Available online at www.sciencedirect.com
SCIENCE DIRECT®
Applied Mathematical Modelling 30 (2006) 563–577
APPLIED MATHEMATICAL MODELLING
www.elsevier.com/locate/apm

Parallelization alternatives and their performance for the convex hull problem

Arturo González-Escribano [a], Diego R. Llanos [a,*,1], David Orden [b,2], Belén Palop [a,3]

[a] *Departamento de Informática, Universidad de Valladolid, Valladolid, Spain*
[b] *Departamento de Matemáticas, Universidad de Alcalá, Spain*

Received 30 March 2005; received in revised form 12 May 2005; accepted 16 May 2005
Available online 20 December 2005

Abstract

High performance machines have become available nowadays to an increasing number of researchers. Most of us might have both an access to a supercomputing center and an algorithm that could benefit from these high performance machines. The aim of the present work is to revisit all existing parallelization alternatives, including emerging technologies like software-only speculative parallelization, to solve on different architectures the same representative problem: The computation of the convex hull of a point set.

Keywords: Automatic parallelization; Manual parallelization; Speculative parallelization; Convex hull; Incremental randomized algorithms

1. Introduction

Parallelizing an algorithm is not a simple task. Although the idea of executing in parallel a sequential algorithm is always attractive, choosing the best parallel alternative is not easy and depends on the architecture of the machine we have access to. Each architecture has its advantages and drawbacks and it is not easy to develop parallel implementations that make good use of the former while mitigating the effect of the latter.

In this work we revisit manual, speculative and automatic implementation of algorithms for both shared- and distributed-memory machines. In order to be able to compare the obtained results, we have chosen a representative problem: the convex hull of a point set. Apart from being of use in many applications and fields,

* Corresponding author. Tel.: +34 983 185 642; fax: +34 983 423 671.
E-mail addresses: arturo@infor.uva.es (A. González-Escribano), diego@infor.uva.es (D.R. Llanos), david.orden@uah.es (D. Orden), b.palop@infor.uva.es (B. Palop).
[1] Partially supported by RII3-CT-2003-506079.
[2] Partially supported by MCYT BFM2001-1153.
[3] Partially supported by MCYT TIC2003-08933-C02-01.

0307-904X/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.apm.2005.05.022

the convex hull problem has revealed itself as a very versatile benchmark, since the shape of the input set determines the expected output size.

Thanks to the use of the software-only speculative engine developed by Cintra and Llanos [1], we were able to compare manual, speculative and automatic parallelization on a shared-memory system. As we will see, speculative parallelization allows to obtain acceptable speedups in the parallel execution of randomized incremental algorithms in comparison with manual procedures, with only a fraction of the software development cost. We have also evaluated the same algorithms on a distributed-memory system, making it possible to compare their scalability on both architectures.

The paper is organized as follows. Section 2 describes the available parallelization schemes: Manual, automatic and speculative, and their relationship with the shared- and distributed-memory underlying architectures. Section 3 introduces the benchmarking problem we will use: the convex hull of a point set in 2D. Different algorithms are described and their time complexity analysis is given. In Section 4 we deal with the manual parallelization of the described convex hull algorithms. Both design and implementation issues are discussed. The automatic parallelization of convex hull algorithms is treated in Section 5, using both compiler-based and speculative approaches. Section 6 shows and compares the experimental results obtained for the different parallelization alternatives. The conclusions are contained in Section 7.

2. Parallel programming

In order to understand the trade-offs inherent to parallel programming, in this section we will briefly review the most common parallel architectures, together with the programming models available for them.

A parallel computer is a multiple-processor computer system supporting parallel programming. Two important categories of parallel computers are multicomputers and multiprocessors. A *multicomputer* is a parallel machine constructed out of multiple computers and an interconnection network. The processors on different computers interact by passing messages to each other. This architecture is also called *cluster of computers* or *distributed-memory systems* [2]. Clusters of computers can be built using commodity computers, but to achieve a good efficiency a high-speed network should be used. In contrast, a *multiprocessor* is a more highly-integrated system in which all CPUs share the access to a single global memory. The shared memory gives communication and synchronization support among processors. This architecture is also called *symmetrical multiprocessors*, *SMP*, or *shared-memory systems* [2].

Different architectures impose different programming models. Some of them are focused on efficiently exploiting the underlying architecture, sacrificing ease-of-programming and reusability, while others are more oriented to a systematic parallel software development. In the following subsections we will discuss different strategies to implement parallel programs.

2.1. Manual parallelization

Traditionally, parallel programs have been manually written using low-level communication and synchronization mechanisms. This allows expert programmers to manually optimize parallel programs for a given application and machine, achieving maximum performance. However, in order to write a parallel program the programmer should know in detail the characteristics of the architecture where the program will run and take low-level decisions about data-decomposition, scheduling, mapping and synchronization procedures. The most popular parallel tools used for implementation can be classified into two categories: message-passing library routines and multithreaded programming. The next paragraphs discuss these alternatives in more detail.

2.1.1. Message-passing library routines

As we mentioned above, a program that will run in a cluster of computers is composed of several processes, one for each processor. The common programming model used in these machines is based on the theory of communicating sequential processes using explicit messages [3]. The programming model based on this theory is called *message-passing*. There exist standard and widely accepted message-passing interfaces, such as MPI [4,5] or PVM [6]. They are typically implemented as portable libraries and they achieve a certain level of

abstraction over the communication mechanisms provided by the machine and the operating system.[4] However, to achieve good performance the programmer should minimize the number of messages among processes, avoiding bottlenecks, stall conditions and idle times, and achieve good load balancing [7].

2.1.2. Multithreaded programming

Programming in a shared-memory system does not need the use of explicit messages to communicate processors. Instead, each processor may read and write on shared-memory locations accessible by all threads. The most common programming model is called *multithreaded programming*. Standard libraries, such as POSIX threads [8], allow a certain degree of abstraction from the underlying architecture. However, the programmer still needs to explicitly spawn threads, schedule tasks among processors and prevent dead-locks or race conditions.

Multithreaded programming has a similar complexity as message-passing programming. Programming in both ways has been shown to be difficult and error-prone. One of the main reasons is that applications are composed by independent processes which evolve without a global state (message-passing), or by threads which may interact in a different way if global breakpoints are introduced (as in the case of multithreading). Debugging such applications is more complicated than debugging sequential ones by at least one degree of freedom [9,10].

