A Multigrain Delaunay Mesh Generation Method for Multicore SMT-based Architectures

Christos D. Antonopoulos [a] Filip Blagojevic [b]

Andrey N. Chernikov [c,*] Nikos P. Chrisochoides [c]

Dimitrios S. Nikolopoulos [b]

[a] *Department of Computer and Communications Engineering, University of Thessaly, Volos, Greece*

[b] *Department of Computer Science, Virginia Tech, Blacksburg, VA 24061*

[c] *Department of Computer Science, The College of William and Mary Williamsburg, VA 23187*

Preprint submitted to Elsevier *23 December 2008*

Abstract

Given the proliferation of layered, multicore- and SMT-based architectures, it is imperative to deploy and evaluate important, multi-level, scientific computing codes, such as meshing algorithms, on these systems. We focus on Parallel Constrained Delaunay Mesh (PCDM) generation. We exploit coarse-grain parallelism at the subdomain level, medium-grain at the cavity level and fine-grain at the element level. This multi-grain data parallel approach targets clusters built from commercially available SMTs and multicore processors. The exploitation of the coarser degree of granularity facilitates scalability both in terms of execution time and problem size on loosely-coupled clusters. The exploitation of medium-grain parallelism allows performance improvement at the single node level. Our experimental evaluation shows that the first generation of SMT cores is not capable of taking advantage of fine-grain parallelism in PCDM. Many of our experimental findings with PCDM extend to other adaptive and irregular multigrain parallel algorithms as well.

Key words: Parallel, Mesh Generation, Delaunay, Multigrain, Multicore, SMT

* Corresponding author.
Email addresses: cda@inf.uth.gr (Christos D. Antonopoulos), filip@cs.vt.edu (Filip Blagojevic), ancher@cs.wm.edu (Andrey N. Chernikov), nikos@cs.wm.edu (Nikos P. Chrisochoides), dsn@cs.vt.edu (Dimitrios S. Nikolopoulos).
URLs: http://inf-server.inf.uth.gr/∼cda (Christos D. Antonopoulos), http://www.cs.vt.edu/∼filip (Filip Blagojevic), http://www.cs.wm.edu/∼ancher (Andrey N. Chernikov), http://www.cs.wm.edu/∼nikos (Nikos P. Chrisochoides), http://www.cs.vt.edu/∼dsn (Dimitrios S. Nikolopoulos).

1 Introduction

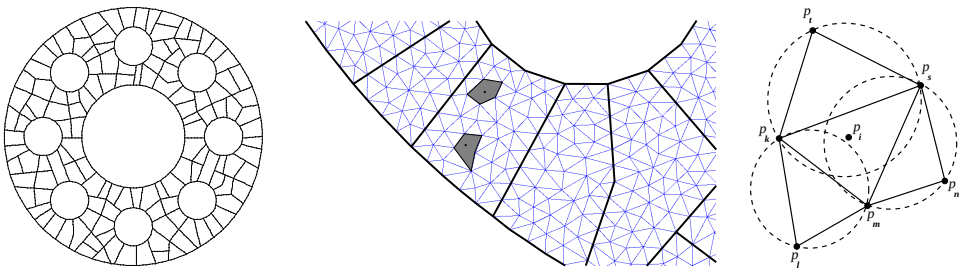
As modern supercomputers integrate more and more processors into a single system, system architects tend to favor layered multiprocessors, since such designs seem to be at the sweet-spot of the cost/performance tradeoff. Most machines in the Top500 list [54] are clusters, often consisting of small-scale SMP nodes. The recent commercial success of simultaneous multithreaded (SMT) processors [41, 56] and multicore processors (CMP) [36] with scalar, superscalar, or SMT cores [33], introduces additional levels in parallel architectures, since more than one threads can co-execute on the same physical processor, sharing some or all of its resources. The efficient exploitation of the functionality offered by these layered architectures introduces new challenges for application developers. Applications that expose multiple levels of parallelism, at different granularities, appear as ideal candidates for the exploitation of the opportunities offered by layered multiprocessors. However, developers have to target both micro-scalability, across the multiple execution contexts of each physical processor, and macro-scalability, across processors or different nodes of the system.

This paper focuses on the design and implementation of parallel mesh generation algorithms and software on such multilevel architectures. Parallel mesh generation is essential in many scientific computing applications in health care, engineering, and science. Such applications often require the creation of meshes with size in the order of billions of elements [34]. Our study provides a macroscopic understanding of the behavior of mesh generation codes on modern parallel architectures. It is a step towards meeting the time and quality constraints set by real-world applications [18]. Moreover, the results of

our study are valid in the context of not only finite element mesh generation methods, but also in the context of other asynchronous multigrain, parallel algorithms.

Parallel mesh generation procedures decompose the original mesh generation problem into smaller subproblems that can be solved (meshed) in parallel. The subproblems can be formulated to be either tightly or partially coupled or even decoupled. The coupling of the subproblems (i.e., the degree of dependency) determines the intensity of the communication and synchronization between processing elements working on different subproblems. The three most widely used techniques for parallel mesh generation are Delaunay, Advancing Front, and Edge Subdivision [25]. In this paper, we use the Delaunay technique because it can mathematically guarantee the quality of the mesh. More specifically, we focus on Constrained Delaunay meshing [15]. The sequential execution time of our parallel implementation (PCDM) is comparable to that of the best to our knowledge sequential implementation [51]. At the same time, PCDM explores concurrency at three levels of granularity: (i) *coarse-grain* at the subdomain level, (ii) *medium-grain* at the cavity level, and (iii) *fine-grain* at the element level. Figure 1 depicts all three levels of granularity, one for each level of parallelization.

In the coarse-grain parallel implementation, the domain is decomposed [39] into $N \gg P$ subdomains, where P is the number of processors (Fig. 1a). $\frac{N}{P}$ subdomains are mapped, using METIS [35], to processors in way that the ratio of interfaces to area is minimized i.e., improve affinity by assigning neighbor subdomains to a single thread, core or processor. In the medium-grain parallel implementation multiple cavities are expanded concurrently by multiple threads. Each thread expands the cavity of a bad-quality triangle. As soon



(a) **(b)** **(c)**

Fig. 1. **(a)** A coarse-grain decomposition into 128 subdomains of a cross-section of a regenerative cooled pipe model. **(b)** Medium-grain parallel expansion of multiple cavities within a single subdomain. **(c)** Fine-grain parallel expansion of a single cavity ($\mathcal{C}(p_i)$) by concurrent testing of multiple triangles ($\triangle(p_k p_l p_m)$, $\triangle(p_m p_n p_s)$, $\triangle(p_s p_t p_k)$).

as each cavity has been calculated, its triangles are deleted and the cavity is retriangulated. In order to preserve the conformity of the mesh, the algorithm has to ensure that there are no conflicts between concurrently expanded cavities. In other words, concurrently expanded cavities are not allowed to share triangles. Finally, in the fine-grain parallel implementation multiple threads work together for a single cavity expansion and thus the degree of parallelism is limited to two, for 2-dimensions and four, for 3-dimensions.

We investigate parallelization approaches for mapping algorithmic multi-grain concurrency to different parallel execution layers in hardware, for clusters built from: (1) conventional, single-thread, single-core processors, and (2) clusters built from SMP nodes with commercially available layered CMP/SMT processors. Our experimental evaluation shows that the coarse-grain, MPI-based approach proves scalable across large numbers of loosely coupled cluster nodes, across different processors within each node, and —under certain

configurations— even across cores within each processor. The coarse granularity of PCDM is an effective means of either reducing execution time, for time-sensitive applications, or of increasing the maximum problem size that can be tackled, by facilitating the exploitation of the total memory and processing power available on multiple nodes. The medium-grain of concurrency in PCDM offers a high degree of parallelism, at an exploitable granularity by today's SMT processors. It allows the effective use of SMT contexts and results in a reduction of execution time on a single core. Finally, we find that the fine-grain parallelism in PCDM is not exploitable on top of the SMTs we use in this study, due to synchronization overhead and lack of hardware support for light-weight thread management.

