# Algorithm, Software, and Hardware Optimizations for Delaunay Mesh Generation on Simultaneous Multithreaded 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*

**Abstract**

This article focuses on the optimization of PCDM, a parallel, 2D Delaunay mesh generation application, and its interaction with parallel architectures based on simultaneous multithreading (SMT) processors. We first present the step-by-step effect of a series of optimizations on performance. These optimizations improve the performance of PCDM by up to a factor of six. They target issues that very often limit the performance of scientific computing codes. We then evaluate the interaction of PCDM with a real SMT-based SMP system, using both high-level metrics, such as execution time, and low-level information from hardware performance counters.

*Key words:* Parallel, Mesh, Generation, SMT, Optimizations, Finite Element,

* 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

Simultaneous multithreading (SMT) and multicore (CMP) processors have lately found their way in the product lines of all major hardware manufacturers [27–29]. These processors allow more than one threads to simultaneously execute on the same physical CPU. The degree of resource sharing inside the processor may range from sharing one or more levels of the cache (CMP processors), to almost fully sharing all processor resources (SMT processors). SMT and CMP chips offer a series of competitive advantages over conventional ones. They are, for example, characterized by better price to performance and power to performance ratios. As a consequence, they gain more and more popularity as building blocks of both multi-layer, high performance compute servers and off-the-shelf desktop systems.

The pervasiveness of SMT and CMP processors changes radically the software development process. Traditionally, evolution across different processor generations alone, would allow single-threaded programs to execute more and more efficiently. This trend, however, tends to diminish. SMT and CMP processors support, instead, thread-level parallelism within a single chip. As a result, parallel software is necessary in order to unleash the computational power of these chips by a single application. Needless to say, rewriting existing sequential software or developing from scratch parallel software comes at an increased cost and complexity. In addition, the development of efficient code for SMT and CMP processors is not an easy task. Resource sharing inside the chip makes performance hard to analyze and optimize, since performance is dependent not only on the interaction between individual threads and the hardware, but also on non-trivial interference between threads on resources

such as caches, TLBs, instruction queues, and branch predictors.

The trend of massive code development or rewriting restates traditional software engineering tradeoffs between ease of code development and performance. For example, programmers may either reuse functionality offered by system libraries (synchronization primitives, STL data structures, memory management, etc.), or reimplement it from scratch, targeting high performance. They may or may not opt for complex algorithmic optimizations, balancing code simplicity and maintainability with performance.

In this paper we present the programming and optimization process of a 2D Parallel Constrained Delaunay Mesh (PCDM) generation algorithm on SMT and multi-SMT systems, with the goal of understanding the performance implications of SMT processors on adaptive and irregular parallel applications, and laying out an optimization methodology, with elements that can be reused across irregular applications. Mesh generation is a central building block of many applications, in the areas of engineering, medicine, weather prediction, etc. PCDM is an irregular, adaptive, memory-intensive, multi-level and multi-grain parallel implementation of Delaunay mesh generation. We select PCDM because it is a provably efficient algorithm that can both guarantee the quality of the final mesh, and achieve scalability on conventional clusters of SMPs, at the scale of 100 or more processors [13].

The main contribution of this paper is a set of algorithmic and systemic optimizations for adaptive and irregular parallel algorithms on SMT processors. In particular, the paper provides a better understanding of multi-level and multi-grain parallelization for layered multi-processors, where threads executing and sharing resources on the same processor are differentiated from

4

threads executed across processors. The algorithmic optimizations presented in this paper pertain to parallel mesh generation algorithms, whereas the systemic optimizations pertain to broader classes of parallel applications with irregular execution and data access patterns, such as N-body simulations and ray-tracing algorithms.

We discuss in detail the exploitation of each of the three parallelism granularities present in PCDM on a real, SMT-based multiprocessor. We present the step-by-step optimization of the code and quantify the effect of each particular optimization on performance. This gradual optimization process results in code that is up to six times faster than the original, unoptimized one. Moreover, the optimized code has sequential performance within 12.3% of Triangle [37], the best to our knowledge sequential Delaunay mesh generation code. The exploitation of parallelism in PCDM allows it to outperform Triangle, even on a single physical (SMT) processor. As a next step, we use low-level performance metrics and information attained from hardware performance counters, to accurately characterize the interaction of PCDM with the underlying architecture.

The rest of the paper is organized as follows: In Section 2 we discuss related work in the context of performance analysis and optimization for layered parallel architectures. In Section 3 we briefly describe the parallel Delaunay mesh refinement algorithm. Section 4 discusses the implementation and optimization of the multi-grain PCDM on an SMT-based multiprocessor. We study the performance of the application on the target architecture both macroscopically and using low-level metrics. Finally, Section 5 concludes the paper.

## 2 Related Work

Although layered multiprocessors have established a strong presence in the server and desktop markets, there is still considerable skepticism for deploying these platforms in supercomputing environments. One reason seems to be that the understanding of the interaction between computationally-intensive scientific applications and these architectures is rather limited. Most existing studies of SMT and CMP processors originate from the computer architecture domain and use conventional uniprocessor benchmarks such as SPEC CPU [24] and shared-memory parallel benchmarks such as SPEC OMP [5] and SPLASH-2 [42]. There is a notable absence of studies that investigate application-specific optimizations for SMT and CMP chips, as well as the architectural implications of SMT and CMP processing cores on real-world applications that demand high FPU performance and high intra-chip and off-chip memory bandwidth. Interestingly, in some real supercomputing installations based on multi-core and SMT processor cores, multi-core execution is de-activated, primarily due to concerns about the high memory bandwidth demands of multithreaded versions of complex scientific applications [2].

This paper builds upon an earlier study of a realistic application, PCDM, on multi-SMT systems [4], to investigate the issues pertinent to application optimization and adaptation to layered shared-memory architectures. Similar studies appeared recently in other application domains, such as databases [17, 43] and have yielded results that stir the database community to develop more architecture-aware DataBase Management System (DBMS) infrastructure [23]. Another recent study of several realistic applications, including molecular dynamics and material science codes, on a Power5-based system

with dual SMT-core processors [22], indicated both advantages and disadvantages from activating SMT, however the study was confined to execution times and speedups of out-of-the-box codes without providing further details.

## 3   Delaunay Mesh Generation

In this paper we focus on the parallel constrained Delaunay refinement algorithm for 2D geometries. Delaunay mesh generation offers mathematical guarantees on the quality of the resulting mesh [14, 20, 30, 35, 38]. In particular, one can prove that for a user-defined lower bound on the minimal angle (below 20.7°) the algorithm will terminate while matching this bound and produce a size-optimal mesh. It has been proven [31] that a lower bound on the minimal angle is equivalent to the upper bound on the circumradius-to-shortest edge ratio which we will use in the description of the algorithm. Another commonly used criterion is an upper bound on triangle area which allows to obtain sufficiently small triangles.

