University of Nevada, Reno

A Parallel Application for Tree Selection in the Steiner Minimal Tree Problem

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science and Engineering

by

Joshua M. Hegie

Dr. Frederick C. Harris, Jr., Thesis Advisor

August, 2015

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

JOSHUA HEGIE

Entitled

A Parallel Application For Tree Selection In The Steiner Minimal Tree Problem

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Dr. Frederick C. Harris, Jr., Advisor

Dr. Mehmet H. Güneş, Committee Member

Dr. Anna K. Panorska, Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

August, 2015

Abstract

A classic optimization problem in mathematics is the problem of determining the shortest possible length for a network of points. One of these problems, that remains relevant even today, is the Steiner Minimal Tree problem. This problem is focused on finding a connected graph for a cloud of points that minimizes the overall distance of the tree. This problem has applications in fields such as telecommunications, determining where to geographically place hubs such that the total length of run cabling is minimized, and for a special case of the problem, circuit design.

Dedication

For my wife Christine and our little girl Etna Rose.

Acknowledgments

I would like to thank my advisor, Dr. Frederick Harris, and my committee members Dr. Mehmet Gunes and Dr. Anna Panorska for their time and suggestions.

I would also like to thank to my family and friends for their love, encouragement, and never-ending support.

A number of figures in this paper were generated using C.a.R. [4].

Contents

List of Figures

Chapter 1

Introduction

This problem was, in its simplest form, proposed by Pierre de Fermat to Evangelista Torricelli, a fellow mathematician, was the question "Find the point such that the sum of its distances from the vertices of a triangle is a minimum"[16]. This problem, which sounds trivial at first, is the foundation of the larger problem that would later be posed by Jakob Steiner in the early 19^{th} century. The answer to Fermat's question, for an equilateral triangle, is the simple average of the points in the originating triangle, shown in Figure 1.1. Past the simplest case, as with many problems, the solution is not as straight forward. The general solution to this problem, for an arbitrary triangle, can be broken down as follows:

1. Create an equilateral triangle out of 2 arbitrary points on the original triangle. The point added to create this new triangle will be referred to as A
2. Circumscribe a circle around the new triangle.
3. Cast a ray from A to the remaining point in the original triangle.
4. Where the newly cast ray crosses the circle, inside of the original triangle, is where the new point should be placed.

Alternately, as shown in Figure 1.2, the intersection of the circles created by the creation of these circles is also the point in space at which the new point should be placed, which is one of the methods of creating a full Steiner tree (FST) discussed by Melzak [20].

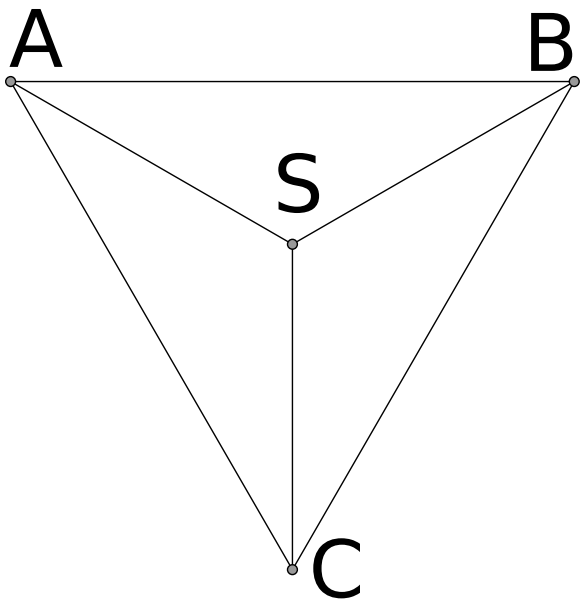


Figure 1.1: Steiner point S for $\triangle ABC$

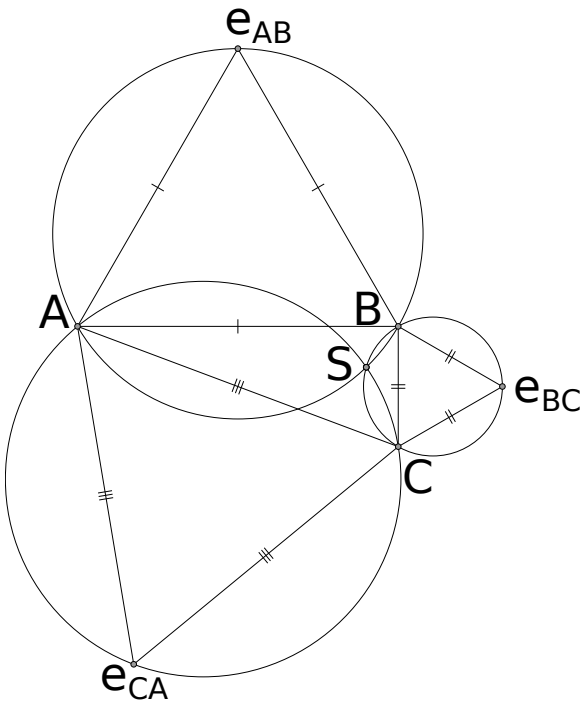


Figure 1.2: Fermat point for a non-equilateral triangle

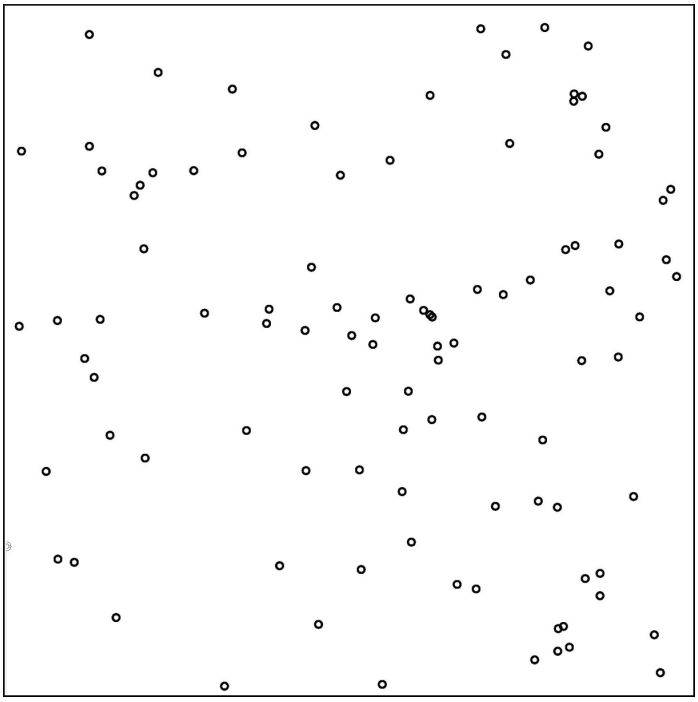


Figure 1.3: An example input to the Steiner Minimal Tree problem [13, Figure 16]

The case of this problem that Jakob Steiner proposed extended Fermat's question to the more general case of "given n points, find the set of line segments with the shortest total length that connects all of the points". With the groundwork laid out by Fermat, the answer to this problem is to convert a set of points, Figure 1.3, utilizing the ability to create Fermat points, into a fully connected tree, Figure 1.4, that satisfies all of the following requirements:

- Each point from the original point cloud has a path to every other point from the original point cloud
- The total distance for the entire graph is minimized
- There are no loops in the resulting graph
- There are no points at which any edges in the graph cross each other
- For n points, no more than $n-2$ Fermat points are added.

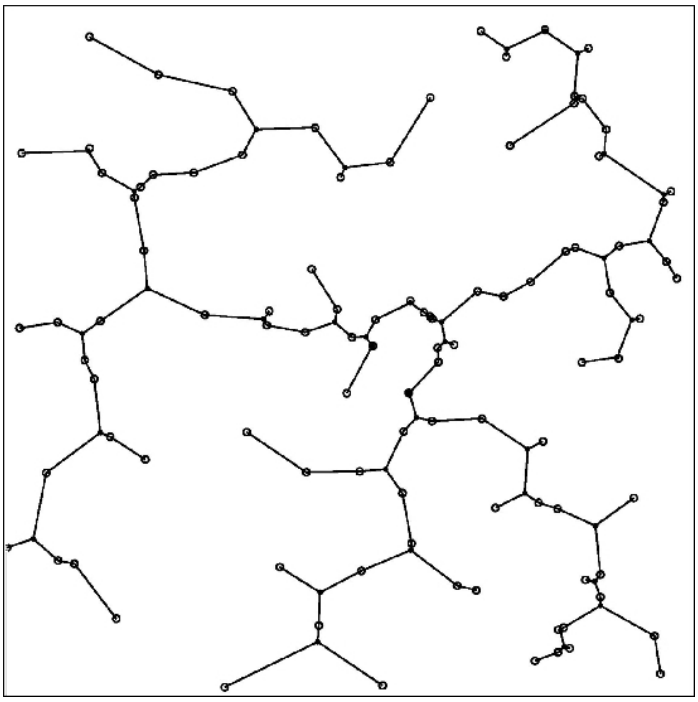


Figure 1.4: A solution for Figure 1.3 [13, Figure 17]

The rest of this thesis will be laid out as follows. Chapter 2 covers both a more in depth background of the Steiner Minimal Tree problem and a short background on GPGPU computing. Chapter 3 describes the solution proposed in this thesis, along with a discussion of the GPGPU and OpenMPI based implementations. Chapter 4 discusses the gathered experimental results. Chapter 5 will quickly recap the solution and results, and will include a discussion of future work.

Chapter 2

Background

2.1 Steiner Minimal Tree Problem

The Steiner Minimal Tree problem has well understood geometrical solutions [5, 6, 7], with several heuristics [1, 2] proposed to help narrow the computational scope. As discussed in Chapter 1, the solution for the case of an arbitrary triangle was solved by Fermat and Torricelli in the early 17^{th} century. The solution to this problem, shown in Figure 1.2, is generated by an approach similar to finding the Napoleon points of a triangle [25]. Each of these points, the Fermat point and the first Napoleon point find the center of a triangle [15].