This paper identifies conditions under which a multilevel, multigrain, parallel mesh generation code can effectively exploit the performance potential of current and emerging multithreaded architectures. Our study also raises the level of understanding for the limitations present when developing efficient parallel algorithms and software for asynchronous, adaptive and irregular applications on current and emerging multilevel parallel architectures.

The rest of the paper is organized as follows. Section 2 discusses previous work. In Section 3 we describe the sequential Delaunay meshing algorithm. Section 4 discusses the parallel multi-level and multi-granular PCDM algorithm. In Section 5 we present the implementation and evaluation of the coarse-, medium-, and fine-grain approaches on real systems. Finally, Section 6 summarizes the paper.

2 Related Work

In this section, we provide a brief coverage of related work on parallel mesh generation and on innovative parallel architectures that enable single-chip multithreaded execution, either across multiple cores, or within a single core.

2.1 Parallel Mesh Generation

In [20] we presented an exhaustive review of parallel mesh generation methods. In this section we focus only on parallel methods which are directly related to PCDM.

The coarse-grain parallel mesh generation method we study in this paper is weakly coupled at its outer level of parallelism, i.e., asynchronous with small messages, and exhibits low communication costs. This method is among the four different parallel mesh generation classes of methods that were developed in our group: (1) tightly coupled [19, 45], (2) partially coupled [11–13], (3) weakly coupled [14, 17], and (4) decoupled [38, 39].

In [11, 13] we presented a theoretical framework and the experimental evaluation of a partially coupled parallel Delaunay refinement (PDR) algorithm for the construction of the uniform guaranteed quality Delaunay meshes. We then extended the PDR approach [10, 12] for the non-uniform case, when the element size is controlled by a user-defined function. The non-uniform PDR algorithm has been implemented on shared memory and offers the possibility to vary the granularity of the refinement units, however its experimental evaluation is still in progress. In the current study, we focus on the weakly coupled

PCDM method [14] with the goal of exploiting multiple levels of parallelism and mapping them efficiently on emerging multilevel parallel architectures.

In [38, 39] we presented the Parallel Delaunay Domain Decoupling (PD^3) method. PD^3 completely decouples the individual subdomains (subproblems), so that they can be meshed independently with no communication and synchronization. However, the construction of decompositions that can properly decouple the mesh is a very challenging problem. Its solution is based on Medial Axis [21, 28, 50] which is expensive and difficult to construct or even to approximate, especially for complex 3-dimensional geometries.

In [6] Blelloch, *et al.* describe a divide-and-conquer projection-based algorithm for constructing Delaunay triangulations of pre-defined point sets in parallel. Walkington and Kadow [32] extended the parallel triangulation method [6] for parallel mesh generation. In contrast to earlier proposed algorithms, the one by Walkington and Kadow further eliminates the sequential step for the generation of an initial mesh.

Besides the Delaunay-based algorithms, a number of other parallel mesh generation algorithms have been published. De Cougny, Shephard, and Ozturan [22] base the parallel mesh construction on an underlying octree. Globisch [26, 27], Löhner and Cebral [40], Chernikov et al. [9], and Ito et al. [30] developed parallel advancing front schemes.

Most of the existing parallel mesh generation methods [7, 17, 22, 24, 31, 32, 38, 44, 47, 49, 58] use only a coarse-grain approach. In [46], however, the authors evaluate three non-Delaunay single-grain approaches based on either a coarse-grain algorithm using the MPI programming paradigm or fine-grain shared-memory algorithms for ccNUMA and multithreading. The fine-grain approach

uses: (i) coloring of triangles, (ii) low-level locks instead of element coloring, or (iii) a combination (hybrid approach) of edge-coloring and low-level locks. Coloring approaches for Delaunay mesh generation methods are computationally expensive, because they require the computation of the cavity graph [1] each time a set of independent points (or cavities) are inserted (or triangulated). Our multigrain approach is based on a coarse-grain, weakly coupled algorithm, in order to achieve scalability at the node level and finer-grain, tightly-coupled approaches in order to explore concurrency at the chip level. The concurrency at the chip level is used to improve the single processor performance of PCDM and outperform —on a single physical processor— state-of-the-art sequential Delaunay mesh generation software [51].

2.2 Deep Parallel Architectures

Conventional wisdom holds that traditional superscalar architectures are facing diminishing returns in terms of performance, power and temperature dissipation. Architectures with inherent thread-level parallelism, such as fine-grain multithreaded processors, simultaneous multithreaded processors [41, 55], multicore designs [5] and layered (e.g. multi-SMT-core designs) [33, 36] are almost unanimously considered as the answer to the limitations of superscalar processors. Thread-level parallel architectures are natural building blocks for layered parallel systems, in which the programmer can exploit multiple levels of parallelism simultaneously, given appropriate algorithmic and system software support. An investigation of parallelization and optimization strategies of ir-

[1] In the cavity graph each cavity is represented by a vertex and two adjacent cavities represent an edge.

regular adaptive computations on layered parallel architectures is a primary contribution of this work.

To our knowledge, besides the prequel to this paper [3], there is no work in the literature investigating the interaction between irregular adaptive parallel applications and emerging multicore and multithreaded architectures. Earlier work [46] has indicated that the Tera MTA, a fine-grain multithreading architecture which uses 128 concurrent instruction streams to mask memory latency was very well suited for fine-grain parallelization and scaling of unstructured applications. Architectures such as the MTA compete in the supercomputing arena against more conventional designs with fewer execution contexts, such as the currently available CMPs and SMTs. As CMPs and SMTs progressively become commodity components, they may gain an edge as the processor of choice over more aggressive designs such as the Tera, or the more recent Cyclops chip [1]. The work in this paper indicates that conventional parallelization strategies using a single-grain approach, such as the directive and lock-based approach used in the Tera MTA [46], can not harness the power of current multithreaded architectures, while at the same time securing scalability. On the contrary, our findings suggest that more effort should be invested in carefully restructuring algorithms and applications to expose multiple levels and granularities of parallelism, in order to cope better with architectural features such as shared cache hierarchies and contention for execution resources.

3 Sequential Mesh Generation

Let V be a set of points in Ω, and T be a set of triangles whose vertices are in V. Then the triangulation $\mathcal{T}$ of V is said to be *Delaunay* if every triangle's circumdisk does not contain points from V. Delaunay mesh generation has been studied extensively [16, 25, 48, 52, 53]. Among the reasons of the popularity of Delaunay methods are useful optimality properties (e.g., the maximization of the minimal angle) and the amenability to the rigorous mathematical analysis.

Typically, a mesh generation procedure starts with the construction of an initial mesh which conforms to the input vertices and segments, and then refines this mesh until the stopping criteria (bounds) on triangle quality and size are met. Parallel finite element codes require "good" quality of elements. The definition of quality depends on the field solver and varies from code to code. In this paper we use geometric criteria specified by the bounds on triangle area ($\bar{\Delta}$) and on triangle circumradius-to-shortest edge ratio ($\bar{\rho}$), which are required by most of the field solvers. The latter is equivalent to a lower bound on a minimal angle [43, 52], for 2D meshes. Guaranteed quality Delaunay methods insert points (p_i) in the circumcenters of triangles that violate the required qualitative criteria, until there are no such triangles left. We use the Bowyer-Watson algorithm [8, 57] to update the triangulation. Figure 2 outlines the pseudocode of a sequential mesher, based on the Bowyer-Watson kernel. The algorithm deletes triangles that are no longer Delaunay and inserts new triangles that satisfy the Delaunay property. It identifies the set of triangles in the mesh whose circumdisks include the newly inserted point p_i. This set is called a *cavity* ($\mathcal{C}(p_i)$). We will denote $\partial\mathcal{C}(p_i)$ to be the set of external edges, i.e., the set of edges which belong to only one triangle in $\mathcal{C}(p_i)$. The triangles in

```
DelaunayRefinement(𝒳, ℳ, Δ̄, ρ̄)
Input: 𝒳 is a PSLG which defines domain Ω
       ℳ is some Delaunay mesh over Ω, which conforms
        to 𝒳. Δ̄ and ρ̄ are desired upper bounds
Output: A modified Delaunay mesh ℳ which respects
         the bounds Δ̄ and ρ̄
1  Q ← {△ ∈ ℳ | (ρ(t) ≥ ρ̄) ∨ (Δ(t) ≥ Δ̄)}
2  while Q ≠ ∅
3      Let △_s ∈ Q
4      BadTriangleElimination(𝒳, ℳ, △_s)
5      Update Q
6  endwhile

BadTriangleElimination(𝒳, ℳ, △_s)
Input: PSLG 𝒳, current mesh ℳ,
       bad triangle to eliminate △_s
Output: A modified Delaunay mesh ℳ
1  p_i ←Circumcenter(△_s)
2  Find S, a set of segments encroached upon by p_i
3  if S ≠ ∅
4      SplitSegments(S)
5  else
6      𝒞(p_i) = {△_t | Incircle(△_t, p_i)}
7      ℳ ← ℳ \ 𝒞(p_i) ∪
       {△(p_i p_m p_n) | (p_m p_n) ∈ ∂𝒞(p_i)}
8  endif

Incircle(△_t, p_i)
Input: Triangle △_t, point p_i
Output: true iff p_i ∈ ◯(△_t)
1  return (p_i ∈ ◯(△_t))
```

