Speculative Parallelization of a Randomized Incremental Convex Hull Algorithm*

Marcelo Cintra[1], Diego R. Llanos[2], and Belén Palop[2]

[1] School of Informatics, University of Edinburgh, Edinburgh, UK,
mc@inf.ed.ac.uk

[2] Departamento de Informática, Universidad de Valladolid, Valladolid, Spain,
{diego|bpalop}@infor.uva.es

Abstract. Finding the fastest algorithm to solve a problem is one of the main issues in Computational Geometry. Focusing only on worst case analysis or asymptotic computations leads to the development of complex data structures or hard to implement algorithms. Randomized algorithms appear in this scenario as a very useful tool in order to obtain easier implementations within a good expected time bound. However, parallel implementations of these algorithms are hard to develop and require an in-depth understanding of the language, the compiler and the underlying parallel computer architecture. In this paper we show how we can use speculative parallelization techniques to execute in parallel iterative algorithms such as randomized incremental constructions. In this paper we focus on the convex hull problem, and show that, using our speculative parallelization engine, the sequential algorithm can be automatically executed in parallel, obtaining speedups with as little as four processors, and reaching $5.15x$ speedup with 28 processors.

1 Introduction

Finding the fastest algorithm to solve a problem is one of the main issues in Computational Geometry. Focusing only on worst case analysis or asymptotic computations leads to the development of complex data structures or hard to implement algorithms. Randomized algorithms appear in this scenario as a very useful tool in order to obtain easier implementations, taking advantage of the remarkable fact that, if we study how the complexity of the algorithm is related with the ordering in which points are processed, only a tiny percentage of the orderings leads to worst case situations.

While sequential implementations of these algorithms lead to good results in terms of complexity, obtaining a parallel version is not straightforward. Sometimes the development of a sequential implementation can be accomplished without much effort, but a parallel implementation of a given incremental algorithm

* The first author has been partially supported by EPSRC under grant GR/R65169/01. The first and second authors have been partially supported by the European Commission under grant HPRI-CT-1999-00026. The third author has been partially supported by MCYT TIC2003-08933-C02-01.

A. Laganà et al. (Eds.): ICCSA 2004, LNCS 3045, pp. 188–197, 2004.
© Springer-Verlag Berlin Heidelberg 2004

is hard to develop, requiring an in-depth understanding of the programming language, the compiler, the parallel tool set and the underlying parallel computing architecture.

In this paper we show how we can use speculative parallelization techniques to automatically parallelize sequential, incremental algorithms with a small number of dependences between iterations, such as randomized incremental constructions. In this paper we focus on the convex hull problem. Using our speculative engine, the algorithm can be automatically parallelized, with only a tiny fraction of the effort needed to design, analyze and program a parallel version (see e.g., [8]).

After analyzing the effect of input set sizes and different shapes of the data distribution, our results show that the speculative version of the sequential algorithm leads to speedups with as little as four processors, reaching a maximum of $5.15x$ for 28 processors.

The rest of the paper is organized as follows. Section 2 describes the randomized planar convex hull problem. Section 3 introduces speculative parallelization and shows how it may be used to easily obtain a parallel version of an iterative algorithm. Section 4 describes the parallel execution environment and discusses the experimental results, while section 5 concludes the paper.

2 Randomized Planar Convex Hull

Given a set S of n points in the plane, the *convex hull* of S, $CH(S)$, is the smallest convex region containing all points in S. We will use $CH(S)$ for the ordered sequence of vertices of the convex region, which are known to be points of S.

Since 1972 when Graham [9] gave the first algorithm to compute the convex hull of a set of points in $O(n \log n)$ time and $O(n)$ space, a lot of effort has been done to find algorithms reaching better lower bounds in time and space. In 1986 Kirkpatrick and Seidel [11] proved the time lower bound of $\Omega(n \log h)$, where h is the number of points in $CH(S)$, and gave the first algorithm within this time bound. In 1996 Chan [3] gave simpler algorithms for the computation of $CH(S)$ in two and three dimensions. With respect to space complexity, Brönnimann *et al.* [1] showed in 2002 that it is possible to implement Chan's optimal time algorithm with only $O(1)$ additional space to the set of points.

