Parallel Prim's algorithm on dense graphs with a novel extension

Ekaterina Gonina and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
egonina2@uiuc.edu, kale@cs.uiuc.edu

November 16, 2007

Abstract

This paper describes parallel implementation of Prim's algorithm for finding a minimum spanning tree of a dense graph using MPI. Our algorithm uses a novel extension of adding multiple vertices per iteration to achieve significant performance improvements on large problems (up to 200,000 vertices). We describe several experimental results on large graphs illustrating the advantages of our approach on over a thousand processors.[1]

1 Introduction

The problem of finding a minimum spanning tree of a graph is an important building block for many graph algorithms, and has been extensively studied by many scientists and mathematicians. The problem has applications in the design of distributed computer and communication networks, wiring connections, transportation networks among cities, and designing pipe capacities in flow networks [1]. It also has indirect applications in problems of image processing, speech recognition, network reliability, and surface homogeneity [1]. The problem of finding a minimum spanning tree is as follows: given a weighted undirected graph G, with N vertices and E edges, we define a minimum spanning tree (MST) as a connected subgraph of G for which the sum of the weights of its edges is minimized. One of the pioneers in solving the problem of MST was R.C. Prim. Prim's algorithm is a greedy algorithm; it starts by selecting an arbitrary vertex as the root of the tree. It then grows the tree by adding a vertex that is closest (has the shortest edge to) the current tree, and adding the shortest edge from any vertex already in the tree to the new vertex. The algorithm terminates once all vertices have been added to the tree. The sum of all added edges is the cost of the MST [2]. The serial computational complexity of the algorithm is $\Theta(N^2)$. This algorithm is best

[1]The above research was facilitated through an allocation of advanced computing resources on the BigBen system at the Pittsburgh Supercomputing Center.

suited for dense graphs. Several parallel versions of MST algorithms appear in literature, however only a few of them describe an implementation of parallel Prim's algorithm [2, 3, 4]. Among the most common parallel algorithms for the MST problem is Boruvka's algorithm: [5, 6, 7, 8, 9]. Many suggest new parallel algorithms for finding minimum spanning trees [10, 11, 12, 13, 14, 15]. However, we could find only one experimental documentation on the problem, Barr's experiments in [3]. His investigation shows results for parallel Sollin's, Kruskal's, and Prim's algorithms on sparse and dense graphs on up to 10 processors. His experiments confirm that Prim's algorithm performed best on 100% dense graphs, as expected [3, page 50]. In this paper we describe our implementation of Prim's algorithm for dense graphs and experimental results we obtained by running the algorithm on 1 to 1024 processors. The machine we used to test the performance of our algorithm was the Cray XT3 in the Pittsburgh Supercomputing Center. We present a discussion of issues involved in large-scale parallelization of Prim's algorithm, along with a novel extension to the basic parallelization of Prim's algorithm that increases its efficiency significantly. We also describe the parallel implementation of our algorithm, and we illustrate the experimental results we obtained by running Prim's algorithm on graph sizes ranging from 5,000 vertices to as many as 200,000 using up to 1024 processors.

2 Baseline Algorithm

We implemented a parallel Prim's algorithm adapted from its sequential version, distributing the matrix across processors. This implementation was also described by Deo and Yoo in [4] and Kumar in [2].

The graph is represented by an N by N adjacency matrix. Each cell (i, j) contains the weight of the edge from vertex i to vertex j. Since the graph is undirected, the matrix is symmetric. The matrix is split up by columns, each processor having N by N/P of the matrix (P = number of processors). Thus, each processor is "responsible" for N/P vertices.

To generate the symmetric matrix we use the following scheme: each processor generates edge weights for its elements below the diagonal using a pseudo-random function and sends a copy of each below-diagonal submatrix to the other processor that requires the submatrix. The edge weights are generated using a pseudo-random function that generates floating point numbers between 0 and 1 in a uniform distribution. This way the graph data stays consistent on any number of processors, since the edge weight depends on the (global) index of the vertices it connects.

The algorithm starts off by splitting up the graph adjacency matrix into N by N/P submatrices among P processors. Processor at index 0 (P_0) controls the flow of the algorithm. Vertex 0 is added as the root of the minimum spanning tree. The procedure of the algorithm then is as follows:

1. Each processor locally determines the closest vertex to the current MST in its data partition.

2. The closest vertex in the global context is then determined by a single processor accumulation operation - a MIN reduction on the edge weights of the closest vertices to the current tree in each processor's local matrix partition.