Fig. 2. A sequential Delaunay refinement algorithm. A more detailed description along with segment splitting strategies, which always guarantee termination, can be found in [53].

the cavity of the "bad" quality triangle are deleted and the cavity is retriangulated by connecting the endpoints of external edges of $\mathcal{C}(p_i)$ with the newly inserted point p_i. The Bowyer-Watson algorithm can be written as follows:

$$\begin{aligned} V' &\leftarrow V \cup \{p_i\}, \\ T' &\leftarrow T \setminus \mathcal{C}(p_i) \cup \{\triangle(p_i p_j p_k) \mid e(p_j p_k) \in \partial\mathcal{C}(p_i)\}, \end{aligned} \tag{1}$$

where $\mathcal{M} = (V, T)$ and $\mathcal{M}' = (V', T')$ represent the mesh before and after the

insertion of p_i, respectively.

We use a *Planar Straight Line Graph* (PSLG) [51] to delimit Ω from the rest of the plane. Each segment in the PSLG is considered *constrained* and must appear (possibly as a union of smaller segments) in the final mesh. Sequential Delaunay algorithms treat *constrained* segments differently from triangle edges [48, 52].

During the implementation of PCDM we tried to employ general optimization rules related to memory allocation and floating point operations. The use of standard STL data structures and standard routines for their handling is advantageous in terms of code readability and code reuse. However, it can also introduce significant overhead compared to optimized, application-specific data structures and algorithms. In PCDM, we implemented from scratch the most commonly used data structures (queues, graphs, triangle descriptors), as well as the necessary interface for their management. We also integrated a custom memory manager, which allocates and manages memory pools for the most commonly used data objects of the application. The memory manager performs reap-style allocation in the memory pools, however, at the same time, it minimizes the overhead for reusing freed objects. Moreover, small-scale algorithmic modifications resulted to the reduction of floating point operations on the critical path of mesh generation. Such modifications were often subject to tradeoffs between performance and code complexity. Their effect on performance is studied in detail in [2].

Table 1 summarizes the execution time for a sequential execution of PCDM and Triangle [53] for the same problem size (a final mesh with 60 million triangles) on an IBM OpenPower720 system with 64-bit Power5 processors

	Exec. Time (sec) for 60M triangles	Triangles generation rate (Sequential execution)	Maximum Problem Size (triangles)
PCDM	124.07	484K triangles/sec	96M
Triangle	114.74	523K triangles/sec	65M

Table 1

Execution time for the generation of a mesh of 60 million triangles, triangles generation rate, and maximum size of a mesh that can be generated by a sequential execution of PCDM and Triangle on a 64-bits system with 8 GB of main memory.

clocked at 1.6 GHz and 8 GB of main memory. Moreover, we report the maximum mesh size that can be generated on the specific system by PCDM and Triangle. Despite the fact that PCDM suffers the overhead for the support of parallelism even in its single-threaded execution, its performance is a mere 8.1% worse compared with that of the heavily optimized, strictly sequential Triangle when generating 60 million triangles. At the same time, PCDM proves more memory-efficient. As a result, it can create a 47.7% larger mesh than Triangle using the same amount of physical memory.

Table 1 also provides motivation for the exploitation of parallelism in the context of mesh generation. Triangle manages to generate triangles at a rate of 523K per second. Even this seemingly high rate often proves inadequate for time-critical applications working on highly-detailed meshes. At the same time, mesh generation clearly proves to be a memory-constrained computation. Less than two minutes of computation are enough to fill the memory of a generously equipped (8 GB) system. The exploitation of parallelism tackles both limitations simultaneously. It allows more processors to cooperate for generating triangles with a higher rate and, at the same time, it facilitates the exploitation of physical memory available in multiple compute nodes for the

creation of larger meshes.

4 Parallel Delaunay Mesh Generation

4.1 Coarse-Grain Parallelism

The domain decomposition procedure [39] creates N subdomains, each of which is bounded by edges that define the boundary of the subdomains. The edges and their endpoints that are shared between two subdomains are duplicated. The interfaces (subdomain boundary edges) are treated as constrained segments, i.e., as edges that need to be in the final mesh and can not be deleted. By the definition of constrained Delaunay triangulation, points inserted at one side of interfaces have no effect on triangles at the other side; thus, no synchronization is required during the element creation process. The case when the new point happens to be very close to a constrained edge is treated separately. Following Shewchuk [52], we use diametral lenses to detect if a segment is encroached upon. A segment is said to be *encroached upon* by point p_i if p_i lies inside its diametral lenses. The *diametral lenses* of a segment is the intersection of two disks, whose centers lie on the opposite sides of the segment on each other's boundaries, and whose boundaries intersect in the endpoints of the segment. When a point selected for insertion is found to encroach upon a segment, another point is inserted in the middle of the segment instead. As a result, inter-process communication is tremendously simplified: the only message between processes working on neighboring subdomains is of the form, "split this interface" and is sent when a newly inserted point encroaches upon the interface edge [17]. As an additional optimization,

messages can be aggregated, in order to reduce both network traffic and the overhead due to the activation of message handlers. The number of "split" messages for aggregation into a single communication message has been set to 128 [14] throughout our experimental evaluation. The dynamic load balancing of the coarse-grain approach has been studied and is out of the scope of this paper. It can be handled by libraries, runtime systems and by domain over-decomposition [4].

4.2 Fine-Grain Parallelism

The innermost level of parallelism is exploited by allowing multiple threads to cooperate during the expansion (identification) of a single cavity. Cavity expansions actually account, on average, for 59% of the total execution time of PCDM on a modern, 1.6 GHz Power5 processor. Algorithmically, each expansion is similar to a breadth-first search of a graph [19]. The neighbors of the offending triangle are initially enqueued at the tail of a queue. Each thread dequeues one element from the head of the queue and independently subjects it to the INCIRCLE() test. If the test is successful, i.e., the circumcenter of the offender resides inside the circumcircle of the examined triangle, the neighbors of the examined triangle are also enqueued at the tail of the queue and the triangle is deleted. The expansion code terminates as soon as the queue is found empty. Synchronization is required among threads expanding the same cavity, in order to ensure —for the sake of both performance and for implementation correctness— that each triangle is subjected to the INCIRCLE() test only once.

Table 2 summarizes statistics from the execution of fine-grain PCDM for a benchmark geometry and two input sets from real-world problems: a key, a

	key		pipe		cylinder	
	1M	**10M**	**1M**	**10M**	**1M**	**10M**
Queue Length	2.05	2.06	2.05	2.06	2.05	2.06
Cavity Population	4.83	5.07	4.83	5.07	4.82	5.07

Table 2

Average queue length and average cavity population (in triangles) for three different inputs, when meshes of 1M or 10M triangles are created.

rocket engine pipe (depicted in Fig 1a), and a cylinder structure used for the study of flows at very high Reynolds numbers [23]. For each input set we create two meshes, one consisting of 1 million and one consisting of 10 million triangles. We evaluate the average queue length and cavity population – in terms of triangles – throughout the execution of the algorithm. Both metrics prove to be independent of the input set used [2] . Cavity population increases slightly as we move to finer meshes, with more triangles. The average queue length, on the other hand, is steadily slightly above 2. Since concurrently executing threads work on different elements of the queue, the fine-grain parallelism of PCDM can be fully exploited by SMT processors with 2 execution contexts per physical processor package [3] . We implemented the fine-grain parallel implementation of cavity expansion by using 2 threads per MPI process used for the coarse-grain, domain-level parallelization scheme described in Section 4.1. The 2 threads that share the address space of any given MPI process to per-

[2] The execution time of PCDM also proves fairly independent of the input set. It depends mainly on the number of triangles in the final, refined mesh. Due to space limitations, we will only provide results from the pipe input set throughout the rest of the paper.

