Parallel algorithms for geometric shortest path problems

Alistair K Phipps

Master of Science
Computer Science
School of Informatics
University of Edinburgh

2004

Abstract

The original goal of this project was to investigate and compare the experimental performance and ease of programming of algorithms for geometric shortest path finding using shared memory and message passing programming styles on a shared memory machine. However, due to the extended unavailability of a suitable shared memory machine, this goal was only partially met, though a system suitable for testing the hypothesis was implemented. The results gained indicated that the programming style did not have a major impact on run time, though the shared memory style appeared to have a lower overhead. It was found that the message passing style was both easier to program and required less code.

Additional experiments were performed on queue type and partitioning method to determine their impact on the performance. It was found that use of a sorted queue had a serious negative impact on the parallelisability of the shortest path algorithms tested, compared with use of an unsorted queue. The use of a multilevel over-partitioning scheme (multidimensional fixed partitioning) gave improved performance with an asynchronous parallel algorithm (by Lanthier et al.), but worsened the performance of a synchronous parallel algorithm (simple parallelisation of Dijkstra's algorithm).

Acknowledgements

I would like to thank my supervisor, Murray Cole, for initially proposing this project and for his help and advice throughout.

I would also like to thank my parents for their support and for giving me the time and space I needed to complete this project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Alistair K Phipps*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Shared memory vs Message passing

The memory architecture of parallel computers can be separated into two main subtypes:

- Shared memory architecture, where all processors share a common memory. The memory can be accessed by any of the processors and data is passed between the processors by storing and retrieving from this shared memory.
- Message passing architecture, where each processor has its own private memory. Each processor can only access its own memory, and data is passed between the processors by the sending and receiving of messages.

These architectures have different ways of passing data between the processors, but is also possible to emulate the mechanisms of one architecture on the other - message passing can be carried out on a shared memory machine, and distributed shared memory can emulate shared memory on a message passing machine.

Several issues affect performance of parallel algorithms, and their impact differs for the different memory architectures. Key relevant issues are:

- Cache coherence. In a shared memory system, whenever one processor needs to read data that another processor has written, its cache must be updated. If multiple processors are reading and writing data on the same cache line, this means that the cache is never effective as it must be constantly updated. This can have a big impact on the performance of algorithms on shared memory systems. In contrast, a distributed memory system using message passing has a separate cache for each processor which is not invalidated or updated directly by other processors, and so cache coherence is not such an issue on message passing systems (or no more so than on a uniprocessor system).
- Synchronisation time. If processors must synchronise at some point in the execution of a program, such as at an explicit barrier, or simply where one processor needs to read data that another processor is writing, this can impact the performance of the program. Synchronisation time depends

on the particular algorithm and system in use, but for the same algorithm, the synchronisation time will tend to be longer for message passing systems due to the overheads involved in communication.

- Granularity, defined as the amount of code executed between synchronisation events. Clearly, a fine-grained algorithm with frequent synchronisation is a problem for either architecture, but more so for message passing systems due to the slower synchronisation time.
- Degree of parallelism. With either architecture, unused processors cause a reduced efficiency, but there is no difference between the architectures unless some kind of process migration is in use, which is not considered here.
- Data locality. Whenever data must be transferred between processors, there is a performance impact. To gain the maximum performance, as much data as possible that much be used at a processor should be available at that processor. This is not a problem in shared memory systems, where all the data is shared, but can be a big factor in message passing systems where data in the wrong location must be passed in a message.

In general these issues affect performance due to the nature of the architecture rather than the nature of the programming style. However, the programming style may also have an effect. For example, cache coherence may be less of an issue when using message passing on a shared memory machine, because the data is copied for use by different processors. Conversely, data locality may become a bigger issue due to the overhead of putting the data into messages and passing it locally.

Some work has been done in the area of comparing the performance of algorithms suited to implementation with either a shared memory or message passing programming style, using the appropriate style on its native architecture.

Fujimoto and Ferenci [1] demonstrated a TCP-based message passing system, such as that used by default with LAM/MPI, has a significantly higher data sharing latency and overhead than a shared memory system, though results with a high performance interconnect such as Myrinet [2] are much closer to that in a shared memory system. Their message passing implementation of a particular algorithm became faster than the shared memory implementation as the number of processors was increased beyond eight.

Clinckmaillie et al. [3] showed that for one particular algorithm, a message-passing implementation can give rise to a greater efficiency than a shared memory implementation. In that example, a better degree of parallelisation, a more efficient use of memory, the overlap of I/O and computation and the absence of cache coherency effects were credited with giving message passing the advantage.

Chandra et al. [4] performed a detailed comparison of both shared memory and message passing implementations of four different algorithms on simulators of the different machines. They determined that message passing and shared memory architectures were approximately equivalent in efficiency for 3 of their 4 programs, with the fourth being faster using message passing due to the costly directory-based invalidate cache coherence algorithm of their simulated shared memory machine.

Some research has also examined the performance of distributed shared memory (DSM) against message passing on a message passing architecture. For example, Dwarkadas et al. [5] compared the performance of various benchmarks on a software DSM system against message passing and found that the DSM was between 0% and 29% slower than message passing. A shared memory style implementation executing on DSM must inevitably be slower than a well-programmed message passing implementation because of the overheads involved.

At least one study has examined the performance of algorithm implementations using a shared memory programming style against those using a message passing programming style on a shared memory architecture. Fiedler [6] compared the performance of two different parallelisations of a hydrodynamics algorithm on a 64-processor shared memory machine - one suited to message passing and one to shared memory. The message passing implementation was found to scale better than the shared memory implementation due to the shared memory implementation causing a large amount of cache misses. However, it should be noted that the parallelisations were different so the study just implied that a message passing style enforced a pattern of data accesses, at least for this particular algorithm, that reduced issues of cache coherency.

Other than the performance differences between message passing and shared memory programming styles, there is also the issue of differences between ease of programming of the two styles.

The ongoing research into DSM, despite the fact that it is always slower than message passing, is based on the notion that shared memory is an easier paradigm for which to program. In one paper on software-based DSM [7], Nikhil et al. state, “it is now widely conceded that [explicit message passing] is too difficult for routine programming”. More recent papers in the field [8] have continued with the work based on this assumption, despite the widespread availability of message-passing frameworks such as the Message Passing Interface (MPI) [9] which simplify the task.

The OpenMP - an Application Programming Interface (API) for shared memory parallel programming - proposal [10] acknowledges the recent popularity of message passing as opposed to shared memory programming, but states the main reason as portability rather than performance or ease of programming. It does acknowledge that POSIX threads may be too low level for the typical scientific application programmer, compared with MPI for message passing.

1.2 Shortest path algorithms

1.2.1 Shortest path definition

A weighted undirected graph is defined by $G = (V, E, w)$ where V is a finite set of vertices, E is a finite set of edges and $w : E \rightarrow \Re^+$ is a function assigning a positive real weight to each edge. Let $d(s,t)$ denote the distance from vertex s to t, defined as the sum of weights along the shortest path between s and t. Let $\pi(s,t)$ denote the shortest path between s and t.

As the graph becomes very large, finding $d(s,t)$ becomes slow. However, it is possible to partition the graph and parallelise the shortest path computation in order to speed up the process.

1.2.2 A note on pseudo-code

Pseudo-code is used in this report to describe the operation of algorithms. Block constructs, such as for-loops and if-statements are expressed using Python notation, where a colon ‘:’ indicates the start of the block and the same level of indenting is used throughout the block, indicating where it ends.

For example, a C fragment:

```
if ( a == b ) {
  while( c[0] > 1 ) {
    c[0]--;
  }
  d = 0;
} else {
  c[0] = 1;
}
```

Would be represented as:

```
if a = b:
  while c[0] > 1:
    c[0] = c[0] - 1
  d = 0
else:
  c[0]= 1
```

1.2.3 Dijkstra sequential implementation

Dijkstra’s algorithm [11], one of the first algorithms created for shortest path finding, determines the shortest path from a single source vertex to every other vertex.

The input data is a source vertex, s and a graph G. The vertices are split into two sets: S, the set of settled vertices, for which the shortest path distance $d(s,u), u \in S$ and shortest path $\pi(s,u), u \in S$ is known, and Q, the set of unsettled vertices, for which we only have a current best estimate of the shortest path. Initially all vertices are unsettled.

A pseudo-code representation of the algorithm is given here. $d(s,u)$ is represented as an element in an array of costs, $d[u]$. The path, $\pi(s,u)$, is represented as a linked list of references to previous vertices, each element being in $previous[u]$. The edge weights are represented as $w[u,v]$, giving the weight for the edge between vertices u and v. The algorithm proceeds as follows:

```
V = set of all vertices
for each v in V:
  d[v] = infinity
  previous[v] = undefined
d[s] = 0
S = empty set
Q = V
while Q is not an empty set:
  u = extract-min(Q)
  S = S union {u}
  for each edge (u,v) incident with u:
    if d[v] > d[u] + w[u,v]:
      d[v] = d[u] + w[u,v]
      previous[v] = u
```

At each step, the unsettled vertex with the shortest distance to the source is made settled as the best path has been found to it. The vertices adjacent to that one are then tested to see if there is a path via this newly settled vertex to the source that is shorter than their previous shortest. If so, the best known distance and previous reference are updated. This is known as the relaxation step - it is like the end of a piece of elastic (vertex v) relaxing back into place after the other end (vertex u) has been pulled away (by vertex v being notified that there is a shorter path via u).

The *extract-min* step can be implemented naively as a linear search through Q, in which case the algorithm has run time $\Theta(n^2)$, where n is the number of vertices. The use of different kinds of queue is examined in Section 1.2.6.

1.2.4 Dijkstra Parallelisation

The standard parallelisation of Dijkstra's algorithm is simply a partitioning of the graph, with Q being shared to extract the minimum vertex. It is commonly described in terms of partitioning of the adjacency matrix. However, since in this project there was no explicit representation of the adjacency matrix, it will be described in terms of the partitioned graph.

The algorithm proceeds as follows on each processor:

```
V = set of all vertices in partition
S = empty set
for each v in V:
  d[v] = infinity
  previous[v] = undefined
d[s] = 0
Q = V
while Q is not an empty set:
  q = extract-min(Q)
  u = global-min(q)
  if u is a member of set of V:
    S = S union {u}
    for each edge (u,v) incident with u:
      if d[v] > d[u] + w[u,v]:
        d[v] = d[u] + w[u,v]
        previous[v] = u
```

Each processor has a local Q and the global minimum of local minimum queue entries is found and used by all processors as the chosen vertex with which to find incident edges.

If this is implemented on a message passing architecture with p processors, *global-min* can be implemented as a all-to-one reduction and then broadcast, where each takes time $\Theta(\log p)$. If there are n vertices in total, and the vertices are split equally amongst the processors with no overlap, there are $\frac{n}{p}$

vertices at each node. The while loop must be executed once for every global vertex, but *extract-min* now only takes time $\Theta(\frac{n}{p})$. The parallel runtime is therefore $\Theta(\frac{n^2}{p}) + \Theta(n \log p)$.

On a shared memory system, *global-min* can be simply implemented in time Θp, though with a much smaller constant factor than the message passing reduction. This gives a runtime of $\Theta(\frac{n^2}{p}) + \Theta(np)$.

Note that each loop of the algorithm operates in lock-step on each processor - it is a synchronous algorithm. Additionally, in a message passing setting the processor must wait for the communication to complete before it can proceed, so no overlap of communication and computation is possible. This need for frequent synchronisation is a limit on the performance of the algorithm as the number of processors increase.

1.2.5 Lanthier Parallelisation

The algorithm given by Lanthier et al. [12], hereafter referred to as the Lanthier algorithm, is specifically designed for implementation in the message passing style. It is based on Dijkstra's algorithm, but modified so that individual nodes can continue processing for longer by increasing the synchronisation intervals.

An outline of the algorithm is shown below:

```
maxcost = infinity
localcost = 0
V = set of all vertices in partition
for each v in V:
  d[v] = infinity
  previous[v] = undefined
d[s] = 0
Q = s
while true:
  if all nodes are done:
    exit
  if Q is empty or localcost > maxcost:
    notify other nodes that we are done
    wait for done notification or cost update
  else:
    u = extract-min(Q)
    localcost = d(s,u)
    if localcost < maxcost:
      if u = t:
        maxcost = localcost
        update other nodes' maxcost
      else:
        for each edge (u,v) incident with u:
          if d[v] > d[u] + w[u,v]:
            d[v] = d[u] + w[u,v]
            previous[v] = u
            decrease cost of v on nodes that share it, and insert it in node's Q if
                less than other node's value for v
```

Each node has a separate *localcost* which is the distance between the source and the last vertex extracted from the queue. As elements of the queue are processed, this value will increase. However, nodes can notify others of updated costs for a particular vertex, which may cause that vertex to get added back into the queue and localcost will decrease once again.

A global *maxcost* is stored which gives the cost of the shortest path found to the target up until that point. Vertices are not removed from the queue if their cost is greater than this maxcost, as they obviously cannot provide a shorter path to the target (though they may be updated later with a lower cost, as described above).

Due to the branches in the algorithm, a simple runtime analysis is not possible. However, it can be seen that although the same vertex may be processed multiple times (if it is processed before a cost update is received), the coupling between each process is much decreased - vertices may be continually processed while they exist on the local queue. The algorithm is therefore asynchronous. In addition, the processors do not have to synchronise during the communication so communication and computation can be overlapped on a message passing architecture.

It should be noted that the algorithm, as expressed in [12], also includes a particular method for termination detection and determining the path by tracing back the previous vertices after algorithm completion.

1.2.6 Queues

The above analyses of the algorithms have been made assuming the queue used was a simple unsorted table. Such an implementation requires a linear search through the table to find and extract the entry with the minimum value, a $\Theta(q)$ operation, where q is the number of entries in the queue. Changing the value of an entry on the queue also requires a search to find the entry, a $\Theta(q)$ operation, then a $\Theta(1)$ update. Inserting an item onto the queue just involves adding it to the table and is a $\Theta(1)$ operation.

Alternatively a sorted queue, where the items are stored in a linked list that is kept sorted, could be used. Finding and extracting the entry with the minimum value then just requires reading and removing the item from the front of the list, a $\Theta(1)$ operation. However, inserting or changing requires linear searches until the correct insertion position is found, a $\Theta(q)$ operation. As the searching stops when the correct location is found, if there are few items in the queue with distance less than the inserted item's distance, the actual time will be much less than this.

A third possibility is to use a heap of some sort. The Fibonacci heap [13] gives the best runtime for Dijkstra's algorithm. With this, *extract-min* (composed of an $\Theta(1)$ *find-min* and an $\Theta(\log q)$ *delete*) is a $\Theta(\log q)$ operation and *insert* is a $O(1)$ operation.

The impact of using these different types of queue is examined in Section 2.4.3, along with a runtime analysis of Dijkstra's algorithm with each and the implications for parallelisation.

Another issue with the queues is that for parallel algorithms, there is a choice between having multiple queues, one per processor, with an operation to find the global minimum (as portrayed above), and having a single queue, shared between all processors, storing all vertices rather than just those in the partition.

Bertsekas et al. [14] compared single vs. multiple queues with label-correcting shortest path algorithms (as opposed to label-setting, such as Dijkstra's) with shared memory. They found that the multiple queue strategy was better, due to the reduction in queue contention. However, in their multiple queue version, the data in each queue was randomly assigned, rather than having a spatial relationship.

1.3 Mapping terrain to a graph

The algorithms above find the shortest path in a graph. However, one important application of shortest path finding algorithms is finding the shortest path on terrain. This geometric shortest path finding is useful when dealing with problems such as routing of roads across a landscape, and many other similar problems.

Terrain information is commonly available in Digital Elevation Model (DEM) format, which consists of an array of values giving a heightfield for the terrain. The wide array of heavily analysed graph-based algorithms suggests that some way of converting the continuous heightfield problem into a graph-based problem would be a good way of solving the problem.

A DEM can be converted to a Triangulated Irregular Network (TIN): a mesh of an arbitrary number (depending on the allowed error relative to the original heightfield) of triangles, where adjacent triangles share two vertices. The TIN is constructed using Delaunay Triangulation, with an algorithm such as Gustav and Stolfi's [15], and triangles may have weights associated with them (a weighted TIN), which may be derived from data such as land usage or gradient.

Although it is possible to treat the triangle edges and vertices as graph edges and vertices, this results in a shortest path that is only accurate in cost to within the length of the triangle edges. The triangles are of varying size and this error can be significant as compared to the total path length, so a better solution is desired.

1.3.1 Graph construction

The solution suggested by Lanthier et al. [16] is to place points known as *Steiner Points* along the triangle edges which form the graph vertices. Graph edges are placed between the graph vertices on opposite sides of each triangle, and also between adjacent graph vertices on a single side of the triangle. Where two triangles share an edge, only one set of Steiner points is placed along the edge. A weight is assigned to each graph edge, equal to the Euclidean distance between the graph vertices if the TIN is unweighted, or this distance times the weight associated with the triangle with the lowest weight that is coincident with the triangle edge if the TIN is weighted.

An example of Steiner point placement (from [12]) is shown in Figure 1.1. In the figure, $\pi(s,t)$ represents the actual shortest path between s and t, and $\pi'(s,t)$ the approximation through the Steiner points. Note that non-path graph edges are not shown on the diagram.

Figure 1.2 (from [16]) shows the graph edges and vertices that are placed on a single face.

Lanthier et al. [16] examined three methods for placing the Steiner points. A full analysis of the accuracy of the paths generated with the different methods and the implications for runtime can be found in that

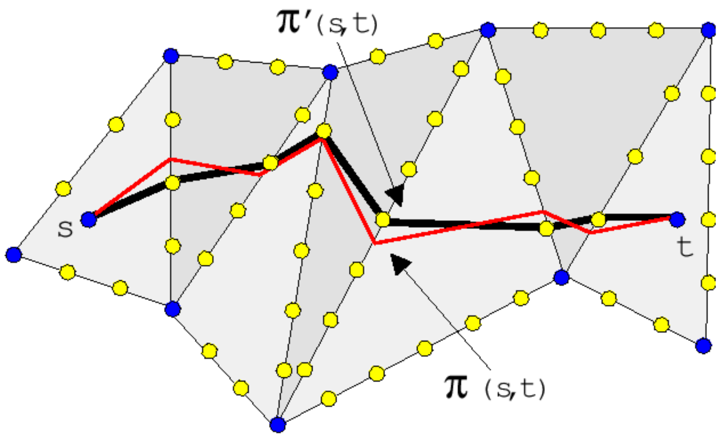

Figure 1.1: Steiner point placement example

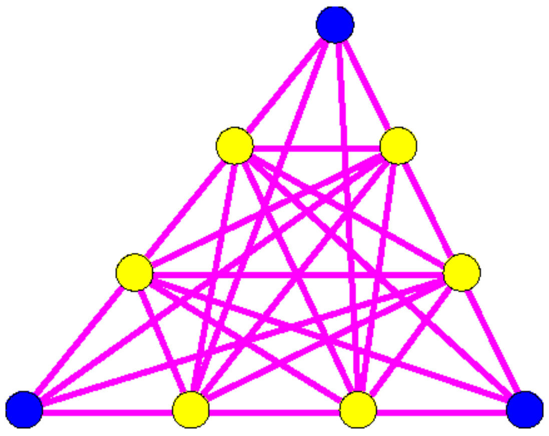

Figure 1.2: Graph constructed from terrain triangle

paper, but, in outline, the methods are:

1. *Fixed*. In which a fixed number of Steiner points are placed along each edge.
2. *Interval*. Steiner points are placed at fixed intervals along each edge, meaning a smaller number of graph vertices are required for the same path accuracy as the Fixed method. The smaller graph results in faster runtime for algorithms applied to the graph.
3. *Spanner*. A more complex method which gives a slightly lower accuracy than the Interval or Fixed schemes but is faster.

1.3.2 Terrain partitioning

As mentioned, in order to use the parallel algorithms, the graph must be partitioned between the processors. When dealing with terrain data, it is most straightforward (and necessary in this implementation - see Section 2.2.2) to partition the terrain rather than the graph.

The partitioning methods referred to here partition in two dimensions in the X-Y plane. Partitioning in this plane makes the most sense for terrain, as a TIN generated from a heightfield will never have two surface points at the same X and Y values but a different Z value, so the 3D surface can effectively be treated as a 2D surface with additional values associated with each vertex (for the Z coordinate).

In the parallelisation of Dijkstra's algorithm, the optimum partitioning splits the terrain so that there are an equal number of graph vertices in each partition, but it makes no difference which particular vertices are placed in the partitions. This is because it is a synchronous algorithm and the parallel step is the *extract-min* operation, and the resulting globally winning vertex is shared amongst all the processors. Any partitioning scheme that increases the number of vertices in each partition will cause the algorithm to slow down, but any partitioning scheme that causes the number of vertices in each partition to be unbalanced will also cause the algorithm to slow down.

The requirements of the partitioning of the terrain are quite different for Lanthier's algorithm. Each processor is free to calculate shortest paths as soon as one of its graph vertices is reached by the algorithm, and can continue afterwards without waiting for other processors due to its asynchronous nature. It is therefore important to as many different processors reach different graph vertices as soon as possible. In addition, the local shortest path calculations occur between vertices and their adjacent vertices, and as these adjacent vertices are likely to be close together in the X-Y plane, having groups of X-Y plane adjacent vertices within the same partition means the most computation can occur without excessive overprocessing due to cost updates from other processors.

The partitioning methods are given in outline below, and a more detailed description of how exactly the terrain data is split is given in Section 2.3.

1.3.2.1 Block partitioning

A simple block partitioning divides up the terrain into $M \times N$ blocks of equal area, as shown in Figure 1.3 (from [12]). This X-Y block partitioning allows the most adjacent vertices to be located within the same block, assuming the partition size is much greater than the face triangle size (so that partition boundaries become less significant). However, this does not make full use of the available processors in the Lanthier algorithm as the processors are not reached quickly. Figure 1.4 (from [12]) shows how the active border of extracted vertices (with *extract-min*) propagates out from the source vertex over time. The black circle represents the active border of computation, the white area contains unreached vertices, and the shaded area settled vertices. Initially (a), all processors but processor 6 are idle. As the border expands (b), 3, 4 and 6 are in use. Eventually (c), 2 is still not in use and 6 has become idle as all the vertices within become settled. Clearly this is not utilising the processors fully.

This partitioning scheme has some benefits for Dijkstra's algorithm as it adds only the minimum number of vertices needed to a partition - those needed to share data between adjacent partitions (see Section 2.3 for more details). However, it does not equally assign the vertices to the different partitions - the number in each will vary depending on the density of the data.

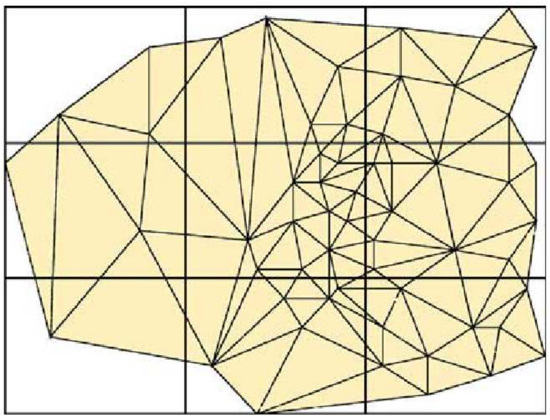

Figure 1.3: Block partitioning of a graph

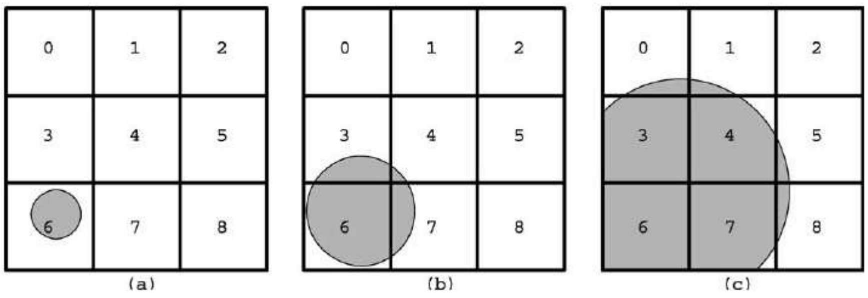

Figure 1.4: Propagation of the active border of computation with block partitioning

1.3.2.2 Multidimensional fixed partitioning

Multidimensional fixed partitioning [12] (MFP) is a multilevel partition that can make fuller use of the processors in an asynchronous algorithm like Lanthier's algorithm by reducing idling. The block partitioning is done recursively, meaning each processor has scattered portions of the block. Figure 1.5 (from [12]) shows how with a second level of partitioning, the active border quickly expands to use more processors than with just a single level. As the active border expands outwards (through (a), (b), (c)) a variety of processors continue to be used.

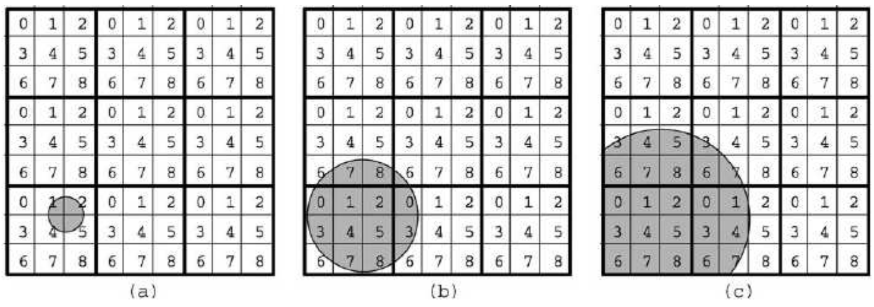

Figure 1.5: Propagation of the active border of computation with 2-level 3x3 MFP partitioning

The partitioning is done adaptively depending on the density of vertices in the partition. Figure 1.6 (from [12]) shows an adaptive 3-level, 3x3 processor configuration partitioning for a sample terrain.

Areas with few vertices are not partitioned further. There is a limit to the depth of the recursive splitting, to prevent the partitions becoming too small and communications costs outweighing the balancing of the data amongst the processors, and there is a minimum cut-off for the number of vertices that may be in a partition before it is considered for splitting.

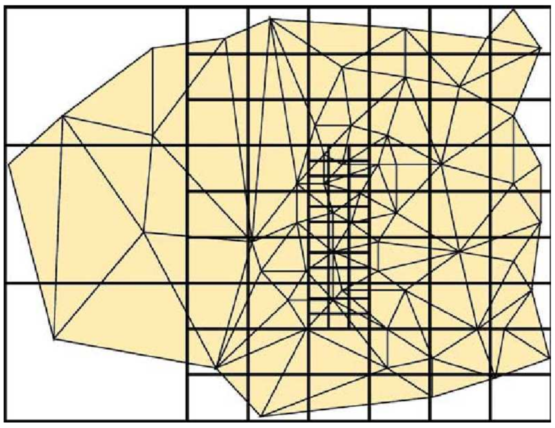
Figure 1.6: 3-level adaptive MFP partitioning of a graph for a 3x3 processor configuration

Processors are mapped to partitions using a particular MFP mapping method [12], which was used in this project but not analysed or compared against alternatives, and will not be described here.

Due to the overpartitioning of data, MFP adds vertices to each partition, and is therefore not ideal for use with Dijkstra's algorithm. It does cause the number of vertices in each partition to be balanced out, but this is only by adding vertices to each partition to bring them up to the level of the others - vertices are never taken away (at least using the splitting method described in Section 2.3). In fact, the greater the level of splitting, the greater the number of added vertices due to the increased sharing.

1.4 Project goals and achievements

The original main goal of this project was to compare practical performance of geometric shortest path algorithms with shared memory and message passing parallel programming styles on a shared memory architecture, and thereby try to reach some conclusions about the differences in performance between shared memory and message passing in general.

Performance was to be judged by the efficiency of the algorithms as the data sets and number of processors scaled, in experiments on a real system. It should be noted that the goal was not to develop the best performing algorithm possible, but merely make a comparison between shared memory and message passing implementations.

Unfortunately, satisfactorily meeting this goal became impossible due to the extended unavailability (at relatively short notice) of the main 64-processor shared memory system which was to have been used for testing. In addition, the system was expected to become available again within time to carry out performance testing, however, in the end it remained unavailable. This meant that the project could not be refocused in time to adjust for the available hardware.

The main goal was therefore weakened but broadened, to:

1. Implement a system suitable for meeting the original main goal, should the shared memory system become available (as at the time it was expected that it may).
2. Examine the performance on any other shared memory systems that were available. Only a heavily-loaded 4-way SMP machine was available, and though testing was performed on this system, the results were not reliable due to the competing loads. Clearly, full testing of algorithm scaling with number of processors was also not possible on a 4-way system.
3. Examine the performance of the message passing algorithms on a system with a message passing architecture (Beowulf cluster).
4. Determine the impact of using an unsorted queue, compared with using a sorted queue in the algorithm implementations.
5. Compare block vs. MFP partitioning for the different algorithms.

These goals have been met, with varying degrees of success.

A secondary goal was to compare the ease of programming of geometric shortest path algorithms using both styles, and again try to draw some general conclusions. Various metrics were used in order to make this comparison, and analysis of the implementations was performed.

The following chapters describe the design, implementation and testing of the system, followed by the results gained and an analysis of them. Some conclusions are then drawn from the insights gained during the project, and an assessment made about the success of the project.

Chapter 2

Design

2.1 Planning the project

In meeting the specified goals of the project, there were a variety of choices of algorithm that could have been implemented. The Dijkstra and Lanthier algorithms were chosen as a basis for comparison, the first since it is so well studied and relatively straightforward, and the second because it provided an algorithm that had been designed specifically for use with terrain data in message passing systems, and some data was already available on performance [12].

It was intended that both a shared memory and message passing implementation of each algorithm would be completed and evaluated. However, a shared memory Lanthier implementation was not completed, although some design work on the problem was carried out. It was not completed due to a combination of lack of time and because it was decided to deemphasise this implementation in favour of improving the others as it became clear the main shared memory machine to be used for testing would not be available.

