Towards Exascale Parallel Delaunay Mesh Generation*

Nikos Chrisochoides, Andrey Chernikov, Andriy Fedorov, Andriy Kot, Leonidas Linardakis, and Panagiotis Foteinos

Center for Real-Time Computing
The College of William and Mary
Williamsburg, VA 23185
Email: {nikos, ancher, fedorov, kot, leonl01, pfot}@cs.wm.edu

Summary. Mesh generation is a critical component for many (bio-)engineering applications. However, *parallel* mesh generation codes, which are essential for these applications to take the fullest advantage of the high-end computing platforms, belong to the broader class of adaptive and irregular problems, and are among the most complex, challenging, and labor intensive to develop and maintain. As a result, parallel mesh generation is one of the last applications to be installed on new parallel architectures. In this paper we present a way to remedy this problem for new highly-scalable architectures. We present a multi-layered tetrahedral/triangular mesh generation approach capable of delivering and sustaining close to 10^{18} of concurrent work units. We achieve this by leveraging concurrency at different granularity levels using a hybrid algorithm, and by carefully matching these levels to the hierarchy of the hardware architecture. This paper makes two contributions: (1) a new evolutionary path for developing multi-layered parallel mesh generation codes capable of increasing the concurrency of the state-of-the-art parallel mesh generation methods by at least 10 orders of magnitude and (2) a new abstraction for multi-layered runtime systems that target parallel mesh generation codes, to efficiently orchestrate intra- and inter-layer data movement and load balancing for current and emerging multi-layered architectures with deep memory and network hierarchies.

1 Introduction

The complexity of programming adaptive and irregular applications on architectures with hierarchical communication networks of processors is an order of magnitude higher than on sequential machines, even for parallel mesh generation algorithms/codes which can be mapped directly on multi-layered architectures. Automatically exploiting concurrency for irregular and adaptive

*This material is based upon work supported by the National Science Foundation under Grants No. CCF-0833081, CSR-0719929, and CCS-0750901 and by the John Simon Guggenheim Foundation.

computation like Delaunay mesh generation is more complex than exploiting concurrency for regular (or array-based) and non-adaptive computations. Static analysis can not be used for adaptive and irregular applications like parallel mesh generation [28]. In [1, 34] we introduced a speculative (or optimistic) method for parallel Delaunay mesh generation which was recently adopted by the parallel compilers community [29, 36] to study abstractions for parallelization of adaptive and irregular applications. This technique has two major problems for high-end computing: (1) although it works reasonably well for the shared memory model, it is communication intensive for distributed memory machines; and (2) its concurrency can be limited by the problem size at the faster (and thus smaller) shared memory layer of the hierarchy.

In this paper we address both problems using a hybrid multi-layer approach which is based on a decoupled approach [30] at the larger (and slower) layers, an extension of an out-of-core weakly coupled method [26, 27] at the intermediate layers, and a speculative or optimistic but tightly-coupled method [1] at the faster (shared memory) layers (i.e., multi-core). The out-of-core layer utilizes additional disk storage and makes it possible to free the main memory for the storage of data used only in the current computation. In addition, we extend our runtime system [3] to efficiently manage both intra- and inter-layer communication in the context of data migration due to load balancing and migration of data/tasks between layers and between nodes across the same layer.

We expect that this paper can have an impact in two different areas: (1) *Mesh Generation:* we present the first highly scalable parallel mesh generation method capable to provide and sustain concurrency on the order of 10^{18}. (2) *Engineering Applications:* for the first time we provide unprecedented scalability for large-scale field solvers for applications like the direct numerical simulations of turbulence in cylinder flows with very large Reynolds numbers [18] and coastal ocean modeling for predicting storm surge and beach erosion in real-time [44]. In these applications three-dimensional simulations are conducted using two-dimensional meshes in the xy-plane which are replicated in the z-direction in the case of cylinder flows or using bathe-metric contours in the case of coastal ocean modeling. In addition, this method can be extended for Advancing Front Techniques. The approach we develop is independent of the geometric dimension (2D or 3D) of the mesh. Although the mesh-generation-specific domain decomposition has been developed only for 2D, a similar argument applies to 3D with the use of alternative decompositions, e.g., graph partitioning implemented in the Zoltan package [16].

This paper is organized as follows. In Section 2 we review the related prior work. In Section 3 we describe the organization of our Multi-Layered Runtime System. In Section 4 we present the proposed Multi-Layered Parallel Mesh Generation algorithm. In Section 5 we put the runtime system and the parallel mesh generation algorithm together. Section 5.1 contains our preliminary performance data, and Section 6 concludes the paper.

2 Background

In this section we present an overview of parallel mesh generation approaches related to the method we present in this paper. In addition we review parallel runtime systems related to our runtime system PREMA (Parallel Runtime Environment for Multicomputer Applications) which we extend to handle multi-layered applications.

2.1 Related Work in Parallel Mesh Generation

There are three conceptually different approaches to mesh generation. *Delaunay meshing* methods (see [20] and the references therein) use the Delaunay criterion for point insertion during refinement. *Advancing front meshing* techniques (see e.g. [39]) build the mesh in layers starting from the boundary of the geometry. Some of the advancing front methods use the Delaunay property for point placement, but no theoretical guarantees are usually available. *Adaptive space-tree meshing* (see e.g. [33]) is based on adaptive space subdivision (e.g., adaptive octree, or body-centric cubic lattice), and can be flexible in the definition of the meshed object geometry (e.g., implicit geometry representation). Certain theoretical guarantees on the quality of the mesh created in such a way are provided by some of the methods in this group.

