Experimental Results of a Coarse-Grained Parallel Algorithm for Spanning Tree and Connected Components

Edson Norberto Cáceres, Henrique Mongelli, Christiane Nishibe
Universidade Federal de Mato Grosso do Sul,
Campo Grande - MS, Brazil
{*edson,mongelli,cnishibe*}*@dct.ufms.br*

Siang Wun Song
Universidade Federal do ABC,
Santo André - SP, Brazil
and Universidade de São Paulo,
São Paulo - SP, Brazil
song@ime.usp.br

POSTER PAPER

ABSTRACT

Dehne et al. present a BSP/CGM algorithm for computing a spanning tree and the connected components of a graph, that requires $O(\log p)$ communication rounds, where p is the number of processors. It requires the solution of the Euler tour problem which in turn is based on the solution of the list ranking problem. In this paper we present experimental results of a parallel algorithm that does not depend on the solution of the Euler tour or the list ranking problem. The proposed algorithm has the practical advantage of avoiding the list ranking computation and is based on the integer sorting algorithm which can be implemented efficiently on the BSP/CGM model. We implemented the proposed algorithm on a Beowulf cluster and on a grid running the InteGrade middleware. We obtained encouraging albeit modest speedup on a small Beowulf cluster and expect good speedups on the grid for larger size graphs and clusters.

KEYWORDS: Parallel algorithm, spanning tree, graph problems.

1. INTRODUCTION

Computing a spanning tree of a given graph is a basic problem. Obtention of an efficient algorithm for such basic problems is very important since they may appear as subproblems of many other problems. Efficient solution of basic subproblems constitutes the main emphasis of the parallel algorithm synthesis method proposed by Reif [13].

The sequential algorithms use depth-first or breadth-first search to solve these problems efficiently. As we will show in the next sections, the parallel solutions for these problems, however, do not use these search methods because they are not easy to parallelize [12].

Based on the PRAM algorithm of Shiloach and Vishkin [15], Dehne *et al* [7] propose a BSP/CGM algorithm that computes a spanning tree, using $O(\log\ p)$ communication rounds and $O(m\ +\ n/p)$ local computation time in each round, where p is the number of processors. It requires the solution of the Euler tour problem which in turn is based on the solution of the list ranking problem. In [2] we present a brief description of a parallel algorithm to compute a spanning tree and connected components of a given graph, that is not based on the solution of the Euler tour or the list ranking problem. The proposed algorithm still requires $O(\log p)$ communication rounds and has the practical advantage of avoiding the list ranking computation which, in spite of presenting an $O(\log p)$ communication rounds complexity, has been shown to require large constants in practical implementations. In this paper we give a complete presentation of the parallel algorithm, with a detailed discussion of the correctness and complexity of the algorithm, together with

experimental results.

The proposed algorithm is based on the integer sorting algorithm which can be implemented efficiently on the BSP/CGM model [4]. We use a special spanning forest, called *strut* [3], of a bipartite graph, to compute a spanning tree and the connected components of a given graph. We first show how we can transform any graph into a bipartite graph. To solve the spanning tree problem we assume a bipartite graph as input. If the input bipartite graph is not connected, the algorithm will compute a spanning tree for each connected component.

2. COARSE-GRAINED MULTICOMPUTER

We consider a simpler version of the Bulk Synchronous Parallel (BSP) model [16], called the *Coarse-Grained Multicomputer* (CGM) model [6]. It uses two parameters: the input size N and the number of processors p. A CGM consists of a set of p processors each with its local memory. A CGM algorithm consists of alternating local computation and global communication rounds separated by a barrier synchronization. (See Figure 1.) In a computation round, each processor processes its data locally. In each communication round each processor sends $O(N/p)$ data and receives $O(N/p)$ data. In the CGM model, the communication cost is modeled by the number of communication rounds. The goal is to design a parallel algorithm that minimizes the number of rounds required.