3. After obtaining the result of the reduction, each processor updates its data structures accordingly to now include the "winner" vertex v that had the shortest edge e to the current MST across all processors. Tree cost is updated to include the weight of e.

4. The process repeats until all the vertices have been added to the minimum spanning tree.

In order to eliminate the need of obtaining the reduction result on one (main) processor and then having it broadcast it to all other processors, we used MPI `Allreduce` function that stores the result of the reduction on each processor. This eliminates the need for extra communication of the broadcast operation.

2.1 Analysis of naive implementation of Prim's algorithm

Since the algorithm's structure involves a significant amount of communication (N reductions) and the computation is very fine-grained, there is a limited range of values of N where it is possible to achieve a speedup.

The following derivations use the following parameters:

Let N = problem size (vertices in the graph)
P = number of processors we use to run the algorithm
P_{min} = minimum number of processors that fit the N by N matrix in their memory
M = the amount of memory/processor
a = proportionality constant of a reduction operation (about 50 us)
b = computation proportionality constant (a few nanoseconds)

The algorithm consists of N computation phases (finding the closest vertex), each followed by a single reduction (checking which vertex is closest across all processors). We maintain an array of size N/P on each processor that contains the shortest edge to the current tree of each vertex for this processor's partition. The array is updated during each iteration and then the shortest edge is found by looping over its N/P entries. Thus the cost of finding the smallest edge to the tree is N/P (to update the array with a new vertex added) + N/P (to find the minimum edge) = 2N/P. The proportionality constant b encompasses this factor. The total execution time is given by:

$$T = total\ number\ of\ iterations * (cost\ of\ each\ iteration) = N * (reduction\ time + \frac{b * N}{P}) \quad (1)$$

When we increase the number of processors P, for a fixed problem size N, the second (computation) term decreases, but the reduction term increases (or at best, stays the same). In order for our algorithm to achieve speedup, the computation time has to dominate communication time. Thus we have:

$$Communication\ time < Computation\ time \quad (2)$$

$$N * reduction\ time < \frac{b * N^2}{P} \quad (3)$$

$$reduction\ time = a * log(P) \quad (4)$$

Combining equations 3 and 4 we get

$$a * N * log(P) \ < \ \frac{b * N^2}{P} \tag{5}$$

$$a * P * log(P) \ < \ b * N \tag{6}$$

We will simplify the equation by assuming $a \approx 1000 * b$ since in most modern supercomputers the reduction time is on the order of several 10s of microseconds and the computation step is usually on the order of a few nanoseconds. Thus, combining this simplification with equation 6 we obtain:

$$1000 * P * log(P) \ < \ N \tag{7}$$

The second condition is that the graph adjacency matrix must fit into the processors' memory. Each cell(i, j) in the matrix contains the weight of the edge connecting vertices i and j represented as a `double`, and thus taking up 8 bytes of memory. Therefore, in order for the graph to fit in the memory the following must be true:

$$N^2 \ < \ \frac{M * P_{min}}{8} \tag{8}$$

$$N \ < \ \sqrt{\frac{M * P_{min}}{8}} \tag{9}$$

If we assume that M = 1GB (as in Cray XT3 machine that we used for our experiments), we obtain:

$$N \ < \ 11585 * \sqrt{P_{min}} \tag{10}$$

Combining equations 7 and 10 yields the following constraints for N:

$$1000 * P * log(P) \ < \ N \ < \ 11585 * \sqrt{P_{min}} \tag{11}$$

Thus, using equation 11 we obtain: for $P = 1$ the constraint on N is $0 \ < \ N \ < \ 11,585$, for $P = 2$ it is $2,000 \ < \ N \ < \ 16,383$, for $P = 4$ it is $8,000 \ < \ N \ < \ 23,170$. It is therefore possible to obtain some speedup from running the algorithm on a problem of size N falling into those constraints. However, when the number of processors goes up to 16, the constraints become $64,000 \ < \ N \ < \ 46,340$ which is not possible. Thus, it is impossible to obtain speedup from running the algorithm on large problems on 16 or more processors. Our theory is indeed confirmed in our experiments, we are able to obtain small speedup on a graph of size 11,000, yet larger problems do not scale.

When a problem gets large, the communication time dominates, and thus, one can not attain any speedup. However, as the problems size gets large, the matrix cannot fit on one processor, so in order to solve the problem we *need* parallel execution. We will now explore this fact.