The sequential Delaunay refinement algorithm works by inserting additional — so-called Steiner — points into an existing mesh with the goal of removing poor quality triangles, in terms of either shape or size, and replacing them with better quality triangles. Throughout the execution of the algorithm the Delaunay property of the mesh is maintained: the mesh is said to satisfy the Delaunay property if every triangle's circumscribing disk (circumdisk) does not include any of the mesh vertices. Usually Steiner points are chosen in the centers (circumcenters) of circumdisks of bad triangles, although other choices are also possible [11]. For our analysis and implementation we use the Bowyer-Watson (B-W) point insertion procedure [7, 41] which consists of the following

7

steps: (1) the cavity expansion step: the triangles whose circumdisks include the new Steiner point $p$ are identified; they are called the cavity $\mathcal{C}(p)$; (2) the cavity triangulation step: the triangles in $\mathcal{C}(p)$ are deleted from the mesh; as a result, an untriangulated space with closed polygonal boundary $\partial\mathcal{C}(p)$ is created; (3) $p$ is connected with each edge of $\partial\mathcal{C}(p)$, and the newly created triangles are inserted into the mesh.

We explore three levels of granularity in parallel Delaunay refinement: coarse, medium, and fine. At the coarse level, the triangulation domain $\Omega$ is decomposed into subdomains $\Omega_i$ which are distributed among MPI processes and used as units of refinement. When Steiner points are inserted close to subdomain boundaries, the corresponding edges are subdivided, and `split` messages are sent to the MPI processes refining subdomains that share the specific edge, to ensure boundary conformity [13]. At the medium granularity level, the units of refinement are cavities; in other words, multiple Steiner points are inserted concurrently into a single subdomain. Since the candidate Steiner points can have mutual dependencies, we check for the conflicts and cancel some of the insertions if necessary. The problem of Delaunay-independent point insertion along with parallel algorithms which avoid conflicts is described in [9–12]. In this paper, however, we study a different approach which allows to avoid the use of auxiliary lattices and quadtrees, at the cost of rollbacks. Finally, at the fine granularity level, we explore the parallel construction of a single cavity (cavity expansion). This is achieved by having multiple threads check different triangles for inclusion into the cavity.

## 4 Implementation, Optimization and Performance Evaluation

In the following paragraphs we discuss the implementation and the optimization process of the three granularities of parallelism in PCDM and their combinations into a new multi-grain implementation we describe in [3]. We also provide insight on the interaction of the application with the hardware on a commercial, low-cost, SMT-based multiprocessor platform. Table 1 summarizes the technical characteristics of our experimental platform. The platform is a 4-way SMP system, with Intel Hyperthreaded (HT) processors. Intel HT processors are based on the simultaneous multithreading (SMT) architecture. Each processor can execute two threads simultaneously. Each thread has its own register file, however it shares the rest of the hardware of the processor (cache hierarchy, TLB, execution units, memory interface etc.) with the other thread. Intel HT processors have become popular in the context of both technical and desktop computing, due to the fact that they offer SMT capabilities at no additional cost[1]. The system has 2 GB of main memory and runs Linux (2.6.13.4 kernel). The compiler used to generate the executables is g++ from the 3.3.4 GNU compiler collection (gcc). Experimental results from larger parallel systems, as well as a direct comparison between different single-grain and multi-grain parallelization strategies appears in [3]. This paper focuses on the optimizations of PCDM, at each of the three levels of parallelization granularity.

Intel HT processors offer ample opportunities for performance analysis through

---

[1] The cost of an Intel HT processor was initially the same as that of a conventional processor of the same family and frequency. Gradually, conventional processors of the IA-32 family were withdrawn.

9

| 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 |
| OS | Linux 2.6.13.4 |
| Compiler | g++, gcc 3.3.4 |

Table 1

Configuration of the Intel HT Xeon-based SMP system used to evaluate the multi-grain implementation of PCDM and its interaction with layered parallel systems.

the performance monitoring counters [25]. The performance counters offer valuable information on the interaction between software and the underlying hardware. They can be used either directly [34], or through higher level data acquisition and analysis tools [1, 8, 18].

Throughout this section we present experimental results applying PCDM on a rocket engine pipe 2D cross-cut domain. The specific engine pipe has been used during the development process of a real rocket engine by NASA. A slight modification to the pipe, not backed up by a thorough simulation and study, resulted in a catastrophic crack, destroying both the pipe and the engine prototype. In the experiments, we generate a 2D mesh of the pipe, consisting of 10 million triangles. The reported execution times include the preprocessing overhead for the domain decomposition, the MPI startup cost, the overhead of reading the subdomains from disk, and the mesh refinement, i.e., the main computational part of the algorithm. We do not report the time required to output the final mesh to disk.

The rest of Section 4 is organized as follows: in Subsection 4.1 we discuss and evaluate the optimizations related to the coarse-grain parallelization level of

PCDM. In Subsection 4.2 we focus on the optimization process of the medium-grain parallelization level of PCDM. Subsection 4.3 briefly discusses the implementation of fine-grain PCDM and presents a low-level experimental analysis of the interaction of fine-grain PCDM with the hardware. Finally, in Subsection 4.4, we discuss the potential of using the additional execution contexts of an SMT processor as speculative precomputation vehicles.

*4.1 Coarse-grain PCDM*

As explained in Section 3, the coarse granularity of PCDM is exploited by dividing the whole spatial domain into multiple sub-domains, and allowing multiple MPI processes to refine different sub-domains. Different MPI processes need to communicate via `split` messages only whenever a point is inserted at a subdomain boundary segment, thus splitting the segment. Such messages can even be sent lazily; multiple messages can be aggregated to a single one, in order to minimize messaging overhead and traffic on the system. We have empirically set the degree of message aggregation to 128.

Each subdomain may be associated with a different refinement workload. We, thus, use domain over-decomposition as a simple, yet effective load balancing method. In our experiments we create 32 subdomains for each MPI process used [2].

Table 2 summarizes the execution time of the coarse-grain PCDM implementation for the generation of 10M triangles on our experimental platform. We

---

[2] The degree of overdecomposition is a tradeoff between the effectiveness of load balancing and the initial sequential preprocessing overhead

|  | Unoptimized | | Optimized | |
|---|---|---|---|---|
|  | 1 MPI/processor | 2 MPI/processor | 1 MPI/processor | 2 MPI/processor |
| 1 processor | 54.0 | 45.1 | 28.4 | 23.5 |
| 2 processors | 27.2 | 23.0 | 14.8 | 11.8 |
| 4 processors | 13.9 | 12.0 | 7.5 | 5.9 |

Table 2

Execution time (in sec) of the original (unoptimized), and the optimized coarse-grain PCDM implementation.

report execution times from using both 1 MPI process per physical processor or 2 MPI processes per physical processor (one per SMT execution context), both before and after applying the optimizations described in the following paragraphs. The optimizations resulted in code that is approximately twice as fast as the original code. Furthermore, the optimizations improved the scalability of the coarse-grain PCDM on a single SMT processor with two execution contexts. SMT speedups of the original code range from 1.15 to 1.19. SMT speedups of the optimized code range from 1.20 to 1.27. This scalability improvement comes in addition to improvements in sequential execution time.

