Parallel Implementation of Boruvka's Minimum Spanning Tree Algorithm

Sun Chung
Anne Condon

Technical Report #1297

May 1996

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.

*Supported in part by NSF grant number CCR-9257241 and by matching awards from Thinking Machines Corporation and Digital Systems Corporation. E-mail addresses of the authors are: sunc@cs.wisc.edu, condon@cs.wisc.edu.

1 Introduction

A dominant emerging parallel architecture consists of a collection of fast processors, connected by a robust communication network [CKP+93, Sni93]. 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. Examples already available include the TMC CM-5 [Joh93], Meiko CS-2 [Row93], Cray T3D [OW93] and the IBM SP-1 [Sni93].

In this paper, we describe our experience with design and implementation of parallel algorithms for the minimum spanning tree problem. 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 [Bor26], which is considered to be "well suited for parallel computation" [Tar83]. 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. 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. Our algorithm is similar to the list ranking algorithms of Vishkin [Vis84] and Cole and Vishkin [CV86], 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. For the graphs used in our experiments, load balancing schemes result in little improvement in running times.

We present our contributions in more detail in Section 1.3, along with a brief summary of our implementation results. First, we describe Borůvka's algorithm and our parallel model in Sections 1.1 and 1.2, respectively.

1.1 Borůvka's Algorithm

The minimum spanning tree problem is as follows. Given a connected, undirected graph G with n vertices and m weighted edges, find a spanning tree of minimum weight. Our minimum spanning tree algorithms can easily be adapted to solve the case where the graph is not connected.

Borůvka's minimum spanning tree algorithm, also known as Sollin's algorithm [Sol77], 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 (with the original vertex names, as they were at the start of the algorithm). Using standard techniques (see [GH85]) 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 $O(m \log n)$.

1.2 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 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; for example 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.

We let n be the number of vertices, and m the number of edges of the graph. We are interested in the case where p is relatively small compared with n, and assume this throughout. We assume that the graph is represented using adjacency lists, and that the edge list of any given vertex is completely stored at one processor (and is not replicated).

1.3 Contributions of This Paper

In Section 3 we show that if we assume that the data remains evenly distributed throughout the algorithm, then a straightforward parallel implementation of Borůvka's algorithm has

running time $\Theta((t_s+t_w)(m \log n)/p)$. This implies a speedup proportional to the number of processors, but the constant $\Theta(t_s+t_w)$ may be huge. We observe that the parallel algorithm suffers the worst slow-down in step 2, where the parallel algorithm employs a $\Theta(n \log n)$ pointer jumping scheme instead of linear time depth first search used in the sequential algorithm.

In Section 4, we consider several alternative implementations of step 2. One simple idea is that of packaging all messages to be sent from one processor to another. This results in relatively few long messages instead of many short messages. Our experimental results show that this results in a dramatic improvement on the CM-5, where p is relatively small and t_s is large. We also consider linear time pointer jumping schemes for step 2 of the algorithm. We present a new randomized pointer jumping scheme that also has expected linear work. Our implementation results show that it is much faster than the $\Theta(n \log n)$ pointer jumping scheme on lists, although it does not beat the simple packaging scheme on the CM-5. However, the algorithm is likely to be useful when p is larger. It is also useful in programming environments, such as the Split-C language, where the exact distribution of data is not known to the programmer and thus the packaging scheme cannot be implemented.

In Section 5.3 we study the degree to which the distribution of data becomes unbalanced as the algorithm progresses. Our simple parallel algorithm (to be presented in Section 3) achieves a speedup proportional to p only if the data (and thus the work distribution) remains approximately uniformly distributed, so this is important to measure. Our empirical results show that the data does not become unduly unbalanced until the subproblem to be solved is very small, by which time the parallel time for the remaining iterations is a tiny fraction of the total running time. From our empirical measurements, we conclude that at least for small p, effort spent on rebalancing the data is not justified for our application.

We tested our algorithm on several graph types, including random ($G_{n,p}$), geometric and structured graphs as well as on data arising in a real instance of the Traveling Salesman problem. On geometric graphs with average degree 9 and 32,000 vertices, 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 algorithm.

On higher density graphs, the speedup factor improves. Communication costs account for most of the slowdown.

2 Background and Related Work

2.1 Minimum Spanning Tree Algorithms

There are three classical $O(m \log n)$ minimum spanning tree algorithms: Borůvka's, Prim's, and Kruskal's [Bor26, Pri57, Kru56]. Of the three, Kruskal's algorithm is reported to be the best choice for many types of graphs, although Borůvka's algorithm is best for very sparse graphs [BHK89]. Also, Borůvka's algorithm is considered to be best suited for parallel computation [Tar83]. Other sequential variants of these algorithms use more elaborate data structures to improve the asymptotic running time. However, these structures are

difficult to implement in a distributed memory parallel environment. (See Knuth [Knu93] and Moret and Shapiro [MS94] for empirical assessments of sequential minimum spanning tree algorithms.)

Parallel variants of the classical algorithms have been designed for different machine models, primarily the PRAM model [HCS79, SJ81, CLC82, CV88, AS87, JM92, CKT94]. Because the PRAM model assumes a synchronous shared memory in which no communication cost is incurred, these algorithms are not well suited to our model. Other parallel algorithms have been developed for fixed interconnection networks such as meshes, butterflies, hypercubes, and shuffle-exchange network [Ben80, DY81, KGGK94, AS87, DV87, Lei83, NMB83]. Although they are practical and efficient on a particular network, these algorithms owe their efficiency to the regular communication patterns made possible by the network or by the assumption that the graphs are dense. Barr *et al.* [BHK89] give an empirical investigation of both sequential and parallel minimum spanning tree algorithms. For Borůvka's algorithm, they report a speedup factor of about 6 with 10 processors, for grid graphs, on the Sequent Symmetry S81 multicomputer.

