Finding Hamiltonian Paths in Tournaments on Clusters - A Provably Communication-Efficient Approach [*]

Chun-Hsi Huang
Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260
ch25@cse.buffalo.edu

Xin He
Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260
xinhe@cse.buffalo.edu

ABSTRACT

This paper presents a general methodology for the communication efficient parallelization of graph algorithms using divide-and-conquer approach and shows that this class of problems can be solved in cluster environments with good communication efficiency.

Specifically, the first practical parallel algorithm, based on EREW BSP model, for finding Hamiltonian paths in tournaments is presented. For all commercially available parallel computing environments, this algorithm uses only $(3\log p+1)$ communication supersteps, which is independent of the tournament size, and can reuse the existing linear-time algorithm in sequential setting.

Experiments have been carried out on a Linux cluster of 32 Sun Ultra5 computers and SGI Origin 2000 with 32 R10000 processors, using MPI. The algorithm performance on Linux Cluster reaches 75% of the performance on SGI Origin 2000 when the tournament size is about one million.

Keywords

BSP, MPI, Tournament, Cluster Computing

1. INTRODUCTION

1.1 Motivation and Contribution

A large number of parallel computing problems in many fields are defined in terms of graphs. However, graph problems have been shown to have considerably less internal structures than many other problems studied. This results in highly data-dependent communication patterns and makes it difficult to achieve communication efficiency, especially while using clusters, which are becoming more and more popular parallel computing platforms due to their high availability and economic appeal. Yet, no prior attempt has been made to tackle graph problems on clusters.

Balancing the load assigned to different processors and minimizing the communication overhead are the core problems in achieving high performance on parallel or distributed systems. Normally, algorithms using divide-and-conquer often need to trade off between these two. However, parallel divide-and-conquer specifies an important class of problems in many fields, such as computational geometry [1, 2], graph theory [7, 12], numerical analysis [9, 6] and optimization [3]. Therefore, designing an approach which reduces the interprocessor communication overhead while balancing the workload becomes essential, especially when implementing such algorithms on cluster environments.

This paper focuses on one such typical example, finding Hamiltonian paths in tournaments, a graph problem that has divide-and-conquer algorithm in sequential setting. Many applications in fields like distributed database, banking system and task schedulings can reduce to this problem. A *tournament* is a directed graph $\Im = (V,E)$ in which, for any pair of vertices u, v, either $(u,v) \in E$ or $(v,u) \in E$, but not both. This models a competition involving n players, where every player competes against every other one. The authors present load balanced and communication efficient partitioning strategies which generate subtournaments as evenly as possible for each processor. Subtournament Hamiltonian path computations are then carried out using existing sequential algorithm. The major features of the main algorithm in this paper are as follows :

1. Inter-processor communication overhead in "divide" stage (partitioning) is reduced by routing data only after the final destination processor has been determined, reducing large amount of data movement.
2. Code reuse from existing sequential algorithm is maximized in "conquer" stage.
3. No additional communication overhead is required in "merge" stage.

For theoretical completeness, this algorithm has also been revised for parallel computing platforms where either each individual processor is of extremely limited local memory or the ratios of computation and communication throughputs are low.

To demonstrate the practical relevance, the algorithm has been implemented on a Linux cluster of 32 333 MHz Sun Ultra 5 processors. For performance comparison, experiments have also been carried out on SGI Origin 2000 with 32 R10000 processors. Speedups on both platforms scale well from 2 to 32 processors. Most importantly, the experimental results conclude that the algorithm performance on Sun Cluster is above 75% of the performance on SGI Origin 2000 while tournament size is about one million. (Note: Refer

[*]Computational resources and technical support provided by Center for Computational Research at State University of New York at Buffalo (UB CCR).

to Remarks section.)