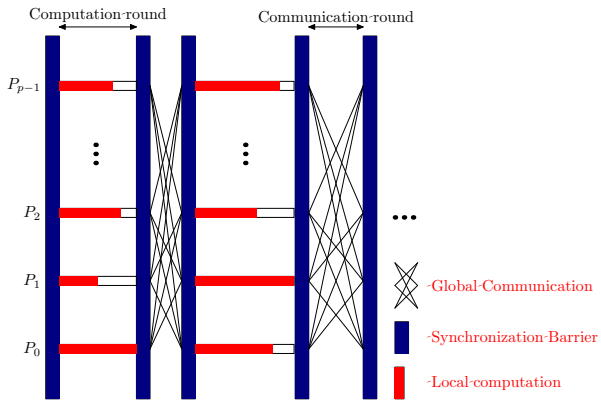

Figure 1. Coarse-Grained Multicomputer CGM Parallel Computation Model.

3. PREVIOUS PARALLEL ALGORITHMS

We define the spanning tree problem and discuss previous parallel algorithms. Consider a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. A spanning tree

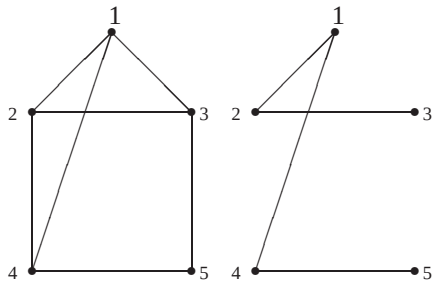

Figure 2. A Graph G and its Spanning Tree T.

$T = (V', E')$ is a subgraph of G that is a tree and contains all the vertices of G, i.e. $V' = V$ and $E' \subset E$.

Figure 2 (a) shows a graph G and Figure 2 (b) a spanning tree T.

Efficient sequential spanning tree algorithms are based on depth-first or breadth-first search. The depth-first search, however, does not present an efficient parallel implementation [12]. The parallel algorithms to obtain a spanning tree are based instead on a PRAM CRCW algorithms by Hirschberg et al. [10] that computes the connected components of a graph with n vertices in $O(\log^2 n)$ time and using n^2 processors. Super-vertices of the graph are successively combined into larger super-vertices. This approach gives rise to several other PRAM algorithms to compute a spanning tree of a given graph [11]. The most efficient of these algorithms is a PRAM CRCW algorithm of $O(\log n)$ time with $O((m+n)\alpha(m,n))/\log n$ processors, where $\alpha(m,n)$ is the inverse of the Ackermann's function [11]. Shiloach and Vishkin [15] present an algorithm that takes $O(\log n)$ time and uses $m + n$ processors. Halperin and Zwick [9] present a randomized PRAM EREW algorithm that finds the connected components of a graph in $O(\log n)$ time, using $O((m + n)/\log n)$ processors.

Bader and Cong [1] implemented the algorithms of Shiloach and Vishkin [15] and Hirschberg *et al* [10] and designed a new randomized algorithm to compute a spanning tree, using symmetric multiprocessors. Cong and Xue [5] present experimental results of an asynchronous spanning tree algorithm on a cluster of SMPs. Setia et al. [14] use a heuristic approach to obtain a parallel minimum spanning tree algorithm and present experimental results on a cluster of SMPs. Notice the number of recent papers that, in addition to theoretical complexity results, emphasize on implementation and experimental results. In this sense our paper aims to contribute on the important issue of comparing the performance of parallel algorithms implemented on a Beowulf cluster and on a grid running the InteGrade middleware.

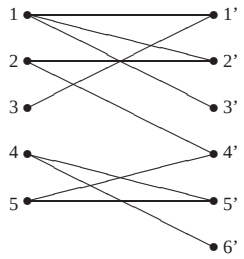

Figure 3. Bipartite Graph $H = (V_1, V_2, E)$

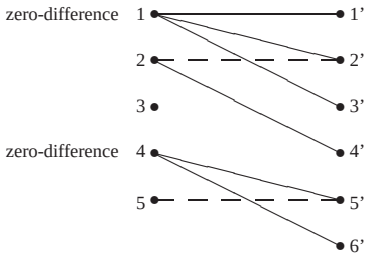

Figure 4. A Strut ST Represented by Solid Lines.

4. DEFINITIONS AND MAIN IDEA

Without loss of generality, we give a parallel spanning tree algorithm for a given input bipartite graph. It is easy to transform any graph into a bipartite graph (see [2]).

Consider a bipartite graph $H = (V_1, V_2, E)$ with vertex sets V_1 and V_2 and edge set E where each edge joins one vertex of V_1 and one vertex in V_2. If v is a vertex of a subgraph H' of H, then $d_{H'}(v)$ denotes the degree of v in H'. Let the vertex set $V_1 = \{u_1, u_2, \cdots, u_{n_1}\}$ and the vertex set $V_2 = \{v_1, v_2, \cdots, v_{n_2}\}$.

