Parallel Implementation of Borůvka's Minimum Spanning Tree Algorithm *

Sun Chung   Anne Condon

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706 USA

Abstract

We study parallel algorithms for the minimum spanning tree problem, based on the sequential algorithm of Borůvka. The target architectures for our algorithm are asynchronous, distributed-memory machines.

Analysis of our parallel algorithm, on a simple model that is reminiscent of the LogP model, shows that in principle a speedup proportional to the number of processors can be achieved, but that communication costs can be significant. To reduce these costs, we develop a new randomized linear work pointer jumping scheme that performs better than previous linear work algorithms. We also consider empirically the effects of data imbalance on the running time. For the graphs used in our experiments, load balancing schemes result in little improvement in running times.

Our implementations on sparse graphs with 64,000 vertices on Thinking Machine's CM-5 achieve a speedup factor of about 4 on 16 processors. On this environment, packaging of messages turns out to be the most effective way to reduce communication costs.

1. Introduction

A dominant emerging parallel architecture consists of a collection of fast processors, connected by a robust communication network [7, 9, 12, 13, 14]. Properties of this type of architecture include a distributed memory, partitioned among processors whose interprocessor communication cost is rather high compared with the computation cost.

For this architecture, we describe our experience with design and implementation of parallel algorithms for the minimum spanning tree (MST) problem: Given a connected,

undirected graph G with n vertices and m weighted edges, find a spanning tree of minimum weight. We are interested in the case that the number of processors p is much less than the size of the graph, and the graph is distributed among the processors. Our algorithms are based on the classical sequential algorithm of Borůvka [4]. Using a simple performance model, together with measurements of implementations on Thinking Machine's CM-5, we analyze and compare alternative implementations.

Briefly, our conclusions are as follows. In principle, a speedup proportional to p can be achieved, but a simple analysis points to two primary sources of slowdown, in addition to the penalty for communication (Section 2). One arises in the use of pointer jumping, which is less efficient than the sequential technique of depth first search, and is very communication intensive. The other is due to imbalance in the distribution of data (and hence work) among processors. To address the first problem, we develop a new randomized linear work pointer jumping scheme that performs better than previous linear work algorithms on lists (Section 3). Our algorithm is similar to the list ranking algorithms of Vishkin [15] and Cole and Vishkin [6], and shows how an idea developed for the PRAM model can be adapted to work effectively in practice. To address the second problem, since a precise analysis of the degree of imbalance of work seems difficult, we consider empirically the effects of data imbalance (Section 4). For the graphs used in our experiments, load balancing schemes result in little improvement in running times.

First, we describe Borůvka's algorithm and our parallel model.

Borůvka's algorithm: This algorithm, also known as Sollin's algorithm, constructs a spanning tree in iterations composed of the following steps (organized here to correspond to the phases of our parallel implementation).

Step 1 (choose lightest) : Each vertex selects the edge with the lightest weight incident on it. Each of the connected components

thus created has one cycle of size two between two vertices that each selects the same edge. Of this pair, the one with the smaller number is designated as the root of the component and the cycle is removed. The component is then a tree.

Step 2 (find root) : Each vertex identifies the root of the tree to which it belongs.

Step 3 (rename vertices) : In the edge lists, each vertex is renamed with the name of the root of the component to which it belongs.

Step 4 (merge edge lists) : Edge lists which belong to the same component are merged into the edge list of the root. In other words, each connected component shrinks into a single vertex.

Step 5 (clean up) : Now the edge lists may have self loops and multiple edges. All self loops are removed. Multiple edges are removed such that only the lightest edge remains between a pair of vertices.

The graph remaining after the ith iteration is the input to the $(i + 1)$st iteration, unless it has just one vertex, in which case the algorithm halts. The output spanning tree is the union of the set of edges selected in step 1, taken over all iterations. Using standard techniques (see [8]) the algorithm can be implemented so that an iteration in which the graph has n vertices and m edges takes $O(n + m)$ sequential time. Furthermore, the number of vertices of the graph at the $(i + 1)$st iteration is at most half of the number of vertices at the ith iteration. Hence, the number of iterations is at most $\log_2 n$, yielding a total running time of $O(m \log n)$.