1.2 Practical Parallel Programming Models

1.2.1 EREW BSP

A model that has received much attention, among the practical ones so far, is the *Bulk Synchronous Parallel* (BSP) Model, proposed by Valiant in 1990 [14]. It was one of the first models that took communication issues into account and abstracted the features of a parallel machine in a few parameters. A BSP machine consists of p processor/memory components communicating through some interconnection medium. A BSP algorithm consists of a sequence of *supersteps.* In each superstep, the processors operate independently performing local computations and global communications by sending and receiving messages. The messages sent in the current superstep is assumed to be received in the next superstep. At the end of a superstep, a barrier synchronization is realized. The run time of a BSP algorithm is the sum of the run times of the supersteps. Let L denote the minimal time of a superstep (measured in basic computation units) and g the ratio of computation and communication throughput.

The term *h-relation* is used to denote a routing problem where each processor has at most h words of data to send to other processors and where each processor is also due to receive at most h words of data from other processors. In each BSP superstep, if at most w arithmetic operations are performed by each processor and the data communicated form an *h-relation*, then the cost of this superstep is $w + h * g + L$. The cost of a BSP algorithm using S supersteps is simply the sum of the costs of all S supersteps and can be given in the following form:

$$\begin{aligned} BSP\ cost &= comp.\ cost + comm.\ cost + \ synch.\ cost \\ &= W + H * g + L * S \end{aligned}$$

where H is the sum of the maximums of the *h-relation*s in each superstep and W is the sum of the maximums of the local computations in each superstep.

1.2.2 weak-CREW BSP

In many cases some slight augmentation for the original EREW BSP model facilitates the algorithm description. A slight variation of the original (EREW) BSP model, weak-CREW BSP, was proposed by Goodrich [8]. It is essentially the same as the BSP model with the following exception :

> During a communication superstep, messages can be duplicated by the interconnection media as long as the destinations for any message are a contiguous set of processors $i, i+1, \cdots, j$. Even with message duplication, the number of messages received by a processor in a superstep is required to be at most $h = O(\frac{n}{p})$.

Another general-purpose model, *Coarse Grained Multicomputer* (CGM), proposed by Dehne [5], is essentially similar to weak-CREW BSP. A CGM computer consists of p processors $P_0, \cdots P_{p-1}$, where each processor has $O(\frac{n}{p})$ local memory. The processors can be connected through any communication medium, i.e. any interconnection network or shared memory. Typically, the local memory is considerably larger than $O(1)$. This feature gives the model its name "coarse grained". A CGM algorithm consists of alternating local computation and global communication rounds. In a communication round, a single h-relation (with $h = O(\frac{n}{p})$) is routed, i.e., each processor sends $O(\frac{n}{p})$ and receives $O(\frac{n}{p})$ data. The run time of the CGM algorithm is the sum of the run times of the computation and communication rounds. A CGM algorithm with λ rounds and computation cost T_{comp} corresponds to a BSP algorithm with λ supersteps, communication cost $O(g\lambda\frac{N}{p})$, where N is the input size, and the same computation cost T_{comp}.

Compared to the BSP model, a computation/communication round in the CGM model is equivalent to a superstep in the BSP model with $L = \frac{n}{p} \times g$, but includes also the "packing" requirement. It is, therefore, a slightly more powerful model than the BSP model. In general, for those problems whose best possible sequential algorithms take $T_s(n)$ time, ideally algorithm designers would like to design a CGM algorithm using $O(1)$ communication rounds and $O(\frac{T_s(n)}{p})$ total local computation time.

1.2.3 In This Paper

Throughout this paper, the authors design algorithms based on the original EREW BSP model; namely, neither network broadcast nor combining capability is assumed. In addition, although the original BSP model definition does not limit the local memory size, the authors follow the CGM constraint that the local memory size of each processor is only $O(\frac{N}{p})$. The authors also follow the CGM constraint that, in each communication round, only $O(\frac{N}{p})$-relation is routed. The algorithm logically follow the EREW BSP model. Yet, while implementing, barrier synchronization is used only when necessary.

