilization and I/O overlap using disk OoC subsystem. Utilization is shown as a fraction of achievable speed. Overlap is shown as a fraction of total time. Pipe geometry.

| Mesh size, $\times 10^6$ triangles | read (%) | write (%) | overlap (%) |
|---|---|---|---|
| 949.47 | 27.74 | 22.19 | 51.67 |
| 1265.96 | 27.93 | 22.35 | 63.00 |
| 1582.44 | 24.51 | 19.61 | 53.48 |
| 2531.91 | 22.40 | 17.92 | 61.50 |
| 3164.89 | 24.01 | 19.21 | 59.86 |
| 3956.11 | 25.59 | 20.47 | 58.38 |

for the PCDM and this results in a much lower normalized speed. It is clear that the in-core generation of very large meshes on large clusters with hundreds of processors is less effective in terms of time than the out-of-core generation of the same meshes on small clusters with limited number of processors and physical memory. Additionally, we see that the OPCDM(b) performs slightly better as problem size increases. It supports our assumption that databases can be used to store out-of-core data.

**Table 5.10**: Normalized speed of the PCDM(estimated) and the OPCDM for large problem sizes. The normalized speed for the PCDM is estimated using statistical data for wait-in-queue time and average per processor performance demonstrated on smaller in-core problems. The normalized speed for the OPCDM is computed from the actual total execution time using 16 PE (2 nodes) with total physical memory of 16 GB on varying problem sizes (there is no wait-in-queue time for the OPCDM). OPCDM(d) and OPCDM(b) denote respectively OPCDM with disk and database OoC subsystems. Brain geometry.

| Mesh size, $\times 10^6$ triangles | number of PEs (est.) | Normalized speed, $\times 10^3$ triangles per second | | |
|---|---|---|---|---|
| | | PCDM(est.) | OPCDM(d) | OPCDM(b) |
| 981.95 | 48 | 7.42 | 55.64 | 55.40 |
| 1282.79 | 64 | 3.50 | 51.34 | 50.84 |
| 1599.01 | 80 | 3.37 | 57.07 | 57.14 |
| 2468.28 | 128 | 1.63 | 52.06 | 55.10 |
| 3316.15 | 160 | 2.13 | 47.37 | 45.73 |
| 3962.73 | 200 | 1.81 | 53.02 | 52.53 |

**Table 5.11**: OPCDM disk utilization and I/O overlap using disk OoC subsystem. Utilization is shown as a fraction of achievable speed. Overlap is shown as a fraction of total time. Brain geometry.

| Mesh size, $\times 10^6$ triangles | read(%) | write(%) | overlap(%) |
|---|---|---|---|
| 981.95 | 27.22 | 21.62 | 55.27 |
| 1282.79 | 30.53 | 22.25 | 59.52 |
| 1599.01 | 23.04 | 20.09 | 52.26 |
| 2468.28 | 22.46 | 16.65 | 61.03 |
| 3316.15 | 23.15 | 18.80 | 62.47 |
| 3962.73 | 28.08 | 20.37 | 57.04 |

## 5.4 Multi-layered Run-Time System

We start by evaluating the performance of the control layer of the MRTS. We tested small problems sizes on CRTC for all three methods and very large problems were tested on SciClone for in-core methods.

Figure 5.2 shows the execution times of the UPDR (16 and 25 PEs) and the OUPDR (16 PEs). The largest problem size on the chart, 175 million elements is too large for UPDR running on 16 processors. We can see that the performance of the

**Table 5.12**: Normalized speed of the PCDM(estimated) and the OPCDM for large problem sizes. The normalized speed for the PCDM is estimated using statistical data for wait-in-queue time and average per processor performance demonstrated on smaller in-core problems. The normalized speed for the OPCDM is computed from the actual total execution time using 16 PE (2 nodes) with total physical memory of 16 GB on varying problem sizes (there is no wait-in-queue time for the OPCDM). OPCDM(d) and OPCDM(b) denote respectively OPCDM with disk and database OoC subsystems. Letter "A" geometry.

| Mesh size, $\times 10^6$ triangles | number of PEs (est.) | Normalized speed, $\times 10^3$ triangles per second | | |
|---|---|---|---|---|
| | | PCDM(est.) | OPCDM(d) | OPCDM(b) |
| 925.78 | 48 | 7.31 | 55.05 | 56.41 |
| 1244.82 | 64 | 3.49 | 52.26 | 50.77 |
| 1585.31 | 80 | 3.37 | 56.90 | 60.23 |
| 2437.68 | 128 | 1.49 | 54.66 | 51.30 |
| 3147.60 | 160 | 2.10 | 48.70 | 49.59 |
| 3974.47 | 200 | 1.69 | 51.66 | 51.14 |

**Table 5.13**: OPCDM disk utilization and I/O overlap using disk OoC subsystem. Utilization is shown as a fraction of achievable speed. Overlap is shown as a fraction of total time. Letter "A" geometry.

| Mesh size, $\times 10^6$ triangles | read(%) | write(%) | overlap(%) |
|---|---|---|---|
| 925.78 | 24.62 | 19.7 | 56.68 |
| 1244.82 | 23.11 | 18.49 | 67.97 |
| 1585.31 | 27.34 | 21.87 | 65.56 |
| 2437.68 | 23.88 | 19.1 | 53.72 |
| 3147.6 | 22.31 | 17.84 | 67.34 |
| 3974.47 | 27.19 | 21.75 | 63.11 |