Since the primary function of the algorithms implemented was essentially processing of data, a data-driven design of the system was undertaken. The design, implementation and testing was carried out iteratively and not as a sequential 'waterfall model' of development. This gave a lower risk of complete failure of the project, since it meant some development was started as soon as possible, and was successful in that even though the shared memory Lanthier implementation was not fully completed, the other algorithms were completed and tested. Using a waterfall model may have meant that all four algorithms were incomplete.

The planning of the project focussed on setting dates for specific milestones. An outline of the plan was:

- Complete initial project plan: outline objectives and milestones and timetable for completion.
- Complete literature review phase: gather papers and other documents relating to the topic and critically assess this previous work, determine where this project fits in and what previous work will be useful.
- Gather terrain data suitable for use in testing algorithms.

- Load terrain data into suitable data structures.
- Complete core data structures (generating Steiner points) and test with sequential Dijkstra implementation.
- Complete implementation of block partitioning.
- Complete implementation of MFP partitioning.
- Complete message passing implementation of parallel Dijkstra.
- Complete message passing implementation of Lanthier algorithm.
- Complete shared memory implementation of parallel Dijkstra.
- Complete shared memory implementation of Lanthier algorithm.
- Complete performance testing on SunFire.
- Complete dissertation.

Later in the project, due to the SunFire being unavailable, the performance testing milestone was modified to testing on the 4-way SMP machine and the Beowulf cluster.

The project timetable was replanned whenever it appeared that a milestone would be missed. This led to the milestones slipping until finally one had to be excluded. In retrospect, pushing extra hard to meet each milestone would have been a better alternative, and will be the method used in future projects.

2.2 Data structures

2.2.1 Data input

The first task in designing the algorithms was to determine the form of input data that they would use. The MFP algorithm was specifically meant for use with Triangulated Irregular Network (TIN) data, so TINs were used as the initial input for all the algorithms.

A TIN consists of a flat list of triangles (faces), each face containing the coordinates of its three vertices. The shortest path algorithms required the following information on the terrain to carry out their processing:

- The terrain vertex locations. These were involved in the shortest path calculations, as the graph edge cost was partially determined by the location of the graph vertices, which themselves were determined by the terrain vertex locations.
- The terrain edges: which two vertices formed the ends of each edge. Information on which vertices were joined by edges in the terrain was required because the graph vertices were placed along terrain edges.
- The terrain faces: which three edges formed each triangle. Information on which edges formed each face was required because each face was allocated an individual weight, because the face

formed the unit of partitioning (see Section 2.3 for details) and because valid graph edges were cross-face if not along a terrain edge.

- A weight for each face. This was required in order to determine the weight (cost) for a graph edge.

2.2.2 Graph generation

Of Lanthier et al.'s three examined methods for Steiner point placement (see Section 1.3.1), the Interval method was chosen as, according to Lanthier, it gave a balance of good accuracy and a low number of vertices. This method involved determining the appropriate number of Steiner points based on the length of an edge, and then placing them at equally spaced intervals along the edge. Placing the Steiner points simply at equal intervals on all faces was briefly considered, but it would have meant a shorter or longer interval between the last and second last Steiner point on each edge, which would have caused local differences in path accuracy greater than those present when the Steiner points were placed evenly along an edge. Lanthier et al. [12] used the Fixed Steiner point placement scheme in their parallel implementation. Use of this scheme would have allowed arrays of fixed size to be used, but this was an implementation detail and it was judged that the variably sized arrays required by the Interval scheme would not be a major issue.

Different alternatives were considered for storing the graph generated from the terrain. The first option was to convert the terrain data into graph data directly, with explicit representation of each graph vertex and edge. This would have required a lot more memory than the original terrain data due to the large number of cross-face graph edges, but would have made for a straightforward implementation of the shortest path algorithms and easier partitioning of data.

The second option considered was to simply store the terrain data and any additional information needed by the graph algorithms. The location of the graph vertices was fixed at the Steiner points, the position of which could be easily determined from the vertex locations of the relevant terrain edge. The placement of graph edges was fixed as being along terrain edges and across each terrain face. This meant that an implicit representation of the graph was possible, in which the graph data was obtained by operations on the terrain data, when required. This clearly has a lower memory requirement, but has some overhead due to positions of Steiner points (graph vertices) having to be calculated each time they were referred to, and a list of adjacent graph vertices having to be determined from the terrain data each time they were needed. The implementation of the shortest path algorithms and partitioning was more complex with this implicit representation.

This second option was used by Lanthier et al. [12]. In addition to the terrain data, they stored the costs of Steiner points associated with a terrain edge in an array associated with the edge. It was decided to do this also, as the additional complexity was outweighed by the reduction in memory and lack of preprocessing requirements.

It was realised that operations such as finding adjacent graph vertices and setting the cost of the graph vertex required significantly different operations depending on whether the graph vertex was coincident with a terrain vertex, or in the middle of a terrain edge. Figure 2.1 shows the adjacent graph vertices to the graph vertex that is coincident with the top right terrain vertex. Adjacent vertices are along the incident terrain edges, on adjoining faces, and on the non-incident edge of (terrain vertex-)incident faces. Figure 2.2 shows the adjacent graph vertices to the graph vertex that is in the middle of the centre edge. Here, adjacent vertices are along the edge of the face (never onto a different face along an edge as the original graph vertex is in the middle of a terrain edge) and on the non-incident edges of (terrain edge-)incident faces.

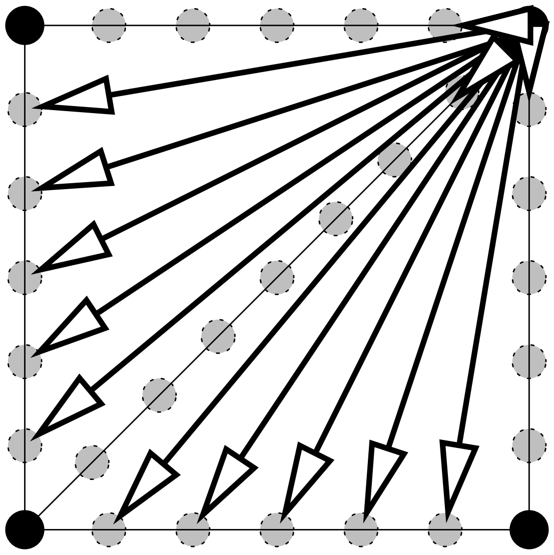

Figure 2.1: Graph vertices adjacent to a graph vertex coincident with top-right terrain vertex

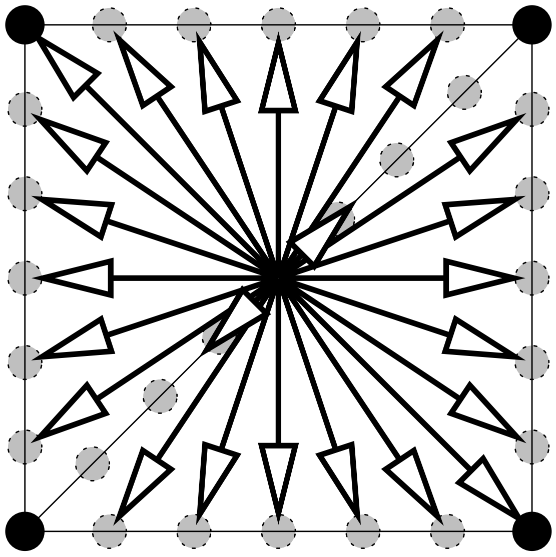

Figure 2.2: Graph vertices adjacent to a graph vertex in middle of centre terrain edge

Setting the cost of a graph vertex located in the middle of a terrain edge, just requires setting a single value in the cost array for that edge. However, a graph vertex coincident with a terrain vertex requires setting values in the cost arrays of all incident terrain edges.

Setting the values in the cost arrays of all incident edges could have been dealt with in two different ways:

1. Treat all graph vertices from different terrain edges, coincident at a terrain vertex, as being separate, joined to each other with a graph edge of cost zero. This made the code simpler as only the cost for a single Steiner point ever had to be updated, rather than setting multiple ones from the same update. This is shown in diagram form in Figure 2.3. The terrain-vertex incident graph vertex is split so that one lies on the end of each edge (dotted circles), with zero-cost graph edges between them (dotted lines). This was implemented, but since it caused a multiplying of the number of effective graph vertices, it had a serious impact on performance and was judged too slow.

2. Update the costs stored in all the edges for a graph vertex coincident with a terrain vertex. This was the solution chosen due to its faster speed.

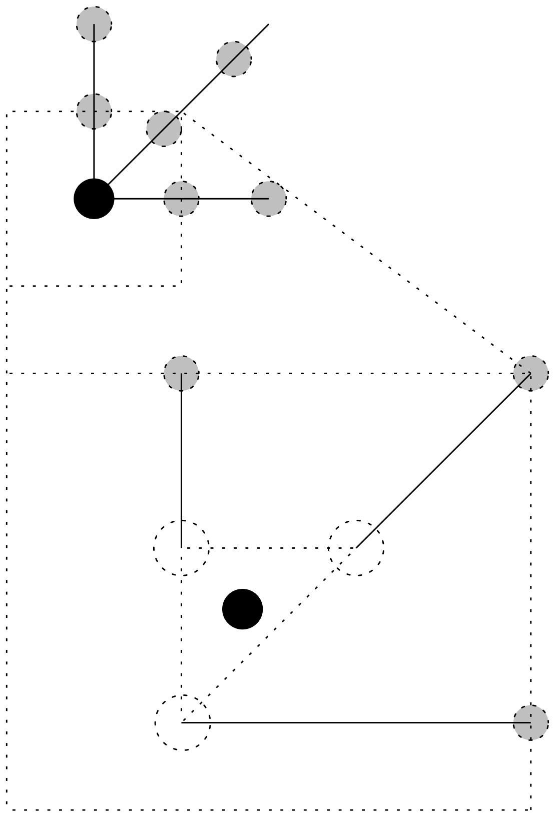

Figure 2.3: Steiner point costs being updated through the use of phantom graph vertices

Dealing with these different operations of terrain vertex and edge vertices was possible in two different ways:

1. Operations could have had conditional sections based on the index of the Steiner point being dealt with. This was implemented, but the differing operations soon became unmanageable due to the branching.

2. The two types of graph vertex could be separated into two datatypes. This was the method that was chosen, but it had its own disadvantages, as sometimes it was required to determine which

type of graph vertex was being dealt with. This implementation issue is examined in more detail in Section 3.3.2.

2.2.3 Data access requirements

In addition to needing access to the data described above, there were some constraints on the structure it was held in:

1. Given a face, be able to determine the component edges within constant time, needed so that determining the adjacent graph vertices within a face could be done quickly.
2. Given an edge, be able to determine the component vertices within constant time, needed so that the graph vertex positions could be determined quickly (for calculating costs).
3. Enable the faces sharing a particular edge to be determined in constant time, needed so that the weight of the face with minimum weight that shares an edge could be determined, for use in ascertaining the cost of a graph edge between Steiner points along a terrain edge.
4. Enable the edges sharing a particular vertex to be determined in constant time, needed so that information about graph edges along terrain edges between different faces could be found quickly.
5. Enable the faces sharing a particular vertex to be determined in constant time, needed so that information about graph edges from a terrain vertex to graph vertices on the opposite edges of all sharing faces could be found quickly.

2.3 Partitioning methods

Since the graph was represented implicitly, it was not possible to directly partition it, as the graph never existed as a separate entity. It was therefore the terrain that had to be partitioned, as outlined in Section 1.3.2. That section gives an overview of the two partitioning methods used. More details on the design used is given here.

2.3.1 Assigning faces to partitions

Although the partitioning clearly specifies where the partition lines should fall and how deep the partitioning should go, there was some choice in how faces that straddle a boundary should be partitioned.

Figure 2.4 shows a simple terrain which will be referred to in comparing splitting methods. It has labelled vertices (V prefix), edges (E prefix) and faces (circled, F prefix). Dotted lines have been overlaid which indicate the borders of the labelled partitions (P prefix).

The splitting method specified by Lanthier [12] for MFP was (where a grid cell refers to a particular partition):

> Each face ... is assigned to each processor whose corresponding grid cell it is contained in or intersects with. Using this technique, faces that intersect the boundary of two adjacent grid cells are shared between the two partitions. We call the shared faces *border faces*.

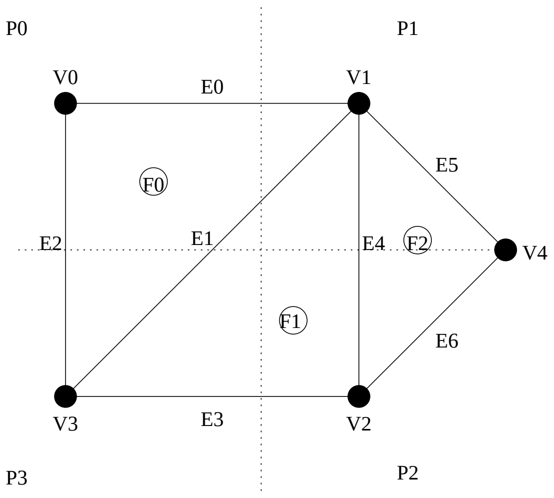

Figure 2.4: Sample terrain and partitions used for splitting example

With this method, intersection testing must be done with every face against the partitions in order to put it in the correct partition(s).

A similar, but simpler method is to test each terrain vertex to determine which partition it lies in, and place the face in each of the partitions the vertices lie in. One time this differs from the above method is when a face that is at the edge of the terrain intersects four partitions but does not have a vertex lying within one of them, as shown in Figure 2.5.

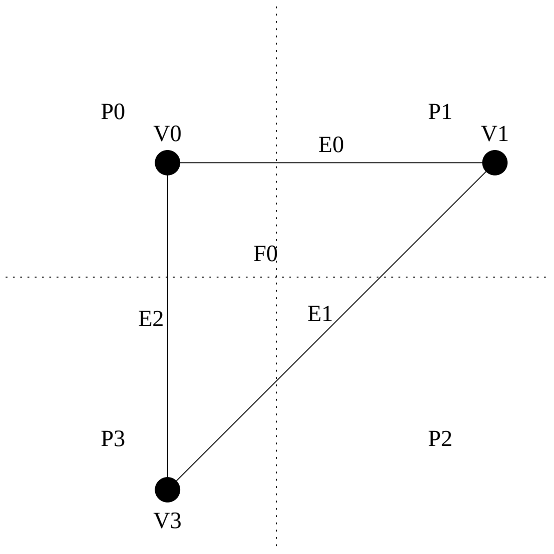

Figure 2.5: Face at edge of terrain under splitting

In this situation, the face will be added to partitions P0, P1, P3 with both methods. However, only the Lanthier method that tests intersections would place the face in partition P2. There is no reason to place the face in P2 as all partitions sharing the face still do shortest path calculations on it, and the processor dealing with P2 would just be duplicating the work of processors dealing with P0,P1,P3 unnecessarily. If there were another face that had a vertex lying in P2, then the face would then get added - and this is the time when doing shortest path calculations with Steiner points along that edge would be useful within that partition.

The only other time the method of placing the vertices differs is where a face wholly encloses a partition. In this case, the face intersection method places the enclosing face into the partition, whereas the vertex placement method does not. Again, there is no reason to place the face in the enclosed partition, as the calculations are done along the edges of the face and the enclosed partition's CPU would just be duplicating effort.

Another way of partitioning the terrain would be to actually split the triangles at the partition boundaries. With this method, the shared edges (graph vertices) would be clearly defined and less communication would be required in sharing cost information as only a maximum of one other processor would need to be updated whenever the cost changed. However, if the faces are simply split at the boundaries, they would no longer be triangles but polygons, and the data structures would have to be adapted to take this into account. More complex tessellation could be carried out near the borders to ensure the faces remain triangles, but this would not only be a possibly lengthy preprocessing step, but would increase the number of faces and hence graph edges and vertices, and probably result in slower algorithm execution.

It was decided to place the faces based on the vertex positions, as it gave a reasonable balance between speed and ease of implementation, and also was comparable to the method chosen by Lanthier et al. and so should have meant the results between the implementations should still have been comparable.

2.3.2 Partitioning execution

Performing block partitioning is a straightforward task, and could simply have been carried out by iterating through the faces and placing them in particular partitions based on their vertex position. This is because it is a single-level structure, and partitions map 1:1 with processors.

However, MFP partitioning is a recursive operation. There are multiple partitions allocated to processors, and the recursive partitioning stops at different levels for different parts of the terrain. Some structure was therefore needed to hold a particular partition, and store the identifier of the processor to which the partition was to be mapped.

Since the operation was recursive, a tree structure was the natural choice for this. Leaves of the tree held partitioned terrain data, the (row,column) of the partition at that level and a mapping from the vertex and edge indexes in that partition to vertex and edge indexes in the original unsplit data, effectively global indexes. When a partition held in a leaf had to be split further, the leaf was made into a branch with split leaves under it. The leaf's global indexes were used to create lists of shared edges and vertices in each partition. This was not an optimal solution - it was simple but would not scale well to a large terrain. A better option might have been to traverse the tree to update the shared lists during the recursive splitting, but an efficient way of doing this was not found.

It was decided to use a limit of 5 levels of recursion and a minimum cut-off of 7 vertices per partition to avoid the partitions becoming too small and communications costs outweighing the benefits of balanced data amongst the processors. This choice was fairly arbitrary, and may not have been the best one. In fact, Lanthier et al. [12] stated that a limit of 2 or 3 levels may be sufficient.

One benefit of the tree structure used was that it could be not only be used for MFP partitioning, but also block partitioning simply by setting the maximum recursion depth to 1. This meant the same, tested code could be used for both partitioning schemes, and any changes to data structures would only require code changes to be made in one place.

It was decided that the partitioning would be considered a preprocessing step and the times taken for it would not be evaluated as part of this project, or used in considering the speed of the algorithms.

2.4 Algorithms

There were some restrictions on the algorithms for the purpose of this project. Firstly, the problem of finding $\pi(s,t)$ (the shortest path, rather than just the cost of it) was not considered, but is $O(n)$ for all the algorithms considered, where n is the number of vertices in the shortest path, and is therefore asymptotically insignificant.

The standard Dijkstra's algorithm is outlined in Section 1.2.3 (sequential) and Section 1.2.4 (parallel), however some specific issues had to be considered here.

2.4.1 Issues: early termination

The problem considered was to find $d(s,t)$ for a particular source s and target t. Lanthier's algorithm was designed specifically for this, but the standard Dijkstra's algorithm finds the shortest path from the source to all other nodes. However, a small modification allows processing to halt as soon as the designated target is found, as demonstrated in the sequential algorithm (notation as in Section 1.2.3, with the addition of t being the target vertex position) below:

```
V = set of all vertices in partition
for each v in V:
  d[v] = infinity
  previous[v] = undefined
d[s] = 0
S = empty set
Q = V
while Q is not an empty set:
  u = extract-min(Q)
  S = S union {u}
  if u == t:
    exit
  for each edge (u,v) incident with u:
    if d[v] > d[u] + w[u,v]:
      d[v] = d[u] + w[u,v]
      previous[v] = u
```

On average, only half the set of Q vertices would need to be tested with this modification, so the number of iterations around the while loop would be cut in half. However, the unsorted queue *extract-min* would still require $O(v)$ iterations to find the minimum (where v is the total number of vertices), so the runtime of the algorithm would be halved (rather than quartered as may be naively expected in an $O(v^2)$

algorithm). However, this comes at the cost of increased jitter in the timed experimental results, so many runs would be required to get a good average time. It was therefore decided to:

- Verify the theoretical halving of average time by adapting the sequential Dijkstra implementation with the modification, and comparing it against the performance of the implementation without the modification.
- Perform performance testing without this modification in place, to allow more accurate results without the jitter caused by early exit from the Dijkstra while-loop.

However, when comparing the Dijkstra results against the Lanthier ones, it should be borne in mind that the appropriate Dijkstra result would be around half that stated here.

2.4.2 Issues: dealing with large graphs

Since the terrains, and hence graphs used were large, a modification was made to the standard Dijkstra's algorithm. This modification involves only adding vertices to Q when reached by the algorithm rather than starting with all vertices explicitly in Q. Unreached vertices - those that are not settled or in Q - have a distance of infinity anyway, so they should never be extracted from Q. Initially the source vertex was put onto Q with the distance set to 0.

The settled set was not stored as a separate entity, but instead a *settled* property was stored along with each vertex, which was initially *false* and was set to *true* whenever the vertex was extracted from the queue.

Another modification was made as the calculation of the appropriate weight could be quite costly (due to having to find the lowest relevant face weight) - the adjacent vertices were initially checked to see if they were already settled before doing any further action with them.

Also, as the actual path was not calculated, the previous array was not required.

The modified sequential Dijkstra algorithm demonstrates these changes:

```
V = set of all vertices in partition
for each v in V:
  d[v] = infinity
d[s] = 0
S = empty set
Q = {s}
while Q is not an empty set:
  u = extract-min(Q)
  set-settled(u)
  for each edge (u,v) incident with u:
    if not get-settled(v):
      if d[v] > d[u] + w[u,v]:
        d[v] = d[u] + w[u,v]
        insert v into Q
```

2.4.3 Issues: choice of queue

As mentioned in Section 1.2.6, there were three main choices of queue. It was intended to do initial development with a simple queueing implementation, and implement a Fibonacci heap if time allowed. However, it did not.

Referring to the modified sequential Dijkstra algorithm of Section 2.4.2, and letting v be the number of vertices and e be the number of edges, there are a maximum of v items on the queue and *extract-min* is executed v times (outer loop executed for each vertex) and insert e times (as it is never executed more than once per edge, as a vertex is set as settled each time). An unsorted queue therefore gives a sequential Dijkstra runtime of $\Theta(v^2+e)=\Theta(v^2)$. A sorted queue gives a runtime of $\Theta(v+ve)=\Theta(ve)$. A Fibonacci heap gives a runtime of $\Theta(v\log v+e)$.

In parallel, the *extract-min* operation is the operation that is being done concurrently with the simple Dijkstra parallelisation. The iterations are done in lock-step with other nodes, so unless there are less incident edges in the partitioned version of the data than there were in the unpartitioned data at all the nodes (which is unlikely given a spatial partition), the sorted type of queue will result in a runtime no better than the sequential implementation, and generally worse because of the overheads involved.

Adamson and Tick [17] compared different shortest path algorithms using a single queue that was either sorted or unsorted, shared across all processors. They determined that for randomly partitioned data with a high number of processors, a sorted queue is better but the unsorted queue does better with a lower number of processors. However, the spatial partitioning here means the unsorted queue does better.

The Lanthier algorithm was asynchronous, so there was not this problem with the parallel step, but many insertions of items onto the queue were required - more than the number of vertices, since sometimes work had to be repeated when an updated cost was received. The unsorted queue was therefore faster for this algorithm as well, though again the Fibonacci heap would be ideal.

Initially, no consideration was made to the parallel runtime issues of using a sorted queue, and since it was considered faster for the sequential version, this was implemented and tested. Testing revealed the performance problems, and it was then replaced by an unsorted queue. Results from both implementations are presented in Section 5.2.1.

2.4.4 Other issues

The only further issue with the parallel Dijkstra algorithm was how to implement the *global-min* operation on the different architectures. This is described in Sections 3.4.1 and 3.4.2.

The Lanthier parallelisation with the different systems is described in Sections 3.4.3 and 3.4.4.

Chapter 3

Implementation

3.1 Technologies

A variety of decisions were made about which technologies to use in the implementation.

3.1.1 Language

All other decisions had to follow from this, as other technologies are generally geared towards specific languages. The language chosen had to be one I had some experience with, and was also likely to lead to development and debugging times that were as fast as possible, as well as support the APIs I needed to use. It also could not be too esoteric, as the system was meant to be run on a SunFire, which would only have standard packages installed.

The options considered were:

- *C*. Discounted because, in my experience, it is hard to modularise and very easy to produce code with it that is unmaintainable and hard to debug. Experienced C programmers can, of course, produce excellent maintainable C, but I would not place myself in that category.
- *C#*. Discounted due to its total lack of message passing parallel programming API implementation, and my lack of experience with the language.
- *Java*. Discounted due to its lack of a mature and well-supported message passing parallel programming API implementation.
- *C++*. This was chosen, due to my wider experience with it and an object-oriented paradigm, and availability of a relatively stable MPI implementation for message passing.

3.1.2 Shared memory API/system

The shared memory API chosen had to allow for a great deal of control over the execution of the simultaneous tasks. It also had to be freely available.

The options considered were:

- *OpenMP*. This was discounted due to it merely providing hints to the compiler, and not allowing explicit control over the actions of tasks.
- *POSIX Threads (PThreads)*. This was chosen due it being widely available and fairly straightforward, and also due to it having been introduced in a previous course I had taken.

3.1.3 Message passing API/system

The message passing API had to be available on all the systems the implementations were to be executed upon, and be as straightforward as possible to use, preferably supporting collective operations.

MPI matched this, and I was somewhat familiar with the basics from a university course, so this was chosen without seriously considering alternatives. The LAM/MPI implementation was used as it was free, readily available, had C++ bindings, and I already had some experience with it. A local version of LAM/MPI had to be built on the target machines due to the installations present not supporting the C++ bindings.

3.1.4 Platform/Compiler

This was limited by the target systems - a Beowulf cluster, running Linux, and a SunFire, running Solaris.

As my personal system and most university systems ran Linux, this was the obvious development platform. The UNIX-like nature also meant that running programs ought to at least have a reasonable chance of being able to be ported straightforwardly to the SunFire.

The standard compiler on Linux systems is the GNU C++ compiler. *g++ 3.3* was used.

3.1.5 Build system

In order to efficiently compile the source code, a build system was needed. The following options were considered:

- *GNU Make*. This is the defacto standard build system on Linux, but its lack of automatic dependency recognition cause Makefiles to be more complex than they need be. Also, though GNU Make is often available on Solaris systems (like the SunFire) as gmake, I could not be sure it would be present, and though Sun's Make is generally compatible, it obviously does not support the GNU Make extensions which are all too easy to use. GNU Make was therefore ruled out.
- *SCons* [18]. This is a python-based build system which has automatic dependency recognition and is very straightforward to use and customise. I had used it on previous projects with success, and, confident that python could be compiled and installed locally on the SunFire if needed, I used this for the project.

3.1.6 Others

Subversion [19] was used for revision control, due to it being free, supporting atomic commits, but also operating under the simple paradigm of checkouts and updates that CVS uses.

Python was used for scripting the creation and processing of data, due to it being easy to use for such tasks.

GNUplot was used for the creation of graphs as it linked well with a scripted approach to graph generation from large numbers of data files.

L^AT_EX was used for the preparation of this dissertation, due its powerful features and the availability of university templates.

3.2 Data sourcing

After the technologies to use were determined, data had to be obtained to be used by the partitioning algorithms (and the sequential Dijkstra implementation), initially in unweighted TIN format (see Section 2.2.1).

Terrain information for the UK was freely available from the Landmap project [20]. An Ordnance Survey 100km x 100km grid square (NN) was downloaded in Digital Elevation Model (DEM) format, which consists of an array of values giving a heightfield for the terrain. This was converted to unweighted TINs of varying resolutions using free third-party software - LandSerf [21]. This method provided all the 'nn-' TINs. The 'test2-' TIN was created by hand for debugging purposes, and consists of two triangles forming a square on the X-Y plane, but with one vertex at a higher Z value in order to give different face weights. This TIN was primarily used for debugging purposes. The 'ls_samp-' TIN was a sample data set provided with LandSerf and was used as the main testing data during the implementation stages, since its number of faces were low enough to give quick results even on a uniprocessor system, but there was enough data to provide a good test of correctness. Diagrammatic representations of the datasets are presented in Appendix A.

Determining the effect of terrain with varying vertex density in different regions was not one of the goals of this project, so the fact that most of the datasets had similar vertex distribution was not an issue.

The weights were created from the unweighted data by using a metric based on the slope of the face. They were set to be the dot product of the unit face normal, with the z-axis, giving a value of 1 for a vertical face and 0 for a flat face, meaning flat faces were lower cost. In fact this meant that completely flat faces had zero cost, and this should have been modified to give even flat faces a non-zero cost for a more realistic weighting. However, this did not affect the results here as none of the data sets had completely flat faces.

3.3 Data structures

3.3.1 Terrain representation

The overall terrain data structure for a partition (*Geometry*) stored a flat list of faces (*Face aList[]*), edges (*Edge eList[]*) and vertices (*Vector vList[]*) for the terrain within that partition.

Face stored indexes into *eList* for each of its three edges plus a weight. *Edge* stored indexes into *vList* for its two vertices. This indexed structure, shown in Figure 3.1, allowed the first two runtime requirements to be met (see Section 2.2.3).