2.2 Related Work on Parallel Models and Graph Algorithms

Narendran *et al.* [NDT93] and Krishnamurthy *et al.* [KLCY94] reported on implementations of min cost flow and connected components algorithms on the CM-5, respectively. As we do in our implementation, they both distribute the vertices evenly among the processors, and store the complete edge list of one vertex at the same processor. The connected components problem is a special case of the minimum spanning tree problem. A similarity between our work and the work of Krishnamurthy *et al.* is the use of pointer jumping as a subroutine in the algorithm.

Culler *et al.* [CKP+93] introduced the LogP model to be used in analyzing the performance of algorithms on the same architectures that we study. However, they focused on overlapping communication with computation, and the problems they studied – broadcast, summation and FFT – have the property that the communication pattern is simple and regular. The LogP model uses the parameters L (latency), o (overhead), g (gap between messages), and P (number of processors). The t_s parameter of our model corresponds to LogP model's latency and overhead combined. It is critical for the LogP model to differentiate the two parameters, because it is concerned with short messages and with overlapping computation with communication. Since we are not concerned with overlapping computation with communication, we do not find it necessary to differentiate latency and overhead.

The LogGP model of Alexandrov *et al.* [AISS95] is an extension of the LogP model that uses another parameter, G (gap per byte), to account for long messages. The LogGP model's G parameter translates to our model's t_w, except for the difference of "per byte" and "per word."

3 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 technique of pointer jumping (see [KR88]). Initially, the root of each component sets its pointer to point at itself. Each remaining vertex initially points at the other endpoint of the lightest edge incident on it. The vertex pointers are then updated by repeated pointer jumps, where in one pointer jump a vertex updates its pointer to equal that of its parent. The following algorithm summarizes this, where the input S is the set of non-root vertices and the input R is the set of 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. Two messages, one each for request and reply, are required to obtain a piece of information from another processor which has it.

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. It is also possible to use an asynchronous version of the algorithm, in which vertices can update their pointers at differing rates. However, our experiments and those of Krishnamurthy *et al.* show that the asynchronous version has a similar running time to the synchronous version, so we do not consider it further.

3.1 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 $O((t_s + t_w)m/p)$. Recall that we are assuming that p is much smaller than n. 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. Thus, the computation costs of the first iteration of the algorithm are split fairly evenly among the processors, because the computation per processor is linear in the

number of edges at that processor. 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.

In step 2, the pointer jumping step, the distribution of work depends on the structure of the components formed in step 1. Consider the expected maximum number of messages sent and received by a processor at an iteration of the repeat loop in which many ($\Omega(p \log p)$) vertices are updating their pointers. Assuming that all vertices in the components have constant indegree, the number of messages of a processor is a constant times the number of vertices at that processor. This means that the expected maximum number of messages at a processor is within a constant factor of the average number of vertices, and thus messages, at a processor. We can then expect that the parallel time for step 2 is $O((t_s+t_w)(n \log n)/p)$. We can extend this argument to show that even if a very small number of vertices have relatively large indegree, then the parallel time is still $O((t_s + t_w)(n \log n)/p)$. Suppose for example that a fraction $O(1/(f(n) \log n))$ of the vertices have indegree $f(n)$ or less. Then the expected maximum number of these vertices at one processor is $O(n \log n/(f(n)p \log n)) = O(n/(f(n)p))$. This processor handles at most $f(n)$ messages requesting the pointer value of each of these vertices, giving a total of $O(n/p)$ messages. This is still within a constant factor of the $\Theta(n/p)$ messages this processor handles for its other vertices (i.e. those with constant indegree).

Even this heuristic analysis, however, is difficult to extend to further iterations. We can 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 $\Omega(p)$ can be achieved, only if the iterations in which the data is badly distributed contribute very little to the running time. 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, so we can still assume that they are randomly and uniformly distributed. 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 5.3.

4 Alternative Parallel Algorithms for Step 2 (find root)

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.

4.1 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. The cost t_s is charged at most p times per synchronized substep, whereas in the simple pointer jumping algorithm without packaging, it is charged $\Theta(n/p)$ times (assuming balanced communication). Therefore, if $p << n/p$, we expect that the packaging scheme will improve the running time of step 2. For larger p, however, the advantage of packaging may be lost.

4.2 A New Randomized Pointer Jumping Algorithm

Both deterministic and randomized list ranking algorithms that require only linear work are well known [Vis84, CV88, AM91]. However, they are not well suited to our application for the following reason: since they are list ranking algorithms, they require that the input data are linear lists, whereas we apply pointer jumping to rooted trees. By "linearizing" the trees, one could apply these techniques. But it is preferable not to do this, since the path from a vertex to the root of its component may be much shorter in the tree than in the linear list. (Goldberg *et al.* [GPS87] present a symmetry-breaking technique which works on rooted trees, but it is impractical for our application.)

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 of the vertices called supervertices are selected. Call this set SV. 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 of Appendix A illustrates the execution of this algorithm on a list.

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 $x \in S$ **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. We include the proof of this in Appendix A. 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 $\Theta(\log n \log \log n)$. A completely asynchronous version of this algorithm could also be implemented, but we have not done this.