1.3 Organization of the Following Sections

The rest of this paper is organized as follows : Section 2 presents the BSP-style (EREW) algorithm for finding Hamiltonian paths in tournaments and details the BSP cost analysis for both local computation and interprocessor communication. Section 3 details the experimental results on a Linux cluster of 32 Sun Ultra5 computers and SGI Origin 2000 with 32 R10000 processors. Section 4 concludes this paper.

2. THE BSP ALGORITHM AND COST

2.1 The BSP Algorithm

Assume a tournament $\mathfrak{T} = (V, E)$ is given, where $|V| = n$, and $|E| = n^2 = N$. Throughout this paper, the size of a tournament $\mathfrak{T} = (V, E)$ will always refer to $|E|$. A trivial, but useful, fact is that any induced subgraph of a tournament is also a tournament.

Given $\mathfrak{T} = (V, E)$, define $\mathfrak{T}(V')$ to be the tournament (induced subgraph) on V', where $V' \subseteq V$. Define *u dominates v* if $(u, v) \in E$, and denote this property by $u > v$. Note that since the directions of the arcs are arbitrary, the domination relation is not necessarily transitive. The notion of domination is extended to sets of vertices: Let A, B be subsets of V. *A dominates B* ($A > B$) if every vertex in A dominates every vertex in B. For a given vertex v, the rest of the vertices are categorized according to their relations with v : $W(v)$ is the set of vertices that are dominated by v and $L(v)$ is the set of vertices that dominate v.

Much work has been done on tournaments [4]. In this paper, the authors concentrate on a classical result : every tournament has a Hamiltonian path [11, 13] and start by stating the theorem for Hamiltonian path.

THEOREM 2.1. *Every tournament contains a Hamiltonian path.*

PROOF. By induction on the number, n, of vertices, the result is clear for $n = 2$. Assume it holds for tournaments on n vertices. Considering a tournament $\mathfrak{T}$ on $n+1$ vertices, let v be an arbitrary vertex of V. By induction hypothesis $\Im(V-\{v\})$ has a Hamiltonian path $v_1, v_2, \ldots, v_n$. If $v > v_1$, then $v, v_1, \cdots, v_n$ is a Hamiltonian path of $\Im$. Otherwise let i be the largest index such that $v_i > v$. If $i = n$ then $v_1, \cdots, v_n, v$ is a Hamiltonian path. If not, $v_1, \cdots, v_i, v, v_{i+1}, \cdots, v_n$ is the desired Hamiltonian path. $\square$

THEOREM 2.2. *In a tournament $\Im$ on n vertices, there exists a vertex v, referred to as mediocre player, for which both $L(v)$ and $W(v)$ have at least $\lfloor \frac{n}{4} \rfloor$ vertices.*

PROOF. Let $I = \{u | \ d_{in}(u) \geq d_{out}(u)\}$. $O = V - I$. Assume without loss of generality that $|I| \geq |O|$. By the pigeonhole principle, there exists a vertex v whose out-degree in $\Im(I)$ is no less than its in-degree in $\Im(I)$. Thus $d_{out}(v) \geq \lfloor \frac{|I|}{2} \rfloor \geq \lfloor \frac{n}{4} \rfloor$ and $d_{in}(v) \geq d_{out}(v) \geq \lfloor \frac{n}{4} \rfloor$ by definition. $\square$

Throughout this paper, $H_\Im$ will be used to denote a Hamiltonian path of a tournament $\Im$. Assuming a mediocre player of $\Im$ is v_m, an observation reveals the fact that

$$H_\Im = H_{\Im(L(v_m))}, v_m, H_{\Im(W(v_m))}.$$

This observation, along with Theorem 2.2, motivate the main algorithm design.