A comprehensive review of parallel mesh generation methods can be found in [14]. In this section we review only those methods related to parallel Delaunay mesh generation. The problem of parallel Delaunay *triangulation* of a specified point set has been solved by Blelloch et al. [4]. A related problem of streaming triangulation of a specified point set was solved by Isenburg et al [21]. In contrast, Delaunay *refinement* algorithms work by inserting additional (so-called Steiner) points into an existing mesh to improve the quality of the elements. In Delaunay mesh refinement, the computation depends on the input geometry and changes as the algorithm progresses. The basic operation is the insertion of a single point which leads to the removal of a poor quality tetrahedron and of several adjacent tetrahedra from the mesh and to the insertion of several new tetrahedra. The new tetrahedra may or may not be of poor quality and, hence, may or may not require further point insertions. We and others have shown that the algorithm eventually terminates after having eliminated all poor quality tetrahedra, and in addition the termination does not depend on the order of processing of poor quality tetrahedra, even though the structure of the final meshes may vary [11, 12, 30]. Therefore, the algorithm guarantees the quality of the elements in the resulting meshes.

The parallelization of Delaunay mesh refinement codes can be achieved by inserting multiple points simultaneously. If the points are far enough from each other, as defined in [11], then the sets of tetrahedra influenced by their insertion are sufficiently separated, and the points can be inserted independently. However, if the points are close, then their insertion needs to be serialized because of possible violations of the validity of the mesh or of the Delaunay property. One way to address this problem is to introduce runtime checks [29, 34] which lead to the overheads due to locking [1] and to rollbacks [34]. Another approach is to decompose the initial geometry [31] and apply decoupled methods [20, 30]. The third approach presented in [8, 9, 11] is to use a judicious way to choose the points for insertion, so that we can guarantee their independence and thus avoid runtime data dependencies and overheads. In [9] we presented a scalable parallel Delaunay refinement algorithm which constructs uniform meshes, i.e., meshes with elements of approximately the same size and in [11] we developed an algorithm for the construction of graded meshes. The work by Kadow and Walkington [23, 24] extended [4, 5] for parallel mesh generation and further eliminated the sequential step for constructing an initial mesh, however, all potential conflicts among concurrently inserted points are resolved sequentially by a dedicated processor [23].

In summary, in parallel Delaunay mesh generation methods we can explore concurrency at three levels of granularity: (i) *coarse-grain* at the subdomain level, (ii) *medium-grain* at the cavity level (this is a common abstraction for many different mesh generation methods), and (iii) *fine-grain* at the element level. The fine-grain can only increase the concurrency by a factor of three or four in two or in three dimensions, respectively. However, a detailed profiling of our codes revealed that up to 24.5% of the cycles is spent on synchronization operations, for both the protection of work-queues and for tagging each triangle upon checking it for inclusion in a cavity. Synchronization is always limited among the two or three threads co-located on the same core, and memory references due to synchronization operations always hit in the cache. However, the massive number of processed triangles results in a high percentage of cumulative synchronization overhead. We will revisit the fine-grain level when there is better hardware support for synchronization.

2.2 Related Work in Parallel Runtime Systems

Because of the irregular and adaptive nature of parallel mesh generation we wish to optimize, we restrict our discussion in this section to software systems which *dynamically* balance application workload and we use the following six important criteria: (1) *Support for data migration.* Migrating processes or threads adds to the complexity of the runtime system, and is often not portable. Migrating data, and thereby implicitly migrating computation is a more portable and simple solution. (2) *Support for explicit message passing.* Message passing is a programming paradigm that developers are familiar with, and the Active Messages [43] communication paradigm we use is a logical extension to that. Explicit message passing is also attractive because it does not hide parallelism from the developer. (3) *Support for a global name-space.* A global name-space is a prerequisite for automatic data migration; applications need the ability to reference data regardless of where it is in the parallel system. (4) *Single-threaded application model for inter-layer interactions.* Presenting the developer with a single-threaded communication model between layers greatly reduces application code complexity and development effort. (5) *Automatic load balancing.* The runtime system should migrate data or computation transparently and without intervention from the application. (6) *Customizable data/load movement/balancing.* It cannot be said that there is a "one size fits all" load balancing algorithm; different algorithms perform well in different circumstances. Therefore, developers need the ability to easily develop and experiment with different application- and machine-specific strategies without the need to modify their application code.

Systems such as the C Region Library (CRL) [22] implement a shared memory model of parallel computing. Parallelism is achieved through accesses to shared regions of virtual memory. The message passing paradigm we employ explicitly presents parallelism to the application. In addition, PREMA does not make use of copies of data objects, removing much of the complexity involved with data consistency and read/write locks. In [17, 42] the authors propose the development of component-based software strategies and data structure neutral interfaces for large-scale scientific applications that involve mesh manipulation tools.

Zoltan [15] and CHARM++ [25] are two systems with similar characteristics to PREMA. Zoltan provides graph-based partitioning algorithms and several geometric load balancing algorithms. Because of the synchronization required during load balancing, Zoltan behaves in much the same way as other stop-and-repartition libraries, whose results are presented in [2]. CHARM++ is built on an underlying language which is a dialect of C++, and provides extensive dynamic load balancing strategies. However, the *pick-and-process* message loop guarantees that entry-point methods execute "sequentially and without interruption" [25]. This may lead to a situation in which coarse-grained work units may delay the reception of load balancing messages, negating their usefulness, as was seen with the single-threaded PREMA results presented in [2]. The Adaptive Large-scale Parallel Simulations (ALPS) library [7] is based on a parallel octree mesh redistribution and targets hexahedral finite elements, while we focus on tetrahedral and triangular elements.