UPDR and that of the OUPDR is very similar (the OUPDR is up to 12% slower) for in-core problem sizes which means that the overhead introduced by the MRTS is small. Figure 5.3 shows the execution times of the NUPDR and the ONUPDR for 2, 4, and 8 PEs[4]. For 4 and 8 PEs, the overhead can be as high as 18% which is acceptable. For 2 PEs the ONUPDR is up to 41% slower. This is explained by the fact that the NUPDR uses a custom memory allocator that shows much lower overhead than the

[4]The NUPDR and current implementation of the ONUPDR are shared memory applications and as such are restricted to a single node

MRTS memory manager in the 2 PEs case. Figure 5.4 shows the execution times of the PCDM (16 and 25 PEs) and the OPCDM[33] for 8 and 16 processors. As is the case with the UPDR and OUPDR, the performance of the OPCDM is very similar to that of the PCDM (up to 13% overhead).

Figures 5.5, 5.6 and 5.7 demonstrate the performance of the out-of-core and storage layers of the MRTS. They show the execution times of the OUPDR (8 and 16 PEs), ONUPDR (2, 4 and 8 PEs) and OPCDM (8 and 16 PEs) for very large problems. These charts demonstrate that the size of very large problems do not degrade the performance of the methods (time increases almost linearly) on MRTS.

**Table 5.14**: Single PE performance of UPDR and OUPDR methods.

| Size | PEs | Time (sec) | | Speed ($\times 10^3$/sec) | |
|---|---|---|---|---|---|
| $\times 10^6$ | | UPDR | OUPDR | UPDR | OUPDR |
| 24 | 4 | 294 | 46 | 20 | 33 |
| 59 | 9 | 295 | 102 | 22 | 36 |
| 109 | 16 | 295 | 176 | 23 | 39 |
| 175 | 25 | 297 | 368 | 24 | 30 |
| 255 | 36 | 293 | 576 | 24 | 28 |
| 353 | 49 | 295 | 802 | 24 | 27 |
| 471 | 64 | 300 | 1133 | 25 | 26 |
| 588 | 81 | 296 | 1386 | 24 | 27 |
| 739 | 100 | 300 | 1745 | 25 | 26 |
| 874 | 121 | 294 | 2111 | 25 | 26 |
| 1284 | n/a | n/a | 3122 | 0 | 26 |
| 1967 | n/a | n/a | 4599 | 0 | 27 |

Tables 5.14, 5.15 and 5.16 reflect the performance of the out-of-core layer as well as the performance of the control layer. Note, the execution time of the original application is from older SciClone cluster since they need the aggregate memory of over a hundred processors. The MRTS applications run on the newer faster CRTC cluster and have faster per PE speed in most cases. Rather than compare the actual speeds in those

