A Parallel Branch-and-Bound Algorithm for Finding Steiner Minimal Trees

Steven A. Spoon

May 1996

Abstract

This paper focuses on the final step of a method studied by Harris at Clemson University for finding the Steiner Minimal Tree of a given set of points. Specifically, the problem this paper faces is as follows: Given a set of points, a set of trees over those points, and a precomputed "compatibility" matrix relating those trees, find the set of trees which forms the minimum cost spanning tree for the graph. The matrix limits what solutions are possible, marking certain pairs of subtrees to be "incompatible." This paper studies the use of parallel exhaustive search techniques to solve this problem.

First, this paper describes a sequential algorithm which uses an exhaustive depth-first search. The algorithm is implemented and timed. Second, this paper describes parallel depth-first algorithm. The parallel algorithm distributes portions of the exhaustive search space to worker nodes on a first-come, first-serve basis; the worker nodes then independently use a depth-first search on each of those individual portions, one worker node per portion. This algorithm, too, is implemented and is timed for a variety of input parameters. Overall, for reasonable choices of parameters, the parallel algorithm achieves good speedup.

1 Problem Definition

Consider any set of points in the plane. It is easy to construct weighted graphs whose nodes are a subset of these points, and whose edges are line segments joining those points. The weight (also "cost" or " length") of each edge is simply the length of the line segment it represents, i.e., the distance between two of the given points. If such a constructed graph is connected, has no loops, and includes all of the given points, then it is called a *spanning tree* of the set of points. If a spanning tree furthermore has the least possible sum of edge lengths, it is called the *mimimum spanning tree* of these points. There are several known algorithms for finding the minimum spanning tree of any given set of points.

Interestingly, it is actually possible to add new points to the set, and for the minimum spanning tree of the new set of points to have a smaller sum of lengths than the minimum spanning tree of the original set. A *Steiner Minimal Tree* is the tree whose nodes include (but are not limitted to) the given set of points, and whose sum of edge lengths is minimized. Any points within the Steiner Minimal Tree that are not in the original set of points are called *Steiner Points*. This paper focuses on finding the Steiner Minimal Tree of a set of points, given some additional useful information from previous work by Dr. Fred Harris at Clemson University.

Harris's work accomplishes three major things that this paper uses. First, Harris describes an algorithm for finding the location of all of the Steiner Points that need to be added to a given set of points. Second, Harris has an algorithm which chooses several trees with the guarantee that the Steiner Minimal Tree can be formed by a collection of these smaller trees. Finally, Harris has an algorithm which generates a "compatibility matrix" for the trees, such that if the (i, j) entry of the compatibility matrix is 0, then tree i and tree j are not both in the solution.

This paper describes a parallel depth-first algorithm to determine which of trees given by Harris's algorithm together form the Steiner Minimal Tree of a set of points.

2 Terms

The following terms are used throughout this paper:

Steiner Minimal Tree the smallest tree whose nodes include a given set of points in the plane.

compatibility matrix a matrix generated by Harris's algorithms which define the "compatibility" of any two distinct trees in the problem set. There are three values a compatibility can take on:

- 1 linked
- 0 incompatible
- -1 neither linked nor incompatible

The compatibility of a tree to itself is undefined.

incompatible Two trees in the problem set are said to be "incompatible" if their entry in the compatibility matrix is a 0.

linked Two trees in the problem set are said to be "linked" if their entry in the compatibility matrix is a 1. Any pair of linked trees contain exactly one node in common.

problem set a set of nodes, a set of Steiner points, a set of trees, and a compatibility matrix. The trees consist of nodes in the problem set plus Steiner Points generated by Fred Harris. Each node and each Steiner point is contained in at least one tree.

complete set of trees a collection of trees from the problem set that cover every node, where all of the trees are connected into a larger tree, and where none of the trees are incompatible.

prefix a set of trees from the problem set which are all connected into a larger tree, where none of the trees are incompatible. A prefix which covers all of the nodes in the problem set is a complete set of trees.

3 Sequential Algorithm

Before developping a parallel algorithm, a sequential algorithm was developped. The sequential algorithm performs a standard depth-first search over a certain search tree, with pruning. The following sections describe that search tree, specify when pruning is possible, and finally give complete pseudocode.

An implementation and some timing results are described in the Observations section.

3.1 The Search Tree

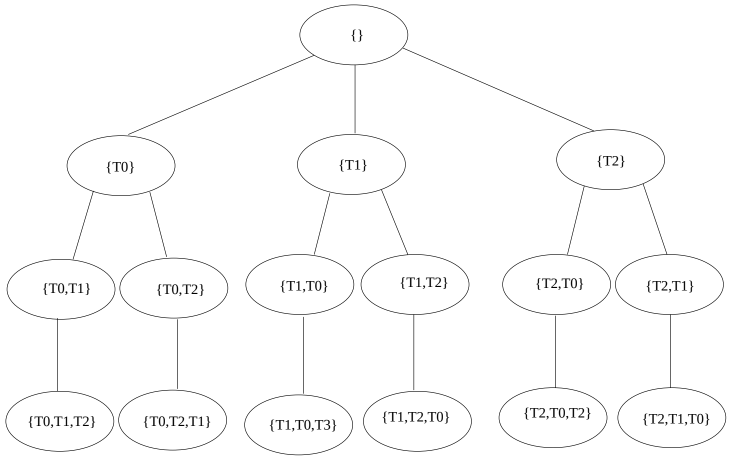
This section describes the search tree that the sequential algorithm searches.

Each node in the search tree corresponds to one "prefix," where a prefix is a set of trees from the problem set. The root of the search tree corresponds to the empty set, and each leaf corresponds to the set of all trees in the problem set. The mapping from search tree nodes to prefixes is many to one.

Each node other than the root has the property that its prefix is the same as its parent's prefix except that it has one additional tree. Furthermore, if a new prefix P can at all be formed by adding one tree to the prefix of node N, then exactly one of N's children corresponds to prefix P.

For example, if a node X contains the prefix {T1, T5, T7}, then node Y with the prefix {T1, T5, T7, T8} could be a child of node X, but node Z with the prefix {T1, T3, T7, T8} could not be. Node Y is not necessarily a child of node X, but there is guaranteed to be one child of X which contains the same prefix as Y.