Winter devised an exact algorithm [28] to solve the Steiner Minimal Tree problem. Winter's solution approached the problem in a serial fashion, and as a result suffered a bottleneck in the more computationally intensive phases of the algorithm. Winter's algorithm solved the Steiner Minimal Tree problem in a number of phases. It would first generate all of the Fermat-Steiner points for the inputs. For each triplet of points, A, B and C, a Fermat-Steiner point S is created. For points A and B the point S_{AB} is added, attempting to connect to C. In the case of only 3 source points there will be 6 possible Fermat-Steiner points: S_{AB}, S_{AC}, S_{BC}, S_{CA}, S_{CB} and S_{BA}. The order of the 2 points attached to the Fermat-Steiner point, AB in S_{AB}, is important for determining where the Steiner-Fermat point is placed. When determining where to place S_{AB} an equilateral point, e_{AB}, is places such that traversing the points $Ae_{AB}B$ goes counterclockwise. Casting a ray from e_{AB} to C results in S_{AB}, if and only if the projected ray crosses $\widehat{AB}$, as

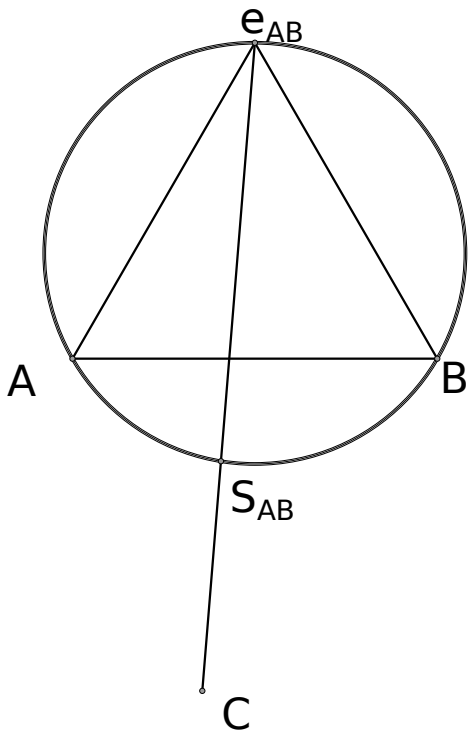


Figure 2.1: Generation of point S_{AB}

shown in Figure 2.1. In the case of trying to form S_{BA} for the same points, the Fermat-Steiner point cannot placed. Figure 2.2 shows that a ray cast from e_{BA} to C will not cross $\widehat{BA}$, and is therefore not a valid point [29].

After this phase, the minimum spanning trees (MST) are generated. In selecting the placement of edges in the graph, a pair of rules are adhered to. An terminal point, A, B or C in Figure 2.1, will always be rank 1, that is have a single edge coming out of them. Additionally, with the exception of the case of $N = 2$, each terminal point will connect to a Fermat-Steiner point, and not another original vertex. A Fermat-Steiner point will have rank 3, and can be connected to at least one terminal point, and can be connected to up to two other Fermat-Steiner points. This kind of connectivity can be seen in Figure 2.3 and Figure 2.4, which shows the possible connectivity for N=4.

As the MSTs are being laid out, a compatibility matrix is being built. A pair of trees is deemed to be compatible if the only overlap between their terminal points is a single vertex (*i.e.* both contain the point A and have no other overlap). A pair of trees is deemed

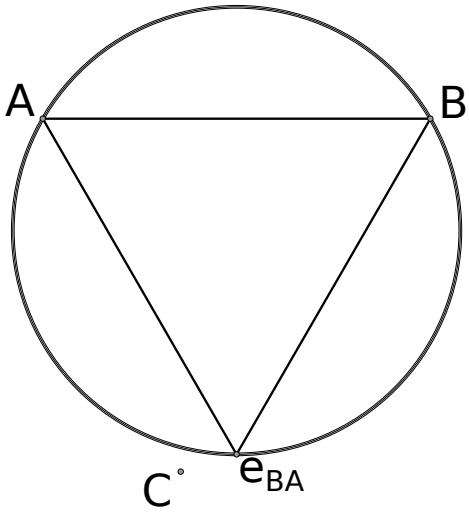


Figure 2.2: Attempted generation of point S_{BA}

to be incompatible if they share more than one terminal point, since in this case there is the risk of adding a cycle into the graph. In the event that neither tree being tested for compatibility shares any vertices, the trees are considered possibly compatible, since the pairing of one with the other doesn't directly break compatibility. In the event that a pair of trees with possible compatibility are paired together, another tree, or trees, with compatibility to both of these trees must be included to bridge the potential gap that has been introduced.

The last phase of the solution is to build out the full Steiner tree (FST) by means of a union of the MSTs. During this phase the current length of the partial FST will be compared to the length of any known FST that covers all of the terminal points. This optimization is simply to cull out sub-optimal solutions before any additional resources are dedicated to them, because the solution that fails this test will not be able to produce a shorter solution. The next test verifies that the Fermat-Steiner points are not over-committed, that they're only present in at most two FSTs. The last test utilizes the compatibility matrix, referred to in the 1985 Winter paper [28] as the reachability matrix. At the end of this phase the shortest tree has been found.

An improvement on Winter's Algorithm was a parallel solution proposed by Harris [10, 11] in the form of PARSTEINER94. Harris' proposed solution was able to solve

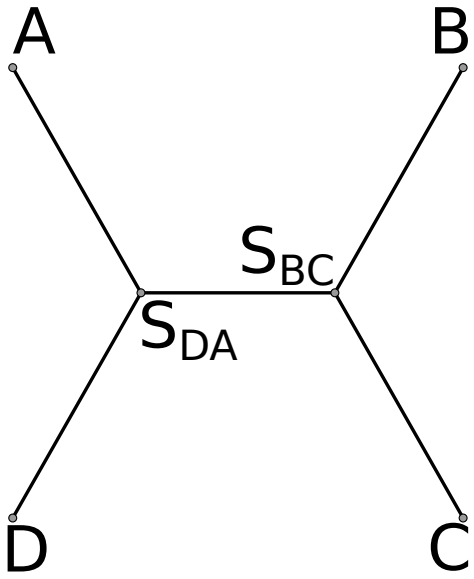


Figure 2.3: One of the valid FST for N=4

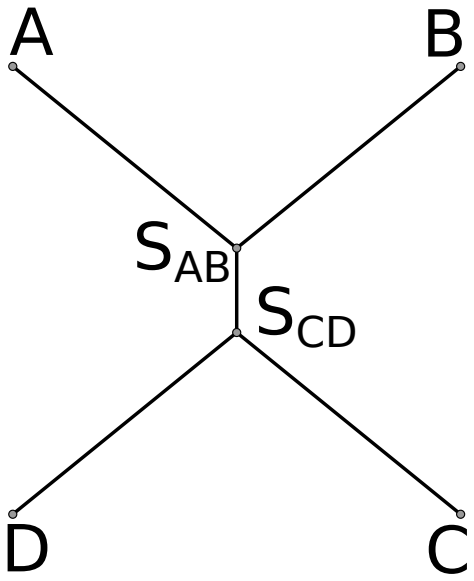


Figure 2.4: The other valid FST for N=4

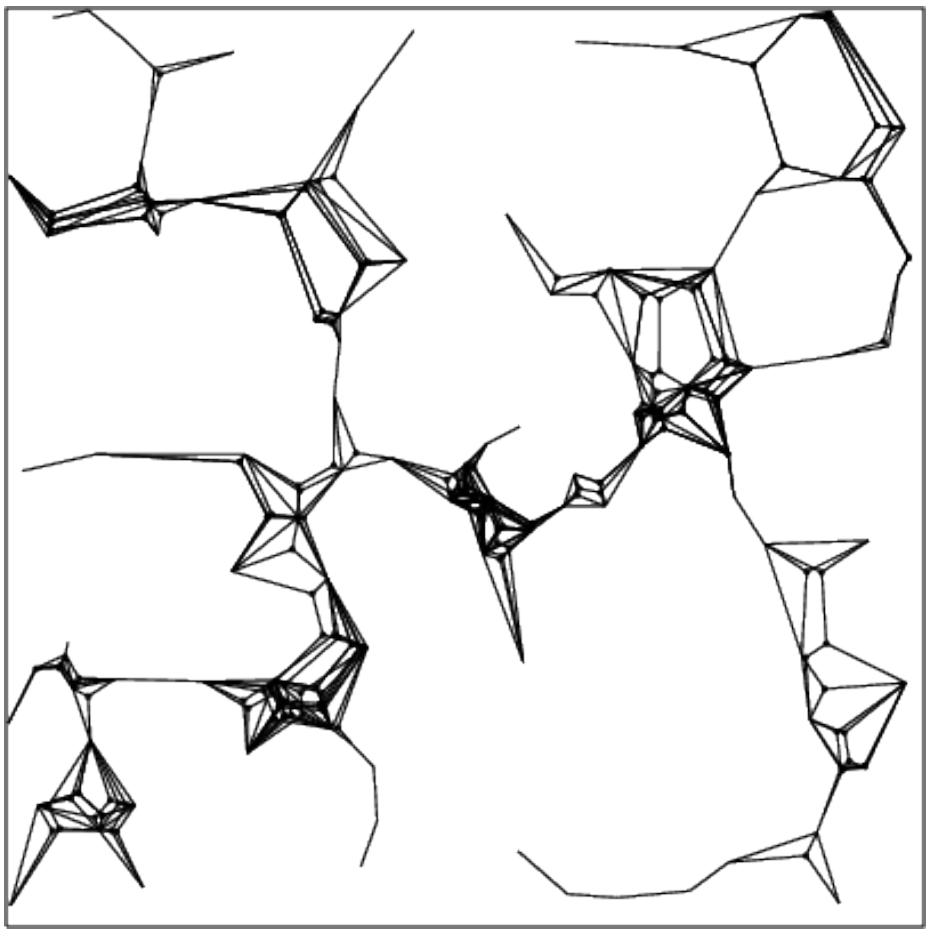


Figure 2.5: T_list for a 100 point tree, similar to [12, Figure 16]

the SMT problem for problems with 100 terminal points, at the time only comparable only to EDSTEINER89 [6], and was able to do so with at least an order of magnitude reduction in computation time [11].

Winter's original algorithm [28, 29] was run entirely in serial, which presented a number of bottlenecks. One such bottleneck is the last phase of Winter's Algorithm, the union of MSTs into the SMT, an example of all FSTs for a problem can be seen in Figure 2.5. Harris exploited an attribute of the graphs. Each of the trees shown in Figure 2.6 can be solved independently of each other, and therefore solved independently of each other.