The charts in Figure 1 itemize the effect of each optimization on execution time. The left chart depicts the execution time of the initial, unoptimized implementation (*original*), of the version after the substitution of STL data-structures (*STL*) described in section 4.1.1, after the addition of optimized memory management (*Mem Mgmt*) described in section 4.1.2, and after applying the algorithmic optimizations (*Algorithmic*) described in section 4.1.3. Similarly, the right diagram depicts the % performance improvement after the application of each additional optimization over the version that incorporates all previous optimizations. Due to space limitations, we report the effect of
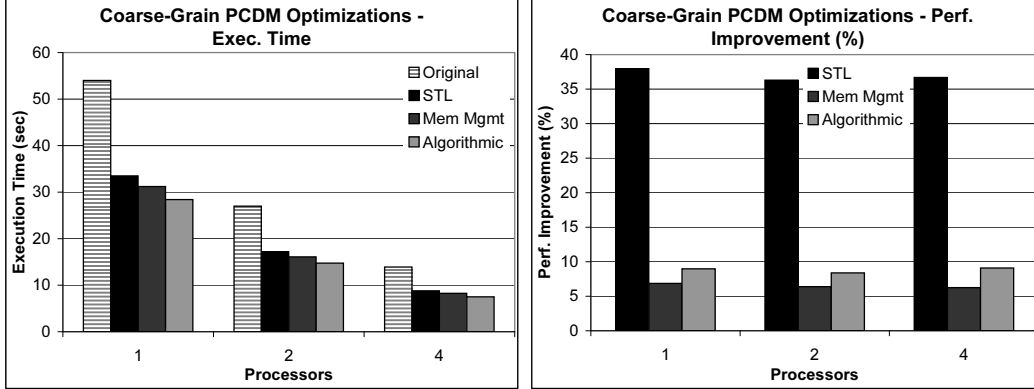
Fig. 1. Effect of optimizations on the performance of coarse-grain PCDM. Cumulative effect on execution time (left diagram). Performance improvement (%) of each new optimization over the coarse-grain PCDM implementation with all previous optimizations applied.

optimizations on the coarse-grain PCDM configurations using 1 MPI process per physical processor. Their effect on configurations using 2 MPI processes per physical processor is quantitatively very similar.

### 4.1.1 Substitution of Generic STL Data-Structures

The original, unoptimized version of coarse-grain PCDM makes extensive use of STL structures. Although using STL constructs has several software engineering advantages in terms of code readability and code reuse, such constructs often introduce unacceptable overhead.

In PCDM, the triangles (elements) of the mesh that do not satisfy the quality bounds are placed in a work-queue. For each of these so called *bad* triangles, PCDM inserts a point into the mesh, at the circumcenter of the element. The insertion of a new point forces some elements around it to violate the Delaunay property. The identification of these non-Delaunay elements is called a *cavity*

13

*expansion.*

During the cavity expansion phase, PCDM performs a depth-first search of the triangles graph, the graph in which a triangle is connected with the three neighbors it shares faces with. The algorithm identifies triangles included in the cavity, and those that belong to the closure of the cavity, i.e., triangles that share an edge with the boundary of the cavity. The population of these two sets for each cavity is *a priori* unknown, thus the original PCDM uses STL vectors for the implementation of the respective data structures, taking advantage of the fact that STL vectors can be extended dynamically. Similarly newly created triangles, during cavity re-triangulations, are accommodated in an STL vector as well.

We replaced these STL vectors with array-based LIFO queues. We have conservatively set the maximum size of each queue to 20 elements, since our experiments indicate that the typical population of these queues is only 5–6 triangles for 2D geometries. In any case, a dynamic queue growth mechanism is present and is activated in the infrequent case triangles overflow one of the queue arrays. Replacing the STL vectors with array-based queues improved the execution time of coarse-grain PCDM by an average 36.98%.

### 4.1.2 Memory Management

Mesh generation is a memory intensive process, which triggers frequent memory management (allocation / deallocation) operations. The unoptimized implementation of coarse-grain PCDM includes a custom memory manager. The memory manager focuses on efficiently recycling and managing triangles, since they are by far the most frequently used data structure of PCDM.

After a cavity is expanded, the triangles included in the cavity are deleted and the resulting empty space is then re-triangulated. The memory allocated for deleted triangles is never returned to the system. Deleted triangles are, instead, inserted in a recycling list. The next time the program requires memory for a new triangle (during retriangulation), it reuses deleted triangles from the recycling list. Memory is allocated from the system only when the recycling list is empty.

During mesh refinement, the memory footprint of the mesh is monotonically increasing, since during the refinement of a single cavity the number of deleted triangles is always less than or equal to the number of created triangles. As a result, memory is requested from the system during every single cavity expansion. The optimized PCDM implementation pre-allocates pools (batches) of objects instead of allocating individual objects upon request. We experimentally determined that memory pools spanning the size of 1 page (4Kb for our experimental platform) resulted in the best performance. When all the memory from the pool is used, a new pool is allocated from the system. Batch memory allocation significantly reduces the pressure on the system's memory manager and improves the execution time of coarse-grain PCDM approximately by an additional 6.5%. The improvement from batch allocation of objects, stems from reducing the calls to the system memory allocator and from improved cache-level and TLB-level locality. Although generic sequential and multi-threaded memory allocators also manage memory pools internally for objects of the same size [6, 19, 21, 36] each allocation-deallocation of an object from/to a pool, carries the overhead of two library calls. Custom batch memory allocation nullifies this overhead.

### 4.1.3 Algorithmic Optimizations

Balancing algorithmic optimizations that target higher performance or lower resource usage, with code simplicity, readability and maintainability is an interesting exercise during code development for scientific applications. When high performance is the main consideration, the decision is usually in favor of the optimized code.

In the case of PCDM, we performed limited, localized modifications in a single, critical computational kernel of the original version. The modifications targeted the reduction or elimination of costly floating-point operations on the critical path of the algorithm.

The specific kernel evaluates the quality of a triangle, by comparing its minimum angle with a predefined, user-provided threshold. Lets assume that $\widehat{C}$ is the minimum angle of triangle $ABC$ and $\widehat{L}$ is the threshold angle. The original code would calculate $\widehat{C}$ from the coordinates of triangle points, using the inner product formula $\widehat{C} = \arccos \frac{\langle \overrightarrow{a}, \overrightarrow{b} \rangle}{\|\overrightarrow{a}\| \cdot \|\overrightarrow{b}\|}$ for the calculation of the angle $\widehat{C}$ between vectors $\overrightarrow{a}$ and $\overrightarrow{b}$. The kernel would then compare $\widehat{C}$ with $\widehat{L}$ to decide whether the specific triangle fulfilled the user-defined quality criteria or not. However, the calculation of $\widehat{C}$ involves costly *arccos* and *sqrt* operations (the latter for the calculation of $\|\overrightarrow{a}\| \cdot \|\overrightarrow{b}\|$).