We define a *strut* ST *in* V_1 as a spanning forest of H such that each $v_i \in V_2$ is incident in ST with exactly one edge of E, and (u_j, v_i) is an edge of ST implies (u_k, v_i) is not an edge of H, for any $u_k \in V_1$, $k < j$. To define a *strut in* V_2, the roles for the sets V_1 and V_2 in the above definition are exchanged.

A vertex $u \in V_1$ where $d_H(u) \neq 0$ is called *zero-difference* in ST if $d_H(u) - d_{ST}(u) = 0$. Otherwise, the vertex is referred to as *non-zero-difference*.

Consider the input bipartite graph $H = (V_1, V_2, E)$ of Figure 3. We have

$V_1 = \{1, 2, 3, 4, 5\}$, $V_2 = \{1', 2', 3', 4', 5', 6'\}$ and
$E = \{(1, 1')(1, 2')(1, 3')(2, 2')(2, 4')(3, 1')(4, 5')$
$(4, 6')(5, 4')(5, 5')\}$.

We can first compute a spanning forest for $H = (V_1, V_2, E)$, obtained by determining a strut ST in H. We obtain a strut (see Figure 4), which is represented by solid lines while the remaining edges are represented by dash lines.

Now compute the zero-difference vertices in V_1. Consider vertex 1. All the (three) edges in H incident with this vertex is also in ST. Thus $d_H(1) - d_{ST}(1) = 0$ and vertex 1 is zero-difference. Likewise vertex 4 is also zero-difference. Vertex 2 has $d_H(2) - d_{ST}(2) = 2 - 1 = 1$ and thus is not zero-difference.

In the example we have two zero-difference vertices. If, however, we have only *one* zero-difference vertex, then the problem is easily solved by adding to ST one arbitrary edge of $H - ST$ incident to each non-zero-difference vertex of ST.

In case there are two or more zero-difference vertices we can do the following. For each zero-difference vertex $u \in V_1$ compact all the vertices $v_i \in V_2$ incident with u by compressing all the vertices v_i onto the smallest of the v_i.

This can be done repeatedly until only one zero-difference vertex remains.

5. SPANNING TREE ALGORITHM

Consider a bipartite graph $H(V_1, V_2, E)$ where $|V_1| = n_1$, $|V_2| = n_2$ and $|E| = m$. Consider p processors with local memory of size $O(m/p)$ or $O((n_1 + n_2)/p)$. We present a CGM algorithm (see Algorithm 1) to compute a spanning tree of a bipartite graph. It takes $O(\log p)$ communication rounds and is based essentially on an integer sorting algorithm. Thus it differs from a previous algorithm [7] that requires Euler tour and list ranking. Though the presented algorithm still presents the same complexity on the number of communication rounds, we expect better experimental results since it depends on the sorting algorithm that can be implemented efficiently on the CGM model.
The following note can be useful in practice. To take into consideration the possible different computing speeds of each of the p processors, the input edges are distributed in the p processors in amount proportional to the relative speed of each processor. Also in the Bucket-Sort of line 3, we use buckets with sizes proportional to the relative speeds of each processor. This makes the algorithm adequate for use in a heterogeneous network of different kinds of computers.

It is instructive to consider the example of Figure 3 to illustrate the several steps of Algorithm 1.

As storage structures, consider the vectors $EDGE$ and $EDGE'$, each containing m elements. Each element is an edge represented by (u, v), with $u \in V_1$ and $v \in V_2$. The

Algorithm 1 CGM Algorithm for Spanning Tree

Input: A bipartite graph $H(V_1, V_2, E)$ where $V_1 = \{u_1, \ldots, u_{n_1}\}$, $V_2 = \{v_1, \ldots, v_{n_2}\}$ and $|E| = m$. An edge (u_i, v_i) of E has a vertex u_i in V_1 and a vertex v_i in V_2. The m edges are equally distributed among the p processors at random.

Output: A spanning tree of H. The edges of the spanning tree are stored in FINAL-SPANNING-TREE.

Phase 0 - preprocessing:

1: Put in FINAL-SPANNING-TREE all edges $(u, v) \in E$ such that degree(u)=1. The resulting edge set is denoted E'.