Our supervertex algorithm is similar to one of the list ranking algorithms of Vishkin [Vis84] which uses the "random mate algorithm," and to that of Cole and Vishkin [CV86] which uses the "2-ruling set algorithm." 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. (Subramonian [Sub92] presents an implementation technique of the "random mate algorithm" which, by broadcasting pairs of random numbers, avoids interprocessor communication that would otherwise be required.)

The differences between their algorithms and ours nicely illustrate how the choice of parallel model influences parallel algorithm design. It also shows that, though 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.

4.2.1 A Lazy Supervertex Scheme

For our application, namely Borůvka's algorithm, an interesting variant of the supervertex algorithm can be employed. In this variant, the algorithm is stopped just before the first recursive call is executed. Thus, the supervertices are defined to be roots of components. As a result, the number of components is greater, and more iterations of the algorithm are required. The advantage is that the time required for step 2 in each iteration is reduced. It is straightforward to show that this algorithm correctly computes the minimum spanning tree of a graph.

5 Experimental Results

In this section we present the implementation results on the CM-5. The program used in the experiments on the CM-5 was written in the C language. For interprocessor communication, message passing routines provided by CM-5's CMMD library were used.

In Section 5.2, we give running times for step 2, implemented using the simple pointer jumping algorithm, the supervertex algorithm, the lazy supervertex algorithm and with packaging. Since the packaging algorithm is the clear winner, we adopt packaging of data at every phase of our algorithm. In Section 5.3 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 5.4 we present our results on the total running time of our algorithm on up to 64 processors.

5.1 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 [MS94] in their empirical study of sequential minimum spanning tree algorithms. The random graphs and the geometric graphs are not necessarily connected. 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 are from the Electronic Library (eLib) for Mathematical Software of Konrad-Zuse-Zentrum Berlin (http://elib.zib-berlin.de/). One is *usa13509.tsp*, representing 13,509 cities with population at least 500 in the continental U.S. (by David Applegate and Andre Rohe), and the other is *fnl4461.tsp*, representing 4,461 cities in the five new federal states of Germany (by Bachem/Wottawa). 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 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.

graph type	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 used in the experiments on the CM-5

5.2 A Comparison of Alternative Implementations of Step 2

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

In Figures 2 and 3, we see 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.

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.

Not surprisingly, the lazy supervertex algorithm was better than the supervertex algorithm. One interesting feature of the lazy supervertex algorithm is that the time taken is very close for all graph types of the same size. That is, the lazy supervertex algorithm tends to smooth out differences in running time due to graph structure. However, since only the time for step 2 is reported, the data here does not show the true story: the implementation using the lazy algorithm incurs extra costs at other steps due to an increase in the number of iterations. Further experiments showed that the total running time of our algorithm, using the lazy supervertex algorithm in step 2, was better than the supervertex and simple algorithms only on very sparse graphs.

Figures 2 and 3 show that the packaging scheme has by far the best performance on all graphs. 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.

5.3 Imbalance of Data and Workload

5.3.1 Imbalance in Graph Distribution

For several graphs, 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 did this for the simple pointer jumping and lazy supervertex algorithms. 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 4, we plotted the imbalance in the number of edges as a function of the percentage of total running time. Since several graphs are superimposed in each of our illustrations, it is not possible to identify which graph is which, but the general trends are portrayed. On all of the random graphs (each with 64,000 vertices), 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. These properties, and similar improvement of running time by rebalancing schemes, are also observed on random graphs of the same size in a range of density, indicating that such effects of rebalancing schemes are not unique for TSP graphs we used but may be common to different graphs with these properties. We conclude the following: (1) for small p, effort spent on rebalancing the data is not justified for our

application; (2) an edge balancing scheme is worthwhile only on graphs with properties such as those described above.

5.3.2 Imbalance in Distribution of Pointer Jumps

We also measured the distribution of pointer jumps in the simple pointer jumping, supervertex, and lazy 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. In Figure 5 we plotted this as a function of the percentage of the running time taken in step 2 of the algorithm. Again, the imbalance is worse as p increases, but even for $p = 64$, 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.

5.4 Total Running Times for Parallel Borůvka's Algorithm using Packaging

We measured the total running time of our parallel Borůvka's algorithm, using packaging of messages at every step of the algorithm. For four graph types, with varying sizes and densities, Figure 6 shows the running time of the sequential versions of Kruskal's and Borůvka's algorithm (both run on one CM-5 processor), as well as the running time of the parallel Borůvka's algorithm on 1 to 64 processors.

The running time of parallel Borůvka's algorithm on 1 processor is somewhat slower than that of the sequential Borůvka's algorithm, 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 Borůvka's 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.

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. For most of the graphs we used, Kruskal's sequential algorithm ran $2 \sim 3$ times faster than Borůvka's sequential algorithm. For most of our sparse graphs, Borůvka's algorithm starts to beat Kruskal's sequential algorithm at 8 processors.

6 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.

In future work, we will extend our analysis of the work and running time of the super-vertex pointer jumping algorithm. In the appendix of this paper, we show that the expected work done is linear on any tree structure. We can extend this to show that the work is linear with very high probability in the case of lists. However, this does not appear to be the case for trees, because as the tree size is reduced, the indegree of the vertices increases quickly. This results in high variance in the work distribution. We are working on a variant of this algorithm that eliminates this problem. Finally, we plan to design and implement a parallel version of Kruskal's algorithm, which is the best of the three classical sequential algorithms on dense graphs. We plan to compare Borůvka's algorithm with Kruskal's algorithm on a wide range of graph densities.

Acknowledgements

We thank Eric Bach and the anonymous referees of IPPS '96 for their valuable comments and suggestions.