Parallel model: We assume a distributed memory model, in which processors communicate using messages. However, our model could easily be adapted to other distributed memory machines such as the Cray T3D and to shared memory abstractions that are built on top of distributed memory machines, such as the Split-C language, where the distinction between local versus non-local data is retained. The machine parameters that we will use are: p, the number of processors, t_s, the time for initiating transmission of a message, and t_w, the transmission time per word. Typically, we can expect that t_s can be much greater than t_w; on the CM-5 message passing system using CMMD operations, $t_s \approx 80-300\mu s$, and $t_w \approx 1-3\mu s$. There is synchronization cost involved in each synchronized step of a parallel algorithm, but we will ignore it in our model because in our experiments, the cost is minimal in the algorithms that we study.

Background and related work: For a historical survey of the classical MST algorithms and their variants, see [8] and [11]. Knuth [10] and Moret and Shapiro [11] present empirical assessments of sequential MST algorithms. The work of Barr et al. [3] is the only empirical study of parallel implementation of MST algorithms which we found in the literature that is related to ours. For further discussion of related work on parallel models and graph algorithms, the reader is referred to the full version of this paper [5].

2. A parallel Borůvka's algorithm

We now describe a parallel version of each step of Borůvka's algorithm.

Step 1 (choose lightest): The edge list of each vertex is searched to find the minimum weight edge from that vertex.

Step 2 (find root): Each vertex finds the root of the tree to which it belongs using the well known pointer jumping algorithm. The input R to the algorithm is the set of root vertices, and the input S is the set of non-root vertices.

> *Simple-Pointer-Jumping-Algorithm(S,R)*
> **repeat** until every vertex in S points to a vertex in R
> **for** each vertex i that does not point to
> a vertex in R **do**
> perform a pointer jump on i

Step 3 (rename vertices): Each processor finds the new name of all vertices listed in its edge lists.

Step 4 (merge): The edges of all vertices in a component are sent to the processor that has the edge list of the root. The edge lists are then merged by that processor.

Step 5 (clean up): Each processor executes the sequential algorithm on its own edge lists.

In our implementation of the pointer jumping algorithm of step 2, processors synchronize at each iteration of the repeat loop.

2.1. Running time

Consider the parallel running time of the first iteration, in which there are n vertices and m edges. Note that there is no communication needed in steps 1 and 5. The amount of work done by a processor in steps 1 and 3 is linear in the number of edges at that processor at the start of the iteration. Similarly, the amount of work done by a processor in steps 4 and 5 is linear in the number of edges at that processor, after the edge lists are moved in step 3.

If we make some simple assumptions about the graph and its initial distribution, we can show that the expected parallel time needed to complete steps 1,3,4 and 5 of the first iteration is $O((t_s + t_w)m/p)$. Suppose also that any vertex (and its edge list) is initially equally likely to be at any processor. Then, we can expect that the maximum number of vertices at a processor is within a constant factor of n/p. This follows from a "balls and bins" analysis, where the processors are the bins and the vertices are the balls. It is well known that if n balls are thrown randomly into p bins, then the expected maximum number of balls per bin is close to the average if $n = \Omega(p \log p)$. If, furthermore, the degrees of the vertices are small (say, a constant independent of n), and roughly equal, then the edges are split evenly among the processors. Also, we can expect that the communication costs

are split fairly evenly among the processors in steps 3 and 4. In step 3, for example, each processor sends one message to query the new name of each vertex occurring in its edge lists. Since the degree of the vertices is constant, the number of distinct vertices arising in the edge lists of a processor is linear in the number of edges. Therefore, $\Theta(m/p)$ queries are needed per processor.

Using a similar argument, we can expect that the communication workload in step 2 is fairly evenly distributed and the expected time for step 2 is $O((t_s + t_w)(n \log n)/p)$ (for details, see [5]).