The algorithmic optimizations are based on the observation that, since $\widehat{C}$ and $\widehat{L}$ represent minimum angles of triangles, they are both less than $\frac{\pi}{2}$. As a result, $\cos \widehat{C}$ and $\cos \widehat{L}$ both $\in (0, 1)$, with cos being a monotonically decreasing function of the angle in the interval $(0, \frac{\pi}{2})$. Therefore, instead of comparing $\widehat{C}$ with $\widehat{L}$ one can equivalently compare $\cos \widehat{C}$ with $\cos \widehat{L}$. This eliminates a time-consuming *acos* operation every time a new triangle is created.

Furthermore, since $\cos\widehat{C}$ and $\cos\widehat{L}$ are both positive, one can equivalently compare $\cos^2\widehat{C}$ with $\cos^2\widehat{L}$. This, in turn, eliminates the *sqrt* operation for the calculation of $\|\overrightarrow{a}\| \cdot \|\overrightarrow{b}\|$.

The specific algorithmic optimizations improved further the execution time of coarse-grain PCDM by an average 8.82%.

## 4.2 Medium-grain PCDM

The medium-grain PCDM implementation spawns threads inside each MPI process. These threads cooperate for the refinement of a single subdomain, by simultaneously expanding different cavities. The threads of each MPI process are bound one-by-one to the execution contexts of a physical processor.

|              | Unoptimized | Optimized |
|--------------|-------------|-----------|
| 1 processor  | 155.0       | 25.9      |
| 2 processors | 80.4        | 13.36     |
| 4 processors | 42.3        | 6.94      |

Table 3

Execution time (in sec) of the original (unoptimized), and the optimized medium+coarse multi-grain PCDM implementation.

Table 3 summarizes the execution time of a multi-grain PCDM implementation that exploits coarse-grain parallelism across processors and medium-grain inside each SMT processor (2 execution contexts per processor for our experimental platform, executing one medium-grain thread each). The unoptimized multi-grain implementation performs almost 3 times worse than the unoptimized coarse-grain one. However, our optimizations result in code that is

17

approximately 6 times faster than the original, unoptimized implementation. The exploitation of the second execution context of each SMT processor allows optimized multi-grain PCDM to outperform the optimized coarse-grain configuration which exploits only one SMT execution context on each physical processor. It is, however, up to 4 processors, slightly less efficient than the coarse-grain configuration that executes 2 MPI processes on each CPU [3].
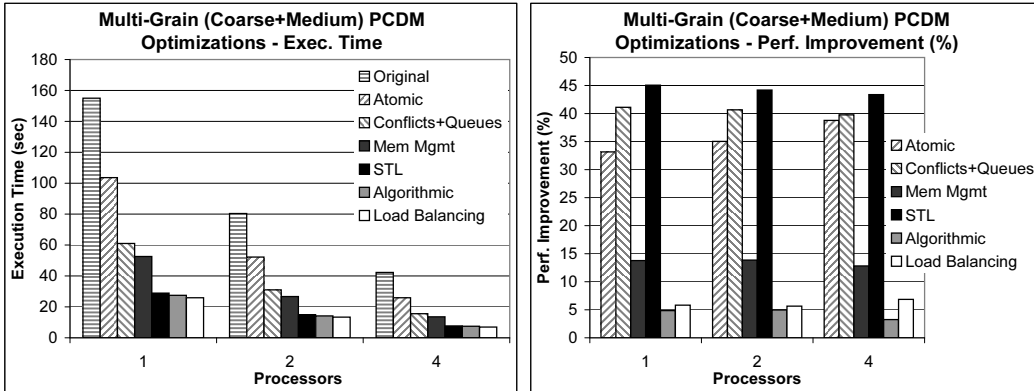


Fig. 2. Effect of optimizations on the performance of multi-grain (coarse+medium) PCDM. Cumulative effect on execution time (left diagram). Performance improvement (%) of each new optimization over the multi-grain PCDM implementation with all previous optimizations applied.

The charts of Figure 2 itemize the effect of each optimization on execution time of the multi-grain (coarse+medium) implementation of PCDM. The left

---

[3] In [3] we evaluate PCDM on larger-scale systems. We find that the use of additional MPI processes comes at the cost of additional preprocessing overhead and we identify cases in which the combination of coarse-grain and medium-grain (coarse+medium) PCDM proves more efficient than a single-level coarse-grain approach. Furthermore, in [3], we evaluate the medium-grain implementation of PCDM on IBM Power5 processors, in which the cores have a seemingly more scalable implementation of the SMT architecture, compared to the older Intel HT processors used in this study.

chart depicts the execution time of:

- The original, unoptimized implementation (*original*),

- The version after the efficient implementation of synchronization operations (*atomic*), discussed in Section 4.2.1,

- The version after the minimization of conflicts and the implementation of a multi-level work queue scheme (*Conflicts+Queues*), described in Sections 4.2.2 and 4.2.3,

- The code resulting after the optimization of memory management (*Mem Mgmt*), covered in Section 4.2.4,

- The version incorporating the substitution of STL with generic data-structures (*STL*), discussed in Section 4.2.5,

- The code resulting after the algorithmic optimizations (*Algorithmic*), described in Section 4.2.6, and

- The version resulting after the activation of dynamic load balancing (*Load Balancing*), discussed in Section 4.2.7.

Similarly, the right chart depicts the % performance improvement after the application of each additional optimization over the version that incorporates all previous optimizations.

### 4.2.1   Synchronization

A major algorithmic concern for medium-grain PCDM is the potential occurrence of conflicts while threads are simultaneously expanding cavities. Multiple threads may work on different cavities at the same time, within the same domain. A conflict occurs if any two cavities —processed simultaneously by different threads— overlap, i.e., have a common triangle or share an edge. In

19

this case, only a single cavity expansion may continue; the rest need to be canceled. This necessitates a conflict detection and recovery mechanism.
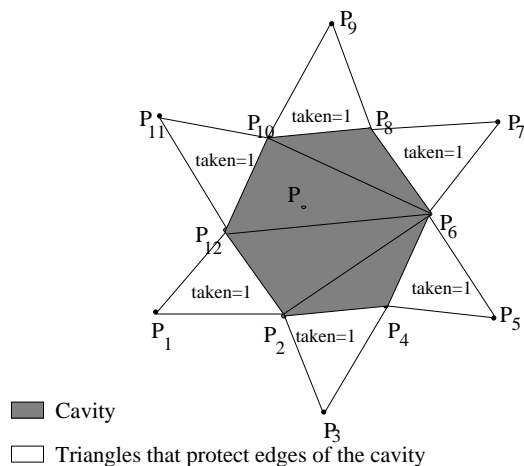


Fig. 3. Layer of triangles that surround a cavity (*closure* of the cavity).

Each triangle is tagged with a flag (`taken`). Whenever a triangle is touched during a cavity expansion (either because it is actually part of the cavity itself or of its closure), the flag is set. The closure of the cavity, namely this extra layer of triangles that surround the cavity —without being part of it— prevents two cavities from sharing an edge (Figure 3) [15, 32]. If, during a cavity expansion, a thread touches a triangle whose flag has already been set, the thread detects a conflict. The cavity expansion must then be canceled.

