Multigrain Parallel Delaunay Mesh Generation: Challenges and Opportunities for Multithreaded Architectures

Christos D. Antonopoulos, Xiaoning Ding, Andrey Chernikov,
Filip Blagojevic, Dimitrios S. Nikolopoulos, and Nikos Chrisochoides*
Department of Computer Science
The College of William & Mary
118 McGlothlin Street Hall
Williamsburg, VA 23187-8795 U.S.A.
{cda,dingxn,ancher,filip,dsn,nikos}@cs.wm.edu

Abstract

Given the importance of parallel mesh generation in large-scale scientific applications and the proliferation of multilevel SMT-based architectures, it is imperative to obtain insight on the interaction between meshing algorithms and these systems. We focus on Parallel Constrained Delaunay Mesh (PCDM) generation. We exploit coarse-grain parallelism at the subdomain level and fine-grain at the element level. This multigrain data parallel approach targets clusters built from low-end, commercially available SMTs. Our experimental evaluation shows that current SMTs are not capable of executing fine-grain parallelism in PCDM. However, experiments on a simulated SMT indicate that with modest hardware support it is possible to exploit fine-grain parallelism opportunities. The exploitation of fine-grain parallelism results to higher performance than a pure MPI implementation and closes the gap between the performance of PCDM and the state-of-the-art sequential mesher on a single physical processor. Our findings extend to other adaptive and irregular multigrain, parallel algorithms.

1 Introduction

As modern supercomputers integrate more and more processors into a single system, system architects tend to favor hybrid, multilevel designs since such designs seem to be at the sweetspot of the cost/performance tradeoff. Most machines in the Top500 list [29] are clusters, often consisting of small-scale SMP nodes. The recent commercial success of simultaneous multithreaded (SMT) processors [16, 19, 31] introduces one additional level 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 hybrid, multilevel architectures, introduces new challenges for application developers.

*N. Chrisochoides is also a Visiting Associate Professor at the Department of Mechanical Engineering, MIT and the Department of Radiology, Harvard Medical School, Boston, MA, U.S.A.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'05 June 20-22, Boston, MA, USA
Copyright 2005 ACM 1-59593-167-8/06/2005 ...$5.00

Applications that expose multiple levels of parallelism, at different granularities, appear as ideal candidates for the exploitation of the opportunities offered by hybrid multiprocessors. However developers have to target both macro-scalability, across processors or nodes and micro-scalability, across the multiple execution contexts of each physical processor.

This paper focuses on the implementation of parallel mesh generation algorithms which are essential in many scientific computing applications in health care, engineering, and sciences. In real-time computer assisted surgery, like image-guided neurosurgery [34] the accuracy and speed are critical; the non-rigid registration of intra-operative MRI data has to complete in the order of few minutes. In direct numerical simulations of turbulence in cylinder flows with sudden drop of drag force around Reynold numbers $Re = 300,000$, on the other hand, high accuracy and a lot of main memory are essential; the size of the mesh is in the order of $Re^{9/4}$ [17] which, with an adaptive mesh refinement, is reduced to a few billions. Our study provides a thorough understanding of the mapping of mesh generation codes on current parallel systems and their interaction with the hardware, as well as guidelines for next generation software and hardware developers, in order to unleash the computational power of SMTs and clusters of SMTs. It is a step towards meeting the time and quality constraints set by real-world applications. 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.

The three most widely used techniques for parallel mesh generation are Delaunay [13], Advance Front, and Edge Subdivision. In this paper, we use the Delaunay technique because it can mathematically guarantee the quality of the mesh. Specifically, we focus on constrained Delaunay triangulation [7] and we explore 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. We investigate a multigrain parallelization approach for clusters built from: (1) conventional, single-thread, single-core processors, (2) low-end, commercially available SMTs and (3) simulated SMTs using modest and realistic architectural extensions for fine-grain synchronization and thread spawning.

Most of the existing parallel mesh generation methods [4, 8, 10, 12, 14, 15, 18, 21, 24, 25, 36] use only a coarse-grain approach, with only one exception [22]. In [22] the authors evaluate three 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 very expensive because they require the computation of the cavity graph[1] each time a set of independent points (or cavities) are inserted (or triangulated). We evaluate the effect of low-level locks in Section 3.

Coarse-grain methods decompose the original mesh generation problem into smaller subproblems that are solved (meshed) in parallel. Subproblems are formulated to be either tightly or partially coupled, or even decoupled. The coupling of subproblems determines the intensity of the communication and the degree of synchronization between the subproblems.

Our multigrain approach (PCDM) is based on a coarse-grain, partially coupled algorithm, in order to achieve scalability at the node level and a finer-grain tightly-coupled approach 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 bring it closer to the performance of Triangle, the state-of-the-art sequential Delaunay mesh generation software [26].

Our experimental evaluation shows that current SMTs are not capable of exploiting fine-grain parallelism in PCDM due to synchronization overhead and lack of hardware support for light-weight threading. However, we find using simulation that modest and realistic hardware extensions for thread synchronization and scheduling allow the multi-grain implementation of PCDM to both achieve higher single-processor performance than sequential PCDM and outperform the coarse-grain, single-level MPI implementation if more than one processors are available.

A major limitation of the fine-grain parallelization of PCDM is that it can effectively use up to two or three[2] hardware execution contexts. This motivates a future direction to explore a medium-grain, optimistic parallelization strategy which increases the granularity and concurrency of PCDM within each subdomain. A preliminary analysis of this parallelization strategy shows that it can effectively exploit the hardware on current and emerging multithreaded processors in order to attain performance benefits.

The main contribution of this paper is the identification of 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 developing efficient parallel algorithms and software for asynchronous, adaptive and irregular applications on current and emerging parallel architectures.

The rest of the paper is organized as follows. In Section 2 we describe both the sequential and the parallel approach for PCDM. In Section 3 we present the implementation of the: (1) coarse-grain approach on an UltraSPARC cluster with 64 uniprocessor nodes and (2) the fine-grain approach on: (i) a hybrid, 4-way, Intel Xeon HyperThreaded shared memory multiprocessor and (ii) a simulated multi-SMT system with processors configured similarly to the Intel HyperThreaded processor and additional functionality for the efficient execution of fine-grain parallel work on the processor's execution contexts. Section 4 summarizes the paper.