Extending even this heuristic analysis to further iterations is difficult, however. Only if the iterations for which the data is badly distributed contribute very little to the running time can we hope to prove that the parallel running time is $O((t_s + t_w)(m \log n)/p)$, that is, that a speedup factor of $\Omega(p)$ is obtained. One thing we can say is that we can expect the distribution of vertices to remain fairly evenly balanced in later iterations, as long as the number of vertices is still large. This is because the location of vertices at the ith iteration is the same as their location in the initial iteration. However, the edge lists are typically growing in length as the iterations progress, and there is more variance in the distribution of the lengths of the edge lists. We did some empirical measurements of the edge distribution over time, and present these in Section 4.3.

3. Parallel algorithms for step 2

The parallel running time of step 2 that uses pointer jumping ($O((t_s + t_w)(n \log n)/p)$) is significantly slower than that of the sequential algorithm which uses linear time depth first search. We now consider two new algorithms for step 2, which aim at reducing the slowdown due to the t_s and $\log n$ factors respectively.

3.1. The packaging algorithm

In this algorithm, at each synchronized substep of step 2, all messages which are transmitted from a processor to another processor are sent in a single package. Thus, each processor sends at most $p - 1$ packages in a synchronized substep, regardless of how many individual pointer updates are performed. Therefore, the cost t_s is charged at most p times per synchronized substep, whereas in the simple pointer jumping algorithm it is charged $\Theta(n/p)$ times (assuming balanced communication). Therefore, if $p << n/p$, we expect that the packaging scheme will be faster than the simple pointer jumping scheme. For larger p, however, the advantage of packaging may be lost.

3.2. A new pointer jumping algorithm

Both deterministic and randomized list ranking algorithms that require only linear work are well known [2, 6, 15]. However, they are not well suited to our application because they have large hidden constants and require that the input data, which are rooted trees, be "linearized."

We developed a new pointer jumping scheme, which we call the *supervertex algorithm*. This randomized scheme can be applied to trees as well as lists and requires only expected linear work.

Roughly, in our algorithm, each component is processed as follows. A randomly chosen subset, SV, of the vertices called supervertices are selected. Each vertex in $S-SV$ performs the simple pointer jumping algorithm until it points to a supervertex. At this point, all vertices but the supervertices drop out and the supervertices repeat the same algorithm recursively (with each vertex again randomly deciding whether to be a supervertex in the next iteration). Once all supervertices are pointing to the root, the remaining vertices update their pointers in one step so that they too point to the root. Figure 1 illustrates the execution of this algorithm on a list.
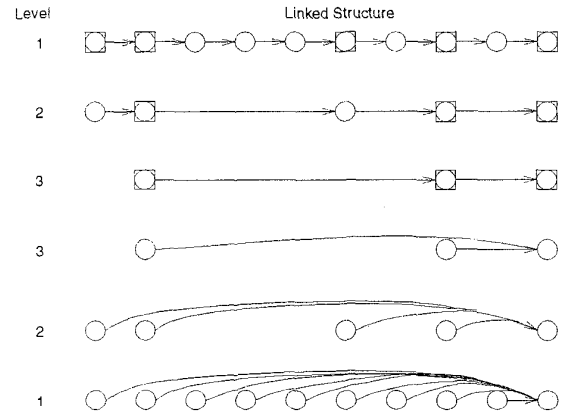

Figure 1. Execution of the supervertex algorithm on a list. The first three rows show the linked structure at the start of the three recursive calls. Vertices in squares are chosen to be supervertices at each of these iterations. The last three rows show the vertices that point to the root at the end of each recursive call, starting from the last (third) level of recursion back to the first.