Updates of the flag variable need to be atomic since two or more threads may access the same triangle simultaneously. Every access to the triangle's flag is performed through atomic `fetch_and_store()` operations. These instructions incur —on the vast majority of modern shared-memory architectures— less overhead than conventional locks or semaphores under high contention, while providing additional advantages such as immunity to preemption. The use of atomic instructions resulted in 33% to 39% faster code than an alternative, naive implementation using POSIX lock/unlock operations for the protection

of the flag.

## 4.2.2  Reduction of Conflicts

The cancellations of cavity expansions —as a consequence of conflicts— directly result in the discarding of already performed computation. The canceled cavity expansion will have to be restarted. It is, thus, critical for performance to minimize the occurrence of conflicts.
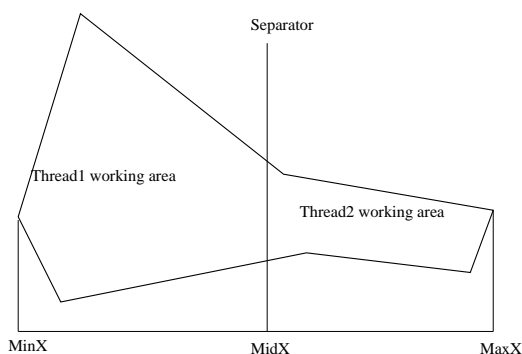


Fig. 4. Separator that splits a sub-domain into different areas.

The optimized multi-grain PCDM implementation isolates each thread to a single area of the sub-domain (Figure 4). We apply a straightforward, computationally inexpensive decomposition, using simple, straight segments, by subdividing an enclosing rectangular parallelepiped of the sub-domain. If, for example, two threads are used, the separator is a vertical line at the middle between the leftmost and rightmost coordinate of the sub-domain. After the isolation of different threads to different working areas, conflicts are likely to occur only close to the borders between areas. Moreover, the probability of conflicts decreases as the quality of the mesh improves [9].

Table 4 summarizes the number of conflicts before and after splitting the

|  | Number of Expanded Cavities | Conflicts Before Splitting | Conflicts After Splitting |
|---|---|---|---|
| Thread 1 | 2,453,034 | 1,199,184 | 3,005 |
| Thread 2 | 2,462,935 | 1,142,578 | 2,603 |
| Total | 4,915,969 | 2,341,762 | 5,608 |

Table 4

Number of conflicts before and after splitting (in two) the working area inside each sub-domain.

working area of each sub-domain. Additional performance data are provided in section 4.2.3[4]. Statically splitting sub-domains is prone to introducing load imbalance among threads. The technique applied to resolve load imbalance is described in Section 4.2.7.

### 4.2.3   Work-Queues Hierarchy

PCDM maintains a global queue of "bad" triangles, i.e., triangles that violate quality criteria. Whenever a cavity is re-triangulated, the quality of the new triangles is checked, and any offending triangle is placed into the queue. Throughout the refinement process threads poll the queue. As long as it is not empty, they retrieve a triangle from the top, and start a new cavity expansion. In medium-grain PCDM, the queue is concurrently accessed by multiple threads and thus needs to be protected.

A straightforward solution for reducing the overhead due to contention is to

[4] The implementation of the conflict reduction technique is interdependent with the work-queues hierarchy design and implementation, presented later in section 4.2.3. As a result the effect of each of these two optimizations on execution can not be isolated and evaluated separately.

use local, per thread queues of bad triangles. Bad triangles that belong to a specific working area of the sub-domain are inserted into the local list of the thread working in that area. Since, however, a cavity can cross the working area boundaries, a thread can produce bad triangles situated at areas assigned to other threads. As a result, local queues of bad triangles still need to be protected, although they are significantly less contended than a single global queue.
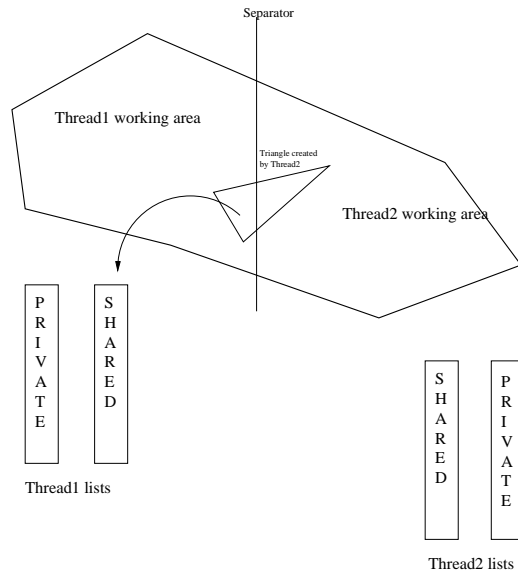


Fig. 5. Local shared and private queues for each thread.

A hierarchical queue scheme with two local queues of bad triangles per thread is applied to further reduce locking and contention overhead. One queue is strictly private to the owning thread, while the other can be shared with other threads, and therefore needs to be protected. If a thread, during a cavity re-triangulation, creates a new bad triangle whose circumcenter is included it its assigned working area, the new triangle is inserted in the private local queue. If, however, the circumcenter of the triangle is located in the area assigned to another thread, the triangle is inserted in the shared local queue of that thread (Figure 5). Each thread dequeues triangles from its private queue as

long as the private queue is not empty. Only whenever the private queue is found empty shall a thread poll its shared local queue.

As expected, the private local queue of bad triangles is accessed much more frequently than the shared local one. During the creation of the mesh of 10M triangles for the pipe domain, using two threads to exploit medium-grain parallelism, the shared queues of bad triangles are accessed 800,000 times, while the private ones are accessed more than 12,000,000 times. Therefore, the synchronization overhead for the protection of the shared queues is practically negligible. Contention and synchronization overhead could be reduced further if a thread moves the entire contents of the shared local queue to the private local queue, upon an access to the shared local queue. However, such a scheme would compromise load balancing, as discussed in Section 4.2.7.

The average performance improvement after reducing cavity expansion conflicts and using the 2-level queue scheme is 40.52%.

### 4.2.4   Memory Management

The memory recycling mechanism of PCDM, described in Section 4.1.2, is not efficient-enough in the case of medium-grain PCDM for two reasons:

- The recycling list is shared between threads and thus accesses to it need to be protected.
- Memory allocation/deallocation requests from different threads cause contention inside the system's memory allocator. Such contention may result in severe performance degradation for applications with frequent memory management operations.

In the optimized medium-grain PCDM we associate a local memory recycling list with each thread. Local lists alleviate the problem of contention at the level of the recycling list and eliminate the respective synchronization overhead. A typical concern whenever private per thread lists are used, is the potential imbalance in the population of the lists. This is, however, not an issue in the case of PCDM since, as explained in Section 4.1.2, the population of triangles either remains the same or increases during every single cavity refinement.

To reduce pressure on the system's memory allocator, medium-grain PCDM also uses memory pools. The difference with coarse-grain PCDM is that memory pools are thread-local and thus do not need to be protected.