Appendix A

Theorem: The expected work done in the supervertex algorithm is linear in the number of vertices in the input.

Proof: We show that the expected work done on any vertex x is $O(1)$. To see this, we consider separately the iterations (recursive levels) in which x is a supervertex, and the single iteration in which x is not a supervertex. If x is a supervertex at an iteration, the work done on x is $O(1)$, since it does not participate in the simple pointer jumping algorithm. Also, the expected number of iterations in which it is a supervertex is 2, since at each iteration it is not a supervertex with probability 1/2. Hence, the total expected work done on x over all iterations in which x is a supervertex is $O(1)$.

If a vertex is not a supervertex at some iteration, then the work done in that iteration is linear in the number of pointer jumps performed. The expected number of jumps performed is at most $\sum_{i=1}^{\infty} i2^{-i} = O(1)$, since with probability 2^{-i} the supervertex that it points to as a result of the simple pointer jumping algorithm is of distance i from this vertex. Once the vertex is not a supervertex in some iteration, it participates in no further iterations. Hence the total expected work is $O(1)$, as required.

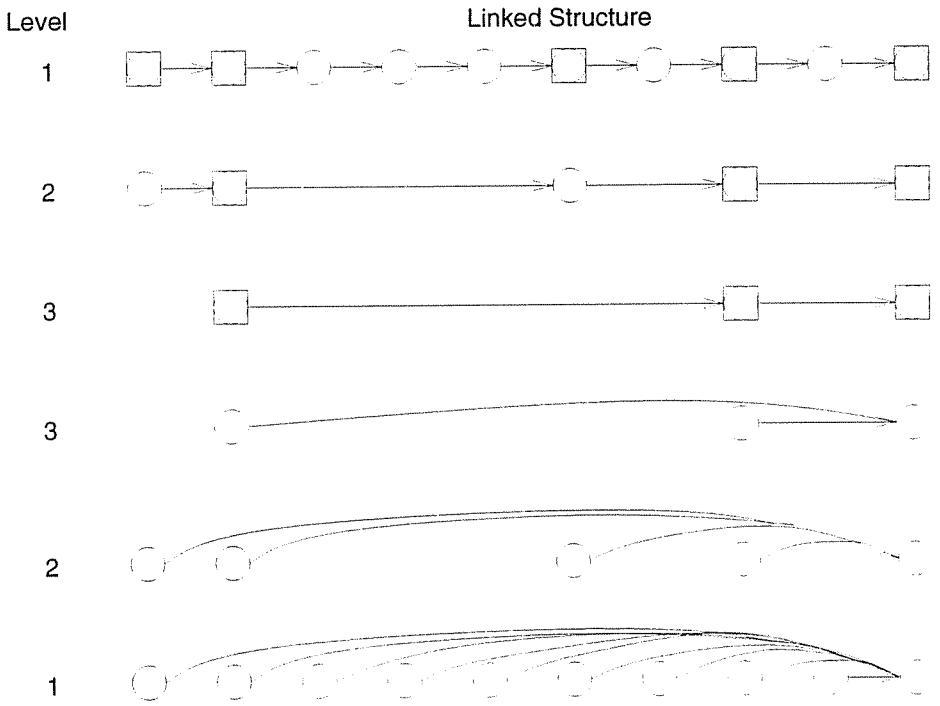


Figure 1: Execution of the supervertex algorithm on a list. The first three rows show the linked structure at the start of the three recursive calls. Vertices in squares are chosen to be supervertices at each of these iterations. The last three rows show the vertices that point to the root at the end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

References

[AISS95] 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.

[AM91] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859-868, 1991.

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36(10):1258-1263, October 1987.

[BHK89] 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.

[Ben80] J. L. Bentley. A parallel algorithm for constructing minimum spanning trees. *Journal of Algorithms*, 1:51-59, 1980.

[Bor26] O. Borůvka. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně (Acta Societ. Scient. Natur. Moravicae)*, 3:37-58, 1926.

[CLC82] F. Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659-665, September 1982.

[CKT94] R. Cole, P. N. Klein, and R. E. Tarjan. A linear-work parallel algorithm for finding minimum spanning trees. *Proc. 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32-53, 1986.

[CV88] 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.

[CKP+93] 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.

[DY81] N. Deo and Y. B. Yoo. Parallel algorithms for the minimu spanning tree problem. *Proc. 1981 International Conference on Parallel Processing*, 188-189, 1981.

[DV87] K. A. Doshi and P. J. Varman. Optimal graph algorithms on a fixed-size linear array. *IEEE Transactions on Computers*, C-36(4):460-470, April 1987.

[GH85] 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.

[GPS87] A. V. Goldberg, S. A. Plotkin and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. *Proc. 19th Annual ACM Symposium on Theory of Computing*, 315-324, 1987.

[HCS79] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461-464, August 1979.

[JM92] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 363-372, 1992.

[Joh93] L. Johnsson. The Connection Machine systems CM-5. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 365-366, June 1993.

[Knu93] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorical Computing*, ACM Press, New York, NY, 1993.

[KR88] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report 408, University of California, Berkeley, 1988.

[KLCY94] A. Krishnamurthy, S. Lumetta, D. E. Culler and K. Yelick. Connected components on distributed memory machines. *DIMACS Implementational Challenge*, 1994.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7(1):48-50, 1956.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*, Benjamin/Cummings, Redwood City, CA, 1994.

[Lei83] F. T. Leighton. Parallel computations using meshes of trees. *Proc. 1983 International Workshop of Graph Theoretic Computer Science*, 1983.

[MS94] 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.