For a more complete example, here is the search tree for a tiny problem set containing 3 trees:



3.2 Pruning

The complete search tree desribed above grows very quickly as the number of trees increases. Furthermore, the search tree is highly redundant – almost every prefix appears in the tree many times.

Luckily, it is not necessary to search the entire search tree – a few simple pruning rules drastically reduce the number of nodes visited. A subtree of the search tree is not searched by the sequential algorithm under discussion if the head of that subtree meets any of the following criteria:

1. The node's prefix is complete.
2. The cost of the prefix in the node is larger than the cost of some complete set that has already been found.
3. The prefix in the node is not connected.
4. Two of the trees in the node's prefix are incompatible.
5. The node's prefix has been studied before.

These pruning checks, taken alone, would be fairly inefficient, especially the check for connectedness. However, the algorithm can take advantage of the depth-first order of the search, and maintain certain information in order to simplify the checks. Using this extra information, all of the above checks can be performed in constant time. The extra information must be updated whenever the search decends a level into the search tree, and each update requires time of order n, n being the number of trees in the problem set.

3.3 Variables for Sequential Algorithms

The next section gives a pseudocode explanation of the sequential algorithm for finding Steiner Minimal Trees; here is a brief description of most of the variable used in that pseudocode.

S a prefix. It corresponds to the current node being visited in the search tree.

COST the sum of the costs of the trees in S.

vectors four sets that are maintained as the algorithm progresses. These form the "extra information" mentioned in the Pruning section. The four vectors are:

- **TREES_LINKED** the set of trees that have a 1 compatibility with at least one of the trees in S.
- **INCOMPATS** the set of trees that have a 0 compatibility with at least one of the trees in S.
- **HISTORY** the set of all trees that, if added to S, will produce a new set of trees we have not studied before.
- **NODES_CONTAINED** the set of nodes that are contained in at least one of the trees in S.

NEXT the number of the next tree the algorithm is currently considering adding to S.

BEST_SOLUTION the cheapest complete set of trees yet found.

BEST_COST the cost of the cheapest complete set of trees yet found.

3.4 Pseudocode

Here is a pseudocode description of this paper's sequential algorithm for finding the Steiner Minimal Tree, given a dataset resulting from Harris's algorithms. A text description follows.

Within the pseudocode, each of the trees in the problem set is given a number between `0` and `(NUM_TREES - 1)`, and the notation `tree[i]` refers to the tree whose number is `i`.

```
S := <empty>
initialize the 4 vectors
COST := 0
NEXT := 0
BEST_COST := INFINITY
DONE := False
while not DONE
    while (S is not complete) and (NEXT < NUM_TREES)
        test tree[NEXT] against the 4 vectors and COST
        if tree[NEXT] passes
            add tree[NEXT] to S
            update the HISTORY vector
            push the 4 vectors and COST
            update the other 3 vectors and COST
            NEXT := 0
        else
            NEXT := NEXT + 1
        end if
    end while
    if (S is complete) and (COST < BEST_COST)
        COST := BEST_COST
        BEST_SOLUTION := S
    end if
    if (S is not empty)
        NEXT := the last element added to S
        NEXT := NEXT + 1
        remove the last element added to S
        pop the 4 vectors and COST
```

```
        else
            DONE := True
        end if
    end while
```

4 Parallel Algorithm

The parallel algorithm studied in this paper is a straightforward extension of the sequential algorithm described in the last section. The parallel algorithm traverses the same search tree as the sequential algorithm, but tries to distribute the work among several processors. Visually, a single "host" node searches the top of the search tree and doles out slices of the rest of the tree to "children" nodes.

The parallel algorithm starts with a "host" node searching the problem search tree. Whenever the host node's traversal reaches a certain predefined depth, the host stops descending and passes prefix within the current node in its search to a "child node." The host then backs up while the chosen child node searches the subtree beneath its given prefix.

When the host's search reaches the predefined depth again, it passes the prefix within the current node to a different child node. If the host is fast enough, it can keep several children nodes busy working on different prefixes, thus achieving parallelism.

4.1 Description of Communication

The host communicates with each of the children; the children never communicate directly with each other.

After initialization, communication between host and child is formed of a sequence of messages. Each message is one of these five types:

MSGT_WORK_PKT This type of message is sent from the host to a child. It gives the child a "work packet" to add to its queue of work. This message includes:

a) the information needed to work on the prefix. Interestingly, the prefix itself need not be included. This information includes the cost of the prefix, the 4 vectors corresponding to the prefix, and the cost of the cheapest solution yet found by any child. This information is collectively referred to as a "work packet".

b) a unique id number to describe this work packet. The child will use this id number when it reports back to the host. There is a one-to-one correspondence between id numbers and prefixes that are distributed to children nodes.

MSGT_DONE This type of message is sent from the host to a child. It tells the child that no more prefixes will be distributed. After the child finishes with the work packets in its queue, it should send a MSGT_SUMMARY message back to the host and then terminate.

MSGT_NO_SOL This type of message is sent from a child to the host. It tells the host that it could not find any useful, valid solution to one of the prefixes it was given. The message includes the id number of the prefix it could not solve and the id number of the child.

MSGT_FOUND_SOL This type of message is sent from a child to the host. It tells the host that it found a solution to one of the prefixes it was given. The message includes the id number of the prefix, the solution itself, and the id number of the child.

MSGT_SUMMARY This is the last message a child sends to the host before the child terminates. It contains summary statistics on the work done by the child during its lifetime, and the id number of the child. The information in this message is not important to the proper functioning of the algorithm, but is useful for analysis.

4.2 Initialization

The host passes two pieces of information to each child when program starts up:

1. the name of the file that holds the problem set.
2. a unique id number for the child. This id number is how the host distinguishes among the children nodes.

4.2.1 Prefix Generators

The core of the computations performed by the host node are performed within a "prefix generator." A prefix generator is simply encapsulated version of the sequential algorithm described above, with two additional properties:

1. A prefix generator is able to stop at any point and save its state, ready to continue at a later point in time.
2. Whenever a prefiex generator is searching and finds a prefix of a given depth, it stops and saves its state.