The execution time of coarse+medium grain PCDM, after memory management-related optimizations were applied, further improved on average by 13.49%.

### 4.2.5 Substitution of STL Data-Structures

In section 4.1.1 we described the substitution of STL constructs with generic data structures (arrays) in the code related to cavity expansion. This optimization is applicable to the medium-grain implementation of PCDM code as well.

The average performance improvement by substituting STL constructs with generic data structures is in the order of 44.21%, 7.21% higher than the performance improvement attained by substituting STL data structures in the coarse-grain PCDM implementation. STL data structures introduce additional overhead when used in multi-threaded code, due to the mechanisms used by STL to guarantee thread-safety.

### 4.2.6 Algorithmic Optimizations

The algorithmic optimizations described in section 4.1.3 are applicable in the case of medium-grain PCDM as well. In fact, on SMT processors such operations, besides their cost, can become a serialization point if the floating-point hardware is shared between threads [40]. These modifications improved the execution time of medium-grain PCDM by approximately 4.35%.

### 4.2.7 Load Balancing

As explained in Section 4.2.2, each sub-domain is divided up into distinct areas, and the refinement of each area is assigned to a single thread. The decomposition is performed by equipartitioning — using straight lines as separators — of a rectangular parallelepiped enclosing the subdomain. Despite being straightforward and computationally inexpensive, this type of decomposition can introduce load imbalance between threads for irregular subdomains (Figure 6).
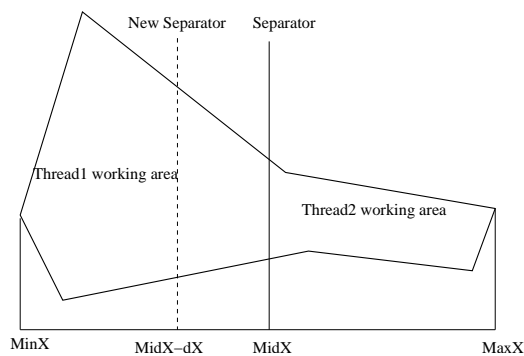


Fig. 6. Uneven work distribution between threads. A "moving separator" technique is used for fixing the load imbalance.

The load imbalance can be alleviated by dynamically adjusting the position of the separators at runtime. The size of the queues (private and shared) of bad

quality triangles is proportional to the work performed by each thread. Large differences in the populations of queues of different threads at any time during the refinement of a single sub-domain are a safe indication of load imbalance. Such events are, thus, used to trigger the load balancing mechanism. Whenever the population of the queues of a thread becomes larger than (100 / Number of Threads)% compared with the population of the queues of a thread processing a neighboring area, the separator between the areas is moved towards the area of the heavily loaded thread (Figure 6). The population of the queues of each thread needs to be compared only with the population of the queues of threads processing neighboring areas. The effects of local changes in the geometry of areas tend to quickly propagate — similarly to a domino effect — to the whole sub-domain, resulting in a globally (intra sub-domain) balanced execution.
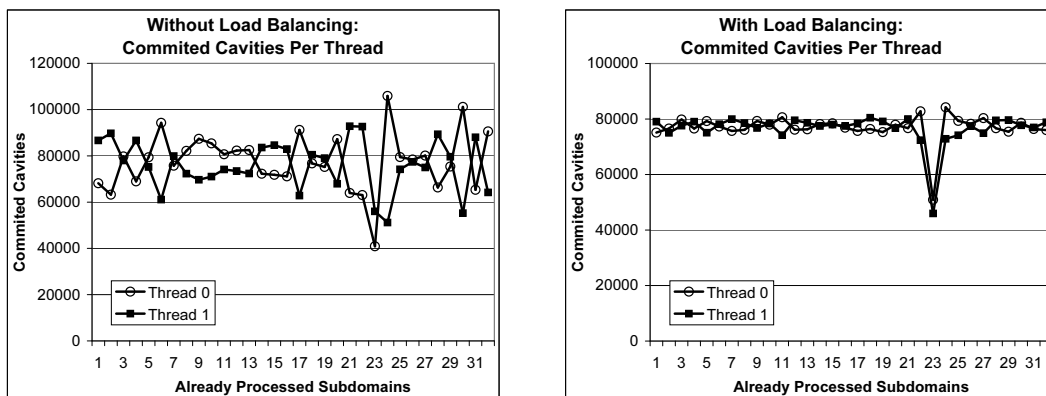


Fig. 7. Difference in the number of processed cavities without (left diagram) and with (right diagram) load balancing.

Figure 7 depicts the difference in the number of processed cavities between two medium-grain PCDM threads that cooperate in the processing of the same sub-domains. In both figures, the x-axis represents the id of the sub-domain being refined, while the y-axis represents the number of expanded cavities by each thread for the specific sub-domain. Before the activation of the load balancing mechanism, there are sub-domains for which a thread processes twice

27

as many cavities as the other thread. On the other hand, when the load balancing technique is activated, both threads perform approximately the same amount of work (cavity expansions) for each sub-domain. The moving separator scheme manages to eliminate work imbalance among threads, at the cost of monitoring the length of triangle lists, re-calculating separators and moving unprocessed triangles between lists upon re-balancing operations. Overall, the execution time improvement attained through better load balancing averages 6.11%.

## 4.3   Fine-grain PCDM

Fine-grain PCDM also spawns threads (a *master* and one or more *workers*) inside each MPI process. The difference with the medium-grain PCDM implementation is that in the fine-grain case the threads cooperate for the expansion of a single cavity. Cavity expansions account for 59% of the total PCDM execution time.

The *master* thread behaves similarly to a coarse-grain MPI process. *Worker* threads assist the *master* during cavity expansions and are idle otherwise. Triangles that have already been tested for inclusion into the cavity have to be tagged so that they are not checked again during the expansion of the same cavity. Similarly to the medium-grain PCDM implementation, we use atomic `test_and_set()` operations to atomically test the value of and set a flag. Each thread queues/dequeues unprocessed triangles to/from a thread-local queue. As soon as the local queue is empty, threads try to steal work from the local queues of other threads. Since the shape of a cavity is, unlike the shape of a sub-domain, not *a priori* known, techniques such as the multi-

28

level queue scheme and the dynamically moving boundary (Sections 4.2.2 and 4.2.3) can not be used in the case of fine-grain PCDM to isolate the working areas of different threads. Accesses to local queues are thus protected with synchronization mechanisms similar to those proposed in [33].

Many of the optimizations described in 4.1 and 4.2 (more specifically *atomic*, *Mem Mgmt*, *STL*, and *Algorithmic*) are applicable in the case of fine-grain PCDM. However, we do not present experimental results on the effect of each particular optimization on performance. We, instead, opt —due to space limitations— to use a fully optimized version and investigate both qualitatively and quantitatively the interaction of that code with SMT processors. The observations from this study can also be generalized in the context of other irregular, multi-grain codes, such as multi-level parallel implementations of N-body calculations [26].