In scientific computations, large problems need to be processed fast. If the problem is too large to fit on one processor we require parallel execution. Our algorithm provides the means of finding minimum spanning trees of very large dense graphs. By splitting up the matrix among the

memories of 100s or 1000s of processors we are able to process graphs with as many as 500,000 vertices. Being able to process large graphs is the major advantage of using a parallel algorithm to compute an MST. However, due to communication time domination, we do not observe any speedup from using more processors. Thus, we focus on the efficiency of the algorithm. We will now describe a novel enhancement that we added to our parallel implementation of Prim's algorithm to significantly improve its efficiency.

3 Novel enhancement to the Algorithm

As the analysis in section 2.1 indicates, the best performance for a dense graph of size N is obtained by using the minimum number of processors $P_{min}(N)$ on which the problem will fit. Given this, is there a way to improve the efficiency of the algorithm on $P_{min}(N)$ processors? Note that the execution time is determined mostly by the number of reductions one needs to perform. Observing that the cost of reducing one number is about the same as reducing a small collection of numbers (say 16), we want to explore if we can reduce the total number of reductions, *by adding multiple vertices in each iteration.* The textbook description of this problem states that one can only add one vertex at a time [2] page 263. But as we show below, it is possible to add multiple vertices per iteration, on average, if we are willing to do some extra computation (an extra reduction in each iteration) and accept that in a small number of cases we may fail to add multiple vertices to the spanning tree. We will denote the number of vertices added each step as K for the rest of this paper. Several difficulties arise with this enhancement:

1. What vertices are we going to attempt to add in the current iteration and how are we going to find them in the distributed matrix?

2. It is difficult to add more than one vertex per iteration due to edge insertion order dependencies imposed by Prim's algorithm, so how are we going to go about doing it and still obtain a minimum spanning tree as a result?

We start by generating an array of K closest vertices to the tree in the current iteration. Because of the distribution of the matrix among the processor memories, we need to perform a global reduction to obtain an array of K globally closest vertices to the current MST. Since we need to keep track of edge-vertex pairs, none of MPI's simple built-in reductions will suffice, thus we implemented a user defined reduction that generates an array of K objects each containing a vertex index and associated edge weight for the K closest vertices to the current partial MST. Thus, after the reduction is complete, we have an array of K *globally* closest vertices to the current growing MST. In our implementation of the reduction, the elements in the resulting array are in non-decreasing order by their edge weights.

After obtaining a list of the K nearest vertices to the current partial MST, it is now time to determine which of those vertices and their associated edges can be safely added to the tree. We must choose edges to add while guaranteeing that we will eventually produce a minimum spanning tree of the whole graph. In order to find how many of the K vertices can be added to the MST during this iteration we perform a series of "checks" on each vertex. For each vertex we perform

the "checks" on we assume that the vertices preceding it in the K array have already been added to the tree. We iterate through the array and perform checks one vertex at a time, either admitting it to a "valid" list or rejecting it. We describe the "checks" as follows:

1. Check if any of the vertices added during this iteration have a shorter distance to the current vertex in question, if so the current vertex is invalid.

2. Check if any of the vertices added during this iteration have a shorter distance to a vertex *not in the tree* than the current vertex's distance. If so the current vertex is invalid.

If both of the checks pass, the vertex is considered "valid" and we proceed to iterate through the rest of the array. As soon as one invalid vertex is found, we stop the process. Each processor performs the checks on its local matrix partition. Thus, after each processor determines the number of valid vertices for its local partition, we need to perform another global reduction to determine the minimum number of vertices across all processors that passed all the checks and can be added to the tree. This time it is a built-in *MIN INT* reduction. After determining the number of vertices that globally passed all the checks, each processor updates its data structures with the new vertices in the MST and then generates a new array of K closest vertices. Similar to the original algorithm, this process repeats untill all the vertices are in the tree and the MST is constructed. The pseudo-code for the algorithm is as follows:

```
function primsMST(graph G){
   countOfverticesInTree = 0
   add vertex 0 to VerticesInTree
   while(verticesInTree<vertices in G){
      // on each processor
      array[K] = findKclosestVertices(VerticesInTree)
      globalArray[K] = AllReduce(array[K])
      // globalArray now contains K globally closest vertices to the tree
      K1 = determineValidVertices(globalArray)
      // K1=number of vertices valid for this processor
      K2 = Allreduce(K1)
      // K2 = number of globally valid vertices
      add globalArray[1..K2] to VerticesInTree, and update data structures
      increment the countOfVerticesInTree by K2
   }
}
```

We now present the proof of correctness of this enhancement to the algorithm.

3.1 Proof of correctness of the new method

Since Prim's algorithm produces a valid MST, we show that our algorithm performs its vertex insertions in the same order as the original Prim's algorithm does, and thus, produces the same valid MST. First we will consider one iteration of the algorithm.