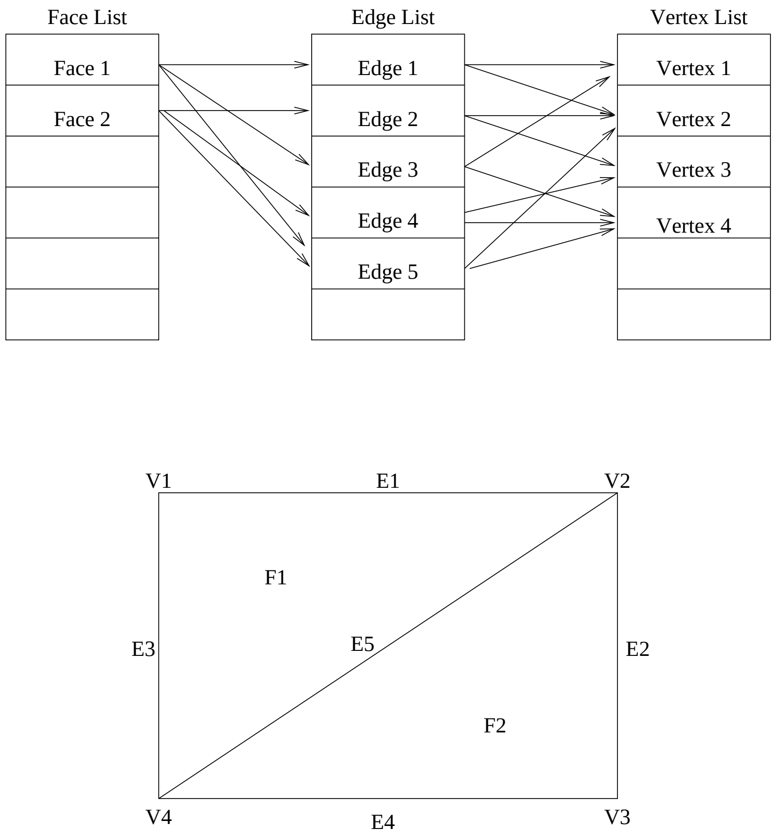

Figure 3.1: Face indexing example

Indexes were used rather than pointers as:

1. With pointers, indices would still have to be stored in the data file (unless some kind of intelligent serialisation method were used), so by using indices during execution, the same representation could be used on-disk as in memory, which was helpful in debugging.

2. Indexes could be potentially passed to other processes via message passing if the method of storing the partitioned data was later changed.

Indexes were local to each processor, rather than global identifiers or indexes local to a particular partition (note that there is only a distinction between a processor's data and a partition's data with MFP partitioning, and not with block partitioning).

Global identifiers were not used because they would have been:

1. Less memory efficient, requiring enough bits to refer to every vertex, edge or face in the entire unsplit terrain, which does not scale well as the terrain gets very large. However, this is only a theoretical issue, as unsigned 32-bit integers were used for the indexes, which would have been more than sufficient for the terrains under test here.

2. Less time efficient, requiring the number of items in all lower-ranked cells to be subtracted before using the global index with the local lists.

3. Messier to program with, due to this subtraction.

Also at the time the type of identifier was being decided, the splitting method had not yet been fixed. Had the splitting caused actual faces to be split, partial edges could have been shared and there would not have been a simple way to refer to these globally while preserving the edge sharing characteristics.

Identifiers local to a partition (with multiple *Geometry* instances per partition) were initially attempted, but ruled out because of:

1. The difficulty of adjusting the references to shared data items during recursive splitting of partitions.

2. The need to send some kind of *Geometry* index to define which *Geometry* is referred to by edge and vertex indices during communication.

Identifiers local to a processor were used because they did not have the issues above. However, using local (rather than global) identifiers gives rise to a need for each partition to store information on which other partitions share the data, and need to be updated when that data changes. For the splitting method chosen, nodes had to notify other nodes about changes to costs of Steiner points on shared edges.

In order to do this, two lists were set up in *Geometry*. *vSharedList* was a list of (cell x, cell y, vertex index in cell), with indexes into *vSharedList* matching the indexes into *vList*.

A custom tool (util_tinconvert) was created which read in the triangles from the TIN data file, added them to the *Geometry* structure one by one (which included setting the weights) and then saved the built *Geometry* structure, using the indexed representation for later reading directly by the algorithm implementations, as described in Section 2.2.1.

In order to split the terrain, a tool was created which read in a *Geometry* and split it according to either block or MFP partitioning, using the method for placing faces in partitions described in Section 2.3.

To meet the last three runtime requirements (see Section 2.2.3, look-up tables were generated at runtime (upon loading of the *Geometry*) to give the reverse mappings from an edge to incident faces, a vertex to incident edges, and a vertex to incident faces. These tables were needed to get the constant time lookup - without them, a linear search would be required, which would be too slow since these operations were required frequently.

It would have been possible to store the reverse mappings along with the forward mappings in the data file. However, this would have been duplication of data and would have just moved the runtime impact to partition generation rather than partition load time. The time was not that long for the testing here, though Table 3.1 shows the runtimes of each operation. For large terrains with multiple runs on the same terrain, it may be worth storing this reverse mapping.

Look-up table	Runtime
Faces incident with Edge	$\Theta(ef)$
Edges incident with Vertex	$\Theta(ve)$
Faces incident with Vertex	$\Theta(vef)$

f: Number of faces, e: Number of edges, v: Number of vertices

Table 3.1: Run time of reverse map creation

An example of the splitting implementation used will be demonstrated. A sample terrain is shown in Figure 3.2. A diagrammatic representation of the face, edge and vertex data structures and how they are linked, without any partitioning, is shown in Figure 3.3. The actual representation of the geometry is shown in Figure 3.4. The reverse mappings have not been shown.

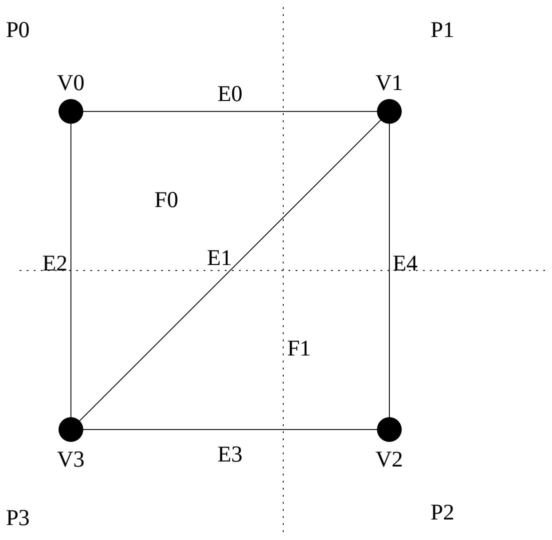

Figure 3.2: Sample terrain and partitions used for splitting example

Performing a split based on vertex locations gives the resulting partitions being stored as shown in Figure 3.5, where the faces, edges and vertices are represented by global tags for clarity.

This is stored in the geometry structure by making use of indexes, with *vSharedList* and *eSharedList* containing appropriate values to store the information on which other partitions share the edges and vertices. The actual geometry structure for each partition is shown in Figures 3.6 and 3.7.

3.3.2 Graph representation

In order to be able to easily work with the implicit representation of the graph, a *GVertex* class was created. As described in Section 2.2.2, operations such as finding adjacent vertices and setting the cost of the vertex required significantly different operation depending on whether the graph vertex was co-incident with a terrain vertex, or along a terrain edge. *GVertex* was therefore subclassed into *GVertexE*

$$\begin{tikzcd}
F0 \arrow[r] \arrow[dr] \arrow[ddr] & E0 \arrow[r] \arrow[dr] & V0 \\
F1 \arrow[r] \arrow[ddr] \arrow[dddr] & E1 \arrow[r] \arrow[ddr] & V1 \\
 & E2 \arrow[uur] \arrow[dr] & V2 \\
 & E3 \arrow[ur] \arrow[r] & V3 \\
 & E4 \arrow[uuur] \arrow[uur] &
\end{tikzcd}$$

Figure 3.3: Sample terrain as stored before partitioning

Index	*aList*	*eList*	*eSharedList*	*vList*	*vSharedList*
0	ieList: 0,1,2; w: w0	ivList: 0,1		pos: (0,1,0)	
1	ieList: 1,3,4; w: w1	ivList: 1,3		pos: (1,1,0)	
2		ivList: 0,3		pos: (1,0,0)	
3		ivList: 2,3		pos: (0,0,0)	
4		ivList: 1,2			

aList: face list, *eList*: edge list, *eSharedList*: shared edge list, *vList*: vertex list, *vSharedList*: shared vertex list, *ieList*: edge index list, *w*: face weight *ivList*: vertex index list, *pos*: vertex position

Figure 3.4: Unpartitioned terrain stored in *Geometry* structure

for terrain-edge graph vertices and *GVertexV* for terrain-vertex graph vertices, with different implementations of the operations in which the different classes of graph vertices differ. *GVertexE* stored a terrain edge index and Steiner point index along the edge, and *GVertexV* stored a terrain vertex index. It should be noted that they did not store information about cost or whether the vertex was settled (for Dijkstra's algorithm), just set those values in the arrays associated with a terrain edge. *GVertex* instances could therefore be deleted and created without any modification of the underlying data, which meant they could straightforwardly be added and removed from the queue.

The ability to look up edges sharing a particular vertex in constant time meant that the cost setting operation of a *GVertexV* simply set the cost on the relevant Steiner point of all incident edges, in $O(n)$, where n was the number of edges incident with the vertex. The other reverse mappings allowed similar advantages for finding adjacent vertices and setting the cost of a *GVertexE*.

GVertex had the following operations:

- *getPosition*. Return the vector position of the Steiner point on the terrain, determined by interpolation between the terrain vertices associated with the edge for a *GVertexE*, and just determined by the associated terrain vertex position for *GVertexV*.
- *getCost*. Return the currently stored best cost for the Steiner point, retrieved from an array associated with the edge. For *GVertexV*, just return the cost associated with the relevant item in the

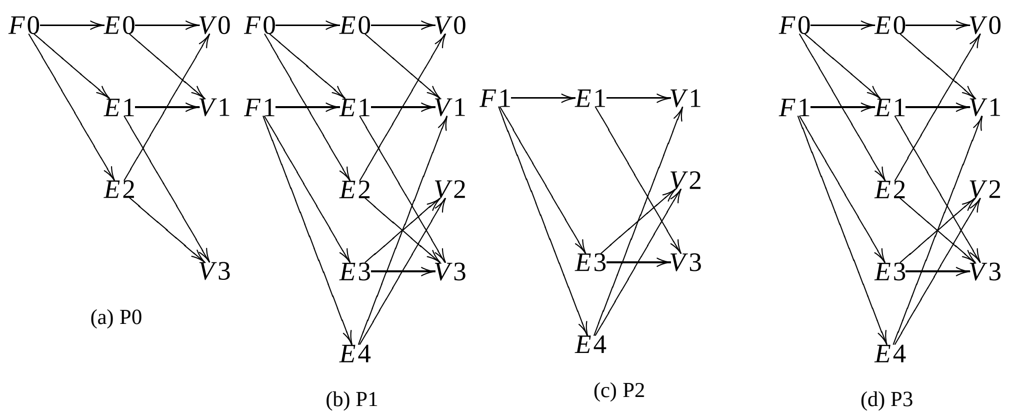

Figure 3.5: Partitioned geometry structures after splitting, with global tags

first incident edge.

- *setCost*. Set the currently stored best cost for the Steiner point, in an array associated with the edge. For *GVertexV*, set the item in the array of all incident edges.
- *getSettled*. Return the flag indicating whether the vertex is settled (used in Dijkstra), retrieved from an array associated with the edge. For *GVertexV*, just return the item from the array in the first incident edge.
- *setSettled*. Set the flag indicating whether the vertex is settled, in an array associated with the edge. For *GVertexV*, set the item in the array of all incident edges.
- *getFaceWeightTo*. Accepts another *GVertex* instance as a parameter, and returns the weight of the lowest-weighted face that could be traversed over a graph edge between the two.
- *getAdjacent*. Return a list of *GVertex* instances representing the graph vertices joined by a graph edge to this one.

There were some problems with how *GVertex* was implemented. The algorithm implementations had to consult a different shared list (*vSharedList* or *eSharedList*) depending on whether the *GVertex* was a *GVertexV* or a *GVertexE* to send an updated cost to the relevant processor. This meant that run-time type information was needed in order to consult the correct list, which was implemented straightforwardly with a method in each class that returned the type as an integer (actually an enumerated type). Branches based on this integer then used the correct list. This could have been implemented better by adding a method to *GVertex* to return a list of shared *GVertex* instances, with an associated partition ID for each, ready for transmission to the relevant nodes.

The run-time type information was also used in marshalling data for sending and receiving in the message-passing systems. This use could have been avoided by making *GVertex* able to serialise itself and using a factory method to create the correct *GVertex* subclass from received data.

If time had allowed, these changes would have been made in order to avoid the need for run-time type information and to make the algorithm code simpler and clearer.

Index	*aList*	*eList*	*eSharedList*	*vList*	*vSharedList*
0	ieList: 0,1,2; w: w0	ivList: 0,1		pos: (0,1,0)	
1		ivList: 1,2		pos: (1,1,0)	
2		ivList: 0,2		pos: (0,0,0)	
3					
4					

aList: face list, *eList*: edge list, *eSharedList*: shared edge list, *vList*: vertex list, *vSharedList*: shared vertex list, *ieList*: edge index list, *w*: face weight *ivList*: vertex index list, *pos*: vertex position

(a) P0

Index	*aList*	*eList*	*eSharedList*	*vList*	*vSharedList*
0	ieList: 0,1,2; w: w0	ivList: 0,1		pos: (0,1,0)	
1	ieList: 1,3,4; w: w1	ivList: 1,3		pos: (1,1,0)	
2		ivList: 0,3		pos: (1,0,0)	
3		ivList: 2,3		pos: (0,0,0)	
4		ivList: 1,2			

aList: face list, *eList*: edge list, *eSharedList*: shared edge list, *vList*: vertex list, *vSharedList*: shared vertex list, *ieList*: edge index list, *w*: face weight *ivList*: vertex index list, *pos*: vertex position

(b) P1

Figure 3.6: Partitioned terrain stored in *Geometry* structures, one per partition, P0-P1

3.3.3 Queues

As stated in Section 2.4.3, unsorted and sorted queues were implemented. A *Queue* template was used for the queue construction in order to allow it to be reused with different types. The only requirement was that the type supported the ¡ and == operators, the former to give a priority ordering and the latter so that items could be identified for modification and deletion. As the queue was composed of *GVertexE* and *GVertexV* instances cast as *GVertex* instances, it had to contain pointers to the instances as *GVertex* was a pure virtual base class. A specialised version of *Queue* was therefore created for handling pointer types, as the instances held had to be dereferenced in order to use the correct operators (rather than just ¡ and == on the pointers themselves).

The use of a *Queue* template throughout allowed the type of queue to be easily changed, just by changing the definition in the include file.

The sorted queue was implemented by storing the items internally in a *deque* (a doubly linked list). This was kept sorted, as whenever an item was added or had its cost updated, a linear search was performed through the list to find the correct location. Removing the top item from the queue was fast, as it just meant taking the first element of the list.

Index	*aList*	*eList*	*eSharedList*	*vList*	*vSharedList*
0	ieList: 0,1,2; w: w1	ivList: 0,2		pos: (1,1,0)	
1		ivList: 1,2		pos: (1,0,0)	
2		ivList: 0,1		pos: (0,0,0)	
3					
4					

aList: face list, *eList*: edge list, *eSharedList*: shared edge list, *vList*: vertex list, *vSharedList*: shared vertex list, *ieList*: edge index list, *w*: face weight *ivList*: vertex index list, *pos*: vertex position

(c) P2

Index	*aList*	*eList*	*eSharedList*	*vList*	*vSharedList*
0	ieList: 0,1,2; w: w0	ivList: 0,1		pos: (0,1,0)	
1	ieList: 1,3,4; w: w1	ivList: 1,3		pos: (1,1,0)	
2		ivList: 0,3		pos: (1,0,0)	
3		ivList: 2,3		pos: (0,0,0)	
4		ivList: 1,2			

aList: face list, *eList*: edge list, *eSharedList*: shared edge list, *vList*: vertex list, *vSharedList*: shared vertex list, *ieList*: edge index list, *w*: face weight *ivList*: vertex index list, *pos*: vertex position

(d) P3

Figure 3.7: Partitioned terrain stored in *Geometry* structures, one per partition, P2-P3

The unsorted queue also stored the items internally as a *deque*. However, the items were not kept sorted. Whenever an item was added, it was just added to the end of the list. Updating the cost of an item required a linear search, as did removing the top item.

Each of the queue implementations was kept in a separate header file, so the desired one could be easily swapped in.

3.4 Algorithms

Details on how the algorithms were implemented are given in the section below. The complete source code for the implementations described can be found in Appendix B. Only the source for the actual top-level algorithms (rather than any of the core library code) is provided there, but it is commented and should be understandable on its own. Pseudo-code describing the implementations in simplified form has been provided here.

3.4.1 Message passing Dijkstra

The shared memory Dijkstra implementation followed the general outline of Section 1.2.4. The key issue was how to implement the *global-min* operation. This operation takes as its input the *GVertex* from the local queue and returns a *GVertex* representing the global vertex with minimum cost, if that vertex exists in the partition. The obvious way to do this was to perform some kind of reduction operation.

If global identifiers for edges had been used, a user-defined datatype could have been constructed for use with MPI, storing edge and Steiner point identifiers and an associated cost. A custom MPI operator for use in the reduction could have been created, which would have found the minimum based on the cost, and through the use of this, a single MPI *AllReduce* operation would have been sufficient to get the relevant *GVertex*.

However, since local identifiers were used, the task was more difficult. Information about another partition's winning edge index is of no use to the other partitions, so the above technique could not be used.

One alternative considered was to do an *AllReduce*, with user-defined datatype and custom reduction operator as above, but have the user-defined datatype store the *GVertex* position rather than edge and Steiner point index. However, without a map from position back to local edge and Steiner point, each node would have to do a slow search through the terrain data to find the corresponding local *GVertex*. This was therefore ruled out.

The method chosen was to carry out the reduction just on the cost values and use the *MIN_LOC* reduction operator in order to get the rank of the winning node. The winner then knows the cell (rank), edge index, Steiner point index or cell, vertex index for the winning *GVertex*, while the others just know the cell (rank). At this point a method had to be found to inform the cells that share the winning edge of the correct edge index and Steiner point or vertex index in their partition.

One way of doing this would be for the winner to broadcast the edge index/SP or vertex index for the winning vertex in the winner's cell to nodes that share the vertex, which would require each node to search through its shared lists to find a matching entry, and hence determine the local *GVertex*. This was ruled out due to the searching (though alternative data structures could have eliminated this).

Another way considered was for the winner to use its shared lists to find all the (cell id, edge index) or (cellid, vertex index) that share the winning *GVertex*, and then to use *ScatterV* to send to each cell that shared the data. However, with *ScatterV*, both sender and receiver have to know the number of items being received. Therefore in order to do this, the winning node would first have to send with *Scatter* a 0 or 1 to each node, to indicate the number of items they should receive in the *ScatterV*. This method was therefore not used.

The way this was actually implemented was as follows. The winner uses its shared lists to find all the (cell id, edge index) or (cell id, vertex index) that share the winning *GVertex*. All nodes then perform a *Scatter*, with the winner being the sender, sending either (vertex index, dummy value) or (edge index, SP index) or (dummy value, dummy value) to every node, depending on whether it is a vertex shared, edge shared or if there is nothing shared. The scatter receivers do nothing if they get (dummy value, dummy value). If they receive a valid value, they create a *GVertex* from it, and remove that same *GVertex* from the top of the queue if it is present, which it will be only if there was a tie in the reduction and the receiver did not win.

Having separate *vSharedList* and *eSharedList* was required for this to work - had there just been an *eSharedList* (and *GVertexE*), then finding the cells that share the Steiner point when dealing with the first or last Steiner point of an edge (i.e. coincident with a terrain vertex and shared with other edges) would have been impossible.

There were additional complications in the implementation. All nodes must take part in any collective communication, therefore whenever a node's queue was empty, it had to broadcast a dummy lowest cost of infinity (a very large number) so that it never won the reduction. When the minimum global value was infinity, the algorithm was complete.

The algorithm implementation, simplified to hide the fact that it was not a true *GVertex* sent (hence the comment about *u.getCost()* in the pseudocode, is shown below. *all-reduce* is the corresponding MPI operation, as is *scatter*.

```
V = set of all vertices in partition
Q = empty queue
for each v in V:
  v.setCost(infinity)
if s is in V:
  s.setSettled(true)
  s.setCost(0)
  Q.push(s)
do until quit:
  if not Q.empty():
    q = Q.top()
    winningcost, winningrank = all-reduce( q.getCost() )
  else:
    winningcost, winningrank = all-reduce( infinity )
  if winningcost = infinity:
    quit
  if rank = winningrank:
    vertexincellarray = get-cells-and-indexes-for-sharers( q )
  u = scatter( vertexincellarray, winningrank )
  if q.position = u.position:
    q.setSettled( true )
    q.setCost( u.getCost() ) // u.getCost() is actually cost received separately
    Q.pop()
  for each GVertex v adjacent to u:
    if not v.getSettled():
      if v.getCost() > (u.getCost() + (cost between u and v)):
        v.setCost( u.getCost() + (cost between u and v) )
        Q.push( v )
```

Another possible way of implementing the termination was initially done - instead of simply terminating when the reduction value was infinity, a mechanism was used whereby a flag was set when the partition had been reached - i.e. when an item had been added to its queue. Once its queue was empty, that node finished the main loop, but it still had to take part in the reduction (since it was a collective communication) and send out infinite values, which was handled by a separate loop that terminated once the final reduction value was infinite. This was done as it made the timing of how long nodes were sitting waiting to terminate slightly easier, but it was basically the same thing, just implemented slightly differently. However, it was changed to the method represented by the pseudocode to make the implementation length more comparable with that of the shared memory Dijkstra (Section 3.4.2).

3.4.2 Shared memory Dijkstra

The shared memory Dijkstra implementation followed the general outline of Section 1.2.4. The data was already partitioned in separate files for the message passing implementations, so this data was reused here. It would have been possible to instead load the unpartitioned data and share it between the processors, with each processor being allocated a portion, but then the partitioning would have had to have been done at runtime rather than as a preprocessing step (or a different method of storing the data would have had to have been used), so this was not done.

As a shared implementation could practically use either a single queue, shared between all processors, or multiple queues, one per processor as with the message passing implementations, a decision had to be made between these options. As stated in Section 1.2.6, Bertsekas et al. [14] found that the multiple queue strategy was better with randomly assigned data in each queue, but here the multiple queues would have been used with spatially assigned data (as the partitions are spatially assigned) so it was not certain this advantage would remain.

A single queue was the first option considered, and was implemented when the sorted queue was used. However, when the queue was changed to be unsorted, it was decided to also change to multiple queues. There were three reasons for this:

1. To give a more direct comparison between the message passing and shared memory implementations, as with both using multiple queues, the only real difference was in whether they used shared memory or message passing to do the communication.
2. It appeared likely that with a single queue, there would be a greater impact from cache coherency protocols than with a queue per thread.
3. The single queue implementation was reliant on the behaviour of the sorted queue and would have required significant changes to support the unsorted queue and operate efficiently.

Unfortunately this meant that the unsorted vs. sorted queue implementations of this algorithm were not directly comparable. In retrospect, either only one aspect of the queueing implementation should have been changed at a time, in order to examine the results with both multiple and single queues, and sorted and unsorted queues, or tests should have been done with multiple sorted queues to complete the results.

The single queue version was implemented by creating a *PTQueue* class, an instance of which was shared amongst all the threads. The algorithm body looked very like that of the sequential Dijkstra implementation as all the sharing and threading was taken care of by *PTQueue*.

PTQueue was not a generalised queue with multiple readers and writers, but was designed to obtain the maximum speed by considering the type and order of calls made by the algorithm body. The main body was as follows. *q* is the shared *PTQueue* instance, *u* and *v* are *GVertex* instances, and their operations are as described in Section 3.3.2. *g* is the local *Geometry*. *—a—* finds the magnitude of a vector. *pushes* is used to temporarily store the items that are to be added to the queue, so that writing of cost information does not occur at the same time as reading.

```
q.waitForPush()
if s is in partition:
  q.push( s, 0 )
while not q.empty():
  u = q.pop( g )
  if u is valid:
    set u as settled
    pushes = empty set
    vs = u.getAdjacent()
    for each v in vs:
      if not v.getSettled():
        if v.getCost() > u.getCost() + (cost between u and v):
          pushes.add( v, u.getCost() + (cost between u and v) )
  q.waitForPush()
  for each item in pushes:
    q.push( item.vertex, item.cost )
```

PTQueue held a *Queue* instance in order to store the actual *GVertex* instances. One option considered was storing an array of *GVertex* instances for each entry in the queue, one array item per cell. This would have required more storage than necessary (a *GVertex* for every cell, for each item in the queue, even though most would only be in one cell) so was ruled out.

Instead, the 'base cell' of a *GVertex* was found, and this was added to the queue. The base cell was defined to be the cell with lowest rank of the cells that share the *GVertex* in question.

The *PTQueue* operations were as follows:

- *push*. This was passed a *GVertex* and a new cost. It found the *GVertex* in all cells that shared it and set the cost in each. It then added the base cell *GVertex* to the queue, using a mutex on the queue to ensure only one simultaneous write.
- *waitForPush*. This was a barrier wait, where all threads would have to pause until the others reached the barrier (the code for the barrier was adapted from Butenhof's book [22]). This was used before any writing was carried out to the queue, as writing to the queue also wrote to each thread's geometry instance.
- *pop*. This retrieved the top item from the queue, barrier waited until all threads had called pop and retrieved the top item, then removed it from the queue. The top item was then returned as the *GVertex* in the caller's cell (rather than in the base cell).

The multiple queue version was implemented similarly to the message-passing Dijkstra, just with a different implementation of *global-min*. The algorithm is shown below. *gv* is the global winning *GVertex* instance. *Q* is the local *Queue*. *mutreduce* is a mutex. *barrier-wait* is a barrier operation, as described above; the number in parenthesis is just a tag that is referred to below.

```
V = set of all vertices in partition
Q = empty queue
for each v in V:
  v.setCost(infinity)
if s is in V:
  s.setSettled(true)
  s.setCost(0)
  Q.push(s)
gv = null vertex
mutreduce = mutex
do until quit:
  if not Q.empty():
    q = Q.top()
    barrier-wait(1), single thread delete gv
    barrier-wait(2)
    if gv null or gv.getCost() > q.getCost():
      mutex-lock(mutreduce)
      if gv is null or gv.getCost() > q.getCost():
        gv = q
      mutex-unlock(mutreduce)
  else:
    barrier-wait(1), single thread delete gv
    barrier-wait(2)
  barrier-wait(3)
  if gv is null:
    quit
  if gv.getPosition() = q.getPosition():
    Q.pop()
  u = vertex-in-this-geometry( gv )
  u.setSettled( true )
  for each GVertex v adjacent to u:
    if not v.getSettled():
      if v.getCost() > (u.getCost() + (cost between u and v)):
        v.setCost( u.getCost() + (cost between u and v) )
        Q.push( v )
```

barrier-wait(1) was required to ensure no thread still required to read the old *gv*. *barrier-wait(2)* was required to ensure that no thread attempted to set a new *gv* before the old one was deleted. *barrier-wait(3)* was required to ensure that all threads had made an attempt to set *gv* before threads started reading the value. The mutex was required to ensure that *gv* did not change in between being checked and being set. The double testing of *gv* was carried out so that it could be trivially checked before being locked (avoiding some costly locks).

The key point to note is that vertices were added to the queues within the partition as a *GVertex* in that partition, and the winning *GVertex* was accessible to all cells, still referring to data in its partition. This was possible because other threads could read the data from every partition, and could therefore determine the cost directly for themselves. The *vertex-in-this-geometry* operation accessed the *vSharedList* or *eSharedList* (depending on the *GVertex* type) in the *Geometry* of the winning *GVertex* and found the corresponding *GVertex* in that cell's *Geometry*. This was an $O(s)$ operation, where s was the number of items in the relevant shared list entry (so s would be less than the number of cells).

3.4.3 Message passing Lanthier

The Lanthier algorithm was specified in Section 1.2.5 in a way that attempts to be neutral of message passing or shared memory implementation, however it was given in Lanthier et al.'s paper [12] as a specifically message-passing algorithm. That form of the algorithm was implemented here, and is reproduced below for reference (with a couple of corrections and expansions). *nx* is the number of columns in the processor configuration, *ny* is the number of rows. *rank* is the MPI rank of the process.

```
V = set of all vertices in partition
for each v in V:
  v.setCost( infinity )
if s is in V:
  s.setCost( 0 )
  Q.push( s )
do until quit:
  if have incoming message:
    cost token (newmaxcost, originatorrank):
      set maxcost to newmaxcost
      if rank != originatorrank:
        send cost token (newmaxcost, originatorrank) to (rank+1)%size
    done token (timesround, originatorrank):
      if rank = originatorrank and timesround = 2:
        send terminate to (rank+1)%size
        quit
      else:
        done = (timesround, originatorrank)
    updated cost msg (GVertex v,cost):
      if cost < v.getCost():
        v.setCost( cost )
        update / reorder v on Q
    terminate msg:
      send terminate to (rank+1)%size
  if Q.empty() || localcost > maxcost:
    if done is not NULL:
      if done.originatorrank = rank:
        done.timesround = done.timesround + 1
      send done token (timesround,originatorrank) to (rank+1)%size
    wait for incoming message
  else:
    v = Q.pop()
    localcost = v.getCost()
    if localcost < maxcost:
      if v = t:
        maxcost = v.getCost()
        send cost token (v.getCost(), rank) to (rank+1)%size
    else:
      for each GVertex u adjacent to v:
        if u.getCost() > (v.getCost() + (cost between u and v)):
        u.setCost( u.getCost() + (cost between u and v) )
        for each GVertex x in u's shared list:
          send updated cost msg (x, u.getCost() ) to (cellx + celly * nx)
```

As can be seen, messages that were effectively to be broadcast were passed round-robin around the nodes, avoiding a collective communication and hence synchronisation. Termination detection was carried out by passing a *done token* around twice. More details can be found in Lanthier et al.'s paper [12].

The main implementation issue was that buffered sends had to be used throughout, with a larger than standard buffer allocated. This was because without buffering, deadlock was possible - the implementation was unsafe. An example of this would be where two processes each try to send an updated cost message to each other at the same time - both processes deadlock when they attempt to send. The solution of adding buffering was not ideal, but it was the only way found to solve the problem and maintain the code outline as specified by Lanthier et al. [12]. If more time were available, it would have been a good idea to attempt to find a safe solution to the problem.