### 4.3.1 Experimental Study

We executed a version of PCDM which exploits both the fine and the coarse granularities of parallelism (*Coarse+Fine*). The fine granularity is exploited within each MPI process, by the two execution contexts available on each HT processor. Multiple MPI processes —on multiple processors— are used to exploit the coarse granularity. We compare the performance of the multi-grain (coarse+fine) version with that of the single-level coarse-grain implementation of PCDM. We have executed two different configurations of the coarse-grain experiments: either one (*Coarse (1 MPI/proc)*) or two (*Coarse (2 MPI/proc)*) MPI processes are executed on each physical processor. In the latter case, the coarse-grain implementation alone exploits all execution contexts of the

system. We also compare the performance of PCDM with that of Triangle [37].



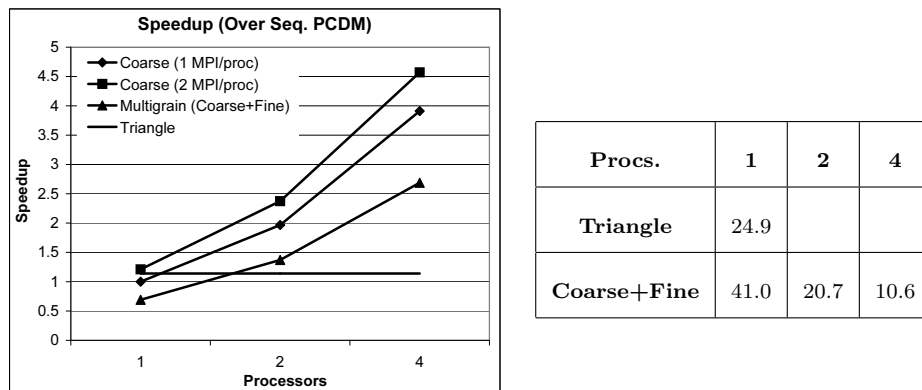| Procs. | 1 | 2 | 4 |
|---|---|---|---|
| Triangle | 24.9 | | |
| Coarse+Fine | 41.0 | 20.7 | 10.6 |

Fig. 8. Speedup with respect to the single-threaded PCDM execution. The table reports the corresponding execution times (in sec) for *Triangle* and *Coarse+Fine*. The respective execution times for *Coarse (1 MPI/proc)* and *Coarse (2 MPI/proc)* can be found in table 2.

Figure 8 depicts the speedup with respect to a sequential PCDM execution. On the specific system, Triangle is 12.3% faster than the optimized, sequential PCDM. The multilevel PCDM code (*Coarse+Fine*) does not perform well. In fact, a slowdown of 44.5% occurs as soon as a second thread is used to take advantage of the second execution context of the HT processor. The absolute performance is improved as more physical processors are used (2 and 4 processors, 4 and 8 execution contexts respectively). However, the single-level version, even with 1 MPI process per processor, consistently outperforms the multi-grain one (by 43.6% on 2 processors and by 45.5% on 4 processors). The performance difference is even higher compared with the coarse-grain configuration using 2 MPI processes per processor. In any case, single- or multi-level (coarse+fine), 2 processors are sufficient for PCDM to outperform the extensively optimized, sequential Triangle, whereas *Coarse (2 MPI/proc)* manages to outperform Triangle even on a single SMT processor.

We used the hardware performance counters available on Intel HT processors, in an effort to identify the reasons that lead to significant performance penalty whenever two execution contexts per physical processor are used to exploit the fine granularity of PCDM. 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 Figures 9a and 9b respectively. Ratios have been calculated with respect to the sequential PCDM execution.
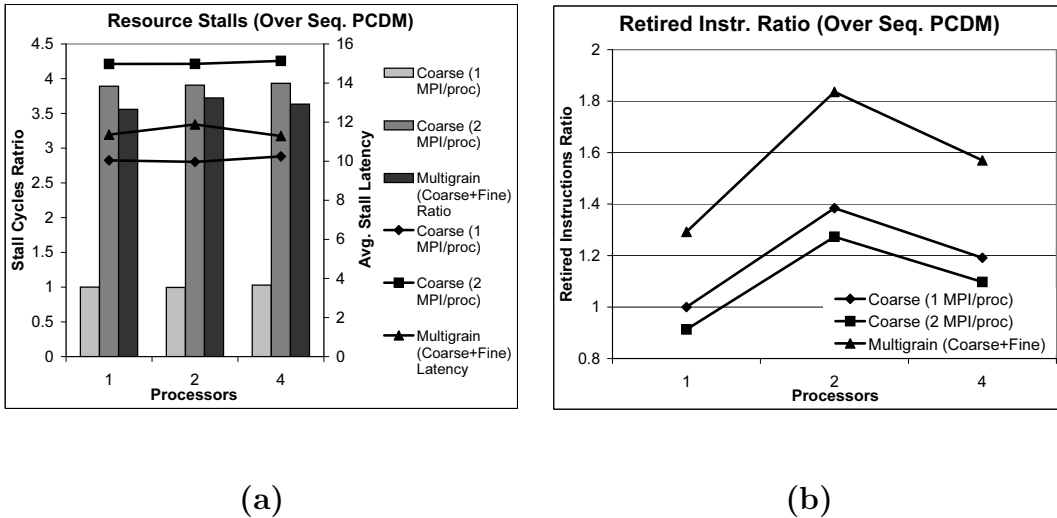


(a)                                                   (b)

Fig. 9. (**a**) Normalized number of stall cycles (with respect to the sequential PCDM execution) and average stall latency, in cycles. (**b**) Normalized number of retired instructions (with respect to the sequential PCDM execution).

The number of stall cycles (Fig. 9a) 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. Whenever two threads share the same processor, the stall cycles are from 3.6 to 3.7 times more for *Coarse+Fine* and 3.9 times more for *Coarse (2 MPI/proc)*. Exploiting the two execution contexts of each HT processor with two MPI processes seems to introduce more stalls. It should, however, be noted that the *worker* thread in the *Coarse+Fine* implementation performs useful computation only during cavity expansions, which account for 59% of the execution time of sequential PCDM. On the contrary, *Coarse (2 MPI/proc)* MPI processes perform useful computation throughout the execution life of the application.

Resource sharing inside the processor has a negative effect on the average latency associated with each stall as well. The average latency is 10 cycles when one thread is executed on each physical processor. When two MPI processes share the same processor it raises to approximately 15 cycles. When two threads that exploit the fine-grain parallelism of PCDM are co-located on the same processor the average latency ranges between 11.3 and 11.9 cycles.

Interesting information is also revealed by the number of retired instructions (Fig. 9b). Whenever two processors are used, the total number of instructions always increases by a factor of approximately 1.4 —with respect to the corresponding single-processor experiments— for the two coarse configurations and the coarse+fine version. 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 total number of

instructions.