[3] For 3-dimensional meshes the average queue length is 3 and thus 3 execution contexts can be used to fully exploit the finer degree of parallelism.

form cavity expansion, are bound, using the `sched_setaffinity()` Linux system call, to the 2 execution contexts of the same SMT processor on our target architecture. This enables faster communication and synchronization between the threads, through the shared L1 and L2 caches of the SMT processor.

The extremely fine granularity of parallelism, with hard to amortize overheads on current architectures, and the limited concurrency exposed at this level motivates the exploration of the medium-grain, optimistic parallelization strategy. The latter increases both the granularity and the concurrency of PCDM within each subdomain.

4.3 Medium-Grain Parallelism

The medium-grain parallelism available in PCDM is exploited by using multiple threads in order to expand multiple cavities at the same time. Each triangle in the mesh is tagged with a flag (`taken`). This flag is set, using an atomic `compare & exchange` operation, whenever the triangle becomes a part of some cavity. If a thread, during a cavity expansion, touches a triangle whose `taken` flag has already been set, the thread detects a conflict and cancels the expansion of the specific cavity.

In order to reduce the number of conflicts, each thread is allowed to process only the triangles that belong to a certain area. The boundaries delimiting different areas are non-constrained straight lines. A suboptimal initial specification of boundaries, especially on irregularly shaped subdomains, may result in load imbalance, should one thread be assigned a larger area to mesh than others. Whenever load imbalance is detected at run-time, our code dynam-

ically compensates by moving the boundaries between neighboring areas to the direction of the most heavily loaded thread [2]. Medium-grain PCDM also benefits from the custom memory manager described in Section 3. In fact, in the case of the medium-grain implementation different threads allocate objects from different memory manager heaps. Apart from the potential spatial locality benefits, this technique also eliminates the pressure onto and the contention inside the memory manager. Such contention can prove a major performance bottleneck for multithreaded applications which, similarly to medium-grain PCDM, are characterized by very frequent object allocations and deallocations.

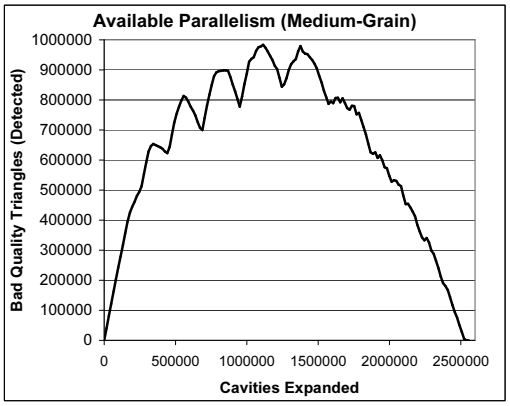


Fig. 3. Generation of a mesh with 10M triangles by medium-grain PCDM. Number of detected, unprocessed "bad quality" triangles throughout the execution life of the application.

Figure 3 depicts the population of triangles that have been detected to violate the quality criteria set by the user, however they have not yet been refined, throughout the execution life of the application (as a function of the number of already expanded cavities). The unrefined triangles count is a good indication of the parallelism of the application. On average, it is close to 634K triangles, thus medium-grain PCDM offers ample parallelism, at a much coarser gran-

ularity —if batches of unrefined triangles are assigned to each thread— than the innermost level of parallelism (fine-grain).

5 Experimental Evaluation on Current and Emerging Parallel Architectures

In the following paragraphs we discuss the exploitability and the mapping of parallelism in PCDM to parallel architectures with heterogeneous characteristics. More specifically, we experiment on:

- A heterogeneous, commodity, off-the-shelf (COTS) Sun cluster. The cluster integrates 64 single-processor nodes based on UltraSPARC IIi+ CPUs, clocked at 650 MHz, with 1 GB main memory each and 32 dual-processor nodes based on UltraSPARC III Cu CPUs, clocked at 900 MHz, with 2 GB of main memory per node. UltraSparcIIi+ processors are equipped with 16 KB direct-mapped L1 data cache, 16 KB 2-way associative L1 instruction cache and 512 KB, 4-way associative L2 Cache. The cache hierarchy of each UltraSparc III Cu processor is slightly more complicated: it consists of 32 KB 4-way associative L1 instruction cache, 64 KB 4-way associative L1 data cache, 2 KB 4-way set associative prefetch cache for software prefetching, 2 KB, 4-way set associative write cache which reduces store latency to L2 cache and finally, 8 MB 2-way associative unified L2 cache. All nodes are interconnected with a 100 Mbps Ethernet network. The experiments on the cluster evaluate the scalability of PCDM when its coarse-grain parallelism is exploited using a message passing programming model (MPI).
- A cluster, consisting of 4 IBM OpenPower 720 nodes. The nodes are interconnected via a Gigabit Ethernet network. Each node consists of 2 Power5

processors clocked at 1.6 GHz, which share 8 GB of main memory. Each physical processor is a chip multiprocessor (CMP) integrating 2 cores. Each core, in turn, supports simultaneous multithreading (SMT) and offers 2 execution contexts. As a result, 8 threads can be executed concurrently on each node. The two threads inside each core share a 32 KB, 4-way associative L1 data cache and a 64 KB, 2-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. We use this system to evaluate both single-level and multilevel/multigrain (coarse+fine or coarse+medium) executions of PCDM.

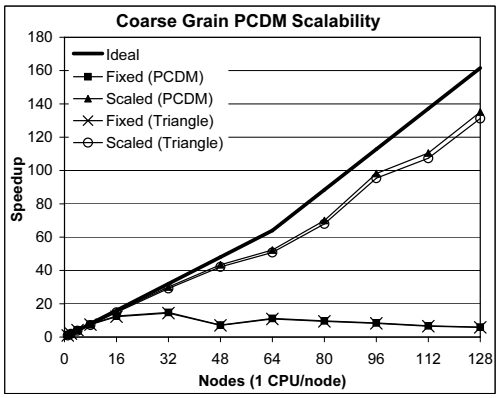
In all experiments we use the pipe model (Fig. 1). In each configuration we generate as many triangles as possible, given the available physical memory and the number of MPI processes and threads running on each node. The times ($T_{par}(W)$) reported for parallel PCDM executions include pre-processing time for the generation of an initial coarse mesh, 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 [51], the best to our knowledge 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. In all cases, we do not report the time required to output the final mesh to disk, since in most real world applications the produced mesh is used directly by a parallel finite element solver, which is executed right at the next step.

We executed coarse-grain PCDM on the COTS Sun cluster, using 1 to 128 MPI processes on 1 to 128 processors respectively. The domain to be meshed is divided up into subdomains, i.e., the meshing problem is divided up into weakly coupled subproblems, so that 32 subdomains correspond to each processor [4]. We performed two sets of experiments. In the first set we execute PCDM on a varying number of processors and produce a fixed size, refined mesh, consisting of 12 million triangles. We calculate the fixed speedup as the ratio $T_{seq}(W)/T_{par}(W)$, where $T_{seq}(W)$ and $T_{par}(W)$ are the sequential and the parallel execution times, respectively, for the specific problem size ($W = 12$ million triangles). We compare the parallel execution time with both the execution time of PCDM on a single processor and the execution time of Triangle. This experiment set focuses on the execution time improvement that can be attained for a specific problem size, by exploiting the coarse-grain parallelism of PCDM.

