Computational Geometry 43 (2010) 663–677

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

Parallel geometric algorithms for multi-core computers ☆

Vicente H.F. Batista [a], David L. Millman [b], Sylvain Pion [c,*], Johannes Singler [d]

[a] Departamento de Engenharia Civil, Universidade Federal do Rio de Janeiro, Caixa Postal 68506, Rio de Janeiro, RJ, 21945-970, Brazil
[b] Department of Computer Science, University of North Carolina, CB 3175 Sitterson Hall, Chapel Hill, NC 27599-3175, USA
[c] Institut National de Recherche en Informatique et en Automatique, 2004 Route des Lucioles, BP 93, 06902 Sophia Antipolis, France
[d] Fakultät für Informatik, Karlsruhe Institute of Technology, Postfach 6980, 76128 Karlsruhe, Germany

ARTICLE INFO

Article history:
Received 7 July 2009
Received in revised form 3 January 2010
Accepted 28 April 2010
Available online 4 May 2010
Communicated by A. Zomorodian

Keywords:
Parallel algorithms
Delaunay triangulations
Box intersection
Spatial sort
Compact container

ABSTRACT

Computers with multiple processor cores using shared memory are now ubiquitous. In this paper, we present several parallel geometric algorithms that specifically target this environment, with the goal of exploiting the additional computing power. The algorithms we describe are (a) 2-/3-dimensional spatial sorting of points, as is typically used for preprocessing before using incremental algorithms, (b) *d*-dimensional axis-aligned box intersection computation, and finally (c) 3D bulk insertion of points into Delaunay triangulations, which can be used for mesh generation algorithms, or simply for constructing 3D Delaunay triangulations. For the latter, we introduce as a foundational element the design of a container data structure that both provides concurrent addition and removal operations and is compact in memory. This makes it especially well-suited for storing large dynamic graphs such as Delaunay triangulations.

We show experimental results for these algorithms, using our implementations based on the Computational Geometry Algorithms Library (CGAL). This work is a step towards what we hope will become a *parallel mode* for CGAL, where algorithms automatically use the available parallel resources without requiring significant user intervention.

1. Introduction

It is generally acknowledged that the microprocessor industry has reached the limits of the sequential performance of processors. Processor manufacturers now focus on parallelism to keep up with the demand for high performance. Current laptop computers all have 2 or 4 cores, and desktop computers can easily have 4 or 8 cores, with many more in high-end computers. This trend incites application writers to develop parallel versions of their critical algorithms. This is not an easy task, from both the theoretical and practical points of view.

Work on theoretical parallel algorithms began decades ago, even parallel geometric algorithms have received attention in the literature. In the earliest work, Chow [1] addressed problems such as intersections of rectangles, convex hulls and Voronoi diagrams. Since then, researchers have studied theoretical parallel solutions in the PRAM model, many of which are impractical or inefficient in practice. This model assumes an unlimited number of processors, whereas in this paper, we assume that the amount of available processors is significantly less than the input size. Both Aggarwal et al. [2] and Akl and Lyons [3] are excellent sources of theoretical parallel *modus operandi* for many fundamental computational geometry prob-

☆ This work has been supported by the INRIA Associated Team GENEPI, the NSF-INRIA program REUSSI, the Brazilian CNPq sandwich PhD program, the French ANR program TRIANGLES, and DFG grant SA 933/3-1.

* Corresponding author.
E-mail addresses: helano@coc.ufrj.br (V.H.F. Batista), dave@cs.unc.edu (D.L. Millman), Sylvain.Pion@sophia.inria.fr (S. Pion), singler@kit.edu (J. Singler).

0925-7721/$ – see front matter
doi:10.1016/j.comgeo.2010.04.008

lems. The relevance of these algorithms in practice depends not only on their implementability, but also on the hardware architecture targeted.

Programming tools and languages are evolving to better support parallel computing. Between hardware and applications, there are several layers of software. The bottom layer, e.g., OpenMP, contains primitives for thread management and synchronization, which builds on OS capabilities and hardware-supported instructions. On top of that, parallel algorithms can be implemented in domain specific libraries, as we show in this paper for the Computational Geometry Algorithms Library (CGAL) [4], which is a large collection of geometric data structures and algorithms. Finally, applications can use the implicit parallelism encapsulated in such a library, without necessarily doing explicit parallel programming on this level.

In this paper, we focus on shared-memory parallel computers, specifically multi-core CPUs, which allow simultaneous execution of multiple instructions on different cores. This explicitly excludes distributed memory systems as well as graphical processing units, which have local memory for each processor core and thus require special code to communicate. As we are interested in practical parallel algorithms, it is important to base our work on efficient sequential code. Otherwise, there is a risk of good relative speedups that lack practical interest and skew conclusions about the algorithms scalability. For this reason, we decided to base our work upon CGAL, which already provides mature codes that are among the most efficient for several geometric algorithms [5]. We investigate the following algorithms: (a) 2-/3-dimensional spatial sorting of points, as is typically used for preprocessing before using incremental algorithms, (b) d-dimensional axis-aligned box intersection computation, and finally (c) 3D bulk insertion of points in Delaunay triangulations, which can be used for mesh generation algorithms, or simply for constructing 3D Delaunay triangulations.

The remainder of the paper is organized as follows: Section 2 describes our hardware and software platform; Section 3 contains the description of the thread-safe compact container used by the Delaunay triangulation; Sections 4, 5 and 6 describe our parallel algorithms, the related work and the experimental results for (a), (b) and (c) respectively; we conclude and present future plans in Section 7.

2. Platform

2.1. OpenMP

For thread control, several frameworks of relatively high level exist, such as TBB [6] or OpenMP [7]. We decided to rely on the latter, which is implemented by almost all modern compilers. As a new feature, the OpenMP specification in version 3.0 includes the `#pragma omp task` construct. This creates a *task*, i.e., a code block that is executed asynchronously. Creating such tasks can be nested recursively. The enclosing region may wait for all direct children tasks to finish using `#pragma omp taskwait`. A `#pragma omp parallel` region at the top level provides a user specified number of threads to process the tasks. When a new task is spawned, the runtime system can decide to run it with the current thread at once, or postpone it for processing by an arbitrary thread. If the task model is not fully appropriate, the program can also just run a certain number of threads and make them process the problem.

2.2. Libstdc++ parallel mode

The C++ STL implementation distributed with the GCC features a so-called *parallel mode* [8] as of version 4.3, based on the Multi-Core Standard Template Library [9]. It provides parallel versions of many STL algorithms. We use some of these algorithmic building blocks, such as `partition`, `nth_element` and `random_shuffle`. The `partition` algorithm partitions a sequence with respect to a given pivot as in quicksort. Applying `nth_element` to a sequence places the element with a given rank k at index k, and moves the smaller ones to the left, the larger ones to the right. The `random_shuffle` routine is used to permute a sequence randomly.

2.3. Evaluation system

We evaluated the performance of our algorithms on an up-to-date machine, featuring two AMD Opteron 2350 quad-core 64-bit processors at 2 GHz and 16 GB of RAM. We used GCC 4.4 (for the algorithms using the task construct), enabling optimization (`-O3` and `-DNDEBUG`). If not stated otherwise, each test was run at least 10 times, and the average over all running times was taken.

2.4. CGAL kernels

Algorithms in CGAL are parameterized by so-called kernels, which provide the type of points and accompanying geometric predicates. In each case, we have chosen the kernel that is most efficient while providing appropriate robustness guarantees: `Exact_predicates_inexact_constructions_kernel` for Delaunay triangulation, and `Simple_cartesian<double>` for the other algorithms, since they perform only coordinate comparisons.