In order to split tournaments as evenly as possible in the partitioning stage, in each round, the mediocre player whose indegree and outdegree are closest is selected and used to split the tournaments from the preceding partitioning round. An important idea here is, during "divide" stages only the mediocre players are communicated among processors. Subtournaments are moved to destination processors only after the splitting process ends, when each subtournament now can actually fit in a single processor. Sequential algorithm for finding Hamiltonian path is now applied in parallel, and the results, along with all the selected mediocre players form the final Hamiltonian path.

The BSP algorithm for finding Hamiltonian paths in tournaments uses the following major data structures :

- $\varepsilon[i][j] \ (0 \leq i, j \leq n-1) \ = \begin{cases} 1 & \text{if } v_i > v_j \\ -1 & \text{if } v_i < v_j, \end{cases}$
- $\wp[i][j] \ (0 \leq i \leq \log p - 1, 0 \leq j \leq n-1) \ = \begin{cases} 0 & \text{if } v_j \in \bigcup_{k=0}^{2^{i-1}-1} L(v_{m_k}) \text{ in partitioning round } i \\ 1 & \text{if } v_j \in \bigcup_{k=0}^{2^{i-1}-1} W(v_{m_k}) \text{ in partitioning round } i, \end{cases}$
- $\iota[i](\vartheta[i]) \ (0 \leq i \leq n-1)$ = indegree (outdegree) of v_i.

Assume BSP processors are labeled $P_0, P_1, \cdots, P_{p-1}$. Processor P_i initially contains

- $\varepsilon[i(\frac{n}{p}) \cdots i(\frac{n}{p}) + (\frac{n}{p}) - 1][0 \cdots n-1]$,
- $\wp[0 \cdots \log p - 1][0 \cdots n-1]$ (initially 0's) and
- ι, $\vartheta[0 \cdots n-1]$.

The algorithm is described as below, with each superstep accompanied by the corresponding MPI primitives, which will be detailed in Section 3.

Algorithm HPT :

1. Each processor computes $\iota[j], \vartheta[j]$ for all local v_j's.
   (MPI_AllReduce)
2. Each processor performs the following steps of partitioning round i ($i = 1$ to $\log p$) :
   (a) Identify the mediocre players $v_{m_0}, \cdots, v_{m_{2^{i-1}-1}}$ in tournaments $\Im_0, \cdots, \Im_{2^{i-1}-1}$:
   ($\Im_0 = \Im$)
      i. Each processor P_k $(0 \leq k \leq p-1)$ locally computes (if any) one mediocre player v_{k_t}, where $\iota[k_t] \geq \frac{|\Im_t|}{4^i}$ and $\vartheta[k_t] \geq \frac{|\Im_t|}{4^i}$, for every $\Im_t$ $(0 \leq t \leq 2^{i-1}-1)$. ($\varepsilon[x][0 \cdots n-1]$ is considered to be in $\Im_{\sum_{y=0}^{i-1} 2^y \times \wp[y][x]}$.)
      ii. Each processor P_k $(0 \leq k \leq p-1)$ sends v_{k_t} to P_t. $(0 \leq t \leq 2^{i-1}-1)$
      (MPI_Send, MPI_Recv, MPI_Barrier)
      iii. Each processor P_k $(0 \leq k \leq 2^{i-1}-1)$ computes the mediocre player v_{m_k} for $\Im_k$.
      iv. Each processor P_k $(0 \leq k \leq 2^{i-1}-1)$ broadcasts v_{m_k} to all the other $p-1$ processors.
      (MPI_Bcast, MPI_Barrier)
   (b) Split tournaments $\Im_0, \cdots, \Im_{2^{i-1}-1}$:
      i. Each processor P_k $(0 \leq k \leq p-1)$ locally performs the following :
         Assign $\wp[i][j]$ 0 (1, respectively) if $v_j \in \bigcup_{k=0}^{2^{i-1}-1} L(v_{m_k})$ $(v_j \in \bigcup_{k=0}^{2^{i-1}-1} W(v_{m_k})$, respectively).
      (Conceptually, let $\Im_{2k} = \Im(L(v_{m_k}))$ and $\Im_{2k+1} = \Im(W(v_{m_k}))$. $(0 \leq k \leq 2^{i-1}-1)$)
   (c) Update incoming and outgoing degrees for all vertices :
      i. Set all $\varepsilon[m_k][j]$'s and $\varepsilon[j][m_k]$'s to 0's.
      ii. Decrease $\vartheta[j]$ by 1 if $v_j \in \bigcup_{k=0}^{2^{i-1}-1} L(v_{m_k})$.
      Decrease $\iota[j]$ by 1 if $v_j \in \bigcup_{k=0}^{2^{i-1}-1} W(v_{m_k})$.
      (MPI_Barrier)
3. Each processor P_k sequentially processes its local data by sending $\varepsilon[x][0 \cdots n-1]$ to $P_{\sum_{y=0}^{\log p-1} 2^y \times \wp[y][x]}$, where $0 \leq k \leq p-1$ and $k(\frac{n}{p}) \leq x \leq k(\frac{n}{p}) + \frac{n}{p} - 1$.
   (MPI_Pack, MPI_Unpack, MPI_Send, MPI_Recv, MPI_Barrier)
4. Each processor P_k $(0 \leq k \leq p-1)$ computes $H_{\Im_k}$.
   (MPI_Barrier)