In parallel with the *lower-bound race*, randomized constructions appear to obtain simpler algorithms which are *expected* to run within good time bounds when the input points follow some uniform distribution and/or are processed in random order [6,14].

Many geometric algorithms and structures are based on convex hulls. Therefore, it is not surprising that so much effort has been done in order to lower its computational complexity. In this work we introduce a new technique that, with very little effort from the implementation's point of view, allows many iterative algorithms to be run in parallel. We will concentrate on an incremental randomized construction because of two main reasons: The first is that incremental constructions are usually easy to implement and show very good expected running times. The second reason is that, in an incremental process, many iterations

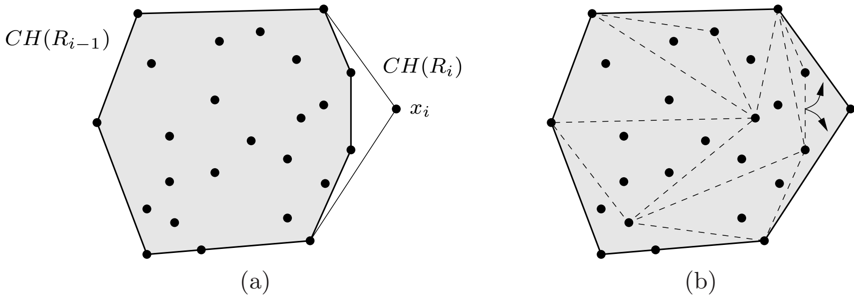

Fig. 1. Clarkson *et al.* algorithm: (a) adding a new point to the convex hull; (b) growth of the convex hull (auxiliary structure shown in dashed lines).

do not change the structure already computed and dependences between processors are relatively rare or, at least, bounded by the number of changes in the structure along the execution.

2.1 Clarkson et al. Algorithm

One of the most efficient and easy to implement randomized incremental algorithms for the construction of the 2-dimensional convex hull, which can be easily extended to higher dimensions, is due to Clarkson, Mehlhorn and Seidel [7]. A brief description of the algorithm follows. More details can be found in [13].

Let S be a set of n points in the plane, let $x_1, x_2, \ldots, x_n$ be a random permutation of the points in S, and call R_i the random subset $\{x_1, x_2, \ldots, x_i\}$. Suppose $CH(R_{i-1})$ is already computed and we want to compute $CH(R_i)$. Point x_i can be inside or outside $CH(R_{i-1})$. If it is inside, obviously, $CH(R_i) = CH(R_{i-1})$. Otherwise, x_i is on the boundary of $CH(R_i)$. All edges in $CH(R_{i-1})$ between the two tangents from x_i to $CH(R_{i-1})$ should be deleted and these two tangents should be added into $CH(R_i)$. See Figure 1(a).

The main idea on Clarkson's *et al.* algorithm is to keep an auxiliary structure that helps finding, in expected $O(\log n)$ time, some edge between the two tangents visible from the new point x_i (see Figure 1(a)) and keeps track of all edges created during the construction of the hull $CH(R_{i-1})$. For each edge in $CH(R_{i-1})$, two pointers are kept for the previous and next edges in the hull. But when an edge should be deleted, these pointers indicate the two new edges in $CH(R_i)$ that caused its deletion. See Figure 1(b).

On each iteration, the algorithm follows the path from the first constructed triangle to the point being inserted and outputs, if it is outside, one edge that is visible from the point. This way, the cost of performing a sequential search for the tangents will be amortized, since all visited edges will be deleted and we can reach the expected $O(n \log n)$ time bound [7].

