Parallelizing 2D-Convex Hulls on clusters: Sorting matters

Pedro Díaz, Diego R. Llanos, Belén Palop.

Abstract— **This article explores three basic approaches to parallelize the planar Convex Hull on computer clusters. Methods which are based on sorting the points are found to be inadequate for computer clusters due to the communication costs involved.**

Keywords— **convex hull, sorting, cluster, computational geometry**

I. Introduction

In this article we discuss several methods for parallelizing the computation of planar Convex Hulls. These methods are based on three sequential Convex Hull algorithms. Two of the algorithms we have considered have optimal time complexity and require a preprocessing step where points are lexicographically ordered. The third one shows the best performance on randomized input set, where complexity is expected (but not guaranteed) to be optimal. In this paper we show that the high communications cost associated to computer clusters makes the sorted input algorithms perform badly.

The rest of the article is organized as follows. In Section II we define the Convex Hull of a set of points in the plane and introduce three algorithms for its computation; in Section III we make some preliminary considerations about computer clusters and their programming model; in Section IV we describe different parallel sorting algorithms for clusters; in Section V we resume the different strategies studied for computing the Convex Hull in a cluster; in Section VI we present the environment setup; in Section VII we describe the results obtained; and Section VIII concludes the paper.

II. The Convex Hull

Computing the Convex Hull of a set of points is one of the most studied problems in Computational Geometry. Practical applications of the Convex Hull of a set of points are multiple and diverse, ranging from Statistics Analysis to Pattern Recognition.

Given a set S of points in the plane, the *Convex Hull* of S, $CH(S)$, can be defined as the convex polygon with minimum area containing S. Figure 1 shows an example point set and its associated Convex Hull.

Computing the planar Convex Hull has a complexity lower bound of $\Omega(n \log n)$. An intuitive demonstration follows: The Convex Hull can be used to sort a set of points $S = \{(x, y)/x \in R, y = x^2\}$. Because the lower bound of a compare-based sort is $\Omega(n \log n)$ the lower bound for computing a Convex Hull of a set of points must be the same.

Facultad de Informática, Universidad Politécnica de Madrid. Email: pdiaz@laurel.datsi.fi.upm.es

Departamento de Informática, Universidad de Valladolid. Email: {diego,belen}@infor.uva.es. D. R. Llanos is partially supported by RII3-CT-2003-506079. B. Palop is partially supported by TIC2003-08933-C02-01.

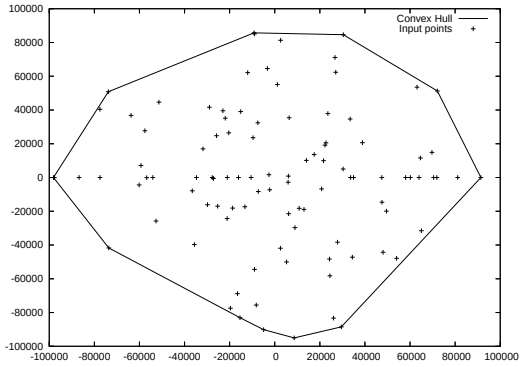

Fig. 1. Convex Hull of a 100-points set

It is out of the reach of this article to review all Convex Hull algorithms in existence. Instead we will briefly explain the algorithms on which our parallelization strategies are based. For a more in-depth review of Convex Hull algorithms the reader is referred to the numerous literature, such as [1], [2] or [3].

As we said before, Convex Hull algorithms can be categorized into two main groups: algorithms that need the input points to be lexicographically sorted and algorithms that operate on non-sorted inputs. We will describe briefly each group.

A. Sorted input algorithms

The *Divide and Conquer* algorithm is an elegant method of calculating the Convex Hull of a set of points based on this widely known algorithm design technique.

Given a 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. These subsets are processed recursively by the algorithm. From these two halves of points we will get two disjoint partial convex hulls, which can be merged in linear time. The complexity of this algorithm is $O(n \log n)$; therefore it is optimal. More information about this algorithm can be found in [1].