Phase I:

1: Initialize $\bar{V_1} := V_1$ and $\bar{V_2} := V_2$ and $\bar{E} := E'$.
2: **for** $\log p$ times **do**
3: Sort the edges (u, v) of $\bar{E}$ by v and then by u.
4: **for** each v_i of V_2 **do**
5: Choose the smallest vertex u_j among all edges (u, v_i) and mark the edge (u_j, v_i). Let ST be the set of the marked edges.
6: **end for**
7: Compute the degree of each vertex $u \in \bar{V_1}$ in $H(\bar{V_1}, \bar{V_2}, \bar{E})$.
8: Compute the degree of each vertex $u \in \bar{V_1}$ in $H_{ST}(\bar{V_1}, \bar{V_2}, ST)$.
9: Using the degrees computed in the previous steps compute the number of zero-difference vertices. Put in FINAL-SPANNING-TREE all edges incident with zero-difference vertices.
10: **if** number of zero-difference vertices = 1 **then**
11: the algorithm terminates
12: **end if**
13: Compact the graph to produce the compacted graph $H(\bar{V_1}, \bar{V_2}, \bar{E})$: Step 1: for each zero-difference vertex $u \in V_1$ denote the vertices incident with u by $v_i \in V_2$. Remove all edges (u, v_i) and put them in FINAL-SPANNING-TREE. Then merge all the vertices v_i onto the smallest of the v_i. Step 2: Rename the labels of the vertices of V_2 of H. Step 3: Remove all repeated edges. Step 4: If there is an edge (u, v) in H with degree(u)=1, remove it and put it in FINAL-SPANNING-TREE.
14: **end for**

Phase II:

1: Compute a spanning forest with the edges that do not belong to ST and removing those with degree($\bar{u}$)=1 where $\bar{u} \in \bar{V_1}$.
2: **for** i:=0 to $\log p$ **do**
3: Processor $2i + 1$ sends its spanning forest to Processor $2i$.
4: Processor $2i$ computes a new spanning forest.
5: **end for**

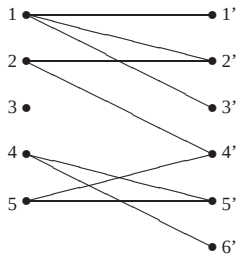

Figure 5. The Bipartite Graph after Preprocessing

edges of E are stored in vector $EDGE$. In our example, the vector $EDGE$ is

$$(1,1')(1,2')(1,3')(2,2')(2,4')(3,1')(4,5')(4,6')$$
$$(5,4')(5,5')$$

The preprocessing phase is quite straightforward. All edges with degree 1 are necessarily in the spanning tree. Thus they can be removed from the input edges and put in the result FINAL-SPANNING-TREE. The removal of such edges is important because otherwise they would give rise to many zero-difference vertices. Figure 5 shows the resulting graph after preprocessing.

Consider now Phase I. Initially we make a copy of $EDGE$ in $EDGE'$.

Lines 3 to 6 of Algorithm 1 obtain a strut ST. First we sort the edges in $EDGE'$ lexicographically in the following way. Given two edges (i, j) and (k, l) then $(i, j) < (k, l)$ when $j < l$ or $((j = l)$ and $(i < k))$. This is done in line 3 of Algorithm 1. Vector $EDGE'$ contains the sorted edges:

$$(1,1')(1,2')(2,2')(1,3')(2,4')(5,4')(4,5')$$
$$(5,5')(4,6')$$

Lines 4 to 6 find a strut ST in V_1. It is represented by the *marked* edges (solid lines of Figure 4). Vector $EDGE'$ represents ST and for our example, we have:

$$(1,1')(1,2')(1,3')(2,4')(4,5')(4,6')$$

A strut ST in V_1 determines a spanning forest of H.

Lines 7 to 9 find the zero-difference and non-zero-difference vertices of the strut ST. We determine the degrees of each of the vertices in V_1 and store in D_H. In our example D_H = $(3, 2, -, 2, 2)$.

Determine now which vertices of V_1 are zero-difference. For this, determine the degree of each of the vertices of V_1

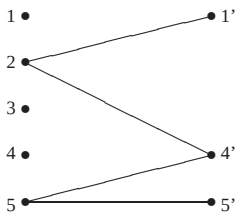

Figure 6. The Compacted Graph.

in $EDGE'$ and store in D_{ST}. Again for our example, D_{ST} = $(3, 1, -, 2, 0)$.