We will formally define the problem as follows:

Let G = the dense graph we are processing with V = set of all vertices and E = set of all edges
V^* = set of vertices to be added in the current iteration,
sorted in increasing order of edge weights connecting the vertices to the current MST
V' = set of vertices in the current MST
$V'' = V \backslash V'$ i.e. set of all vertices not in the MST
v_i = vertex at index i
$e_{v_i v_j}$ = edge between vertices v_i and v_j
se_{v_i} = shortest edge between $v_i \in V^*$ and any $v \in V''$
e_{v_i} = edge of $v_i \in V^*$ at the time of generation of V^*

Our algorithm adds vertex v_i iff

1. $\forall v_i,\ v_j \in V^* | j < i,\ e_{v_i v_j} > e_{v_i}$ (assumed $v_j \in V'$ at the check time of v_i) (check 1)
2. $\forall v_i,\ v_j \in V^* | j < i,\ se_{v_j} > e_{v_i}$ (assumed $v_j \in V'$ at the check time of v_i) (check 2)

Assume both our algorithm and original Prim's algorithm start at the same point. We show that our algorithm added $v_0, v_1, ..., v_{K-1}$ to V' iff the original Prim's algorithm also added $v_0, v_1, ..., v_{K-1}$ to V' connected to the MST by $e_{v_0}, e_{v_1} ... e_{v_{K-1}}$ in its consecutive iterations.

Proof by strong induction: Assume that $\forall v_i \in V^* \mid i < K-1$ our algorithm added v_i (connected by e_{v_i}) to the MST in the same order as the original Prim's algorithm. Consider v_{K-1}. Show that our algorithm adds v_{K-1} connected by $e_{v_{K-1}}$ to the MST iff the original Prim's algorithm does as well.

- If our algorithm added v_{K-1} connected by $e_{v_{K-1}}$ to V' then by (1) and (2) $e_{v_{K-1}}$ is the shortest edge to the current MST, since V^* is sorted and there is no edge in the current MST (with already $v_0, v_1, ..., v_{K-2} \in V'$) that is shorter than $e_{v_{K-1}}$. Thus, the original Prim's algorithm would pick v_{K-1} connected by $e_{v_{K-1}}$ for its next addition to the MST.

- If Prim's algorithm added v_{K-1} connected by $e_{v_{K-1}}$ to the MST, then $e_{v_{K-1}}$ is the shortest edge among all edges "coming out" of the MST (with already $v_0, v_1, ..., v_{K-2} \in V'$). Thus, check 1 and check 2 of our algorithm will pass on vertex v_{K-1}, since all se_{v_j} and $e_{v_{K-1} v_j}$ will be greater than $e_{v_{K-1}}$ ($j < K-1$) since $e_{v_{K-1}}$ is the globally shortest edge from the current MST (containing $v_0, v_1, ..., v_{K-2}$), and thus the checks will pass and our algorithm would add v_{K-1} to the MST.

By the strong induction argument our algorithm adds vertices in the same order connected by the same edges to the growing MST as the original Prim's algorithm in one iteration. Since the process repeats for each iteration until all vertices in G are in the MST, in each step the shortest edges are added to MST. Thus the algorithm constructs a minimum spanning tree of G.

An illustration of one iteration for K=5 of the new method is shown in Figure 1. The table on the right at each step represents V^* for this iteration. E denotes e_{v_i}, SE denotes se_{v_i} - (the shortest

edge from a vertex in the tree to a vertex not in the tree), and the circled vertex is the one being considered (checked) in the current step. The MST contains vertex F, and we wish to add the 5 vertices at once to the MST. (Assume that there are other vertices in the graph, but are not shown). In step 1 the MST contains vertex F and the spanning tree cost is 0.0. Vertex C is added in this step since it has the shortest edge to F in the whole graph. In step 2 the MST now contains vertices F and C and vertex B is under consideration. B is added since $(e_{BC} = 5.7) > (e_B = 1.5)$ and $(se_C = 2.9) > (e_B = 1.5)$. Similarly in step 3 vertex A is added since $e_A = 1.7$ is less than e_{AC}, e_{AB}, se_C, and se_B. In step 4, however, vertex E does not pass the checks since $(se_A = 0.3) < (e_E = 2.6)$ (the edge from A to D is 0.3) and, thus it not is added to the tree in this iteration. These vertices are actually added to the tree if the program is run on 1 processor. On multiple processors, the valid number of vertices in V^* is first locally determined on each processor's local matrix partition and then, via a global *MIN INT* reduction, the minimum number of vertices that globally passed the checks on all processors is determined, and those vertices are added to the MST with their corresponding edges.

