



Algorithm, software, and hardware optimizations for Delaunay mesh generation on simultaneous multithreaded architectures

Christos D. Antonopoulos^a, Filip Blagojevic^d, 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, United States

^c Department of Computer Science, The College of William and Mary, Williamsburg, VA 23187, United States

^d Lawrence Berkeley National Lab, Berkeley, CA 94720, United States

ARTICLE INFO

Article history:

Received 31 August 2006

Received in revised form

31 December 2008

Accepted 16 March 2009

Available online 31 March 2009

Keywords:

Parallel

Mesh

Generation

SMT

Optimizations

Finite element

ABSTRACT

This article focuses on the optimization of PCDM, a parallel, two-dimensional (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.

Published by Elsevier Inc.

1. Introduction

Simultaneous multithreading (SMT) and multicore (CMP) processors have lately found their way in the product lines of all major hardware manufacturers [29–31]. These processors allow more than one thread 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 are gaining more

and more popularity as building blocks of both multilayer, high-performance compute servers and off-the-shelf desktop systems.

The pervasiveness of SMT and CMP processors radically changes 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, is tending 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 the functionality offered by system libraries (synchronization primitives, STL data structures, memory management,

* Corresponding address: Department of Computer Science, The College of William and Mary, Williamsburg, PO Box 8795, VA 23187, United States.

E-mail addresses: cda@inf.uth.gr (C.D. Antonopoulos), fblagojevic@lbl.gov (F. Blagojevic), ancher@cs.wm.edu (A.N. Chernikov), nikos@cs.wm.edu (N.P. Chrisochoides), dsn@cs.vt.edu (D.S. Nikolopoulos).

URLs: <http://inf-server.inf.uth.gr/~cda> (C.D. Antonopoulos), <http://www.cs.vt.edu/~filip> (F. Blagojevic), <http://www.cs.wm.edu/~ancher> (A.N. Chernikov), <http://www.cs.wm.edu/~nikos> (N.P. Chrisochoides), <http://www.cs.vt.edu/~dsn> (D.S. Nikolopoulos).

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 two-dimensional (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, multilevel and multigrain 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 [14].

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 multilevel and multigrain parallelization for layered multiprocessors, where threads executing and sharing resources on the same processor are differentiated from 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 [40], to our knowledge the best 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 multigrain 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 [26] and shared-memory parallel benchmarks such as SPEC OMP [6] and SPLASH-2 [47]. 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 multicore and SMT processor cores, multicore 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 [5], 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 [18,48], and have yielded results that are stirring the database community to develop more architecture-aware DataBase Management System (DBMS) infrastructure [25]. Another recent study of several realistic applications, including molecular dynamics and material science codes, on a Power5-based system with dual SMT-core processors [24], 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 [15,22,32,38,41]. 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 [33] 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 one 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 [12]. For our analysis and implementation we use the Bowyer–Watson (B–W) point insertion procedure [8,46], which consists of the following 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 Ω is decomposed into subdomains Ω_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 [14]. At the medium granularity level, the units of refinement are cavities;

Table 1

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

Processor	4, 2-way Hyperthreading, Pentium 4 Xeon, 2 GHz
Cache	8 KB L1, 64B line/512 KB L2, 64B line/1 MB L3, 64B line
Memory	2 GB RAM
OS	Linux 2.6.13.4
Compiler	g++, gcc 3.3.4

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 [10–13]. In this paper, however, we study a different approach which allows us 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 multigrain implementation we describe in [4]. 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 multigrain parallelization strategies appear in [4]. 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 the performance monitoring counters [27]. The performance counters offer valuable information on the interaction between software and the underlying hardware. They can be used either directly [37], or through higher level data acquisition and analysis tools [1,9,19].

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 Section 4.1 we discuss and evaluate the optimizations related to the coarse-grain parallelization level of PCDM. In Section 4.2 we focus on the optimization process of the medium-grain parallelization level of PCDM. Section 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 Section 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 subdomains, 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 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 Fig. 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 percentage 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 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.

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.

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

Table 2

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

	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

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 (right diagram).

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 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 (4 Kb 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 multithreaded memory allocators also manage memory pools internally for objects of the same size [7,21,23,39], 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