Thus the zero-difference vertices are vertices $\{1, 4\}$ and the non-zero-difference vertices are vertices $\{2, 3, 5\}$.

Line 13 produces a compacted graph. For the two zero-difference vertices 1 and 4, step 1 removes edges $(1, 1'), (1, 2'), (1, 3')$, and also $(4, 5'), (4, 6')$, puts them in FINAL-SPANNING-TREE. Then vertices $\{1', 2', 3'\}$ are merged onto vertex $1'$, and vertices $\{5', 6'\}$ are merged onto vertex $5'$, Step 2 renames vertices $2'$ and $3'$ as $1'$ and renames $6'$ as $5'$. The compacted graph is shown in Figure 6.

6. DISCUSSION OF THE ALGORITHM

Lemma 1 *Let $H = (V_1, V_2, E)$ be a bipartite graph and let S be a strut in V_1. Let H' be the graph obtained from H by adding to S exactly one edge from $E - S$ that is incident to each non zero-difference vertex of V_1. Then H' is acyclic. Moreover, if V_1 contains exactly one zero-difference vertex then H' is a spanning tree of H.*

Proof.
We can see that S is a set of stars whose centers are vertices in V_1. When we add exactly one edge from $E-S$ incident to each non zero-difference vertex of V_1, the star whose center v_j is a non-zero-difference vertex will be connected to at most a different star of center v_i. Then, the resultant graph H' is acyclic.

By the definition of strut, the vertex degree of each vertex in V_2 is exactly one, so there will be exactly $|V_2|$ edges in S. As V_1 has exactly one zero-difference vertex, the number of edges that can be added is $|V_1| - |V_2| - 1$ and H' is a spanning tree of H. □

Theorem 1 *The number of zero-difference vertices in $\bar{V_1}$ after step 13 is at least divided by 2 in each iteration.*

Proof.

Let $\bar{V_1}'$ and $\bar{V_2}'$ be the partitions before step 13 and let $\bar{V_1}$ and $\bar{V_2}$ be the partitions after step 13. Let k be the number of zero-difference vertices in V_1. After step 13, the number of vertices in $\bar{V_2}$ is also k since after compaction, for each zero-difference vertex in $\bar{V_1}$, only the vertex in $\bar{V_2}'$ with smallest label is kept in $\bar{V_2}$. Since all the vertices in $\bar{V_1}$ have degree greater than 1, the number of zero-difference vertices in $\bar{V_1}$ is $|\bar{V_2}|$, e.g., $k/2$. □

Theorem 2 *Algorithm 1 computes the spanning tree of $H = (V_1, V_2, E)$ with $O(\log p)$ communication rounds and $O((m + n)/p)$ local computation time in each round.*

Proof. Phase I of the algorithm can be executed in $O(\log p)$ communication rounds by using a sorting algorithm that takes constant number of communication rounds (e.g. [4]) in each iteration.

The m input edges are distributed equally among the p processors. Each processor thus contains edges (u, v) where $u, v \in V_1 \cup V_2$. The size of $V_1 \cup V_2$ is $n_1 + n_2$ which can be larger than the local memory of each individual processor. It can be shown that after $\log p$ iterations, the number of remaining vertices of $\bar{V_1} \cup \bar{V_2}$ is bounded by $O((n_1 + n_2)/p)$. Thus at the end of phase I, since each processor contains edges (u, v) where $u, v \in \bar{V_1} \cup \bar{V_2}$, all the remaining vertices are stored in every processor. At this time we proceed to phase II.

Phase II consists of $O(\log p)$ communication round during which the individual spanning trees in the processors are combined. □

Note that this algorithm can be used to determine the connected components of a graph. In each iteration, the spanning tree algorithm determines each of the sublists of $EDGE'$ formed by edges $(u, v), u = u_i$ that forms a tree, labeled by $EDGE'_{u_i}$. At the end of the algorithm, we can represent each tree with the smallest vertex. Each of the different vertices represent a connected component of the graph.

9. EXPERIMENTAL RESULTS

Table 1. Number of Vertices and Edges of the Bipartite Graph.