2 PARALLEL DELAUNAY MESHING

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.

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

[2]For 2- and 3-dimensional meshes respectively.

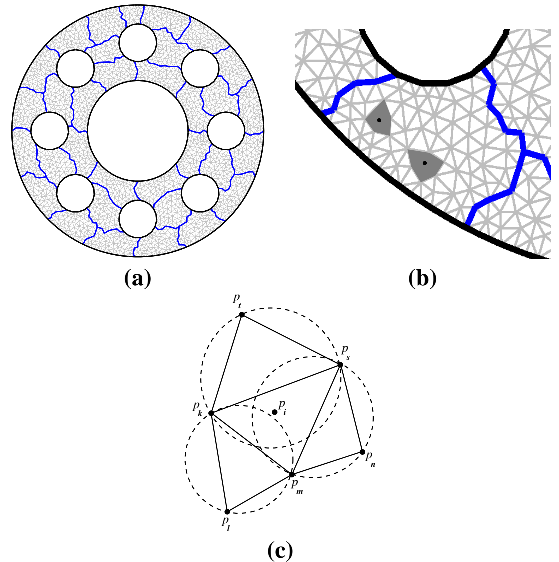

Figure 1. (a) A coarse-grain decomposition of the pipe cross-section into 32 subdomains. (b) Medium-grain parallel expansion of multiple cavities within a single subdomain. (c) Fine-grain parallel expansion of a single cavity ($\mathcal{C}p_i$) by concurrent testing of multiple triangles ($\triangle(p_k p_l p_m)$, $\triangle(p_m p_n p_s)$, $\triangle(p_s p_t p_k)$).

In this paper we use geometric criteria like triangle area ($\bar{\Delta}$) and circumradius-to-shortest edge ratio ($\bar{\rho}$). 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 [5, 35] to update the triangulation. 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 circumcircles include the newly inserted point p_i. This set is called a *cavity* ($\mathcal{C}p_i$). The triangles in 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 sequential Delaunay refinement algorithm can be parallelized at three levels of granularity, which are depicted in Figure 1 and described in the following paragraphs.

2.1 Coarse Grain Parallelism

In the coarse-grain parallel implementation, an initial mesh of the domain is created first. Then, the mesh is partitioned into $N \gg P$ subdomains, where P is the number of processors (Fig. 1a). Finally, subdomains are mapped to processors. 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 or runtime systems [3].

The domain decomposition procedure described above creates N subdomains, each of which is bounded by edges of the initial coarse triangulation. 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 [27], 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, interprocess communication is tremendously simplified: the only message between processes working on neighbouring subdomains is of the form, "split this interface" and is sent when a newly inserted point encroaches upon the interface edge.

2.2 Fine-Grain Parallelism

The innermost level of parallelism is exploited by allowing multiple threads to cooperate during the expansion of a single cavity. The procedure of cavity expansion lasts from 4 to 6 μsec per cavity on a modern 2 GHz Intel Xeon processor. However, millions of cavity expansions are required for the generation of a typical mesh. As a result, cavity expansion actually accounts, in average, for 58% of the total execution time of the mesh generation code. Algorithmically, the expansion is similar to a breadth-first search of a graph. The neighbors of the offending triangle are initially enqueued at the tail of a queue. At each step, one element is dequeued from the head of the queue and is subjected 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.

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

	key		pipe		cylinder	
	1M	10M	1M	10M	1M	10M
Queue Length	2.053	2.058	2.052	2.058	2.053	2.058
Cavity Population	4.828	5.074	4.830	5.066	4.841	5.069

Table 1 summarizes statistics from the execution of fine-grain PCDM for three input sets from real-world problems: A key, a rocket engine pipe (depicted in Fig 1a), and a cylinder structure used for undersea oil exploration. 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[3]. 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 exploited by SMT processors with two execution contexts per physical processor package.

2.3 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. Bad-quality triangles, i.e., triangles that do not satisfy the qualitative constraints set by the user, are organized in a queue. Each thread dequeues a bad-quality triangle and expands its cavity.

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

As soon 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. In case a shared triangle is detected, only one of the cavities in which the triangle participates can be retriangulated. The other cavities are canceled and are re-expanded later.

We have experimentally evaluated the degree of medium granularity concurrency exposed by PCDM. More specifically, we have applied PCDM on the key data set and created a mesh with 1 million triangles. We have performed five experiments, attempting to concurrently expand 32, 64, 128, 256 or 512 cavities. These experiments simulate the parallel execution with 32, 64, 128, 256 and 512 threads respectively. In each experiment we recorded the number of bad-quality triangles available throughout the execution. Moreover, we recorded the percentage of cavity expansions that finished without conflicts. The product of the available bad-quality triangles and the percentage of successful expansions at each time snapshot provides a statistical estimation of the available parallelism.

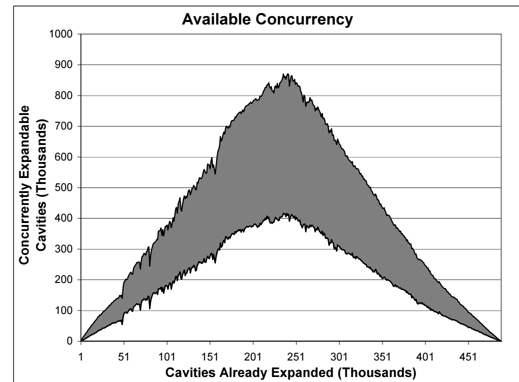

Figure 2. Statistical estimation of the available parallelism throughout the execution life of a medium-grain PCDM, when 32 to 512 processors are used. The lower and upper curves correspond to the minimum and maximum estimation respectively.