Think now on a computer with several processors and suppose that we assign each processor one iteration of the algorithm. We expect that only $O(\log n)$ iterations will produce changes in the convex hull. This means that most of

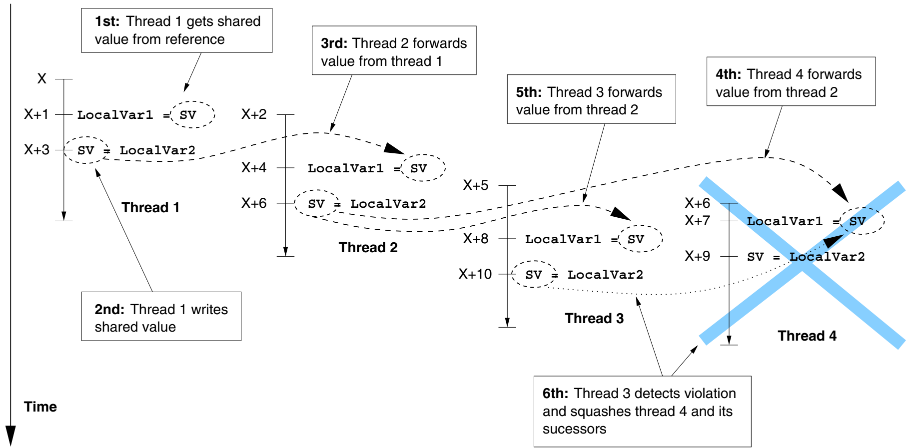

Fig. 2. Speculative parallelization.

the iterations are independent in the sense that they can be run at the same time using the same computed structure. In the next section we introduce a technique called *speculative parallelization*, explaining how this technique can help to speed up the execution of many randomized incremental algorithms sharing this property.

3 Speculative Parallelization

The basic idea under speculative parallelization (also called *thread-level speculation*) [4,12,15] is to assign the execution of different *blocks* of consecutive iterations to different threads, running each one on its own processor. While execution proceeds, a software monitor ensures that no thread consumes an incorrect version of a value that should be calculated by a predecessor, therefore violating sequential semantics. If such a *dependence violation* occur, the monitor stops the parallel execution of the offending threads, discards iterations incorrectly calculated, and restart their execution using the correct values. See Figure 2.

The detection of dependence violations can be done either by hardware or software. Hardware solutions (see e.g., [5,10,16]) rely on additional hardware modules to detect dependences, while software methods [4,12,15] augment the original loop with new instructions that check for violations during the parallel execution. We have presented in [4] a new software-only speculative parallelization engine to automatically execute in parallel sequential loops with few or no dependences among iterations. The main advantage of this solution is that it makes possible to parallelize an iterative application automatically by a compiler, thus obtaining speedups in a parallel machine without the cost of a manual parallelization. To do so, the compiler augments the original code with function calls to perform accesses to the structure shared among threads, and to monitor the parallel execution of the loop.

3.1 Types of Data Dependences

From the parallel execution point of view, in each iteration two different classes of variables can appear. Informally speaking, *private* variables will be those that are always written in each iteration before being used. On the other hand, values stored in *shared* variables are used among different iterations. It is easy to see that if all variables are private, then no dependences can arise and the loop can be executed in parallel. Shared variables may lead to dependence violations only if a value is written in a given iteration and a successor has consumed an outdated value. This is known as the Read-after-Write (RAW) dependence. In this case, the latter iteration and all its successors should be re-executed using the correct values. This is known as a *squash* operation.

To simplify squashes, threads that execute each iteration do not change directly the shared structure: instead, each thread maintains a *version* of the structure. Only if the execution of the iteration succeeds, changes are reflected to the original shared structure, through a *commit* operation. This operation should be done in order for each block of iterations, from the non-speculative thread (that is, the one executing the earliest block) to the most-speculative one. If the execution of the iteration fails, version data is discarded. The next section discusses these operations in more detail.

3.2 Augmenting the Convex-Hull Algorithm for Speculative Execution

Clarkson *et al.* algorithm shown in section 2.1 relies on a structure that holds the edges composing the current convex hull. Whenever a new point is added, the point is checked against the current solution. It is easy to see that this structure should be shared among different iterations. If the point is inside the hull, the current solution is not modified. Otherwise, the new convex hull should be calculated to contain the new edges defined by the point. From the speculative execution point of view, each time a new point modifies the convex hull the parallel execution of subsequent iterations should be restarted, thus degrading performance. Fortunately, as execution proceeds new points are less likely to modify the current solution, and large blocks of iterations can be calculated in parallel without leading to dependence violations. This is why speculative parallelization is a valid technique to speed up the execution of this kind of algorithms.