3 Multi-Layered Runtime System

The application we target (parallel mesh generation) naturally lends itself to a hierarchical partitioning of work (specifically: domain, subdomain, independent subdomain region, and cavity). At the first two levels of this hierarchy, we use the concept of *mobile object*, or Mobile Work Unit (MWU), as an abstraction for work partitioning. MWU is a container, which is not attached to a specific processing element, but, as its name suggests, can migrate between address spaces of different nodes. Work processing is facilitated by means of sending *mobile messages*, which are directed to MWUs. As we showed in [3], this abstraction is extremely convenient for the development of mesh genera-

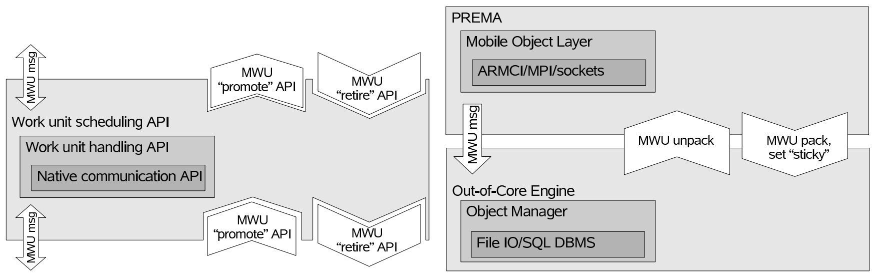

Fig. 1. Left: an abstraction for the hierarchical design of one runtime system layer. The layers are arranged vertically, such that the arrows represent the transfer of data between the adjacent layers. Right: a 2-layer instantiation of the proposed design which we tested using traditional out-of-core parallel mesh generation methods [26, 27].

tion codes, and is indispensable for one of the most challenging problems in parallel mesh generation: dynamic data/load movement/balancing.

Deep memory and network architecture hierarchies are intrinsic to the state-of-the-art High Performance Computing (HPC) systems. Based on our experience, MWU abstraction is effective in handling data movement, work distribution and load-balancing across a single layer of the HPC architecture hierarchy (among the nodes and disk storage units), while large-to-small work subdivision vertically aligns with the hierarchy of the architecture: mesh subdomains, for meshes with over 10^{18} elements, can be too large to fit in memory, while cavities can be processed concurrently at the level of a CPU core at a lower communication/synchronization cost. The objective of the multi-layered runtime system design is to provide communication and flow control support to leverage the hierarchical structure of both the application work partitioning and HPC architecture.

In our previous work on runtime systems we explored various possibilities for the design and the implementation of load-balancing on a Cluster of Workstations (CoW) [3]. In this paper, our design approach is based upon three levels of abstraction, as shown in Fig. 1(left). At the lowest level, there is *native communication* infrastructure, which is the foundation for implementing the concept and basic *MWU handling* routines (migration and MWU-directed communication). Given the ability to create and migrate MWUs, the *scheduling framework* implements high-level logic by monitoring the status of the system and the available objects, and rearranges them accordingly across the processing elements horizontally, or moving them up and down the vertical hierarchy. An important feature of the design is the MWU-directed communication. The life cycle of an MWU is determined by the messages (mostly, work requests) it receives from other MWUs and processing elements, and the status of the system. Depending on its status, availability of work, as well as the degree and nature of concurrency which can be achieved, an MWU can be "retired" to a lower level (characterized by lower degree of concurrency, when no work is pending for MWU, or when there are no resources to keep it at the current layer), or "promoted" to an upper layer (e.g., due to availability of resources or request for fast synchronization due to unresolved dependencies).

As a specific example of how multi-layered design can be realized, we implemented a two-layered framework based on the abstract design presented above (see Fig. 1, right). The top layer is an expanded version of the PREMA system [3]. The native communication can be either one among ARMCI [35], MPI or TCP sockets. The abstraction of mobile work units is realized by MOL [13], and high-level MWU scheduling is determined by the dynamic load-balancing policies implemented within the Implicit Load-balancing Library [3]. Overall, this layer is responsible for the maintenance of a balanced work distribution across a single layer of nodes.