In a program, a prefix generator PREFGEN can be accessed in the following manners:

PREFGEN.init("datafile", DEPTH) this loads a problem set from file "datafile" and initializes the generator. The generator is prepared to generate prefixes of length DEPTH.

PREFGEN.next_prefix this finds the next prefix of size DEPTH, if there are any, and returns it. If there are no more prefixes, the return is undefined, and the following flag is set to TRUE.

PREFGEN.done this flag is FALSE under normal operation; the only time it becomes TRUE is when all possible prefixes have been generated.

The purpose of a prefix generator is to find all possible prefixes of a given length. These prefixes are the ones that will be farmed out to children nodes in the parallel algorithm.

4.3 Child Algorithm

```
Download DATA_FILE_NAME and ID_NUM from host
Load problem set from file "DATA_FILE_NAME"
WORK_QUEUE := <empty>
DONE := FALSE

repeat
    while (WORK_QUEUE is empty) or (an incoming message is pending)
        Receive a message MESSAGE from host
        case MESSAGE.type
            MSGT_WORK_PKT:
                add the prefix in MESSAGE to WORK_QUEUE
                if MESSAGE.best_cost_global > BEST_COST_YET
                    BEST_COST_YET := MESSAGE.best_cost_global
                end if

            MSGT_DONE:
                DONE := TRUE
        end case
    end while

    if WORK_QUEUE is not EMPTY
        put a prefix from WORK_QUEUE into CUR_PREFIX
        do a depth-first search starting with CUR_PREFIX
        if a solution is found
            return a MSGT_FOUND_SOL message to the host
        else
            return a MSGT_NO_SOL message to the host
        end if
    end if

until (DONE) and (WORK_QUEUE is empty)


build and send a MSGT_SUMMARY message to the host
```

4.4 Host Algorithm

```
load DATA_FILENAME into PREFGEN
BEST_COST_YET := INFINITY

spawn NUM_CHILDREN children nodes

for each child CHILD
send DATA_FILENAME to CHILD
send CHILD.ID to CHILD
endfor

for N := 1 to PREF_CACHE_SIZE
    for ID := 1 to NUM_CHILDREN
        NEXT_PREF := PREFGEN.next_prefix
        send NEXT_PREF to child[ID]
        add NEXT_PREF to CHILD_DATA(ID).prefix_list
    endfor
endfor


while not NEXT_PREF.done
    wait for an incoming message MESSAGE
    if (MESSAGE.type = MSGT_FOUND_SOL) and (MESSAGE.cost < BEST_COST_YET)
    then
        BEST_COST_YET := MESSAGE.cost
        BEST_SOL_YET := MESSAGE.solution
    endif
    remove MESSAGE.prefix from CHILD_DATA(MESSAGE.child_id).prefix_list
    NEXT_PREF := PREFGEN.next_prefix
    send NEXT_PREF to MESSAGE.child_id
    add NEXT_PREF to CHILD_DATA(MESSAGE.child_id).prefix_list
endwhile
```

```
    while there are still outstanding packets
    do
    wait for an incoming message
    store that message in MESSAGE
    if (MESSAGE.type = MSGT_FOUND_SOL) and (MESSAGE.cost < BEST_COST_YET)
    then
        BEST_COST_YET := MESSAGE.cost
        BEST_SOL_YET := MESSAGE.solution
    endif

    remove MESSAGE.prefix from CHILD_DATA(MESSAGE.child_id).prefix_list
endwhile
```

5 An Extension to the Parallel Algorithm

In parallel with the author, Jennifer Ludwick of Clemson has been working on the Steiner Minimal Tree problem. Instead of a branch and bound approach, she has been studying an approach using a parallel genetic algorithm (PGA).

For larger problem sizes, Ludwick's PGA performs much more rapidly than the author's parallel branch and bound (PB+B) algorithm. To balance this, Ludwick's algorithm usually does not find the optimal Steiner Tree for these same large problems.

One wonders whether the PB+B algorithm could be used to improve on the solutions generated by Ludwick's PGA.

Here is an algorithm we have tried:

1. Run Ludwick's PGA and produces several very good, but in general not optimal, solutions.
2. Extract the set of trees S_0 which are common to all of those solutions.
3. Run the PB+B algorithm, but restrict it to considering solutions which are a superset of S_0.

The restriction mentioned in step 3 can be achieved with just two modification to the PB+B algorithm from the last section. First, instead of beginning the search from the top of the search tree, the algorithm is modified to begin its search from a node corresponding to S_0. This is accomplished by initializing the host node's prefix generator with S_0 instead of with the empty set.

Second, before a set of trees can be accepted as a potential, an additional check needs to be made: the new PB+B must check that the set is "connected"; that is, its element trees form a connected graph. The old PB+B was able to traverse the search tree in such an order that every set of trees considered was connected; the new PB+B, however, could be forced to start from a set S_0 whose trees are not connected, and thus the algorithm could search many unconnected sets.

This second requirement leads to an interesting point: even though the new PB+B is restricted to traversing a smaller portion of the search tree, it does not necessarily search a smaller number of sets. The reason is that, though many sets of trees are ruled out by the restriction, potentially many new sets of unconnected trees are searched that would not have searched in the old PB+B algorithm.

To face this point, the new algorithm can be slightly modified so that it can never search a set it did not search in the old algorithm. One step needs to be inserted in the previous algorithm:

1. Run Ludwick's PGA and produces several very good, but not optimal, solutions.
2. Extract the set of trees S_0 which are common to all of those solutions.

2.5. Choose the largest subset S_1 of S_0 which is connected.

3. Run the PB+B algorithm, but restrict it to considering solutions which are a superset of S_1.

6 Observations

Both the sequential and parallel algorithms described were been implemented and timed. The algorithms were implemented using `gcc` compiler on Sun Sparcs. The operating system was Solaris 2.4. The parallel algorithm used PVM ("Parallel Virtual Machine"), version 3, for communication over an ethernet network connecting the Sparcs.