3.4.4 Shared memory Lanthier

Insufficient time remained for a shared memory Lanthier implementation. However, some initial design work was carried out, which is described here.

With shared memory, *maxcost* could be a global variable. The queue could be shared, or there could be one per thread, but the multiple queue implementation was considered for the same reasons as for the shared memory Dijkstra implementation.

The essence of the algorithm is given below:

```
g_maxcost = infinity
g_finish = false
localcost = 0
V = set of all vertices in partition
for each v in V:
  v.setCost( infinity )
if s is in V:
  s.setCost( 0 )
  Q.push( s )
do until quit:
  if all Q's are empty:
    g_finish = true
    quit
  if Q is empty or localcost > g_maxcost:
    wait for Q to update or g_finish to change
  else:
    u = Q.pop()
    localcost = u.getCost()
    if localcost < g_maxcost:
      if u = t:
        g_maxcost = localcost
      else:
        for each GVertex u adjacent to v:
          if u.getCost() > (v.getCost() + (cost between u and v)):
          u.setCost( u.getCost() + (cost between u and v) )
          for each GVertex x in u's shared list:
```

```
        x.setCost( u.getCost() )
        add x to relevant partition's Q
```

The above requires every partition's queue to be accessible from all other partitions, and some locking would have to be done on the access to each. Locking would probably also be required on *g_finish*. The waiting for *g_finish* or *Q* to update could likely be done using condition variables.

3.4.5 Source and target specification

A shortest path algorithm requires some mechanism for specifying the source and, where relevant, target positions of the path. Typically, these would be specified as positions anywhere within the terrain area. However, here the shortest path was being found on a graph (converted from the terrain), and what was needed at the start of each algorithm was positions specified by a *GVertex*.

It would have been possible to implement a search through the faces of the terrain in order to find the face enclosing an arbitrary position specified. However, since the terrain information was stored in such a way that it was not fast to find a face from a position, this would have involved searching through the faces. After the enclosing face was found, some way of finding the *GVertex* which provided the shortest path would have to be carried out.

Since the shortest path algorithm was being investigated, rather than this different problem, it was decided to require the source and target positions to be coincident with terrain vertex positions. A tool, *util_pickvertex*, was created to facilitate this by picking a random vertex from a specified data file, and returning its position for use as the source or target to be passed to an algorithm implementation.

3.4.6 Event logging

A good logging system is a project in itself, however the requirements of the logging here were:

1. Allow messages to be output from the program to standard output - file output was not required. This meant that simply using *cout* was sufficient.
2. Facilitate outputting arbitrary information. This was implemented by making the *Log* class inherit *streambuf* and *ostream* so that it inherited the $<<$ operators for standard types and much of the outputting functionality.
3. Be thread-safe - multiple threads may wish to write a log message at the same time, and the final output had to be intelligible. This was implemented by putting mutex locking and unlocking within the *Log* constructor and destructor, so that only one *Log* instance could be created at a time. Unfortunately this explicit locking was required as *cout*, unlike *printf*, is not guaranteed to be thread-safe.
4. Allow different levels of log message - at least debugging messages that should not normally be output, and standard messages that should always be output.
5. Have minimal runtime impact when debug logging turned off.

6. Allow logging to be turned on and off per module.

The last three requirements were met by the use of macros: *LOGD* for debug logging and *LOG* for regular logging, defined as:

```
#ifdef DEBUG
#define LOGD( a ) if(0) ; else {Log __internallog( "DBG1" ); __internallog << a; }
#else
#define LOGD( a ) if(1) ; else {Log __internallog( "DBG1" ); __internallog << a; }
#endif

#define LOG( a ) {Log __internallog( "INFO" ); __internallog << a; }
```

The *DEBUG* define could be set per module. If set, then a local logging instance was created and the message output. When the ending brace at the end of the line was met, the logger was deleted. Either way, the if statements were either always true or always false, and would get optimised out by the compiler. This gave zero runtime overhead without DEBUG set.

Other solutions, such as the runtime checking of log levels and having a more complex levels and log output system, were discounted, because they would have been more complex and required longer to implement (when this was a very small part of the project) and would not have had zero runtime overhead.

3.5 Data storage

Data storage was performed by providing $<<$ and $>>$ operators for the different classes that needed to be loaded and saved from disk. This meant that the standard file and other I/O streams would be able to read and write the data, properly formatted, with type safety.

As the design dictated that the partitioning would be a preprocessing step, the use of some temporary storage for partitioned data was inevitable. However, there was some choice in how the partitioned data was stored.

The method of having data being stored in a single file, with a lookup table at the start of the file pointing to the data for a single partition, was ruled out as it would be unnecessarily complex to implement, and because in a message-passing system without a shared filesystem, the data for other nodes would unnecessarily have to be stored on all the nodes.

Two other options were considered:

1. The partitioned terrain data could be stored in a file for each node. This had the advantage of simplicity and meant that the sequential Dijkstra implementation could be straightforwardly executed on each individual data partition for testing purposes. It also meant that a quick glance at the file could show the number of faces in the file, and hence how evenly partitioned the data was between the partitions. However, more disk space than necessary was taken up, due to the duplication of the terrain data between all the different partition sizes and partitioning methods.

2. Partitioning data could be stored in a file for each node, referring to the unpartitioned terrain data file. This would use less storage and be more scalable as the actual terrain data would not be duplicated.

The first method was implemented due to it being easier to implement, and disk space was not at a premium.

Chapter 4

Testing

Once implementation was complete, the software had to be tested for correctness, performance, and a determination made about ease of programming. The methods used and rationale for the relevant decisions are given in this chapter.

4.1 Verification

4.1.1 Component testing

The first stage in verification was to create test harnesses for key elements of the system in order to fully test them without the complexities of testing within the full-blown algorithm program environment. In an ideal situation, with sufficient time and with a more complex system than this one, test harnesses would have been created for all components, ideally before the components themselves were implemented (test driven development), to form good tests and to help define the exact behaviour of the components. However, the small amount of time available meant that test harnesses were created only when there appeared to be a problem with a system that was hard to track down due to the complexities and variations in runs of the algorithm.

A test harness was created for the logging system, *test_log*, to ensure that the logging operated as expected and did not miss any messages. If problems had occurred when the logging system was used in a multithreaded environment, this test harness would have been expanded to use multiple threads, but as it happened the thread safety appeared to work perfectly well and the test harness was not expanded to cover this.

Another was created as a simple test of MPI functionality, *test_mpi*. This was more of a test to ensure the libraries and headers were set up correctly, and did not do an exhaustive test of MPI operations.

The final test harness created was to test the queueing algorithms, *test_queue*. This tested both value and pointer implementations of the queue, to ensure items were added and removed as expected.

The general method for programming the test harnesses was to have operations that used the items being tested together with the expected outcome of the operation. Assertions were written which were tested after each operation to ensure the assertions were true. If the assertion was false, an exception was thrown and the test harness exited, giving details of the errors encountered.

4.1.2 Algorithm testing

After the components appeared to work correctly, the next step was verifying that the algorithms gave the expected output. This was done by configuring the algorithm applications to output the costs found for each Steiner point after the run had completed. The sequential Dijkstra implementation was used as a basis for comparison, and the results compared with several runs, mainly with the small *ls_samp* data set, of the parallel algorithms, using different processor configurations and partitioning schemes.

It was difficult to directly compare the output generated, as the sequential Dijkstra implementation was using an unpartitioned data set, whereas the parallel algorithms were using partitioned data except when testing on a 1x1 processor configuration. Testing with more processes therefore required finding the edge index in a particular partition that matched the edge index in the unpartitioned data, and checking the costs. This was performed manually for several different Steiner points against the sequential Dijkstra output. More thorough testing was performed between different parallel implementations, as they were using the same partitioning and the edge and Steiner point indexes matched between them. GNU diff [23] was used to compare these outputs.

4.2 Performance testing

4.2.1 Choice of systems

The goal here was to get a cross-section of results for different systems that would be comparable across shared memory and message passing implementations.

The main target platform during the project was a SunFire, a 64 processor shared memory system. This had implementations of MPI and PThreads available, and testing upon it would have allowed a direct comparison between MPI (using message passing with fast shared memory communications) and PThreads (directly using the shared memory capabilities). In addition, during performance runs the operating CPUs would have been reserved purely for the tested application - there would have been no contention with other users or software - which should have given very good data. Unfortunately, the system was unavailable due to extended maintenance from 6 weeks before the end of the project, during the performance testing period.

As the SunFire was unavailable, testing was carried out instead on a 4-way Symmetric Multiprocessing (SMP) machine - a server with 4 Pentium-4 Xeon 2.8GHz processors and 4GB RAM. This allowed MPI and PThreads implementations to be compared on the same machine. Unfortunately, this system was far from ideal as it was the main departmental server and always heavily loaded. This was still the best machine available to do this kind of testing on, and the results are presented here. However, due to the heavy loading, often the separate threads and processes were not actually running on separate CPUs and not getting much CPU time. Though many execution runs were made, this effect was still significant and has affected the results. This system is referred to in the results as ‘4-way SMP’ or ‘smp4’.

Another target platform was a Beowulf cluster, composed of 16 Pentium-4 Xeon 1.7GHz Linux machines with 2GB of RAM each. A Beowulf cluster is a cluster of inexpensive machines with a dedicated network interface for message passing between the nodes. Use of the cluster allowed an examination of how the MPI implementations scaled in a true distributed memory environment. The original plan was to use a 64-node Beowulf cluster that was available. However, it used Redhat 7.3, when other departmental computers were using Redhat 9.0, which meant that applications compiled on the other systems would not execute, due to the differing library versions. Additionally, applications could not be easily compiled on the system due to it only having an old version of gcc, which would not work with the C++ code in this project (which made heavy use of templates in places). Although compiling a local copy of gcc on the Beowulf was not infeasible, it would have meant a much longer development round-trip time during testing (with testing changes locally, then having to recompile on the Beowulf with its slow, overutilised disk subsystem), and it was decided to use the 16-node Beowulf instead. As mentioned, a single disk subsystem appeared to be shared between all the nodes on both clusters, which meant disk access was extremely slow. This was factored out of the testing by just recording the algorithm time and excluding the load times, by the use of timing statements at the appropriate code locations. Another problem was that access was not controlled to the Beowulf and other processes could compete with tests and cause times to be longer. No solution was found to this problem. However, the 16-node cluster did not appear to be heavily loaded during testing, and sufficient runs were made where the effect of this should have been minimal. This system is referred to in the results as ‘16-node Beowulf’ or ‘bw16’.

For one of the tests (comparing standard sequential Dijkstra against the version with termination when the target is reached), a Pentium-M 1.4GHz system with 768MB RAM was used. This system is referred to as ‘uniprocessor’ or ‘uni’.

4.2.2 Variables chosen

The initial goal was to see how the algorithms scaled with:

- Problem size. This is the size of the terrain in terms of number of faces. Note that with a TIN, the number of faces is proportional to the number of vertices (there is only one extra vertex for every added face). This is equivalent to the size of the graph.
- Number of processors. The processors were treated as a 2D mesh of processors that the partitions got allocated to.

During the implementation process, the impact of the following variables were also tested:

- Partition type. The initial plan was to just test with the block partitioning, then switch to MFP. However, it was initially anticipated that MFP would be faster in all cases, when further examination showed that it should only be faster for the Lanthier algorithm. Examining this was useful as it could be seen how unevenness in the partitioning affected the results.
- Queue type. The initial plan was just to use the sorted queue, and the implications of this were not

realised until well into the implementation and testing, simply due to a lack of thought about the queue. It was originally planned that the sorted queue would just be a holdover until a Fibonacci-heap based queue was implemented, but time did not allow for this. The unsorted queue was implemented, and new results with this queue compared against the old results.

4.2.3 Execution

A script was written in Python in order to create data files for the different types of partition, for the different processor configurations and for the different input data sets. This script was then executed to do the partitioning on each host system.

Another script was written in order to carry out the execution of the tests. It allowed for automatically running *util_pickvertex* to pick a random vertex from the appropriate dataset for source and/or target, then running each different program appropriately across the different sets of input data. The script ran each program several times in order to attempt to minimise the effect of other processes intermittently running at the same time on the machine. The information written by each algorithm implementation to standard output was captured and redirected to a file, named so it could be uniquely identified, including a timestamp and hostname information.

Originally, each algorithm application's output to standard output, as well as including information on the time taken for various subtasks during execution, also included the costs to reach each vertex. This information was useful during debugging, but caused a lot of disk space to be used during testing runs, and the cost output was suppressed for the purposes of the performance testing.

4.2.4 Result processing

Due to the volume of information, it would have been impossible to manually collect the data and add it into a spreadsheet. Instead, a script was written in Python to process the result logs from the programs into data files and GNUPlot scripts - which GNUPlot then processed to produce the graphs in this document. A simple mean and standard deviation of each run was calculated, and 1 standard deviation error bars were plotted on the graphs in order for the reader to gain some idea of the accuracy of the data. Minimum and maximum data was also gathered so this could be used instead of the standard deviation error bars, where appropriate.

No attempt to remove outliers was made, because although the run times for the Dijkstra algorithms should have been relatively stable between runs due to them not stopping when the target vertex was found (though different sources still give different runtimes due to the degree to variations in partitioning), the Lanthier algorithm terminated early when the target was found which caused a significant variation in times. Removing outliers would be an exercise in futility for such an algorithm. Instead, the number of runs completed was made as high as possible in order to gain a more accurate average time.

4.3 Technology comparison

4.3.1 Measurement of programming complexity

Software complexity is notoriously difficult to measure, but number of Source Lines of Code [24] (SLOCs) [24] and an estimate of programming and debugging time (based on recollection and source code control logs) was used to judge this. Although these measures are in themselves objective, the values were influenced by the implementor's experience and even the order in which the implementations were done.

SLOCs are defined as (from [24]):

> A physical source line of code (SLOC) is a line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character.

This metric was readily derived using free SLOCCount [24] software.

Other techniques which claim to provide a more valid estimate of coding complexity were briefly evaluated, including Function Point Analysis [25] and logical SLOCs [24]. Though these may give a better estimates of program complexity, tools were not readily available to do automated analysis, and due to the inaccuracies in these estimates, it did not seem appropriate to carry out such a lengthy analysis.

4.3.2 Measurement of mental complexity

In addition to the metrics of physical complexity, a judgement was made on how difficult the problems were to think about. This was purely subjective, but perhaps still valuable.

Chapter 5

Results and Analysis

5.1 Correctness

A simple demonstration of correctness will be given here, though a more thorough check was carried out during testing.

Picking a random vertex from the *ls_samp* data set to act as the target: $(1,1,0.220441)$. This vertex has index 2 (0 based) in the unsplit data. It is the first vertex of edge with index 2. When split with a 2*x*2 processor configuration and block partitioning, the vertex has index 2 and the edge index 2 in cell $(1,1)$

Picking another random vertex to act as the source: $(0.917575, 0.516713, 0.677355)$.

Executing *test_seqdijkstra idx/ls_samp_idx.txt 0.917575 0.516713 0.677355* gives the cost for edge 2, Steiner point 0 as 0.205406.

Executing *test_ptdijkstra part/ls_samp_idx.txt/block/2x2/part.txt 2 2 0.917575 0.516713 0.677355* gives the cost for thread 3, edge 2, Steiner point 0 as 0.205406.

Executing *mpirun -np 4 test_mpidijkstra part/ls_samp_idx.txt/block/2x2/part.txt 2 2 0.917575 0.516713 0.677355* gives the cost for node 3, edge 2, Steiner point 0 as 0.205406.

Executing *mpirun -np 4 test_mpilanthier part/ls_samp_idx.txt/block/2x2/part.txt 2 2 0.917575 0.516713 0.677355 1 1 0.220441* gives the cost for node 3, edge 2, Steiner point 0 as 0.205406.

The algorithms agree, the value found is reasonable and appears to be correct.

5.2 Performance

Only the result graphs referred to by the text are contained within this section. Wherever possible, variables have been kept fixed in order to ensure the results are presented in as clear a way as possible. However, a full set of the primary result graphs is available in Appendix C.

The algorithm time given in the following graphs and tables was derived as the maximum time taken for the algorithm by any of the threads or processes during a particular run, averaged over all the runs. It excludes the time required to load the data, set up the initial data structures (such as the lookup tables for determining edges incident with a vertex etc) and write out the results. These elements were excluded so as to highlight the time taken by the core of the algorithm and hence emphasise the differences between the results from the different programming styles.

5.2.1 Sorted vs Unsorted Queue

Figure 5.1 shows how the execution time of the MPI Dijkstra implementation varies with number of processors, using the *nn_65386* data set, the largest data set tested and the MFP partitioning scheme on the 4-way SMP machine. Plots are shown for both unsorted and sorted queue implementations.

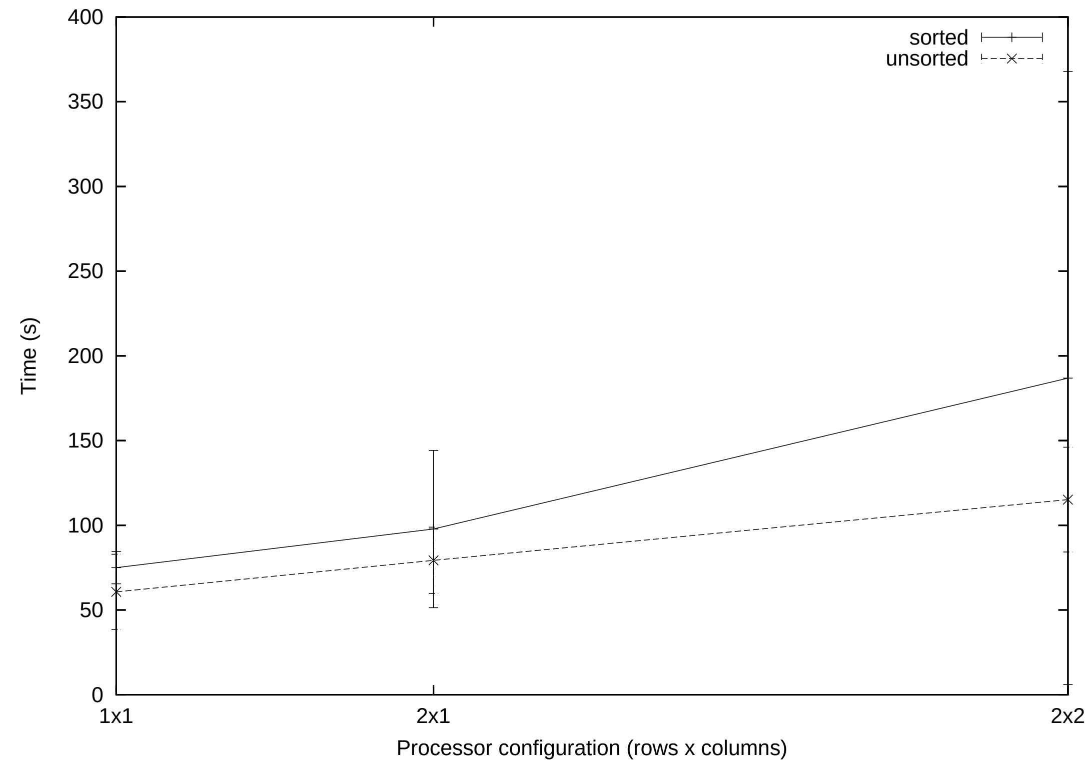

Figure 5.1: Algorithm time vs. number of processors for unsorted and sorted queues, MPI Dijkstra, SMP.

In selecting this graph, the largest data set was chosen to give the clearest difference in runtime. The MFP partitioning scheme was selected because it gives a more even number of vertices in each partition (though as noted in Section 1.3.2, the number of vertices tends to increase with the increasing number of processors), so highlighting any possible parallelism. The 4-way SMP machine was chosen as the Beowulf did not perform well with the Dijkstra implementation.

As can be seen, there is slow-down for both types of queues, but less so with the unsorted queue. The difference in performance between the queues increases as the number of processors increases, which would seem to show that the unsorted queue scales better with the number of processors. However, there is slow down with both, it is not possible to verify the assertion from Section 2.4.3 that no useful parallelism occurs with the sorted queue.

Figure 5.2 shows the same graph, but for the MPI Lanthier implementation on the 16-node Beowulf.

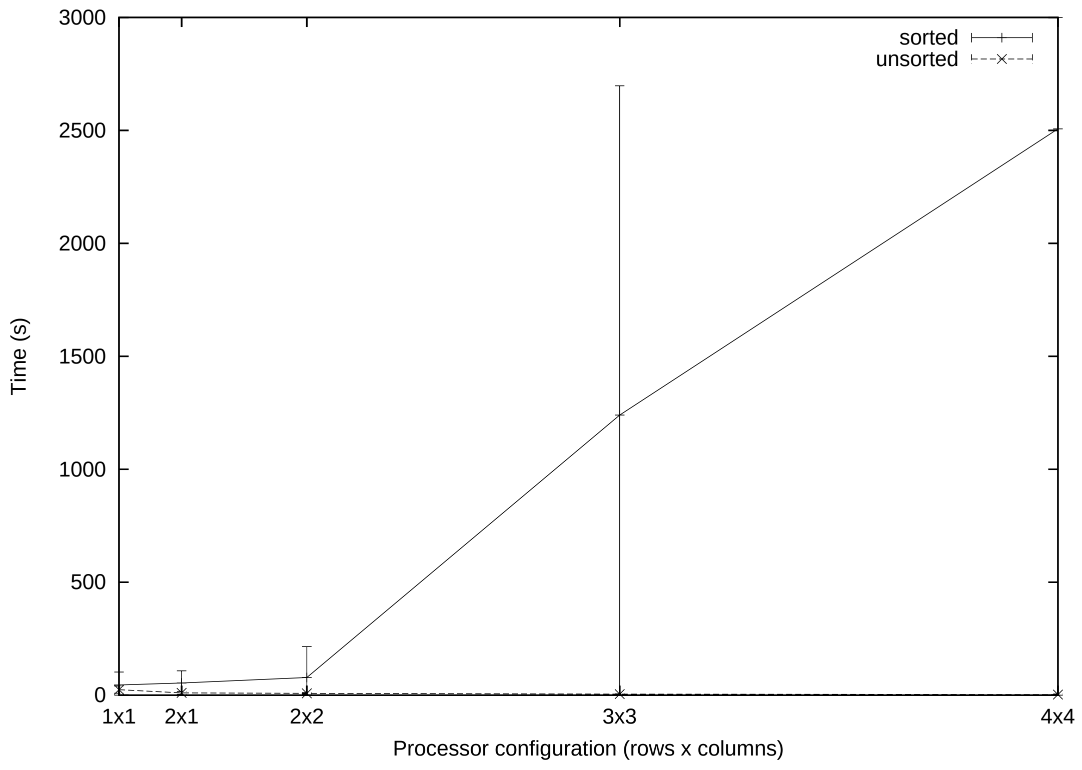

Figure 5.2: Algorithm time vs. number of processors for unsorted and sorted queues, MPI Lanthier, Beowulf.

Here, the Beowulf was selected because it performed reasonably well with the Lanthier algorithm and as it had 16 processors, the graph could show the scaling with a larger number of processors than was possible with results from the SMP machine.

As can be seen, the sorted queue causes extremely significant slowdown with the Lanthier algorithm. The unsorted queue gives much faster times (near the bottom of the graph scale) and has a speed-up when more processors are used. This is likely due to the reason stated in Section 2.4.3 - a lot of insertions onto the queue are required with this algorithm, which are $\Theta(1)$ with the unsorted queue rather than $\Theta(v)$ with the sorted queue, where v is the number of vertices.

Note that the PThread Dijkstra implementation's sorted vs. unsorted queue results were not directly comparable (see Section 3.4.2), so they are not compared here.

The results to follow exclusively use the unsorted queue because, as stated in Section 2.4.3 and demonstrated here, there were parallel scaling problems with the sorted queue.

5.2.2 Dijkstra vs Dijkstra with early termination

As stated in 2.4.1, it was expected that the Dijkstra algorithm with early termination when the target vertex is reached would have an average runtime half that of the regular Dijkstra algorithm, but with variation between zero and the time of the regular algorithm causing significantly increased jitter. A graph of algorithm time vs data set size for Dijkstra without early termination (seq) and with early termination (seq-tgt) for a uniprocessor system is shown in Figure 5.3. The values shown here are averages over runs, with error bars showing the minimum and maximum times over the runs.

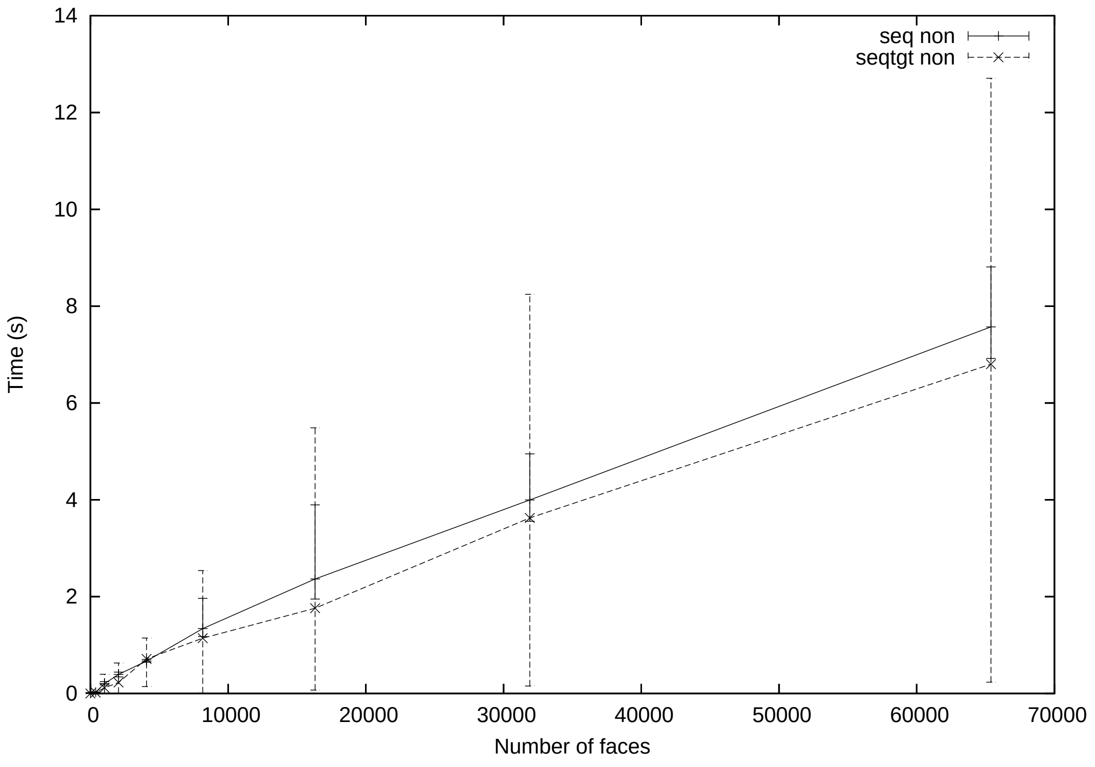

Figure 5.3: Algorithm time vs number of faces, sequential Dijkstra with and without early termination, Uniprocessor

The large error bars for the Dijkstra with early termination clearly show the increased jitter. However, though the minimum times were close to zero, as expected, the maximum times were around 1.5 times the time for sequential Dijkstra without early termination. As the values were distributed between these minimum and maximum points, the average time with early termination was higher than expected, though about 1.3 times faster than without early termination.

In order to do the early termination, the position of every vertex removed from the queue was checked against the target position. This meant that there were v vector comparisons (where v is the number of vertices), or $3v$ floating point comparisons. It is likely that this testing caused the slowdown, and caused the average time to be greater than that expected.

Nevertheless, the decreased jitter without early termination still means not doing early termination was a good idea, but when Lanthier and Dijkstra are compared, factoring this difference between the algorithms in should only be done by dividing the Dijkstra time by around 1.3. Dijkstra results to follow are without early termination.

5.2.3 MFP vs. block partitioning

Section 1.3.2 claimed for Dijkstra's algorithm, a block partitioning is non-optimal because of an uneven distribution of vertices between the partitions, but MFP partitioning is likely to be even slower due to the increase in number of vertices caused by overpartitioning. It also claimed that for Lanthier's algorithm, block partitioning does not make best use of the available processors and MFP gives a greater degree of parallelism.

Figure 5.4 shows the average algorithm times on the 4-way SMP machine for the different types of algorithm and partitioning method.