The *Incremental* algorithm also calculates the Convex Hull of a set of points in $O(n \log n)$ time. This algorithm works in an incremental way, adding

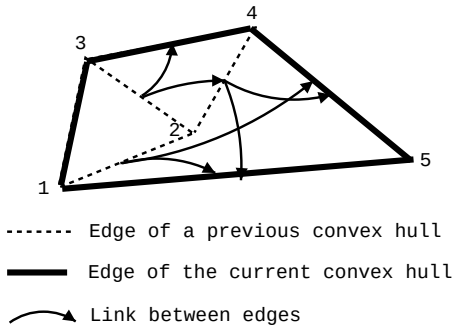

Fig. 2. Relationship between edges in RIC algorithm

on each step a point outside the Convex Hull.

B. Randomized Incremental Construction (RIC) of the Convex Hull

The *Randomized Incremental Construction of the Convex Hull* (RIC from now on) is also an incremental algorithm. Unlike the sorted input Incremental algorithm this one does not need to previously sort the points.

Given a point set S_i and its associated Convex Hull $CH(S_i)$, computing the Convex Hull associated to S_{i+1} (S_i augmented with a new point p) is done in the following way: first we have to determine if p lies within or outside $CH(S)$. If p is inside $CH(S)$ we can ignore p because $CH(S_{i+1})$ will be equal to $CH(S_i)$. If p is outside $CH(S)$ a new Convex Hull is constructed adding p to $CH(S)$, using the same algorithm that was used in the sorted input Incremental algorithm.

How do we decide if p is inside or outside $CH(S_i)$? This can not be done in less than $O(n)$ worst case time, but can be done in $O(\log n)$ average case time if the right data structures are chosen. It is clear that if p is outside $CH(S_i)$ at least one of its edges will not be part of $CH(S_{i+1})$, being replaced with two new edges of $CH(S_{i+1})$. If we place pointers on the replaced edges to the edges that replaced them in $CH(S_{i+1})$ a $O(\log n)$ average case search from the initial convex hull to the current one can be implemented (Figure 2). More information about this procedure can be found in [4].

It is not difficult to see that the RIC algorithm has a worst case runtime complexity of $O(n^2)$. The reason that makes RIC interesting on this study is the fact that it runs in $O(n \log n)$ time on the average case, usually outperforming in execution time other asymptotically optimal algorithms.

III. Parallel programming on clusters

Several theoretical parallel programming models and paradigms such as *PRAM* or *BSP* have been proposed over the years, each one offering its own view on how a parallel processing machine should look like or behave. The algorithms and methods proposed in this article are designed to work on computer clusters, which are real parallel systems. No generalization to other parallel systems has been sought.

A *computer cluster* is a group of (often homogeneous) computer systems, interconnected using some network technology. Each computer system of the cluster is called a *node*.

Parallelism is usually achieved on a cluster running one or several programs on each node, each one exchanging data with the others using some kind of message-passing protocol. The message-passing paradigm is particularly appropriate for programming parallel applications that run on clusters and therefore several standards exist. The Message Passing Interface (MPI) [5] is probably the most popular one and thus it is the one used in the implementation of the algorithms explained here.

Compared to local memory access, computer clusters have a well-known drawback: their slow interconnection speed and bandwidth. These factors limit what kind of parallel algorithms can be adapted to run on such systems; fine-grained parallel programs which involve intensive intercommunication between processes often do not perform or scale properly when implemented on a cluster.

We will use the term *process* to refer to a program run on a node of the cluster which is part of a parallel application. In order to let the processes of a parallel program communicate with each other some kind of identification scheme must be implemented. The MPI identification scheme is the one used in this article. In this scheme each process of the parallel application is assigned a natural number, starting with number zero. The identification number of a process is called the *rank* of the process and identifies the process in the parallel application.