2.1.3. New trends and emerging technologies

Given the shortcomings of the programming models described above, other languages and models have been proposed to introduce a higher level of abstraction. The most promising ones are those that restrict the synchronization structures available to the programmer to increase the program analyzability, allowing the compiler to use simple cost models to drive mapping optimizations [11].

A relevant class of these restricted and structured programming models is known as *nested-parallel programming models* such as OpenMP [12,13], nested forms of BSP [14,15] or Cilk [16]. These models provide simple *cobegin-coend*-like constructs that in some cases can be nested. Communication and synchronization only occur at the beginning or end of the construct. For example, divide and conquer and other recursive-decomposition design strategies map naturally on nested parallelism. This simple composite nature is the base for more extensible parallel programming languages, that may include libraries of well-known specialized parallel structures typically known as *parallel skeletons* [17]. Flexible compiler frameworks that unify nested-parallelism compiling techniques have also been proposed by González-Escribano et al. [18].

Compilers for such structured models only support a limited range of parallel structures (OpenMP) or are still experimental (NestedBSP). Most of them are not yet mature to deal with highly irregular applications that can still be efficiently programmed with manual, unstructured mechanisms. For example, there are few implementations of OpenMP compilers which support nested parallelism, and some of them may present serious performance limitations [19,20]. Although there is an active research field focusing on structured parallelism, the research community has not yet agree on an architecture-independent, widely accepted structured parallel programming model or language.

2.2. Compiled-based automatic parallelization

Compiled-based parallelization moves to the compiler the complexity of generating a parallel version of a sequential program. Parallelizing compilers are available for many shared memory architectures. Most of them are exclusively focused on *loop-level* parallelization: they typically attempt to parallelize only `DO` loops in Fortran and `for` loops in C, with integer indexes and iteration counters known at runtime [21]. An initial *data-dependence analysis* determines if the candidate loop is composed by iterations that do not depend on the results calculated in previous ones of the same loop. Only loops that meet these criteria can be parallelized

[4] In fact, message-passing interfaces are also implemented on shared-memory architectures using specific and faster synchronization mechanisms.

safely [22]. Besides this, many compilers avoid to parallelize a loop when they estimate that the overhead produced by the execution of a parallel section may overcome any performance gain.

Sequential code parallelization may appear to be a shortcut. However, detecting data dependences in generic sequential code at compilation time is not always possible. Moreover, different sequential algorithms or solutions for the same application may present different degrees of potential parallelism. Some of them may even be inherently sequential and not parallelizable at all. Nevertheless, this approach is a major trend on parallel programming. The sequential programmers' expertise, accumulated along decades, and the huge legacy of sequential code are important reasons to support this approach.

2.3. Speculative parallelization

The most promising technique to automatically parallelize loops when dependences cannot be determined at compile time is called speculative parallelization. This technique, also called *thread-level speculation* [1,23,24], assigns 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* occurs, the monitor stops the parallel execution of the offending threads, discards iterations incorrectly calculated and restarts their execution using the correct values.

The detection of dependence violations can be done either by hardware or software. Hardware solutions (see e.g. [25–27]) rely on additional hardware modules to detect dependences, while software methods [1,23,24] augment the original loop with new instructions that check for violations during the parallel execution. We have presented in [1] 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 development cost of a manual parallelization. To do so, the compiler augments the original code with function calls to access the shared data structure and to monitor the parallel execution of the loop.

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 dependence. In this case, the latter iteration

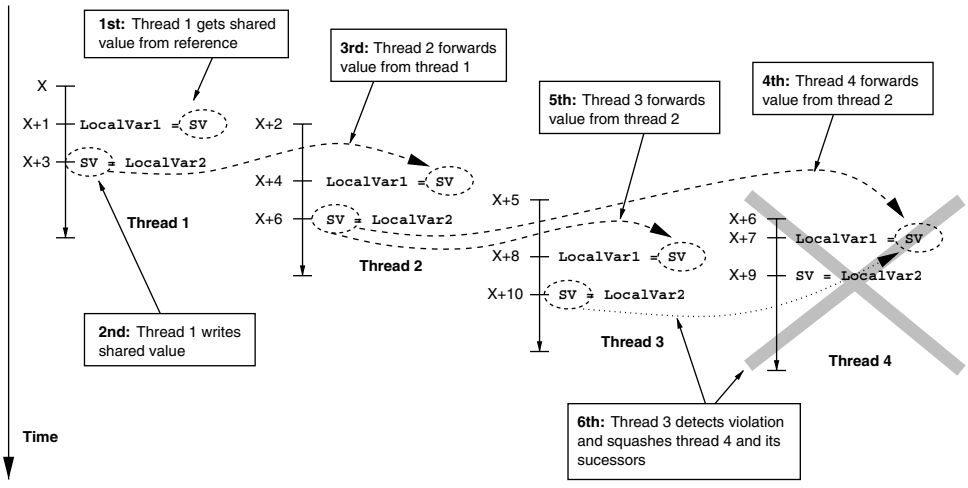

Fig. 1. Speculative parallelization.

and all its successors should be re-executed using the correct values. This is known as a *squash* operation. See Fig. 1.

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 preserving the total 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, its version data is discarded.

The key advantage of speculative parallelization is that there is no need to develop a parallel version of an algorithm in order to execute it in parallel: the speculative parallelization engine does it automatically. Some simple adjustments, within the capabilities of modern compilers, have to be done to the sequential code. A summary of these changes follows:

Thread scheduling: For each loop, blocks of consecutive iterations should be distributed among different threads.
Speculative loads: As long as each thread maintains its own version copy of the shared structure, all original reads to this structure should be augmented with code that searches backwards for the most up-to-date version of the value being read. This operation is known as *forwarding*.
Speculative stores: Any modification to the shared structure performed by a thread may lead to dependence violations, if a successor thread has forwarded an incorrect value. Therefore, all writes to the shared structure should be augmented with code that searches forwards for threads that could have consumed a wrong value. If one is found, the consumer and all its successors are squashed and their execution is restarted with the correct value.
Thread commit: After executing a block of iterations, each thread should call a function that checks its state and performs the commit when appropriate.

3. The convex hull problem

In order to illustrate the different parallelization alternatives, we have chosen the convex hull problem: Given a set S of n points in the plane, the *convex hull* of S is the smallest convex region containing all points in S. We will use $CH(S)$ for the list of vertices of this convex region, which are known to be points of S. In 2D we consider $CH(S)$ an ordered list.