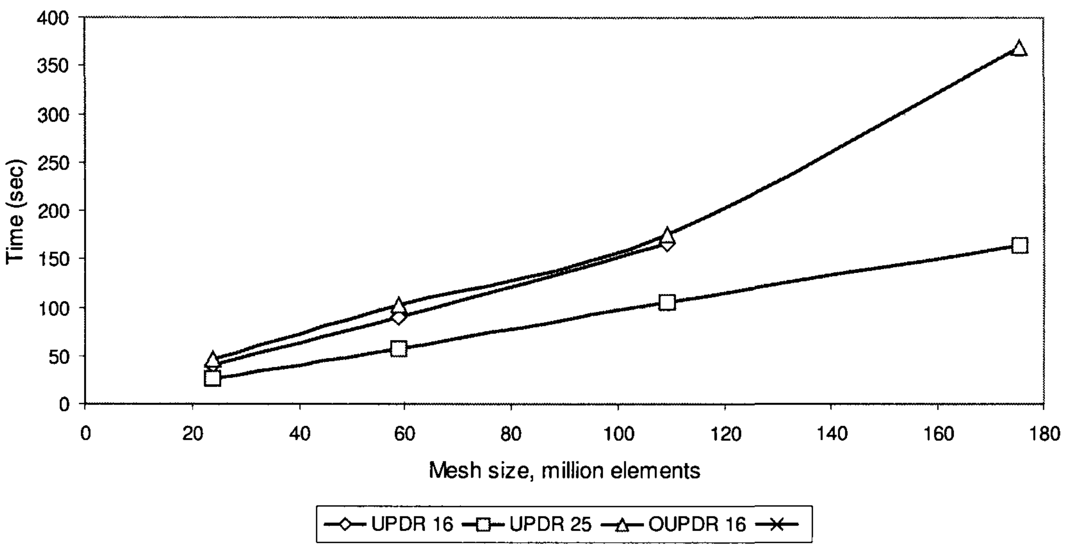


**Figure 5.2**: Execution times for UPDR and OUPDR for in-core problem sizes

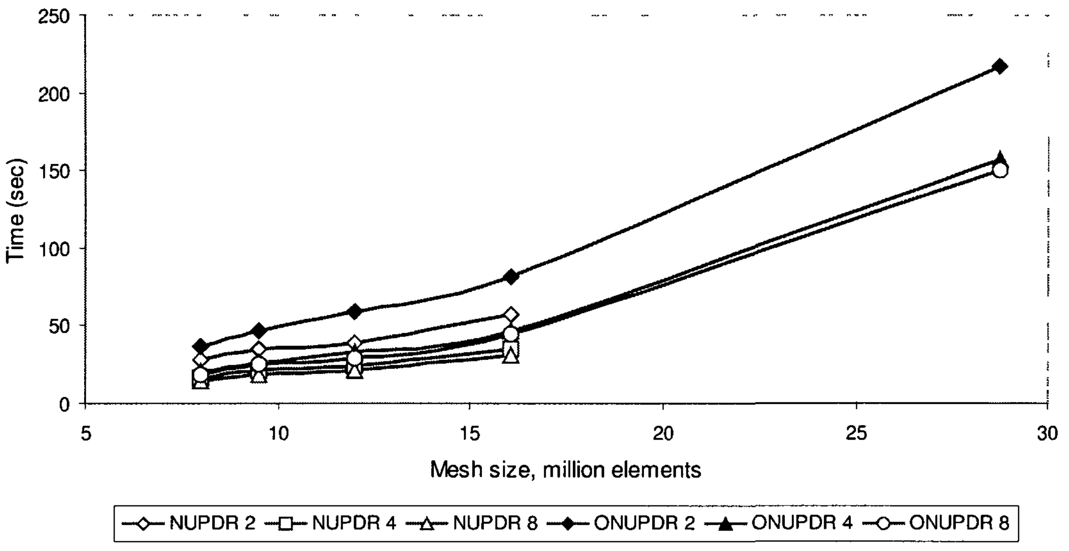


**Figure 5.3**: Execution times for NUPDR and ONUPDR for in-core problem sizes

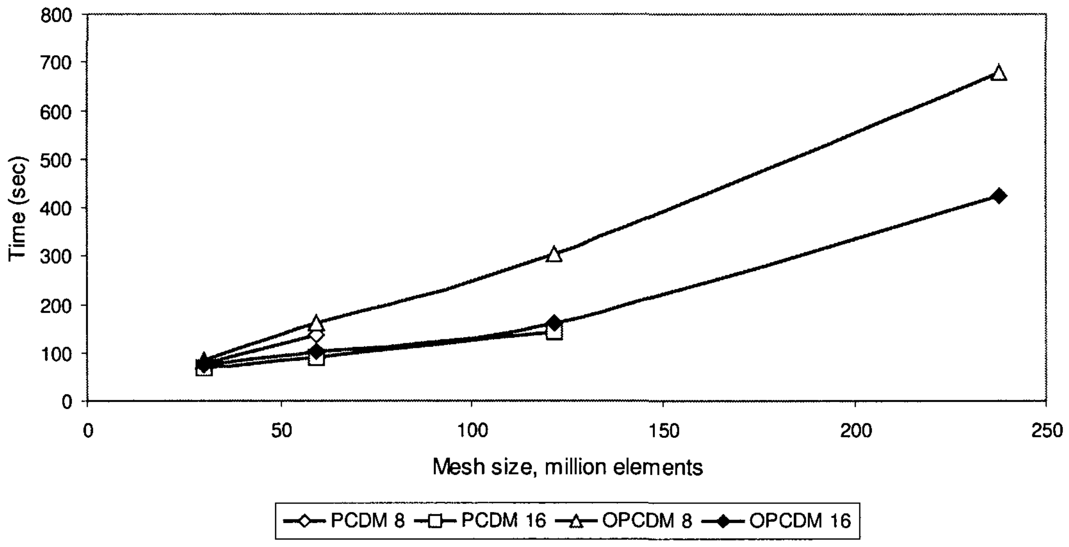


**Figure 5.4**: Execution times for PCDM and OPCDM for in-core problem sizes

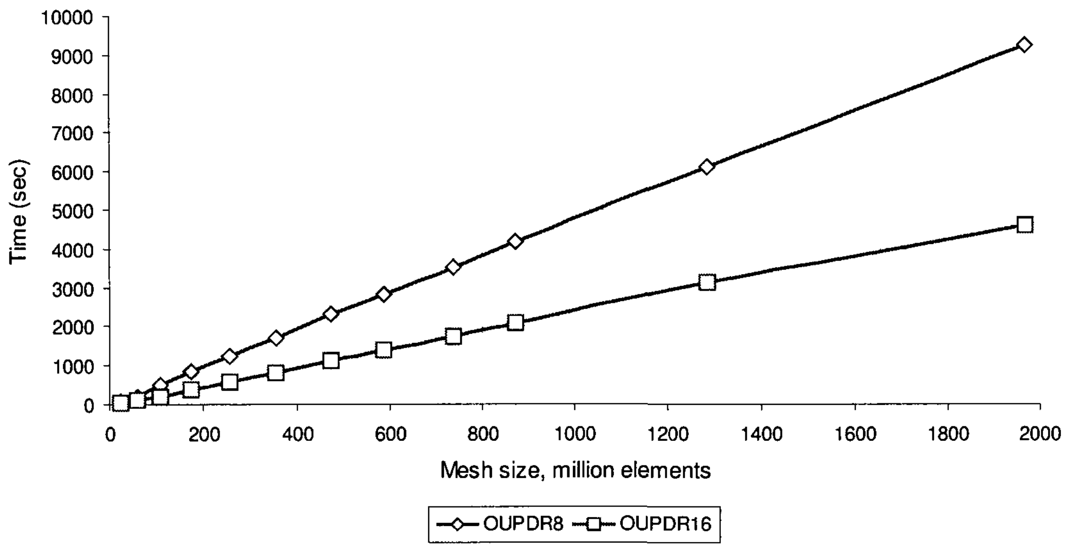


**Figure 5.5**: Execution times for OUPDR for out-of-core problem sizes

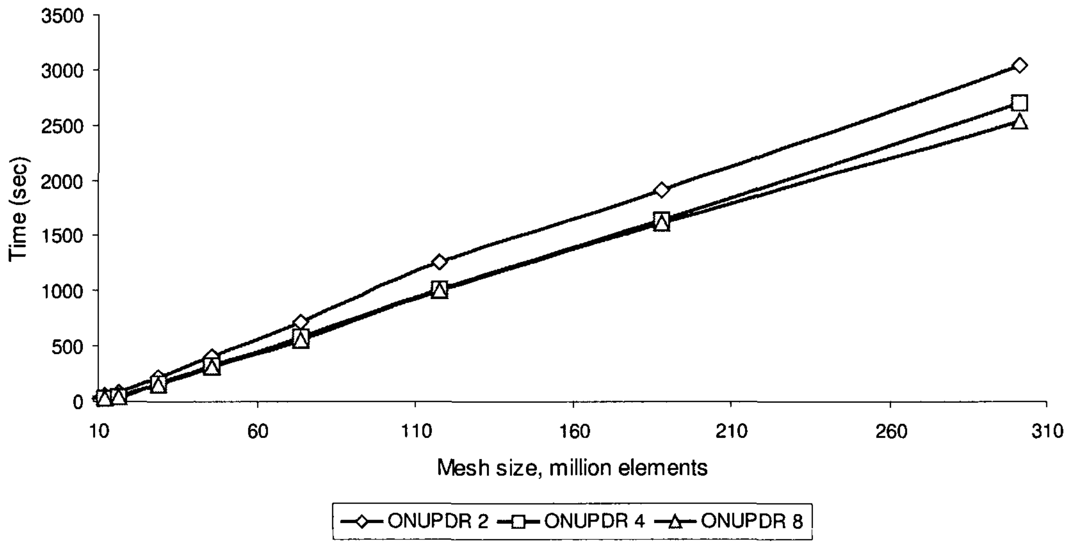


**Figure 5.6**: Execution times for ONUPDR for out-of-core problem sizes

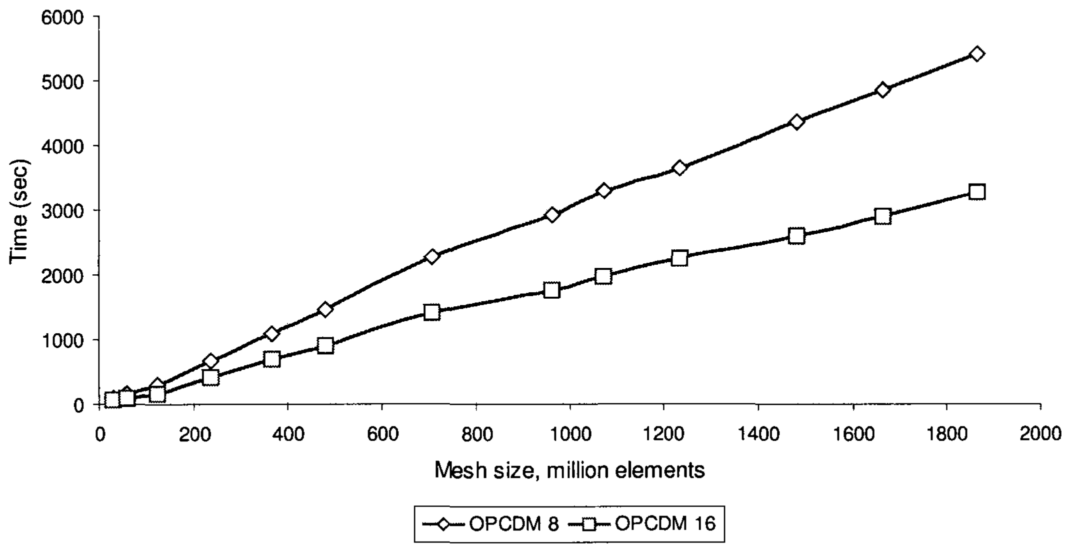


**Figure 5.7**: Execution times for OPCDM for out-of-core problem sizes

**Table 5.15**: Single PE performance of NUPDR and ONUPDR methods.

| Size, | Time (sec) | | Speed ($\times 10^3$/sec) | |
|---|---|---|---|---|
| $\times 10^6$ | NUPDR | ONUPDR | NUPDR | ONUPDR |
| 8 | 17 | 20 | 119 | 100 |
| 9 | 21 | 27 | 114 | 89 |
| 12 | 24 | 33 | 124 | 90 |
| 16 | 35 | 46 | 115 | 86 |
| 29 | n/a | 157 | n/a | 46 |
| 46 | n/a | 322 | n/a | 36 |
| 74 | n/a | 589 | n/a | 31 |
| 118 | n/a | 1016 | n/a | 29 |
| 188 | n/a | 1638 | n/a | 29 |
| 301 | n/a | 2702 | n/a | 28 |