The results are depicted in Figure 2. The upper and lower curves in the diagram correspond to the maximum and minimum estimated degree of parallelism across all 5 experiments. Even in the worst case scenario, i.e., if we consider the minimum estimation of the degree of parallelism and the maximum number of threads (512), medium-grain PCDM exposes enough parallelism to efficiently exploit all 512 execution contexts. Throughout the execution life of the application, the expected degree of parallelism is less than the number of threads only during the expansion of the last 1000 cavities, namely when the mesh has already been refined enough to almost totally conform with the qualitative criteria set by the user. It should also be pointed out that – in the worst case scenario – an average of 400 expandable cavities correspond to each thread. Considering an expansion time of 4 to 6 μsec per cavity, the granularity of the available work chunks ranges between 1.6 and 2.4 msec.

The percentage of successful cavity expansions is highly dependent on the selection of the initial bad-quality triangles. Bad-quality triangles which are closely situated in the 2D plane tend to be also close in the queue. However, concurrently expanding the cavities of neighboring triangles often ends up in collisions, which result to a percentage of canceled cavity expansions often higher than 30%. A simple strategy of randomly selecting triangles from the queue though, significantly reduces the percentage of collisions in

the range of 6% to 10%. We intend to evaluate even more sophisticated strategies, which guarantee non-conflicting cavity expansions by limiting the minimum distance between concurrently targeted triangles, according to the dynamically changing qualitative characteristics of the mesh [6].

A fully optimized, parallel, medium-grain implementation of PCDM was not available at the time of writing this paper, in order for us to compare it fairly against the coarse-grain, fine-grain and dual-grain (coarse and fine) implementations of PCDM. However, our preliminary experiments show that the medium-grain sequential implementation's performance is, without extensive optimizations, within a small factor (two to three) of the performance of Triangle. Based on these results and our preliminary analysis of the concurrency and granularity of tasks in the medium-grain PCDM algorithm we expect this algorithm to be appropriate for processors with more than two execution contexts and for systems in which the fine-grain implementation suffers excessive overhead due to synchronization and thread management. Tuning and evaluating the implementation of this algorithm on multithreaded processors is the first priority of our future work.

3 PCDM Mapping on Current and Emerging Parallel Architectures

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

- An homogeneous, commodity, off-the-self (COTS) cluster. The cluster integrates 64 single-processor nodes based on UltraSPARC IIi+ CPUs, clocked at 650 MHz, with 1 GB main memory each. 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 hybrid, 4-way, shared memory multiprocessor, based on Intel Xeon HyperThreaded (HT) processors. Intel HT processors are the most wide-spread commercial processors offering an early implementation of simultaneous multithreading. They allow up to two threads to potentially execute concurrently on the same processor, sharing a common set of resources such as execution units, cache, TLB, etc. As long as the threads do not have conflicting resource requirements they execute in parallel. If, however, this is not the case, their execution is serialized. Each processor runs at 2.0 GHz. The system has 2 GB of memory and runs the Linux 2.4.25 kernel. We evaluate both the scalability of the MPI-based, coarse-grain parallelization across physical processors, as well as the efficiency of a fine grain parallelization targeted to the two execution contexts available on each processor package.
- Driven by shortcomings we identified in the low-level support offered by Intel HT processors for the exploitation of fine-grain parallelism, we also experiment on simulated SMT processors. These SMTs are configured similarly to Intel HT processors, with roughly equal amount of resources. However, they offer a few modest hardware extensions which allow the efficient execution of fine-grain parallel work on the processor's execution contexts. It is realistic to expect that some or all of these hardware extensions will be available in emerging SMT processors. Each simulated processor has two CPU cores, which share internal processor blocks as well as L1, L2, and L3 caches. We consider 1-, 2- and 4-way shared memory multiprocessor configurations based on the simulated SMT processor, with 2 GB of main memory.

3.1 Execution on a COTS Cluster

We have executed PCDM on a COTS cluster, using 1 to 64 processors. The domain to be meshed is divided in 1024 subdomains, i.e., the meshing problem is divided in 1024 partially coupled subproblems. 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 10 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 parallel execution time respectively for the specific problem size ($W = 10$ 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 [26], the best known sequential implementation for Delaunay mesh generation which has been heavily optimized and manually fine-tuned. 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. The problem size equals approximately 10M triangles per processor. In other words, the problem size gradually increases from 10 to 640 million 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 processors in order to tackle problem sizes that are out of the reach of sequential algorithms due to both computational power and memory limitations.

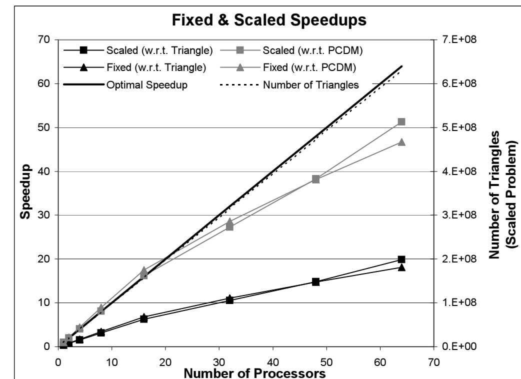

Figure 3. Fixed and scaled speedups of the coarse-grain PCDM on a 64 processor cluster. The speedups have been calculated using as a reference either the single-processor execution time of PCDM or the execution time of the best known sequential mesher.

Fig. 3 summarizes the experimental results from both sets. The diagram also depicts the optimal speedup, as well as the problem size used in the scaled experiment set.

PCDM scales very well, in both experiment sets, when the sequential PCDM execution time is used as a reference. For up to 16 processors the speedup is actually superlinear, due to the fact that in the parallel experiments the total cache and main memory available to the application is proportional to the number of processors. For cluster configurations with 32 or more processors the attained speedup is lower than the optimal. It measures up to 51 and 47 on 64 processors for the scaled and fixed experiment sets respectively. This can be attributed to both an increase in communication requirements and load-balancing issues. More specifically, as more subdomains are meshed in parallel, more points are introduced at the boundaries of subdomains, thus sending messages to

processors which mesh immediately neighboring subdomains. At the same time, given that the number of subdomains is fixed to 1024 in our experiments, the increase in the number of processors results in a reduction in the number of subdomains corresponding to each processor. There are thus fewer opportunities to compensate for a subdomain with computational requirements deviating significantly from the average.