2.2 Computation and Communication Complexities

Assume the sequential algorithm for finding Hamiltonian paths in tournaments of size N is $T_s(N)$. Then the BSP cost breakdown of algorithm HPT can be derived as Table 1.

Therefore, the following theorem is derived :

THEOREM 2.3. *The Hamiltonian path of a tournament $\Im = (V, E)$, whose $|V|$=n and $|E|$=n^2=N, can be determined on a p-processor BSP computer with $O(\frac{N}{p})$ local memory, using $T_s(\frac{N}{p}) + \theta(\frac{N}{p} \log p)$ local computation time and $3(\log p + 1)$ supersteps, in which the total communication time is $g \times (\frac{N}{p} + 2p \log p + p)$.*

The number of supersteps needed is $3(\log p + 1)$, independent of the tournament size N. And the total size of the routed relations is only $(\frac{N}{p} + 2p \log p + p)$, enabling the predictable communication efficiency in cluster environments. This technique can be applied in solving problems cited in Section 1 in the same computation and communication efficient manners on clusters.

Table 1: **Algorithm HPT : BSP Cost Breakdown**

	Cost		
Step	comp.	comm.(in relation)	synch.
1	$\frac{N}{p}$	p	L
2(a)i	$(\frac{N}{p})\log p$		
2(a)ii		$p\log p$	$L\log p$
2(a)iii	$(\frac{N}{p})\log p$		
2(a)iv		$p\log p$	$L\log p$
2b	$(\frac{N}{p})\log p$		
2(c)i	$(\frac{N}{p})\log p$		
2(c)ii	$(\frac{N}{p})\log p$		$L\log p$
3	$\frac{N}{p}$	$\frac{N}{p}$	L
4	$T_s(\frac{N}{p})$		L

3. EXPERIMENTAL RESULTS

To demonstrate the practical relevance, algorithm HPT has been implemented on UB CCR's Linux cluster of 64 333 MHz Sun Ultra 5 processors. The processors are equipped with 100 Mbit/s Sun Happy Meal ethernet card and 3 Nortel Network 450-24T Ethernet switches for cascading modules to connect backplanes. The operating system is RedHat Linux 6.0. A portable MPI implementation MPICH 1.2.2 is used to support the interprocessor communication and PBS 2.1p15 is used for batch job submissions.

For performance comparison, the code has also been tested on UB CCR's SGI Origin 2000 with 32 R10000 processors, with distributed shared memory banks, accessed by a high-speed and low-latency interconnection network, CrayLink. The SGI machines run on Irix operating system, also supporting a portable MPI implementation, MPT, and PBS batch job queues.