**Table 5.16**: Single PE performance of PCDM and OPCDM methods.

| Size | PEs | Time (sec) | | Speed ($\times 10^3$/sec) | |
|---|---|---|---|---|---|
| $\times 10^6$ | | PCDM | OPCDM | PCDM | OPCDM |
| 30 | 4 | 308 | 73 | 24 | 26 |
| 59 | 8 | 296 | 101 | 25 | 37 |
| 122 | 16 | 319 | 163 | 24 | 47 |
| 238 | 32 | 310 | 425 | 24 | 35 |
| 366 | 48 | 327 | 707 | 23 | 32 |
| 480 | 64 | 304 | 918 | 25 | 33 |
| 706 | 96 | 324 | 1408 | 23 | 31 |
| 963 | 128 | 299 | 1772 | 25 | 34 |
| 1074 | n/a | n/a | 1986 | n/a | 34 |
| 1235 | n/a | n/a | 2256 | n/a | 34 |
| 1480 | n/a | n/a | 2614 | n/a | 35 |
| 1662 | n/a | n/a | 2900 | n/a | 36 |
| 1864 | n/a | n/a | 3285 | n/a | 35 |

tables we want to see the trend as we increase the problem size. We can see that the original applications as well as the MRTS implementations seem to maintain more or less constant speed. This means that as we increase the problem size the MRTS is able to sustain the performance level. Additionally, for the original applications this means they scale rather well [13, 15, 17, 18].

**Table 5.17**: Overlap of computation, communication and out-of-core disk I/O in the OUPDR.

| Size | Time | Comp | Comm | Disk | Overlap (%) | | |
|---|---|---|---|---|---|---|---|
| $\times 10^6$ | (sec) | (%) | (%) | (%) | min | max | avg |
| 24 | 46 | 88 | 18 | 0 | 1 | 7 | 6 |
| 59 | 102 | 85 | 16 | 0 | 0 | 2 | 1 |
| 109 | 176 | 86 | 21 | 0 | 2 | 8 | 7 |
| 175 | 368 | 65 | 15 | 36 | 4 | 19 | 16 |
| 255 | 576 | 61 | 12 | 51 | 8 | 29 | 24 |
| 353 | 802 | 58 | 11 | 61 | 6 | 35 | 30 |
| 471 | 1133 | 57 | 13 | 64 | 11 | 38 | 33 |
| 588 | 1386 | 55 | 13 | 70 | 5 | 46 | 38 |
| 739 | 1745 | 54 | 14 | 73 | 5 | 48 | 41 |
| 874 | 2111 | 51 | 18 | 73 | 6 | 54 | 42 |
| 1284 | 3122 | 52 | 18 | 76 | 5 | 57 | 46 |
| 1967 | 4599 | 53 | 16 | 82 | 20 | 63 | 50 |

**Table 5.18**: Overlap of computation, synchronization and out-of-core disk I/O in the ONUPDR.

| Size | Time | Comp | Sync | Disk | Overlap (%) | | |
|---|---|---|---|---|---|---|---|
| $\times 10^6$ | (sec) | avg (%) | avg (%) | avg (%) | min | max | avg |
| 8 | 20 | 98 | 2 | 0 | 0 | 0 | 0 |
| 9 | 27 | 99 | 1 | 0 | 0 | 0 | 0 |
| 12 | 33 | 98 | 2 | 0 | 0 | 0 | 0 |
| 16 | 46 | 98 | 2 | 0 | 0 | 0 | 0 |
| 29 | 157 | 51 | 1 | 81 | 5 | 38 | 33 |
| 46 | 322 | 40 | 1 | 103 | 7 | 52 | 43 |
| 74 | 589 | 36 | 1 | 112 | 7 | 56 | 48 |
| 118 | 1016 | 35 | 1 | 116 | 17 | 58 | 52 |
| 188 | 1638 | 32 | 1 | 123 | 18 | 64 | 56 |
| 301 | 2702 | 33 | 0 | 124 | 17 | 64 | 58 |

Tables 5.17, 5.18 and 5.19 are presented to demonstrate the out-of-core performance

**Table 5.19**: Overlap of computation, communication and out-of-core disk IO in the OPCDM.

| Size | Time | Comp | Comm | Disk | Overlap (%) | | |
|---|---|---|---|---|---|---|---|
| $\times 10^6$ | (sec) | avg (%) | avg (%) | avg (%) | min | max | avg |
| 30 | 73 | 49 | 53 | 0 | 0 | 2 | 2 |
| 59 | 101 | 64 | 36 | 0 | 0 | 0 | 0 |
| 122 | 163 | 94 | 12 | 0 | 2 | 7 | 5 |
| 238 | 425 | 66 | 7 | 50 | 4 | 27 | 23 |
| 366 | 707 | 62 | 5 | 64 | 8 | 36 | 30 |
| 480 | 918 | 60 | 4 | 72 | 6 | 43 | 36 |
| 706 | 1408 | 61 | 3 | 76 | 10 | 50 | 40 |
| 963 | 1772 | 57 | 3 | 87 | 6 | 56 | 47 |
| 1074 | 1986 | 58 | 3 | 88 | 8 | 63 | 49 |
| 1235 | 2256 | 59 | 3 | 91 | 9 | 65 | 53 |
| 1480 | 2614 | 58 | 3 | 95 | 14 | 67 | 57 |
| 1662 | 2900 | 59 | 4 | 98 | 10 | 73 | 60 |
| 1864 | 3285 | 60 | 4 | 97 | 7 | 74 | 62 |