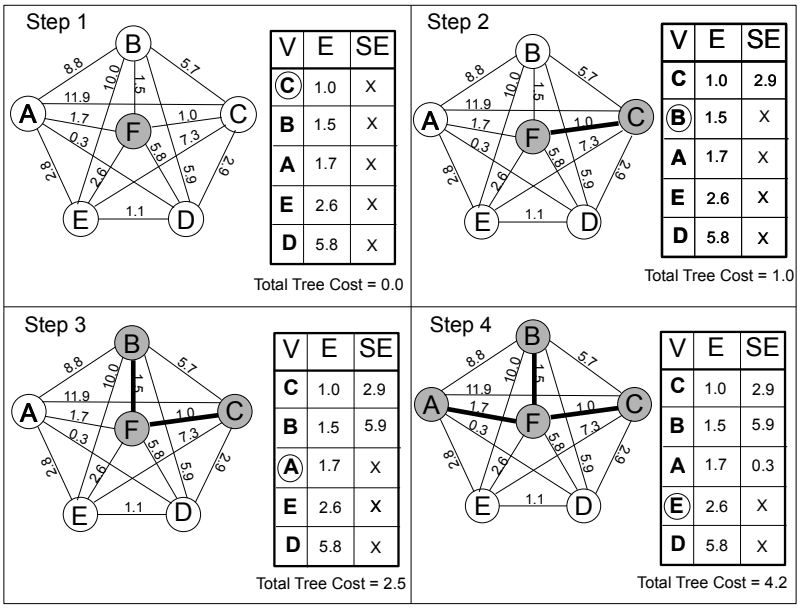

Figure 1: Example of one check iteration

3.2 Analysis of the new method

We now show a brief analysis of our new technique. We will use the same naming convention as with the analysis of the original implementation, we only add variable K = number of vertices added per iteration of the algorithm.

With the technique of adding multiple vertices per iteration the computation time increases as

follows:

$$Computation\ time\ =\ Time\ to\ find\ K\ closest\ vertices\ +\ check\ time \tag{12}$$

The time to find K closest vertices for the whole algorithm is $\frac{N^2}{P*K}$ since we need to process all vertices in the graph, but we are doing N/K iterations instead of N. The time to perform checks consists of finding shortest distance to K vertices for each processor's N/P vertices N/K times. Thus, we obtain:

$$\frac{N^2}{P*K}\ +\ K*\frac{N}{P}*\frac{N}{K} \tag{13}$$

$$\frac{N^2}{P*K}\ +\ \frac{N^2}{P}\ =\ \frac{N^2+K*N^2}{K*P}\ =\ \frac{(K+1)*N^2}{K*P} \tag{14}$$

The communication time, decreases by a factor of $\frac{2}{K}$, since we need to perform 2 reductions per iteration and there are a total of N/K iterations.

$$Communication\ time\ =\ \frac{2N}{K}*log(P) \tag{15}$$

Thus, the computation time increases by a factor of $\frac{K+1}{K}$ and the communication time decreases by a factor of $\frac{2}{K}$. Thus, the total number of iterations decreases substantially by a factor of K and the computation time increases by a small factor of $\frac{K+1}{K}$. This allows us to obtain significant performance improvement for running the algorithm on large graphs with varying values of K. We present our experimental results in the next section.

4 Experimental Results

In this section we describe the experimental results we obtained by running our proposed algorithm on graphs of various sizes on the Cray XT3 Supercomputer. The only publication we found that contained experimental results on parallel Prim's algorithm was Barr et. al ([3]). In their experiments, Barr et. al. investigated the performance of parallel Kruskal's, Sollin's and Prim's algorithms on graphs of various densities on 1 to 10 processors. They confirmed that Prim's algorithm performs best on dense graphs. In our research we only focus on dense graphs, thus using the best algorithm for the problem. In Barr's experiments, Prim's algorithm obtained speedup of 4.4 on 10 processors. Our best speedup obtained for small graphs is 2.91, however, the comparison to Barr's results is not very relevant since the XT3 Supercomputer is much faster than the parallel machines Barr used in his experiments in 1989.

Figure 2 and Table 1 show a decrease in the execution times and thus resulting speedups for our algorithm on up to 16 processors. The graph sizes are given in number of vertices. The speedup was obtained by dividing the best sequential time by the parallel time.