The input to the following algorithm is a set S of vertices, each with an associated pointer, forming a rooted tree. Assume that the root of the tree is already identified and points to itself.

```
Supervertex-Pointer-Jumping-Algorithm(S)
    if |S| > 2 then
        for each vertex x ∈ S do
            with probability 1/2, make x a supervertex
        let SV be the set of supervertices, plus the root
        execute Simple-Pointer-Jumping-Algorithm(S−SV,SV)

        for each vertex x in SV do
            perform one pointer jump on x
        comment: at this point the supervertices form
        a rooted tree

        recursively apply the algorithm to SV
        comment: at this point, all vertices in SV point
        to the root

        for each vertex x in S−SV do
            perform a pointer jump on x
        comment: at this point, all vertices point
        to the root
```

It is straightforward to show that the expected work performed by this algorithm is linear in the number of vertices, regardless of the tree structure of the vertices (for proof, see the full paper [5]). The expected number of levels of recursion is $\Theta(\log n)$. Also, the expected number of synchronized substeps at each recursive level may be $\Theta(\log \log n)$. This is because in a list of size $\Theta(n)$, the expected maximum distance between two supervertices is $\Theta(\log n)$. Hence the total expected number of synchronized substeps in the worst case (that is, a list) is $\Theta(\log n \log \log n)$.

Our supervertex algorithm is similar to some of the list-ranking algorithms of Vishkin [15] and Cole and Vishkin [6]. Their algorithms (defined for lists only) can also be thought of as selecting supervertices that proceed to another iteration of pointer jumping while the remaining vertices drop out. Their method is designed to ensure that the work per vertex at each recursive step is constant and the total number of synchronized steps is $O(\log n)$ as opposed to the $\Theta(\log n \log \log n)$ steps of our algorithm. However, their method of choosing supervertices is more complicated, requiring communication between each vertex and its parent.

The differences between them nicely illustrate how the choice of parallel model influences parallel algorithm design. It also shows that although PRAM algorithms may not be tailored for more practical environments, they do contain valuable ideas that can be adapted to real machines. In this case, the valuable idea is that of using randomization to eliminate vertices from the pointer jumping process.

4. Experimental results

In this section we present the implementation results on the CM-5. The program was written in the C language.

For interprocessor communication, message passing routines provided by CM-5's CMMD library were used.

In Section 4.2, we give running times for step 2, implemented using the simple pointer jumping algorithm, the supervertex algorithm, and the packaging algorithm. Since the packaging algorithm is the clear winner, we adopt packaging of data at every phase of our algorithm. In Section 4.3 and 4.4, we examine the increase in imbalance of the data and in the communication needed in the pointer jumping algorithms, as the algorithm proceeds. Finally, in Section 4.5 we present our results on the total running time of our algorithm on up to 64 processors (for detailed results including figures not shown here, see [5]).

name	description
str 0	At each iteration with n vertices, two vertices form a pair ($n/2$ components).
str 1	At each iteration with n vertices, $\sqrt{n}$ vertices form a linear chain (approx. $\sqrt{n}$ components).
str 2	At each iteration with n vertices, $n/2$ vertices form a linear chain and the other $n/2$ vertices form pairs (approx. $n/4$ components).
str 3	At each iteration with n vertices, $\sqrt{n}$ vertices form a complete binary tree (approx. $\sqrt{n}$ components).

Table 1. Structured graphs

4.1. Graph types

We ran our algorithm on four kinds of graphs: random graphs ($G_{n,p}$), random geometric graphs ($U_{n,k}$), structured graphs and TSP graphs. The TSP graphs arise in an application of the traveling salesman problem.

A random $G_{n,p}$ graph has n vertices with each pair connected independently with probability p. A geometric graph $U_{n,k}$ has n vertices, each with outdegree k. The connections are determined as follows: n points corresponding to the vertices are chosen randomly and uniformly on the unit square in the Cartesian plane. Each vertex is then connected to its k nearest neighbors. These graphs were used by Moret and Shapiro [11] in their empirical study of sequential MST algorithms. We tested our algorithm on graphs with 32,000 and 64,000 vertices, with average degree ranging from 1.6 to 12.8.

The TSP graphs used in the experiments, usa13509.tsp and fnl4461.tsp, are from the Electronic Library (eLib) for Mathematical Software of Konrad-Zuse-Zentrum Berlin (http://elib.zib-berlin.de/). The data does not show connections between cities, but indicates "Euclidean 2D." We chose the edges randomly, for a range of probabilities.