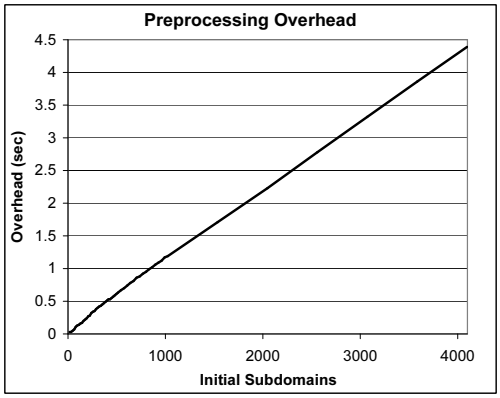
In the second experiment set, we scale the problem size linearly with respect to the number of processors (P). The problem size equals approximately 12M triangles per processor. In other words, the problem size gradually increases from 12 million to 1.536 billion triangles. We now calculate the scaled speedup as the ratio $P \times T_{seq}(W)/T_{par}(P \times W)$. Once again, we use as a reference the sequential execution times of both PCDM and Triangle. The second set outlines the ability of the parallel algorithm to efficiently exploit more than one pro-

[4] Over-decomposition of the domain is necessary for load-balancing reasons. We have experimentally determined 32 subdomains per MPI process to be a good trade-off between decomposition overhead and load balancing.

cessors in order to tackle problem sizes that are out of the reach of sequential algorithms due to both computational power and memory limitations.



(a)



(b)

Procs.	1	2	4	8	16	32	48	64	80	96	112	128
Subdomains	1	64	128	256	512	1024	1536	2048	2560	3072	3584	4096
Triangle	57.6											
PCDM Fixed	59.3	28.4	14.1	8.0	5.3	5.2	9.9	7.5	8.9	10.3	12.7	14.3
PCDM Scaled	59.3	59.6	60.2	64.0	62.6	64.3	67.3	74.8	77.7	71.4	77.5	75.3

Fig. 4. (a) Fixed and scaled speedups of the coarse-grain PCDM on a 128 processor cluster. 12M triangles are created in the fixed problem experiments, and 12M triangles are created by each processor in the scaled problem size experiments. The speedups have been calculated using as a reference either the single-processor execution time of PCDM or the execution time of Triangle. The diagram also depicts the ideal speedup. (b) Preprocessing overhead for the generation of the initial, coarse mesh, as a function of the number of initial subdomains. The table summarizes the corresponding execution times (in sec.). It also reports the number of subdomains used.

Fig. 4a depicts the experimental results from both sets as well as the ideal speedup. The slope of the ideal speedup line changes after 64 processors to account for the fact that processors 64-127 are more powerful (UltraSparc III) than processors 0-63 (UltraSparc IIi+), resulting to 53% faster execution for

both sequential PCDM and Triangle. The ideal speedup has been calculated with respect to the sequential execution on an UltraSparc IIi+ processor. For example, on 64 processors the ideal speedup equals 64, however on 96 and 128 processors, when UltraSparc III processors are also used, its value is 112.8 and 161.5 respectively. Fig. 4b depicts the overhead for the creation of the initial decomposition, as a function of the number of initial subdomains. The table under the diagrams summarizes the total execution times of all experiments. The table also reports the number of subdomains used (32 per processor if more than one processors are available).

In all cases the calculated speedups when using sequential PCDM and Triangle as the basis for the calculation of the speedup are almost indistinguishable, due to the minimal – approximately 3% for the specific problem sizes of 12M triangles (fixed) and 12M triangles per processor (scaled) – performance difference between the sequential executions time of PCDM and Triangle.

The preprocessing overhead ranges between 70 msec for the creation of 64 subdomains (execution with 2 processors) to 4.39 sec for the creation of 4096 subdomains (execution with 128 processors). The overhead is practically linear to the number of subdomains [39].

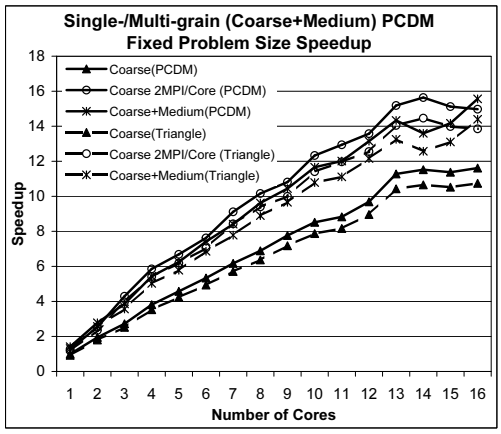
For the fixed problem size experiments, coarse-grain PCDM scales almost linearly on up to 16 processors and keeps slightly scaling up to 32 processors. The wall-clock execution times for the generation of a mesh of 12M triangles on 16 and 32 processors are 5.3 sec. and 5.2 sec., respectively. In fact, on the 2- and 4-processor configurations coarse-grain PCDM attains super-linear speedups, even compared with the heavily optimized Triangle, due to the additional cache memory available to the application when more than 1 processors are

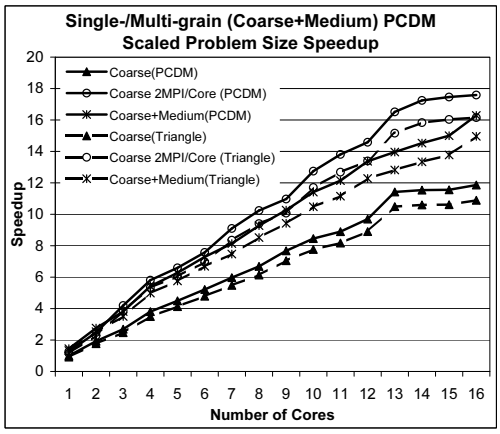
used. In the 32 processor configuration PCDM generates 2.3M triangles/sec. Using more than 32 processors slows down PCDM. On 32 processors the computation corresponding to each processor is already lightweight. The additional computational power made available by using more processors can not, thus, be effectively exploited and does not manage to outweigh overheads such as the generation of additional MPI processes, the sequential generation of more subdomains, and the communication (initial data distribution, messages between neighboring subdomains during the execution, collection of results after the generation of the mesh). For example, in a 32 processors execution the preprocessing overhead alone corresponds to 22.9% of the total execution time.

The scalability potential of coarse-grain PCDM is more evident when the problem size is scaled with the number of processors. The speedup on 128 processors equals 135.13 and 131.26 when PCDM and Triangle are used as a reference point, respectively. On the 128 processor configuration PCDM generates triangles with an average rate of 20.4M triangles/sec. However, these speedups are within 78.8% and 76.5%, respectively, of the ideal expected speedup on the specific system, and the efficiency tends to drop as more processors are used. Once again, as more MPI processes are used, the cost of preprocessing and MPI overheads become an important portion of the total execution time. This observation motivates the exploitation of the other two granularities of parallelism available in PCDM on multi-layered parallel architectures.

Medium-grain PCDM extends the MPI implementation of PCDM to exploit parallelism inside each MPI process. Medium-grain PCDM differs from coarse-grain in that multiple threads are allowed to work on the same subdomain. The threads can independently expand and refine different cavities, however in order to guarantee mesh conformity two cavities processed independently by different threads are not allowed to share a face (triangle in 3D or edge in 2D). Our implementation of medium-grain PCDM includes an extensive set of optimizations, including algorithmic optimizations, modified data structures and synchronization mechanisms to reduce contention, and techniques to reduce conflicts between threads sharing the resources of an SMT processor [2]. The main algorithmic challenge of medium-grain is the detection of and recovery from conflicts, which requires synchronization among threads working on the same subdomain.