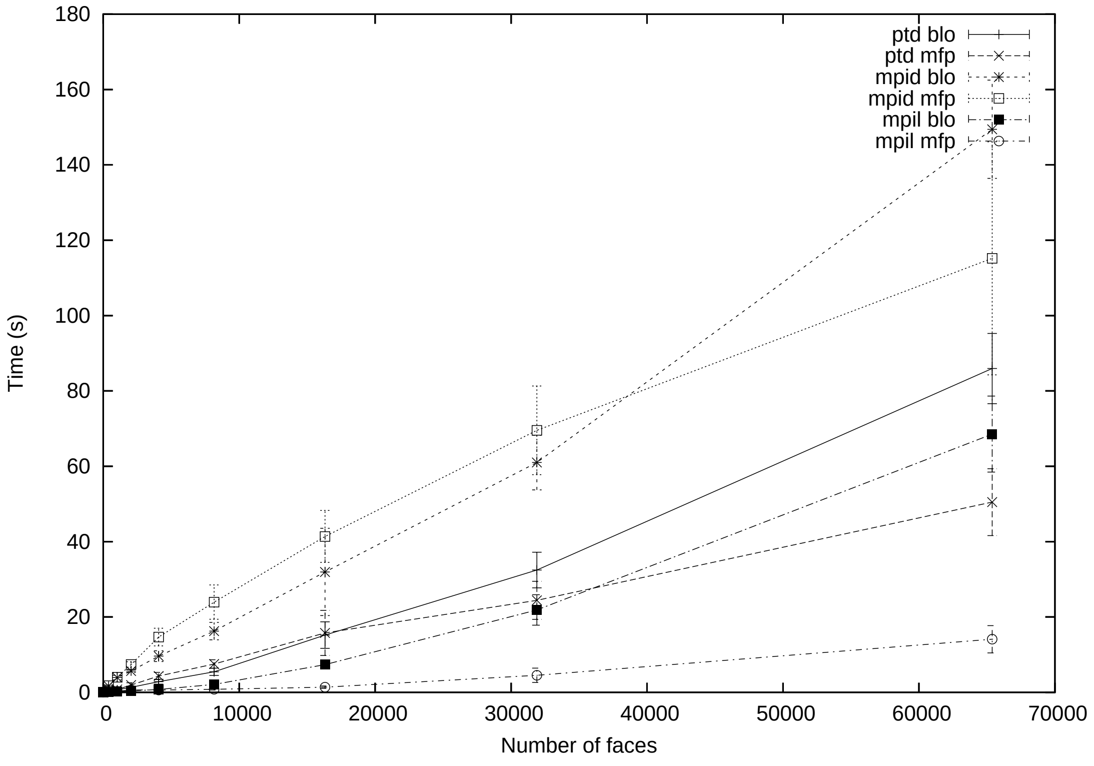

Figure 5.4: Algorithm time vs number of faces, 2x2 processors, SMP

For both Dijkstra implementations the block partitioning appears slightly faster below 30000 faces, though above this MFP seems slightly faster. However, all the results are within the 1 standard deviation error bars and it is not possible to conclusively say that one is faster than the other. It is quite possible that the 2x2 processor configuration did not lead to a large enough degree of overpartitioning for the increased number of vertices to have an apparent effect on performance. More experimental runs with more processors would allow a conclusive answer to be found here.

In contrast for MPI Lanthier, the MFP partitioning gives significantly faster execution, as expected.

Figure 5.5 shows the same graph but for 9 nodes of the Beowulf.

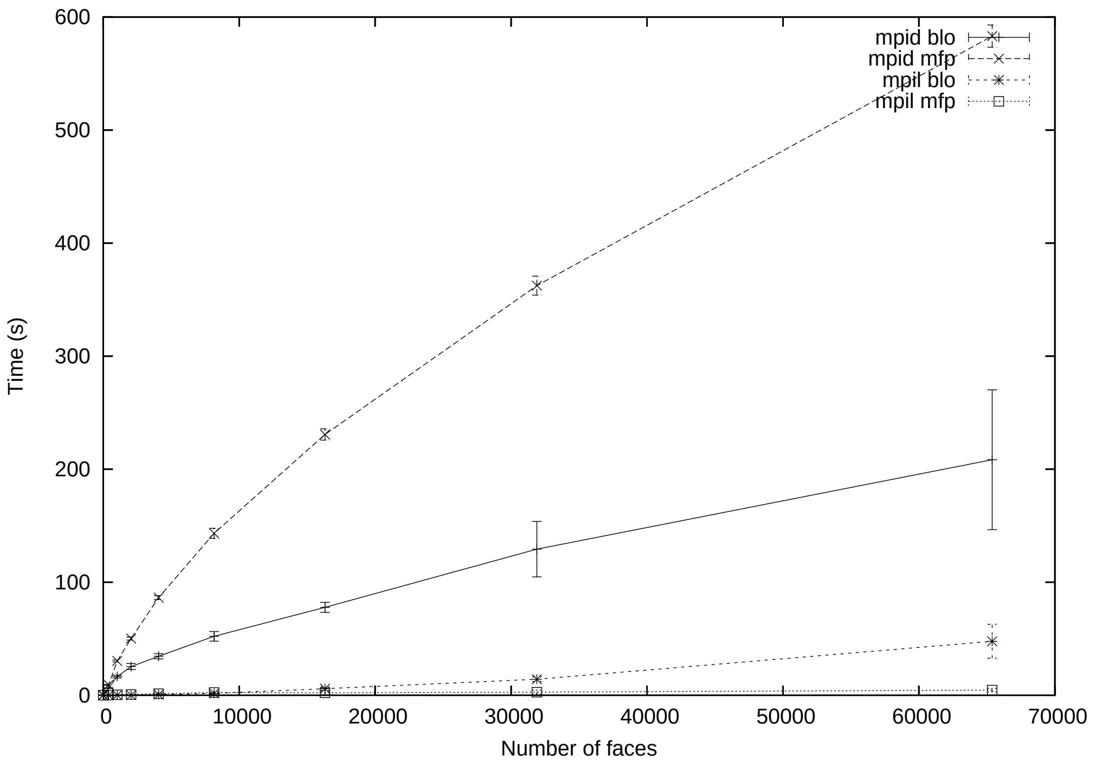

Figure 5.5: Algorithm time vs number of faces, 3x3 processors, Beowulf

The graph for 3x3 rather than 4x4 is shown, because with 4x4 processors the Lanthier implementation is so much faster that it is hard to see when plotted on the same axis as the Dijkstra results.

On this system, with MPI Dijkstra the MFP partitioning gives a significantly slower runtime. This was as expected (due to the overpartitioning leading to more vertices) and is likely more pronounced here due to the larger number of processors used leading to a greater degree of duplication in the vertices.

With MPI Lanthier, the MFP partitioning gives a significantly faster execution, as expected.

The sections to follow use results based on block partitioning for the Dijkstra algorithm and MFP partitioning for the Lanthier algorithm.

5.2.4 Message passing performance on a message passing system

Figure 5.6 shows the performance of the Lanthier algorithm on different processor configurations on the Beowulf. The performance of the sequential Dijkstra algorithm on a single node of the cluster is also shown, for comparison. Note that the Lanthier algorithm with 1 processor is faster because it does early termination when the target is found.

As can be seen, with a small number of faces, the smaller number of processors gives the fastest result. However, as the number of faces in the dataset increase, the optimum number of processors increases until with the largest dataset, the 4x4 processor configuration gives the fastest performance. This is likely because each added processor in the MFP partition causes extra duplication of vertices and additional

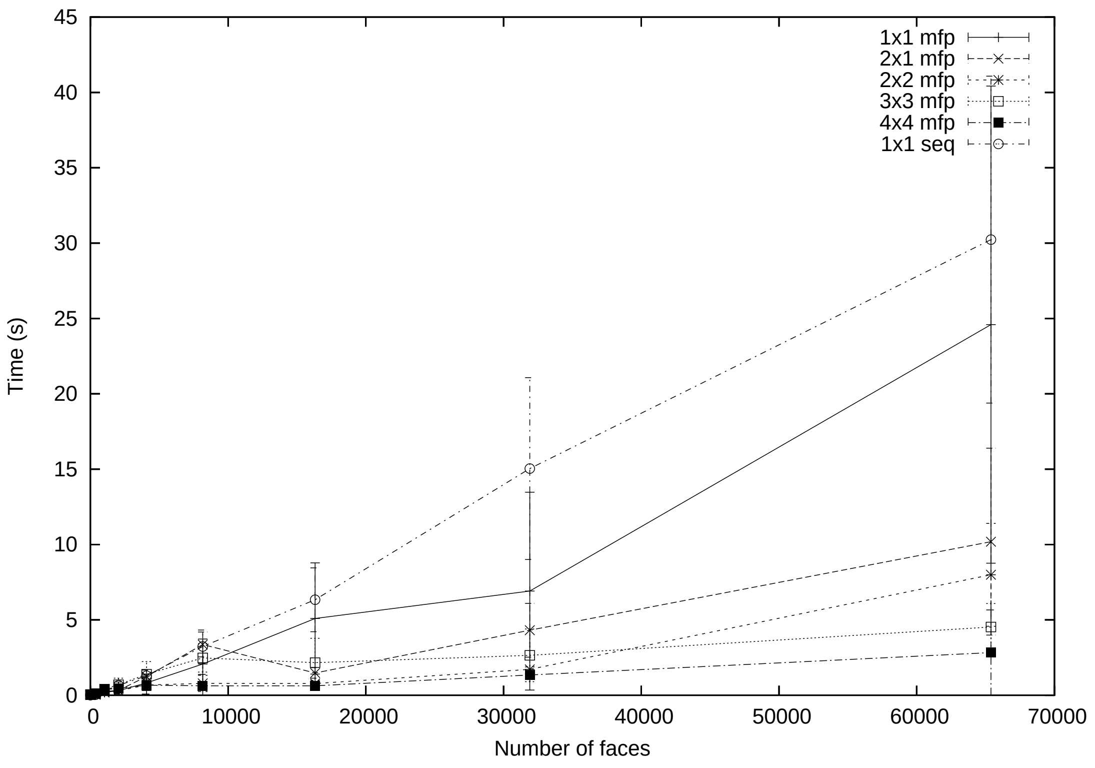

Figure 5.6: Algorithm time vs number of faces, MPI Lanthier and Sequential Dijkstra, Beowulf

communication costs in updating them. As the number of processors increases relative to the number of vertices, these communication costs become more important, giving the results shown.

Section 5.2.6 compares the performance of this MPI Lanthier implementation and that described by Lanthier et al. [12].

Figure 5.7 shows the performance of the MPI Dijkstra algorithm on different processor configurations on the Beowulf.

The MPI Dijkstra performance is very poor compared to the MPI Lanthier performance on the Beowulf, and increasing the number of processors causes the slowdown to worsen. Every processor configuration has a decreased absolute runtime compared to the sequential Dijkstra implementation, even only using 1 processor. This slowdown with 1 processor is due to the overheads incurred in checking the list of shared vertices and edges (even though they are empty) and making calls to MPI Reduction and Scatter, even though no actual communication was involved. The parallel performance is likely very poor because it is dominated by the communication time (the time taken to do the reduction and scatter each iteration).

Träff [26] investigated different parallelisations of Dijkstra's algorithm. The algorithm implemented here for both message passing and shared memory was the one he referred to as the *simple parallel algorithm*. He noted that it was wholly communication-bound and, though its performance varies with the particular graph, often it leads to slow-down rather than speed-up versus a sequential implementation. That is also what is shown here. He said that performance can be improved by making various modifications, the best of which was to move to an approach based on updating costs rather than finding a global minimum. This update approach is the one taken by the Lanthier algorithm, and the performance

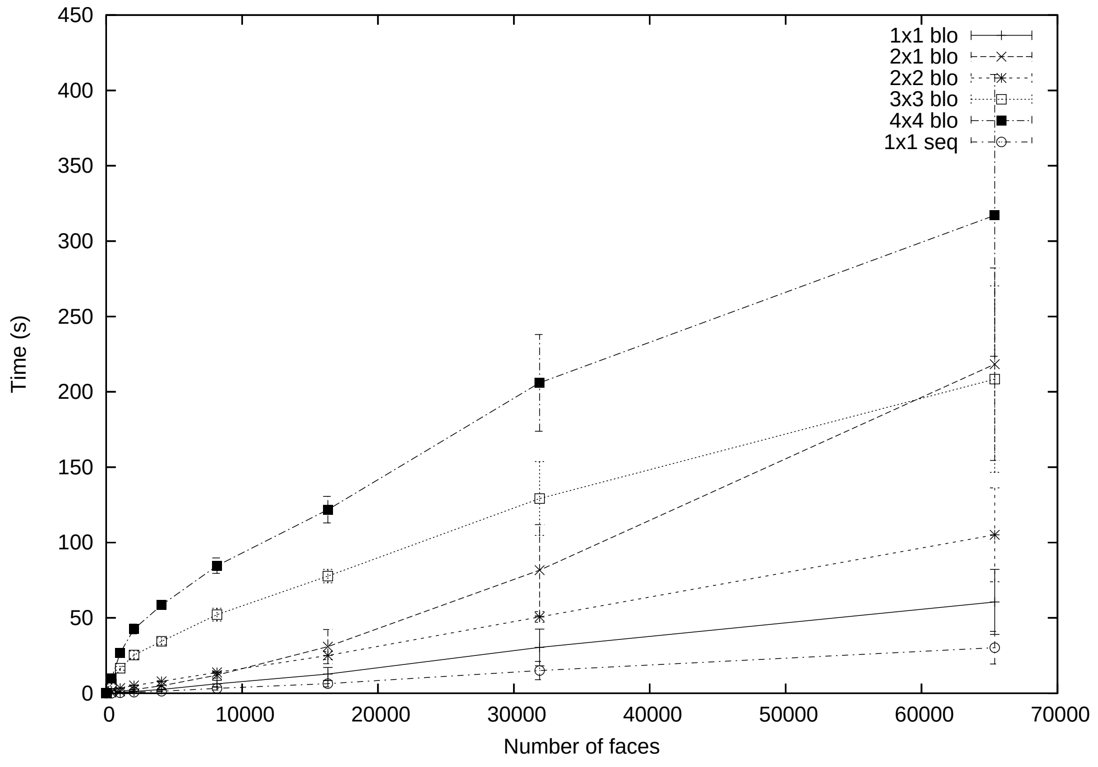

Figure 5.7: Algorithm time vs number of faces, MPI Dijkstra and Sequential Dijkstra, Beowulf

difference is clearly demonstrated here.

Figure 5.8 shows the algorithm time when the communications time (reduction and scatter times) has been factored out.

This clearly demonstrates that the communications time is indeed the reason for the slowdown, with more processors requiring less actual computation per processor.

This is also demonstrated by comparing with the results from the 4-way SMP system, which obviously has much faster MPI communication as it is done using shared memory rather than TCP. Figure 5.9 shows the MPI Dijkstra algorithm time on this system and shows at least some scaling with number of processors. Shared memory results are discussed further in Section 5.2.5.

5.2.5 Message passing vs shared memory programming styles on a shared memory system

In order to compare the two programming styles directly on the shared memory system, the MPI and PThread Dijkstra implementations were examined, executing on the 4-way SMP machine. As mentioned in Section 4.2.1 this machine had varying loads during the testing and the results here are likely to be inaccurate because of this and any conclusions drawn here must be viewed with caution.

Figure 5.10 shows the performance using one processor and demonstrates the overhead involved in the different implementations just being executed on the single processor. The sequential Dijkstra algorithm

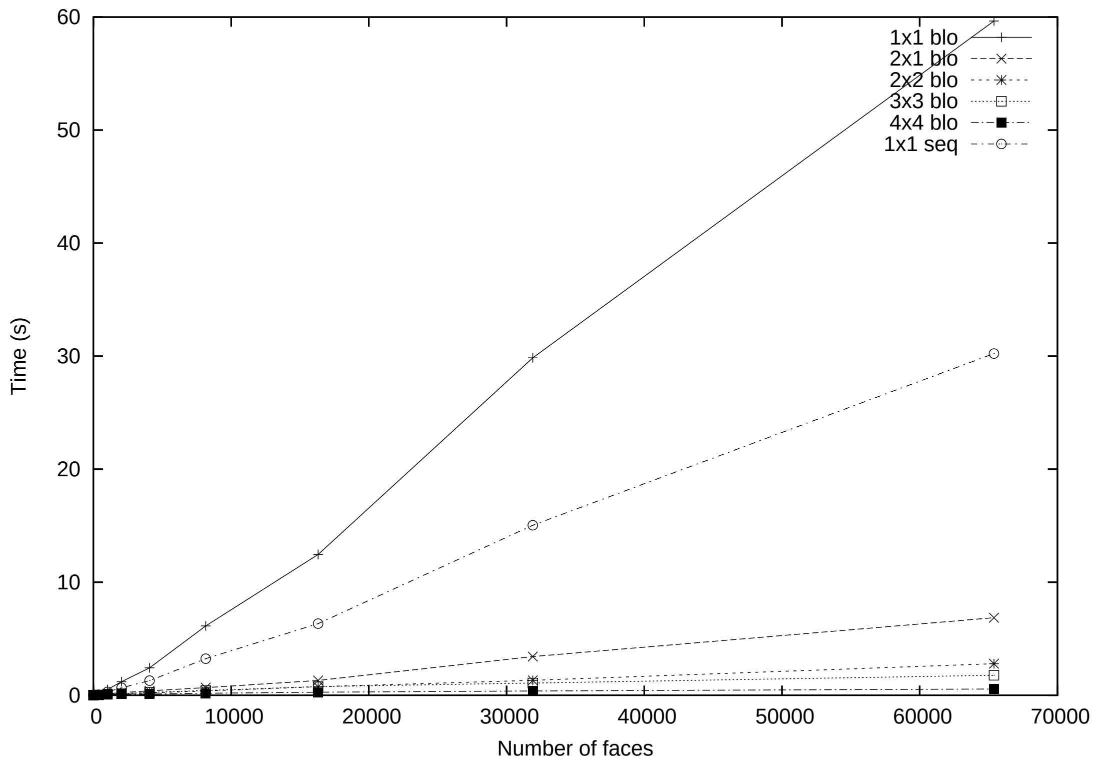

Figure 5.8: Non-communication algorithm time vs number of faces, MPI Dijkstra and Sequential Dijkstra, Beowulf

acts as a baseline for comparison. The PThread Dijkstra implementation is about 1.5 times slower than the sequential implementation. This overhead is caused by data structures being locked unnecessarily. The MPI Dijkstra implementation is about 3 times slower than the sequential implementation. This is due to the greater overhead of the calls into the MPI library and the fact that data is copied into buffers for sending and receiving, and then likely copied again by the MPI implementation into its internal buffers. This copying of data is unavoidable when using a message passing system and can be significant in a shared memory architecture. By contrast, in a message passing architecture, the time taken to copy data is generally negligible compared to the communication time.

From this result, it appears that when cache coherency is not an issue, the threaded shared memory approach has the potential to give better performance than a message passing approach.

Figure 5.11 shows the performance using two processors.

The first thing to note is that the performance of both algorithms is very poor, demonstrating considerable slowdown compared to the sequential implementation. Once again however, the PThread implementation gives better performance, though the margin of difference is smaller than with the single processor. It is likely that the difference is smaller because cache coherency is now a factor.

Figure 5.12 shows the performance using four processors.

Once again the PThread implementation gives better performance, with still significant slowdown all around.

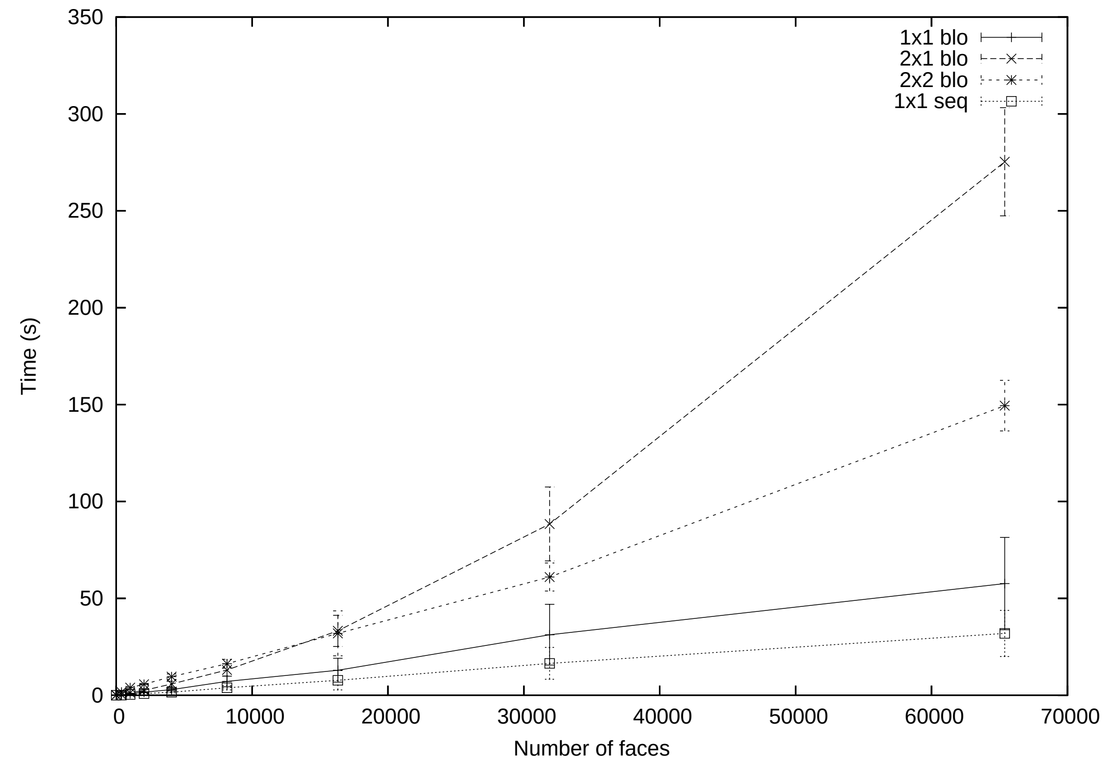

Figure 5.9: Algorithm time vs number of faces, MPI Dijkstra and Sequential Dijkstra, 4-way SMP

With all the processor configurations, the MPI implementation was within a constant factor of the performance of the PThreads implementation, and there did not appear to be any issues with the scaling of the implementations using either programming style as the size of the data set increased.

Figure 5.13 shows the performance of the PThread Dijkstra implementation with the different processor configurations tested on the 4-way SMP.

As can be seen the 2x1 processor configuration performed significantly worse than the 2x2 configuration, though both performed worse than a single processor. Referring back to Figure 5.9, it shows the performance of the MPI Dijkstra implementation with the different processor configurations scaled exactly the same as with PThread Dijkstra.

The reason for this poor performance is unclear, and though it is possible that adding additional processors causes significant overhead which is only overcome once a certain extra number of processors are reached, it seems more likely to be to do with the fact that the processors were heavily loaded at the time. It was not possible to reserve particular processors, or even force the threads to execute on different processors, and a process migration daemon constantly moved processes to balance the load on the system. Possibly four concurrent threads would be more likely to be allowed to run on separate processors than just two. Undoubtedly context switching was part of the reason for the slowdown.

Another reason may be that the 2x1 block partitioning led to an uneven distribution of the vertices whereas the 2x2 configuration gave a more even split. Figure 5.14 shows the 16314 face TIN data (by way of an example), and it does indeed show a greater density of vertices on the left hand side that would cause less improvements to be noticed in a 2x1 split than, say, a 1x2 split or 2x2. Further tests

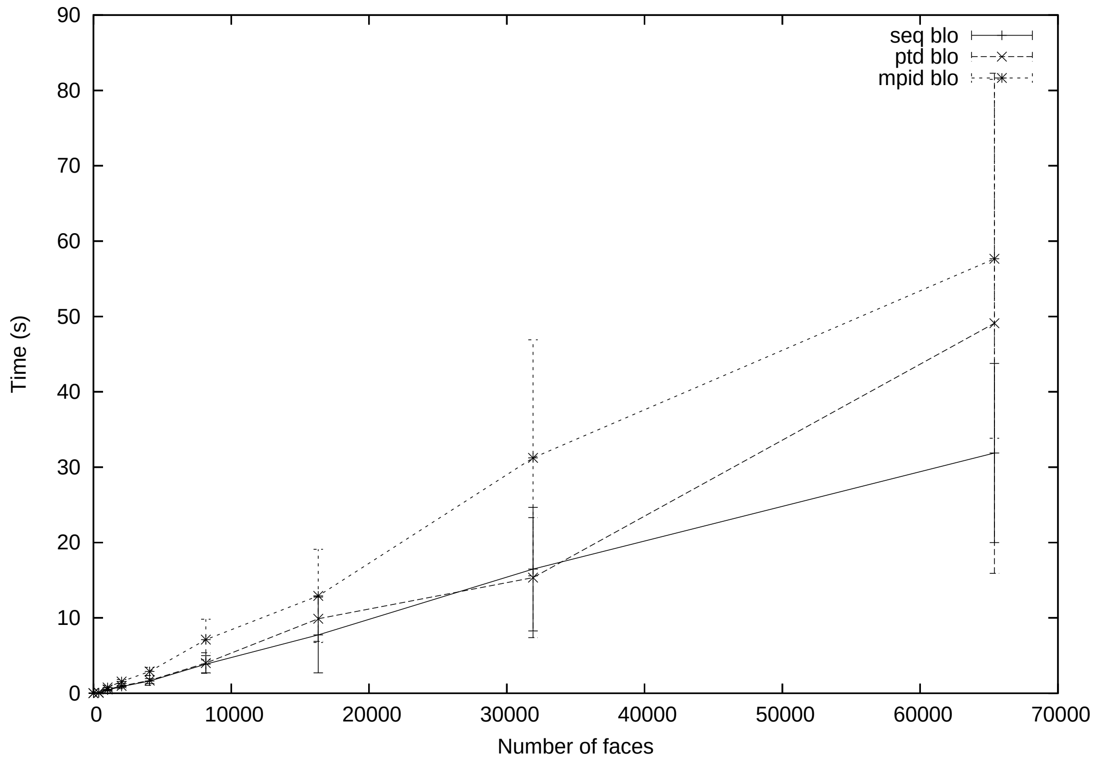

Figure 5.10: Algorithm time vs number of faces, PThread Dijkstra, MPI Dijkstra and Sequential Dijkstra, 1x1 processors on 4-way SMP

would be required of a 1x2 split or differently distributed data in order to rule this out as the cause for the peak with 2x1 processors. If tests had been possible on higher order processor configurations, this point would be less important.

Based on the above results, although a shared memory programming style appears as if it might lead to faster performance, it is not possible to say conclusively how the different styles scale with increasing number of processors, unless further tests are carried out on a system with exclusive use of the processors.

5.2.6 Comparison of Lanthier algorithm with previous results

The results of Lanthier et al. [12] for performance of their algorithm on a Beowulf cluster are reproduced in Figure 5.15.

The comparable results gained here are shown in Figure 5.16. The sequential implementation used for comparison was the sequential Dijkstra with the unsorted queue. This is not the fastest known sequential algorithm because of the use of the unsorted queue rather than an efficient heap implementation, so strictly speaking the speedup comparisons are not accurate, but it acts as a good basis for comparison here, particularly as the Lanthier implementation used also used an unsorted queue. Also, in Section 5.2.2 it was found that the sequential implementation would be 1.3 times faster with early termination, but here using this factor gave efficiencies of greater than 100%, so a factor of 1.6 was used to account

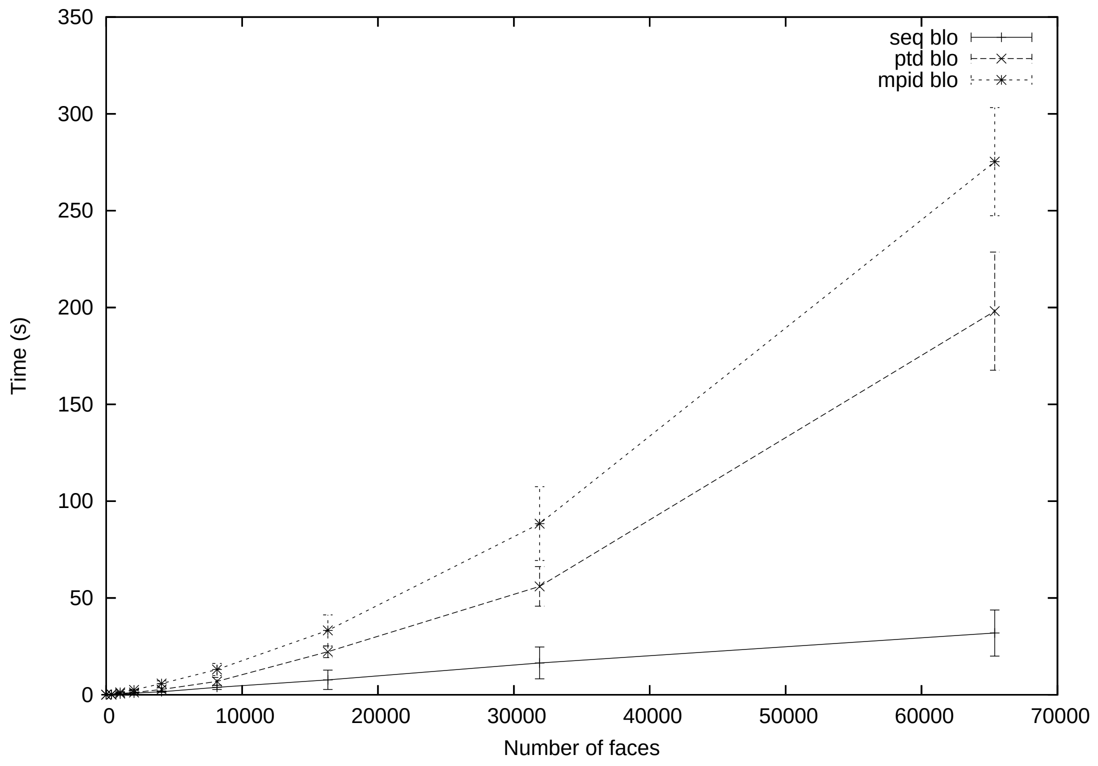

Figure 5.11: Algorithm time vs number of faces, PThread Dijkstra, MPI Dijkstra and Sequential Dijkstra, 2x1 processors on 4-way SMP

for the early termination of the Lanthier algorithm.

The results for the large data set are very similar to those found by Lanthier et al. and show that the MPI Lanthier implementation was a reasonable one. The poorer efficiencies with the smaller data sets (significantly smaller data sets than those in Lanthier et al.'s graph) are to be expected because of the costs of communication in the Beowulf system.

5.3 Implementation complexity

5.3.1 Programming complexity

Table 5.1 gives a list of SLOCs (see Section 4.3.1 for definition) for each of the top-level implementations, excluding the lines which output logging data. The PThread implementation also gives the SLOCs including the barrier code, which is considered a library item rather than necessary for this particular implementation, but is noted here for completeness. The core code was the same for each implementation, and is not considered here.

As the table shows, the number of lines of code was significantly more for the PThread implementations than for the MPI implementation of the Dijkstra algorithm. This was likely because MPI presents a higher level interface to the programmer than PThreads does.