The structured graphs described in Table 1 are designed to test extreme cases of the algorithm in different ways. Note
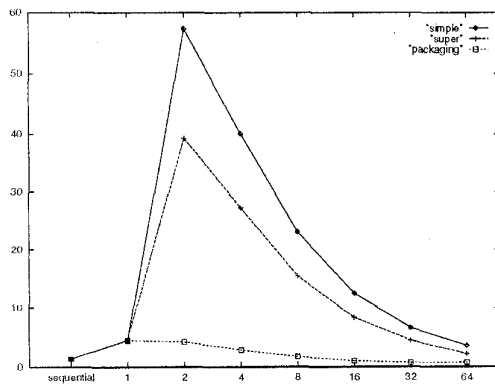
Figure 2. Running time of step 2 for str 1 graph, n = 64,000. The x-axis is the number of processors; the y-axis is the running time of step 2 in seconds.
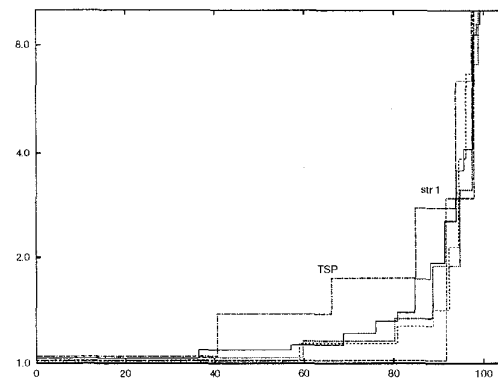

Figure 3. Imbalance in the distribution of edges among 64 processors (y-axis, in log scale) as a function of the percentage of total running time (x-axis) for random (n = 64,000, d = 3.2), geometric (n = 64,000, d = 4.9), TSP (n = 13509, d = 13.5), str 0 and str 1 (n= 64,000, d = 4.0) graphs. The steps correspond to the moments when imbalance changes, as a result of moving from one iteration to the next in the algorithm.

that the shape of the components affects the running time of the pointer jumping algorithm in step 2. Also, the number of components formed at an iteration, which becomes the number of vertices at the next iteration, affects the total number of iterations of the algorithm. To vary the density of the structured graphs, we also added edges of high weight such that they do not affect the structure of the components formed during the algorithm.

4.2. Alternative implementations of step 2

We implemented the following algorithms for step 2: the simple pointer jumping algorithm, the supervertex algorithm, and the packaging algorithm.

On all graph types and sizes we used, we observe that, for all algorithms except the packaging algorithm, there is a huge increase in the running time on 2 processors, as opposed to 1 processor. Figure 2, which shows the running time of step 2 for str 1 graph, is very typical of all cases in this regard.

As expected, the supervertex algorithm showed the most improvement over the simple pointer jumping scheme on structured graphs in which components contain long paths, such as the str 1 and str 2 graphs. However, it performed slightly worse on the structured graph str 0, the TSP graphs and the random graphs. The reason for the relatively good performance of the simple pointer jumping algorithm on the latter graphs is because the components formed at each iteration of the algorithm are very shallow, and thus the algorithm performs only $O(n)$ work. For example, in the str 0 graphs, every vertex does just at most one jump in order to find the root of its component.

On all graph types and sizes we used, the packaging

scheme has by far the best performance. As p increases, however, the running time remains fairly constant, though the message lengths are decreased. This is because the increased number of messages, which is proportional to p, cancels out the speedup effect of the decreased message lengths.

4.3. Imbalance in graph distribution

We computed at each iteration the ratio of the maximum number of vertices and edges at a processor, over the average number of vertices and edges at a processor. We use this ratio as our measure of imbalance of the data at an iteration. Consideration of the balls and bins analogy where the vertices are balls and the processors are bins leads us to expect that the imbalance would increase as p increases. This is because, as the number of bins increases from a constant up to the number n of balls, the ratio of the expected maximum number of balls per bin over the average goes from a constant to $\Theta(\log n/\log\log n)$. Our measurements show that the imbalance does indeed increase as the number of processors increases. However, even with 64 processors, the rate of increase of imbalance is moderate until the last few iterations, when it increases rapidly.