When the algorithm is run on graphs of size larger than 11,000, the matrix does not fit on one processor (Cray XT3 1GB of memory/node) and thus we need parallel execution. We do not observe any significant speedup from increasing the number of processors our algorithm is

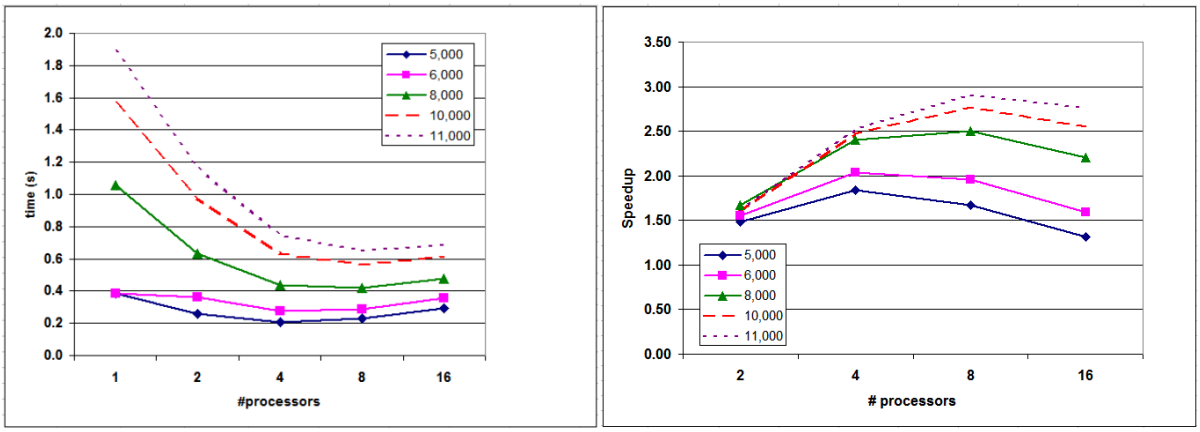

Figure 2: Parallel Prim's performance on small graph sizes on Cray XT3. Execution time (left) and obtained speedup (right)

# processors	5,000 vert	6,000 vert	8,000 vert	10,000 vert	11,000 vert
1	0.385(1)	0.385(1)	1.056(1)	1.563(1)	1.887(1)
2	0.259(1.48)	0.363(1.56)	0.630(1.68)	0.977(1.60)	1.174(1.61)
4	0.209(1.84)	0.278(2.04)	0.439(2.40)	0.632(2.47)	0.749(2.52)
8	0.231(1.67)	0.288(1.96)	0.422(2.50)	0.565(2.76)	0.648(2.91)
16	0.293(1.31)	0.354(1.60)	0.479(2.20)	0.613(2.55)	0.684(2.76)

Table 1: Execution time in seconds and speedup in parentheses of Prim's algorithm on small graph sizes (data from Cray XT3)

run on. Figure 3 illustrates an example of it for N=25,000, 50,000 and 100,000. Thus, the best performance on large graphs is obtained by running the original parallel Prim's algorithm (adding 1 vertex per iteration) on the smallest number of processors that fit the graph matrix. Once we attempt to increase the number of processors, the communication overhead dominates over the computation on each processor and, thus, no speedup is observed.

However, we observe significant performance improvement by running our algorithm with the new enhancement of adding multiple vertices per iteration. We show data for graphs of sizes 25,000, 50,000, 100,000, and 200,000 on 32-1024 processors. Our experimental results are summarized in Figure 4. Each problem size is run on the smallest number of processors that fits it, and the corresponding values are given in the legend.

We show the results for running the algorithm with K=2 and increasing it up to 32. For 25,000 vertex graph the performance improvement is pretty insignificant: the K=4 case is only 1.05 times faster than K=2 case on 32 processors (2.64 vs 2.77 seconds). However, when we increase the graph size to 50,000 we observe a significant performance improvement: for 50,000 graph the

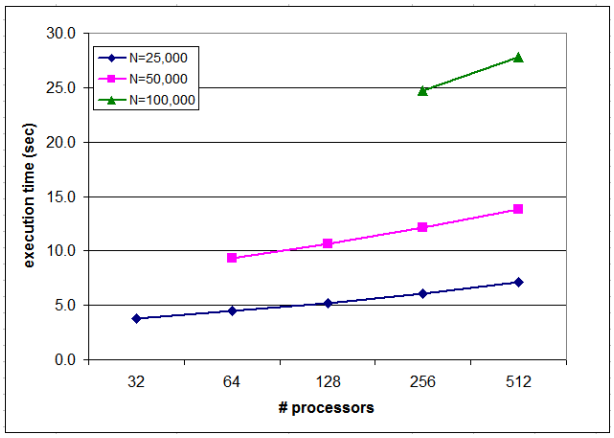

Figure 3: Parallel Prim's execution time (seconds) on large graphs with increasing number of processors