Instance	$\|V_1\| = \|V_2\|$	$\|E\|$
G_1	1024	262.144
G_2	2048	1.048.576
G_3	4096	4.194.304
G_4	8192	8.388.608

Table 1 shows the number of vertices and edges for each graph used as input graphs in the experiments. As the number of edges is strictly greater than the number of vertices, we assume the graph possesses the same number of vertices in each of partitions, i.e. $|V_1| = |V_2|$. For the tests, we generated the input bipartite graphs at random, as follows. To generate a bipartite graph with $|V_1| = |V_2|$ vertices and $|E|$ edges, we choose one vertex from V_1 and one from V_2 at random and put an edge connecting both vertices. This is done until we reach the desired $|E|$ edges.

9.1. The Platform Used in the Experiments

We carried out the experiments on a Beowulf cluster and on a cluster running the InteGrade middleware [8]. The InteGrade middleware allows the implementation of a computational grid with non-dedicated computing resources, by using the idle capacity of existing computer laboratories. We compare the execution of the proposed parallel algorithm on a cluster with only MPI support and on a grid using the InteGrade middleware and MPI.

The cluster is composed of AMD 1.6 GHz and P4 2.6 GHz processors with 1-2 Gbytes of memory. The communication is through a Gigabit Ethernet switch. The MPI used is the LAM/MPI 7.1.2. The InteGrade middleware used is the version 0.4.

Table 2. Running Times (in Seconds) on the Beowulf Cluster.

p	G_1	G_2	G_3	G_4
1	0.217107	0.950689	5.240836	11.789449
2	0.235769	1.161584	4.836589	10.615158
4	0.230011	1.133226	4.350016	9.687900
8	0.240889	1.117800	3.360593	8.812110

9.2. Resutls Obtained

Table 3. Running Times (in Seconds) on the Cluster Running InteGrade Middleware.

p	G_1	G_2	G_3	G_4
1	0.251695	1.099503	5.090014	11.643704
2	1.677849	5.432931	18.379176	29.222423
4	2.361967	8.001241	25.201421	37.953199
8	2.600342	10.314014	29.228178	43.632104

Table 2 presents the execution times of the parallel algorithm on the Beowulf cluster for several graph instances. In instance G_1, as the number of edges is small, the execution in one single processor is the fastest. In instance G_2, the execution with one processor is also the fastest. However, from $p = 2$ on, the execution time diminishes as the number of processors is increased. In instances G_3 and G_4, the running times decrease as we increase the number of processors.

Table 3 shows the execution times of the spanning tree algorithm on the cluster running the InteGrade middleware. Here we notice a different behavior. Due to the overhead of the middleware, as we increase the number of processors, the running times increase, which is not a desirable behavior in parallel computation.

On the Beowulf cluster, we notice a modest speedup for larger instances. The compaction of the graph in phase I requires the communication of data among processors, in order to redistribute the compacted edges for the next iteration of the algorithm. The amount and size of the messages exchanged among the processors contribute negatively on the performance of the algorithm on the cluster with the InteGrade middleware. Finally, Figure 7 compares the results of tables 2 and 3.

10. CONCLUSION

We have presented a CGM algorithm for computing a spanning tree of a graph and its connected components. It takes $O(\log p)$ communication rounds and is based on an integer sorting algorithm. Thus it differs from a previous algorithm [7] that requires the computation of Euler tour and list ranking. Though the presented algorithm still presents the same complexity on the number of communication rounds, we expect it to give good experimental results since it depends on the sorting algorithm that has efficient CGM implementations.

The results on the grid running the InteGrade middleware, leave much to be desired. However, albeit with modest speedup, the experimental results on the Beowulf cluster are encouraging, specially if we use larger clusters.

ACKNOWLEDGMENTS

We thank the referees for their comments. This work has been partially supported by CNPq Proc. No. 55.0895/07-8, 30.1652/09-0, 62.0171/06-5, FUNDECT 41/100.115/2006, and CAPES PVNS edital 20/2009.