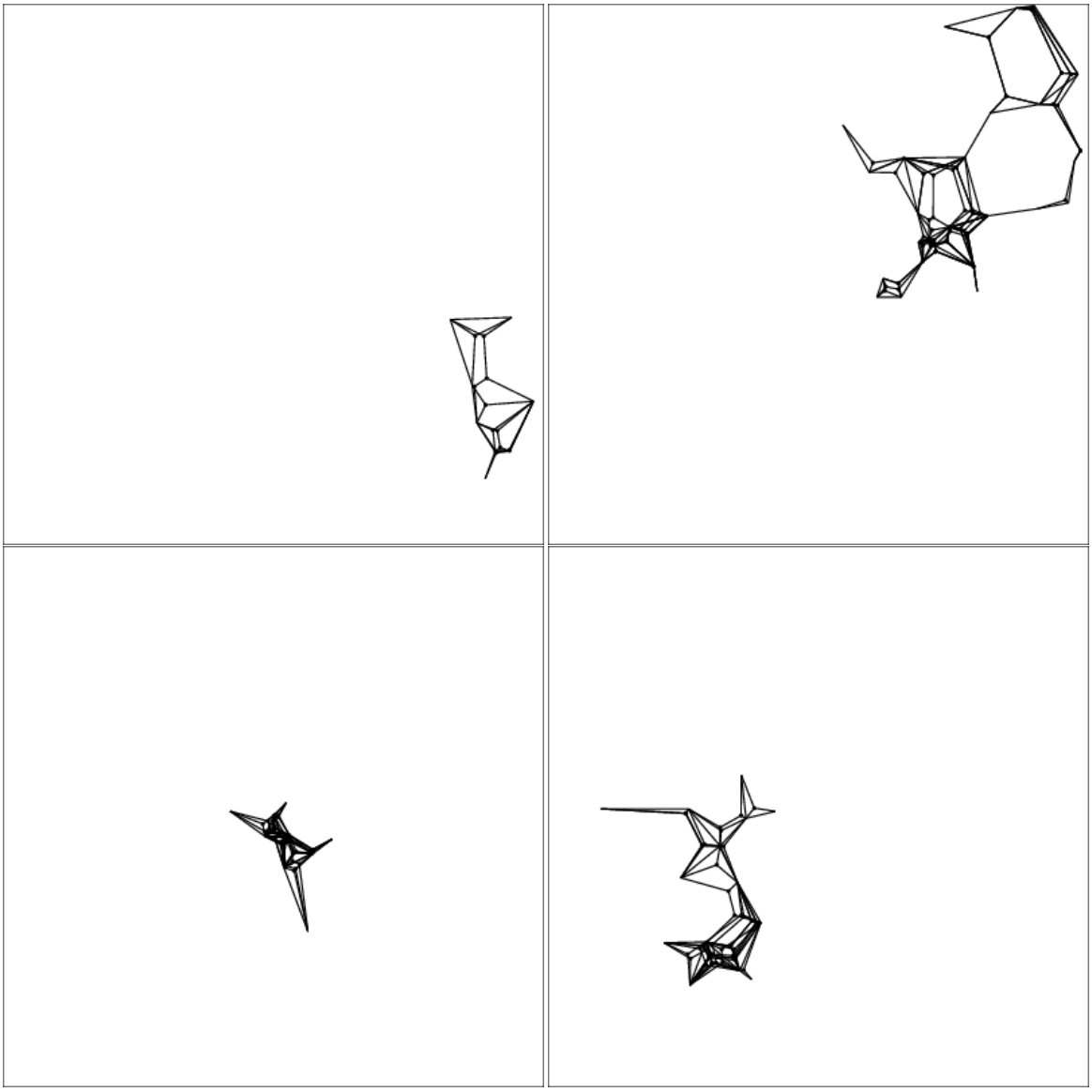


Figure 2.6: A selection of split graphs from Figure 2.5

2.2 CUDA

The Graphics Processing Unit (GPU) is a specialized coprocessor, designed for efficient handling of a fixed function rendering pipeline. Traditionally, this pipeline has focused on the process of converting 3D geometry, described in terms of triangles, into a 2D representation, outputting pixels for displaying on the screen. Since GPUs were designed with specific functionality in mind, that is the computations needed to render an image on a screen, they were designed with a high degree of parallel processing, since each pixel could generally be treated as a separate computation. In 2001 the nVidia GeForce 3 launched with a programmable vertex shader [19], which allowed for more than just rendering to be done on a GPU. Projects were able to take advantage of the new programming interface [3, 18] to do scientific computing. The workaround to that GPUs, at this time, still only were dedicated rendering devices was to find ways of representing a problem in terms of the graphics pipeline. By finding ways of describing a problem such that it had a mapping from how traditional programming to rendering concepts, researchers were able to leverage this resource to their advantage. An example of this is a GPGPU tutorial by Dominik Göddeke [8]. Göddeke covered a number of topics in this paper, focusing on how to map concepts found in more traditional programming languages like C/C++, FORTRAN, etc… into OpenGL [24, 26] constructs. Examples of these mappings include thinking in terms of textures instead of arrays, considering shaders instead of compute kernels and thinking of the draw operation in OpenGL instead of a compute operation. Using these types of techniques it became possible to see a high degree of speedup in a number of applications [3], up to $5x+$ for some applications.

In 2007 nVidia introduced the CUDA programming language for doing computations directly on nVidia GPUs, with OpenCL [14, 27], a vendor agnostic library, introduced in 2009 [1]. In terms of raw computational power, as seen in Figure 2.7, GPUs have quickly outpaced the theoretical computing capability of CPUs released in the same time-frames. An interesting divergence that can be seen in Figure 2.7 is that there are

[1] Despite a number of similarities between CUDA and OpenCL, only CUDA will be considered for the remainder of this paper

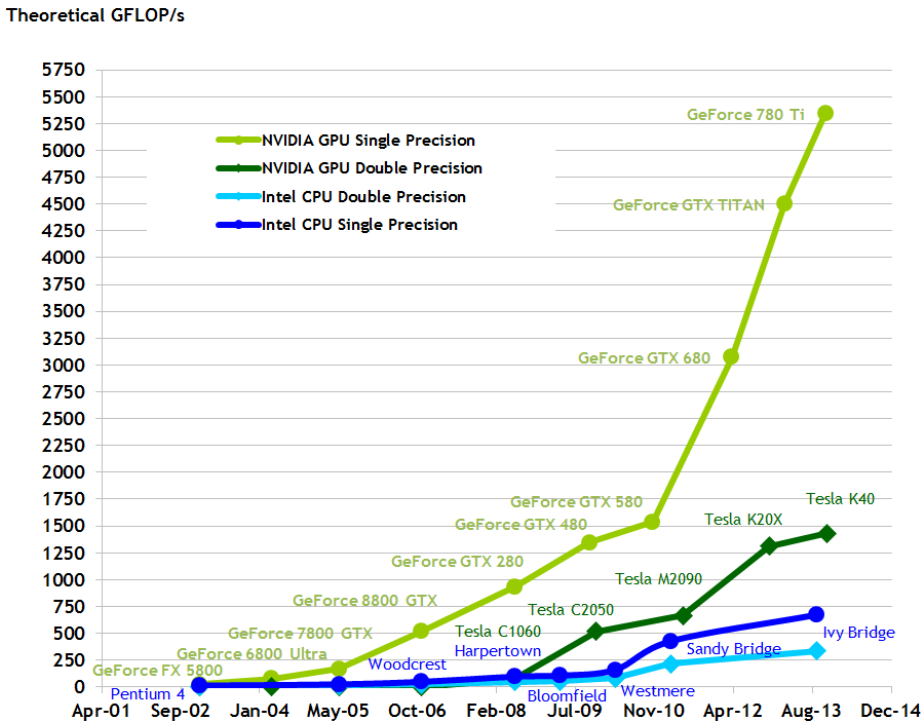


Figure 2.7: Plot provided by nVidia comparing the number of floating point operations per second (FLOPS) of various Intel CPUs and nVidia GPUS [22]

GFLOP values for both double and single precision floating point operations for each generation of CPU, but that there are different devices listed for each precision type for the GPUs. This dichotomy is rooted in the device's intended purpose. The single precision devices are consumer grade hardware, which is primarily intended for use in the entertainment industry. The double precision devices listed in Figure 2.7, however, are specifically designed with high performance computing workloads in mind, having features such as ECC RAM to help differentiate these lines from their consumer counterparts. The number of threads possible for a single GPU when coupled with the cost for a single precision device makes GPGPU programming appealing. As a point of reference, at the time of writing a single GeForce GTX 780 Ti with 2880 CUDA cores [23] can be obtained for approximately $370 from a variety of retailers.

Figure 2.8 presents a representation of how CUDA arranges processing. At the lowest level are threads, which are analogous to a thread in CPU based programming. Each

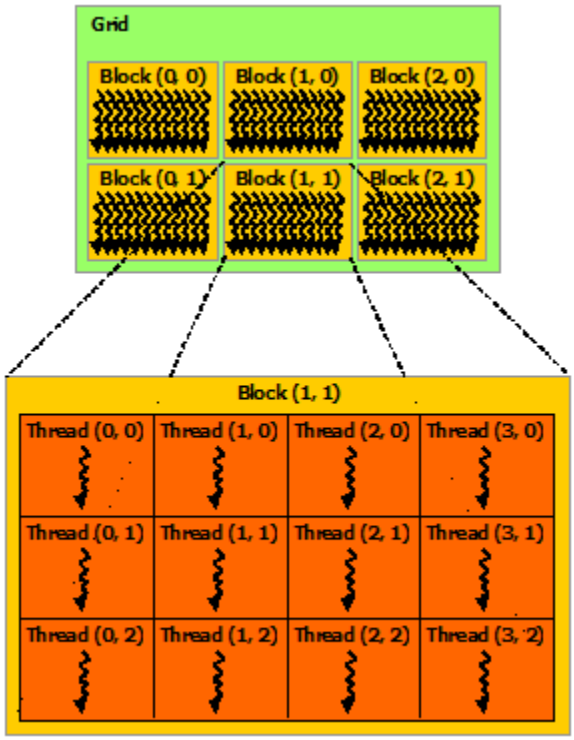


Figure 2.8: Depiction of the processing hierarchy of CUDA threads [22]

thread will be assigned computations to do, generally in the form of a compute kernel. These threads are arranged into blocks, which can contain between 32 and 512 threads. All of the threads in a block will all operate with the same kernel in parallel. Due to the existence of an upper limit on the number of threads in a block there is also the concept of a grid, which is a logical grouping of blocks. Similar to the threads within a block, all blocks within a grid will operate on the same compute kernel. An additional advantage that the CUDA environment possesses is granularity of memory allocations, as depicted in Figure 2.9. The levels of memory allocations enable thread local memory and a degree of message passing. Message passing can be accomplished using the global or per block memory allocations, removing the need to constantly remove data from the GPU (device) to the calling code (host) to redistribute shared values. Additionally, with the per thread allocation model, it's possible to allocate local temporary variables without needing to allocate space in a more global scope.