4 Multi-Layered Parallel Mesh Generation

Figure 2 presents the pseudo-code for the multi-layered (hybrid) parallel mesh generation algorithm. It starts with the initial Planar Straight Line Graph (PSLG) $\mathcal{X}$ which defines the domain Ω and the user-defined bounds on circumradius-to-shortest edge length ratio and on the size of the elements. First, we apply a Domain Decomposition procedure [31] to decompose Ω into N non-overlapping subdomains: $\Omega = \bigcup_{i=1}^{N} \Omega_i$ with the corresponding PSLGs $\mathcal{X}_i$, where N is the number of computational clusters. Then the boundary of each Ω_i is discretized using the Parallel Domain Delaunay Decoupling (PD^3) procedure [30] such that subsequent refinement is guaranteed not to introduce any additional points on subdomain boundaries. Next each subdomain represented by $\mathcal{X}_i$ is loaded onto a selected node from cluster i. Then $\{\mathcal{X}_i\}$ are further decomposed using the same method [31] into even smaller subdomains. However, in this case the boundaries of the subdomains are not discretized since PD^3 uses the worst case theoretical bound on the smallest edge length, which generally leads to over-refined meshes in practice. Instead, we use Parallel Constrained Delaunay Meshing (PCDM) algorithm/software [10] which at the cost of some communication introduces points on the boundaries as needed. Specifically, we use its out-of-core implementation (OPCDM) [27]. In addition we take advantage of the shared memory offered by multi-core systems and use the multi-threaded algorithm/implementation we presented in [1]. The meshes produced by the Multithreaded PCDM (MPCDM) algorithm are not constrained by the artificial subdomain boundaries and therefore generally have an even smaller number of elements than the meshes produced by the PD^3 algorithm.

```
ScalableParallelDelaunayMeshGeneration($\mathcal{X}$, $\bar{\rho}$, $\bar{\mathcal{A}}$)
**Input:** $\mathcal{X}$ is the PSLG which defines the domain $\Omega$
          $\bar{\rho}$ is the upper bound on circumradius-to-shortest edge length ratio
          $\bar{\mathcal{A}}$ is the upper bound on element size
**Output:** A distributed Delaunay mesh $\mathcal{M}$ which respects the bounds $\bar{\rho}$ and $\bar{\mathcal{A}}$
 1  Use MADD($\mathcal{X}$, $N$) to decompose the domain into subdomains
       represented by $\{\mathcal{X}_i\}$, $i = 1, \ldots, N$, where $N$ is the number of clusters
 2  Use PD$^3$($\{\mathcal{X}_i\}$, $\bar{\rho}$, $\bar{\mathcal{A}}$), to refine the boundaries of $\mathcal{X}_i$
 3  Load each of the $\mathcal{X}_i$, $i = 1, \ldots, N$, to a node $n_i$ in cluster $i$
 4  do on every node $n_i$ simultaneously
 5     Use MADD($\mathcal{X}_i$, $M_i$) to decompose each subdomain
          into even smaller subdomains $\mathcal{X}_{ij}$, $j = 1, \ldots, M_i$
 6     Distribute the subdomains $\mathcal{X}_{ij}$, $j = 1, \ldots, M_i$, among $P_i$ nodes in cluster $i$
 7     do on every node in cluster $i$ simultaneously
 8        Use OPCDM($\{\mathcal{X}_{ij}\}$, $\bar{\rho}$, $\bar{\mathcal{A}}$) to refine the subdomains
 9     enddo
10  enddo

OPCDM($\{\mathcal{X}_k\}$, $\bar{\rho}$, $\bar{\mathcal{A}}$)
11  Let $Q$ be the set of subdomains that require refinement
12  $Q \leftarrow \{\mathcal{X}_k\}$, $Q_o \leftarrow \emptyset$
13  while $Q \cup Q_o \neq \emptyset$
14     $\mathcal{X} \leftarrow$ Schedule($Q$, $Q_o$)
15     MPCDM($\mathcal{X}$, $\bar{\rho}$, $\bar{\mathcal{A}}$)
16     Update $Q$ (the operation of finding any new subdomains that need
          refinement, e.g., after receiving messages, and inserting them into Q)
17  endwhile

MPCDM($\mathcal{X}$, $\bar{\rho}$, $\bar{\mathcal{A}}$)
18  Construct $\mathcal{M} = (V, T)$ an initial Delaunay triangulation of $\mathcal{X}$
19  Let $PoorTriangles$ be the set of poor quality triangles in $T$
       with respect to $\bar{\rho}$ and $\bar{\mathcal{A}}$
20  while $PoorTriangles \neq \emptyset$
21     Pick $\{t_i\} \subseteq PoorTriangles$
22     do using multiple threads simultaneously
23        Compute the set of Steiner points $P = \{p_i\}$ corresponding to $\{t_i\}$
24        Compute the set of Steiner points $P' \subseteq P$ which encroach upon constrained edges
25        $P \leftarrow P \setminus P'$
26        Replace the points in $P'$ with the corresponding segment midpoints
27        Compute the set of cavities $C = \{\mathcal{C}(p) \mid p \in P \cup P'\}$,
             where $\mathcal{C}(p)$ is the set of triangles whose circumscribed circles include $p$
28        if $C$ create conflicts
29           Discard a subset of $C$ and the corresponding points from $P \cup P'$
                such that there are no conflicts
30        endif
31        BowyerWatson($V$, $T$, $p$), $\forall p \in P \cup P'$
32        RemoteSplitMessage($p$), $\forall p \in P'$
33     enddo
34     Update $PoorTriangles$
35  endwhile

Schedule($Q$, $Q_o$)
36  while $Q \neq \emptyset$
37     $\mathcal{X} \leftarrow$ pop($Q$)
38     if $\mathcal{X}$ is in-core return $\mathcal{X}$   else ScheduleToLoad($\mathcal{X}$), push($Q_o$, $\mathcal{X}$)   endif
39  endwhile
40  $\mathcal{X} \leftarrow$ pop($Q_o$)
41  if $\mathcal{X}$ is in lower-layer or out-of-core Load($\mathcal{X}$) endif
42  return $\mathcal{X}$

BowyerWatson($V$, $T$, $p$)
43     V $\leftarrow V \cup \{p\}$
44     T $\leftarrow T \setminus \mathcal{C}(p) \cup \{(p\xi) \mid \xi \in \partial\mathcal{C}(p)\}$,
          where $(p\xi)$ is the triangle obtained by connecting point $p$ to edge $\xi$
```

Fig. 2. The multi-layered parallel mesh generation algorithm.

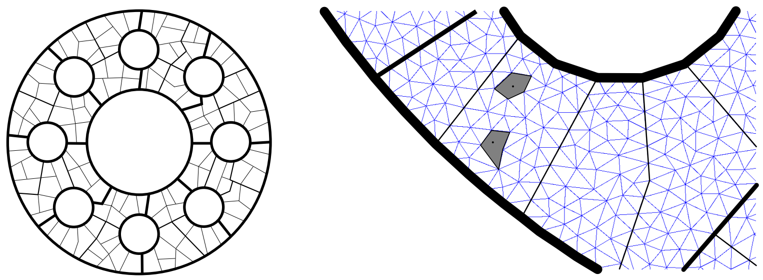

Fig. 3. (Left) Thick lines show the decoupled decomposition of the geometry into 8 high level subdomains which are assigned to different clusters. Medium lines show the boundaries between the subdomains assigned to separate nodes within a cluster. Thin lines show the boundaries between individual subdomains assigned to the same node. **(Right)** Parallel expansion of multiple cavities within a single subdomain using the MPCDM algorithm.

4.1 Domain Decomposition Step

We use the Medial Axis Domain Decomposition (MADD) algorithm/software we presented in [31]. MADD can produce domain decompositions which satisfy the following three basic criteria: (1) The boundary of the subdomains create good angles, i.e., angles no smaller than a given tolerance Φ_o, where the value of Φ_o is determined by the application which uses the domain decomposition. (2) The size of the separator should be relatively small compared to the area of the subdomains. (3) The subdomains should have approximately equal size, area-wise. This approach is well suited for both uniform and graded domain decomposition. Before the subdomains become available for further processing by the PCDM method they are discretized using the pre-processing step from PD^3 [30, 32] which guarantees that any Delaunay algorithm can generate a mesh on each of the subdomains in a way that does not introduce any new points on the boundary of the subdomains (i.e., the algorithm terminates and can guarantee conformity and Delaunay properties without the need to communicate with any of the neighbor subdomains).