One of the reasons for the choice of the convex hull is that, in addition to be considered a main topic in Computational Geometry [28], convex hulls have attracted the interest of researchers from a wide range of areas. Among others, they have been proved to be useful in applications such as pattern recognition [29–31], including algorithms for detecting human faces [32], reading a license plate [33] or analyzing soil particles [34]; computer graphics [35–37], computerized tomography [38,39], collision detection [40], prediction of chemical equilibrium [41] or ecology [42].

Many different design approaches lead to optimal (or expected optimal) solutions. These include divide and conquer, sweep line, incremental randomized constructions, quickhull, Graham scan and many others. This fact, together with the possibility of adjusting its execution time and expected output size, makes the convex hull an excellent benchmark for testing different architectures and programming models. These adjustments can be easily done by changing the number of input points and/or the shape of the input set.

We will now review some well-known algorithms to compute the convex hull of a given set of points. For the sake of simplicity we focus on the 2D case, although all these algorithms can be generalized to higher dimensions. More details can be found, for example, in [43] or [44].

3.1. Divide and conquer algorithm

The *divide and conquer* algorithm is an elegant method for computing the convex hull of a set of points based on this widely-known algorithmic design technique.

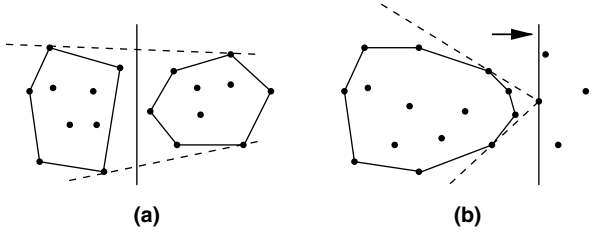

Fig. 2. Computing the convex hull: (a) divide and conquer algorithm; (b) sweep line algorithm.

Given a lexicographically sorted list of points, the algorithm proceeds in the following way: if the list has three or less elements, the convex hull can be constructed immediately. If the list has more elements it is partitioned into two halves, on the left and right halfplanes with respect to a vertical line, thus dividing the set into two equal-sized subsets (see Fig. 2(a)). These subsets are processed recursively by the algorithm. From these two halves of points we get two disjoint convex hulls, that can be merged in linear time. Hence, the overall complexity of this algorithm is optimal $\Omega(n \log n)$.

3.2. Sweep line algorithm

The sweep line algorithm for constructing the convex hull relies also on the lexicographic order of the set of points. Given a sorted list of points, the convex hull is computed adding at each iteration the next one on the list. The two tangents from the point to the existing convex hull are found in logarithmic time and all edges between them are then deleted (see Fig. 2(b)). It is not difficult to see that each point produces two segments and that each segment is deleted at most once. Hence, the amortized cost for the whole computation is optimal $\Omega(n \log n)$.

3.3. Clarkson et al.'s algorithm

Clarkson et al.'s is an incremental randomized algorithm. These algorithms allow to obtain simple codes which are *expected* to run within good time bounds, what makes them a powerful tool in computational geometry and optimization [45–47]. In their general formulation, the input of an incremental randomized algorithm is a set of elements (not necessarily points), for which a certain output needs to be computed. The main feature is that the elements are added in random order, determined by the choice of a random permutation at the beginning. The algorithm proceeds incrementally by adding the input elements one by one and obtaining the intermediate results.

One of the most efficient and easy to implement randomized incremental algorithms for the construction of convex hulls is due to Clarkson et al. [48]. We describe now briefly the algorithm (more implementation details can be found in [49]).

Given S 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 $\mathrm{CH}(R_{i-1})$ is already computed and we want to compute $\mathrm{CH}(R_i)$. Point x_i can be inside or outside $\mathrm{CH}(R_{i-1})$. If it is inside, obviously $\mathrm{CH}(R_i) = \mathrm{CH}(R_{i-1})$. Otherwise, x_i is on the boundary of $\mathrm{CH}(R_i)$. All edges in $\mathrm{CH}(R_{i-1})$ between the two tangents from x_i to $\mathrm{CH}(R_{i-1})$ should be deleted and these two tangents should be added into $\mathrm{CH}(R_i)$. See Fig. 3(a).

The main idea on Clarkson et al.'s 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 Fig. 3(a)) and keeps track of all edges created during the construction of the hull $\mathrm{CH}(R_{i-1})$. For each edge in $\mathrm{CH}(R_{i-1})$, 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 $\mathrm{CH}(R_i)$ that caused its deletion. See Fig. 3(b).

On each iteration the algorithm follows the path from the first triangle constructed to the point being inserted and outputs, if this is outside, one edge that is visible from the point. This way, the cost of searching

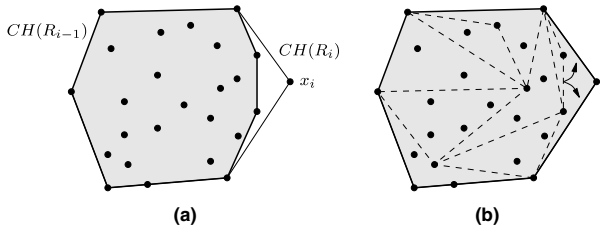

Fig. 3. Clarkson et al.'s algorithm: (a) adding a new point to the convex hull; (b) growth of the convex hull (auxiliary structure shown in dashed lines).

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 [48].

4. Manual parallelization of convex hull algorithms

As we showed in Section 2, there are two ways of exploiting the parallelism inherent to a sequential algorithm: to manually develop a parallel version or to rely on automatic parallelization to do it. In this section we will center our analysis in the manual development of parallel versions of the convex hull algorithms described in the preceding section.

4.1. Design issues

4.1.1. Sorting in parallel

Sorting plays an important role in many algorithms. Executing sorting algorithms in parallel has attracted the attention of researchers during the last thirty years (see, for example, [50,51] or [52]). There is a high dependence on the underlying architecture when choosing a sorting algorithm. Distributed-memory algorithms pay a high price for communication. Hence, low intercommunication profile algorithms are needed. In this work we will use the parallel Mergesort algorithm to sort points in our distributed-memory environment, since it has been proved to show better performance on small or medium-size clusters of computers despite its simplicity [53]. On the other hand, using a shared-memory system does not suffer from communication bottlenecks. We have chosen parallel odd-even transposition (POET) for sorting the input set in our shared-memory system. This algorithm is a parallel variant of bubble sort, simple to implement and with a good performance [51].