Speedups are lower if the execution time of Triangle is used as a basis for comparison. The speedups attained on 64 processors for the fixed and scaled experiments are 18 and 20 respectively. This can be attributed to the fact that the heavily optimized code of Triangle is 2.58 times faster than the sequential PCDM code on the specific single-processor nodes. Due to its focus on parallelism, PCDM is not susceptible to the same degree of optimizations as Triangle. In the rest of the paper we focus on the exploitation of the multilevel parallelism opportunities offered by PCDM as well as of the multiple hardware contexts available in modern SMTs, in order to close the gap in the single processor performance between PCDM and Triangle.

We have repeated our experiments with similar results on an heterogeneous, 128 processor cluster. The experimental results indicate that PCDM keeps performing well and scales up to 128 processors. However, due to the heterogeneity of the cluster, no formal quantitative metrics can be reported beyond that general trend.

3.2 Execution on a Hybrid, SMT-based Multiprocessor

As a next step, we evaluated the performance of PCDM on commercially available hybrid SMPs, which are based on simultaneous multithreaded (SMT) processors. More specifically, we experimented on a 4-way system built of Hyperthreaded Pentium 4 Xeon processors. Table 2 outlines the configuration of the system.

Table 2. Configuration of the Intel HT Xeon-based SMP system used to evaluate the fine-grain implementation of PCDM on current SMT processors.

Processor	4, 2-way Hyperthreading, Pentium 4 Xeon, 2 GHz
Cache	8 KB L1, 64B line / 512KB L2, 64B line / 1MB L3, 64B line
Memory	2 GB RAM

We used the two execution contexts available on each processor 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.

3.2.1 Implementation Issues

We applied only limited, local optimizations 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 [22]. 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. Moreover, contention on a lock does not result to memory traffic on the system bus and activation of the cache-coherence protocol, since both contending threads execute on the same processor and share all levels of the cache hierarchy. An alternative queue implementation could employ lock-free techniques. However, lock-free data structures outperform lock-based ones only in the presence of multiprogramming or when access to shared resources is heavily contended [20]. None of these conditions holds in the case of fine-grain PCDM.

In order to both guarantee the correctness of the algorithm 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 with a *tag* field. Threads check the *tag* field 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 *tag* has already been set, the triangle is discarded and is not subjected to the INCIRCLE() test.

3.2.2 Experimental Evaluation

We performed three sets of experiments on a 4-way SMP, in order to evaluate the ability of Intel Hyperthreaded (HT) processors to exploit parallelism available in PCDM. More specifically, we first experimented with a version of PCDM which exploits both the fine and the coarse granularities of parallelism (*MPI+Fine*). The fine granularity is exploited by the two execution contexts available on each HT processor. At the same time, every pair of threads executing on each physical processor corresponds to an MPI process. Multiple MPI processes – on multiple processors – are used to exploit the coarse granularity. We also experimented with the MPI-only implementation of PCDM. In this case, all the execution contexts are used to execute MPI processes. The latter work on different subdomains of the problem, i.e., at the coarse granularity level. We have executed 2 versions of the MPI-only experiments: MPI processes are either spread to as many physical processors as possible (*Diff Procs*) or are packed together on different execution contexts of the same processor (*Same Proc*). If for example 4 execution contexts are used, in the first version they are spread across 4 different processors, whereas in the second version threads are executed on the 4 execution contexts of 2 physical processors.

Figure 4 depicts the speedup with respect to a sequential PCDM execution. In all cases we are producing meshes of 10 million triangles for the key input set. On the specific system, Triangle is 2.3 times faster than the sequential PCDM. The multilevel PCDM code (*MPI+Fine*) does not scale well. In fact a slowdown of 1.44 occurs as soon as a second thread is used to take advantage of the second execution context of the HT processor. The performance is improved as more physical processors are used (4 and 8 execution contexts), however 4 physical processors are required before

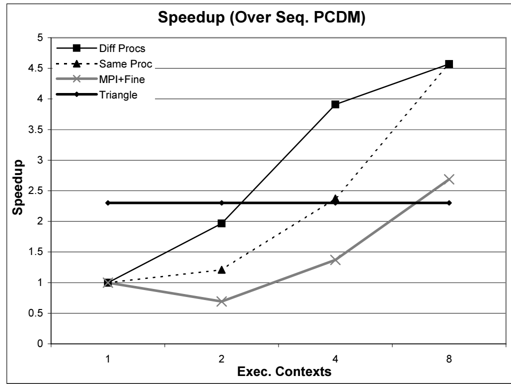

Figure 4. Fixed speedup with respect to the single-threaded PCDM execution

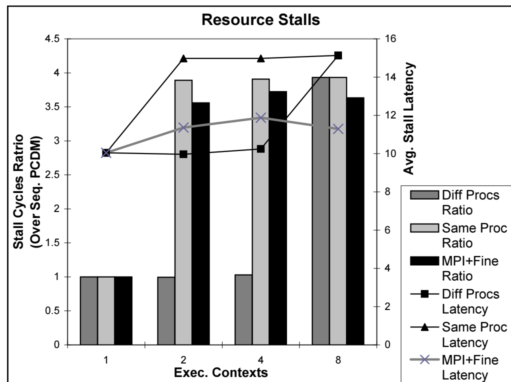

Figure 5. Number of stall cycles (with respect to the single-threaded PCDM execution) and average stall latency.

PCDM outperforms Triangle.

The performance of the MPI-only version, when MPI processes are spread across processors (*Diff Procs*) is significantly better. The use of just 2 execution contexts yield performance close to that of Triangle. When 4 execution contexts are used, PCDM is significantly faster than Triangle. However, packing MPI processes on as few physical processors as possible (*Same Proc*) results in performance penalties. The speedup lies between that of the *Diff Procs* and *MPI+Fine* versions. The exploitation of the second execution context on a single physical processor results in a speedup of 1.21. 4 execution contexts, on 2 physical processors are necessary in order to match the performance of Triangle.