All parallel algorithms described on this article follow the master-slave model of parallel programming. In this model, one of the processes acts as the master and the rest work as slaves. The master's function is to conduct the slave "workforce", supplying them with data to be processed when necessary. The slave processes have a mere computational task in the cluster.

IV. Sorting on computer clusters

Sorting plays such an important role on a lot of Convex Hull algorithms. Two different parallel sorting methods are discussed in this section.

As we said in the preceding section, the main disadvantage of computer clusters is the high cost of interprocess communication. Having this limitation in mind we propose in this section two parallel sorting methods which have a low intercommunication profile. These algorithms, although pretty basic and simplistic, have been found to have better performance on small to medium-size clusters with commodity interconnection networks than other sophisticated algorithms such as parallel shellsort [6] or bitonic sort [7].

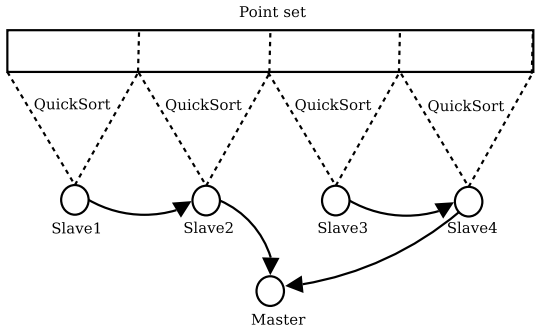

Fig. 3. Parallel mergesort

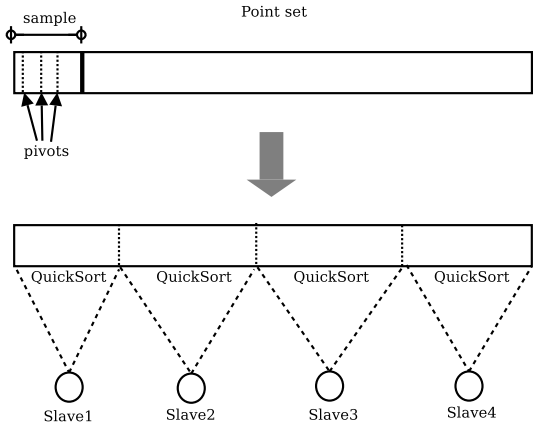

Fig. 4. Parallel Naive Quicksort

A. Parallel mergesort

The first algorithm described here is the parallel variant of the well-known mergesort algorithm. On this algorithm the master subdivides the input points into p sublists, sending one sublist to a different slave process. Each slave receives and sorts its sublist, using a sequential quicksort.

The sorted sublists at each slave should then be merged. In this algorithm we will use a two-phase merge procedure. In the first phase slaves with odd rank R send its sublist to slave $R+1$, which merges it with its local sublist. In the second phase slaves with even rank send their merged sublists to the master process, which constructs the final sorted list of points. Figure 3 illustrates a parallel mergesort with four slaves.

B. Parallel Naive Quicksort

Several parallel variations of the well known quicksort algorithm have been proposed in the literature (see, e.g., [6], [8], [7]). Unfortunately most of these variations imply a level of intercommunication between processes that makes them impractical for computer clusters or other distributed architectures whatsoever.

A simplistic variation of the quicksort algorithm that to the best of our knowledge reduces communication to a minimum is described here. This algorithm is also known as *Parallel Samplesort* [7].

The algorithm starts with the master sorting a small sample of the point set. This sample is used to find $p-1$ pivots, which subdivide the point set into p sublists. These sublists are sent to the slave processes, where they are sequentially sorted and sent back to the master. No merging is necessary. Figure 4 illustrates this algorithm.