[NDT93] B. Narendran, R. De Leone and P. Tiwari. An implementation of the ϵ-relaxation algorithm on CM-5. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 183-192, June 1993.

[NMB83] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt. Efficient VLSI networks for parallel processing based on orthogonal trees. *IEEE Transactions on Computers*, C-32:21-23, June 1983.

[OW93] W. Oed and M. Walker. An overview of Cray Research computers including the Y-MP/C90 and the new MPP T3D. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 271-272, June 1993.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389-1401, November 1957.

[Row93] D. Roweth. The Meiko CS-2 system architecture. *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, p. 213, June 1993.

[SBR73] R. G. Saltman, G. R. Bolotsky, and Z. G. Ruthberg. Heuristic cost optimization of the federal Telpak network. Tech. Note 787, National Bureau of Standards, Washington, D.C., June 1973.

[SJ81] C. Savage and J. Jájá. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal of Computing*, 10(4):682-691, November 1981.

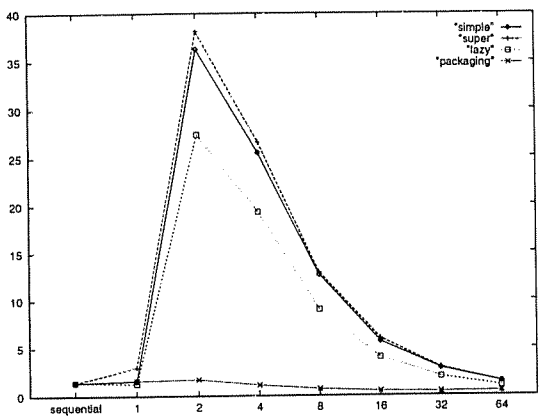
[Sni93] M. Snir. Issues and directions in scalable parallel computing. Research Report Number RC 18940 (82749), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, 1993.

[Sol77] M. Sollin. An algorithm attributed to Sollin. In *Introduction to the Design and Analysis of Algorithms*, S. E. Goodman and S. T. Hedetniemi, eds., sec. 5.5, McGraw-Hill, Cambridge, MA, 1977.

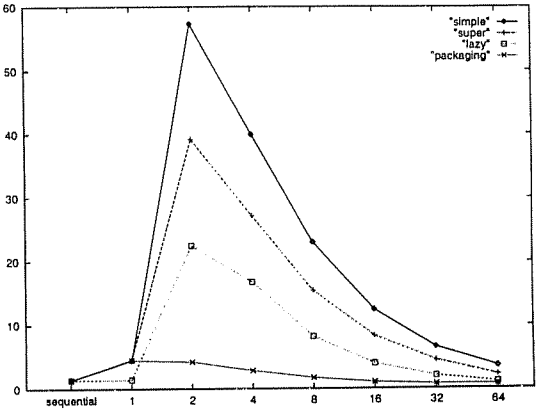
[Sub92] R. Subramonian. The influence of Limited Bandwidth on Algorithm Design and Implementation. *Proc. Dartmouth Institute for Advanced Graduate Studies (DAGS) 1992 Conference on CD-ROM*, P.A. Gloor, F. Makedon, and J.W. Matthews, eds., TELOS, Santa Clara, CA, 1994.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

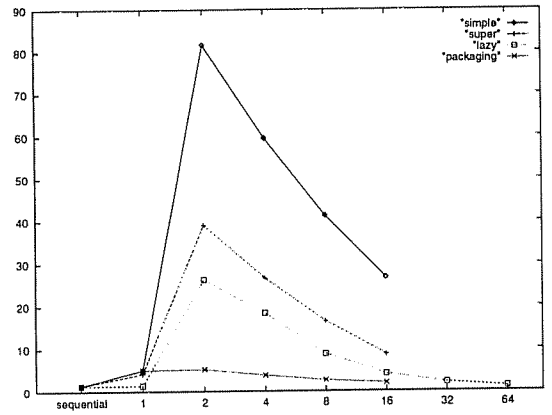
[Vis84] U. Vishkin. Randomized speed-ups in parallel computation. *Proc. 16th Annual ACM Symposium on Theory of Computing*, 230-239, 1984.



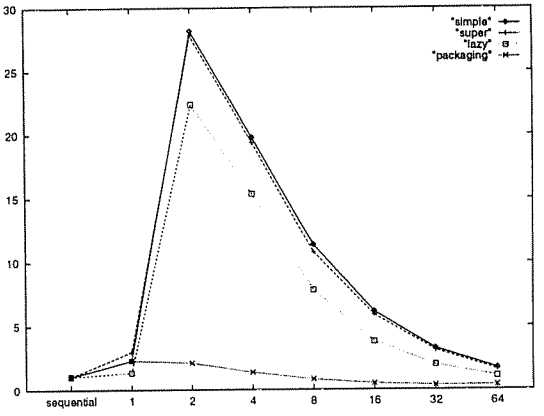
str 0 graph: $n = 64,000$.



str 1 graph: $n = 64,000$.

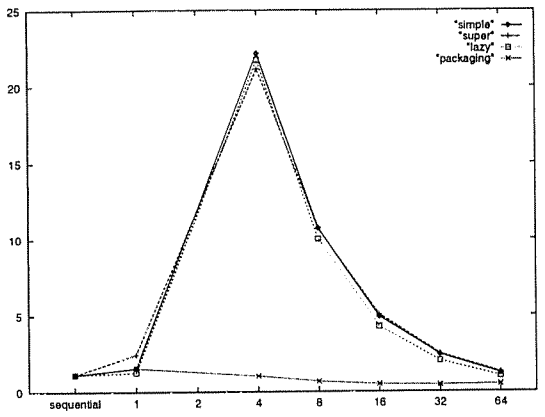


str 2 graph: $n = 64,000$.