We used the hardware performance counters available on Intel HT processors, in order to identify the reasons that lead to performance penalties whenever two execution contexts per physical processor are used. We focused on the number of stalls, the corresponding number of stall cycles, as well as the number of retired instructions in each case. We measure the cumulative numbers of stall cycles, stalls and instructions from all threads participating in each experiment. The results are depicted in the diagrams of figures 5 and 6 respectively. Ratios have again been calculated with respect to the sequential PCDM execution.

The number of stall cycles (Fig. 5) is a single metric that provides insight into the extent of contention between the two threads running on the execution contexts of the same processor. It indicates the number of cycles each thread spent waiting because an internal processor resource was occupied by either the other thread or by previous instructions of the same thread. The average per stall latency, on the other hand, indicates how much performance penalty each stall introduces. The stall cycles suffered during the sequential execution are 4.3 billion. Whenever two threads share the same processor, the stall cycles are from 3.56 to almost 4 times more. However, resource sharing has also a negative effect on the average latency associated with each stall. The average latency is 10 cycles when there is one thread per physical processor. When two MPI processes share the same processor it raises to approximately 15 cycles. If two threads that exploit the fine-grain parallelism of PCDM are located on the same processor the average latency ranges between 11.3 and 11.9 cycles.

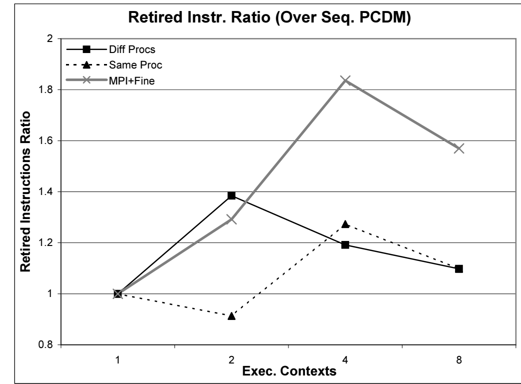

Figure 6. Number of retired instructions, with respect to the single-threaded PCDM execution.

Interesting information is also revealed by the number of retired instructions (Fig. 6). Whenever 2 MPI processes are used (2 execution contexts in the *Diff Procs* case and 4 execution contexts in the *Same Proc* and *MPI+Fine* cases), the total number of instructions is maximized. We have traced the source of this problematic behavior to the internal implementation of the MPI library, which attempts to minimize response time by performing active spinning whenever a thread has to wait for the completion of an MPI operation. Active spinning produces very tight loops of "fast" instructions with memory references that hit into the L1 cache. If more than two processors are used, the cycles spent spinning inside the MPI library are reduced, with an imminent effect on the number of instructions.

Another interesting observation is that the multilevel version of the algorithm (*MPI+Fine*) results in more instructions than the *Same Proc* MPI-only version. The reason is again active spinning. Unfortunately Intel HT processors do not offer an efficient hardware mechanism for fine control on the suspension and resumption of a thread running on an execution context. The mechanisms offered by the operating system are too heavy-weight for applications with extremely fine-grain parallelism, such as PCDM. The cost of a suspend/resume cycle using OS primitives equals the cost for the expansion of several hundreds of cavities. As a consequence, whenever the *worker* thread idles, it actively spins on triangle queues waiting for work, thus issuing a large number of instructions. In fact the extent of active spinning in *MPI+Fine* code is significantly higher than in the MPI-only versions of the algorithm. This explains the lower average stall latency of the *MPI+Fine* implementation: Many stalls are caused by instructions issued in the context of spinning loops. However these instructions retire quickly and do not introduce high latencies.

A significant side-effect of active spinning is the performance penalty suffered by computational threads that share the same processor with spinning threads. Both threads share a common set of processor resources, such as execution units and instruction queues. The instructions issued by the spinning threads tend to fill

the queues, thus delaying potentially useful instructions issued by the other execution context. Our experimental evaluation indicates a slowdown of more than 25% when a PCDM thread is executed together with an active spinning thread on the same physical CPU.

As explained in section 2.2, attention has been paid to minimize interaction between two PCDM threads working on the same cavity expansion. A detailed profiling indicated, though, that up to 23% of the cycles is spent on synchronization operations. Synchronization is limited among two threads 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. Moreover, the full-software implementation of mutual exclusion algorithms introduces active spinning at the entrance of protected code regions, should access to these regions be contended.

Another fundamental reason that prohibits Intel HT processors from efficiently supporting fine-grain parallelism is the lack of hardware support for light-weight threading. Such support includes hardware primitives for efficient thread spawning and joining, queuing of hardware threads and dispatching to the execution contexts of the processor. The lack of such support forces programmers to implement similar functionality in software. As a result, the PCDM implementation organizes unprocessed triangles in queues and uses kernel threads as "virtual processors" which dispatch and process triangles. If adequate hardware functionality were available, unprocessed triangles would be naturally translated to hardware threads, rendering all the aforementioned software support unnecessary. Such an implementation would also eliminate problems identified earlier, such as active spinning whenever the *worker* thread is idling, synchronization for access to the queues etc.

3.2.3 Alternative Exploitation of Execution Contexts in SMTs - Speculative Precomputation

As is the case with most pointer-chasing codes, PCDM suffers from poor cache locality. Previous literature has suggested the use of speculative precomputation (SPR) [9] for speeding up such codes on SMTs and CMPs [9, 33]. SPR exploits one of the execution contexts of the processor in order to precompute addresses of memory accesses that lead to cache misses and preexecute these accesses, before the computation thread. In many cases, the precomputation thread manages to execute faster than and ahead of the computation thread. As a result, data are prefetched timely into the caches.