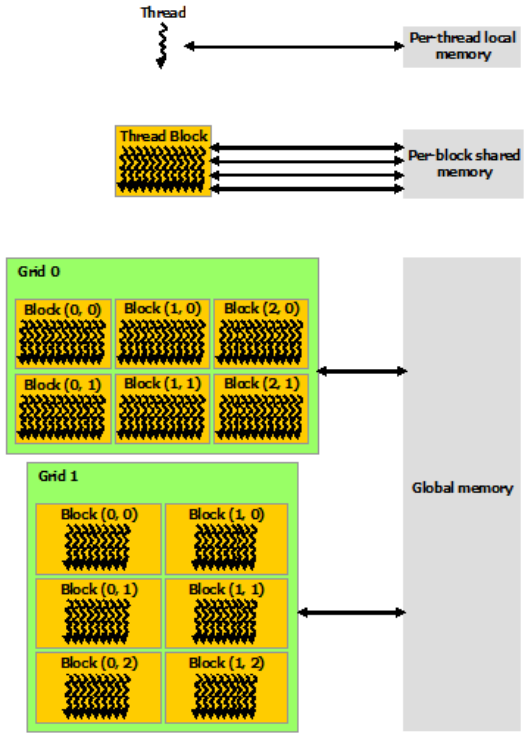


Figure 2.9: Hierarchy of the scopes of memory available in CUDA [22]

Chapter 3

Proposed Solution

3.1 General Principles

Given the particular problem discussed in Section 2.1, Algorithm 3.1 was developed. Algorithm 3.1 describes the process through which a collection of trees is evaluated. There are several possible outcomes for an input to this algorithm: "is a solution", "is valid but doesn't solve" and "is valid with longer distance (than the current solution)". A set of trees that comes up "valid with longer distance (than the current solution)" can still produce valid solutions with the addition of other trees, but this branch can be pruned by virtue of that it will never produce a better solution than what has currently been found. This leaves 2 cases for sets of trees to consider are is a solution and valid but doesn't solve. The simpler case of these two is when the current selection of trees is a solution, that is all of the terminal nodes are included and there are no compatibility issues. In this case a test of the newly found solution's distance, the sum length of all of the trees it includes, against the currently stored solution's length. In the event that the new solution is shorter, it should be stored, otherwise it should be discarded. The slightly more complicated case is when the current set of trees has a shorter distance than the current solution, but is not in its current configuration a solution itself. In this case new inputs are created for the next round of processing. Before going into how new inputs are created, the storage of data in memory must be discussed. This solution treats the combinations of trees as a bit vector, with a particular bit being set to '0' or '1' indicating its inclusion or exclusion. The bit vector shown in Figure 3.1 is, in this imple-

mentation, stored inside of a standard 32 bit integer. This decision was made to allow for a higher degree of compactness, as opposed to an array booleans. The deciding factor, in favor of memory, that forced this decision was that a 32 bit integer could contain 32 bits of information, where using 32 booleans to do the same would have a footprint of $8x$ as much[21]. With this in mind, the collections of trees that are valid, but aren't yet a solution are processed for creating new inputs. This process attempts to place a single '1' into the bit vector, past the last modified bit. As an example should Figure 3.1 be considered to have length 3, that is bit 3 was the last one set, two more inputs could be generated for the next phase of processing, one with tree 4 included and tree 5 not included and the other with tree 4 left out and tree 5 added in. Prior to storing the new values two more tests are run. The new sets of trees are tested for total distance against the solution, preventing a sub-optimal solution from being considered, and are tested for compatibility, to prevent the processing of a set of trees that are not valid. An invalid set of trees will not, no matter how many other trees are added to it, produce a valid solution, so all branches under the set can be pruned. Should a set of trees pass this last set of tests, it can be stored for processing.

Tree Number	1	2	3	4	5
	1	1	1	0	0

Figure 3.1: An example bit vector with trees 1, 2 and 3 included

A \ B	1	2	3	4	5
1	0	-1	1	0	1
2	-1	0	1	0	0
3	1	1	0	1	0
4	0	0	1	0	0
5	1	0	0	0	0

Figure 3.2: Compatibility matrix

Algorithm 3.1 Logic for determining whether or not to continue a branch of the solution. In this snippet, len refers to the furthest in set bit (*i.e.* if trees 1 and 3 are enabled len would be 3) and dist refers to the best total solution thus far. For this to work, the solution distance is set to FLT_MAX before doing any processing.

```
if Solution then
    if New Solution Distance < Solution Distance then
        Store new solution
    else
        Discard solution
    end if
else if Distance < Solution Distance then
    for i = len; i < n+1; ++i do
        if (Adding tree i does NOT break compatibility) &
                (Distance + tree i Distance < Solution Distance) then
            Keep new tree set to be worked on
        else
            Prune this branch
        end if
    end for
else
    Abandon this branch
end if
```

Figure 3.2 can be traversed recursively to create the search tree seen in Figure 3.3. Note that there are a number of duplicates (*e.g.* 123 and 315) present in the tree and, as

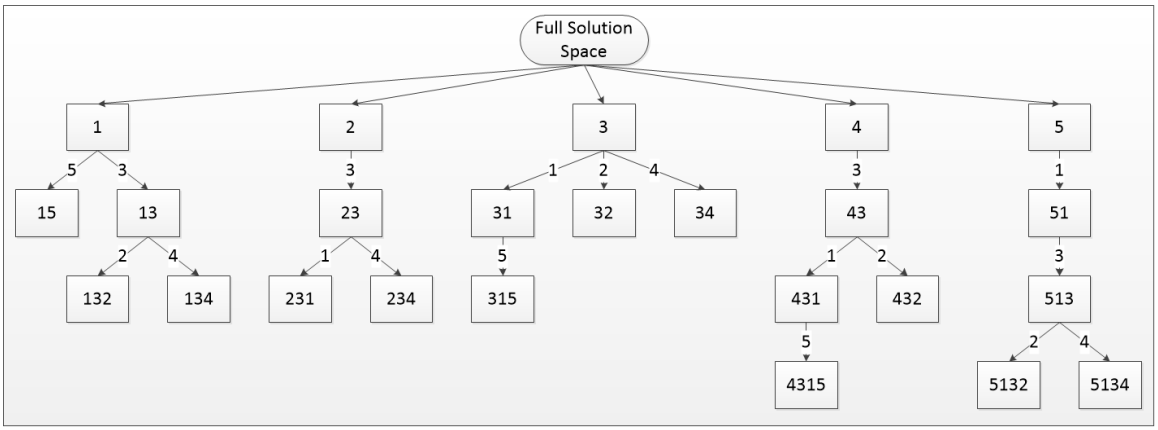


Figure 3.3: Recursive search for Figure 3.2

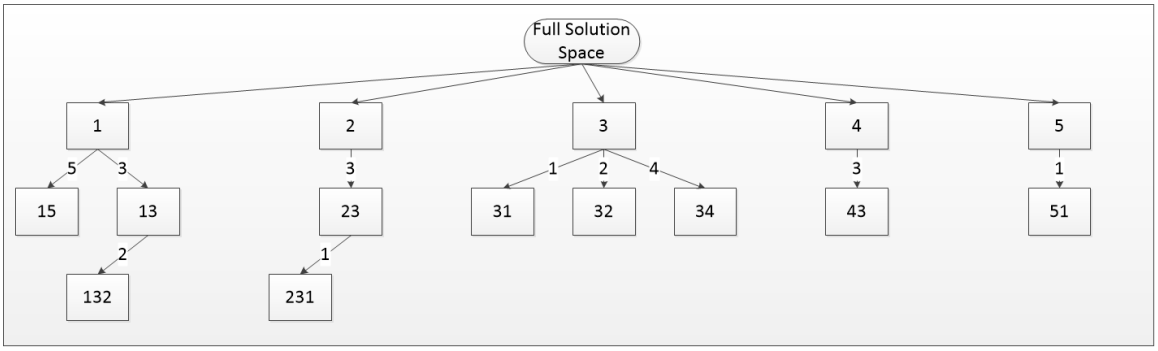


Figure 3.4: Figure 3.3 with incompatible branches removed

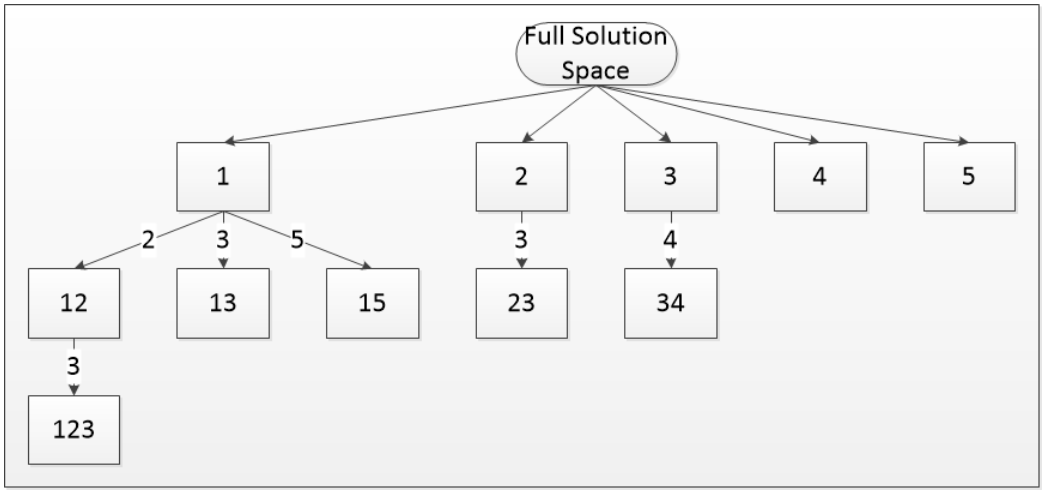


Figure 3.5: Search tree generated by the proposed algorithm

Tree 1 only	1	0	0	0	0
Tree 2 only	0	1	0	0	0
Tree 3 only	0	0	1	0	0
Tree 4 only	0	0	0	1	0
Tree 4 only	0	0	0	0	1

Figure 3.6: Initial inputs for Figure 3.2

depicted, a number of incompatible entries. Figure 3.4 depicts this same tree with the incompatible entries removed, but still contains duplicate entries.