V. Computing the Convex Hull in parallel

Table I resumes the strategies studied for computing the Convex Hull in parallel. Strategies `DaC-M`, `DaC-Q`, `Inc-M` and `Inc-Q` are similar: all of them perform the following steps:

1. Sort the input points in parallel (either using Naive Quicksort or Mergesort)
2. At the master processor, partition the sorted points into p sublists, and send each sublist to a different slave processor
3. At each slave processor, compute the Convex Hull of the received sublist (either using the Divide and Conquer algorithm or the Incremental algorithm). Return the result to the master
4. At the master processor, receive the partial Convex Hulls from each slave and merge them.

$CH(S)$	**Mergesort**	**Naive Quicksort**	**No sort**
DaC	`DaC-M`	`DaC-Q`	
Incr.	`Inc-M`	`Inc-Q`	
RIC			`RIC-RIC`

TABLE I
Parallelization strategies and their variants: Divide-and-Conquer (DaC), Sorted-input Incremental (Incr.) and Randomized Incremental (RIC).

Strategy `RIC-RIC` is defined as follows:

1. At the master processor, partition the point set into p sublists, and send each sublist to a different slave processor
2. At each slave processor, compute the Convex Hull of the received sublist using the sequential RIC algorithm. Return the points of the resulting Convex Hull to the master
3. At the master processor collect the points from each slave processor and calculate its Convex Hull using the sequential RIC algorithm

VI. Environment setup

All the experimental results showed in the next section were measured on a small cluster of computers. The technical specifications are described in Table II.

Although each node has two processors only one has been used in our experiments, to force network communication among processes. The message-passing programming interface used is LAM, a widely used MPI implementation. The algorithms are implemented in the C programming language (source code available on request).

Architecture:	Intel x86
CPU:	2× Intel Xeon™, 2.4 Ghz
RAM size:	896 MB
Sec. storage:	30GB SCSI hard disk
Network:	Dual Gigabit Ethernet
O.S.	GNU/Linux

TABLE II
PER-NODE HARDWARE DESCRIPTION

Algorithm	**Time (s)**
RIC	5.327
Divide and Conquer	26.11
Incremental	29.829

TABLE III
RUNNING TIME OF THE SEQUENTIAL ALGORITHMS

VII. RESULTS

Experimental results measured in the cluster follows. The main input set consists of 40×10^6 points in a disc-like shape. This shape is particularly appropriate because the resulting Convex Hull will have many edges but without being a degenerate case.

Table III gives the running time of the sequential Convex Hull algorithms on a single node of the cluster. A sequential Quicksort was used to sort the points when needed.

Figure 5 shows the number of *slave* processes versus the total running time of the parallel Convex Hull algorithms explained in this article.

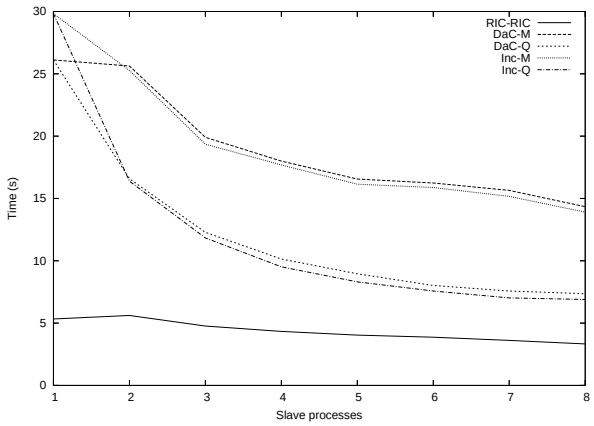

Fig. 5. Number of slave processes with respect to total running time for each parallel Convex Hull algorithm

Figure 6 shows absolute speedups calculated against the best sequential algorithm (which for average case input is `RIC`) of each parallel algorithm. Figures 7 and 8 show the relative speedups and efficiencies (calculated against the sequential version of the same algorithm).