In Figure 3, we plotted the imbalance in the number of edges among 64 processors Since several graphs are superimposed in the figure, it is not possible to identify which

306

graph is which, but the general trends are portrayed. On all of the random graphs, even when $p = 64$, the imbalance is less than 1.3 for 75% of the running time. The TSP and str 1 graphs are worse, but in general we see that the imbalance is less than 2 after 90% of the running time. The relatively poor imbalance in the str 1 graphs after the first iteration is because the size of the graphs decreases by a factor of $\sqrt{n}$, and the small size of the graph by the second iteration leads to poor imbalance.

We also found that the edge imbalance is worse than the vertex imbalance. This appears to be because the length of the edge lists varies more over time.

We implemented several simple schemes that rebalance data at the end of each iteration of the algorithm. One scheme, which we call the E-balancing scheme, redistributes edge lists such that, for 90% of the graphs used, the edge imbalance for $p \leq 32$ is less than 1.1 for 90% of the running time. For $p = 64$, the imbalance is less than 1.2 for 80% of the running time for all but the TSP graphs. Without the time for rebalancing taken into account, the average improvement, taken over all runs in which improvement was made, was less than 2%. In less than 5% of the runs we made, mostly on TSP graphs with $p = 64$, more than 5% improvement was observed, with a maximum of 8.2% (5.8% with rebalancing time counted) and an average of less than 4%. The TSP graphs we used ($n = 13509$, $d = 13.5$) have the property that the number of edges does not decrease in the second and third iterations as fast as the other graphs do, and consequently the running times for the two iterations are not much smaller than the first iteration. Moreover, their relatively greater density results in greater variation in the lengths of edge lists and greater imbalance of edge list distribution among processors. We conclude that an edge balancing scheme is worthwhile only on graphs with properties such as those just listed for the TSP graphs.

4.4. Imbalance in pointer jumps

For the same graphs of Section 4.3, we measured the distribution of pointer jumps in the simple pointer jumping and supervertex algorithms. In this case, we counted the number of queries and responses each processor makes, and defined the imbalance to be the ratio of the maximum, taken over all the processors, divided by the average. As with the imbalance of edge distribution, the imbalance becomes worse as p increases, but even for $p = 64$, shown in Figure 4, there is almost no imbalance in any of our graphs until almost 80% of the running time of step 2 is completed. This is because the time taken by step 2 in the first iteration of the algorithm is large compared with further iterations, and initially vertices are very evenly distributed. For 95% of the running time, the imbalance is typically much less than 2.
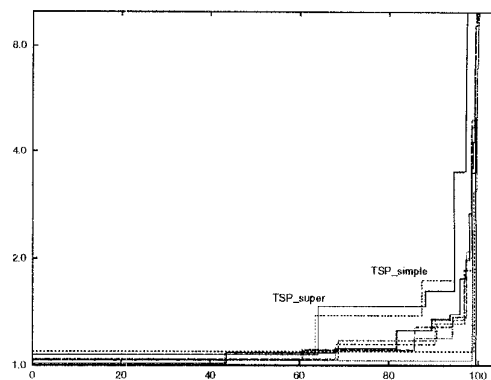

Figure 4. Imbalance in the distribution of pointer jumps among 64 processors.

4.5. Total running time

We measured the total running time of our parallel algorithm, using packaging of messages at every step of the algorithm. For all graph types, we observe the following: The 1-processor parallel algorithm is somewhat slower than the sequential algorithm (run on one CM-5 processor), and the running time on 2 processors is not much better than on 1 processor. This is because the communication costs are already high even with 2 processors. However, on 4 processors the parallel algorithm is always better than the sequential Borůvka's algorithm and good speedup continues up to 32 processors. The results for 64 processors are not much better than for 32 processors. This is in part because of our use of the packaging scheme, in which the number of messages per processor stays constant for all p, and the fact that the number of messages, rather than their size, affects the running time. (For denser graphs, we would expect good speedup for 64 processors and more.) All this is illustrated in figure 5 which shows for random graphs the running time of the sequential versions of Kruskal's and Borůvka's algorithm, as well as the running time of the parallel Borůvka's algorithm on 1 to 64 processors. (Due to insufficient memory on the machine and/or contention during communication, results were not obtainable for (d) and (f) on 2 processors.)