## 4.4   Alternative Methods for the Exploitation of Execution Contexts

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) [16] for speeding up such codes on SMTs and CMPs [16, 39]. SPR exploits one of the execution contexts of the processor in order to precompute addresses of memory accesses that lead to cache misses and pre-execute 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 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. As explained in the previous section, 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. 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 — as explained earlier — the

latency of a sleep/wakeup cycle spans the expansion time of hundreds of cavities. An additional problem is that the maximum possible run-ahead distance between the precomputation and computation thread is equal to the degree of available concurrency, namely the number of unprocessed elements in the "bad" triangles queues. This number equals approximately 2 in our fine-grain 2D experiments. This precludes the use of the precomputation thread in batch precompute/sleep cycles.

## 5 Conclusions

As SMT processors become more widespread, parallel systems are being built using one or more of these processors. The ubiquitousness of SMT processors necessitates a shift towards parallel programming, especially in the context of scientific computing. The development of parallel codes is not an easy undertaking, especially if high performance is the end-goal. Code optimization is a valuable step of the development process, however the programmer has to both identify performance bottlenecks and evaluate complex tradeoffs. At the same time, adaptive and irregular applications are a challenging target for any parallel architecture. Investigating whether emerging parallel architectures are well suited for such applications is, therefore, an important undertaking. Our paper makes contributions towards these directions, focusing on PCDM, an multi-level, multi-grain parallel mesh generation code. PCDM is representative of adaptive and irregular parallel applications that present several challenges to parallel execution hardware, including fine-grain task execution and synchronization, load imbalance and poor data access locality. PCDM was selected due to the fact that it is a scalable parallel implementation of mesh

34

generation, which at the same time guarantees the quality of the final mesh.

We first presented a step-by-step optimization of the two outer granularities of PCDM. Despite the fact that PCDM is the direct target of these optimizations, most of them are generic enough to be applicable to other applications of the same class. We evaluated and presented the effect of each individual optimization on performance. The resulting optimized code was up to 6 times more efficient than the original one.

As modern parallel systems integrate many execution contexts organized — due to technical limitations— in more and more levels, system architects are faced with a choice between performance and programmability. They can present all the computational resources of the system to the programmer in a uniform way, in order to facilitate programming. Alternatively, they can export details of the architecture to the programmer, by differentiating the handling of computational resources at different levels of the system, thus enabling the efficient execution of demanding codes. Most commercially available multiprocessors (based on Intel HT, AMD Opteron or IBM Power processors) follow the former approach. A recent, notable exception in this trend has been the Sony/Toshiba/IBM Cell chip.

Next-generation system software has a significant role in this emerging environment; it can bridge these two alternatives. New compilers, operating system kernels and run-time libraries need to be developed specifically for layered parallel architectures, with the goal of hiding complex architectural details from the programmer, but at the same time exploiting in an educated manner the structural organization of the hardware in order to unleash the performance potential of modern parallel architectures.

## Acknowledgments

## References

[1] Intel VTune Performance Analyzer for Linux. http://www.intel.com/cd/software/products/asmona/eng/vtune/index.htm, 2006. Intel Corporation.

[2] NERSC Bassi System Administrators. Personal communication. http://www.nersc.gov/nusers/resources/bassi/, 2006.

[3] Christos D. Antonopoulos, Filip Blagojevic, Andrey N. Chernikov, Nikos P. Chrisochoides, and Dimitris S. Nikolopoulos. A multigrain Delaunay mesh generation method for multicore SMT-based architectures. *Journal on Parallel and Distributed Computing*. Submitted in Aug. 2006.

[4] 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.

[5] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B.

Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, London, UK, 2001. Springer-Verlag.

[6] E. Berger, K. Mckinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, November 2000.

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

[8] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (Supercomputing '00) (CDROM)*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[9] 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.

[10] 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.

[11] Andrey N. Chernikov and Nikos P. Chrisochoides. Generalized Delaunay mesh refinement: From scalar to parallel. In *Proceedings of the 15th International Meshing Roundtable*, pages 563–580, Birmingham, AL, September 2006. Springer.

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

[13] 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.

[14] L. Paul Chew. Guaranteed quality mesh generation for curved surfaces. In *Proceedings of the 9th ACM Symposium on Computational Geometry*, pages 274–280, San Diego, CA, 1993.

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

[16] 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.

[17] C. Colohan, A. Ailamaki, J. Gregory Steffan, and T. Mowry. Optimistic Intra-Transaction Parallelism on Chip Multiprocessors. In *Proc. of the 31st International Conference on Very Large Databases*, pages 73–84, Trondheim, Norway, August 2005.

[18] Matthew Curtis-Maury, Christos D. Antonopoulos, and Dimitrios S.Nikolopoulos. PACMAN: A PerformAnce Counters MANager for Intel Hyperthreaded Processors. In *Proceedings of the Third International Conference on the Quantitative Evaluation of Systems (QEST'06), Tools Session*, Riverside, CA, USA, September 2006. IEEE Computer Society Press.

[19] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proc. of*

the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 313–323, Montreal, Canada, June 1998.

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

[21] Google. Google performance tools. http://goog-perftools.sourceforge.net/.

[22] A. Gray, J. Hein, M. Plummer, A. Sunderland, L. Smith, A. Simpson, and A. Trew. An Investigation of Simultaneous Multithreading on HPCx. Technical Report 0604, EPCC – University of Edinburgh, April 2006.

[23] Stavros Harizopoulos and Anastassia Ailamaki. StagedDB: Designing Database Serverse for Modern Hardware. *IEEE Data Engineering Bulletin*, 28(2):11–16, 2005.

[24] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[25] Intel Corporation. *IA-32 Intel Architecture Software Developers Manual. Volume 3B: System Programming Guide, Part 2*, March 2006.

[26] Jess A. Izaguirre, Scott S. Hampton, and Thierry Matthey. Parallel Multigrid Summation for the N-body Problem. *Journal of Parallel and Distributed Computing*, 65:949–962, 2005.

[27] R. Kalla, B. Sinharoy, and J. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March 2004.

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

[29] D. Koufaty and D. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[30] Gary L. Miller. A time efficient Delaunay refinement algorithm. In *Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms*, pages 400–409, New Orleans, LA, 2004.

[31] 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.

[32] 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.

[33] 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.

[34] Mikael Pettersson. Perfctr: Linux Performance Monitoring Counters Kernel Extension. http://user.it.uu.se/ mikpe/linux/perfctr/current, June 2006.

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

[36] S. Schneider, C. Antonopoulos, and D. Nikolopoulos. Scalable Locality-Conscious Multithreaded Memor Allocation. In *Proc. of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pages 84–94, Ottawa, Canada, June 2006.

[37] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*,

volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

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

[39] 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.

[40] Tanping Wang, Filip Blagojevic, and Dimitrios S. Nikolopoulos. Runtime Support for Integrating Precomputation and Thread-Level Parallelism on Simultaneous Multithreaded Processors. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems (LCR '04)*, pages 1–12, New York, NY, USA, 2004. ACM Press.

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

[42] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, New York, NY, USA, 1995. ACM Press.

[43] Jingren Zhou, John Cieslewicz, Kenneth Ross, and Mihir Shah. Improving Database Performance on Simultaneous Multithreading Processors. In *Proc. of the 31st International Conference on Very Large Databases*, pages 49–60, Trondheim, Norway, June 2005.