of the MRTS applications. These tables show computation, communication (or synchronization for ONUPDR) and disk I/O as a percentage of total execution time. The last three columns show overlap of computation, communication/synchronization and disk I/O which we compute as $Overlap = \frac{Comp+Comm+Disk-Total}{Total} \times 100\%$, where $Comp$ is the computation time, $Comm$ is the communication/synchronization time, $Disk$ is the disk I/O time and $Total$ is the total execution time. MRTS is designed to promote overlapping of communication and I/O and our data show we have been very successful at it. The overlap is over 50% for large problems and can be as high as 62%. This means the MRTS is capable of tolerating high latencies rather well and accommodate data-intensive application.

The MRTS can use and supports either GCD or TBB multi-threading libraries to utilize shared-memory computing. Since GCD availability on non-Apple systems is very limited yet we had to use an older system running an experimental version of FreeBSD:

Dell PowerEdge 6600 with 4 Intel Xeon MP 1.47 GHz processor and 16 GB of memory.

**Table 5.20**: The comparison of performance of the computing layer implementations.

| Size, | Threading Building Blocks | | | Grand Central Dispatch | | |
|---|---|---|---|---|---|---|
| $\times 10^6$ | T1(sec) | T4(sec) | Spdup | T1(sec) | T4(sec) | Spdup |
| 7.97 | 49.20 | 24.94 | 1.97 | 46.29 | 27.54 | 1.68 |
| 9.49 | 60.98 | 31.88 | 1.91 | 61.89 | 34.05 | 1.82 |
| 11.98 | 70.38 | 32.93 | 2.14 | 71.17 | 37.84 | 1.88 |
| 16.04 | 114.59 | 56.66 | 2.02 | 115.31 | 60.11 | 1.92 |

Table 5.20 shows sequential time (T1), parallel time with 4 PEs (T4) and relative speedup (Spdup) for the ONUPDR with TBB and GCD implementations of the computing layer. Size is the number of elements in the resulting mesh, a pipe cross-section geometry was used for all experiments. The speedup is comparable to the speedup of the NUPDR, We can see that GCD implementation is slightly slower yet we can see similar trends for both implementations.

# Chapter 6

# Conclusion and Future work

This thesis aims to provide an approach for *effective* computing of large irregular scientific problems such as unstructured mesh generation. The main contributions are design, implementation and evaluation of the Multi-layered Run-Time System, a practical parallel out-of-core runtime system, which enables out-of-core computing for both new and existing applications with small cost in performance and labor.

We followed an evolutionary approach to out-of-core computing. First, we designed and implemented several custom out-of-core codes based on existing state of the art in-core algorithm, PDR. We achieved good performance with low overhead (as low as 19% for our best method) and were able to demonstrate the effectiveness of out-of-core approach. That is, using our custom out-of-core codes we were able to solve larger than otherwise possible problems, or solve problems of the same size (compared to the in-core method) using significantly fewer PEs. Compared to the in-core method, this allows to potentially achieve shorter wall-clock time (time between user submits his application and gets the results back) on shared computing resources. In fact, in the case of SciClone cluster the wall-clock time of our best out-of-core method can be as low as one third of the wall-clock time of the in-core method.

Despite good performance, developing custom out-of-core codes from scratch is time consuming and labor intensive task. Additionally, restructuring of in-core algorithms is often required, which means the same out-of-core solution cannot be easily applied to a different algorithm. To counter this we focused on a runtime system rather than a single application. Our next step was to design and implement out-of-core support for an application based on PREMA framework. If successful, this can be reused for any application built on top of PREMA. We designed and implemented an out-of-core version of the PCDM method. Focusing on the framework rather than specific application simplified the porting process and still produced acceptable results in terms of performance. The Out-of-core PCDM adds little overhead compared to the PCDM and shows high overlap (up to 68%) of computation, communication and disk I/O.

Finally, we used our experience to design and implement the MRTS, which enables out-of-core computing automatically for any application that was built on top or ported to this runtime system. The MRTS extends PREMA by adding "free" out-of-core support and interfaces for fine-grain parallelism. We ported existing in-code methods, UPDR, PCDM and NUPDR, to demonstrate that the porting process is not overly complex. In fact, the porting of PCDM which uses PREMA programming model was straightforward.

We used traditional CoWs to perform an evaluation of our implementation using three parallel unstructured mesh generation methods with a wide spectrum of memory access patterns and communication/synchronization requirements to stress test the

MRTS. In particular, the NUPDR was used to test multi-threaded performance, the UPDR was used to test structured communication with some synchronization, and the PCDM was used to test fully asynchronous communication. Furthermore, each application tested the out-of-core subsystem. The performance of the MRTS-based codes was slightly worse than that of custom out-of-core codes (overlap 50% on average and up to 61% for large problem sizes). However, this is small price to pay for shorter and simplified development or porting.