Figures 9, 10, 11, 12 and 13 show running time breakdowns of each algorithm, decomposed in three factors:

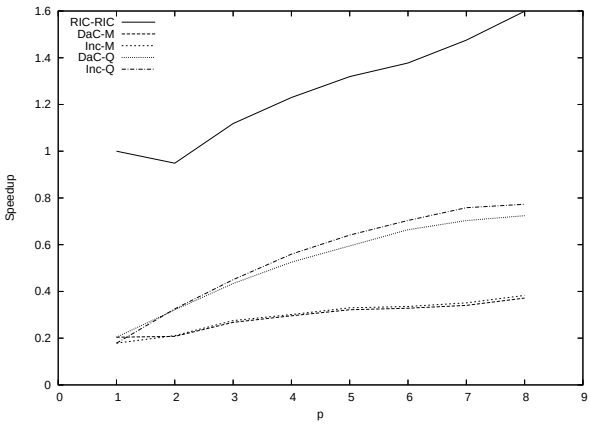

Fig. 6. Absolute speedups

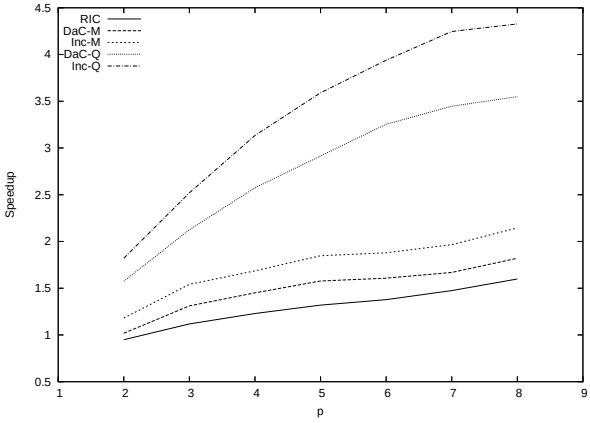

Fig. 7. Relative speedups

- Exec. time: Time spent by the program executing the algorithm's code
- Contention + comms. time: Time spent by the message-passing library (LAM) on communications and synchronization
- LAM overhead: Time spent inside the LAM libraries when not blocked or exchanging data

Since not all processes of a parallel application have the same execution pattern it is difficult to exactly measure and classify their running times into one of the three categories mentioned above. To obtain a general view of the times involved, the approach used here was to measure each of these factors on each process and sum them to obtain three values which then are normalized to divide each running time into three sections. Since we used certain non-blocking communication primitives, some periods of time are used both for the execution of the algorithm and for the underlying communications. We have accounted them as communication time.

The most relevant result is that the RIC algorithm is the fastest one in terms of execution time, although its relative speedup is the lowest one. This behavior can be explained in terms of Amhdahl's Law. In the execution of the algorithm we have a communication time (about 2.8 seconds in our case) in order to transfer a partition of the dataset to each one of the slaves. This time is fixed for any number of slaves