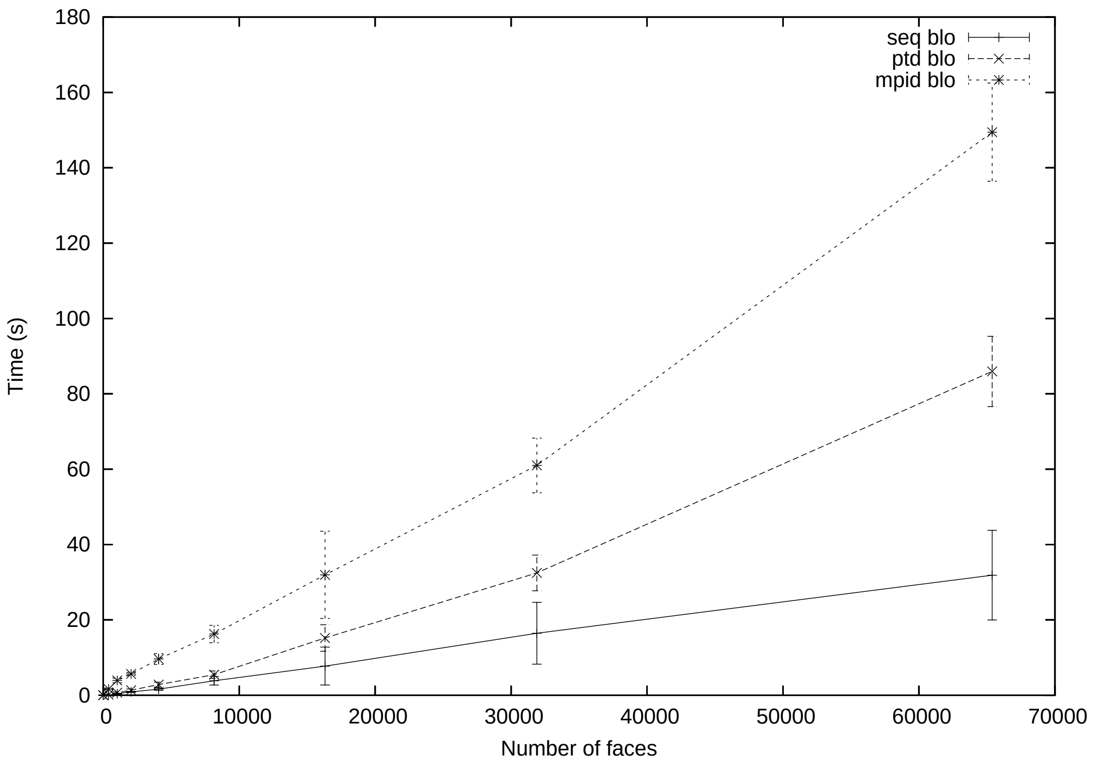

Figure 5.12: Algorithm time vs number of faces, PThread Dijkstra, MPI Dijkstra and Sequential Dijkstra, 2x2 processors on 4-way SMP

For example, consider the PThreads-based code snippet below, which safely sets the global value *g_v* to the minimum of all threads' *v* values. This following would be executed by all threads:

```
if( g_v > v )
{
  pthread_mutex_lock( &g_mutReduce );
  if( g_v > v )
  {
    g_v = v;
  }
  pthread_mutex_unlock( &g_mutReduce );
}
```

The MPI-based equivalent is shown below, with the local value in *s* and the global value received into *r*:

```
MPI::COMM_WORLD.Allreduce( &s, &r, 1, MPI::INT, MPI::MIN );
```

This demonstrates that MPI is indeed a higher level interface, and shows why MPI implementations tend to be shorter. However, it should be taken into account that the PThreads library is relatively small compared with MPI, and just because the MPI source code is shorter does not mean that the binaries are smaller when linked statically. But the MPI binaries also have to do more - communication between separate systems requires error handling and other features that need not be considered in a shared memory system - though these would not be required for an MPI implementation designed specifically for a shared memory system.

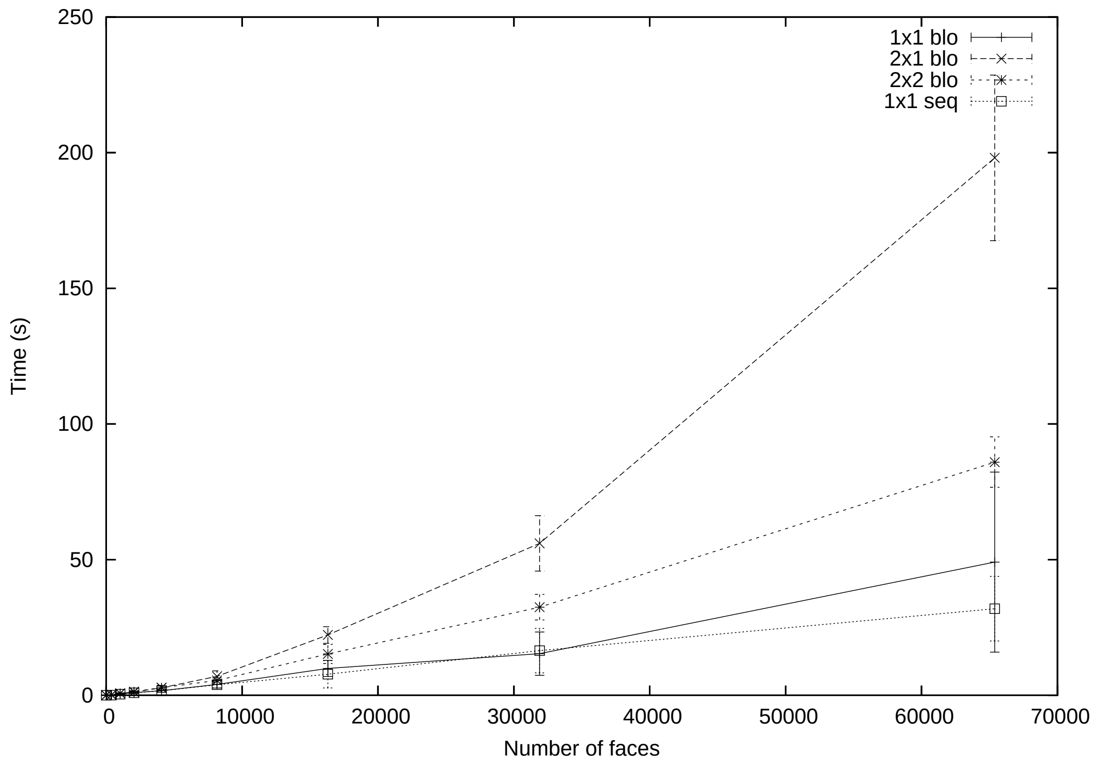

Figure 5.13: Algorithm time vs number of faces, PThread Dijkstra and Sequential Dijkstra on 4-way SMP

Both implementations were significantly longer than the sequential Dijkstra implementation. This shows the overhead involved in writing parallel applications and demonstrates that, for example, threading should not be used where it can be avoided.

Table 5.2 shows an estimate of implementation and debugging time for the different implementations. Values for the sequential Dijkstra implementation are not shown because its development was inextricably linked with the development of the core library code, which meant its implementation took much longer than the others.

The differences here are partially explained by the chronological order of development. The first parallel implementation attempted was the MPI Dijkstra implementation, and this required a significant amount of thought about the design of the parallelisation which meant the process took longer. The MPI Lanthier implementation was carried out next, and again there was some design work required in order to implement the algorithm. The PThread Dijkstra implementation with the single queue was the next attempted, and as the single queue operation required a different strategy from that employed with multiple queues, again the design and implementation time was significant. This implementation also had the most complex locking strategies and issues with the implementation of those meant that a significant amount of time was spent debugging. The PThread Dijkstra with multiple queues was implemented after this, after experience had been gained both with PThreads (with the single queue implementation) and with implementing a multiple queue Dijkstra algorithm (with the MPI Dijkstra implementation), and this was a relatively quick and painless process. Finally, around 2 days worth of design work was

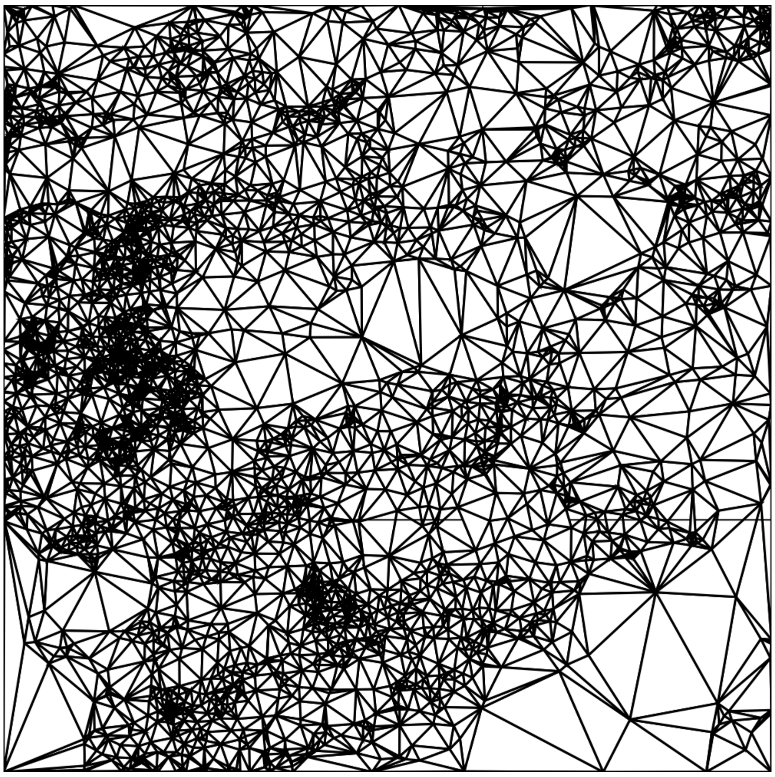

Figure 5.14: TIN data: nn_16314

Implementation	SLOCs
Sequential Dijkstra	124
PThread Dijkstra	250
PThread Dijkstra (including barrier)	316
PThread Dijkstra (single queue)	341
MPI Dijkstra	214
MPI Lanthier	289

Table 5.1: SLOCs for the algorithm implementations

carried out for the PThread Lanthier algorithm. Rather than a parallelisation from scratch, this involved taking an algorithm designed for message passing and trying to convert it to work in the shared memory style of programming, which meant that even though quite some time was spent on it, only a small amount of progress was made. The requirement for waiting until variables changed state meant that condition variables were required which made the design more complex than that required for the Dijkstra algorithm.

Debugging took longer with shared memory because the bugs are so hard to track down. Without careful design, one thread can write to data owned by another thread and cause problems. Additionally, the ordering of concurrent threads' operations can vary depending on exactly how they are scheduled, which can make problems very hard to reproduce. However, threaded applications can be more easily debugged using a standard debugger such as the GNU Debugger which supports threads. In contrast, though there is some support for debuggers in MPI, it is mostly a case of adding a lot of debugging print

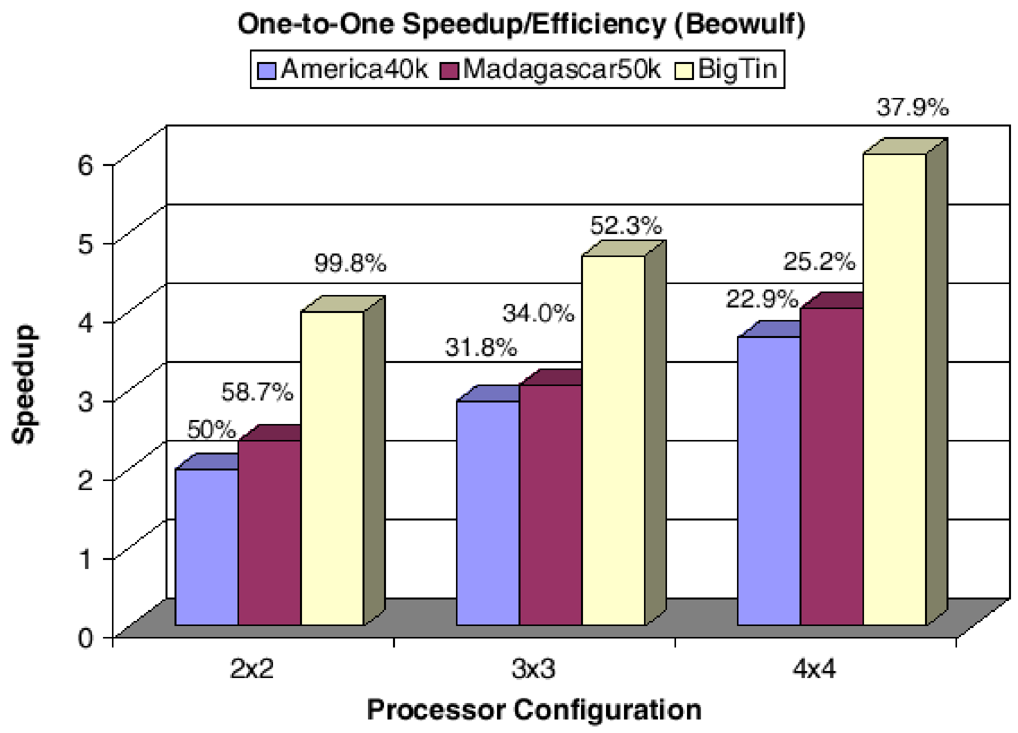

Figure 5.15: Speedup and efficiency of MPI Lanthier on a Beowulf system with different processor configurations and data sets (Lanthier et al.)

Implementation	Design/Implementation Time (days)	Debugging Time (days)
PThread Dijkstra	1	0
PThread Dijkstra (single queue)	7	4
PThread Lanthier (incomplete)	2+	-
MPI Dijkstra	8	2
MPI Lanthier	7	2

Table 5.2: Design, implementation and debugging time for the algorithm implementations

statements to programs in order to determine where it crashes or has problems. Another valuable tool that was used in the debugging of the PThreads implementation was the Valgrind memory checker - it could check to see when memory was double freed or a memory address was overwritten when it should not have been - and it scheduled threads based on a strict time-slice policy, which meant results were more reproducible and subtle bugs in the locking more likely to occur than on a real system. But even though there were good tools for use with PThreads, the subtlety of the bugs meant debugging required more time.

The knowledge gained during the development and subsequent debugging of the single queue PThread Dijkstra algorithm was that the key to successful program development with PThreads is to have a clear design, including a per-thread data ownership strategy and clear locking mechanism. In contrast, MPI much more forgiving of coding without a clear design as it enforces a rigid structure on process interaction.

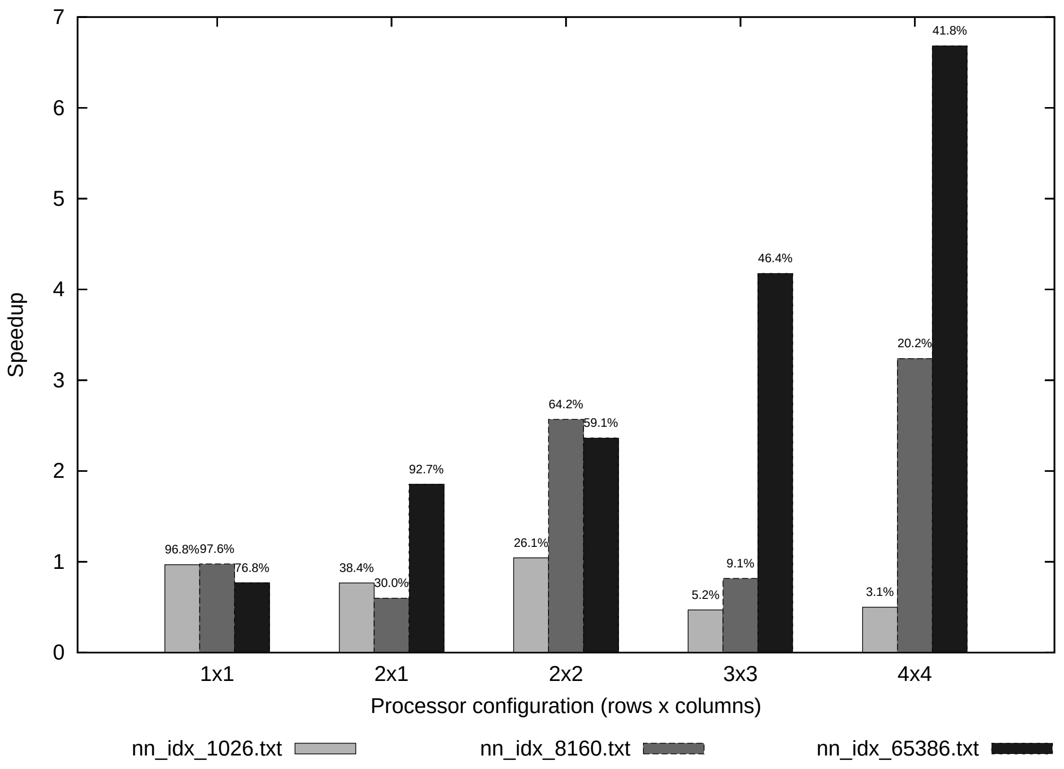

Figure 5.16: Speedup and efficiency of MPI Lanthier on the 16-node Beowulf with different processor configurations and data sets

5.3.2 Mental complexity

There is a difference in how easy it is to think about shared memory and message passing systems. In a message passing system, memory is not shared and so each processor is effectively self-contained. The interactions are clearly defined by the messages being passed between the processors. This means when thinking about the problem, each process can be thought of as its internal functioning plus its interface, and the group of processes do not generally have to be considered as a single entity.

In contrast, with a shared memory system, the processes share memory and are inextricably linked by accessing the same data. The interactions between the processes must be clearly defined by the programmer in order to achieve a correct program, but this is not enforced by the memory model or any other entity. Although there is a clear delineation between global storage and thread-local storage, there is no protection of this distinction in standard systems. It is therefore made all too easy to introduce bugs, as mentioned in Section 5.3.1, but it is also harder to think about the system because the system must be considered as a whole rather than merely considering the separate processes.

To some degree, these notions of how difficult the problems are to think about are defined by the programmer's previous experience. However, it does appear that message passing programming styles offer a simpler mental model of a system than shared memory programming styles.

Chapter 6

Conclusions

6.1 Performance comparisons

The results found allow the following conclusions to be drawn:

- Multiple unsorted queues perform better than multiple sorted queues with the Dijkstra and Lanthier algorithms, at least when a message passing programming style is used, though this is likely to be true for shared memory as well.
- Performing early termination for the Dijkstra algorithm by checking the target vector to see if it has been reached only results in a performance improvement of between 1.3 and 1.6 times, rather than the 2 times that might be expected in theory, because of the additional vector equality tests required.
- The use of MFP partitioning with the Lanthier algorithm gives significantly faster execution than a simple block partitioning. With the Dijkstra algorithm the increased number of vertices causes slower execution with MFP than with the block partitioning, particularly on systems where communication is slow and as the number of processors increases. However, the time taken to partition, though not examined as part of this project, was significant with the recursive MFP partitioning and should be taken into account when considering partitioning schemes. In particular, further investigation of recursion depth versus preprocessing time would be worthwhile.
- The optimum number of processors for the Lanthier algorithm on a message passing system varies depending on the data set size. This is particularly noticeable when dealing with small data sets on a message passing architecture.
- On both systems tested, the Lanthier algorithm gives improved results due to its asynchronous nature. As the simple Dijkstra parallelisation used was synchronous, its performance was significantly worse on the message passing system than on the shared memory system.
- The Lanthier algorithm implemented compares well with the results quoted by Lanthier et al. [12].

- On a shared memory system, the shared memory style gives slightly better performance than the message passing style, due to the reduced overhead and avoidance of the need for copying of data. Both styles appeared to scale similarly with a similarly implemented algorithm as the size of the data set increased. However, conclusions about scaling with the number of processors are not possible due to the loading of the system being used for performance testing.

It is interesting to note that the methodology of Chandra et al. [4] used simulators rather than actual systems for their comparisons between architectures. Using a simulator of a shared memory machine would have had some benefit here. Firstly, it would allow for a much more in depth look at exactly what caused inefficiencies in algorithm implementations, rather than the intuition and guesswork that is largely required in experimental analysis without the use of a simulator. Secondly, it would allow for simulations to be done, albeit slowly, on a uniprocessor system or a heavily loaded system without affecting the experimental results, which was a major problem in this project. However, there is no guarantee that the results gleaned on simulators will translate to real implementations. New tools such as the cache coherence plugin of the Valgrind program supervision framework [27], though not yet mature enough for use in this project, should at least give the first benefit by allow actual program binaries to be checked for issues with cache coherence through near real-time simulation of the local processors, and allow the performance bottlenecks of shared memory programs to be found and improved relatively easily.

6.2 Complexity comparisons

As has been seen, the message passing Dijkstra implementation had fewer lines of code than the equivalent shared memory implementation. The message passing implementation took significantly longer to create and debug, but as both were equivalent, the knowledge gained in creating the message passing implementation was used in creating the shared memory implementation. It seems likely that the message passing system would have been equally fast to create had the shared memory implementation been carried out first.

The initial single-queue shared memory implementation had significantly more lines of code and took much longer to code and debug, but this was a harder problem, and was the first threaded application created here.

As stated in the OpenMP API proposal [10], the MPI interface is at the right level for scientific application programming. It vastly simplifies the programming of message passing applications, compared with, for example, using UDP packets or TCP sockets. It was perhaps unfair to compare it with PThreads for shared memory, as PThreads is such a low-level API. The original reasons for not using OpenMP were valid ones (lack of low level control), but perhaps a more fair comparison against MPI would be with OpenMP - an interface designed with the scientific application programmer in mind. In fact, at least one performance comparison has been made between MPI and OpenMP programming styles [28] which found that OpenMP requires a greater programming effort than MPI and only occasionally gives greater performance, with MPI being faster in some cases.

It does not seem possible to draw a conclusion about the programming complexity of message passing versus shared memory programming styles solely on the basis of the APIs used here. Although it was found that message passing was simpler, this conclusion was inextricably linked with the APIs used and would vary had different ones been chosen. However, the results here in conjunction with the previous work would seem to imply that message passing does indeed have a greater ease of programming than the shared memory style.

It also seems that message passing provides a simpler mental model of the task at hand compared with the shared memory programming style.

There are therefore four reasons why a message passing style should be preferred over a shared memory style:

1. Message passing offers a simpler mental model of a system.
2. The message passing style is easier to program for than the shared memory style.
3. There is little performance degradation associated with using a message passing programming style on shared memory systems, both with the algorithms tested here and in previous work (see Section 1.1).
4. Message passing is more flexible as it also allows implementations to be executed on message passing architectures, which are becoming more common in the form of cluster and grid computing, rather than requiring a high powered single machine as is needed for shared memory implementations.

6.3 Project achievements

In the following sections, the project's modified goals (as defined in Section 1.4) are examined to see which of them were achieved.

6.3.1 Goal 1

> Implement a system capable of allowing the practical performance of geometric shortest path algorithms with shared memory and message passing parallel programming styles to be compared on a shared memory machine.

This goal has been met almost in its entirety. A correct parallel Dijkstra algorithm implementation was completed using both programming styles, which alone is suitable for comparing the performance. In addition, an implementation was completed in the message passing style and some design work done in the shared memory style using the Lanthier algorithm suitable for acting as another basis for comparison. If a suitable shared memory system had been available in order to allow the original goal to be met, it is likely that the shared memory style Lanthier implementation would have been completed as the additional investigative goals would not have been added.

6.3.2 Goal 2

> Test such a system on any available machine and, if possible, try to draw some conclusions about performance.

Experiments aimed at meeting this have been carried out, though the issues with contention with other jobs on the machine used meant that the results were unreliable (see Section 5.2 for the relevant results and analysis). It was therefore not possible to draw firm conclusions about performance. Had a suitable shared memory system been available, there would have been no reason this goal could not have been met given the implementation available.

6.3.3 Goal 3

> Examine the performance of the message passing algorithms on an available message passing system (Beowulf cluster).

Experiments were carried out on the 16-node Beowulf in order to meet this goal (see Section 5.2.4 for the results). The performance of each of the algorithms was examined, and the Lanthier algorithm implementation was compared against that of Lanthier et al. [12]. This goal was met in full.

6.3.4 Goal 4

> Determine the impact of using an unsorted queue, compared with using a sorted queue in the algorithm implementations.

Testing was carried out with the algorithms using both an unsorted and sorted queue, and the results were presented in Section 5.2.1. Unfortunately the PThread Dijkstra implementations using the sorted and unsorted queues were not comparable due to the change from single to multiple queues. With this exception, the goal was successfully met.

6.3.5 Goal 5

> Compare block vs. MFP partitioning for the different algorithms.

These partitioning methods were compared for use with the different algorithms, and the results presented in Section 5.2.3. Once again, this goal was met in full.

6.3.6 Goal 7

> Compare the ease of programming of geometric shortest path algorithms using both programming styles, and try to draw some general conclusions.

This was analysed and the results presented in Section 5.3, and this goal was completed as well as it could be. Judgements have been made about the ease of programming of each programming styles, through both objective and necessarily subjective means. However, this will doubtless remain an area of contention, as those with different backgrounds and experience will find one system easier than the other.

6.4 Suggestions for further work

Some questions remain unanswered by this project, and the following would be useful topics for further research:

- Doing experimental runs with different limits on the recursion levels of the MFP partitioning. A limit of 5 levels of recursion and a minimum cut-off of 7 vertices per partition was used, but a limit of 2 or 3 levels may have been more appropriate, as suggested by Lanthier et al. [12].
- Experimentation into different methods of splitting the terrain between processors for the Lanthier algorithm.
- Analysis of the partitioning preprocessing time, to determine if this affects the results, and an investigation into possible parallelisation of the partitioning.
- Completion of the PThread Lanthier implementation. This would complete the set of tests and provide a useful additional benchmark of comparison.
- Optimisation of the implementations. Only limited optimisation has been done on the algorithm implementations. Fully optimised implementations, particularly the use of a Fibonacci heap for the queues, and avoiding cache coherency issues with PThreads, is an area that allows for much further work.
- Redoing of shared memory benchmarks with exclusive use of machines. As mentioned, the shared memory benchmarks here were carried out on machines with shared access that were heavily loaded at the time. This was unavoidable due to the extended SunFire downtime.
- Scaling up the number of processors further. This would be of interest, to see if the observed trends continue with more processors, and would have been carried out had the SunFire been available.
- Scaling up the size of the datasets further. Most of the experiment runs were quite short, with the data set sizes being set based on initial MPI Dijkstra runs which were very slow. Further experiments with the Lanthier algorithm in particular using larger data sets would be interesting to see how the scaling continues.
- Examination of other algorithms. As noted elsewhere, research has been carried out on shared memory vs message passing for some other algorithms, but many have not been researched, both in the area of geometric shortest path finding, and others.

6.5 Final remarks

This project has had its share of setbacks, with the unavailability of the SunFire machine making it difficult to proceed. However, the goals were modified slightly to take account of this and some useful results have been obtained which compare well with previous work in the area. As well as the implementation and experimental results that have been produced during this project, some areas of the theory appear to be novel - in particular, no previous work was found that specifically examined the differences in performance or programming ease between the shared memory and message passing programming styles for shortest path algorithms as has been done here.

In conclusion, although some questions remain unanswered, overall the project has been a successful one, both in generating results and as a valuable learning experience.

Bibliography

[1] R Fujimoto and S Ferenci. RTI performance on shared memory and message passing architectures. *1999 Spring Simulation Interoperability Workshop*, March 1999.

[2] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15 (1):29–36, 1995.

[3] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Performance issues of the parallel PAM-CRASH code. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):3–11, Spring 1997.

[4] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 61–73. ACM Press, 1994.

[5] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 186–197. ACM Press, 1996.

[6] Robert A. Fiedler. Optimization and scaling of shared-memory and message-passing implementations of the zeus hydrodynamics algorithm. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–16. ACM Press, 1997. ISBN 0-89791-985-8.

[7] Rishiyur S. Nikhil. Cid: A parallel, “shared-memory” C for distributed-memory machines. In *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 376–390.

[8] Holger Karl. Bridging the gap between distributed shared memory and message passing. *Concurrency: Practice and Experience*, 10(11–13):887–900, September 1998.

[9] MPI Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall/Winter 1994.

[10] OpenMP Forum. OpenMP: A proposed industry standard API for shared memory programming. *http://www.openmp.org/ Technical Report*, October 1997. URL `http://www.math.utah.edu/~beebe/openmp/doc/paper.pdf`.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271, 1959.

[12] Mark Lanthier, Doron Nussbaum, and Jörg-Rüdiger Sack. Parallel implementation of geometric shortest path algorithms. *Parallel Computing*, 29(10):1445–1479, October 2003.

[13] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[14] D. P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *J. Optim. Theory Appl.*, 88(2):297–320, 1996.

[15] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Trans. Graph.*, 4(2):74–123, 1985.

[16] Mark Lanthier, Anil Maheshwari, and Jörg-Rüdiger Sack. Approximating weighted shortest paths on polyhedral surfaces. In *6th Annual Video Review of Computational Geometry, Proc. 13th ACM Symp. Computational Geometry*, pages 485–486. ACM Press, 4–6 1997.

[17] P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest-path problem. *International Journal of Parallel Programming*, 20(4):271–298, August 1991.

[18] SCons: a software construction tool, 2004. URL `http://www.scons.org/`.

[19] Subversion: version control system. URL `http://subversion.tigris.org/`.

[20] Landmap - britain from space. URL `http://www.landmap.ac.uk/`.

[21] Jo Wood. Landserf: visualisation and analysis of surfaces. URL `http://www.soi.city.ac.uk/~jwo/landserf/`.