The process of generating the compact tree begins with an array of bits. Consider the following array (Figure 3.6 for the 5 tree data set proposed in Figure 3.2, where a '1' indicates that the tree in that position is included in the current value and a '0' indicates the opposite. This starting array meets an important set of requirement, namely that a single tree will always be compatible with itself. Starting from this case, the second generation of of combinations can be generated. Figure 3.7 shows what happens when the first pass of generation occurs, by adding only a single tree at a time in, and always only adding to the left of the last included tree, which was inspired in part by Knuth's generation of all binary trees algorithms [17]. In the case of only including the last tree, Figure 3.7e, once it's processed, no new inputs to the next cycle are created, which defines one of the ending conditions for data generation. By iterating this way, the tree shown in Figure 3.5 is generated in a manner similar to a breadth first search. This will generate the $2^n - 1$ possible nodes to be checked, in the worst case. In all likelihood multiple branches of the search space will be removed from consideration, discussed in Algorithm 3.1.

The advantage to this approach is that, as shown in Figure 3.8. As can be seen in Figure 3.8a, the top level values can be processed in parallel, allowing for all of the trees to be processed independently of each other. Similarly Figure 3.8b shows how the various trees, A and B in the figure, are independent of each other and can solved as such.

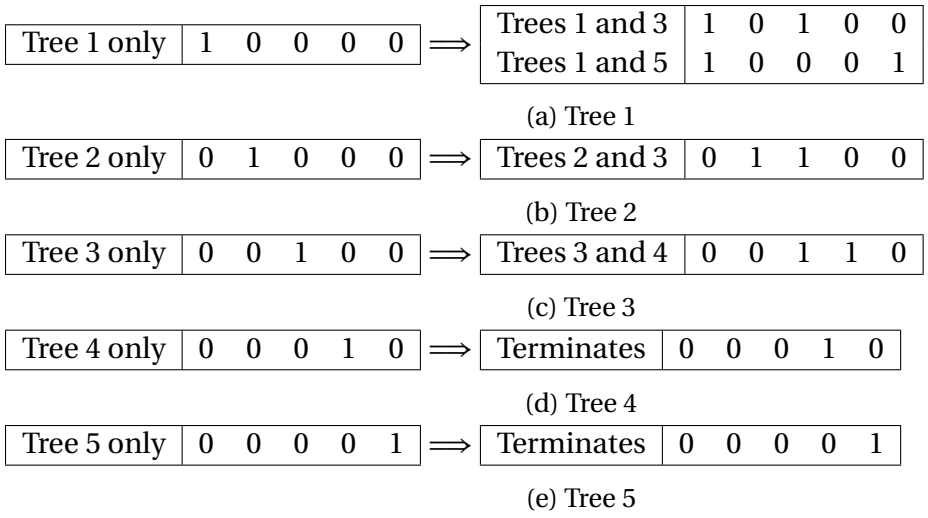
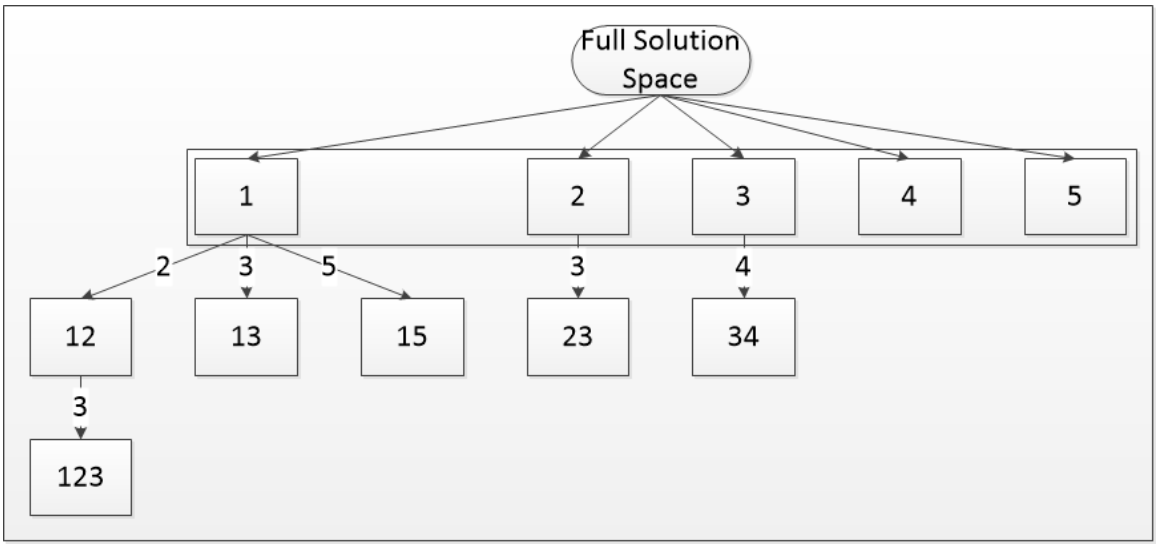


Figure 3.7: New inputs after 1 iteration

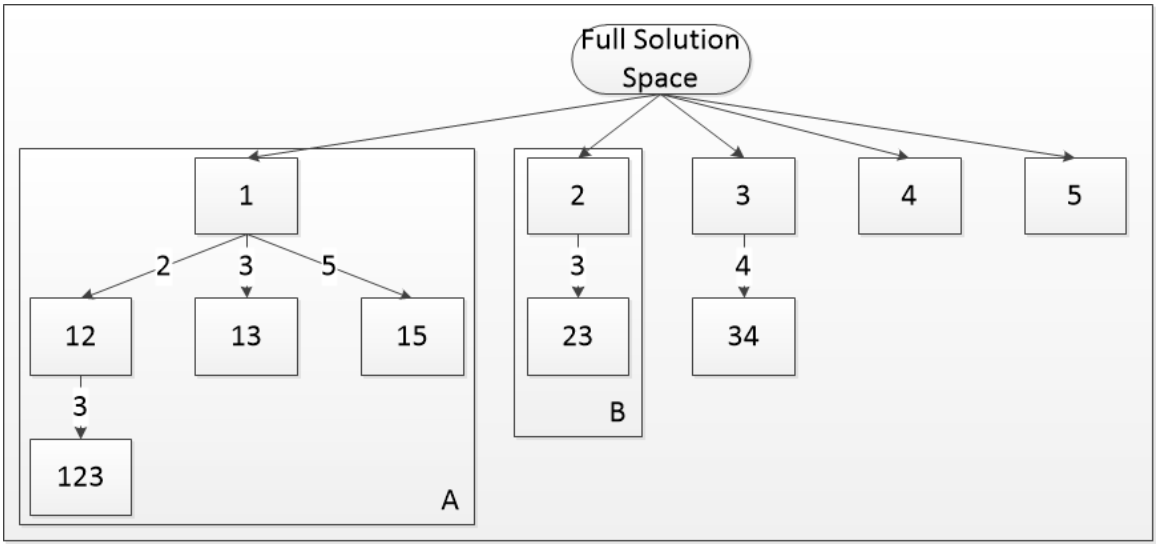
3.2 CUDA

Section 2.2 covered how GPGPU processing has been revolutionizing a number of computational fields, and the portion of the Steiner Minimal Tree problem that has been taken on here was an excellent fit for the highly parallel processing that GPGPU provides. Figure 3.8 showed a number of levels at which the FST merge operation can have. By capitalizing on that parallelism, and the high number of threads that can be run on even a single GPU, the goal of the GPGPU application is to process as many inputs in parallel as possible. When looking at Figure 3.8a, each of the circled nodes an be evaluated at the same time. In fact, a CUDA thread can be dedicated to each node to do the processing, which has been taken advantage of for this application.

A master and worker approach has been adopted for this solution. The master process is run on CPU, and is responsible for storing unprocessed data and passing data to the compute kernels. The workers, in this implementation, are the CUDA threads, which are responsible for processing a single set of trees, provided by the CPU based master process. Before doing any processing, the input is tested for whether the distance, that is the length of all included trees, is shorter than the currently found solution if one has been found. After the distance test, each set of trees is then evaluated for whether it's a



(a) Parallelism in the initial inputs



(b) Visualization of how subsets of the search tree are parallel

Figure 3.8: Visualizations of how the search space can be traversed in parallel

solution or not. In the event that a new solution has been found, it's stored to be tested against later. In the event that the current input is not a solution, a new set of inputs are generated. During the generation of outputs from this phase, which will be inputs later on, a number of tests will be run. The first test is to determine if the length of the current trees plus the next tree that is being evaluated for addition exceeds the current solution length, nothing is added to the outputs, otherwise the solution might still be better and is till considered for continued processing. The next test is whether the currently con-

sidered collection of trees remains a valid solution, based on the compatibility matrix. If both of these tests are passed, the collection of trees, and its total distance are stored to be passed out to the master process. The outputs from each iteration of the CUDA kernel are offloaded to the master process, which is responsible for storing the unprocessed results to be passed to future iterations of the CUDA kernel. The unfortunate reality of what's currently available for GPU hardware is that memory is a premium. A possible downside to doing GPGPU computations is the amount of available RAM. Current mid-range graphics cards, such as the GTX 780 Ti [23] mentioned in Section **??**, are configured with 3 GB of RAM. This can be problematic, especially for this application, because of the amount of data that can be produced at the end of each iteration of the solver. Each of the input can possibly generate at most $n-1$ outputs. Due to how the data is stored this means that the output size grows as follows, per generated value: 1 32 bit integer for length, 1 32 bit floating point value for the distance and $n\%32+1$ 32 bit integers for the bit vector representing tree inclusion. These values have to be multiplied by the possible number of outputs to determine how much memory has to be allocated per input. Table 3.1 shows a breakdown of how much memory is required as the number of terminal nodes grows. While this does not initially look dire, with even a 256 tree input can only produce 10200 KB of output, the next step to determine how much memory will be used is to then allocate that much memory per thread This implementation was able to run with 1024 blocks of 256 threads for a total of 262144 total threads requires 2673868800 B, approximately 2.6 GB, of memory dedicated just to outputs for the case of $n = 256$.

3.3 OpenMPI

Similar to the solution discussed in Section 3.2 of Chapter 3, the OpenMPI [9] solution makes use of a master and worker approach. This solution is, in fact, almost identical to the CUDA solution, with a few minor exceptions. The first, and greatest, difference is that this code is designed to run not on the GPU, but instead on multiple CPUs, using OpenMPI to facilitate passing between the master and worker processes. The role of the master process remains largely the same between the two code bases, including using

n	length	dist	trees	total
1	4	4	4	0
32	4	4	4	372
64	4	4	8	1008
96	4	4	12	1900
128	4	4	16	3048
160	4	4	20	4452
192	4	4	24	6112
224	4	4	28	8064
256	4	4	32	10200

Table 3.1: The rate at which memory requirements grow; all values are in bytes

almost all of the same data structures. The major change is in how the communication is handled, the OpenMPI library.