We have evaluated the use of the second hyperthread for indiscriminate precomputation, by cloning the code executed by the computation thread and stripping it from everything but data accesses and the memory address calculations. The precomputation thread successfully prefetched all data touched by the computation thread. However, the execution time was higher than that of the 1 thread per CPU or 2 computation threads per CPU versions. Intel HT processors do not provide mechanisms for low overhead thread suspension / resumption. As a result, when the precomputation thread prefetches an element, it performs active spinning until the next element to be prefetched is known (as a result of the algorithm execution by the compute thread). However, active spinning slows down - as reported earlier - the computation thread by more than 25%. We tried to suspend/resume the precomputation thread using the finest-grain sleep/wakeup primitives available by the OS. In this case, the computation thread does not suffer a slowdown, however the latency of a sleep/wakeup cycle spans the expansion time of hundreds of cavities. An additional problem is that the maximum possible runahead distance between the precomputation and computation thread is equal to the degree of concurrency, namely approximately 2 in the fine-grain 2D case. This precludes the use of the precomputation thread in batch precompute/sleep cycles.

3.3 Hardware Support for Fine-Grain Multithreading

The previous discussion revealed weaknesses in the design of current, commercially available SMTs, that do not allow the efficient exploitation of fine-grain parallelism. In this section we discuss a set of potential architectural extensions, similar to extensions that have already been proposed for fine-grain and speculative multithreaded processors, and project the impact that these architectural optimizations will have on performance. We focus on hardware support for synchronization and thread management, since emerging processor architectures can easily provide realistic support for almost zero-cost synchronization and thread spawning/joining.

3.3.1 Extensions for Fine-Grain Synchronization

The problem of synchronization latency on multithreaded processors has been addressed in earlier work. Fine-grain synchronization on a word-by-word basis can be enabled by a full/empty bit, an architectural feature used first in the Tera MTA [2, 28], by special-purpose synchronization registers, such as those found on Cray XMP, and by other mechanisms. In this work we consider the use of a lock-box [30], as an efficient mechanism for synchronization, which can be implemented with modest hardware cost. A lock-box is a small buffer in the processor with one entry per thread, including the lock address, the address of the locking instruction and a valid bit. On a failed attempt to acquire a lock with a read-modify-write instruction, the acquiring thread blocks and is flushed from the processor. On a release, the address of the lock is compared – with a parallel, associative search – against all the contents of the lock box. If a match is found, the matching thread is woken up. We estimate the latency of the entire critical path of a critical section using the lock box to 10 cycles, following the design suggestions in [30].

3.3.2 Extensions for Fine-Grain Thread Spawning

Several multithreaded processor designs, including simultaneous multithreaded processors with embedded support for dynamic precomputation [9], threaded multipath execution processors [32] and implicitly multithreaded processors [23], support automatic thread spawning in hardware. Besides multiple hardware contexts and program counters, this hardware support includes a mechanism for communicating the live-in register values to a newly spawned thread, and instructions to spawn and join threads. Some designs allow thread spawning in the context of the same basic block, while others extend this mechanism to support function calls issued on separate hardware contexts [1]. Although many of the related studies assume negligible thread spawning latencies, communicating register values requires extra processor cycles, and in some cases, register spilling to memory. Related studies estimate the latency between 2 and 10 cycles depending on the assumptions [1, 23, 32]. In this work, we simulate a hardware thread spawning mechanism which supports register communication between threads executing across basic blocks and function boundaries. We conservatively assume a latency of 10 cycles for thread spawning. This latency includes register communication and potential queuing of threads.

3.3.3 Experimental Evaluation of Hardware Extensions

We have used a multi-SMT simulator based on SimICS [11], to evaluate the impact of limited, realistic hardware support for thread execution and synchronization on the performance of the fine-grain implementation of PCDM. Table 3 shows the parameters of our multi-SMT simulator. We simulated the functionality of a lock box

and a hardware thread spawning mechanism. Notice that, whenever possible, our simulated system is configured with exactly the same amount of resources offered by our real multi-SMT platform. This allows us to isolate the impact of our hardware extensions to the performance of the fine-grain parallel execution.

Table 3. Simulation parameters for the multi-SMT system used to evaluate the fine-grain implementation of PCDM on emerging microprocessors.

Processor	4, 2-way Hyperthreading, Pentium IV ISA, 2 GHz
Cache	8 KB L1, 64B line size, 2-cycle hit, 7-cycle miss penalty
	512KB L2, 64B line size, 14-cycle miss penalty
	1MB L3, 64B line size, 120-cycle miss penalty
Memory	2 GB RAM
Thread-spawning latency	10 cycles
Lock critical path latency	10 cycles

We conducted complete system simulations – including system calls and OS overhead – using different levels of hardware support, with the multilevel implementation of PCDM, which exploits the fine and coarse granularities of parallelism. We executed the code with no hardware support for thread spawning and synchronization (labeled *SW*), with hardware support for thread spawning only (labeled *HWT*), with hardware support for synchronization only (labeled *HWL*) and with hardware support for both thread spawning and synchronization (labeled *HWT+HWL*). The codes were executed on 1, 2 and 4 simulated Hyperthreaded processors, with the Pentium IV ISA. The multithreaded version uses both execution contexts of each physical processor to process triangles during cavity expansion operations. If hardware support for threading is provided, one hardware thread is created and used for the processing of each triangle. On software threading versions, 2 kernel threads are created per physical processor and process triangles from queues. If hardware support for synchronization is available, it is used for both the protection of queues and the tagging of processed triangles. We compared the fine-grain PCDM against the MPI implementation. The latter was executed with two MPI processes per simulated Hyperthreaded processor (*Same Proc* configuration).

Figure 7 shows the total number of cycles (top chart) and the number of cycles normalized to the execution cycles of the MPI version (bottom chart) of PCDM. The reported results are from the complete execution of the program, creating a mesh of 1 million triangles[4] for the key input set. The results indicate that the fine-grain threaded implementation of PCDM imposes additional overhead, compared with the pure-MPI implementation. With no hardware support, the fine-grain multithreading overhead has a dramatic impact, yielding code which is 2.5 times slower than the MPI code, on a single HT processor. This overhead is purely attributed to the lack of hardware support and drops rapidly with hardware extensions for fine-grain multithreading. On one HT processor, hardware support for fast synchronization reduces the execution time of PCDM by 42%. Hardware support for thread spawning reduces the execution time by 17%. With the two hardware mechanisms combined, execution time drops by 53%, showing an almost perfectly cumulative effect of the improvements due to hardware support for synchronization and threading. Compared to the MPI code, the fine-grain threaded code is 17% slower, assuming full hardware support. However, it should also be noted that the use of both execution contexts of each processor in combination with the hardware support results in code that is 20% faster than the sequential PCDM.