6.1 Sequential Algorithm

The sequential algorithm only has one input: the dataset. Here are the results of running 5 different data through the sequential program:

Dataset	Trees	Nodes	Time (seconds)
data.10.31	46	22	40.340000
data.01.27	58	28	3546.929932
data.10.8	60	21	1254.520020
data.07.25	71	26	37946.648438
data.06.29	85	27	134680.625000

6.2 Parallel Algorithm

The parallel algorithm has the following four parameters:

- Dataset to solve
- Number of children nodes
- Depth of prefixes to send to children
- Maximum number of prefixes a child could have at once ("buffer size")

Interestingly, early trial runs showed that the host node consumes only a tiny amount of the potential computing power of a processor. After these early runs, the host process was always forced to share a processor with one of the children nodes. Thus, in all of the data given, the number of children processors is the same as the total number of processors.

The following tables and graphs summarize the parallel runs made.

Dataset data.01.27, 4 nodes								
Prefix Depth	Buffer Size	Host CPU	Children times				Total Time	Speedup
1	1	0.53	887	1212	675	638	1215	2.92
1	2	0.58	912	1303	571	629	1304	2.72
1	3	0.53	953	1375	476	624	1376	2.58
1	4	0.61	1019	1536	472	687	1541	2.3
1	5	0.53	1026	1588	574	647	1593	2.23
2	1	1.02	875	867	1135	868	1140	3.11
2	2	0.95	915	887	1017	890	1017	3.49
2	3	0.90	855	779	763	1169	1169	3.03
2	4	0.99	1026	811	753	786	1026	3.46
2	5	0.86	1060	842	682	804	1061	3.34
3	1	1.89	839	838	839	855	855	4.15
3	2	1.98	837	848	837	837	850	4.17
3	3	1.95	842	837	836	847	847	4.19
3	4	1.85	962	790	814	790	963	3.68
3	5	2.16	882	820	833	821	882	4.02
4	1	4.40	878	879	881	879	881	4.03
4	2	4.24	963	964	963	974	974	3.64
4	3	4.30	963	959	981	959	981	3.62
4	4	4.15	972	972	972	986	987	3.59
4	5	4.09	969	970	969	989	989	3.59
5	1	8.70	981	980	981	980	981	3.62
5	2	8.89	977	977	978	977	977	3.63
5	3	8.80	988	988	989	988	989	3.59
5	4	9.16	996	996	996	997	997	3.56
5	5	9.28	1017	990	990	991	1017	3.49
6	1	17.61	1008	1006	1008	1007	1008	3.52
6	2	17.99	1042	1042	1043	1042	1042	3.4
6	3	19.22	1033	1033	1033	1032	1034	3.43
6	4	18.11	1025	1026	1026	1026	1026	3.46
6	5	18.53	1015	1016	1015	1014	1016	3.49
7	1	35.30	1048	1048	1048	1049	1049	3.38
7	2	36.31	1052	1052	1052	1052	1053	3.37
7	3	38.05	1077	1078	1079	1078	1079	3.29
7	4	36.80	1062	1061	1062	1061	1062	3.34
7	5	37.22	1055	1055	1055	1056	1057	3.36
8	1	80.37	1161	1161	1160	1161	1161	3.06
8	2	81.71	1159	1159	1159	1159	1159	3.06
8	3	82.59	1190	1190	1190	1191	1191	2.98
8	4	81.74	1158	1156	1158	1158	1158	3.06
8	5	82.69	1211	1211	1212	1212	1212	2.93
9	1	195.69	1480	1480	1478	1479	1480	2.4
9	2	199.34	1607	1607	1607	1608	1608	2.21
9	3	200.06	1608	1607	1609	1608	1609	2.2
9	4	199.48	1618	1618	1618	1618	1619	2.19
9	5	197.39	1568	1568	1567	1568	1568	2.26