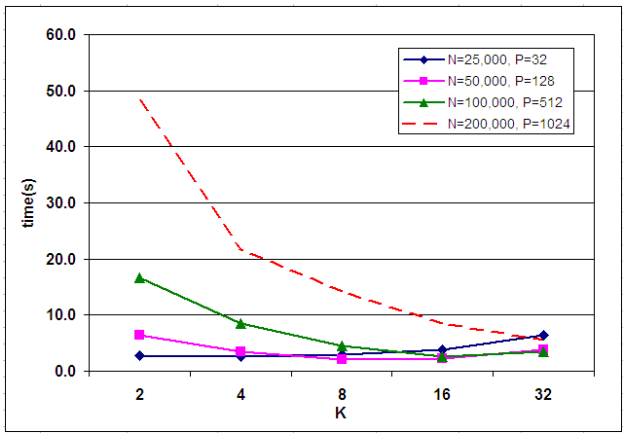

Figure 4: Parallel Prim's execution time (seconds) on large graphs with increasing K value

K=8 case runs 3.1 times faster than K=2 algorithm on 128 processors (2.06 seconds vs. 6.45 seconds). Similarly for problem of 100,000 vertices we get a 6.37 fold improvement with K=16 case over K=2 case on 512 processors (2.61 seconds compared to 16.6 seconds). Finally, for graph of size 200,000 vertices the performance improves by 8.7 for running the algorithm with K=32 (48.0 seconds vs. 5.5 seconds). This significant improvement is due to our novel enhancement reducing the number of reductions/iterations that needs to be preformed. Instead of performing N iterations, we perform N/K iterations in the best case. However, our experiments show there is a limit to the optimal K value, after which we observe execution time increase (see Figure 5). Thus, for each graph size we can successfully add up to that optimal K number of vertices per iteration, once we attempt to add more vertices, the checks tend to fail, and thus work is wasted generating the K array.

Our experiments show that we can obtain small speedups on graphs of small size, yet when we attempt to process large graphs, no speedup is attainable because of communication overhead. However, our new technique of adding multiple vertices per iteration yields significant performance improvement on large graphs. Thus, we are able to process large graph problems efficiently.

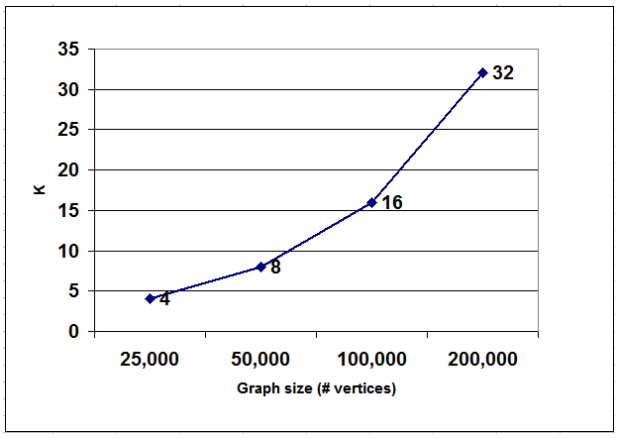

Figure 5: Optimal value of K for each graph size that yields most significant performance improvement

4.1 Future Work

In our experiments, we observed that the communication overhead dominates computation time when running the algorithm on large graphs. It is due to the idle time during each reduction operation between each phase of the algorithm. We illustrate one iteration of the algorithm in Figure 6.

The iteration starts with each processor determining K closest vertices in its local matrix partition. Reduction 1 determines K globally closest vertices to the current tree. This is followed by the checks function to determine valid vertices that can be added. Reduction 2 determines the minimum number of valid vertices across all processors, that number of vertices is added to the