4.1.2. Sorted-input, parallel convex hull algorithms

We have seen in Section 3 that the sweep line and divide and conquer algorithms need a sorted input set. Once the set of points is lexicographically ordered, we have to design how to divide the work among processors. We will split the input set into p vertical strips, each of them with n/p points, where p is the number of processors. Each processor applies then the corresponding sequential algorithm in $O(n/p\log(n/p)) \in O(n\log n)$ time. When finished, we obtain p disjoint convex hulls that can be merged as in the last step of the divide and conquer strategy in time $O(p^2)\cdot O(n/p) \in O(n)$, keeping the $\Omega(n\log n)$ lower bound.

4.1.3. Parallel Clarkson et al.'s algorithm

Clarkson et al.'s complexity analysis relies on the randomness on the insertion order of the points. Thus, in parallel, each processor will be given n/p different points chosen at random. Since the input set is no longer sorted, the partial convex hulls obtained are not disjoint and we need a different procedure to merge them.

Our design proposal is to apply again Clarkson et al.'s algorithm sequentially in the merging step. A key observation is that the size of the partial convex hulls is smaller than n/p. Actually, we can determine the order of their size depending on the original shape of the input set: [54,55] prove that the expected size of CH(S) is $O(k\log n)$ for n points uniformly distributed in a k-gon, and $O(\sqrt[3]{n})$ in a disc. Hence, the expected cost of this extra execution will stay below the overall cost.

4.2. Implementation issues

After discussing design issues in order to compute different solutions of the convex hull algorithm in parallel, now we will discuss how they can be implemented using different programming models. We will focus on implementations using highly-efficient manual parallelization with standard unstructured techniques, such as message-passing and multithreading programming. Explicit parallel solutions programmed with structured programming languages such as OpenMP can be straightforwardly derived from the manual implementations described here, with similar performance trends. In conjunction with the techniques described, we will use the master–slave paradigm [56] for distributing tasks among processors.

4.2.1. Manual parallelization: message-passing

This parallel programming tool assumes a distributed-memory model. Data should be manually distributed among processes explicitly by messages and partial results should be sent back to a single process to be merged. One of the main issues to consider when using a message-passing library is related to the details about grouping data to be sent on contiguous memory positions to form a message buffer. The programmer should also properly couple send-receive operations to avoid mismatched communications.

Master–slave paradigm can be easily programmed with message-passing. A general description of its implementation follows. The message-passing library assigns a different number to identify each process. This descriptor can be used by each process to recognize itself as the master or a slave. Data is read and partitioned by the master in as many parts as the number of available slaves reported by the library. The data fragments are then sent to their corresponding slave processes. These slave processes simply receive the data, compute local results and send them to the master. Only when all slaves have returned the partial results, the received solutions are merged sequentially in the master.

Parallel convex hull implementations of both, sweep line and divide and conquer algorithms, has typically two master–slave stages: one for a parallel-sorting followed by a second one that computes in each slave the convex hull of a set of disjoint points. Clarkson's algorithm does not need sorting. Thus, it is implemented as a single master–slave stage with a considerable reduction in the number of messages.

It is worth noting that implementing communications with a message-passing library is not always straightforward. One of the most common problems is how to transfer blocks of unpredictable sizes, such as the convex hulls generated by each slave. Thus, in general each data transfer between processes should be decomposed into two consecutive messages: one containing only the size and a second one with the data.

4.2.2. Manual parallelization: multithreading

On the other hand, multithreading assumes a shared-memory model. In multithreaded programs the main process spawns as many threads as the number of available processors, assigning a particular task to each one. As any thread can access to any portion of data, the issues to be considered now are related to thread synchronization. Shared memory is now used to indicate to each thread the location and sizes of the data which it should work on.

Master–slave paradigm can also be safely implemented with multithreading. Typically, the main thread acts as the master and it directly spawns other threads that run the slave program. There is no need of self-identification by number. The slave threads should wait while the master reads the input data and computes the data-partition limits. After notifying the slaves to begin the process, the master should also wait in order to merge the results, using synchronization mechanisms provided by the library, such as semaphores and barriers.

Communications between master and slaves are carried out by writing and reading on specific shared memory locations. This communication should be carefully matched in order to avoid difficult-to-debug race conditions.

As we showed in the message-passing case, convex hull algorithms with sorted input need two master–slave stages to complete the task. However, as more master–slave stages are issued the cost of spawn-join threads becomes more significant. For these situations it is generally more efficient to spawn threads only once and use barrier mechanisms to synchronize between consecutive master–slave stages.

Table 1
Automatic parallelization of sequential convex hull algorithms

Algorithm	Number of loops	Unrecognized loops	Loops with dependences	Multiple exits	Parallelized loops
Divide and conquer	18	12	0	6	0
Sweep line	8	0	3	5	0
Clarkson et al.'s	11	4	3	4	0

5. Automatic parallelization of convex hull algorithms

5.1. Compiled-based automatic parallelization

As we showed in subsection 2.2, this technique consists of using a parallelizing compiler to obtain a parallel version of a sequential algorithm. We compiled the C version of our convex hull algorithms with a state-of-the-art parallelizing compiler, the Sun Forte Developer 7 C compiler [21], using the `-xautopar` and `-xdepend` options to generate a parallel version of the sequential code.

None of the loops that appear in the code was automatically parallelized by the compiler, due to different reasons. Some of them were not recognized as `for` loops suitable for parallelization, others have multiple exit points and the remaining loops presented possible data dependences among iterations. Consequently, the compiler refused to generate a parallel version of these applications. Table 1 summarizes the results.

5.2. Speculative parallelization

To evaluate the speculative parallelization techniques, we chose the incremental randomized algorithm by Clarkson et al.'s Among the different algorithms described to solve the convex hull problem, only this one is suitable for speculative parallelization, because the remaining algorithms work with sorted input sets, thus producing dependences between each two consecutive iterations.

Clarkson et al.'s algorithm, described in Section 3.3, 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 updated with two new edges. From the speculative execution point of view, most times 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.

In order to compare the performance of the speculative version against the sequential algorithm, we have implemented a Fortran version of Clarkson et al.'s algorithm, augmenting the sequential code manually for speculative parallelization [57]. This task could be performed automatically by a state-of-the-art compiler. A complete and detailed description of these operations can be found in [1]. Results are shown in Section 6.