MPI is a standard specification for message passing libraries. MPT (on SGI Origin 2000) and MPICH (on Sun Cluster) are both portable implementations of the full MPI specification for a wide variety of parallel and distributed environments, including parallel computers, clusters of workstations, integrated distributed environments (computational grids) and shared-memory symmetric multiprocessors. Yet both MPT and MPICH provide unified system calls for communication between processes via the various communication media. Programming with MPI libraries makes it possible for developing portable and efficient parallel programs. The codes on Sun Cluster and on SGI Origin 2000 are exactly the same. Since MPI interface provides wide support for most commercially available parallel machines, and current systems that conform to the BSP computer model include networks of workstations, distributed memory processor arrays, and shared memory multiprocessors, algorithm HPT can easily be ported to any commercially available machines.

The program is about 450 lines, written in C, using MPI (Message Passing Interface) library [10] for interprocessor communication. Although the algorithm follows BSP style, while coding, due to high cost, MPI_Barrier was used only when necessary. This program uses MPI_Bcast for broadcasting. Some collective communication routines, such as MPI_Allreduce, MPI_Allgather, are also used for global communication. MPI_Scatter and MPI_Gather are used to distribute input data and collect results. (During the experiments, data collection time is not included when measuring parallel running time.) MPI_Send and MPI_Recv are used for message passing. MPI_Pack and MPI_Unpack are used for packing messages whose destination processors are the same. Timings were done by invoking MPI_WTime routine, which is intended to be a high-resolution, elapsed (or wall) clock. Throughout this section, HPT(N,p) is used to denote the parallel running time of algorithm HPT, where N and p stand for tournament size and number of processors respectively.

3.1 On Sun Ultra5 Cluster

Figures 1 and 2 show the average running times for input sizes ranging from 0.1 to 8 millions on up to 32 processors. When two processors are used, the speedup is about 1.75, approximately 87% of the optimal speedup. When the number of processors scales to 16, the speedup is about 9.4, which is about 58% of optimal speedup. (Here the p-processor speedup is measured based on HPT(N,1)/HPT(N,p)).

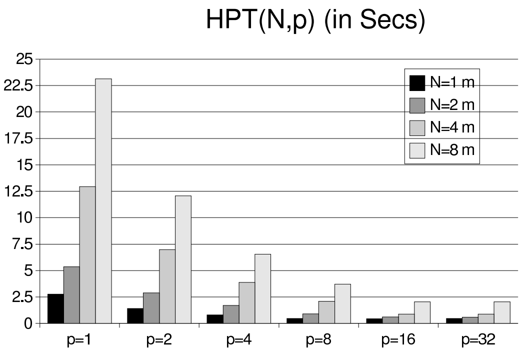

Figure 1: Sun Ultra5 Cluster running times (1)

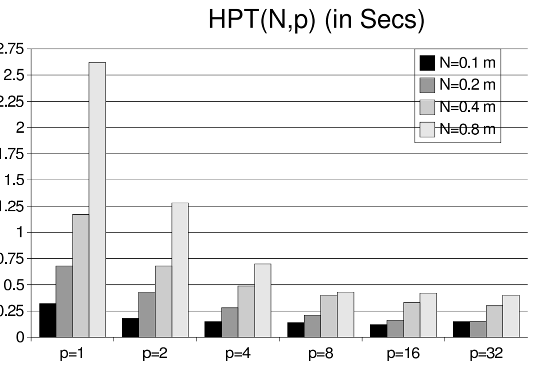

Figure 2: Sun Ultra5 Cluster running times (2)

3.2 On SGI Origin 2000

Algorithm HPT's behavior has also been investigated on UB CCR's SGI Origin 2000 machine, using up to 32 processors. The experiment was done also on input sizes from 0.1 to 8 millions. Each data point presented was obtained as the average of 5 test runs, each on a different randomly generated 2-d array.

Figures 3 and 4 depict the parallel running times when input sizes range from 0.1 to 8 millions.