4.2 Parallel Delaunay Mesh Generation Step

We use two different approaches, for different layers of the multi-layered architecture: (1) combine a coarse- and medium-grain (speculative-based) approach which is designed to run on a multi-core processor and (2) combine coarse- and coarser-grain which is designed after the traditional out-of-core PCDM method, for a multi-processor node as well as a cluster of nodes. First we describe the in-core PCDM method [10]. The PSLGs for all subdomains are triangulated in parallel using well understood sequential algorithms, e.g., described in [37, 40]. Each triangulated subdomain contains the collections of the constrained edges, the triangles, and the points. For the point insertion, we use the Bowyer-Watson (B-W) algorithm [6, 45]. The constrained (boundary) segments are protected by diametral lenses [38], and each time a segment is encroached, it is split in the middle; as a result, a *split* message is sent to the neighboring subdomain [10]. PCDM is designed to run on multi-processor nodes and clusters of nodes, i.e., it uses the message passing paradigm. Each process lies in its own address space and uses its own copy of a custom memory allocator. Second, the time corresponding to low aggregation decreases as we increase the number of processors; this can be explained by the growth of the utilized network and, consequently, the aggregate bandwidth. Similar studies for new HPC architectures need to be repeated and this parameter will be adjusted accordingly i.e., this parameter is machine specific.

Next we describe the two variations of PCDM we use for the multi-layered algorithm of Figure 2. First, we use the *Out-of-Core (OPCDM) approach (line 8 of the hybrid algorithm)* [27] which utilizes the bottom layer of the HPC architectures, i.e., the processing units with the large storage devices. Before processing a subdomain (using MPCDM) in the main loop we check whether the next subdomain in queue is in-core and mark it as sticky if it is or post a non-blocking load request for that subdomain. Second, after all bad triangles for a subdomain are processed we check whether the next subdomain in queue is in-core. If it is not we push it back in 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 Run-Time System (RTS) marks subdomains with multiple incoming messages as sticky and may attempt to prefetch them. Additionally, when processing incoming messages (when the application is polling), the RTS first executes messages addressed to in-core subdomains regardless of the order in which messages were received (the order of the messages sent to the same subdomain is preserved). The execution order of the subdomains does not affect neither correctness/quality nor termination for our algorithm.

Second, the *Multithreaded (MPCDM) approach (line 15 of the multi-layered algorithm)* [1] which targets the top layer of the HPC architecture, i.e., utilizes the fastest processing unit (hardware supported threads of cores). The threads create and refine individual cavities concurrently, using the B-W algorithm. MPCDM is synchronization-intensive mainly because threads need to tag each triangle while working on a cavity, to detect conflicts during concurrent cavity triangulation. Each subdomain is divided up into distinct areas (in order to minimize conflicts and overheads due to rollbacks), and the refinement of each area is assigned to a single thread. The decomposition is performed by equipartitioning — using straight lines as separators (strip-partitioning) that form a rectangular parallelogram enclosing the subdomain. Despite being straightforward and computationally inexpensive, this type of decomposition can introduce load imbalance between threads for irregular subdomains. The load imbalance can be alleviated by dynamically adjusting the position of the separators at runtime. The size of the queues (private and shared — of triangles that intersect the thread-separator) of bad quality triangles is proportional to the work performed by each thread. Large differences in the populations of queues of different threads at any time during the refinement of a single subdomain are a safe indication of load imbalance. Such events are, thus, used to trigger the load balancing mechanism. Whenever the population of the queues of a thread becomes larger than (100 / Number of Threads)% compared with the population of the queues of a thread processing a neighboring area, the separator between the areas is moved towards the area of the heavily loaded thread.

5 Putting It All Together

In this Section we present the highlights of the implementation for the multi-layered algorithm. The following implementation details are pertinent to the description of the runtime system, which we discussed previously: (1) hierarchical decomposition of work into MWUs, (2) interaction of the algorithm implementation with those units (via run-time system API), and (3) the management of MWUs by the run-time system.

The construction and the registration of the MWUs with the runtime system take place immediately after the decomposition of the input domain in line 5 of the algorithm, see Figure 2. A subdomain has dependencies on the neighboring subdomains, which share a common boundary, and may require coordination in order to process points inserted at that boundary. After the subdomains are defined, their movement, work processing, and communication (i.e., delivery of the Split messages) are handled transparently by the runtime system. The work processing is implemented in two mobile message handlers: subdomain refinement and split point processing subroutines.

We approach the issue of load-balancing across the nodes by using the dynamic load-balancing framework of PREMA [3]. *Intra-layer object migration* is triggered by the imbalance of work assigned to different subdomains due to different levels of refinement, different domain geometry, and, consequently, different rates of split messages arriving at each subdomain. *Inter-layer migration* of the MWUs is required for the efficient memory utilization, and the ability of the given layer to handle larger problem sizes. Scheduling of the MWUs between the PREMA and the OoCS follows the scheme described in the previous Section. The complex issue we will have to resolve, for truly (i.e., greater than two layers of processors) multi-layered architectures like the HTMT Petaflops design [41], is how to handle guaranteed delivery of the mobile messages in the causal order. With current two-layered architectures this is not a problem.

5.1 Preliminary Data

In this Section we report some of the preliminary results for the implementations of the three individual levels of the proposed hybrid algorithm: Domain

Decomposition, Coarse+medium granularity (PCDM) and Coarse+coarser granularity (OPCDM). We evaluated the performance of the Domain Decomposition procedure on the fastest platform we had in our availability (dual Intel Pentium 3.6GHz). For the evaluation of the performance of the upper two levels of the algorithm (coarse+medium and coarse+coarser, i.e., traditional out-of-core) we used a cluster consisting of four IBM OpenPower 720 nodes. The nodes are interconnected via a Gigabit Ethernet network. Each node consists of two 1.6 GHz Power5 processors, which share eight GB of main memory. Each physical processor is a chip multiprocessor (CMP) integrating two cores. Each core, in turn, supports simultaneous multithreading (SMT) and offers two execution contexts. As a result, eight threads can be executed concurrently on each node. The two threads inside each core share a 32 KB, four-way associative L1 data cache and a 64 KB, two-way associative L1 instruction cache. All four threads on a chip share a 1.92 MB, 10-way associative unified L2 cache and a 36 MB 12-way associative off-chip unified L3 cache. The results for each of the three levels are as follows:

Domain Decomposition Given the Chesapeake Bay model, we can sequentially decompose it using MADD into two subdomains in less than 0.5 seconds. This model is defined by 13,524 points and has 26 islands (i.e., quite complex geometry and resolution), see Figure 4. These two subdomains can be distributed to two cores and decomposed in parallel into four subdomains in less than 0.5 seconds. If we continue this way by building a logical binary tree over 10^{12} cores, the model can be decomposed into 10^{12} (or approximately 2^{40}) coarse grain subdomains in less than 40 seconds, assuming that half of this time is spent on communication. All subdomains satisfy the properties required by the Parallel Constrained Delaunay Mesh (PCDM) generation algorithm which we apply on each of these subdomains.

Coarse+medium granularity On the medium grain level, the PCDM method can expose up to 8×10^5 potential concurrent cavity expansions per subdomain [1]. This level of the algorithm was evaluated (see Table 1) on the pipe model, see Figure 3. In each configuration we generate as many triangles as possible, given the available physical memory and the number of MPI processes and threads running on each node. The times reported for parallel PCDM executions include pre-processing time, domain decomposition, MPI bootstrap time, data loading and distribution, and the actual computation (mesh generation) time. We compare the execution time of parallel PCDM with that of the sequential execution of PCDM and with the execution time of Triangle [37], the best known sequential implementation for Delaunay mesh generation which has been heavily optimized and manually fine-tuned. For sequential executions of both PCDM and Triangle the reported time includes data loading and mesh generation time. On a single processor, we can significantly improve the performance attained by using a single core, compared with the coarse-grain only implementation. In the fixed problem size, it proves 29.4% faster than coarse-grain when one MPI process is executed by a single core and 10.2% faster when two MPI processes correspond to each core

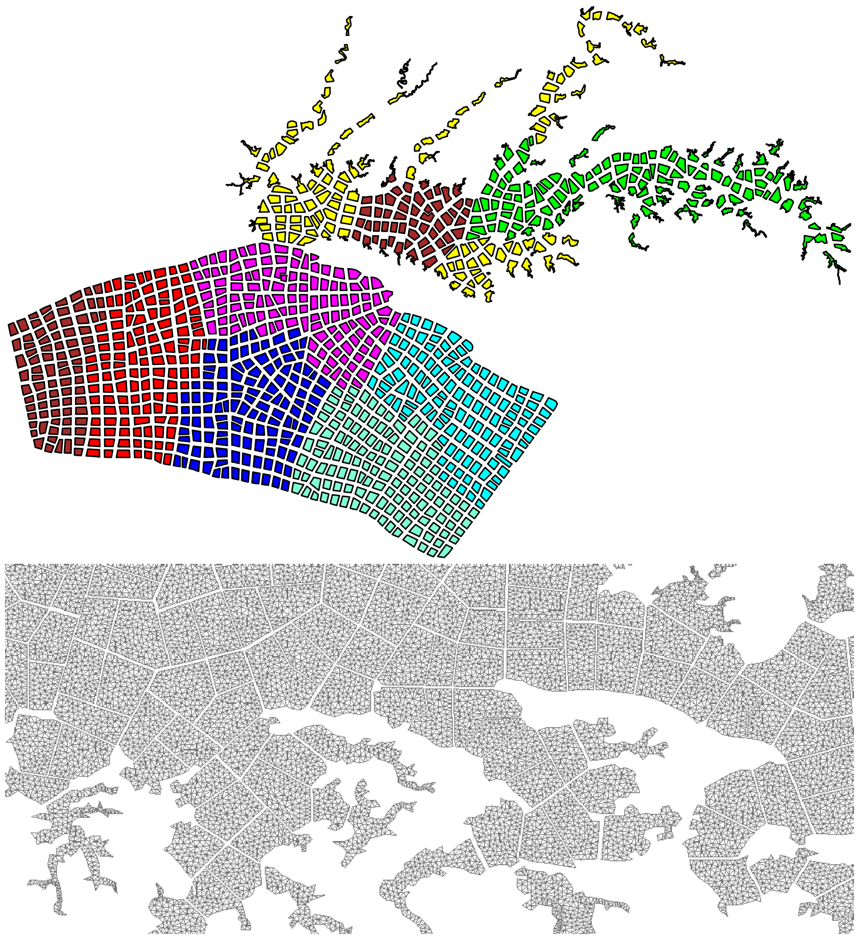

Fig. 4. (Top) The Chesapeake Bay model decomposed into 1024 subdomains that are mapped onto eight clusters of a multi-layered architecture. The assignment of subdomains to clusters is shown with different colors. The use of PD^3 eliminates communication between clusters, however, the use of the multi-layered PCDM in each of the original subdomains requires inter-layer communication and some synchronization at the top level. (Bottom) Part of the Chesapeake Bay model meshed in a way that satisfies conformity and Delaunay properties; thus, correctness and termination can be mathematically guaranteed.

(one per SMT context). In the scaled problem size the corresponding performance improvements are in the order of 31% and 12.7% respectively. Moreover, coarse+medium grain PCDM outperforms on a single core the optimized, sequential Triangle by 15.1% and 13.7% for the fixed and scaled problem sizes respectively. On the fine grain level, the element-level concurrency allows us to process three or four elements concurrently (in 2D and 3D respectively), bringing the total potential concurrency to over 10^{18}.