Figure 5 depicts the results of the experimental evaluation of the coarse and coarse+medium granularities of PCDM on a cluster of 4 IBM OpenPower 720 nodes, each with two 2-way CMP Power5 processors. Each core is, in turn, a 2-way SMT. We report results on a coarse-grain execution using one MPI process per processor core (`Coarse`). We have also experimented with using two MPI processes per core, i.e., one MPI process per SMT execution context (`Coarse (2/core`)). Finally, we have evaluated the performance of the multi-grain (coarse+medium) approach, in which one coarse-grain MPI process is assigned to each processor core, however two medium-grain threads inside each MPI process cooperate for the refinement of a single subdomain. Each





Cores	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Triangle Fixed	114.7															
Coarse Fixed	124.1	63.8	45.6	32.5	27.1	23.3	20.1	18.0	16.0	14.6	14.1	12.8	11.0	10.8	10.9	10.7
Coarse Fixed (2/Core)	97.4	49.0	28.9	21.2	18.6	16.3	13.6	12.2	11.5	10.1	9.6	9.1	8.2	7.9	8.2	8.3
Coarse+Medium Fixed	87.5	44.7	32.2	22.8	19.9	16.7	14.7	12.9	11.9	10.6	10.3	9.4	8.7	9.1	8.8	8.0
Triangle Scaled	28.4															
Coarse Scaled	31.0	32.2	34.6	32.5	34.5	35.6	36.3	37.1	36.3	36.6	38.3	38.3	35.3	37.6	40.2	41.8
Coarse Scaled (2/Core)	24.5	25.0	22.1	21.3	23.5	24.5	23.8	24.2	25.4	24.3	24.7	25.5	24.4	28.3	26.6	28.1
Coarse+Medium Scaled	21.4	22.5	24.4	22.8	24.7	25.5	26.6	26.7	27.2	27.1	28.1	27.8	28.8	29.9	31.0	30.4

Fig. 5. Fixed and scaled speedups of the coarse grain and the coarse+medium grain PCDM on a cluster of 4 IBM OpenPower 720 nodes. The speedups have been calculated using as a reference either the single-thread execution time of PCDM or the execution time of the best known sequential mesher (Triangle). 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. The table summarizes the corresponding execution times (in sec).

thread is executed by one of the SMT execution contexts of each core. For fixed problem size experiments we generate meshes of 60M triangles. In the scaled problem size experiments we create 15M triangles per used processor

core.

On a single processor, medium+coarse grain PCDM significantly improves 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 (one per SMT context). In the scaled problem size the corresponding performance improvements attained by medium+coarse grain are in the order of 31% and 12.7% respectively. Moreover, medium+coarse 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.

Medium+coarse grain PCDM remains the optimal solution when two cores are available. When more than two cores (i.e., more than one physical processors) are used however, the coarse-grain only approach with two MPI processes per processor (`Coarse (2/core)`) outperforms the multi-grain one. It has been explained in Section 4.3 that medium-grain threads have to perform a synchronization operation every time they touch a triangle. Although our implementation employs low-overhead, non-blocking atomic operations, the frequency of these operations is extremely high —approximately once every 500 nsec. Atomic operations impede memory accesses from other threads running on the same node. This results to performance penalties when more than four PCDM threads execute on each node (two cores, two SMT contexts/core) [5]. It should be noted that the synchronizing threads always reside on the same processor. As a result, should modern processors have adequate support

[5] Up to eight threads can co-execute on each node of our platform.

for efficient inter-processor thread synchronization, the performance effects of synchronization operations should be limited within each physical processor, instead of affecting all threads running on the same node. Unfortunately, this is not the case for commercially available SMT / CMP processors. In any case, a coarse-grain only parallelization has the downside of either not exploiting the second execution context of each processor (`Coarse`), or using twice as many MPI processes on a specific system (`Coarse (2/Core)`). The use of additional MPI processes is, in turn, associated with additional overheads for preprocessing and MPI bootstrapping. As more cores are used, these overheads become significant, in both the fixed and the scaled problem sizes. It is clear, from Figure 5, that both single-level approaches (`Coarse` and `Coarse (2/Core)`) face scalability difficulties for 13 or more cores. The medium+coarse grain PCDM implementation continues attaining performance improvements even for 13 or more cores. In fact it manages to outperform again `Coarse (2/Core)` with 16 cores in the fixed problem size case. In the scaled problem size experiments it closes the performance gap with `Coarse (2/Core)`. In fact the scaled speedup improvement from 14 to 16 cores is considerably higher in the medium+coarse case (12%) than in the coarse-grain only experiments (2%). We expect that if more than 16 cores were available on our experimental platform, the multilevel medium+coarse grain PCDM would outperform the single-level coarse-grain approach for the scaled problem size as well. It should be taken into account that the medium+coarse execution, using 2 threads for the exploitation of the medium grain, requires half the initial subdomains, i.e. half the initial preprocessing overhead. The latter is non-negligible and grows linearly with the number of cores used for the exploitation of the coarse grain (Figure 4(b)). The same observation holds for the MPI startup overhead as well.

In summary, a multi-grain (medium+coarse) approach manages to outperform single-level PCDM, at least in certain areas of the configuration space. It allows the exploitation of SMP contexts inside each processor core without incurring additional preprocessing and MPI startup overhead. It is important to note that medium+coarse grain PCDM offers performance improvements on currently commercially available hardware, without requiring special support from either the OS or the processor. Unfortunately, this is not the case for the coarse+fine multi-grain implementation presented in Section 5.3.

5.3 Fine-Grain: Execution on a SMT-based Multiprocessor

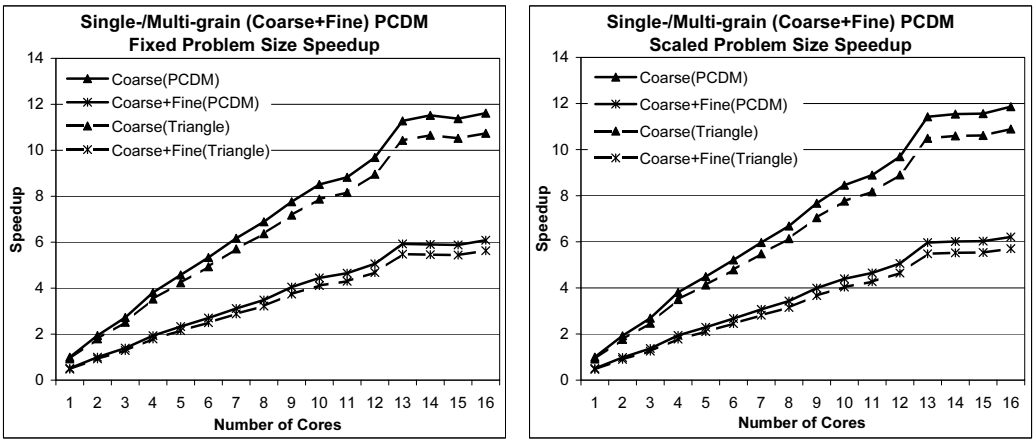
As a next step, we evaluated the performance of a fine+coarse multi-grain PCDM implementation on the same layered, CMP/SMT based multiprocessor. We used the two execution contexts available on each core in order to exploit the finest granularity of parallelism available in PCDM. In other words, both execution contexts work in parallel to expand the cavity of the same, bad-quality triangle. Each execution context accommodates one kernel thread. Kernel threads are created once and persist throughout the life of the application. Each thread is bound to a specific execution context and is not allowed to migrate. The two kernel threads that are bound on the same processor have distinct roles: The *master* thread has the same functionality as the typical MPI process used to process a subdomain of the original domain. The *worker* thread assists the *master* during cavity expansions, however it is idling when the *master* executes code unrelated to cavity expansion.

Apart from the optimizations described earlier (substitution of STL data structures, customized memory manager), we applied only limited, local code

modifications in order to minimize the interaction between the threads executing on the two execution contexts of each processor, thus reducing the contention on shared data structures. More specifically, we substituted the global queue previously used for the breadth first search on the triangles graph with two separate, per execution context queues. As soon as an execution context finishes the processing of a triangle, it attempts to dequeue another, unprocessed triangle from its local queue. If the queue is empty, it attempts to steal a triangle from the queue of the other execution context. Every triangle whose circumcircle includes the circumcenter of the bad-quality triangle (INCIRCLE() test), has to be deleted. Such triangles are pushed in a local, per execution context stack and are deleted in batch, after the cavity is expanded. Their neighbors, which also have to be subjected to the INCIRCLE() test, are enqueued in the local queue of the execution context. Despite the fact that the introduction of per thread queues reduces the interaction between threads, the requirement of work stealing necessitates the use of locks for the protection of queues. The functionality of these locks is similar to the low-level locks proposed in [46]. However, the locks are only contended when both threads access concurrently the same queue, one attempting to access its local queue and the other to steal work. An alternative queue implementation would employ lock-free techniques. However, complex lock-free data structures outperform lock-based ones only in the presence of multiprogramming or when access to shared resources is heavily contended [42]. None of these conditions holds in the case of fine-grain PCDM.