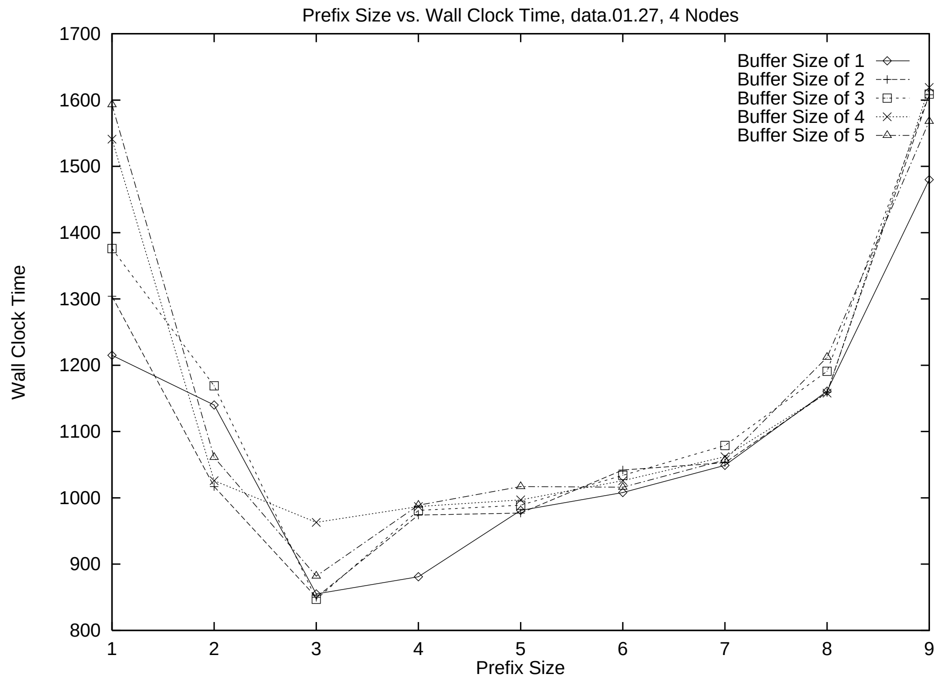
Dataset `data.01.27`, 8 nodes												
Prefix Depth	Buffer Size	Host CPU	Children times								Total Time	Speedup
1	1	0.54	887	1187	431	589	69	102	161	97	1187	2.99
1	2	0.51	917	1248	434	592	25	102	163	35	1248	2.84
1	3	0.56	918	1266	434	591	24	103	162	35	1267	2.8
1	4	0.57	918	1261	434	591	24	103	162	35	1263	2.81
1	5	0.51	918	1248	434	591	25	103	163	34	1249	2.84
2	1	0.96	562	354	356	361	355	585	409	605	606	5.85
2	2	0.95	733	294	329	273	298	604	454	598	734	4.83
2	3	0.93	734	235	218	156	322	705	542	621	734	4.83
2	4	0.97	746	239	144	142	311	733	546	652	747	4.75
2	5	0.97	746	240	135	85	354	742	556	740	748	4.74
3	1	2.13	390	474	521	399	595	391	389	388	598	5.93
3	2	2.03	357	356	521	355	765	357	470	377	766	4.63
3	3	2.11	802	355	407	355	355	377	412	457	804	4.41
3	4	2.13	828	336	480	369	498	335	348	337	829	4.28
3	5	2.13	829	363	302	498	491	361	300	317	830	4.27
4	1	4.36	442	443	475	444	450	444	442	444	477	7.44
4	2	4.24	433	435	575	433	436	431	431	435	576	6.16
4	3	4.38	445	446	447	447	465	445	445	465	467	7.6
4	4	4.64	445	443	443	443	443	443	482	448	482	7.36
4	5	4.04	431	432	550	432	446	462	431	441	551	6.44
5	1	8.48	462	463	462	463	464	464	464	464	465	7.63
5	2	9.36	465	464	479	465	465	465	464	466	479	7.4
5	3	9.10	477	477	477	477	477	477	477	478	478	7.42
5	4	8.50	467	467	467	468	467	467	467	467	469	7.56
5	5	9.84	475	474	470	474	473	473	474	474	475	7.47
6	1	18.20	532	533	533	531	532	532	531	530	533	6.65
6	2	18.16	521	521	521	521	521	521	522	522	522	6.79
6	3	17.72	518	518	519	516	518	518	518	523	523	6.78
6	4	18.75	509	510	508	510	510	509	510	510	511	6.94
6	5	18.41	514	514	515	516	514	514	513	516	516	6.87
7	1	37.42	588	590	587	589	588	590	588	590	590	6.01
7	2	38.22	599	600	599	599	599	599	599	598	600	5.91
7	3	37.78	591	591	591	590	591	591	590	591	592	5.99
7	4	37.83	587	588	587	588	587	587	587	586	588	6.03
7	5	37.19	584	585	584	584	584	584	585	584	586	6.05
8	1	83.12	788	789	789	788	788	789	789	789	790	4.49
8	2	84.40	818	819	818	817	818	818	818	818	820	4.33
8	3	83.05	816	815	817	815	816	815	815	815	817	4.34
8	4	83.14	818	817	816	817	817	815	817	818	818	4.34
8	5	85.67	816	817	816	817	816	816	817	818	818	4.34
9	1	200.57	1433	1434	1433	1433	1433	1434	1434	1435	1436	2.47
9	2	205.13	1591	1591	1590	1591	1590	1589	1591	1591	1591	2.23
9	3	204.79	1580	1581	1581	1580	1580	1581	1580	1580	1581	2.24
9	4	203.56	1582	1583	1583	1582	1583	1583	1583	1584	1585	2.24
9	5	203.00	1550	1551	1550	1550	1550	1551	1552	1552	1553	2.28

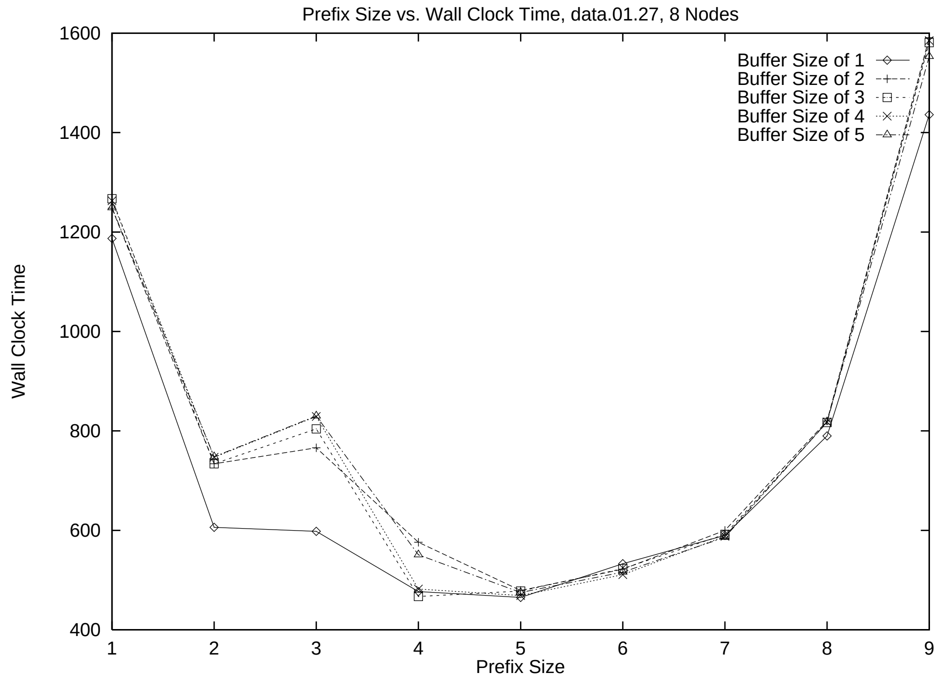
Dataset `data.07.25`, 4 nodes								
Prefix Depth	Buffer Size	Host CPU	Children times				Total Time	Speedup
3	1	305.00	6783	6786	6785	6786	6787	5.59
3	3	313.00	6784	6784	6784	6797	6797	5.58
4	1	1453.00	6753	6753	6753	6753	6753	5.62
4	3	1298.00	6712	6712	6712	6712	6712	5.65