The MRTS is implemented on top of established software libraries and standards like TBB/GCD for multi-threading and ARMCI/MPI for both one- and two-sided message passing. This permits incremental application development for multi-layered parallel architectures. Moreover, it allows for an evolutionary approach to the migration of complex applications (e.g., parallel mesh generation) from traditional parallel platforms to emerging massively parallel platforms.

In the future, I plan to continue working on MRTS and make it available on emerging platforms, like Blue Waters supercomputer. While there is no possibility (nor necessity) for traditional out-of-core computing on Blue Waters its highly hierarchical architecture makes principles used in the design and development of the MRTS and out-of-core applications relevant. Since many applications can benefit from large memory of Blue Waters but cannot take advantage of the high level of concurrency, one possible scenario is to partition resources into compute nodes and memory nodes and use MRTS for managing data flow between them.

GPU computing is gaining popularity among scientific applications and is conceptually similar to traditional out-of-core computing with system memory replacing the disk. I see potential usefulness of MRTS in this area in the near future and plan to add implementation of the computing layer using OpenCL for compatibility with both high-count multi-core processors and GPUs.

To summarize, we presented an approach for *effective* computing of large irregular scientific problems such as unstructured mesh generation. We showed that out-of-core computing allows solving larger than otherwise possible problems as well as getting the results faster on shared computing resources. We designed, implemented and evaluated the MRTS, which permits out-of-core computing with many application by simply porting an existing or developing a new applications for the MRTS. While porting and development are greatly simplified performance is not sacrificed.

# Bibliography

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] Christos D. Antonopoulos, Filip Blagojevic, Andrey N. Chernikov, Nikos P. Chrisochoides, and Dimitris S. Nikolopoulos. A multigrain Delaunay mesh generation method for multicore SMT-based architectures. *Journal on Parallel and Distributed Computing*, 69:589–600, 2009.

[3] Apple Inc. Grand Central Dispatch (GCD) Reference, 2009.

[4] Eshrat Arjomandi, William G. O'Farrell, Ivan Kalas, Gita Koblents, Frank Ch. Eigler, and Guang R. Gao. ABC++: Concurrency by inheritance in c++. *IBM Systems Journal*, 34(1):120–137, 1995.

[5] Kevin Barker, Andrey Chernikov, Nikos Chrisochoides, and Keshav Pingali. A load balancing framework for adaptive and asynchronous applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, February 2004.

[6] Kevin Barker, Andrey Chernikov, Nikos Chrisochoides, and Keshav Pingali. A load balancing framework for adaptive and asynchronous applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, February 2004.

[7] Kevin Barker and Nikos Chrisochoides. Practical performance model for optimizing dynamic load balancing of adaptive applications. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 28.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Adrian Bowyer. Computing Dirichlet tesselations. *Computer Journal*, 24:162–166, 1981.

[9] Chialin Chang, Joel Saltz, and Alan Sussman. Chaos++: A runtime library for supporting distributed dynamic data structures. Technical report, Gregory V. Wilson, Editor, Parallel Programming Using C, 1995.

[10] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming

on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park AZ USA, 1989.

[11] ANDREY CHERNIKOV AND NIKOS CHRISOCHOIDES. Three-dimensional semi-generalized point placement method for delaunay mesh refinement. In *16th International Meshing Roundtable*, pages 25–44, Seattle, WA, October 2007.

[12] ANDREY CHERNIKOV AND NIKOS CHRISOCHOIDES. Algorithm 872: Parallel 2d constrained delaunay mesh generation. *ACM Transactions on Mathematical Software*, 34:6–25, January 2008.

[13] ANDREY CHERNIKOV AND NIKOS CHRISOCHOIDES. Three-dimensional delaunay refinement for multi-core processors. In *22nd ACM International Conference on Supercomputing*, pages 214–224, Island of Kos, Greece, June 2008.

[14] ANDREY CHERNIKOV AND NIKOS CHRISOCHOIDES. Generalized two-dimensional delaunay mesh refinement. *SIAM Journal on Scientific Computing*, 31:3387–3403, 2009.

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

[16] ANDREY N. CHERNIKOV AND NIKOS P. CHRISOCHOIDES. Parallel 2D graded guaranteed quality Delaunay mesh refinement. In *Proceedings of the 14th International Meshing Roundtable*, pages 505–517. Springer, September 2005.

[17] ANDREY N. CHERNIKOV AND NIKOS P. CHRISOCHOIDES. Parallel guaranteed quality Delaunay uniform mesh refinement. *SIAM Journal on Scientific Computing*, 28:1907–1926, 2006.

[18] ANDREY N. CHERNIKOV AND NIKOS P. CHRISOCHOIDES. Algorithm 872: Parallel 2D constrained Delaunay mesh generation. *ACM Transactions on Mathematical Software*, 34(1):1–20, January 2008.

[19] JAEYOUNG CHOI AND J. J. DONGARRA. Scalable linear algebra software libraries for distributed memory concurrent computers. In *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, FTDCS '95, pages 170–, Washington, DC, USA, 1995. IEEE Computer Society.