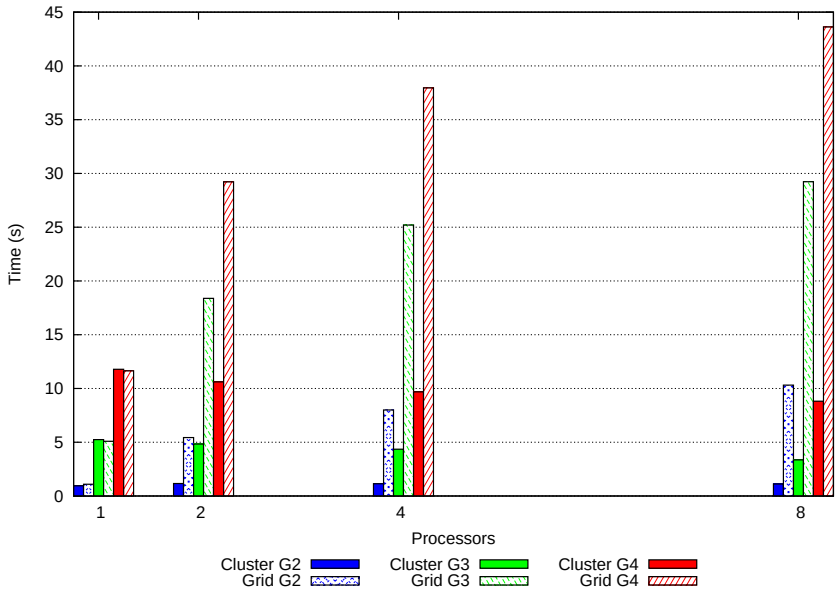

Figure 7. Comparing Experimental Results on a Beowulf and on the InteGrade Grid.

REFERENCES

[1] D.A. Bader and G. Cong, "A fast parallel spanning tree algorithm for symmetric multiprocessors (SMPs)," *J. Parallel Distrib. Comput.*, Vol. 65, No. 9, pp. 994–1006, 2005.

[2] E.N. Cáceres, F. Dehne, H. Mongelli, S.W. Song, and J.L. Szwarcfiter, "A coarse-grained parallel algorithm for spanning tree and connected components," Euro-Par, Pisa, Italy, pp. 828–831, 2004.

[3] E.N. Cáceres, N. Deo, S. Sastry, and J.L. Szwarcfiter, "On finding euler tours in parallel," *Parallel Processing Letters*, Vol. 3, No. 3, pp. 223–231, 1993.

[4] A. Chan and F. Dehne, "A note on coarse grained parallel integer sorting," *Parallel Processing Letters*, Vol. 9, No. 4, pp. 533–538, 1999.

[5] G. Cong and H. Xue, "A scalable, synchronous spanning tree algorithm on a cluster of SMPs," 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1-6, 2008.

[6] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable parallel geometric algorithms for coarse grained multicomputers," Proc. ACM 9th Annual Computational Geometry, San Diego, CA, pp. 298–307, 1993.

[7] F. Dehne, A. Ferreira, E. Cáceres, S.W. Song, and A. Roncato, "Efficient parallel graph algorithms for coarse grained multicomputers and BSP," *Algorithmica*, Vol. 33, No. 2, pp. 183–200, 2002.

[8] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G.C. Bezerra, "InteGrade: object-oriented grid middleware leveraging the idle computing power of desktop machines," *Concurrency and Computation: Practice and Experience*, Vol. 16, No. 5, pp. 449–459, 2004.

[9] S. Halperin and U. Zwick, "An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM (extend abstract,)" SPAA '94: Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures, Cape May, NJ, pp. 1–10. 1994.

[10] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate, "Computing connected components on parallel computers," *Comm. ACM*, Vol. 22, pp. 461–464, 1979.

[11] R.M. Karp and V. Ramachandran, HANDBOOK OF THEORETICAL COMPUTER SCIENCE - J. van Leeuwen (ed.), volume A, chapter 17, pp. 869–941, Elsevier, Amsterdam, Netherlands/MIT Press, Cambridge, MA, 1990.

[12] J.H. Reif, "Depth-first search is inherently sequential," *Inf. Process. Lett.*, Vol. 20, No. 5, pp. 229–234, 1985.

[13] J.H. Reif, SYNTHESIS OF PARALLEL ALGORITHMS, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1993.

[14] R. Setia, A. Nedunchezhian, S. Balachandran, "A new parallel algorithm for minimum spanning tree problem," Proc. International Conference on High Performance Computing (HiPC), pp. 1-5, 2009.

[15] Y. Shiloach and U. Vishkin, "An $o(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, Vol.3, pp. 57–63, 1982.

[16] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, Vol. 33, No. 8, pp. 103–111, 1990.