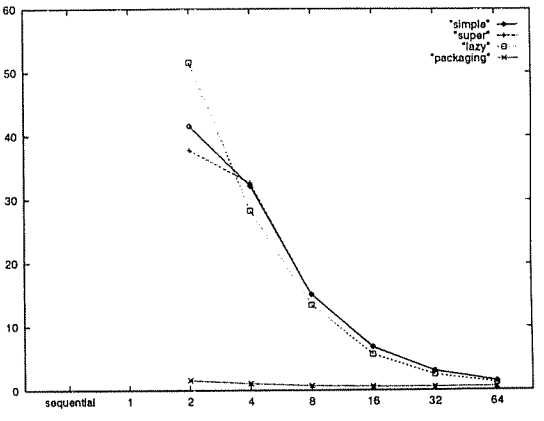


str 3 graph: $n = 64,000$.

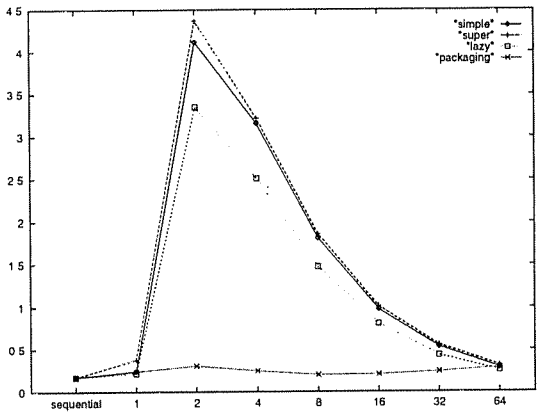
Figure 2: Running Time of Step 2. The graphs show the running time of step 2 for four structured graphs with $n = 64,000$. (For structured graphs, the running time of step 2 is not affected by the average degree d.) For each graph, results of four algorithms are shown: simple pointer jumping, supervertex algorithm, lazy supervertex, and packaging. The x-axis is the number of processors; the y-axis is the running time of step 2. All times shown are in seconds. For str 2, which requires the largest number of pointer jumps of all structured graphs, results were not obtainable for $p = 32$ and 64, except for the lazy supervertex algorithm, due to lack of buffer space.



random graph: $n = 64,000$, $d = 6.4$.

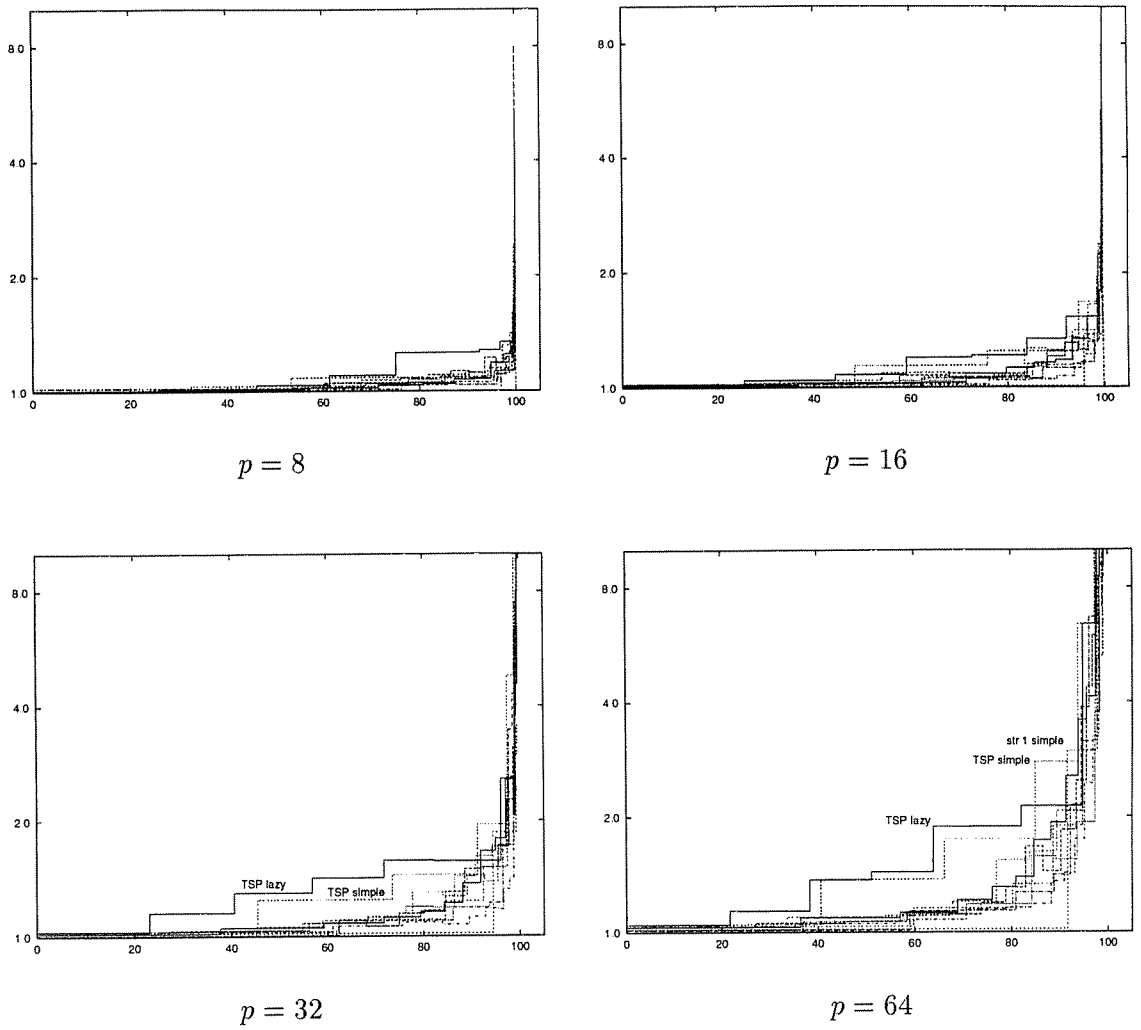


geometric graph: $n = 64,000$, $d = 9.2$.