Unlike with a GPU the resource dynamics are reversed. The amount of memory per CPU core will be much higher, the hardware referenced in Chapter 4 has 16 GB of RAM available per compute core. With the additional resources it becomes more feasible to work on multiple sets of trees for each pass. With the additional work, a shortcut that was worked into the CPU based worker processes was to shorten the compatibility check. A disadvantage to CUDA is that there's a performance penalty when threads within a block end up running different instructions. This condition, referred to as warp divergence, requires careful planning to avoid. With a CPU based solver this was a less of an issue than with the CUDA; each thread can be running any arbitrary operation without interfering with each other when run on the CPU, and each thread generally has access to far more RAM than on a GPU. This adds an additional possibility for "wasting" time, while waiting, either for new data to work on or to send data back to the master thread to be aggregated for later dissemination, however, because there is not a synchronized point in the processing when all of the scattered processes are guaranteed to be at the same point in their execution. The adopted solution is to have the master process poll for data from the worker processes, specifically a count of how many results will be returned back, and have the master process idle until it has work to do. This

approach leaves, as the number of available processors are increased, an decreasingly small amount of resources idle, and with a larger number of worker threads, the less likely it becomes that data is not being passed back to be stored.

Chapter 4

Results

4.1 Hardware

The hardware for the OpenMPI results were configured as:

- Two E5-2650v2 2.6GHz Intel 8 Core CPUs with hyperthreading disabled
- 256 GB of RAM
- Intel X520 DP 10Gbps interconnect
- Connected through a Brocade VDX switch

The hardware for the CUDA results were configured as:

- Two E5-2620 2GHz Intel 6 Core CPUs with hyperthreading enabled
- 64 GB of RAM
- Eight nVidia GTX 780 GPUs with 3 GB of ram each

4.2 Results

Using the process proposed in Chapter 3, an algorithm was designed and written to take advantage of the additional opportunity for parallel computations. Figure 4.1 shows the average reunite, in ms, for the OpenMPI based solver. As the number of presented CPUs increases the total run time decreases, which is expected given the high degree of

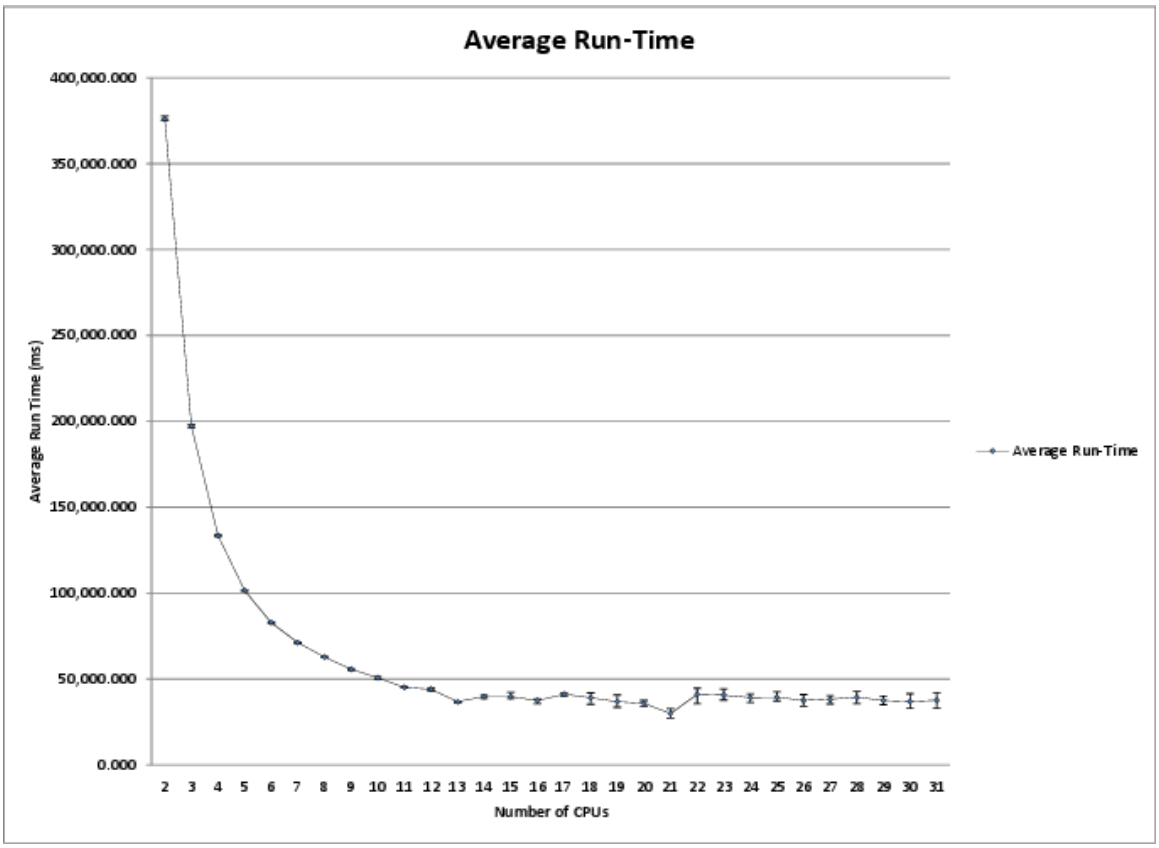


Figure 4.1: Run time for a varied number of CPUs (85 sub-tree/27 node input)

parallelism. Looking at the next set of figures, the elements composing the various runs can be discerned.

Figure 4.2 shows the amount of time in ms, across all of the worker threads across the entire run time, that worker threads sat idle. A steady decrease in idle time can be observed up through around 13 CPUs, with an increase in idle time past that point. The increase idle time at 14 CPUs is caused by contention between the various worker threads over the coordinating thread. What's measured by "idle" time is the length of time that a worker thread was without data to work on, and is relying on the coordinating thread to provide it with new input to work on. As a result of there being only a single coordinating process, for this implementation, there reaches a point at which the serial process becomes a bottleneck, unable to deal with data as quickly as it's coming in. The high variance in time spent idle takes into account that some threads will get through imme-

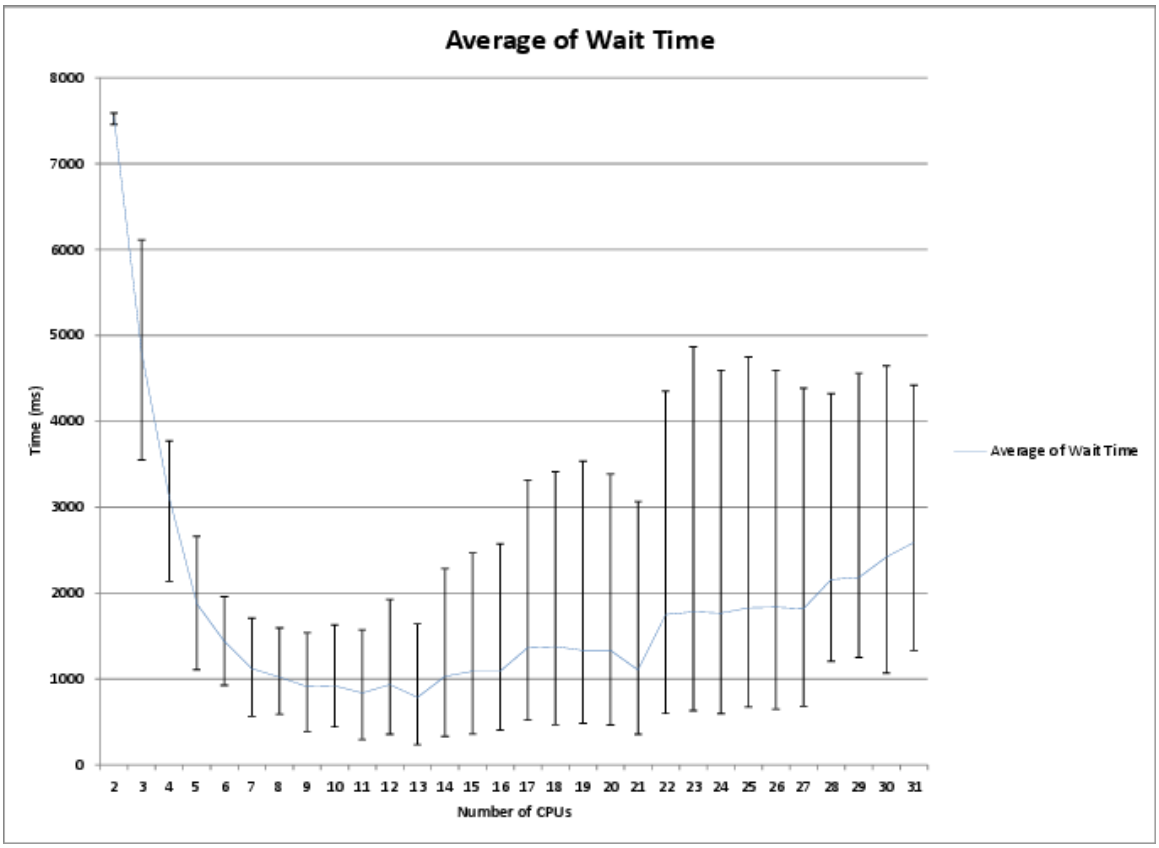


Figure 4.2: Worker thread time spent idle (85 sub-tree input/27 unique edge points)

diately, and not have a high wait time, while some of the threads will inevitably get stuck in queue.

Figure 4.3 depicts the time spent in network calls. Similar to the idle time metric, there exists a minimum at 13 CPUs, with both the amount of time spent in these calls increasing as more threads are added. Similar, again, to the idle time metric the bottleneck causing this behavior is the parallel processing becoming bottlenecked at the coordinating node. Despite the MPI sends being asynchronous, not requiring a blocking session, the amount of work done by the coordinating node causes it to not check for available data, causing the sending thread to become stalled until the transfer completes.