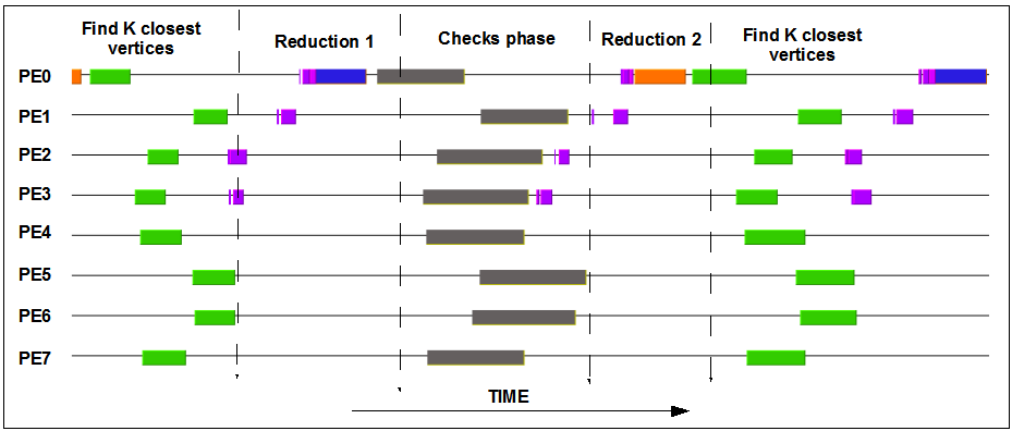

Figure 6: Timeline of one iteration of Prim's algorithm (adding K vertices/iteration version).

tree and the process repeats. It is easy to notice that a large portion of each CPU time is idle, while the reduction result is being computed. To gain some improvement we could overlap the idle time during the reduction with some useful work. To accomplish this we would need asynchronous message passing. We have experimented with this idea using Charm++ programming language that offers asynchronous message passing among parallel objects. Charm++ is a portable parallel programming language based on the migratable object programming model and resultant virtualization of processors. Using the asynchronous messages to our advantage we overlapped some useful work with the communication idle time: while the processors are waiting for the Reduction 2 result (number of vertices that are valid and can be added in the current iteration) we update the data structures as if all K vertices passed the checks. In case of a success, all K vertices are added, we can skip the data structure update in the next function thus saving computation time. We have not observed significant improvement with this technique, however we plan to experiment more with this idea and get more promising results taking advantage of Charm's asynchronous message passing.

5 Summary and Conclusion

We analyzed effective parallelization of Prim's algorithm for dense graphs. A simple analysis showed that it is not possible to attain strong speedups for graphs beyond a small size. Further, for a given graph with N vertices, the best performance is attained on the minimum number of processors, $P_{min}(N)$, on which the graph will fit. Since we still wish to effectively parallelize this computation, we looked for opportunities for improving the efficiency of the algorithm on $P_{min}(N)$ processors. A novel technique we identified attempts to add multiple vertices to the tree in each iteration. A checking step with an additional reduction is then necessary. Although it may fail in some iterations, we showed, empirically, that this strategy leads to significantly improved performance. In one case, the performance improvement was almost ten-fold.

We presented performance data of our implementation on PSC's BigBen, a Cray XT3 machine.

Emprical data upto 1024 processors and for N up to 200,000, validates the benefits of our strategy. We think further improvements can be obtained on machines and programming systems that support asynchronous reductions, by off-loading some portion of the computation during one of the reduction phases in each iteration. In any case, we believe that we have presented one of the most efficient parallel Prim's algorithm and its implementation to date.

References

[1] R.L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, January 1995.

[2] G. Karypis A. Grama, A. Gupta and V. Kumar. *Introduction to Parallel Computing*. Addison Weslesy, second edition, 2003.

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

[4] N.Deo and Y.B. Yoo. Parallel algorithms for the minimum spanning tree problem. In *International Conference of Parallel Processing*, pages 188–189, August 25-28 1981.

[5] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, 1982.

[6] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Comput. Surv.*, 16(3):319–348, 1984.

[7] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.*, 66(11):1366–1378, 2006.

[8] Sun Chung and Anne Condon. Parallel implementation of boruvka's minimum spanning tree algorithm. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, page 302, April 1996.

[9] F. Dehne and S. Gotz. Practical parallel algorithms for minimum spanning trees. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 366, Washington, DC, USA, 1998. IEEE Computer Society.

[10] Donald B. Johnson and Panagiotis Metaxas. A parallel algorithm for computing minimum spanning trees. *J. Algorithms*, 19(3):383–401, 1995.

[11] Micah Adler, Wolfgang Dittrich, Ben Juurlink, Miroslaw Kutylowski, and Ingo Rieping. Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 27–36, New York, NY, USA, 1998. ACM Press.

[12] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[13] Y. K. Dalal. A distributed algorithm for constructing minimal spanning trees. *IEEE Trans. Softw. Eng.*, 13(3):395–405, 1987.

[14] David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005.

[15] H. Ahrabian and A. Nowzari-Dalini. Parallel algorithms for minimum spanning tree problem. *International Journal of Computer Mathematics*, 79(4), 2002.