TSP graph: $n = 13509$, $d = 27.0$.

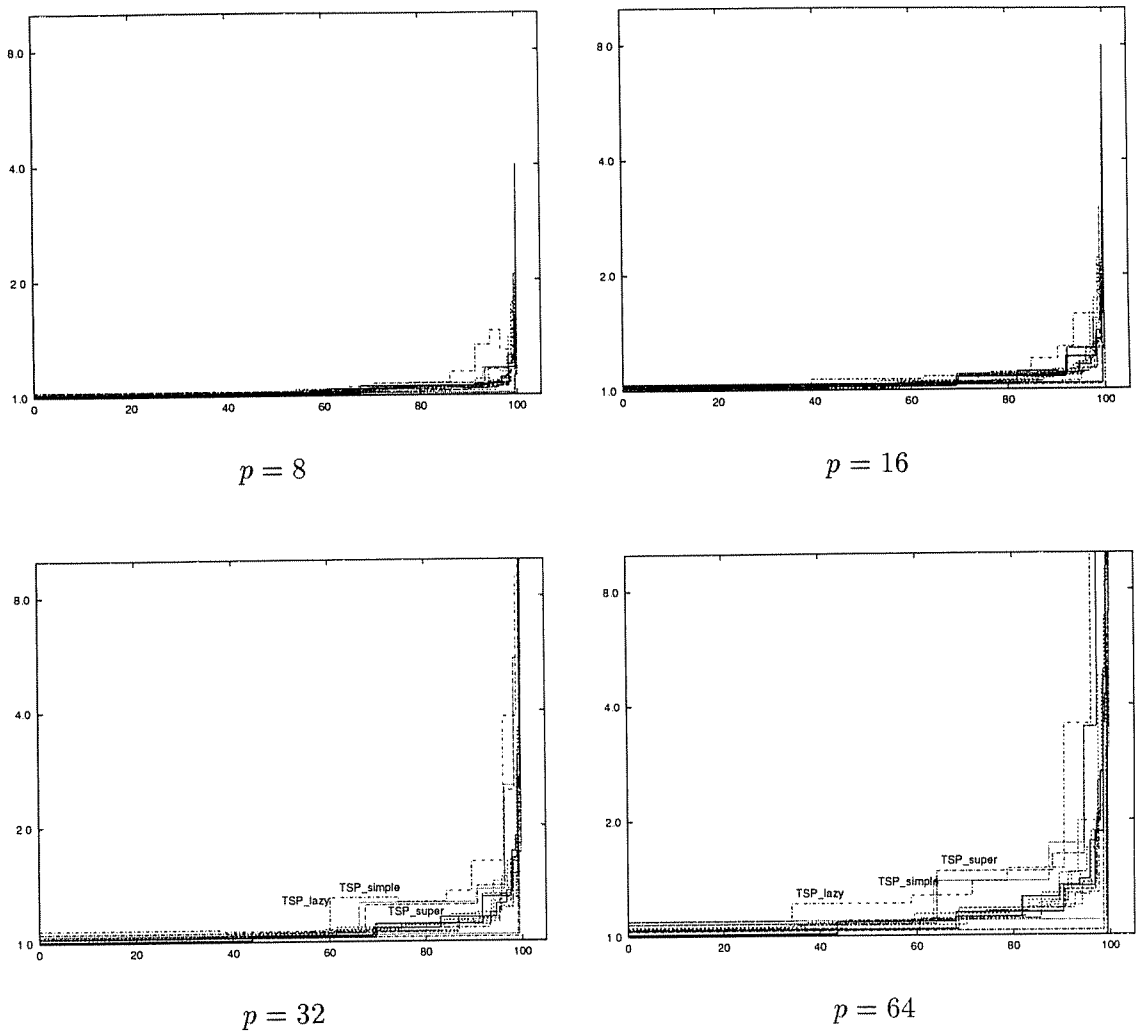
Figure 3: Running Time of Step 2. The graphs show the running time of step 2 for random, geometric, and TSP graphs. For each graph, results of four algorithms are shown: simple pointer jumping, supervertex algorithm, lazy supervertex, and packaging. The x-axis is the number of processors; the y-axis is the running time of step 2. All times shown are in seconds. Due to limited memory space, some results were not obtainable and are not shown here (random, for $p = 2$; geometric, for sequential and $p = 1$).