To compare the performance of the speculative version against the sequential algorithm, we have implemented a Fortran version of Clarkson *et al.* algorithm, augmenting the sequential code manually for speculative parallelization. This task can be performed automatically by a state-of-the art compiler. A complete and detailed description of these operations can be found in [4]. A summary of the changes made in the sequential code follows.

Thread scheduling. For each loop, blocks of consecutive iterations are distributed among different threads.

Speculative loads and stores. As long as each thread maintains its own version copy of the shared structure, all original reads and writes to this structure should be augmented with a procedure call that performs the operation required and checks for possible violations. For example, a read of the shared structure such as `psource = hull(e,source)` should be replaced with the following code:

```
! Calculate linear position of element in shared structure
position = e + NumEdges * (source-1)
! Perform load operation, returning value in "psource"
call specload(position,MyThreadID,psource,hull)
```

Thread commit. After executing a block of iterations, each thread calls a function that checks its state and performs the commit when appropriate.

After augmenting the code for speculative parallelization, we compared its performance with the sequential version under different configurations and with several input sets. Results are shown in the next section.

4 Experimental Results

The experiments performed to measure the execution time of both sequential and parallel versions of the algorithm were done on a Sun Fire 15K symmetric multiprocessor (SMP), equipped with 900MHz UltraSparc-III processors, each with a private 64 KByte 4-way set-associative L1 cache, a private 8 MByte direct-mapped L2 cache, and 1 GByte of shared memory per processor. The system runs SunOS 5.8. The application was compiled with the Forte Developer 7 Fortran 95 compiler using the highest optimization settings for our execution environment: `-O3 -xchip=ultra3 -xarch=v8plusb -cache=64/32/4:8192/64/1`

Times shown in the following sections represent the time spent in the execution of the processing loop of the application. The time needed to read the input set and the time needed to output the convex hull have not been taken into account. The application had exclusive use of the processors during the entire execution and we use wall-clock time in our time measurements.

4.1 Design of the Input: Shape and Size

The number of violations between executions is bounded by the number of points lying outside the convex hull computed up to their insertion. Depending on how quickly the growing convex hull tends to the final one, the number of dependences changes. We have thus designed four different input sets: The first two are sets of 10 and 40 million random points in a square, where we expect violations to lower rather quickly after some iterations; The two others are sets of 10 and 40 million random points in a disk, where the final convex hull is expected to have size $O(\log n)$ and violations will happen more often. We will not analyze degenerate cases like a set of points on a circle, since every iteration is dependent