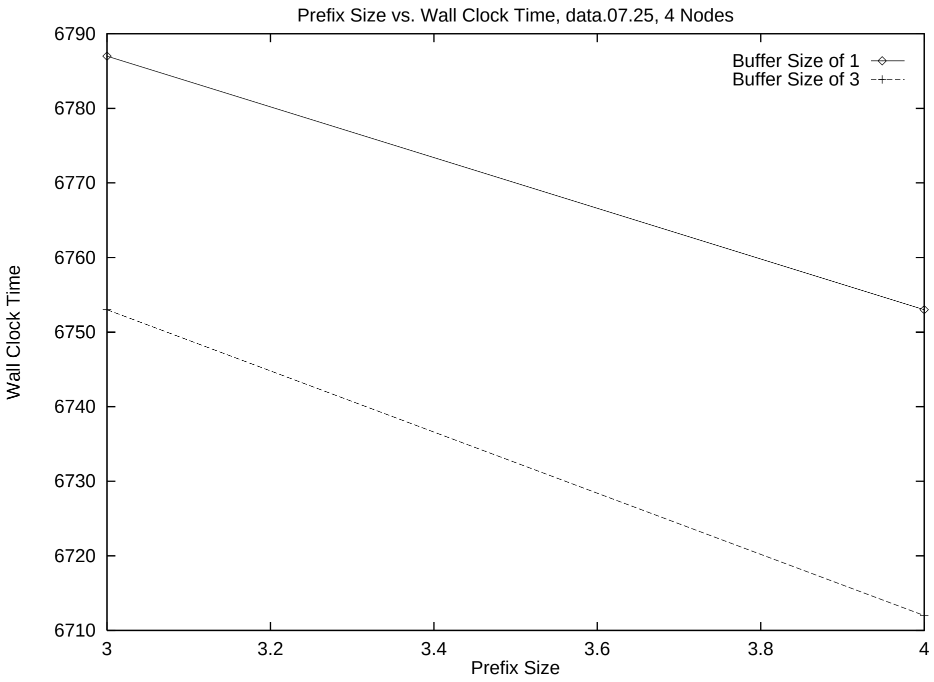
Dataset `data.07.25`, 8 nodes													
Prefix Depth	Buffer Size	Host CPU	Children times								Total Time	Speedup
2	1	1.80	5989	4513	4514	4846	4562	4517	4720	4515	5990	6.33
2	3	1.67	8112	3963	3961	5460	3946	4112	4135	3951	8112	4.68
3	1	6.58	4525	4525	4525	4525	4526	4525	4675	4525	4675	8.12
3	3	6.88	4530	4530	4540	4530	4601	4531	4530	4532	4601	8.25
4	1	32.44	4529	4529	4528	4529	4529	4529	4529	4552	4553	8.33
4	3	33.35	4511	4511	4511	4511	4511	4510	4511	4511	4512	8.41
5	1	143.44	4773	4773	4774	4773	4774	4774	4773	4774	4774	7.95
5	3	146.05	4832	4832	4832	4832	4833	4832	4832	4833	4833	7.85

Dataset `data.10.08`, 4 nodes								
Prefix Depth	Buffer Size	Host CPU	Children times				Total Time	Speedup
1	1	0.53	481	325	327	348	481	2.61
1	2	0.54	529	398	323	238	530	2.37
1	3	0.53	547	408	338	204	547	2.29
1	4	0.59	551	411	347	194	551	2.28
1	5	0.58	552	411	349	193	552	2.27
2	1	1.27	352	352	352	352	352	3.56
2	2	1.35	347	348	374	348	375	3.35
2	3	1.23	346	355	356	347	357	3.51
2	4	1.33	343	343	402	343	403	3.11
2	5	1.24	344	367	344	345	367	3.42
3	1	4.60	366	366	366	366	366	3.43
3	2	4.49	367	367	367	366	367	3.42
3	3	4.28	367	367	367	367	367	3.42
3	4	4.24	358	358	407	358	407	3.08
3	5	4.46	367	367	367	368	368	3.41
4	1	18.52	410	409	410	409	409	3.07
4	2	19.23	414	414	414	414	414	3.03
4	3	18.80	423	424	423	424	424	2.96
4	4	19.49	424	425	424	425	425	2.95
4	5	19.31	416	417	416	417	417	3.01
5	1	77.02	576	576	576	577	577	2.17
5	2	79.50	607	606	607	607	607	2.07
5	3	78.58	612	611	612	611	612	2.05
5	4	78.40	631	631	631	632	632	1.99
5	5	78.31	665	665	665	665	666	1.88
6	1	297.10	1256	1257	1256	1257	1257	0.998
6	2	303.75	1462	1462	1462	1462	1462	0.858
6	3	302.66	1459	1459	1459	1459	1460	0.859
6	4	302.24	1438	1438	1437	1438	1438	0.872
6	5	302.44	1411	1412	1412	1412	1412	0.888