[20] EDUARDO F. D'AZEVEDO AND JACK DONGARRA. 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, 2000.

[21] FRANK DEHNE, WOLFGANG DITTRICH, AND DAVID HUTCHINSON. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *In Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

[22] James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. Prospectus for the development of a linear algebra library for high-performance computers. Technical Report ANL/MCS-TM-97, 9700 South Cass Avenue, Argonne, IL 60439-4801, USA, 1987.

[23] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[24] A. Fedorov and N. Chrisochoides. Location management in object-based distributed computing. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 299–308, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Andriy Fedorov and Nikos Chrisochoides. Communication support for dynamic load balancing of irregular adaptive applications. In *2004 International conference on parallel processing workshops (ICPPW'04)*, pages 555–562, 2004.

[26] Paul-Louis George and Houman Borouchaki. *Delaunay Triangulation and Meshing. Application to Finite Elements.* HERMES, 1998.

[27] Intel Corporation. Intel(R) Threading Building Blocks Reference Manual, 2008.

[28] Jaakko Järvi and John Freeman. Lambda functions for c++0x. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 178–183, New York, NY, USA, 2008. ACM.

[29] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

[30] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[31] Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. Effective out-of-core parallel delaunay mesh refinement using off-the-shelf software. *ACM Journal on Experimental Algorithmics.* In press.

[32] Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. Out-of-core parallel delaunay mesh generation. In *IMACS World Congress Scientific Computation, Applied Mathematics and Simulation*, number 17, July 2005.

[33] Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. Parallel out-of-core delaunay refinement. In *IEEE Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, pages 183–195, Sophia, Bulgaria, September 2005.

[34] Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. Effective out-of-core parallel delaunay mesh refinement using off-the-shelf software. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 125–125, Washington, DC, USA, 2006. IEEE Computer Society.

[35] ANDRIY KOT, ANDREY CHERNIKOV, AND NIKOS CHRISOCHOIDES. The evaluation of an effective out-of-core run-time system in the context of parallel mesh generation. In *IEEE International Parallel and Distributed Processing Symposium*, 2011. To appear.

[36] LEONIDAS LINARDAKIS AND NIKOS CHRISOCHOIDES. Graded delaunay decoupling method for parallel guaranteed quality planar mesh generation. *SIAM Journal on Scientific Computing*, 30:1875–1891, March 2008.

[37] J. NIEPLOCHA, V. TIPPARAJU, M. KRISHNAN, AND D. K. PANDA. High performance remote memory access communication: The armci approach. *International Journal of High Performance Computing Applications*, 20(2):233–253, Summer 2006.

[38] JAREK NIEPLOCHA, BRUCE PALMER, MANOJKUMAR KRISHNAN, HAROLD TREASE, AND EDOARDO APR. Advances, applications and performance of the global arrays shared memory programming toolkit. *Intern. J. High Perf. Comp. Applications*, 20, 2005.

[39] MARK H. NODINE AND JEFFREY SCOTT VITTER. Greed sort: optimal deterministic sorting on parallel disks. *J. ACM*, 42(4):919–933, 1995.

[40] JOHN SALMON AND MICHAEL WARREN. Parallel out-of-core methods for N-body simulation. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[41] JONATHAN RICHARD SHEWCHUK. *Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, 1997.

[42] S. TOLEDO AND F. GUSTAVSON. 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*, pages 28–40, 1996.

[43] TIANKAI TU AND DAVID R. O'HALLARON. A computational database system for generatinn unstructured hexahedral meshes with billions of elements. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2004. IEEE Computer Society.

[44] TIANKAI TU AND DAVID R. O'HALLARON. Extracting hexahedral mesh structures from balanced linear octrees. In *Proceedings of the 13th INternational Meshing Roundtable*, pages 191–200, 2005.

[45] JEFFREY SCOTT VITTER AND MARK H. NODINE. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:107–114, 1993.

[46] JEFFREY SCOTT VITTER AND ELIZABETH A. M. SHRIVER. Algorithms for parallel memory ii: Hierarchical multilevel memories. *ALGORITHMICA*, 12:148–169, 1993.

[47] Jeffrey Scott Vitter, Elizabeth A. M. Shriver, and Elizabeth A. M. Shriver Z. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12:110–147, 1994.

[48] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Int. Symp. on Comp. Arch.*, pages 256–266. ACM Press, May 1992.

[49] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. volume 20, pages 256–266, New York, NY, USA, April 1992. ACM.

[50] David F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *Computer Journal*, 24:167–172, 1981.

# VITA

## Andriy Kot

Andriy Kot was born in Ternopil, Ukraine in 1981. From 1997 through 2003 he studied in the Ternopil Academy of National Economy in Ternopil, Ukraine where he received Master's (2003) degrees in Computer Engineering with "red diploma" distinction. His Master's thesis was on the information security systems, incorporating both software and hardware components.

In Fall of 2001 Andriy Kot started studying in the Department of Computer Science at the College of William and Mary where he received his Master's degree in Computer Science. His Master's project was on out-of-core computing and parallel run-time systems.

Since Spring of 2004 he is pursuing a Ph.D. degree in the Department of Computer Science at the College of William and Mary, where he is currently a Ph.D. candidate and a research assistant.