The last metric collected for the CPU based implementation is time spent working. Figure 4.4 depicts how much time was spent, per thread, on performing productive calculations. Due to the larger scale of that particular graph, Figure 4.6 contains a ta-

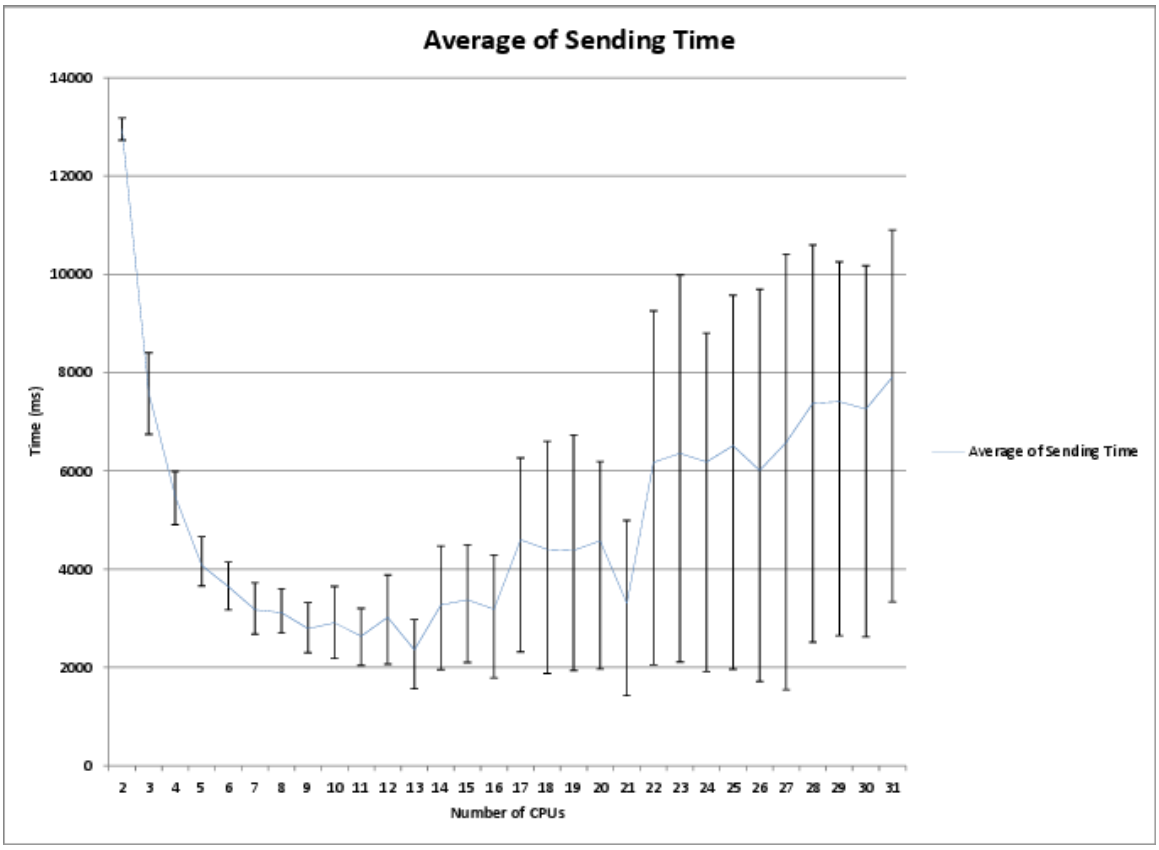


Figure 4.3: Worker thread time spent on network calls (85 sub-tree input/27 unique edge points)

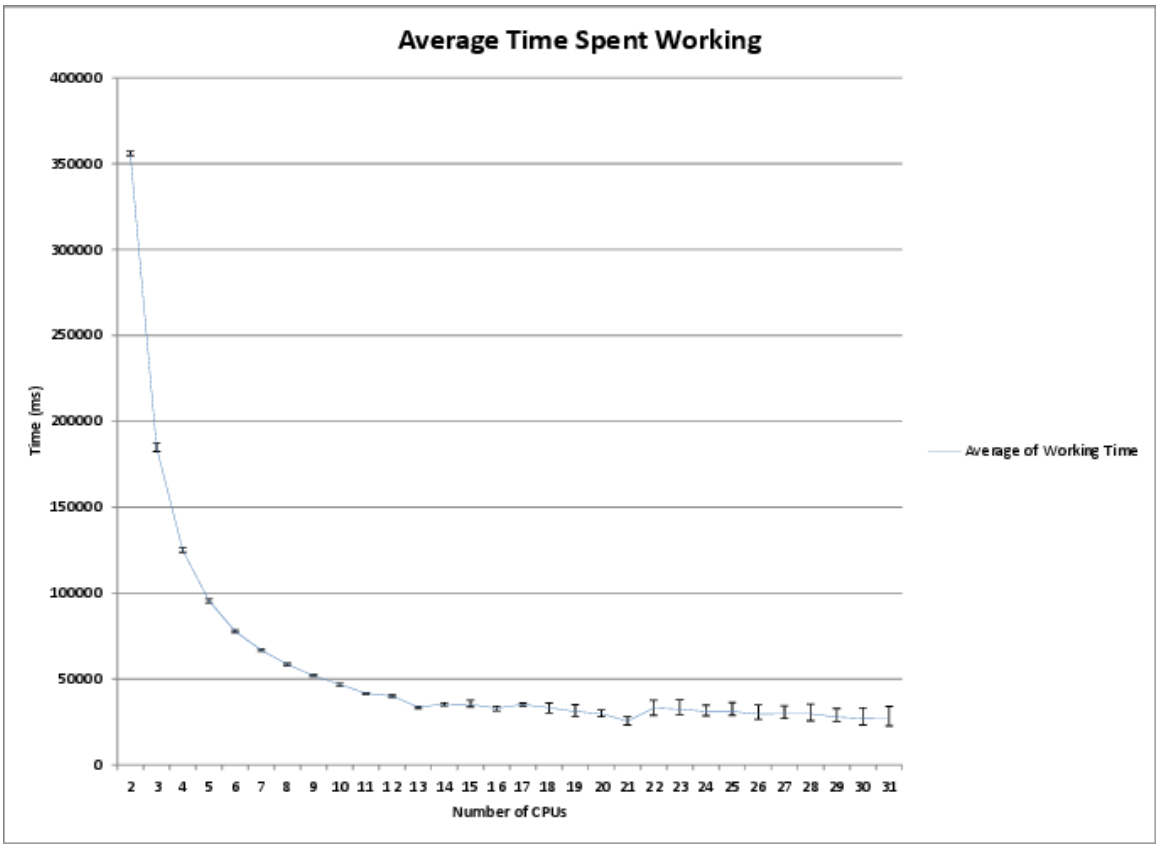


Figure 4.4: Worker thread time spent doing computations (85 sub-tree input/27 unique edge points)

ble of the data used to generate the plot. Despite the increases in time spent waiting, Figure 4.2, and time spent in network IO, Figure 4.3, working time trends in a generally decreasing manner. This is because the data represented in Figure 4.4 is the total time over the course of generating a solution that processes spent working. This decrease in required time is an encouraging result, indicating that less time is needed per CPU as additional CPUs are added in.

The overall speedup and efficiency for the OpenMPI program can be seen in Figure 4.5. Speedup was computed as $S = \frac{T_2}{T_n}$, where T_n is the time spent with n CPUs and T_2 is the runtime for the case where $n = 2$. Near linear speedup can be seen up until 11 CPUs, and past that point there is no fixed pattern of speedup or slowdown, with a plateau between a speedup of 9 and 10. The portion of the graph that is not in

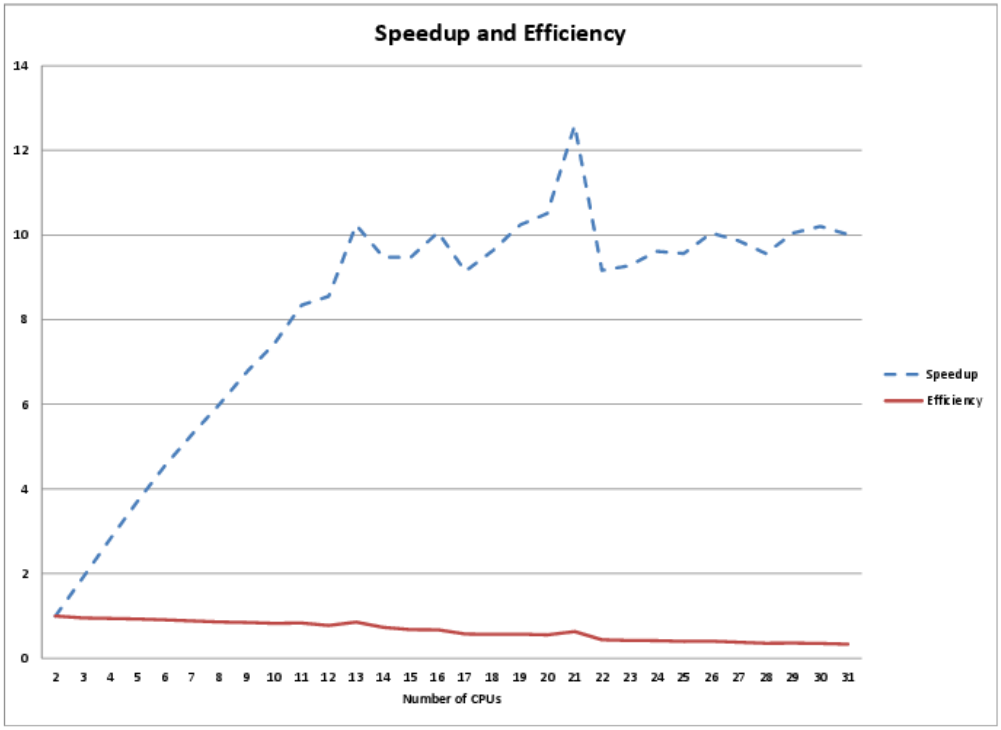


Figure 4.5: Speedup and efficiency for the OpenMPI algorithm

a discernible pattern can be explained by the cost of communications, specifically that beyond 11 CPUs is where the cost of adding an additional CPU brought increased communication cost. Efficiency, on the other hand, was computed as $E = \frac{S_n}{n}$, where S_n is the calculated speedup for n processors, and n is the number of processors. Despite observed increases in speedup efficiency is seen to decrease as additional CPUs are added. this decrease in efficiency can be attributed to the higher cost of communication as each additional CPU is added to the set.