Cores	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
Triangle Fixed	114.7								
Coarse Fixed	124.1	63.8	32.5	23.3	18.0	14.6	12.8	10.8	10.7
Coarse Fixed (2/Core)	97.4	49.0	21.2	16.3	12.2	10.1	9.1	7.9	8.3
Coarse+Medium Fixed	87.5	44.7	22.8	16.7	12.9	10.6	9.4	9.1	8.0
Triangle Scaled	28.4								
Coarse Scaled	31.0	32.2	32.5	35.6	37.1	36.6	38.3	37.6	41.8
Coarse Scaled (2/Core)	24.5	25.0	21.3	24.5	24.2	24.3	25.5	28.3	28.1
Coarse+Medium Scaled	21.4	22.5	22.8	25.5	26.7	27.1	27.8	29.9	30.4

Table 1. Execution times (in sec.) of the coarse grain and the coarse+medium grain PCDM in 2D on a cluster of four IBM OpenPower 720 nodes. As a sequential reference we use either the single-thread execution time of PCDM or the execution time of the best known sequential mesher (Triangle). Triangle quality in all tests is fixed to 20° degrees minimum angle bound. We present coarse-grain PCDM results using either one MPI process per core (`Coarse`) or one MPI process per SMT execution context (`Coarse (2/core)`). 60M triangles are created in the fixed problem size experiments. 15M triangles correspond to each processor core in the scaled problem size experiments.

Coarse+coarser granularity Our evaluation (see Table 2) demonstrated that OPCDM is an effective solution for solving very large problems on computational resources with limited physical memory. We are able to generate meshes that otherwise would require 10 times the number of nodes using in-core implementation. The performance of the implementation was evaluated in 2D in terms of mesh generation speed[2]. We define per-processor mesh generation (normalized) speed as the average number of elements generated by a single processor over a unit time period, and it is given by $V = \frac{N}{T \times P}$, N is the number of elements generated, P is the number of processors in the configuration and T is execution time. We observe that the overhead introduced by the out-of-core functionality is not large: the per-processor mesh generation speed is only 33% slower for the meshes that fit completely in-core. At

[2]To date, there is no agreed upon standard to evaluate the performance of out-of-core parallel mesh generation codes. The existing metrics for in-core parallel algorithms are not sufficient for this task.

the same time, for the cases when we do use out-of-core functionality, up to 82% of disk I/O is overlapped with the computation.

Mesh size, $\times 10^6$ triangles	number of nodes	Normalized speed, $\times 10^3$ triangles per second		
		PCDM	OPCDM(d)	OPCDM(b)
158.25	8(1)	242.45	156.22	160.11
316.50	16(2)	240.54	160.20	165.06
633.07	32(4)	239.82	157.67	161.08

Table 2. Normalized speed (on a cluster of 4 IBM OpenPower 720 nodes) of the PCDM in 2D 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) refer to the experiments performed with the disk object manager and the database object manager respectively.

6 Conclusions

We presented a multi-layered mesh generation algorithm capable to quickly generate and sustain in the order of 10^{18} of concurrent work units with granularity large enough to amortize overhead for hardware threads on current multi-threaded architectures. In addition we presented a multi-layered communication abstraction and its implementation on current 2-layered multi-core architectures. We used the resulting runtime system to implement a multi-layered parallel mesh generation code on IBM OpenPower 720 nodes (two-layered HPC architecture). The parallel mesh generation method/software mathematically guarantees termination, correctness, and quality of the elements. The mathematical guarantees are crucial for the size of problems we target, because even a single failure to solve a small subproblem my require the recomputation of the whole problem. Our implementation indicates that: (1) we pay very small overhead to generate very large number of concurrent work units, (2) intra-layer communication overhead is very small [10], (4) very large percentage (more than 80%) of inter-layer communication can be tolerated, (5) synchronization required only at the highest level where there is very fast hardware support, (5) work load balancing can be handled transparently with small overhead [3] at the coarse-grain layer (6) load balancing at the medium-grain layer can be handled easily and with low overhead within the application and (7) our out-of-core subsystem allows us to significantly decrease the processing times due to the reduction of wait-in-queue delays. However, the more complex multi-core and multi-CPU multi-layered designs will demand new hierarchical location management directories and policies, which will be a major future research effort (out of the scope of this paper) related to the system design.

Acknowledgments We thank the anonymous reviewers for detailed comments which helped us improve the manuscript. We thank Professor Harry Wang and Dr. Mac Sisson from Virginia Institute of Marine Science for providing the data for the shoreline of the Chesapeake Bay.

References

1. Christos D. Antonopoulos, Xiaoning Ding, Andrey N. Chernikov, Filip Blagojevic, Dimitris S. Nikolopoulos, and Nikos P. Chrisochoides. Multigrain parallel Delaunay mesh generation: Challenges and opportunities for multithreaded architectures. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 367–376, Cambridge, MA, 2005. ACM Press.
2. K. Barker and N. Chrisochoides. An evalaution of a framework for the dynamic load balancing of highly adaptive and irregular applications. In *Supercomputing Conference*. ACM, November 2003.
3. 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.
4. G. E. Blelloch, J.C. Hardwick, G. L. Miller, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24:243–269, 1999.
5. G. E. Blelloch, G. L. Miller, and D. Talmor. Developing a practical projection-based parallel Delaunay algorithm. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 186–195, Philadelphia, PA, May 1996.
6. Adrian Bowyer. Computing Dirichlet tesselations. *Computer Journal*, 24:162–166, 1981.
7. Carsten Burstedde, Omar Ghattas, Georg Stadler, Tiankai Tu, and Lucas C. Wilcox. Towards adaptive mesh PDE simulations on petascale computers. In *Proceedings of Teragrid*, 2008.
8. 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 Annual International Conference on Supercomputing*, pages 48–57, Malo, France, 2004. ACM Press.
9. Andrey N. Chernikov and Nikos P. Chrisochoides. Parallel guaranteed quality Delaunay uniform mesh refinement. *SIAM Journal on Scientific Computing*, 28:1907–1926, 2006.
10. 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.
11. Andrey N. Chernikov and Nikos P. Chrisochoides. Three-dimensional Delaunay refinement for multi-core processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 214–224, Island of Kos, Greece, 2008. ACM Press.
12. L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report TR89983, Cornell University, Computer Science Department, 1989.
13. N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile object layer: a runtime substrate for parallel adaptive and irregular computations. *Adv. Eng. Softw.*, 31(8-9):621–637, 2000.