[4] We had to experiment with a smaller problem size to limit the duration of the simulations. We simulated a few representative configurations with the larger problem size and found that the results agreed completely with the results obtained with the small problem size. The fine-grain implementation was at least 10% faster than the monolithic MPI implementation in all cases.

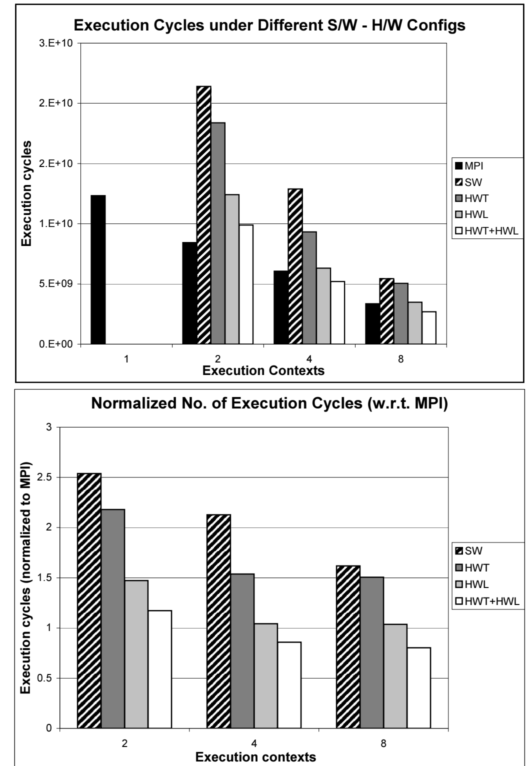

Figure 7. Execution cycles under different levels of hardware support (top diagram). Execution cycles under the same S/W and H/W configurations, normalized with respect to the pure MPI version on the same number of execution contexts (bottom diagram).

Interesting observations can be made by investigating the scalability of the two implementations. For the fine-grain implementation, the threading overhead is constant and does not increase with the addition of more processors, since there are always 2 threads per MPI process. As Fig. 7 conveys, the scalability of all fine-grain threaded implementations on one to four Hyperthreaded processors is good and the parallel efficiency is more than 87% in all cases. On the contrary, the MPI implementation does not scale well on multiple Hyperthreaded processors, and achieves only up to 62% efficiency. As more MPI processes are used, the overhead due to the communication between different processors and active spinning increases linearly. The fine-grain version does not suffer from these phenomena and the only performance penalty it incurs is due to thread management. This penalty is however mitigated with the hardware implementation.

Overall, the hybrid version utilizing fine-grain multithreading within each processor scales better and achieves higher performance than the monolithic MPI version, as work is distributed over more Hyperthreaded processors. On 2 and 4 Hyperthreaded processors, the hardware-assisted fine-grain version (*HWL+HWT*) outperforms the MPI-only version by 15% and 20% respectively. Therefore, the combination of the coarse- and fine-grain parallelism extracted from PCDM succeeds in better utilizing system resources.

We expect more aggressive hardware mechanisms for thread management and synchronization to be present in the upcoming generations of multithreaded processors. For example, the IBM

Power5 [16] includes dynamic thread switching in hardware to release the resources of threads holding locks and accelerate the execution of locks by redistributing the resources of the processor between critical and non-critical threads. More aggressive support will be a natural aftereffect of advances in technology and the need to meet the requirements of applications with fine-grain parallelism.

4 CONCLUSIONS

As multithreaded processors become more widespread and parallel systems are being built using these processors, it is imperative to investigate the interaction between applications and these processors in more detail. Adaptive and irregular applications are a challenging target for any parallel architecture, therefore investigating whether multithreaded processors are well suited for such applications is an important undertaking. Our paper makes contributions towards this direction, focusing on mesh generation algorithms. Fast, high quality mesh generation is a prerequisite for a multitude of real-world medical and engineering applications.

Earlier work [22] 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 less execution contexts, such as most commercially available SMTs. 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 [22], 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 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.

In this paper we focused on PCDM, a parallel, guaranteed-quality mesh generator. PCDM exposes parallelism in three granularities. We exploit the coarsest grain with an MPI-only implementation which proves to scale well on a 64 and a 128 processor cluster. Its ability to use more than one processors allows it to solve problems faster than Triangle, the best, 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 memory limitations. However, the sequential version of PCDM performs significantly worse than Triangle. We, thus, investigated whether the multiple execution contexts available in modern SMTs can be used to efficiently exploit the fine-grain parallelism of PCDM and to close the performance gap between PCDM and Triangle on a single processor, using multithreading out of the box. Initial experimental results on Intel Hyperthreaded processors indicated that the overheads related to fine-grain parallelism management and execution overrun potential benefits, resulting often to performance degradation. Through careful and detailed profiling, using the hardware performance counters available on the processor, we identified the most important bottlenecks. Following this evaluation, we suggested a few modest hardware extensions that will allow emerging SMTs to efficiently execute parallelism at the finest granularity offered by PCDM. We evaluate the performance impact of these extensions on a simulated SMT multiprocessor. The new mechanism allows the multithreaded version of PCDM to outperform the sequential one on a single physical processor. A multilevel version, using both threading and MPI outperforms a pure, single-level MPI version if more than one processors are available.

Despite the inefficiency of thread management mechanisms on current SMT processors, we were able to match the performance of the best known sequential algorithm (Triangle) with four threads and outperform it with eight threads on a quad SMP. We foresee that hardware improvements for multithreading on SMTs coupled with our multigrain-multithreading substrate will bring the crossover point to an even lower number of threads.

PCDM is memory bound due to extensive pointer chasing. Multithreading can thus be employed to reduce memory latency, via speculative precomputation. Our results have shown that existing processors lack the efficient thread management mechanisms needed to implement timely and low-interference speculative precomputation. Investigating speculative precomputation with more aggressive SMT hardware and in conjunction with the fine-grain multithreading implementation of PCDM is left as future work.