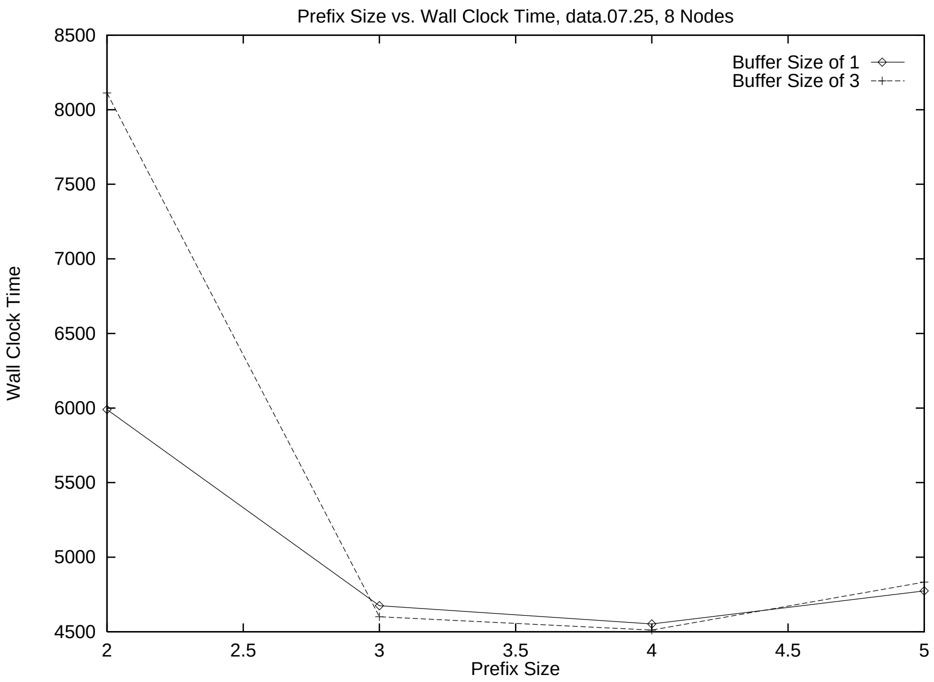
Dataset `data.10.08`, 8 nodes												
Prefix Depth	Buffer Size	Host CPU	Children times								Total Time	Speedup
1	1	0.55	481	253	131	156	74	218	74	74	481	2.61
1	2	0.57	498	264	139	158	74	221	40	52	498	2.52
1	3	0.64	499	265	140	159	74	221	35	53	500	2.51
1	4	0.58	500	265	140	158	75	222	35	52	499	2.51
1	5	0.62	499	264	140	159	74	221	35	53	500	2.51
2	1	1.23	155	156	166	163	154	157	155	157	167	7.51
2	2	1.13	168	146	152	226	146	149	146	146	226	5.55
2	3	1.27	177	171	140	192	149	168	139	139	192	6.53
2	4	1.29	196	139	161	198	137	166	138	138	199	6.3
2	5	1.35	220	199	143	220	143	174	143	144	220	5.7
3	1	4.35	176	176	176	176	176	176	176	176	177	7.09
3	2	4.48	176	176	175	175	175	175	176	176	176	7.13
3	3	4.47	178	177	178	177	177	177	177	180	180	6.97
3	4	4.13	173	173	172	173	173	173	173	173	173	7.25
3	5	4.61	174	174	174	174	173	174	173	180	180	6.97
4	1	19.20	252	251	252	252	252	252	252	253	253	4.96
4	2	18.08	265	265	266	266	266	265	265	266	267	4.7
4	3	18.58	264	264	264	265	265	264	265	265	265	4.73
4	4	18.60	262	262	261	262	261	262	262	262	262	4.79
4	5	19.34	263	263	263	262	263	262	263	264	264	4.75
5	1	80.64	530	530	530	530	530	530	530	530	530	2.37
5	2	79.31	642	642	642	642	642	642	642	641	643	1.95
5	3	80.68	639	640	640	640	640	639	639	640	641	1.96
5	4	83.33	651	651	651	651	651	652	652	652	652	1.92
5	5	81.16	662	662	663	663	663	662	662	664	664	1.89
6	1	305.32	1935	1935	1935	1935	1935	1935	1935	1935	1935	0.648
6	2	306.69	2213	2213	2213	2213	2213	2213	2213	2213	2214	0.567
6	3	312.74	2156	2157	2157	2157	2157	2156	2157	2158	2158	0.581
6	4	310.65	2242	2242	2241	2241	2241	2242	2242	2242	2242	0.56
6	5	309.86	2333	2333	2331	2332	2332	2333	2332	2332	2333	0.538


Prefix Size vs. Wall Clock Time, data.01.27, 4 Nodes
Wall Clock Time
Prefix Size
Buffer Size of 1
Buffer Size of 2
Buffer Size of 3
Buffer Size of 4
Buffer Size of 5
1700
1600
1500
1400
1300
1200
1100
1000
900
800
1
2
3
4
5
6
7
8
9

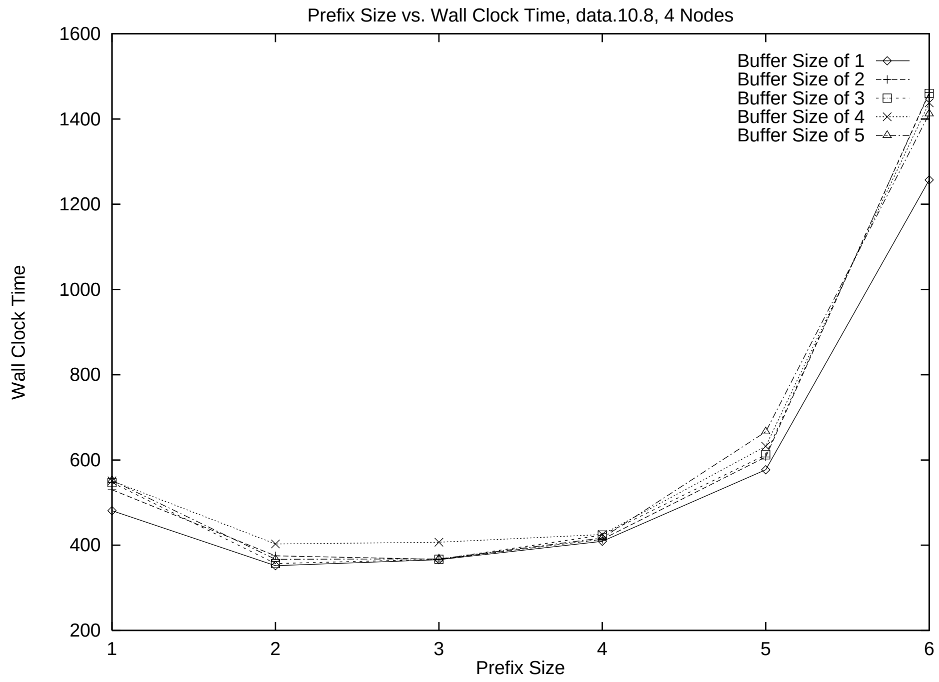

Prefix Size vs. Wall Clock Time, data.01.27, 8 Nodes
Wall Clock Time
Prefix Size
Buffer Size of 1
Buffer Size of 2
Buffer Size of 3
Buffer Size of 4
Buffer Size of 5
1600
1400
1200
1000
800
600
400
1
2
3
4
5
6
7
8
9