The speedups achieved when the input size is above 1 million are almost linear as shown in Fig 3.

4. CONCLUSION

The authors observed that when the tournament size is above 1 million, the algorithm performance on Sun Cluster reaches about 75% of the performance on SGI Origin 2000. This shows that, with the presented communication-efficient approach, an important class of algorithms using the well-known divide-and-conquer technique, can in fact be implemented on low-cost cluster platforms in both computation and communication efficient manners.

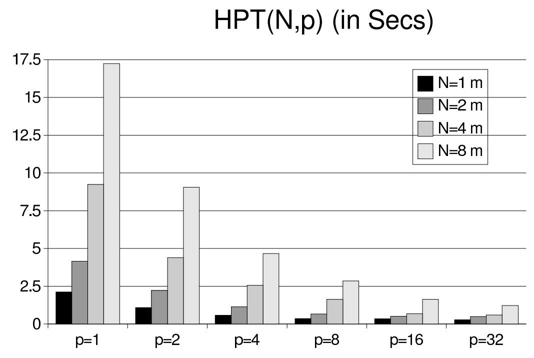

Figure 3: SGI Origin 2000 running times (1)

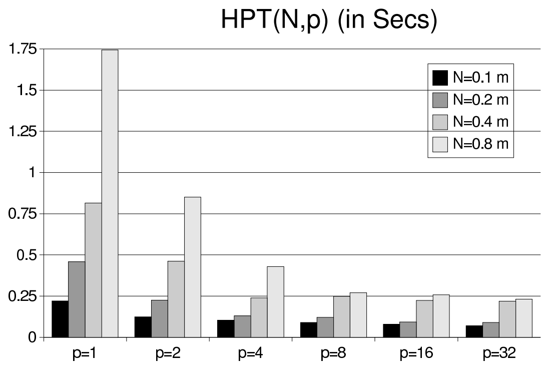

Figure 4: SGI Origin 2000 running times (2)

Long communication latency and the overhead of synchronization tend to be the two critical factors that greatly affect the speedup of a parallel algorithm on present multiprocessor architectures. Algorithm HPT is intended to minimize both. The parallel algorithm design is based on BSP programming model, which conforms to most of the commercially available parallel machines. Two portable MPI implementations, MPICH and MPT, are used in the experiments to justify the portability of the algorithm.

5. REFERENCES

[1] A. Aggarwal, B. Chazelle, L.J. Guibas C.O'Dunlaing, and C.K. Yap. Parallel Computational Geometry. *Algorithmica*, 3:293–327, 1988.

[2] S.G. Akl. Optimal Parallel Algorithms for Computing Convex Hulls and for Sorting. *Computing*, 33:1–11, 1984.

[3] B.A. Chalmers and S.G. Akl. Optimal Parallel Algorithms for a Transportation Problem. In *Proceedings of the Canadian Conference on Electrical and Computing Engineering*, 1991, 36.1.1-36.1.6.

[4] L.W. Beineke and R.S. Wilson. *Selected Topics in Graph Theory*. Academic Press, New York/London, 1978.

[5] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. In *Proc. 9th ACM Annual Computational Geometry*, 1993, 298-307.

[6] T.L. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall International, London, 1992.

[7] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[8] Michael T. Goodrich. Communication-Efficient Parallel Sorting. In *Proc. 28th ACM Symp. on Theory of Computing*, 1996, 247-256.

[9] S. Lakshmivarahan and S.K. Dhall. *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems.* McGraw-Hill, 1990.

[10] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[11] L. Redei. Ein Kombinatorischer Satz. *Acta Litt. Sci. Szeged*, 7:39–43, 1934.

[12] J.H. Reif. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, 1993.

[13] D. Soroker. Fast Parallel Algorithms for Finding Hamiltonian Paths and Cycles in a Tournament. *Journal of Algorithms*, 9:276–286, 1988.

[14] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.