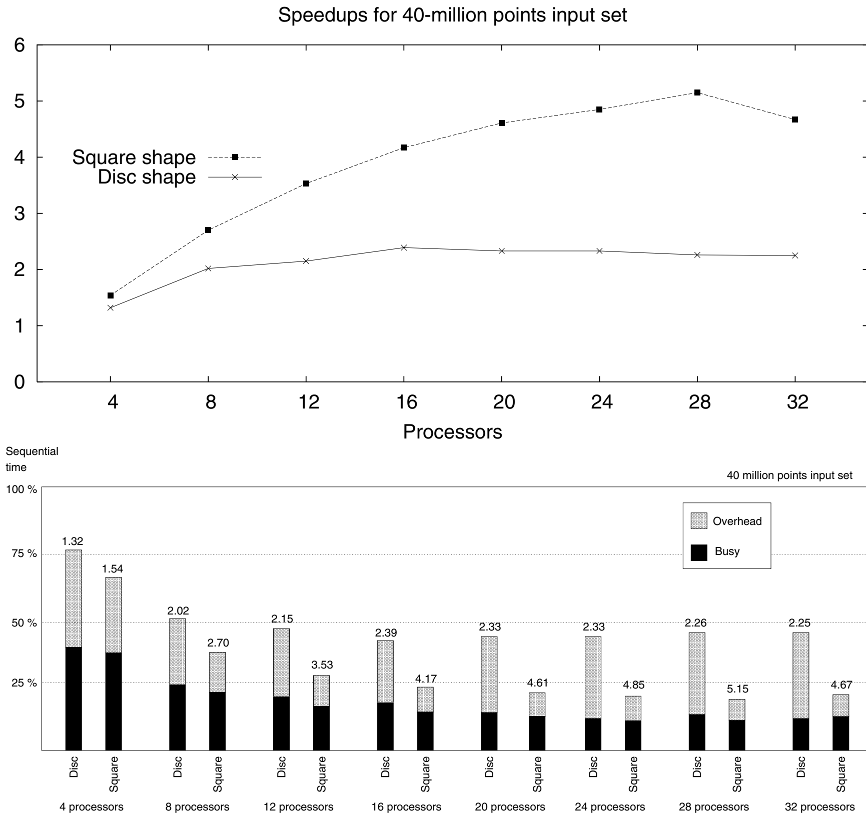

Fig. 3. Speedups and execution breakdown for 40 million points problem size.

on the previous ones and the problem is inherently non-parallel. Smaller input sets were not considered, since their sequential execution time took less than ten seconds in the system under test.

The sets of points have been generated using the random points generator in CGAL 2.4 [2] and have been randomly ordered using its *shuffle* function.

4.2 Overall Speedups

Figure 3 shows the effect of executing the parallel code with the 40 million points problem size for square and disc input sets. Results are normalized with respect to the corresponding sequential execution time. Results are shown for 4 to 32 processors. Execution time breakdowns are divided into "overhead" time (spent in different operations such as synchronization, commit, and loads/stores) and "busy" time that reflects the original loop calculations. Figure 4 shows the effect of executing the parallel code with the 10 million points problem size for the square and disc input sets.

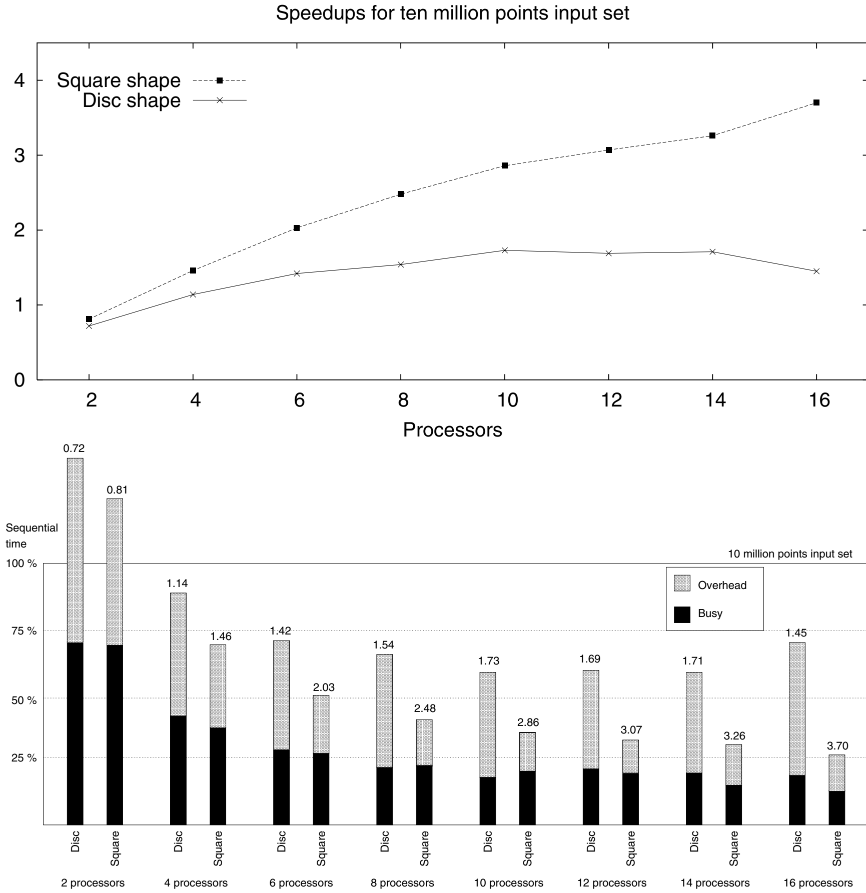

Fig. 4. Speedups and execution breakdown for 10 million points problem size.