In order to both guarantee implementation correctness and to avoid performing redundant INCIRCLE() tests, it must be ensured that the same triangle is not subjected to the test more than once during a cavity expansion. This is

possible, since up to three paths —one corresponding to each neighbor— may lead to the same triangle during the breadth-first search of the triangles graph. In order to eliminate this possibility, the data structure representing each triangle is extended —similarly to the medium-grain PCDM implementation— with a *taken* flag. Threads check the *taken* flag of triangles before processing them and try to set it using an atomic, non-blocking `test_and_set` operation. If the atomic operation fails or the *taken* flag has already been set, the triangle is discarded and is not subjected to the INCIRCLE() test.



Cores	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Coarse+Fine Fixed	241.3	124.0	89.0	63.9	53.1	45.6	39.6	35.3	30.3	27.5	26.2	24.0	20.4	20.5	20.5	19.8
Coarse+Fine Scaled	61.1	63.1	67.6	63.8	67.3	69.3	70.4	71.9	69.4	70.0	72.9	73.0	67.0	71.6	76.5	79.2

Fig. 6. Fixed and scaled speedups of the coarse+fine grain PCDM on a cluster of 4 IBM OpenPower 720 nodes. We report again the speedups of single-level, coarse–grain PCDM, with 1 MPI process per core (`Coarse`) as a basis for comparison. The speedups have been calculated using as a reference either the single-thread execution time of PCDM or the execution time of Triangle. 60M triangles are created in the fixed problem size experiments. 15M triangles correspond to each processor core in the scaled problem size experiments. The table summarizes the corresponding execution times (in sec).

Figure 6 depicts the results of the experimental evaluation of multi-grain (fine+coarse) PCDM on a cluster of 4 IBM OpenPower 720 nodes. Each of the two fine-grain threads inside an MPI process is executed by one of the SMT execution contexts of each core. We report again, as a basis for comparison, the speedups attained by the single-level, coarse-grain implementation using one MPI process per core (`Coarse`). Similarly to the medium-grain evaluation, for fixed problem size experiments we generate 60M triangles and for scaled problem size experiments we create 15M triangles per processor core used.

Interestingly enough, the exploitation of fine-grain parallelism in PCDM does not improve PCDM performance. On the contrary, it results in a significant slowdown, even compared with the `Coarse` configuration on the same number of cores. The latter does not use the second SMT execution context of each core. The slowdown ranges between 1.9 and 1.98.

The granularity of fine-grain PCDM is too fine to be exploitable by current commercially available hardware. Although cavity expansions account, as explained previously, for 59% of the total execution time, each sequential cavity expansion lasts only between 3.16 and 4.74 μsec on the specific system.

Fine-grain PCDM is characterized by the same high synchronization intensity as the medium-grain implementation. The `taken` flag of each triangle that is touched has to be atomically checked for being `false` and set to `true`. This results in a synchronization operation approximately every 500 nsec. Apart from their own overhead, such operations tend to delay computation-related memory accesses. The overhead due to synchronization operations has been experimentally evaluated to be more than 25% of the execution time on the specific platform. Once again, it should be pointed out that synchronizing

threads always reside on the same core. However, the lack of efficient inter-processor, or, even better, inter-core synchronization support results in the propagation of the effects of synchronization to all threads executing on each node.

Another weakness of the fine-grain approach is that parallelism can be exploited for only 59% of the application's execution time. Together with the limited degree of parallelism available at the fine granularity level (slightly more than 2 for 2D geometries), this reduces the maximum expected speedup on a 2-way SMT core to a mere 1.42. A more significant side-effect, though, has to do with the behavior of the *worker* thread during the remaining 41% of the application's execution life. First generation commercially available SMT processors do not offer hardware support for efficiently suspending/resuming the execution of threads on specific SMT execution contexts. All suspend/resume operations have to be handled through the operating system, thus significantly increasing their overhead and limiting the granularity with which such operations can be used. In any case, threads can not be suspended/resumed with the frequency required by fine-grain PCDM. The only other alternative for the implementation of the *worker* thread is to allow it to spin on its work queue, waiting for work to be distributed by the *master* thread. Threads on an SMT processor, however, share almost all processor resources. Spinning instructions usually have no uncached memory dependencies. They thus achieve a very low CPI and can be issued with a very high frequency, filling processor instruction queues. This, in turn, delays useful, computation-related instructions by the *master* thread. We have experimentally quantified the *master* thread slowdown due to interference from a spinning *worker* thread to be approximately 12%.

Obviously, a fine+coarse multi-grain PCDM execution is not a viable approach

for the optimal exploitation of SMT contexts on a multi-level parallel system, at least not without additional hardware support. In [2, 3] we evaluate the effect of additional hardware support for synchronization and thread management on the performance of fine-grain PCDM. We find that with with minimal additional hardware support for thread creation and fast synchronization through registers, a coarse+fine multi-grain implementation can outperform coarse-grain PCDM, even if in the coarse-grain implementation 2 MPI processes are allowed to execute on each core.

6 Conclusions

Multithreaded processors become more and more widespread and layered parallel systems are being built using these processors. Multilevel, multigrain parallel codes seem like a good vehicle for the exploitation of the performance potential of this class of parallel systems, however their development and mapping to the architecture is not a trivial undertaking. This is especially true for adaptive and irregular applications. Our paper makes contributions towards this direction, focusing on mesh generation algorithms. Fast, high quality mesh generation is a critical module for a multitude of real-world medical and engineering applications. We focused on PCDM, a parallel, guaranteed-quality mesh generator. PCDM exposes parallelism in three granularities.

We exploited the coarsest grain with an MPI-only implementation which proves to scale up to 32 processors for fixed problem sizes and up to the total processors available on our cluster (128) for problem sizes that scale with the number of processors. Moreover, a coarse-grain approach is, under certain configurations, the optimal choice for the exploitation of execution contexts on

SMT processors. Its ability to use more than one processors/cores/execution contexts allows it to solve problems faster than Triangle, the best to our knowledge, hand-optimized, sequential Delaunay mesh generation software. At the same time PCDM can tackle problem sizes which can not be addressed by Triangle, due to processing power and memory limitations. Coarse-grain PCDM managed to generate a mesh of 1.536 billion triangles on 128 processors in 75.3 sec, at a rate of 20.4M triangles/sec. For comparison, Triangle created – in the best case between the two experimental platforms – a maximum of 65 million triangles in 114.74 sec, at a rate of 523K triangles/sec. The sequential version of PCDM performs slightly (up to 9.2%, for the sequential, scaled problem size execution on the IBM cluster) worse than Triangle, due to overheads to accommodate parallel processing. We, thus, investigated exploiting the two finer degrees of granularity of PCDM on SMT processors, in order to improve the single-processor performance of the algorithm.

We investigated whether the multiple execution contexts available in modern SMTs can be used to efficiently exploit the fine-grain parallelism of PCDM, using multithreading out of the box. Experimental results on IBM Power5 processors indicated that the overheads related to fine-grain parallelism management and execution overrun potential benefits, resulting overall in a significant performance degradation (approximately 2 times slower execution), compared even with a coarse-grain configuration that exploits half the execution contexts of the system. Moreover, the analysis of PCDM revealed that its fine-grain parallelism can utilize at most two (or four) execution contexts per physical processor, for 2- (or 3)-D meshes.

A medium-grain, optimistic, parallel implementation of PCDM provided significantly more work at a coarser granularity. We exploited medium-grain par-

allelism in PCDM at the SMT processor level. PCDM proved scalable at the processor level, without any additional hardware support, despite the frequent synchronization required between concurrently executing threads. It improved performance on a single core by up to 15%, compared even with the optimized Triangle. Moreover, a coarse+medium multi-grain approach proved to be the optimal configuration choice, when there are few threads executing on each node. Our experimental results also indicate that the coarse+medium approach is expected to be the configuration of choice when the number of MPI processes required to exploit all execution contexts grows too much, affecting preprocessing and MPI bootstrap overheads.