The analysis of PCDM revealed that its fine-grain parallelism can utilize at most two (or three) execution contexts per physical processor, for 2- (or 3)-D meshes. However, a medium-grain parallel implementation of PCDM can provide enough work to keep up to 512 execution contexts busy. As a natural extension of this work, we intend to evaluate the performance of medium-grain PCDM on SMTs with more than two execution contexts per processor.

Many of the performance problems in SMTs can be traced down to the extensive resource sharing and the lack of performance isolation between the execution contexts of each processor. Chip multiprocessors (CMPs) play dominant role in the roadmaps of most major processor manufacturers. This class of processors integrates more than one complete processor cores on a single chip. The multiple cores share only one or more upper levels of the cache hierarchy. Therefore, we expect that CMP-based SMPs will be able to efficiently support on-chip multithreading, even in cases SMTs can not due to conflicts in the resource requirements of the co-executing threads. A natural next step of our work is to investigate the interaction of demanding irregular and adaptive applications, such as PCDM, on CMP-based systems.

ACKNOWLEDGMENTS

This work was supported in part by the following NSF grants: Career award CCF-0346867, ACI-0312980, EIA-0203974, ACI-0085963, and EIA-9972853. We would like to thank Chaman Verma for his initial implementation of the medium-grain PCDM algorithm and the anonymous referees for their valuable comments.

5 REFERENCES

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO–31)*, pages 226–236, Dallas, TX, November 1998.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. of the 4th ACM International Conference on Supercomputing (ICS'90)*, pages 1–10, Amsterdam, The Netherlands, June 1990.

[3] K. Barker, A. Chernikov, N. Chrisochoides, and K. 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, G. L. Miller, and D. Talmor. Developing a Practical Projection-Based Parallel Delaunay Algorithm. In *12th Annual Symposium on Computational Geometry*, pages 186–195, 1996.

[5] A. Bowyer. Computing Dirichlet Tesselations. *Computer Journal*, 24:162–166, 1981.

[6] A. N. Chernikov and N. 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. ACM Press, 2004.

[7] L. P. Chew. Constrained Delaunay Triangulations. *Algorithmica*, 4(1):97–108, 1989.

[8] L. P. Chew, N. Chrisochoides, and F. 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.

[9] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA–2001)*, pages 14–25, Göteborg, Sweden, July 2001.

[10] H. de Cougny and M. Shephard. Parallel Volume Meshing Using Face Removals and Hierarchical Repartitioning. *Comp. Meth. Appl. Mech. Engng.*, 174(3-4):275–298, 1999.

[11] F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Proc. of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[12] J. Galtier and P. L. George. Prepartitioning as a Way to Mesh Subdomains in Parallel. In *Special Symposium on Trends in Unstructured Mesh Generation*, pages 107–122. ASME/ASCE/SES, 1997.

[13] P. L. George and H. Borouchaki. *Delaunay Triangulation and Meshing: Applications to Finite Element*. Hermis, Paris, 1998.

[14] M. T. Jones and P. E. Plassmann. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. In *Proceedings of the Scalable High-Performance Computing Conference*, 1994.

[15] C. Kadow. Adaptive Dynamic Projection-Based Partitioning for Parallel Delaunay Mesh Generation Algorithms. In *SIAM Workshop on Combinatorial Scientific Computing*, February 2004.

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

[17] G. E. Karanidakis and S. A. Orszag. Nodes, Modes and Flow Codes. *Physics Today*, 46:34–42, 1993.

[18] L. Linardakis and N. Chrisochoides. Parallel Domain Decoupling Delaunay Method. *SIAM Journal on Scientific Computing*, in print, accepted Nov. 2004.

[19] 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.

[20] 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.

[21] D. Nave, N. Chrisochoides, and L. P. Chew. Guaranteed Quality Parallel Delaunay Refinement for Restricted Polyhedral Domains. In *SCG '02: Proceedings of the 18th Annual Symposium on Computational geometry*, pages 135–144. ACM Press, 2002.

[22] L. Oliker and R. Biswas. Parallelization of a Dynamic Unstructured Application Using Three Leading Paradigms. In *Supercomputing '99*, 1999.

[23] I. Park, B. Falsafi, and T. Vijaykumar. Implicitly-Multithreaded Processors. In *Proc. of the 30th Annual International Symposium on Computer Architecture (ISCA-2003)*, pages 98–109, San Diego, CA, June 2003.

[24] M.C. Rivara, D. Pizarro, and N. Chrisochoides. Parallel Refinement of Tetrahedral Meshes Using Terminal-Edge Bisection Algorithm. In *13th International Meshing Roundtable*, September 2004.

[25] 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.

[26] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Proceedings of the 1st Workshop on Applied Computational Geometry*, pages 123–133, Philadelphia, PA, 1996.

[27] J. R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, 1997.

[28] A. Snavely. Multi-processor Performance of the Tera MTA. In *Proc. of the IEEE/ACM Supercomputing'98: High Performance Networking and Computing Conference (SC'98)*, Orlando, Florida, November 1998.

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

[30] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, Orlando, FL, January 1999.

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

[32] S. Wallace, B. Calder, and D. Tullsen. Threaded Multiple Path Execution. In *Proc. of the 25th Annual International Symposium on Computer Architecture (ISCA–25)*, pages 238–249, Barcelona, Spain, June 1998.

[33] H. Wang, P. Wang, R. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. Speculative Precomputation: Exploring the Use of Multithreading for Latency. *Intel Technology Journal*, 6(1), February 2002.

[34] S. K. Warfield, M. Ferrant, X. Gallez, A. Nabavi, and F. A. Jolesz. Real-Time Biomechanical Simulation of Volumetric Brain Deformation for Image Guided Neurosurgery. In *Supercomputing*, 2000.

[35] D. F. Watson. Computing the n-Dimensional Delaunay Tesselation with Application to Voronoi Polytopes. *Computer Journal*, 24:167–172, 1981.

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