14. Nikos P. Chrisochoides. A survey of parallel mesh generation methods. Technical Report BrownSC-2005-09, Brown University, 2005. Also appears as a chapter in Numerical Solution of Partial Differential Equations on Parallel Computers (eds. Are Magnus Bruaset and Aslak Tveito), Springer, 2006.
15. K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of dynamic load-balancing tools for parallel applications. In *Proc. of the Int. Conf. on Supercomputing*, Santa Fe, May 2000.
16. K.D. Devine, E.G. Boman, L.A. Riesen, U.V. Catalyurek, and C. Chevalier. Getting started with zoltan: A short tutorial. In *Proc. of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*, 2009. Also available as Sandia National Labs Tech Report SAND2009-0578C.
17. L. Diachin, A. Bauer, B. Fix, J. Kraftcheck, K. Jansen, X. Luo, M. Miller, C. Ollivier-Gooch, M.S. Shephard, T. Tautges, and H. Trease. Interoperable mesh and geometry tools for advanced petascale simulations. *Journal of Physics: Conference Series*, 78(1):012015, 2007.
18. Suchuan Dong, Didier Lucor, and George Em Karniadakis. Flow past a stationary and moving cylinder: DNS at Re=10,000. In *Proceedings of the 2004 Users Group Conference (DOD_UGC'04)*, pages 88–95, Williamsburg, VA, 2004.
19. Andriy Fedorov and Nikos Chrisochoides. Location management in object-based distributed computing. In *Proceedings of Cluster*, San Diego, California, 2004.
20. Paul-Louis George and Houman Borouchaki. *Delaunay Triangulation and Meshing. Application to Finite Elements.* HERMES, 1998.
21. Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink. Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056, 2006.
22. K. Johnson, M. Kaashoek, and D. Wallach. CRL: High-performance all-software distributed shared memory. In *15th Symp. on OS Prin. (COSP15)*, pages 213–228, December 1995.
23. Clemens Kadow. *Parallel Delaunay Refinement Mesh Generation.* PhD thesis, Carnegie Mellon University, 2004.
24. Clemens Kadow and Noel Walkington. Design of a projection-based parallel Delaunay mesh generation and refinement algorithm. In *4th Symposium on Trends in Unstructured Mesh Generation*, Albuquerque, NM, July 2003. http://www.andrew.cmu.edu/user/sowen/usnccm03/agenda.html.
25. L. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA '93*, pages 91–108, 1993.
26. 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 IEEE International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006. Available online at http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1639361.
27. Andriy Kot, Andrey N. Chernikov, and Nikos P. Chrisochoides. Out-of-core parallel Delaunay mesh generation. In *17th IMACS World Congress Scientific Computation, Applied Mathematics and Simulation*, Paris, France, 2005. Paper T1-R-00-0710.
28. Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *Architectural Support for Programming Languages and Operating Systems*, 2008.
29. Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not.*, 42(6):211–222, 2007.
30. Leonidas Linardakis and Nikos Chrisochoides. Delaunay decoupling method for parallel guaranteed quality planar mesh refinement. *SIAM Journal on Scientific Computing*, 27(4):1394–1423, 2006.

31. Leonidas Linardakis and Nikos Chrisochoides. Algorithm 870: A static geometric medial axis domain decomposition in 2D Euclidean space. *ACM Transactions on Mathematical Software*, 34(1):1–28, 2008.
32. Leonidas Linardakis and Nikos Chrisochoides. Graded Delaunay decoupling method for parallel guaranteed quality planar mesh generation. *SIAM Journal on Scientific Computing*, 30(4):1875–1891, March 2008.
33. Scott A. Mitchell and Stephen A. Vavasis. Quality mesh generation in higher dimensions. *SIAM Journal for Computing*, 29(4):1334–1370, 2000.
34. Démian Nave, Nikos Chrisochoides, and L. Paul Chew. Guaranteed–quality parallel Delaunay refinement for restricted polyhedral domains. In *Proceedings of the 18th ACM Symposium on Computational Geometry*, pages 135–144, Barcelona, Spain, 2002.
35. J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler runtime systems. In *Proceedings RTSPP IPPS/SDP '99*, / 1999. ID: bib:Nieplocha.
36. M. Scott, M. Spear, L. Dalessandro, and V. Marathe. Delaunay triangulation with transactions and barriers. In *Proceedings of 2007 IEEE International Symposium on Workload Characterization*, 2007.
37. Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.
38. Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1–3):21–74, May 2002.
39. J. Shöberl. NETGEN: An advancing front 2d/3d-mesh generator based on abstract rules. *Computing and Visualization in Science*, 1:41–52, 1997.
40. H. Si and K. Gaertner. Meshing piecewise linear complexes by constrained Delaunay tetrahedralizations. In *Proceedings of the 14th International Meshing Roundtable*, pages 147–163, San Diego, CA, September 2005. Springer.
41. Thomas Sterling. A hybrid technology multithreaded computer architecture for petaflops computing, / 1997. TY: STD; CAPSL Technical Memo 01, Jet Propulsion Library, California Institute of Technology, California, jan 1997.
42. A.C. To, W.K. Liu, G.B. Olson, T. Belytschko, W. Chen, M.S. Shephard, Y.W. Chung, R. Ghanem, P.W. Voorhees, D.N. Seidman, C. Wolverton, J.S. Chen, B. Moran, A.J. Freeman, R. Tian, X. Luo, E. Lautenschlager, and A.D. Challoner. Materials integrity in microsystems: a framework for a petascale predictive-science-based multiscale modeling and simulation system. *Computational Mechanics*, 42:485–510, 2008.
43. 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.
44. Roy A. Walters. Coastal ocean models: Two useful finite element methods. *Recent Developments in Physical Oceanographic Modeling: Part II*, 25:775–793, 2005.
45. David F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *Computer Journal*, 24:167–172, 1981.