6. Experimental results

To compare the behavior of the algorithms studied on different architectures, we have implemented parallel versions of the divide and conquer, sweep line and Clarkson et al.'s algorithms. Different versions of each algorithm have been implemented (see Table 2): message-passing and multithreaded as manual parallelization versions, and compiler-based and speculative as automatic parallelization versions.

We have used two different input sets to compare these algorithms. The first one is a set of $n = 40$ million random points in a square, where the final convex hull is expected to have size $O(\log n)$. The second one is a set of $n = 40$ million random points in a disc, where the final convex hull is expected to have size $O(\sqrt[3]{n})$. We will

Table 2
Summary of parallel versions of the algorithms evaluated

	Architecture	Divide and conquer	Sweep line	Clarkson et al.'s
Message-passing	Distributed memory	•	•	•
Multithreaded	Shared memory	•	•	•
Compiled-based	Shared memory	•	•	•
Speculative	Shared memory			•

not analyze degenerate cases like sets of points on a circle. Smaller input sets were not considered, since their sequential execution time took only a few seconds in the systems under test. On the other hand, the use of bigger input sets leads to similar results in terms of speedup [57], so we do not consider them either. The sets of points have been generated using the random points generator in CGAL 2.4 [58] and have been randomly ordered using its *shuffle* function. Consequently, the expected performance of these sets is the same as for any other with the same shape and size.

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

6.1. Shared-memory system setup

The shared-memory versions of our applications were executed on a Sunfire 15 K symmetric multiprocessor (SMP). This system is equipped with 52 UltraSparc-Processors at 900 MHz, each one with a private 64 KByte, 4-way set-associative L1 cache, a private 8 MByte, direct-mapped L2 cache and 32 GByte of shared memory. The system runs SunOS 5.8.

The speculative version of the application was compiled with the Forte Developer 7 Fortran 95 compiler, with the following optimization settings for our execution environment: `-O3 -xchip=ultra3 -xarch=v8plusb -cache=64/32/4:8192/64/1`. The remaining versions were compiled with the Forte Developer 7 C compiler, with the following optimization settings: `-xO3 -xchip=ultra3 -xarch=v8plusb -cache=64/32/4:8192/64/1`.

6.2. Distributed-memory system setup

The distributed-memory versions of our applications were executed on an IBM Regatta system. This system is composed by 50 IBM p690+ Regatta nodes, each node with 32 processors. The p690+ nodes are equipped with IBM Power4+ processors, each one running at 1.7 GHz clock rate, with a private 32 KByte, 2-way set-associative data L1 cache, a private 64 KByte, two-way set-associative instruction L1 cache, a shared L2 cache and 1 GByte of memory per processor. The system runs AIX OS.

The message-passing versions of the applications were compiled with the VisualAge C compiler, version 6, with calls to MPI library routines. We used the following optimization settings for our execution environment: `-q64 -O3 -qarch=pwr4 -qhot`.

6.3. Sequential performance

Table 3 shows the time spent by each sequential algorithm. The performance of the algorithms evaluated is very different. Surprisingly, Clarkson et al.'s algorithm is much faster than divide and conquer and sweep line algorithms, despite having a worst-case $O(n^2)$ complexity. It is also worth noting the difference in the execution time spent by the Power4+ processor of the Regatta system with respect to the performance of the UltraSparc-III processor of the Sunfire system. The difference in speed is not only due to the processors but also to the different optimizations performed by each compiler on the source code.

Table 3
Sequential execution time for the algorithms evaluated

	Processor	Divide and conquer (s)	Sweep line (s)	Clarkson et al.'s (s)
Square input set	UltraSparc-III	79.47	83.05	17.96
Square input set	IBM Power4+	21.03	18.66	6.13
Disc input set	UltraSparc-III	92.64	102.86	17.23
Disc input set	IBM Power4+	19.83	16.82	6.71

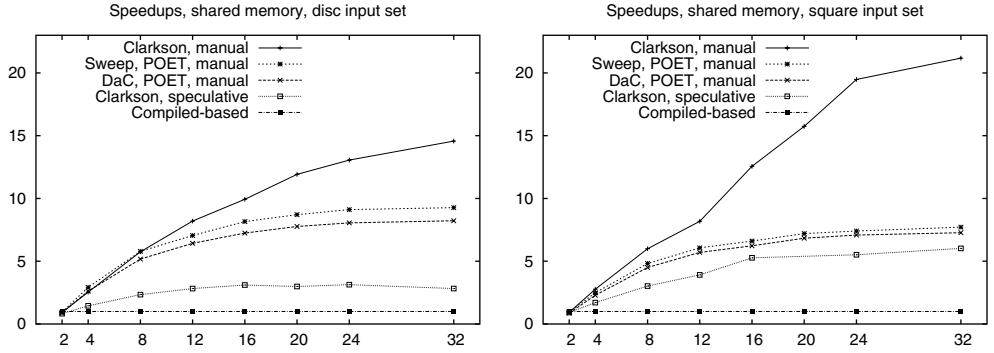

Fig. 4. Relative speedups, shared-memory environment.

6.4. Performance of the algorithms on the shared-memory environment

Fig. 4 shows the performance results of the algorithms considered in terms of relative speedups. The results show that the parallel Clarkson et al.'s manual implementation has the best scalability, with a maximum performance of 14.57× for 32 processors when processing the disc input set and an even better performance of 21.18× for 32 processors with the square input set. Meanwhile, the divide and conquer and sweep line algorithm performances are very similar. Finally, the speculative version of Clarkson's algorithm gives a maximum performance of 3.13× for the disc input set and 6.02× for the square input set. Although not as high as the performance of manually parallel versions, this performance is quite good considering that the parallelizing compiler was not able to extract the inherent parallelism of the sequential algorithm, thus giving a speedup of 1×.

To better evaluate the relative importance of sorting vs. computation in the performance of divide and conquer and sweep line algorithms, Fig. 5 shows the time consumed by both algorithms in each stage of the disc-input-set processing. Both real and normalized time are considered. The results show that sorting consumes most of the time. The normalized bars also show that the scalability of the sorting stage is slightly worse than the one of the convex hull computation.