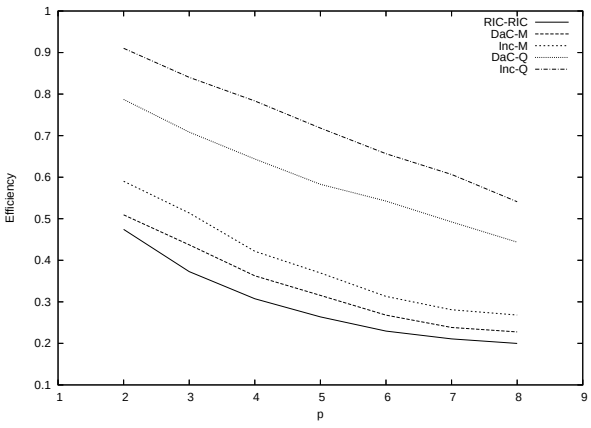

Fig. 8. Relative efficiencies

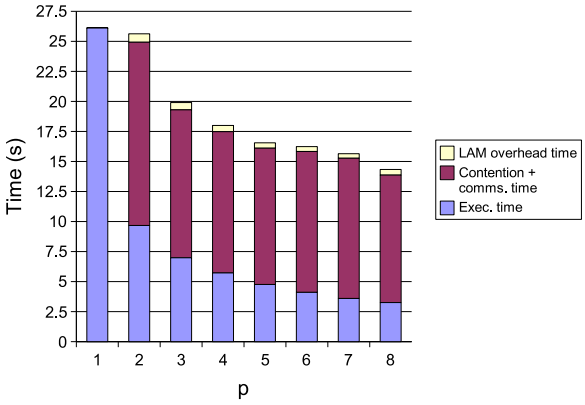

Fig. 9. Time decomposition for `DaC-M`

between two and eight. The communication can not be avoided, as long as the cluster is a distributed-memory system. According with Amhdahl's Law, this fixed time imposes a limit to the speedup that can be obtained with a parallel version of the algorithm, regardless of the number of slaves (Figure 14).

Sending data across the network can be done in $O(n)$ time. As we said earlier, the calculation of the Convex Hull can not be done in less than $O(n \log n)$ time. Therefore, as the input set size increases, the time to compute $CH(S)$ will be more important than the time needed to send the data across the network.

VIII. Conclusions

Sorting is a preliminary step mandatory in many Computational Geometry algorithms. The need to previously sort the input point set introduces a whole family of problems which can be summarized as follows:

- More communication is involved. In shared-memory parallel systems with high speed interconnection buses this is not as problematic as in clusters, where each byte is costly sent or received.
- Parallel sorting is not a trivial subject. By requiring the points to be sorted (in parallel) we