The CUDA results are shown in Figure 4.7. The total times are not as good as would be expected, with the total time to run being much worse than all but the 2 or 3 CPU cases (1 or 2 worker threads). The most noticeable operations that consumed compute time were memcopies (transitioning data into and out of GPU memory), inserts (on the

CPU Count	Average run time	Deviation over average	Deviation under average
2	375,987.447	1,528.311	794.475
3	197,242.615	631.602	951.555
4	133,353.567	325.658	451.513
5	101,418.218	221.056	190.709
6	82,793.507	329.681	359.852
7	71,085.376	221.599	424.378
8	62,679.415	339.563	328.530
9	55,602.947	319.153	452.573
10	50,661.399	565.518	770.815
11	45,081.210	195.503	321.639
12	43,978.661	629.574	820.358
13	36,736.456	404.478	712.543
14	39,678.322	905.141	1,113.461
15	39,693.592	2,513.385	1,202.485
16	37,374.648	769.417	1,726.080
17	41,124.806	823.252	1,181.962
18	39,057.179	2,758.457	3,823.185
19	36,742.410	3,875.882	3,251.961
20	35,784.666	1,741.198	1,659.946
21	29,860.324	2,786.590	2,588.256
22	41,036.910	3,511.293	5,440.161
23	40,542.227	3,611.424	2,786.803
24	39,121.094	2,135.784	2,804.364
25	39,327.872	3,150.254	2,061.030
26	37,429.495	3,495.221	3,364.651
27	38,127.223	2,203.983	2,622.255
28	39,320.267	3,384.035	3,676.564
29	37,430.998	2,474.077	2,382.423
30	36,854.374	4,524.546	3,925.281
31	37,562.134	4,408.150	4,709.208

Figure 4.6: Raw data for run times

host adding the data to be worked on into long term memory) and reads on the host out of long-term memory. The uniformly largest consumers of time were inserts and retrieves from the long term storage kept on the host. This storage is a data structure specialized for this application, with special considerations taken into account for the rate of growth that is observed in memory consumption. The targeted nature of this data structure, an array of linked lists sorted by length, that is where in the bit vector the last bit was set, helps keep the memory footprint of the application down. Inputs that have a longer distance cannot make as many outputs, and have been optimized for being selected first, which results in a slightly more stable memory footprint over time. The cost, however, associated with this structure is in having to touch several lists to place data in the correct locations for later use. With the large scale at which new data can be generated, it is trivial for the CPU bound thread that controls this data structure to become the bottleneck for computations. A stacking of work, where the CPU bound process did the inserts while the GPU threads did each batch of computations was considered, but because of how little time was required to complete each round of computations, the same stacking up effect was observed.

The third highest consumer of time was the use of memcpys between the host and the device. Unfortunately, as discussed in Section **??**, there is not enough memory to do the entire process on the GPU. This process of shuffling data across the PCI bus presents a non-trivial work stoppage that serves two unfortunately necessary functions. The first functional reason for having this transfer is to avoid running out of memory in the GPU output buffer by offloading as much data as possible to the CPU bound coordinating process. This leaves as much space as possible available to do computations in, and avoids the problem of underestimating the amount of memory needed overall. The second reason for handling memory management in this manner, for this iteration of the code, is to alleviate the placement of items in memory. As GPU threads finish their work, they have a dedicated space in a shared array for storing results. By having the coordinating thread on the host handle work assignments, the overhead of solving for array compaction is alleviated.

Run	Total	Insert (host)	Mem Copies	Data reads (host)	Reduce (device)	Memset (host/device)	Main Kernel (device)
1	12128327	2508208.082	6254182.239	2272549.769	0.107	11622.805	3172.287
2	12107307	2598544.815	6167095.599	2326829.472	0.110	11523.093	3099.145
3	11950473	2556518.289	6092710.089	2281914.471	0.110	11014.895	3409.229
4	11846055	2609687.906	5801411.181	2417331.436	0.116	11281.914	3268.981
5	117400790	2613413.368	5761355.538	2344166.657	0.114	11650.457	3136.845

Figure 4.7: GPGPU run times, all times in ms

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The Steiner Minimal Tree problem has a number of well understood solutions. The smallest case, where there are only 3 points, was solved by Pierre de Fermat in the 17th century. Fermat's work provided the basis for solving the more complicated case of minimizing the graph for an arbitrary number of points, as posed by Jakob Steiner in the early 19th century.

A number of computational algorithms have been proposed to solve this problem. As a result of the available hardware, the solutions provided by Cockayne and Hewgill [5] and Winter [28] largely focused optimizing what geometry was generated to speed up their serial code. A number of improvements were made to the serial code [6, 29] that greatly shortened the run times of these serial applications. In 1994 Harris proposed a parallel algorithm [10] that improved on Winter's algorithm, and provided an order of magnitude decrease in the processing time required to solve the Steiner Minimal Tree problem.

The work presented here attempted to improve on Harris' work by attempting to add an additional parallel solver to assist with handling of the most computationally intensive portion of the problem - that is, determining which set of generated trees form the shortest solution. It was observed that the traversal of the decision tree (determining which trees to include in the solution) could be parallelized in addition to Harris' existing work in splitting the larger overall graph into more manageable sections. A pair

of parallel solvers was designed to take advantage of the fact that all branches of a given decision tree can be traversed simultaneously. The first solver is a GPGPU application, written in CUDA, which attempts to solve as many nodes on the decision tree as possible in parallel. It was determined that the GPGPU application solver will find a solution given enough time, but is limited by the necessity of frequently transferring data between the GPU and the application running on the host computer. This limitation is brought about by the rate at which the compute kernel's memory requirements grow and the relative lack of memory available in GPU hardware. The second solver was a multi-server parallel solution that utilized a message passing library to handle communications. As a result of how the communications could be staged in this implementation, the timing results were better than were observed with the GPGPU implementation, requiring almost $100x$ less time on average to run. Despite the better results seen with the multi-server approach, a higher than anticipated amount of each thread's time was spent unproductively, either sending/receiving data or waiting on new data to become available. As a result of this cost of adding additional CPUs, the overall efficiency of the algorithm was fairly low.

5.2 Future Work

Given the lackluster results from the presented implementations, there remains a good deal of future work for the algorithms presented here. The GPGPU solver can be improved by preventing the constant transfer of data between the GPU and the host. This can be accomplished by the creation of thread safe array storage in GPU memory, or by better arranging the inputs that are passed into the compute kernel to prevent the generation of more outputs than can be processed as inputs. The OpenMPI solver would also greatly benefit from a reduction in communications, and could potentially be solved by adding additional space on the worker threads to store inputs for later consumption, only offloading to the master thread as needed.

Winter and Zachariasen [29] provided an improvement to the previous algorithm that greatly decreased the complexity of the possible solutions. This improvement al-

lowed for the solving of problems up to 140 points, up from 15 as presented in Winter's 1985 paper [28], by greatly reducing the number of Fermat points generated, which leads to a reduction in the number of trees that are generated. There are a number of applications in this process that can be parallelized, such as placing the Fermat points and building the connected trees. A full rewrite of Winter and Zacheriasen's improved solution, parallelizing it where possible, could provide significant decreases in the required runtime. Parallelizing the placement of Fermat points, done early on in the algorithm, would be expected to provide some runtime benefits. A large number of comparisons are needed, generating a Fermat point for all adjacent pairs of points with their neighbors, that do not have any dependence on each other. Harris' improvements to the earlier version of Winter's algorithm [10], in parallelizing the concatenation step, as well as the solution presented here, would also be applicable improvements, likely providing large appreciable performance gains.

Bibliography

[1] John E. Beasley. A heuristic for euclidean and rectilinear steiner problems. *European Journal of Operational Research*, 58(2):284–292, 1992.

[2] John E. Beasley and F. Goffinet. A delaunay triangulation-based heuristic for the euclidean steiner problem. *Networks*, 24(4):215–224, 1994.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[4] C.a.R. Compass and Ruler - Construct and Rule. `http://car.rene-grothmann.de/doc_en/index.html` (Accessed Aug 2, 2015).

[5] Ernest J. Cockayne and Denton E. Hewgill. Exact computation of steiner minimal trees in the plane* 1. *Information Processing Letters*, 22(3):151–156, 1986.

[6] Ernest J. Cockayne and Denton E. Hewgill. Improved computation of plane steiner minimal trees. *Algorithmica*, 7(1):219–229, 1992.

[7] M. Fampa and K.M. Anstreicher. An improved algorithm for computing steiner minimal trees in euclidean d-space. *Discrete Optimization*, 5(2):530–540, 2008.

[8] Dominik Göddeke. *Gpgpu-basic math tutorial*. Univ. Dortmund, Fachbereich Mathematik, 2005.

[9] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2006.

[10] Frederick C Harris Jr. *Parallel Computation of Steiner Minimal Trees*. PhD thesis, Clemson University, Clemson, SC, May 1994.

[11] Frederick C Harris Jr. Parallel computation of steiner minimal trees. In David H. Bailey, Peter E. Bjorstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virgia J. Torczan, and Layne T. Watson, editors, *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 267–272, San Francisco, California, 1995. SIAM.

[12] Frederick C Harris Jr. Steiner minimal trees: An introduction, parallel computation, and future work. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization Vol II*, pages 105–157. Kluwer Academic Publishers, Dec. 1998.

[13] Frederick C Harris Jr and Rakhi C Motwani. Steiner minimum trees: An introduction, parallel computation, and future work. In Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham, editors, *Handbook of Combinatorial Optimization, 2nd Edition*, pages 3133–3177. Springer, 2013.

[14] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[15] Clark Kimberlings. *ENCYCLOPEDIA TRIANGLE CENTERS*. 2012.

[16] Cut The Knot. The Fermat Point and Generalizations. `http://www.cut-the-knot.org/Generalization/fermat_point.shtml` (Accessed July 19, 2015).

[17] Donald E Knuth. The art of computer programming, volume 4, fascicles 0-4. 2009.

[18] Jens Kruger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society, 2003.

[19] David Luebke and Greg Humphreys. How gpus work. *Computer*, (2):96–100, 2007.

[20] Z.A. Melzak. On the problem of steiner. *Canad. Math. Bull*, 4(2):143–148, 1961.

[21] Microsoft Developer Network. Fundamental Types (C++). `https://msdn.microsoft.com/en-us/library/cc953fe1.aspx` (Accessed July 19, 2015).

[22] Nvidia. CUDA. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (Accessed July 23, 2015).

[23] Nvidia. GTX 780 Ti. `http://www.nvidia.com/gtx-700-graphics-cards/gtx-780ti/` (Accessed July 26, 2015).

[24] ARB OpenGL, Mason Woo, Jackie Neider, and Tom Davis. Opengl programming guide. *Addison-Wesley*, 1999.

[25] JF Rigby. Napoleon revisited. *Journal of Geometry*, 33(1-2):129–146, 1988.

[26] Randi J Rost, Bill M Licea-Kane, Dan Ginsburg, John M Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL shading language*. Pearson Education, 2009.

[27] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[28] Pawel Winter. An algorithm for the steiner problem in the euclidean plane. *Networks*, 15(3):323–345, 1985.

[29] Pawel Winter and Martin Zachariasen. Euclidean steiner minimum trees: An improved exact algorithm. *Networks*, 30(3):149–166, 1997.