6.5. Performance of the algorithms on the distributed-memory environment

Fig. 6 shows the performance results of the algorithms considered in terms of relative speedups. The results show that the parallel Clarkson et al.'s manual implementation has again the best scalability, with a maximum performance of 15.16× for 64 processors with the disc input set and 23.05× for 64 processors with the square input set.

The main differences with shared memory are in the performance of the parallel version of the divide and conquer and sweep line algorithms. The maximum performance obtained is below 1.7× for all configurations. The reason is the poor performance of sorting algorithms in distributed-memory machines, motivated by the need of physically sending the data to be sorted to each slave. Shared-memory systems, on

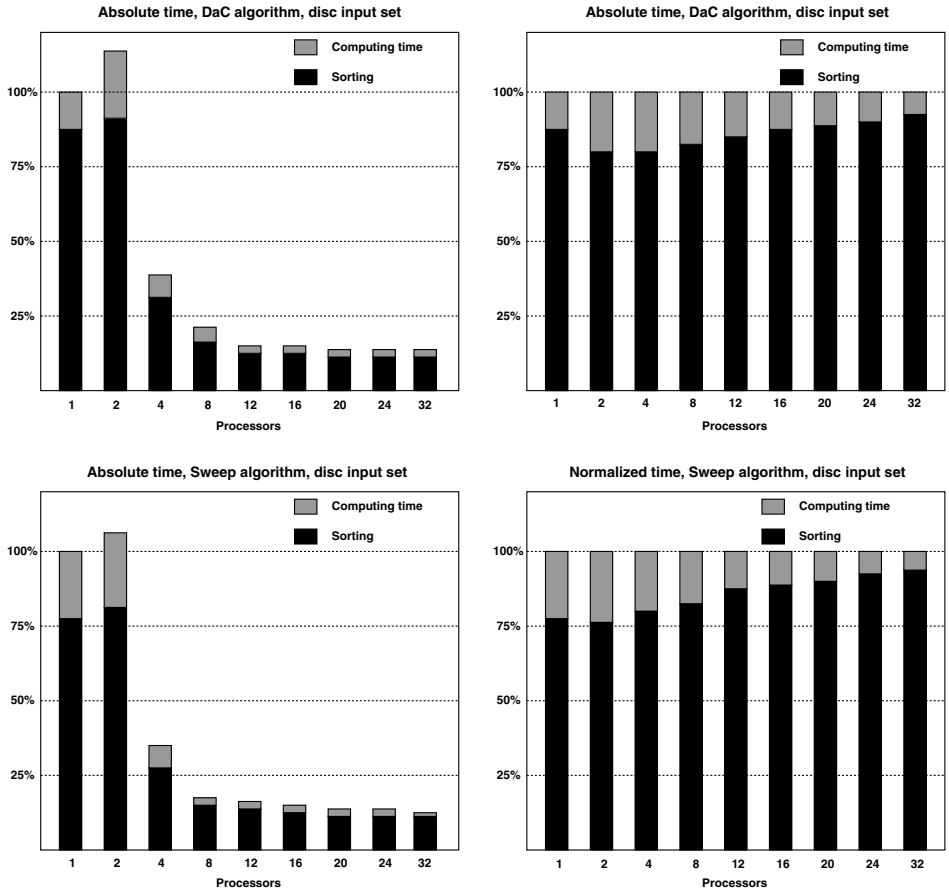

Fig. 5. Sorting-to-computation ratio, shared-memory environment.

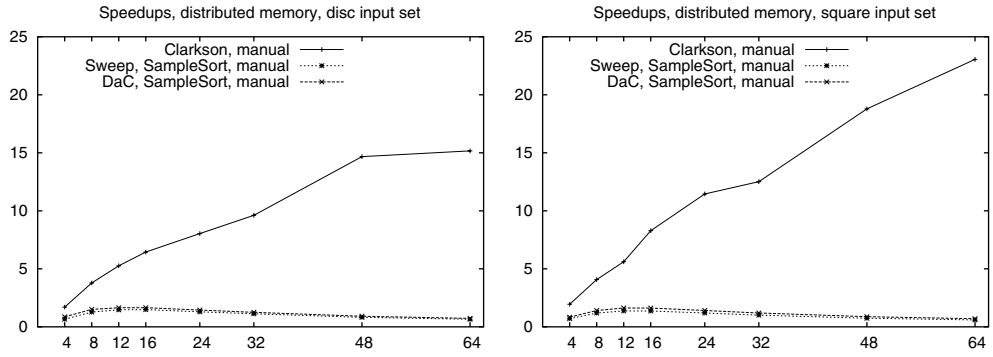

Fig. 6. Relative speedups, distributed-memory environment.

the other hand, allow each slave to work on their particular memory region, avoiding communication overheads.

Finally, Fig. 7 shows the time consumed by divide and conquer and sweep line algorithms at each stage of the disc input set processing. Both real and normalized time are considered. These times include the commu-