From our experiments we can draw the following observations:

- The higher the input size, the higher the speedups obtained, because more and more points can be processed in parallel without modifying the current convex hull.
- The system scales well, allowing better speedups when adding more processors. As can be seen in figure 3, our experiments show a maximum speedup of $5.15x$ with 28 processors for the square input set.
- A significant part of the time is spent in the original calculations. Our experiments shows that the main source of overhead are accesses to the shared structure, in particular load operations.
- As expected, speedups are poorer for the disc input sets, since they have a richer set of edges in the solution, and more memory operations are needed

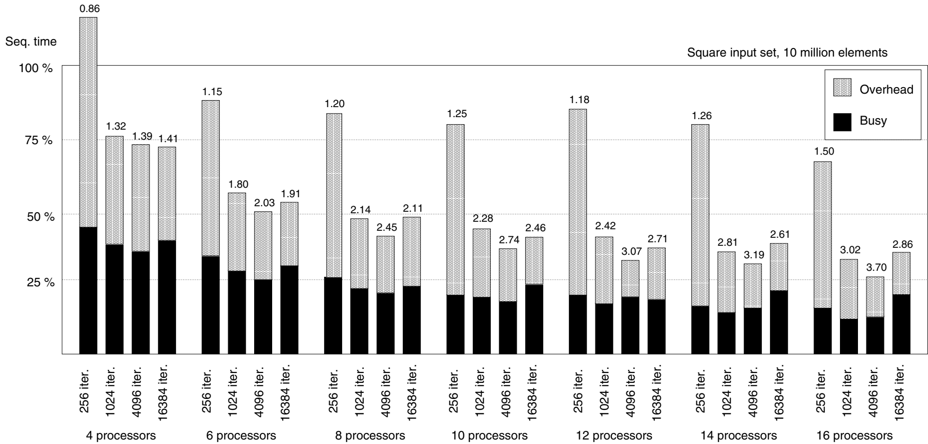

Fig. 5. Execution breakdowns for different block sizes, with a window size equal to the number of processors [4].

to determine whether a given point is inside the current solution. However, we already obtain speedups with as little as four processors.

– Choosing a higher block size does not necessarily lead to better speedups. Optimum block size is a trade-off between having few blocks to execute and having few threads to squash, and also depends on the size of the input set and its shape. Figure 5 shows speedups for different block sizes for one of our input sets. In general, values between 1K and 4K iterations lead to acceptable results for all input sets considered in this work.

5 Conclusions

Parallel implementations of incremental algorithms are hard to develop and require an in-depth understanding of the problem, the language, the compiler and the underlying computer architecture. In this paper we have shown how we can use speculative parallelization techniques to execute automatically in parallel the randomized incremental convex hull algorithm. Choosing an adequate block size, good speedups can be obtained for different workloads with a negligible implementation cost.

Acknowledgments. We would like to thank Pedro Ramos for his helpful comments concerning randomized algorithms.

References

1. H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. In-place planar convex hull algorithms. In *Proc. of the 5th Latin American Symp. on Theor. Informatics (LATIN'02)*, pages 494–507, April 2002.
2. CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org/`.
3. T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16:361–368, 1996.
4. M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proc. of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–24, June 2003.
5. M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proc. of the 27th Intl. Symp. on Computer Architecture (ISCA)*, pages 256–264, June 2000.
6. K. L. Clarkson. Randomized geometric algorithms. In Ding-Zhu Du and Frank Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lect. Notes Series on Computing*, pages 149–194. World Scientific, 2nd edition, 1995.
7. K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, 1993.
8. M. Ghouse and M. Goodrich. Fast randomized parallel methods for planar convex hull construction. *Comput. Geom. Theory Appl.*, 7:219–236, 1997.
9. R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.
10. L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, October 1998.
11. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.
12. M. Gupta and R. Nim. Techniques for run-time parallelization of loops. *Supercomputing*, November 1998.
13. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 2000.
14. K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.
15. L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.
16. G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Intl. Symp. on Computer Architecture (ISCA)*, pages 414–425, June 1995.