[22] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[23] GNU diffutils. URL `http://www.gnu.org/software/diffutils/diffutils.html`

[24] David A Wheeler. SLOCcount user's guide, 2004. URL `http://www.dwheeler.com/sloccount/sloccount.html`

[25] J. B. Dreger. *Function point analysis*. Prentice-Hall, Inc., 1989.

[26] Jesper Larsson Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21(9):1505–1532, September 12, 1995.

[27] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[28] Géraud Krawezik. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127. ACM Press, 2003.

Appendix A

Data Sets

Table A.1 gives a list of the data sets used and refers to figures showing the data sets. The data sets are shown by black lines along the terrain edges, looking down onto the X-Y plane. The Z values are not shown, however, wherever the gradient is steepest, more faces will be generated by the TIN creation algorithm.

Name	No. Faces	Source	Figure
test2	2	Hand created	A.1
ls_samp	392	Landserf example	A.2
nn_tin_1026	1026	Landmap	A.3
nn_tin_2045	2045	Landmap	A.4
nn_tin_4077	4077	Landmap	A.5
nn_tin_8160	8160	Landmap	A.6
nn_tin_16314	16314	Landmap	A.7
nn_tin_31902	31902	Landmap	A.8
nn_tin_65386	65386	Landmap	A.9

Table A.1: TIN data used

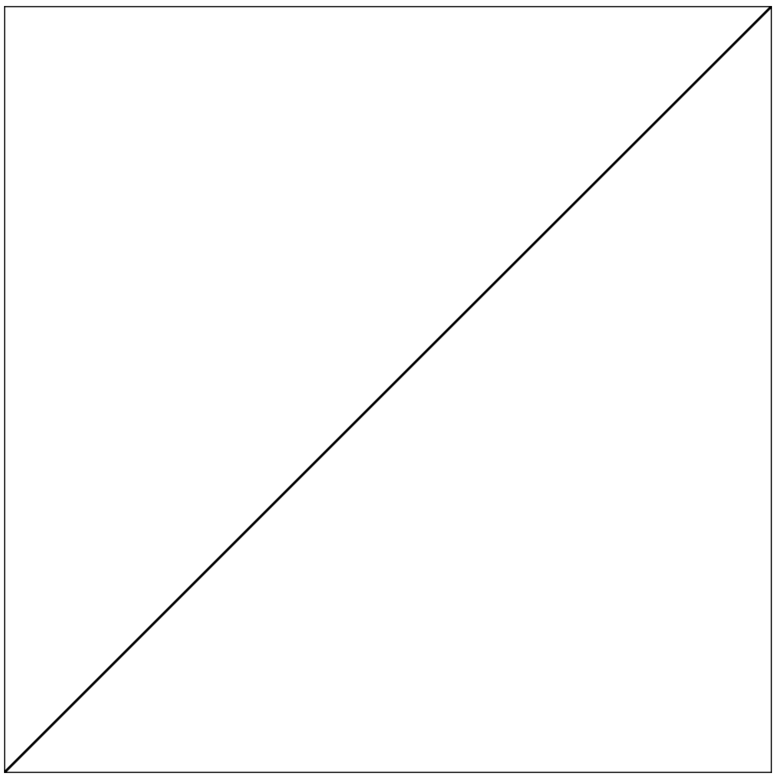

Figure A.1: TIN data: test2

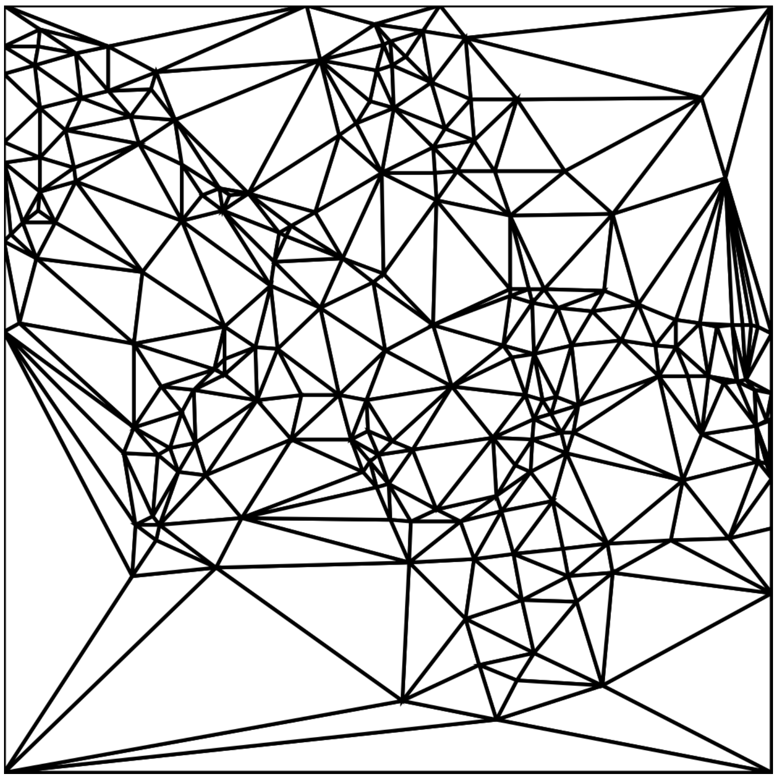

Figure A.2: TIN data: ls_samp

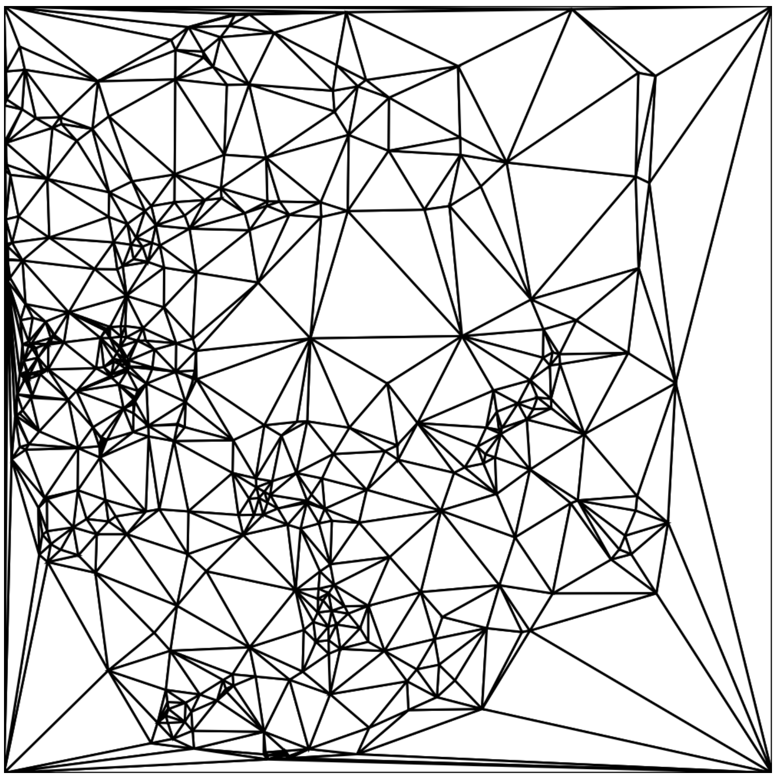

Figure A.3: TIN data: nn_1026

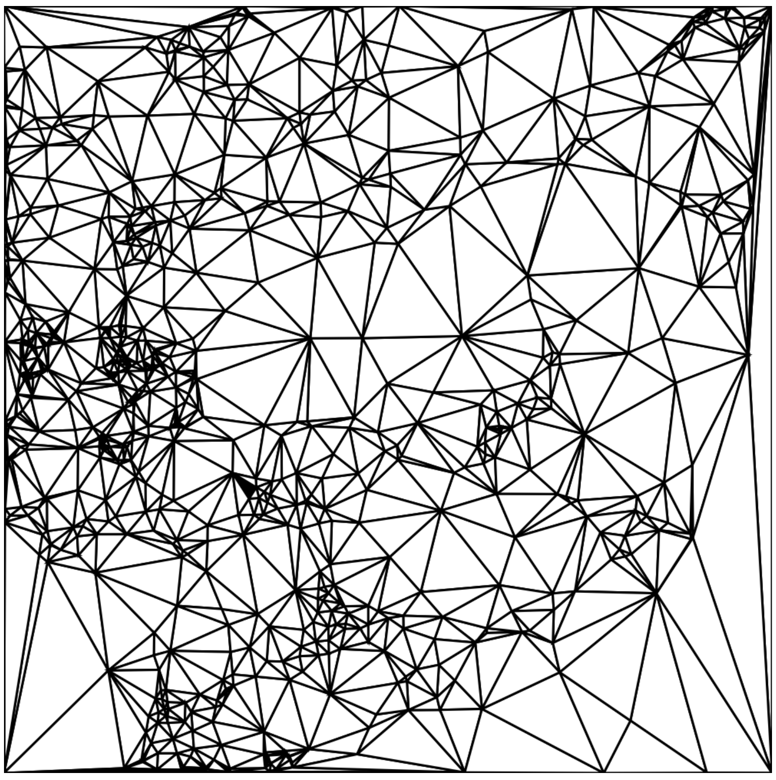

Figure A.4: TIN data: nn_2045

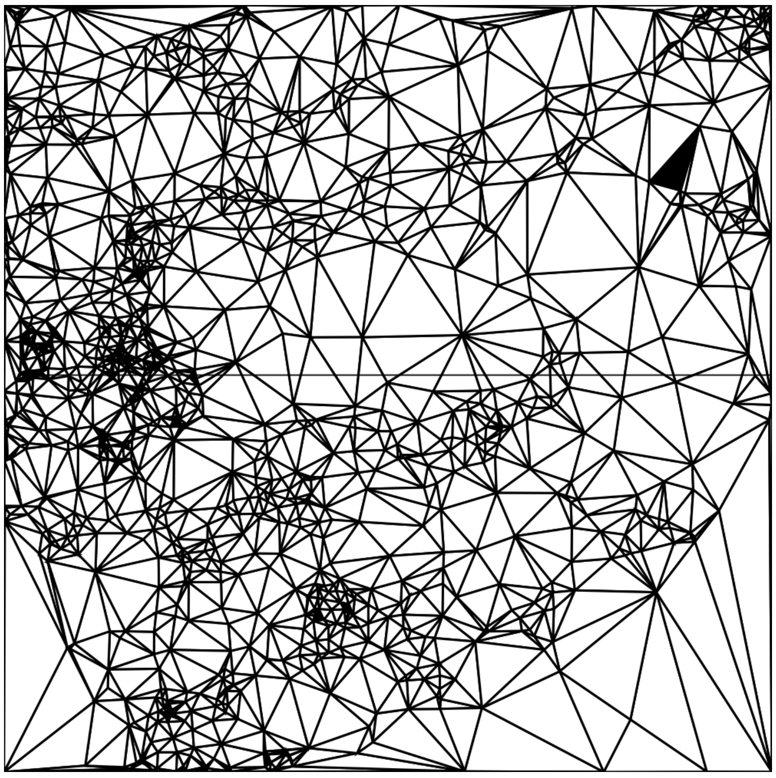
Figure A.5: TIN data: nn_4077

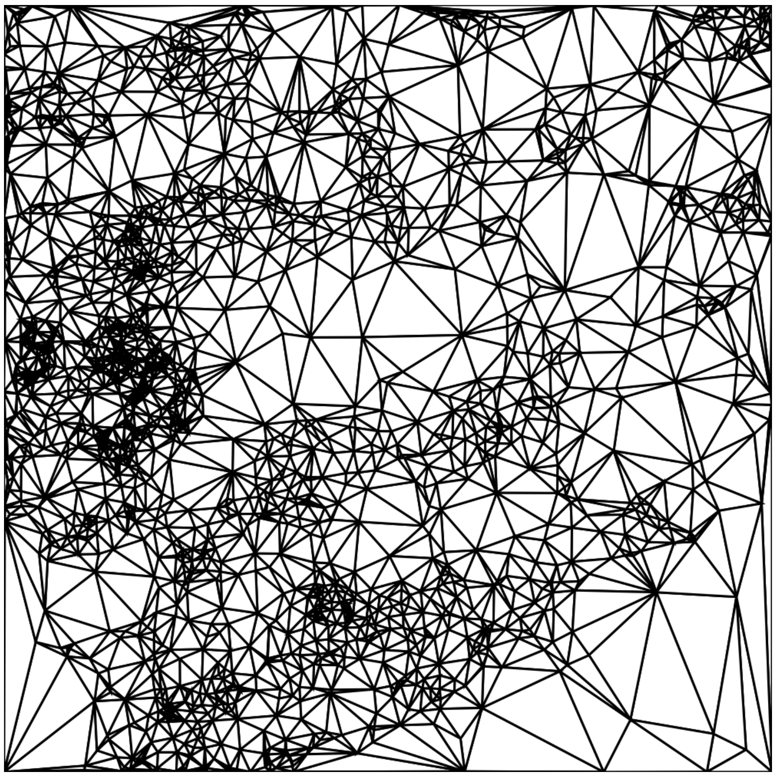
Figure A.6: TIN data: nn_8160

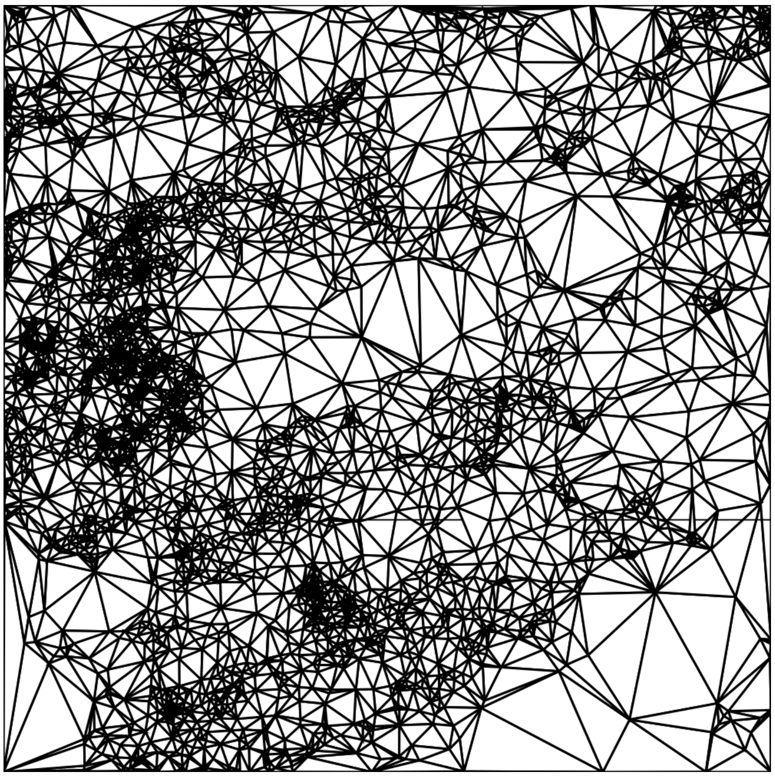

Figure A.7: TIN data: nn_16314

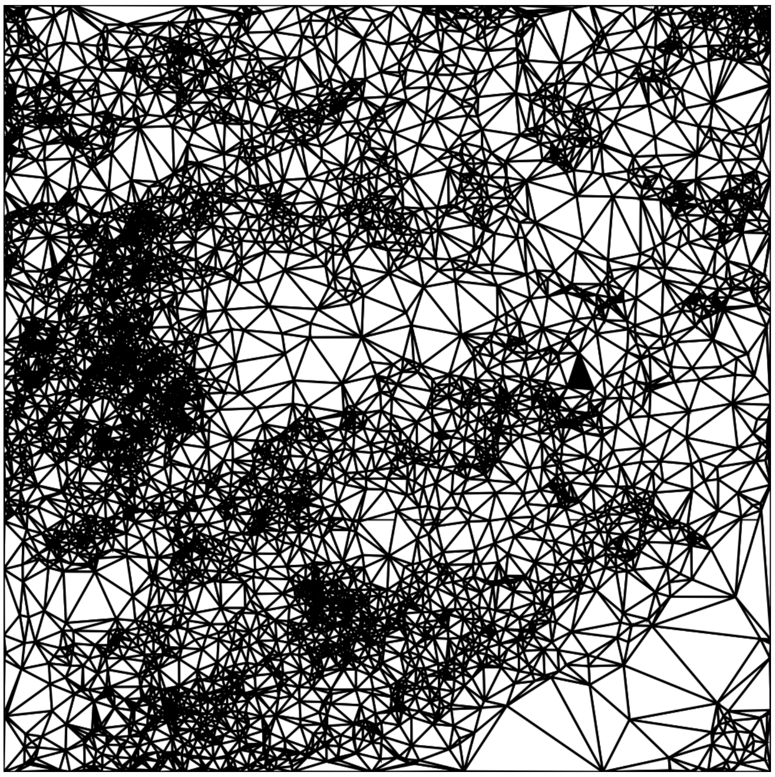

Figure A.8: TIN data: nn_31902

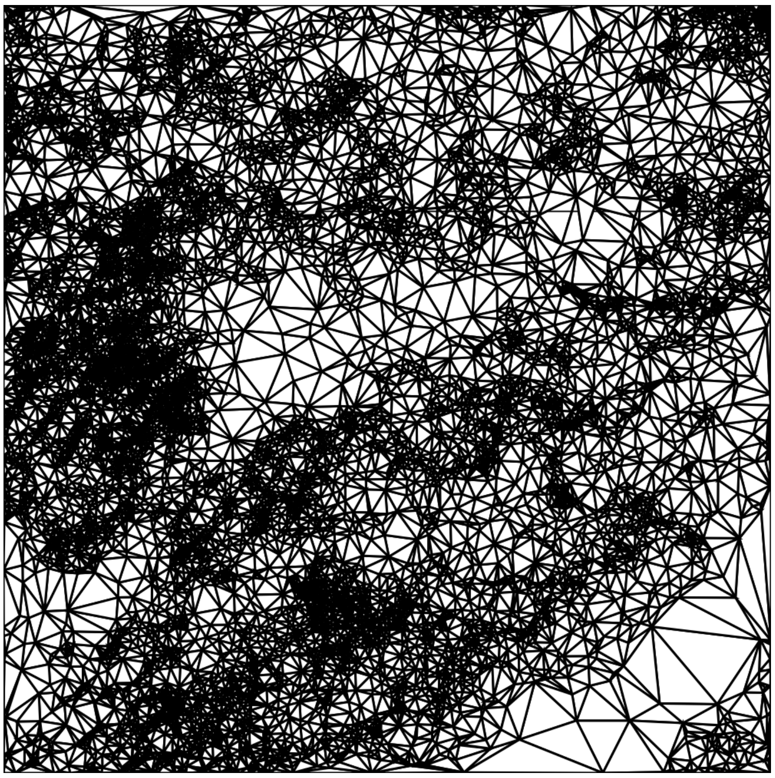
Figure A.9: TIN data: nn_65386

Appendix B

Selected Code

This appendix gives the top-level algorithm implementations, but not the core library code which is used. The complete code can be found in the */home/v1aphipp/PROJECT/src* directory on DICE.

B.1 Message passing Dijkstra implementation

```
/* test_mpidijkstra
 * Copyright (C)2004 Alistair K Phipps.  gspf@alistairphipps.com.
 *
 * Test program that does MPI distributed memory parallel dijkstra
 */

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <mpi.h>
#include "../core/log.h"
#include "../core/timer.h"
#include "../core/exception.h"
#include "../core/geometry.h"
#include "../core/gvertex.h"
#include "../core/face.h"
#include "../core/edge.h"
#include "../core/queue.h"
#include "../core/utility.h"

using namespace std;

int main( int argc, char * argv[] )
{
  try
  {
    Timer t;
```

```
t.reset();
int rank, size;
unsigned int nx, ny;  // number of rows / cols in MFP split
MPI::Init( argc, argv );
rank = MPI::COMM_WORLD.Get_rank();
size = MPI::COMM_WORLD.Get_size();
if( argc != 7 )
  throw Exception( "6 args required: tininputprefix.txt numcols numrows sourcex
      sourcey sourcez" );
nx = Utility::strToUI( argv[2] );
ny = Utility::strToUI( argv[3] );
Vector3F vSource;
vSource._x = Utility::strToD( string( argv[4] ) );
vSource._y = Utility::strToD( string( argv[5] ) );
vSource._z = Utility::strToD( string( argv[6] ) );

if( nx * ny != size )
  throw Exception( "numcols * numrows does not match MPI world size" );
unsigned int cellx = rank % nx;
unsigned int celly = rank / nx;
LOG( "Started, rank: " << rank << " (cell " << cellx << ", " << celly << ") of "
    << size << endl );
ostringstream oss;
oss << argv[1] << "." << cellx << "." << celly;
ifstream ifTin( oss.str().c_str() );
if( !ifTin )
  throw Exception( oss.str() + " could not be opened for reading" );
LOG( rank << " reading geometry: " << oss.str() << endl );
Geometry g;
ifTin >> g;
ifTin.close();
LOG( rank << " read geometry: " << oss.str() << endl );
LOG( rank << " number of vertices: " << g._vList.size() << endl );
LOG( rank << " number of edges: " << g._eList.size() << endl );
LOG( rank << " number of faces: " << g._aList.size() << endl );
t.stop();
double timeLoad = t.getSecs();
t.reset();
// build geometry lookup tables
g.buildLists();
// initialise costs to "infinity" and settled to false
for( vector< Edge >::iterator i = g._eList.begin(); i != g._eList.end(); i++ )
{
  for( vector< double >::iterator j = i->_fCostList.begin(); j != i->_fCostList.
      end(); j++ )
  {
    *j = 999999999;
  }
  for( vector< bool >::iterator j = i->_bSettledList.begin(); j != i->
      _bSettledList.end(); j++ )
  {
    *j = false;
  }
```

```
}
LOGD( rank << " setting geometry" << endl );

t.stop();
double timeInit = t.getSecs();
t.reset();
// find source vertex
bool bFound = false;
GVertex *pv;
for( unsigned int i = 0; i < g._vList.size() && !bFound; i++ )
{
  if( g._vList[i] == vSource )
  {
    bFound = true;
    pv = new GVertexV( i, &g );
    pv->setCost( 0.0 );
  }
}
t.stop();
double timeFindSrc = t.getSecs();
t.reset();

Queue< GVertex* > q; // queue of unsettled verts
// put source vertex into queue
// - note this call gets done by all threads that share the vertex, but it doesn'
    t matter as one will win the reduction
if( bFound )
{
  q.push( pv );
}

LOGD( rank << " entering queue loop" << endl );
struct
{
  double val;
  int rank;
} us, ur; // send and receive
static Vector2I *v2b = new Vector2I[size];  // send buffer
static Vector2I v2r; // receive buffer

Timer t2;
double timeScatter = 0.0;
double timeReduce = 0.0;
while( 1 )
{
  // extract element with smallest cost
  GVertex *pq = NULL;
  if( !q.empty() )
  {
    LOGD( rank << " reducing with element from queue" << endl );
    pq = q.top();
    us.val = pq->getCost();
  }
```

```
else
{
  LOGD( rank << " reducing with dummy element" << endl );
  us.val = 999999999;
}
us.rank = rank;
// do global reduction to determine element with global smallest cost
LOGD( rank << " performing AllReduce" << endl );
t2.reset();
MPI::COMM_WORLD.Allreduce( &us, &ur, 1, MPI::DOUBLE_INT, MPI::MINLOC );
t2.stop();
timeReduce += t2.getSecs();
LOGD( rank << " completed AllReduce" << endl );

// if ur.val = 999999999, everyone is finished
if( ur.val == 999999999 )
  break;

// if we won...
if( ur.rank == rank )
{
  LOGD( rank << " sending in scatter" << endl );
  // fill buffer with edge/sp (ie,is) or vertex (iv,3000000) or nothing
      (3000000,3000000), one sent to each cell...
  // fill send buffer with dummy values
  for( unsigned int i = 0; i < size; i++ )
  {
    v2b[i] = Vector2I( 3000000, 3000000 );
  }
  // if it's a (terrain) vertex...
  if( pq->getType() == GVertex::GVERTEX_V )
  {
    // find all (cellid, vertexid) for cells that share this vertex
    vector< Vector3I > v3CellList = g._vSharedList[pq->asVector2I()._x];
    for( unsigned int i = 0; i < v3CellList.size(); i++ )
    {
      v2b[v3CellList[i]._x + v3CellList[i]._y * nx] = Vector2I( v3CellList[i].
          _z, 3000000 );
    }
    // set up to send to ourself
    v2b[rank] = pq->asVector2I();
  }
  // if it's on an edge...
  if( pq->getType() == GVertex::GVERTEX_E )
  {
    // find all (cellid, edgeid) for cells that share this edge
    vector< Vector3I > v3CellList = g._eSharedList[pq->asVector2I()._x];
    for( unsigned int i = 0; i < v3CellList.size(); i++ )
    {
      v2b[v3CellList[i]._x + v3CellList[i]._y * nx] = Vector2I( v3CellList[i].
          _z, pq->asVector2I()._y );
    }
    // set up to send to ourself
```

```
        v2b[rank] = pq->asVector2I();
      }
    }
    // do scatter so winner sends vertex or edge info to nodes that share it
    t2.reset();
    MPI::COMM_WORLD.Scatter( v2b, 1, MPI::TWOINT, &v2r, 1, MPI::TWOINT, ur.rank );
    t2.stop();
    timeScatter += t2.getSecs();
    // all nodes must now check receive buffer to see if they received a vertex or
        edge
    if( v2r._x == 3000000 && v2r._y == 3000000 )
    {
      LOGD( rank << " received dummy in scatter" << endl );
    }
    else
    {
      GVertex *pu;
      if( v2r._y == 3000000 )
      {
        LOGD( rank << " received vertex (id " << v2r._x << ") in scatter" << endl )
            ;
        pu = new GVertexV( v2r._x, &g );
      }
      else
      {
        LOGD( rank << " received edge (id " << v2r._x << ", sp " << v2r._y << ") in
             scatter" << endl );
        pu = new GVertexE( v2r._x, v2r._y, &g );
      }
      // if we received the same vertex that is at the top of our queue, remove it
      if( !q.empty() && ( *pu == *pq ) )
      {
        q.pop();
        delete pq;
      }
      // make vertex settled
      pu->setSettled( true );
      // set cost of vertex
      pu->setCost( ur.val );

      // relax non-settled vertices...
      // for each vertex v adjacent to u and not in s, if current distance to v is
          > distance to u + distance from u to v, replace current distance to v
          with distance to u + distance from u to v and add v to q if it's only now
           been reached
      vector< GVertex* > adjacent = pu->getAdjacent();
      LOGD( rank << " Element with smallest cost: " );
      LOGD( rank << "       " << pu->asVector2I() << ", cost: " << pu->getCost() <<
           endl );
      for( vector< GVertex* >::iterator ppv = adjacent.begin(); ppv != adjacent.end
          (); ppv++ )
      {
        LOGD( rank << "     adjacent element: " );
        LOGD( rank << "       " << (*ppv)->asVector2I() << ", cost: " << (*ppv)->
            getCost() << endl );
        if( !(*ppv)->getSettled() ) // if not settled
        {
          LOGD( rank << " v: " << (*ppv)->getPosition()._x << "," << (*ppv)->
              getPosition()._y << "," << (*ppv)->getPosition()._z << "\n" );
          LOGD( rank << " u: " << pu->getPosition()._x << "," << pu->getPosition().
              _y << "," << pu->getPosition()._z << "\n" );
          // face weight we multiply by is:
          // - for edge-joined SPs, the weight of the lowest-weighted face that is
              coincident with the edge
          // - for cross-face joined SPs, the weight of the face crossed
          double fW = pu->getFaceWeightTo( **ppv );
          double duv = ~((*ppv)->getPosition() - pu->getPosition()) * fW;
          if( (*ppv)->getCost() > pu->getCost() + duv )
          {
            // set cost to new value via u
            if( (*ppv)->getCost() == 999999999 )  // reached for first time, add to
                 Q
            {
              LOGD( rank << " Pushing vertex onto Q" << std::endl );
              (*ppv)->setCost( pu->getCost() + duv );
              q.push( *ppv );
            }
            else  // already in Q, but need to re-order vertex according to new
                cost
            {
              LOGD( rank << " *CHANGING* cost of vertex on Q" << std::endl );
              (*ppv)->setCost( pu->getCost() + duv );
              q.change( *ppv );
            }
            LOGD( rank << "        cost updated to: " << (*ppv)->getCost() << endl
                );
          }
          else
          {
            // don't need ppv anymore as we didn't add it to the q
            delete *ppv;
          }
        }
        else
        {
          // don't need ppv anymore as we didn't add it to the q
          delete *ppv;
        }
      }
      // we're done with pu
      delete pu;
    }
  }
  t.stop();
  double timeAlgo = t.getSecs();
  t.reset();
```

```
    LOG( rank << ": Final list...\n" );
    for( unsigned int i = 0; i < g._eList.size(); i++ )
    {
      for( unsigned int j = 0; j < g._eList[i]._fCostList.size(); j++ )
      {
        LOG( rank << ": SP... edge index: " << i << ", SP index: " << j << ", cost: "
             << g._eList[i]._fCostList[j] << ", settled: " << g._eList[i].
            _bSettledList[j] << endl );
      }
    }
    MPI::Finalize();
    t.stop();
    double timeFinal = t.getSecs();
    LOG( rank << ": Loading time: " << timeLoad << endl );
    LOG( rank << ": Structure initialisation time: " << timeInit << endl );
    LOG( rank << ": Time to find source vertex: " << timeFindSrc << endl );
    LOG( rank << ": Algorithm time: " << timeAlgo << endl );
    LOG( rank << ": ... of which carrying out reduction: " << timeReduce << endl );
    LOG( rank << ": ... of which carrying out scatter: " << timeScatter << endl );
    LOG( rank << ": Finalisation time: " << timeFinal << endl );
    LOG( rank << ": Total processing time (structure init + algorithm): " << timeInit
         + timeAlgo << endl );
  }
  catch( Exception& e )
  {
    LOG( e.what() << endl );
  }
  return 0;
}
```