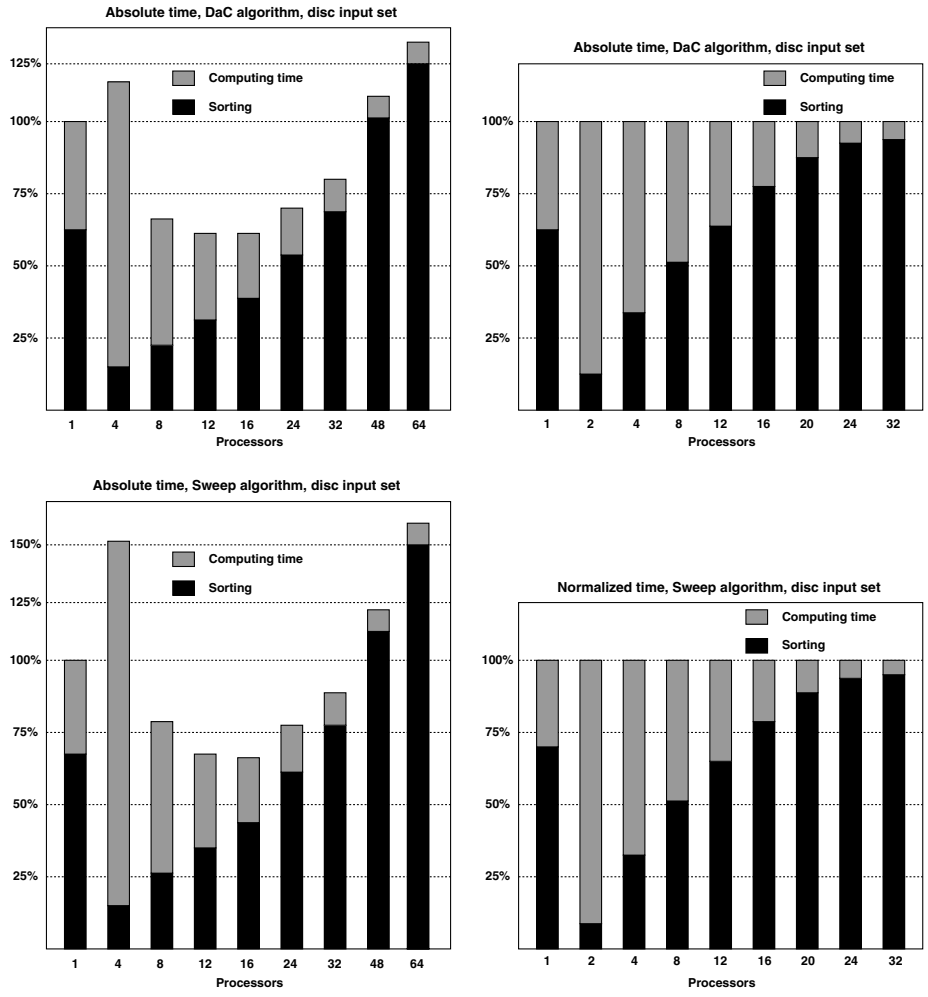

Fig. 7. Sorting-to-computation ratio, distributed-memory environment.

nication overhead at each stage. The results confirm that sorting scales much worse than the two convex hull algorithms in a distributed-memory environment.

7. Conclusions

In this work we have reviewed the parallelizing techniques available nowadays. We have seen that distributed-memory systems require a good knowledge of the problem to be solved, carefully balancing the work to be done on different processors, avoiding the transfer of big data sets and mistrusting apparently simple tasks like sorting. On the other hand, shared-memory systems force a careful synchronization among processors. We have seen that speculative parallelization helps us using a shared-memory system without requiring manual parallelization of the algorithm, since it will automatically try to execute it in parallel. Our results show that programs can be significantly accelerated with this technique, even those where compilers are unable to find any parallelism. The elementary restriction is that the portion we want to execute in parallel should not be inherently sequential.

We have also introduced a new benchmarking problem: The convex hull of a set of points in 2D. Several algorithmic techniques can be applied to this problem, each one being a different challenge for each architecture. Moreover, the size or the shape of the input set can be changed in order to test the behavior of the system

we are dealing with. Thus, we can easily increase the sequential execution time or change the complexity of the output, while being able to predict their order theoretically.

Acknowledgements

Part of this work was carried out while David Orden visited the Departamento de Informática, Universidad de Valladolid, with support of the Universidad de Alcalá. We would like to thank the anonymous reviewers for their valuable suggestions, Manuel Abellanas for his helpful comments concerning the convex hull problem, Pedro Díaz for developing the manual parallelization versions of the convex hull applications, Marcelo Cintra for his contribution in the development of the speculative engine, and Edinburgh Parallel Computing Center (EPCC) and HPCx Consortium for the main computer resources used in this work.

References

[1] M. Cintra, 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), June 2003, pp. 13–24.
[2] D.E. Culler, J. Pal Singh, Parallel Computer Architecture: A Hardware/Software Approach, first ed., Morgan Kauffman, 1999, ISBN 1-55860-343-3.
[3] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, A theory of communicating sequential processes, J. ACM 31 (3) (1984) 560–599.
[4] Message Passing Interface Forum. WWW. Available from: <http://www.mpi-forum.org/>.
[5] The MPI Forum. MPI: a message passing interface. In: Proc. of the conference on Supercomputing'93, ACM, 1993, pp. 878–883.
[6] A. Geist, A. Bequelin, J. Dongarra, W. Jiang, R. Mancheck, V. Suderam, PVM: Parallel Virtual Machine, a User Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, Mass., 1994, ISBN 0-262-57108-0.
[7] Dynamic load balancing of mesh-based applications on parallel systems, in: G. Lonsdale (Ed.), Appl. Math. Modell. 25 (2) (2000) 81–175.
[8] D.R. Butenhof, Programming with POSIX Threads, Addison Wesley Professional, 1997, ISBN 0201633922.
[9] K.S. Gatlin, Trials and tribulations of debugging concurrency, ACM Queue 2 (7) (2004) 67–73.
[10] S. Gorlatch, Send-Recv considered harmful? myths and truths about parallel programming, in: V. Malyshkin (Ed.), PaCT'2001, LNCS, vol. 2127, Springer-Verlag, 2001, pp. 243–257.
[11] D.B. Skillicorn, D. Talia, Models and languages for parallel computation, ACM Comput. Surv. 30 (2) (1998) 123–169.
[12] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel Programming in OpenMP, first ed., Morgan Kaufmann Publishers, 2001, ISBN 1-55860-671-8.
[13] OpenMP organization. WWW. Available from: <http://www.openmp.org>.
[14] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping, The Paderborn University BSP (PUB) library—design, implementation, and performance. In: Proc. IPPS/SPDP'99, San Juan, Puerto Rico, Apr 1999. Computer Society, IEEE.
[15] C.W. Kessler, NestStep: nested parallelism and virtual shared memory for the BSP model. In: Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas (USA), June–July 1999.
[16] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system. In: Proc. of the 5th SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 1995, pp. 207–216.
[17] M. Cole, Frame: an imperative coordination language for parallel programming, Technical Report EDI-INF-RR-0026, Division of Informatics, University of Edinburgh, September 2000.
[18] A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, R. Portales-Fernández, J.A. Caminero-Granja, A preliminary nested-parallel framework to efficiently implement scientific applications, in: M. Daydé et al. (Eds.), VecPar'2004 (Selected Papers), LNCS, vol. 3402, Springer-Verlag, 2005, pp. 539–553.
[19] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, J. Labarta. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors. In: ICS'99, Rhodes, Greece, 1999, pp. 294–301.
[20] Y. Tanaka, K. Taura, M. Sato, A. Yonezawa, Performance evaluation of OpenMP applications with nested parallelism. In: S. Dwarkadas, (Ed.), Proc. 5th Int. Workshop LCR'2000 (Selected Papers), vol. 1915 of LNCS. Springer-Verlag, May 2000.
[21] R.P. Garg, I. Sharapov, Techniques for Optimizing Applications, first ed., Sun Blueprints, Prentice Hall, 2002, ISBN 0-13-093476-3.
[22] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures, first ed., Morgan Kaufmann Publishers, 2002, ISBN 1-55860-286-0.
[23] M. Gupta, R. Nim, Techniques for run-time parallelization of loops, Supercomputing (November) (1998).
[24] L. Rauchwerger, D.A. Padua, The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization, IEEE Trans. Parallel Distributed Syst. 10 (2) (1999) 160–180.
[25] M. Cintra, J.F. Martínez, J. Torrellas, Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In: Proc. of the 27th Intl. Symp. on Computer Architecture (ISCA), June 2000, pp. 256–264.
[26] L. Hammond, M. Willey, 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), October 1998, pp. 58–69.