On geometric graphs with average degree 9 and 32,000 vertices, for all the TSP graphs, and for other graphs with fewer vertices but higher average degree, we observed a speedup factor of about 4, on 16 processors, over the sequential Borůvka's algorithm. In general, Kruskal's sequential algorithm ran $2 \sim 3$ times faster than Boruvka's sequential algorithm. For most of our sparse graphs, Boruvka's algorithm starts to beat Kruskal's sequential algorithm at 8 processors.
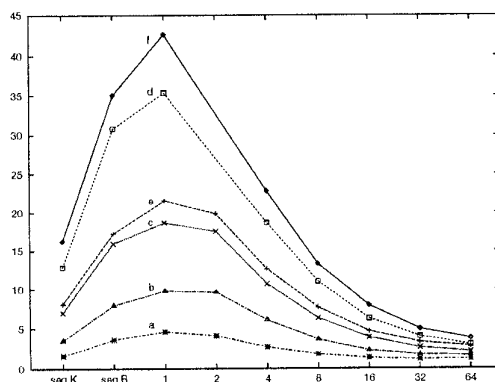
Figure 5. Total running time of packaged Borůvka's algorithm for random graphs: n = 32,000, average degree d = 1.6 (a), 3.2 (b), 6.4 (c), and 12.8 (d); and n = 64,000, d = 3.2 (e) and 6.4 (f). "seq K" is the running time of Kruskal's sequential MST algorithm.

5. Conclusions

Our heuristic analysis of the running time was very useful in predicting the advantages of packaging, the high communication cost of the simple pointer jumping algorithm, and the low imbalance in the distribution of vertices until iterations where the graph is small. Our empirical measurements showed that the imbalance of edges is also low in random and geometric graphs. The performance model proved to also be useful as a basis for designing a practical pointer jumping algorithm.

The speedups obtained by our implementation are far from optimal but we believe that, given the high communication costs, it is difficult to improve the implementation of the standard Borůvka's algorithm for sparse graphs. However, the speedup factor improves for denser graphs. For random graphs, it is certainly possible that refinements of Borůvka's algorithm that exploit the fact that heavy edges are unlikely to be in the MST could lead to improved results. On the CM-5, improvements can be made using Active Messages, though we chose CMMD message passing routines because we were interested in a machine model with high communication costs.

Acknowledgements

We thank Eric Bach and the anonymous referees for their valuable comments and suggestions.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[2] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859-868, 1991.

[3] R. S. Barr, R. V. Helgaon and J. L. Kennington. Minimal spanning trees: An empirical investigation of parallel algorithms. *Parallel Computing*, 12(1):45-52, October 1989.

[4] O. Borůvka. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně (Acta Societ. Scient. Natur. Moravicae)*, 3:37-58, 1926.

[5] S. Chung and A. Condon. Parallel implementation of Borůvka's minimum spanning tree algorithm. Technical Report 1297, Computer Sciences Department, University of Wisconsin, Madison, 1996.

[6] R. Cole and U. Vishkin. Approximate parallel scheduling, Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Computing*, 17(1):128-142, 1988.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken. LogP: Towards a realistic model of parallel computation. *4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 1993.

[8] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43-57, 1985.

[9] S. L. Johnsson. The Connection Machine systems CM-5. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 365-366, June 1993.

[10] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorical Computing*, ACM Press, New York, NY, 1993.

[11] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 15:99-117, American Mathematical Society, Providence, RI, 1994.

[12] D. Roweth. The Meiko CS-2 system architecture. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, p. 213, June 1993.

[13] W. Oed and M. Walker. An overview of Cray research computers. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 271-272, June 1993.

[14] M. Snir. Issues and directions in scalable parallel computing. Research Report Number RC 18940 (82749), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, 1993.

[15] U. Vishkin. Randomized speed-ups in parallel computation. *Proc. 16th Annual ACM Symposium on Theory of Computing*, 230-239, 1984.