In the future work, we intend to verify the scalability advantages of our multi-grain algorithms on systems with larger number of nodes than the systems used in this study, and in particular on newer large-scale systems, some of which have already migrated to configurations built from multicore and multi-threaded processors. We also plan to investigate novel architectural paradigms for the implementation of parallel mesh generation algorithms, and in particular transactional memory [29], which seems suitable for the medium-grain parallelization presented in this paper. Transactional memory enables fast implementation and fairly efficient execution of codes with frequent, non-contested synchronization. Some preliminary results indicate that transactional memory is conceptually a suitable architectural model for Delaunay mesh generation [37], although current hardware and software implementations have inadequate semantics. Exploring these semantics and the adaptation of mesh generation algorithms to transactional systems is an interesting path to follow in future research.

Acknowledgments

This work was supported in part by the following NSF grants: EIA-9972853, ACI-0085963, EIA-0203974, ACI-0312980, CCS-0750901 Career award CCF-0346867, CNS-0521381, CCF-0833081 and DOE grant DE-FG02-05ER2568, as well as by the John Simon Guggenheim Foundation. We thank the anonimous reviewers for helpful comments.

References

[1] G. Almási, C. Caşcaval, J. Casta nos, M. Denneau, D. Lieber, J. Moreira, and H. Warren. Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture. *ACM SIGARCH Computer Architecture News*, 31(1):26–38, January 2003.

[2] Christos D. Antonopoulos, Filip Blagojevic, Andrey N. Chernikov, Nikos P. Chrisochoides, and Dimitris S. Nikolopoulos. Algorithm software and hardware optimizations for Delaunay mesh generation on simultaneous multithreaded architectures. *Journal on Parallel and Distributed Computing.* In revision, May 2007.

[3] 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.

[4] 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.

[5] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proc. of the 27th Annual International Symposium on Computer Architecture (ISCA'2000)*, pages 282–293, Vancouver, Canada, June 2000.

[6] 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.

[7] 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.

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

[9] Andrey Chernikov, Kevin Barker, and Nikos Chrisochoides. Parallel programming environment for mesh generation. In *Proceedings of the 8th International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 805–814, Honolulu, HI, June 2002.

[10] Andrey N. Chernikov and Nikos P. Chrisochoides. Parallel guaranteed quality planar Delaunay mesh generation by concurrent point insertion. In *Proceedings of the 14th Annual Fall Workshop on Computational Geometry*, pages 55–56, Cambridge, MA, November 2004. Electronic proceedings published at http://cgw2004.csail.mit.edu/proceedings.pdf.

[11] 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.

[12] 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, San Diego, CA, September 2005. Springer.

[13] Andrey N. Chernikov and Nikos P. Chrisochoides. Parallel guaranteed quality Delaunay uniform mesh refinement. *SIAM Journal on Scientific Computing*, 28:1907–1926, 2006.

[14] 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.

[15] L. Paul Chew. Constrained Delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.

[16] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report TR89983, Cornell University, Computer Science Department, 1989.

[17] L. Paul Chew, Nikos Chrisochoides, and Florian Sukup. Parallel constrained Delaunay meshing. In *ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation*, pages 89–96, Northwestern University, Evanston, IL, 1997.

[18] N. Chrisochoides, A. Fedorov, A. Kot, N. Archip, P.M. Black, O. Clatz, A. Golby, R. Kikinis, and S.K. Warfield. Toward real-time image guided neurosurgery using distributed and Grid computing. In *Proc. of IEEE/ACM SC06*, 2006.

[19] Nikos Chrisochoides and Démian Nave. Parallel Delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58:161–176, 2003.

[20] 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.

[21] Tim Culver. *Computing the Medial Axis of a Polyhedron Reliably and Efficiently*. PhD thesis, The University of North Carolina at Chapel Hill, 2000.

[22] Hugues L. de Cougny, Mark S. Shephard, and Can Ozturan. Parallel three-dimensional mesh generation. *Computing Systems in Engineering*, 5:311–323, 1994.

[23] 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.

[24] J. Galtier and P. L. George. Prepartitioning as a way to mesh subdomains in parallel. In *Proceedings of the 5th International Meshing Roundtable*, pages 107–121, Pittsburgh, PA, 1996.

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

[26] Gerhard Globisch. On an automatically parallel generation technique for tetrahedral meshes. *Parallel Computing*, 21(12):1979–1995, 1995.

[27] Gerhard Globisch. Parmesh – a parallel mesh generator. *Parallel Computing*, 21(3):509–524, 1995.

[28] Halit Nebi Gürsoy. *Shape interrogation by medial axis transform for automated analysis*. PhD thesis, Massachusetts Institute of Technology, 1989.

[29] L. Hammond, B. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K.Olukotun. Programming with Transactional Coherence and Consistency. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, Boston, MA, October 2004.

[30] Y. Ito, A. M. Shih, A. K. Erukala, B. K. Soni, A. N. Chernikov, N. P. Chrisochoides, and K. Nakahashi. Generation of unstructured meshes in parallel using an advancing front method. In *Proceedings of the 9th International Conference on Numerical Grid Generation in Computational Field Simulations*, San Jose, CA, June 2005.

[31] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In *Scalable High Performance Computing Conference*, pages 478–485. IEEE Computer Society Press, 1994.

[32] Clemens Kadow. Adaptive dynamic projection-based partitioning for parallel Delaunay mesh generation algorithms. In *SIAM Workshop on Combinatorial Scientific Computing*, San-Francisco, CA, February 2004.

[33] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March 2004.

[34] G.E. Karniadakis and S.A. Orszag. Nodes, modes, and flow codes. *Physics Today*, 46:34–42, 1993.

[35] George Karypis and Vipin Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Version 4.0*. University of Minnesota, September 1998.

[36] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE MICRO*, 25(2):21–29, March/April 2005.

[37] Milind Kulkarni, L. Paul Chew, and Keshav Pingali. Using transactions in Delaunay mesh generation. In *Proceedings of the Workshop on Transactional Memory Workloads*, pages 23–31, Ottawa, Canada, June 2006.

[38] 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.

[39] 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.

[40] Reinald Löhner and Juan R. Cebral. Parallel advancing front grid generation. In *Proceedings of the 8th International Meshing Roundtable*, pages 67–74, South Lake Tahoe, CA, 1999.

[41] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), February 2002.

[42] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, Pennsylvania, United States, 1996.

[43] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. A Delaunay based numerical method for three dimensions: Generation, formulation, and partition. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 683–692, Las Vegas, NV, May 1995.

[44] 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.

[45] Démian Nave, Nikos Chrisochoides, and L. Paul Chew. Guaranteed–quality parallel Delaunay refinement for restricted polyhedral domains. *Computational Geometry: Theory and Applications*, 28:191–215, 2004.

[46] Leonid Oliker and Rupak Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CD-ROM)*, page 39, New York, NY, USA, 1999. ACM Press.

[47] Maria-Cecilia Rivara, Daniel Pizarro, and Nikos Chrisochoides. Parallel refinement of tetrahedral meshes using terminal-edge bisection algorithm. In *13th International Meshing Roundtable*, Williamsburg, VA, September 2004.

[48] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, 1995.

[49] R. Said, N.P. Weatherill, K. Morgan, and N.A. Verhoeven. Distributed parallel Delaunay mesh generation. *Computer Methods in Applied Mechanics and Engineering*, (177):109–125, 1999.

[50] Evan Conway Sherbrooke. *3-D shape interrogation by medial axial transform*. PhD thesis, Massachusetts Institute of Technology, 1995.

[51] 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.

[52] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, 1997.

[53] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1–3):21–74, May 2002.

[54] TOP-500 Supercomputer Sites. http://www.top500.org, November 2004.

[55] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, pages 392–403, St. Margherita Ligure, Italy, June 1995.

[56] UltraSPARC©IV Processor Architecture Overview. Technical report, Sun Microsystems, February 2004.

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

[58] R. Williams. *Adaptive Parallel Meshes with Complex Geometry*. Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, 1991.