[27] G. Sohi, S. Breach, T. Vijaykumar, Multiscalar processors. In: Proc. of the 22nd Intl. Symp. on Computer Architecture (ISCA), June 1995, pp. 414–425.
[28] R. Seidel, Convex hull computations, in: J.E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, CRC Press, New York, 1997, pp. 361–375.
[29] M.J. Denber, Method for automatically registering a document having a plurality of pages, 1996. Xerox Corporation, US Patent 5 483 606.
[30] G.T. Toussaint, R.S. Poulsen, Some new algorithms and software implementation methods for pattern recognition research. In: Proc. IEEE Internat. Comput. Software Applications Conf., 1979, pp. 55–63.
[31] S. Yu, M. Thonnat, Description of object shapes by apparent boundary and convex hull, Pattern Recogn. (1993) 95–107.
[32] M. Park, C. Park, M. Park, C. Lee, Algorithm for detecting human faces based on convex-hull, Opt. Express 10 (2002) 274–279.
[33] R.C. González, J.A. Herrera, Apparatus for reading a license plate, 1989. Perceptics Corporation, US Patent 4 817 166.
[34] D. Luo, Pattern Recognition and Image Processing, Horwood Series in Engineering Science, Horwood Publishing, 1998, ISBN 1-898563-52-7.
[35] R. Kumagai, Image processing method, 1992. Ezel Inc., US Patent 5 086 482.
[36] Y. Wexler, R. Chellappa, View synthesis using convex and visual hulls. In: Proceedings of the British Machine Vision Conference, 2001.
[37] C.E. Zair, E. Tosan, Fractal geometric modeling in computer graphics, Fractals. Complex Geom. Patterns Scaling Nature Soc. 5 (2) (1997) 45–61.
[38] J. Tsao, A. Stundzia, M. Ichise, Fully automated establishment of stereotaxic image orientation in six degrees of freedom for technetium-99m-ECD brain SPECT, J. Nucl. Med. 39 (3) (1998) 503–508.
[39] F.H. Little, D.L. Hampson, Method of using a priori information in computerized tomography, 1990, General Electric Company, US Patent 4 969 110.
[40] H.D. Sherali, J.C. Smith, S.Z. Selim, Convex hull representations of models for computing collisions between multiple bodies, Eur. J. Oper. Res. 135 (3) (2001) 514–526.
[41] D. Hildebrandt, D. Glasser, Predicting phase and chemical-equilibrium using the convex-hull of the Gibbs free-energy, The Chem. Eng. J. Biochem. Eng. J. 54 (3) (1994) 187–197.
[42] W.M. Getz, C.C. Wilmers, A local nearest-neighbor convex-hull construction of home ranges and utilization distributions, Ecography 27 (4) (2004) 489–505.
[43] M. De Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry, second ed., Springer, 2000, ISBN 3-540-65620-0.
[44] J. O'Rourke, Computational Geometry in C, second ed., Cambridge University Press, 1998.
[45] K. Mulmuley, Randomized algorithms in Computational Geometry, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, North-Holland Publishing Co., 2000, pp. 703–724 (Chapter 16).
[46] K. Mulmuley, O. Schwarzkopf, Randomized algorithms, in: J.E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, CRC Press, New York, 1997, pp. 633–652 (Chapter 34).
[47] S. Rajasekaran, P.M. Paradalos, J.H. Reif, J.D.P. Rolim (Eds.), Handbook of Randomized Computing: vols. I and II, Combinatorial Optimization, vol. 9, Kluwer Academic Publishers, 2001, ISBN 0-7923-6959-9.
[48] K.L. Clarkson, K. Mehlhorn, R. Seidel, Four results on randomized incremental constructions, Comput. Geom. Theory Appl. 3 (4) (1993) 185–212.
[49] K. Mehlhorn, S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, Cambridge, UK, 2000.
[50] D. Bitton, D.J. DeWitt, D.K. Hsaio, J. Menon, A taxonomy of parallel sorting, ACM Comput. Surv. 16 (3) (1984) 287–318.
[51] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, second ed., Addison-Wesley, 2003, ISBN 0-201-64865-2.
[52] D.S. Hirschberg, Fast parallel sorting algorithms, Commun. ACM 21 (8) (1978) 657–661.
[53] P. Díaz, D.R. Llanos, B. Palop, Parallelizing 2D-convex hulls on clusters: sorting matters. In: Proc. XV Jornadas de Paralelismo, Almería, Spain, September 2004, pp. 247–252, ISBN 84-8240-714-7.
[54] H. Raynaud, Sur l'enveloppe convexe des nuages de points aléatoires dans $\mathbb{R}^n$, J. Appl. Probab. 7 (1970) 35–48.
[55] A. Renyi, R. Sulanke, Über die konvexe hülle von *n* zufällig gerwählten punkten II, Z. für Wahrscheinlichkeit. verwandte Gebiete 3 (1964) 38–147.
[56] H. Davis, J. Hennessy, Characterizing the synchronization behavior of parallel programs, in: PPEALS '88: Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems, ACM Press, 1988, ISBN 0-89791-276-4, pp. 198–211.
[57] M. Cintra, D.R. Llanos, B. Palop, Speculative parallelization of a randomized incremental convex hull algorithm. In: ICCSA 2004 Proc. Intl. Conf. on Computer Science and its Applications, vol. 3045 of LNCS, pp. 188–197, Perugia, Italy, May 2004, Springer-Verlag, ISSN 0302-9743.
[58] CGAL, Computational Geometry Algorithms Library. Available from: <http://www.cgal.org/>.