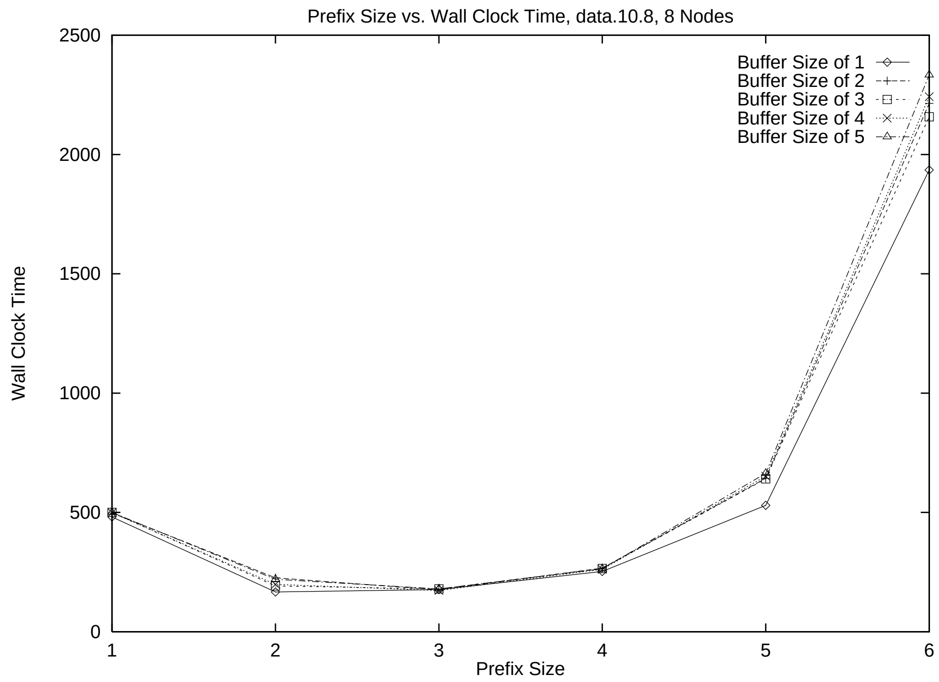


Prefix Size vs. Wall Clock Time, data.07.25, 4 Nodes
Buffer Size of 1
Buffer Size of 3
Wall Clock Time
Prefix Size
6790
6780
6770
6760
6750
6740
6730
6720
6710
3
3.2
3.4
3.6
3.8
4



Prefix Size vs. Wall Clock Time, data.07.25, 8 Nodes
Buffer Size of 1
Buffer Size of 3
Wall Clock Time
Prefix Size
8500
8000
7500
7000
6500
6000
5500
5000
4500
2
2.5
3
3.5
4
4.5
5


Prefix Size vs. Wall Clock Time, data.10.8, 4 Nodes
Buffer Size of 1
Buffer Size of 2
Buffer Size of 3
Buffer Size of 4
Buffer Size of 5
Wall Clock Time
Prefix Size
1600
1400
1200
1000
800
600
400
200
1
2
3
4
5
6


Prefix Size vs. Wall Clock Time, data.10.8, 8 Nodes
Buffer Size of 1
Buffer Size of 2
Buffer Size of 3
Buffer Size of 4
Buffer Size of 5
Wall Clock Time
Prefix Size
2500
2000
1500
1000
500
0
1
2
3
4
5
6

6.3 Extended Parallel Algorithm

The initial tests with the extended parallel algorithm were disheartening; in one case, the new program took longer to complete than the old one.

However, after making the modification described at the end of the section on the extended parallel algorithm, quite nice results were obtained. The algorithm finished significantly faster, and in all cases where the optimal solution is known, the algorithm actually found the optimal solution. Here is a table summarizing the results; note that entries with the "components in prefix" column greater than 1 did not run with the modification.

Dataset	Trees	CPU's	Prefix	Components in Prefix	Time (seconds)	Time for original PB+B
data.01.27	58	4	9,11,12,14,21	3	1770	880
		4	9,11,12,14,16,21	4	780	880
data.10.8	60	4	0	1	170	350
		4	0,14,19	3	150	350
		4	0,3,5,14,19	5	53	350
data.07.25	71	4	10,11,14	3	4379	6800
data.02.27	91	8	19,20,21	1	354,831	>600,000
		8	23,24	1	˜ 194,000	>600,000

7 References

1. Frederick Harris, Jr. Parallel Computation of Steiner Minimal Trees. PhD Dissertation, Department of Computer Science, Clemson University, Clemson, South Carolina, 1994.

2. Jennifer Ludwick. A Parallel and Genetic Approach to the Steiner Tree Extraction Problem. Masters Paper, Department of Computer Science, Clemson University, Clemson, South Carolina, 1996.

3. P. Berman and V. Raimayer. Improved Approximations of the Steiner Tree Problem. *Journal of Algorithms*, vol. 17, no. 3, pp. 381-408, Nov. 1994.

4. K. Kalpakis and A.T. Sherman. Probabilistic Analysis of an Enhanced Partitioning Algorithm for the Steiner Tree Problem in R^d. *Networks*, vol. 24, no. 3, pp. 147-56, May 1994.

5. E. Dahlhaus. A Parallel Algorithm for Computing Steiner Trees in Strongly Chordal Graphs. *Discrete Applied Mathematics*, vol. 51, no. 1-2, pp. 47-61, June 22, 1994.

6. J. griffith, G. Robins, J. S. Salowe, and TongTong Zhang. Closing the Gap: Near-Optimal Steiner Trees in Polynomial Time. *IEEE Transactions on Computer Aided Design and Describing Integrated Circuits Systems*, vol. 13, no. 11, pp. 1351-65, Nov. 1994.

7. S. Chopra. Comparison of Formulations and a Heuristic for Packing Steiner Trees in a Graph. *Annual of Operations Research*, vol. 50, pp. 143-71, Sept. 1994.

8. Jin Zhongi, Shu Huaying, and Guo Hong Ming. Using Neural Networks to Solve Steiner Tree Problem. Proceedings of TENCON '93, 1993 IEE Region 10 Conference on "Computer, Communication, Control, and Power Engineering," Beijing, China, pp. 807-810, October 19-21, 1993.