$p = 8$ $p = 16$

$p = 32$ $p = 64$

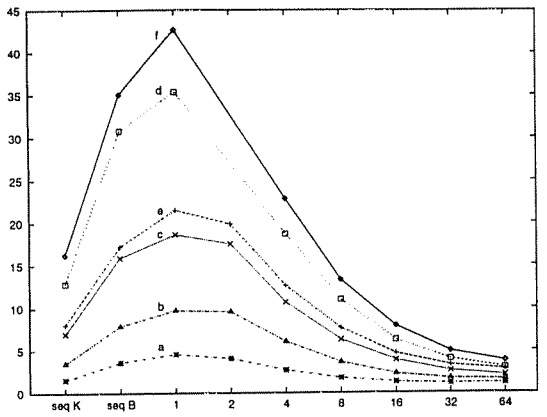
Figure 4: Imbalance in the Distribution of Edges. The graphs show the imbalance in the distribution of edges among processors (y-axis) given as a function of the percentage of total running time (x-axis). Results are shown in logscale for five graph types: random ($n = 64,000$, $d = 3.2$), geometric ($n = 64,000$, $d = 4.9$), TSP ($n = 13509$, $d = 13.5$), str 0 and str 1 ($n = 64,000$, $d = 4.0$); two algorithms each (simple pointer jumping and lazy supervertex algorithm). The distribution of edges for the supervertex algorithm is the same as that of the simple pointer jumping. The steps in the imbalance function correspond to the moments when imbalance changes, as a result of moving from one iteration to the next in the algorithm.



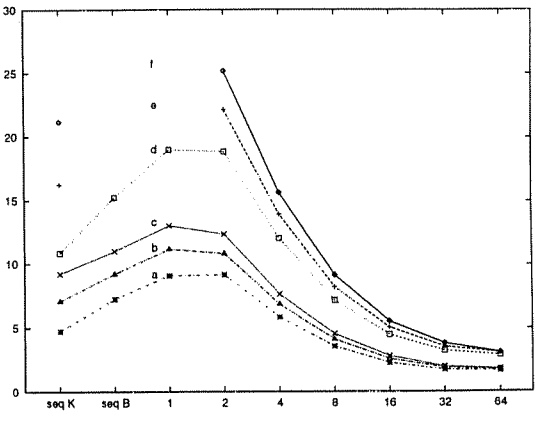
$p = 8$ $p = 16$

$p = 32$ $p = 64$

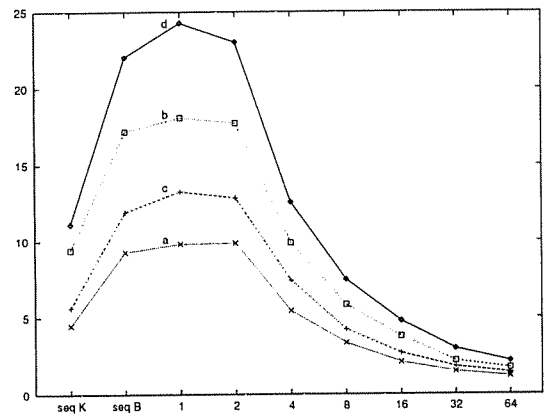
Figure 5: Imbalance in the Distribution of Pointer Jumps. The graphs show the imbalance (y-axis) in the distribution of pointer jumps per processor given as a function of the percentage of total running time (x-axis). Results are shown in logscale for five graph types: random ($n = 64,000$, $d = 3.2$), geometric ($n = 64,000$, $d = 4.9$), TSP ($n = 13509$, $d = 13.5$), str 0 and str 1 ($n = 64,000$, d irrelevant); three algorithms each (simple pointer jumping, supervertex, and lazy supervertex algorithm). The steps in the imbalance function correspond to the moments when imbalance changes, as a result of moving from one iteration to the next in the algorithm.



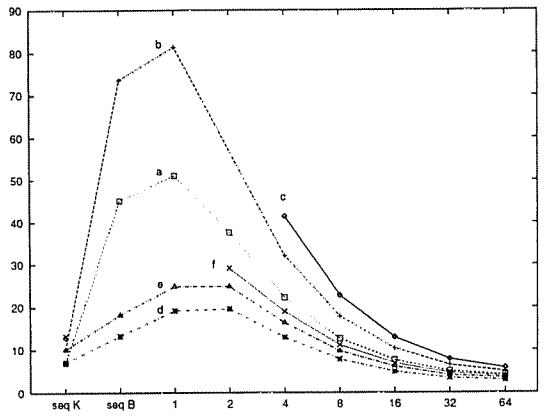
random graphs: $n = 32{,}000$, average degree $d = 1.6$ (a), 3.2 (b), 6.4 (c), and 12.8 (d); and $n = 64{,}000$, $d = 3.2$ (e) and 6.4 (f).



geometric graphs: $n = 32,000$, $d = 4.9$ (a), 7.1 (b), and 9.3 (c); and $n = 64,000$, $d = 4.9$ (d), 7.1 (e), and 9.2 (f).



TSP graphs: $n = 4461$, $d = 44.6$ (a) and 89.2 (b); and $n = 13509$, $d = 13.5$ (c) and 27.0 (d).



structured graphs: $n = 64,000$, $d = 4.0$ (a), 6.0 (b), and 8.0 (c) for str 0, and $d = 4.0$ (d), 6.0 (e), and 8.0 (f) for str 1.

Figure 6: Total Running Time of Packaged Borůvka's Algorithm. The x-axis shows the numbers of processors used; "seq K" and "seq B" are for Kruskal's and Borůvka's sequential algorithm, respectively. The y-axis shows the total running time. All times shown are in seconds. Due to insufficient memory on the machine and/or contention during communication, results were not obtainable for some of the graphs on a small number of processors (random, for $p = 2$; geometric $n = 64,000$, for sequential and $p = 1$; str 0 and 1, for sequential and $p = 1$ and 2). The results for str 2 and str 3 graphs show a similar pattern as those of str 0 and str 1 graphs and are not shown here.