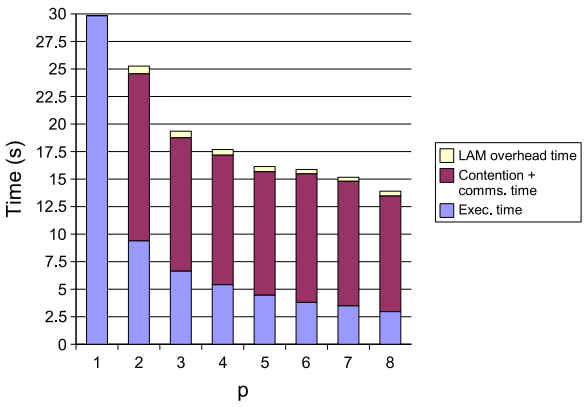

Fig. 10. Time decomposition for `Inc-M`

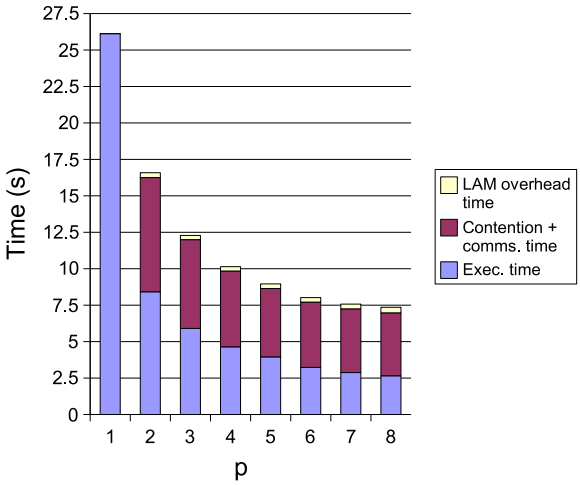

Fig. 11. Time decomposition for `DaC-Q`

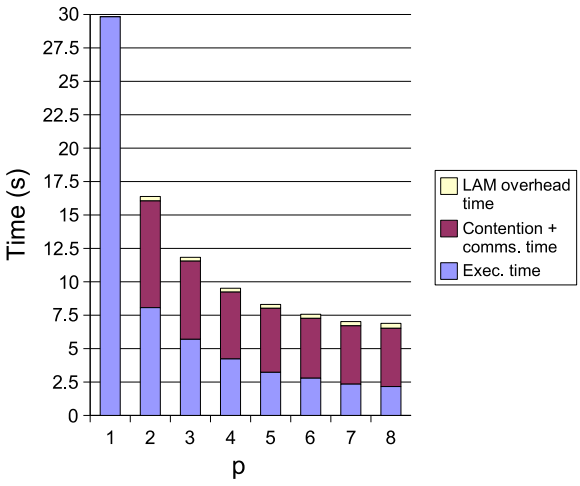

Fig. 12. Time decomposition for `Inc-Q`

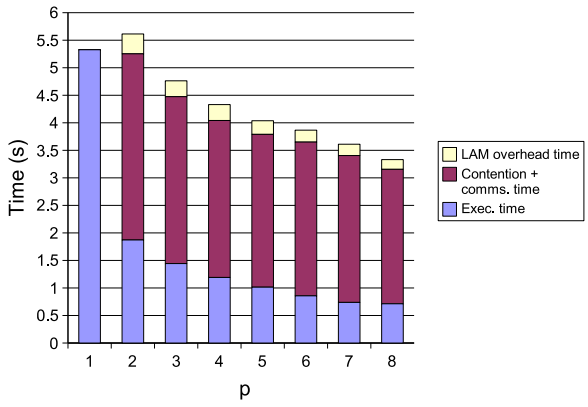

Fig. 13. Time decomposition for `RIC-RIC`

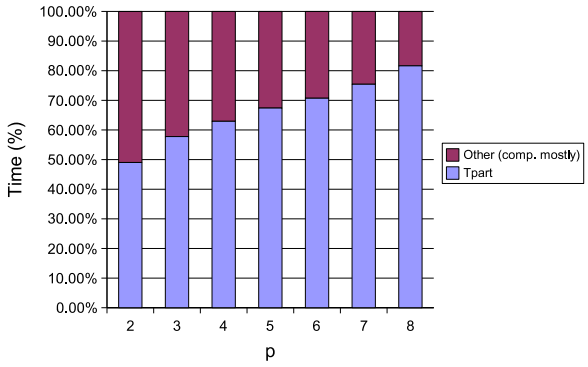

Fig. 14. Influence of transfer time (Tpart) in the total running time on the parallel RIC algorithm

are adding complexity to an otherwise straightforward solution.

Our first conclusion is that Convex Hull algorithms which require a preliminary sort of the input set are not appropriate for parallelization on a cluster. Methods that do not require this preliminary sort are likely to be much faster even if they are not asymptotically optimal.

We have also found that the communication cost in computer clusters is too high for focusing on a purely data-divide approach for this problem, regardless of the particular algorithm used. Heuristics such as an initial filtering to discards points that will not be in the final convex hull may be applied to reduce the input size and thus the time spent on communications.

References

[1] Joseph O'Rourke, *Computational Geometry in C*, Cambridge University Press, second edition, 1998.

[2] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, second edition, 2000.

[3] Franco P. Preparata and Michael Ian Shamos, *Computational Geometry: An Introduction*, Springer Verlag, 1993.

[4] Kurt Mehlhorn and Stefan Nher, *LEDA : A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999.

[5] MPI Forum, "MPI: A message passing interface," in *Proc. of the 1993 Intl. Conf. on Supercomputing*, november 1993, pp. 878–883.

[6] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to Parallel Computing*, Addison Wesley, second edition, 2003.

[7] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina, *Parallel Computing Works*, Morgan Kaufmann Publishers, 1994.

[8] Youran Lan and Magdi A. Mohamed, "Parallel quicksort in hypercubes," in *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*. 1992, pp. 740–746, ACM Press.

[9] Janez Brest, Aleksander Vreze, and Viljem Zumer, "A sorting algorithm on a pc cluster," *ACM Symposium on Applied Computing*, vol. 2, 2000.

[10] M. Diallo, Alfonso Ferreira, Andrew Rau-Chapling, and Stephane Ubeda, "Scalable 2d convex hull and triangulation algorithms for coarse grained multicomputers," 1996.