B.2 Shared memory Dijkstra implementation

```
/* test_ptdijkstra
 * Copyright (C)2004 Alistair K Phipps.  gspf@alistairphipps.com.
 *
 * Test program that does PThread shared memory parallel dijkstra
 */

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <pthread.h>
#include "../core/log.h"
#include "../core/timer.h"
#include "../core/exception.h"
#include "../core/geometry.h"
#include "../core/gvertex.h"
#include "../core/face.h"
#include "../core/edge.h"
#include "../core/queue.h"
```

```
#include "../core/utility.h"
#include "barrier.h"

using namespace std;

// globals
unsigned int g_nx, g_ny;
string g_prefix;  // partition filename prefix
Vector3F g_vSource; // source vertex position
pthread_mutex_t g_mutReduce;
barrier_t g_barReduce1;
barrier_t g_barReduce2;
barrier_t g_barReduce3;
GVertex *g_pv;

struct TParam
{
  int rank;   //! thread ID number
};

void* thread( void *threadid )
{
  try
  {
    Timer t;
    t.reset();
    TParam *tp = static_cast< TParam* >( threadid );
    int rank = tp->rank;
    unsigned int cellx = rank % g_nx;
    unsigned int celly = rank / g_nx;
    LOG( "Started, rank: " << rank << " (cell " << cellx << ", " << celly << ") of "
        << g_nx * g_ny << endl );
    ostringstream oss;
    oss << g_prefix << "." << cellx << "." << celly;
    ifstream ifTin( oss.str().c_str() );
    if( !ifTin )
      throw Exception( oss.str() + " could not be opened for reading" );
    LOG( rank << " reading geometry: " << oss.str() << endl );
    Geometry g;
    ifTin >> g;
    ifTin.close();
    LOG( rank << " read geometry: " << oss.str() << endl );
    LOG( rank << " number of vertices: " << g._vList.size() << endl );
    LOG( rank << " number of edges: " << g._eList.size() << endl );
    LOG( rank << " number of faces: " << g._aList.size() << endl );
    t.stop();
    double timeLoad = t.getSecs();
    t.reset();
    // build geometry lookup tables
    g.buildLists();
    // initialise costs to "infinity" and settled to false
    for( vector< Edge >::iterator i = g._eList.begin(); i != g._eList.end(); i++ )
    {
```

```
  for( vector< double >::iterator j = i->_fCostList.begin(); j != i->_fCostList.
      end(); j++ )
  {
    *j = 999999999;
  }
  for( vector< bool >::iterator j = i->_bSettledList.begin(); j != i->
      _bSettledList.end(); j++ )
  {
    *j = false;
  }
}
LOGD( rank << " setting geometry" << endl );

t.stop();
double timeInit = t.getSecs();
t.reset();
// find source vertex
bool bFound = false;
GVertex *pv;
for( unsigned int i = 0; i < g._vList.size() && !bFound; i++ )
{
  if( g._vList[i] == g_vSource )
  {
    bFound = true;
    pv = new GVertexV( i, &g );
    pv->setCost( 0.0 );
  }
}
t.stop();
double timeFindSrc = t.getSecs();
t.reset();

Queue< GVertex* > q; // queue of unsettled verts
// put source vertex into queue
// - note this call gets done by all threads that share the vertex, but it doesn'
    t matter as one will win the reduction
if( bFound )
{
  q.push( pv );
}

LOGD( rank << " entering queue loop" << endl );

Timer t2;
double timeReduce = 0.0;
while( 1 )
{
  // extract element with smallest cost
  GVertex *pq = NULL;
  t2.reset();
  if( !q.empty() )
  {
    LOGD( rank << " reducing with element from queue" << endl );
    pq = q.top();
    // wait for everyone to get ready for reduction
    if( barrier_wait( &g_barReduce1 ) == -1 )
    {
      if( g_pv != NULL )
        delete g_pv;
      g_pv = NULL;
    }
    //FIXME: double barrier is ugly - think of a better way
    barrier_wait( &g_barReduce2 );
    if( g_pv == NULL || g_pv->getCost() > pq->getCost() )
    {
      pthread_mutex_lock( &g_mutReduce );
      if( g_pv == NULL || g_pv->getCost() > pq->getCost() )
      {
        g_pv = pq;
      }
      pthread_mutex_unlock( &g_mutReduce );
    }
  }
  else
  {
    // wait for everyone to get ready for reduction
    if( barrier_wait( &g_barReduce1 ) == -1 )
      g_pv = NULL;
    barrier_wait( &g_barReduce2 );
  }
  // wait for everyone to make their contribution
  barrier_wait( &g_barReduce3 );
  t2.stop();
  timeReduce += t2.getSecs();
  LOGD( rank << " completed AllReduce" << endl );

  // if g_pv is still NULL, everyone is finished - global queue empty
  if( g_pv == NULL )
    break;

  GVertex *pu = NULL;
  // if we won... (based on position rather than the pointer)
  if( (pq != NULL) && (*g_pv == *pq) )
  {
    // if it’s actually us that won (based on the pointer) then make a copy of it
         in pu, else we’ll delete it while it might still be needed
    if( g_pv == pq )
    {
      if( g_pv->getType() == GVertex::GVERTEX_V )
        pu = new GVertexV( pq->asVector2I()._x, pq->getGeometry() );
      if( g_pv->getType() == GVertex::GVERTEX_E )
        pu = new GVertexE( pq->asVector2I()._x, pq->asVector2I()._y, pq->
            getGeometry() );
    }
    else
      pu = pq;
    // remove it from our local queue
    q.pop();
  }
  else
  {
    // if it’s a (terrain) vertex...
    if( g_pv->getType() == GVertex::GVERTEX_V )
    {
      // find all (cellid, vertexid) for cells that share this vertex
      vector< Vector3I > &v3CellList = g_pv->getGeometry()->_vSharedList[g_pv->
          asVector2I()._x];
      for( unsigned int i = 0; i < v3CellList.size(); i++ )
      {
        // if we find ourselves
        if( v3CellList[i]._x == cellx && v3CellList[i]._y == celly )
        {
          pu = new GVertexV( v3CellList[i]._z, &g );
          break;
        }
      }
    }
    // if it’s on an edge...
    if( g_pv->getType() == GVertex::GVERTEX_E )
    {
      // find all (cellid, vertexid) for cells that share this edge
      vector< Vector3I > &v3CellList = g_pv->getGeometry()->_eSharedList[g_pv->
          asVector2I()._x];
      for( unsigned int i = 0; i < v3CellList.size(); i++ )
      {
        // if we find ourselves
        if( v3CellList[i]._x == cellx && v3CellList[i]._y == celly )
        {
          pu = new GVertexE( v3CellList[i]._z, g_pv->asVector2I()._y, &g );
          break;
        }
      }
    }
  }
  // if we shared the winning vertex...
  if( pu != NULL )
  {
    // make vertex settled
    pu->setSettled( true );
    // set cost of vertex
    pu->setCost( g_pv->getCost() );

    // relax non-settled vertices...
    // for each vertex v adjacent to u and not in s, if current distance to v is
        > distance to u + distance from u to v, replace current distance to v
        with distance to u + distance from u to v and add v to q if it’s only now
         been reached
    vector< GVertex* > adjacent = pu->getAdjacent();
    LOGD( rank << " Element with smallest cost: " );
```

```
LOGD( rank << "        " << pu->asVector2I() << ", cost: " << pu->getCost() <<
     endl );
for( vector< GVertex* >::iterator ppv = adjacent.begin(); ppv != adjacent.end
    (); ppv++ )
{
  LOGD( rank << "     adjacent element: " );
  LOGD( rank << "       " << (*ppv)->asVector2I() << ", cost: " << (*ppv)->
      getCost() << endl );
  if( !(*ppv)->getSettled() ) // if not settled
  {
    LOGD( rank << " v: " << (*ppv)->getPosition()._x << "," << (*ppv)->
        getPosition()._y << "," << (*ppv)->getPosition()._z << "\n" );
    LOGD( rank << " u: " << pu->getPosition()._x << "," << pu->getPosition().
        _y << "," << pu->getPosition()._z << "\n" );
    // face weight we multiply by is:
    // - for edge-joined SPs, the weight of the lowest-weighted face that is
        coincident with the edge
    // - for cross-face joined SPs, the weight of the face crossed
    double fW = pu->getFaceWeightTo( **ppv );
    double duv = ~((*ppv)->getPosition() - pu->getPosition()) * fW;
    if( (*ppv)->getCost() > pu->getCost() + duv )
    {
      // set cost to new value via u
      if( (*ppv)->getCost() == 999999999 )  // reached for first time, add to
           Q
      {
        LOGD( rank << " Pushing vertex onto Q" << std::endl );
        (*ppv)->setCost( pu->getCost() + duv );
        q.push( *ppv );
      }
      else  // already in Q, but need to re-order vertex according to new
          cost
      {
        LOGD( rank << " *CHANGING* cost of vertex on Q" << std::endl );
        (*ppv)->setCost( pu->getCost() + duv );
        q.change( *ppv );
      }
      LOGD( rank << "        cost updated to: " << (*ppv)->getCost() << endl
          );
    }
    else
    {
      // don't need ppv anymore as we didn't add it to the q
      delete *ppv;
    }
  }
  else
  {
    // don't need ppv anymore as we didn't add it to the q
    delete *ppv;
  }
}
// we're done with pu
```

```
        delete pu;
      }
    }
    t.stop();
    double timeAlgo = t.getSecs();
    t.reset();

    LOG( rank << ": Final list...\n" );
    for( unsigned int i = 0; i < g._eList.size(); i++ )
    {
      for( unsigned int j = 0; j < g._eList[i]._fCostList.size(); j++ )
      {
        LOG( rank << ": SP... edge index: " << i << ", SP index: " << j << ", cost: "
             << g._eList[i]._fCostList[j] << ", settled: " << g._eList[i].
            _bSettledList[j] << endl );
      }
    }
    t.stop();
    double timeFinal = t.getSecs();
    LOG( rank << ": Loading time: " << timeLoad << endl );
    LOG( rank << ": Structure initialisation time: " << timeInit << endl );
    LOG( rank << ": Time to find source vertex: " << timeFindSrc << endl );
    LOG( rank << ": Algorithm time: " << timeAlgo << endl );
    LOG( rank << ": ... of which carrying out reduction: " << timeReduce << endl );
    LOG( rank << ": Finalisation time: " << timeFinal << endl );
    LOG( rank << ": Total processing time (structure init + algorithm): " << timeInit
         + timeAlgo << endl );
  }
  catch( Exception& e )
  {
    LOG( e.what() << endl );
  }
  return 0;
}

int main( int argc, char * argv[] )
{
  try
  {
    // set up global vars
    if( argc != 7 )
      throw Exception( "6 args required: tininputprefix.txt numcols numrows sourcex
          sourcey sourcez" );
    g_prefix = string( argv[1] );
    g_nx = Utility::strToUI( argv[2] );
    g_ny = Utility::strToUI( argv[3] );
    g_vSource._x = Utility::strToD( string( argv[4] ) );
    g_vSource._y = Utility::strToD( string( argv[5] ) );
    g_vSource._z = Utility::strToD( string( argv[6] ) );

    barrier_init( &g_barReduce1, g_nx * g_ny );
    barrier_init( &g_barReduce2, g_nx * g_ny );
    barrier_init( &g_barReduce3, g_nx * g_ny );
    pthread_mutex_init( &g_mutReduce, NULL );

    // set up array of threads
    pthread_t *threads;
    threads = new pthread_t[g_nx * g_ny];

    // and thread parameters
    TParam *params;
    params = new TParam[g_nx * g_ny];

    // create threads
    for( unsigned int i = 0; i < g_nx * g_ny; i++ )
    {
      LOG( "R: Creating thread " << i << endl );
      params[i].rank = i;
      if( pthread_create( &threads[i], NULL, thread, static_cast< void* >( &params[i]
           ) ) )
        throw Exception( "pthread_create failed" );
    }

    // we're done...
    LOG( "R: Exiting" << endl );
    pthread_exit( NULL );
  }
  catch( Exception& e )
  {
    cerr << e.what() << endl;
  }
  return 0;
}
```

B.3 Shared memory Dijkstra implementation with a shared queue

B.3.1 Main program

```
/* test_ptdijkstra
 * Copyright (C)2004 Alistair K Phipps.  gspf@alistairphipps.com.
 *
 * Test program that does PThreads shared memory parallel dijkstra
 */

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <pthread.h>
#include "../core/log.h"
#include "../core/timer.h"
#include "../core/exception.h"
#include "../core/geometry.h"
#include "../core/gvertex.h"
```

```
#include "../core/face.h"
#include "../core/edge.h"
#include "../core/queue.h"
#include "../core/utility.h"
#include "ptqueue.h"

using namespace std;

// data shared between threads
unsigned int g_nx, g_ny;
string g_prefix;  // partition filename prefix
PTQueue *pg_q;  // pointer to shared queue of unsettled vertices
Vector3F g_vSource; // source vertex position

struct TParam
{
  int rank;   //! thread ID number
};

struct GVertexCost
{
  GVertexCost( void )
  {}
  GVertexCost( GVertex* pvt, double fCostt )
    : pv( pvt ), fCost( fCostt )
  {}
  GVertex* pv;
  double fCost;
};

void* thread( void *threadid )
{
  try
  {
    Timer t;
    t.reset();
    TParam *tp = static_cast< TParam* >( threadid );
    int rank = tp->rank;
    unsigned int cellx = rank % g_nx;
    unsigned int celly = rank / g_nx;
    LOG( "Started, rank: " << rank << " (cell " << cellx << ", " << celly << ") of "
        << g_nx * g_ny << endl );
    ostringstream oss;
    oss << g_prefix << "." << cellx << "." << celly;
    ifstream ifTin( oss.str().c_str() );
    if( !ifTin )
      throw Exception( oss.str() + " could not be opened for reading" );
    LOG( rank << " reading geometry: " << oss.str() << endl );
    Geometry g;
    ifTin >> g;
    ifTin.close();
    LOG( rank << " read geometry: " << oss.str() << endl );
    LOG( rank << " number of vertices: " << g._vList.size() << endl );
    LOG( rank << " number of edges: " << g._eList.size() << endl );
    LOG( rank << " number of faces: " << g._aList.size() << endl );
    t.stop();
    double timeLoad = t.getSecs();
    t.reset();
    // build geometry lookup tables
    g.buildLists();
    // can't be bothered dereferencing all the time
    PTQueue& g_q = *pg_q;
    g_q.setGeometryPtr( &g );
    // initialise costs to "infinity" and settled to false
    for( vector< Edge >::iterator i = g._eList.begin(); i != g._eList.end(); i++ )
    {
      for( vector< double >::iterator j = i->_fCostList.begin(); j != i->_fCostList.
          end(); j++ )
      {
        *j = 999999999;
      }
      for( vector< bool >::iterator j = i->_bSettledList.begin(); j != i->
          _bSettledList.end(); j++ )
      {
        *j = false;
      }
    }
    LOGD( rank << " setting geometry" << endl );
    t.stop();
    double timeInit = t.getSecs();
    t.reset();
    // find source vertex
    bool bFound = false;
    GVertex *pv;
    for( unsigned int i = 0; i < g._vList.size() && !bFound; i++ )
    {
      if( g._vList[i] == g_vSource )
      {
        bFound = true;
        pv = new GVertexV( i, &g );
      }
    }
    t.stop();
    double timeFindSrc = t.getSecs();
    t.reset();
    // wait for all threads to load geometry before doing a push
    g_q.waitForPush();
    t.stop();
    double timeWaitLoad = t.getSecs();
    t.reset();
    // put source vertex into queue
    // - note this call gets done by all threads that share the vertex, but it doesn'
        t matter as the queue ensures the same vertex only gets added once
    if( bFound )
    {
      g_q.push( pv, 0.0f );
      delete pv;
    }

    LOGD( rank << " entering queue loop" << endl );

    vector< GVertexCost > vPushes;

    Timer t2;
    double timeWaitingForEmpty = 0.0;
    double timeWaitingForPush = 0.0;
    while( 1 )
    {
      t2.reset();
      if( g_q.empty() )
        break;
      t2.stop();
      timeWaitingForEmpty += t2.getSecs();

      // extract element with smallest cost and remove it
      GVertex *pu = g_q.pop( &g );

      // don't do anything unless the element is in our cell
      if( pu )
      {
        // make vertex settled - we only have to do this locally (in this geometry)
            as other threads got the same item off the queue and are setting it
            settled in their geometries
        pu->setSettled( true );

        // relax non-settled vertices...
        // for each vertex v adjacent to u and not in s, if current distance to v is
            > distance to u + distance from u to v, replace current distance to v
            with distance to u + distance from u to v and add v to q if it's only now
             been reached
        vector< GVertex* > adjacent = pu->getAdjacent();
        LOGD( rank << " Element with smallest cost: " );
        LOGD( rank << "       " << pu->asVector2I() << ", cost: " << pu->getCost() <<
             endl );
        for( vector< GVertex* >::iterator ppv = adjacent.begin(); ppv != adjacent.end
            (); ppv++ )
        {
          LOGD( rank << "     adjacent element: " );
          LOGD( rank << "       " << (*ppv)->asVector2I() << ", cost: " << (*ppv)->
              getCost() << endl );
          if( !(*ppv)->getSettled() ) // if not settled
          {
            LOGD( rank << " v: " << (*ppv)->getPosition()._x << "," << (*ppv)->
                getPosition()._y << "," << (*ppv)->getPosition()._z << "\n" );
            LOGD( rank << " u: " << pu->getPosition()._x << "," << pu->getPosition().
                _y << "," << pu->getPosition()._z << "\n" );
            // face weight we multiply by is:
            // - for edge-joined SPs, the weight of the lowest-weighted face that is
                coincident with the edge
            // - for cross-face joined SPs, the weight of the face crossed
            double fW = pu->getFaceWeightTo( **ppv );
            double duv = ˜((*ppv)->getPosition() - pu->getPosition()) * fW;
            if( (*ppv)->getCost() > pu->getCost() + duv )
            {
              // set cost to new value via u
              LOGD( rank << " Pushing vertex onto Q" << endl );
              vPushes.push_back( GVertexCost( *ppv, pu->getCost() + duv ) );
              LOGD( rank << "        cost updated to: " << pu->getCost() + duv <<
                  endl );
            }
            else
            {
              // don't need ppv anymore as we didn't add it to the pushes/lowers
                  lists
              delete *ppv;
            }
          }
          else
          {
            // don't need ppv anymore as we didn't add it to the pushes/lowers lists
                as it was settled
            delete *ppv;
          }
        }
        // we're done with pu
        delete pu;
      }
      // apply updates (write to arrays + add to queue) and then clear pushes/lowers
      t2.reset();
      g_q.waitForPush();
      t2.stop();
      timeWaitingForPush += t2.getSecs();
      for( vector< GVertexCost >::iterator i = vPushes.begin(); i != vPushes.end(); i
          ++ )
      {
        g_q.push( i->pv, i->fCost );
        delete i->pv;
      }
      vPushes.resize( 0 );
    }
    t.stop();
    double timeAlgo = t.getSecs();
    t.reset();
    LOGD( rank << ": Final list...\n" );
    for( unsigned int i = 0; i < g._eList.size(); i++ )
    {
      for( unsigned int j = 0; j < g._eList[i]._fCostList.size(); j++ )
      {
        LOGD( rank << ": SP... edge index: " << i << ", SP index: " << j << ", cost:
            " << g._eList[i]._fCostList[j] << ", settled: " << g._eList[i].
            _bSettledList[j] << endl );
      }
    }
    t.stop();
    double timeFinal = t.getSecs();
    LOG( rank << ": Loading time: " << timeLoad << endl );
    LOG( rank << ": Structure initialisation time: " << timeInit << endl );
    LOG( rank << ": Time to find source vertex: " << timeFindSrc << endl );
    LOG( rank << ": Time waiting for other threads to complete loading: " <<
        timeWaitLoad << endl );
    LOG( rank << ": Algorithm time: " << timeAlgo << endl );
    LOG( rank << ": ... of which waiting for empty: " << timeWaitingForEmpty << endl
        );
    LOG( rank << ": ... of which waiting for other threads to finish reading: " <<
        timeWaitingForPush << endl );
    LOG( rank << ": Finalisation time: " << timeFinal << endl );
    LOG( rank << ": Total processing time (structure init + algorithm): " << timeInit
         + timeAlgo << endl );
  }
  catch( Exception& e )
  {
    cerr << e.what() << endl;
  }
  return NULL;
}

int main( int argc, char * argv[] )
{
  try
  {
    // set up global vars
    if( argc != 7 )
      throw Exception( "6 args required: tininputprefix.txt numcols numrows sourcex
          sourcey sourcez" );
    g_prefix = string( argv[1] );
    g_nx = Utility::strToUI( argv[2] );
    g_ny = Utility::strToUI( argv[3] );
    g_vSource._x = Utility::strToD( string( argv[4] ) );
    g_vSource._y = Utility::strToD( string( argv[5] ) );
    g_vSource._z = Utility::strToD( string( argv[6] ) );

    pg_q = new PTQueue( g_nx, g_ny );
    // set up array of threads
    pthread_t *threads;
    threads = new pthread_t[g_nx * g_ny];

    // and thread parameters
    TParam *params;
    params = new TParam[g_nx * g_ny];

    // create threads
    for( unsigned int i = 0; i < g_nx * g_ny; i++ )
    {
      LOG( "R: Creating thread " << i << endl );
      params[i].rank = i;
      if( pthread_create( &threads[i], NULL, thread, static_cast< void* >( &params[i]
          ) ) )
        throw Exception( "pthread_create failed" );
    }

    // we're done...
    LOG( "R: Exiting" << endl );
    pthread_exit( NULL );
  }
  catch( Exception& e )
  {
    cerr << e.what() << endl;
  }
  return 0;
}
```

B.3.2 Threaded queue wrapper source

```
#include <vector>
#include <pthread.h>
#include "../core/log.h"
#include "../core/exception.h"
#include "../core/geometry.h"
#include "../core/gvertex.h"
#include "ptqueue.h"

using namespace std;

PTQueue::PTQueue( unsigned int iX, unsigned int iY )
{
  _iX = iX;
  _iY = iY;
  int r = pthread_mutex_init( &_mutW, NULL );
  if( r != 0 )
    throw Exception( "Error initialising writer mutex" );
  barrier_init( &_barT, iX * iY );
  barrier_init( &_barP, iX * iY );
  barrier_init( &_barW, iX * iY );
  _pgList.resize( iX );
  for( unsigned int i = 0; i < iX; i++ )
  {
    _pgList[i].resize( iY );
  }
}

void PTQueue::setGeometryPtr( Geometry *pg )
{
  pthread_mutex_lock( &_mutW );
  _pgList[pg->_v2Cell._x][pg->_v2Cell._y] = pg;
  pthread_mutex_unlock( &_mutW );
}

bool PTQueue::empty( void )
```

```
{
  // wait here until all threads have called empty() - i.e. have finished writing to
      the queue
  int status = barrier_wait( &_barT );
  if( status > 0 )
    throw Exception( "barrier_wait in PTQueue::empty failed" );

  return _q.empty();
}
void PTQueue::push( GVertex* pv, double fCost )
{
  pthread_mutex_lock( &_mutW );
  GVertex* pu = setGlobalVertexCostAndGetVertexInBaseGeometry( pv, fCost );
  _q.lower( pu ); // always do lower as someone might have got there first and pushed
       it
  pthread_mutex_unlock( &_mutW );
}

GVertex* PTQueue::top( Geometry *pg )
{
  // if there isn't a match - i.e. this GVertex is not in the requested cell, return
      NULL
  GVertex *pr = NULL;

  // get vertex that's on top of queue - it is in base cell
  // don't try reading the queue if someone else is writing to it
  pthread_mutex_lock( &_mutW );
  GVertex *pu = _q.top();
  pthread_mutex_unlock( &_mutW );
  // we need to find out equivalent GVertex in requested cell
  if( pu->getType() == GVertex::GVERTEX_V )
  {
    // if it's already in the cell we want, return it
    if( pu->getGeometry() == pg )
      pr = new GVertexV( *( static_cast< GVertexV* >( pu ) ) );
    else
    {
      // go through all cells in shared list
      vector< Vector3I > &vl = pu->getGeometry()->_vSharedList[pu->asVector2I()._x];
      for( vector< Vector3I >::const_iterator i = vl.begin(); i != vl.end(); i++ )
      {
        // if we find a matching one...
        if( i->_x == pg->_v2Cell._x && i->_y == pg->_v2Cell._y )
        {
          // return the appropriate vertex
          pr = new GVertexV( i->_z, pg );
          break;
        }
      }
    }
  }
  else if( pu->getType() == GVertex::GVERTEX_E )
  {
    // if it’s already in the cell we want, return it
    if( pu->getGeometry() == pg )
      pr = new GVertexE( *( static_cast< GVertexE* >( pu ) ) );
    else
    {
      // go through all cells in shared list
      vector< Vector3I > &el = pu->getGeometry()->_eSharedList[pu->asVector2I()._x];
      for( vector< Vector3I >::const_iterator i = el.begin(); i != el.end(); i++ )
      {
        // if we find a matching one...
        if( i->_x == pg->_v2Cell._x && i->_y == pg->_v2Cell._y )
        {
          // return the appropriate vertex
          pr = new GVertexE( i->_z, pu->asVector2I()._y, pg );
          break;
        }
      }
    }
  }
  else
    throw Exception( "Unsupported GVertex type in PTQueue.top" );

  return pr;
}

void PTQueue::waitForPush( void )
{
  int status = barrier_wait( &_barW );
  if( status > 0 )
    throw Exception( "barrier_wait in PTQueue::waitForPush failed" );
}

GVertex* PTQueue::pop( Geometry *pg )
{
  GVertex *vR = top( pg );
  // wait for all threads to have called pop (i.e. we are done with top)
  int status = barrier_wait( &_barP );
  if( status > 0 )
    throw Exception( "barrier_wait in PTQueue::pop failed" );
  // only one waiting thread gets a status of -1 - make it delete the top item from
      the queue
  if( status == -1 )
  {
    pthread_mutex_lock( &_mutW );
    delete _q.pop();
    pthread_mutex_unlock( &_mutW );
  }
  return vR;
}

GVertex* PTQueue::setGlobalVertexCostAndGetVertexInBaseGeometry( GVertex *pv, double
    fCost )
{
```

```
// returns *new* GVertex
GVertex *pr;
// set cost in this geometry
pv->setCost( fCost );

if( pv->getType() == GVertex::GVERTEX_V )
{
  // start off with a copy of the one we're passed in
  pr = new GVertexV( *(static_cast< const GVertexV* >( pv ) ) );
  // go through all cells in shared list
  vector< Vector3I > &vl = pv->getGeometry()->_vSharedList[pv->asVector2I()._x];
  for( vector< Vector3I >::const_iterator i = vl.begin(); i != vl.end(); i++ )
  {
    LOGD( "cell: " << pv->getGeometry()->_v2Cell << " gvertexv: " << pv->asVector2I
        () << " shared cellx,celly,index: " << *i << endl );
    // set cost
    GVertexV v = GVertexV( i->_z, getGeometryFromCell( Vector2I( i->_x, i->_y ) ) )
        ;
    v.setCost( fCost );
    // if it has a lower cell number, set pR to that instead
    if( ( i->_x + i->_y * _iX ) < ( pr->getGeometry()->_v2Cell._x + pr->getGeometry
        ()->_v2Cell._y * _iX ) )
    {
      delete pr;
      pr = new GVertexV( i->_z, getGeometryFromCell( Vector2I( i->_x, i->_y ) ) );
    }
  }
}
else if( pv->getType() == GVertex::GVERTEX_E )
{
  // start off with a copy of the one we're passed in
  pr = new GVertexE( *(static_cast< const GVertexE* >( pv ) ) );
  // go through all cells in shared list
  vector< Vector3I > &el = pv->getGeometry()->_eSharedList[pv->asVector2I()._x];
  for( vector< Vector3I >::const_iterator i = el.begin(); i != el.end(); i++ )
  {
    LOGD( "cell: " << pv->getGeometry()->_v2Cell << " gvertexe: " << pv->asVector2I
        () << " shared cellx,celly,index: " << *i << endl );
    // set cost
    GVertexE e = GVertexE( i->_z, pv->asVector2I()._y, getGeometryFromCell(
        Vector2I( i->_x, i->_y ) ) );
    e.setCost( fCost );
    // if it has a lower cell number, set pr to that instead
    if( ( i->_x + i->_y * _iX ) < ( pr->getGeometry()->_v2Cell._x + pr->getGeometry
        ()->_v2Cell._y * _iX ) )
    {
      delete pr;
      pr = new GVertexE( i->_z, pv->asVector2I()._y, getGeometryFromCell( Vector2I(
            i->_x, i->_y ) ) );
    }
  }
}
return pr;
```

```
}

Geometry* PTQueue::getGeometryFromCell( Vector2I v2C ) const
{
  return _pgList[v2C._x][v2C._y];
}
```

B.3.3 Threaded queue wrapper header

```
#ifndef PTQUEUE_H
#define PTQUEUE_H

#include <vector>
#include <pthread.h>
#include "../core/vector2.h"
#include "../core/queue.h"
#include "barrier.h"

// forward declarations
class GVertex;
class Geometry;

class PTQueue
{
public:
  PTQueue( unsigned int iX, unsigned int iY );
  void setGeometryPtr( Geometry *pg );
  bool empty( void );
  void push( GVertex* pv, double fCost );
  GVertex* top( Geometry *pg );
  GVertex* pop( Geometry *pg );
  void waitForPush( void );
private:
  GVertex* setGlobalVertexCostAndGetVertexInBaseGeometry( GVertex *pv, double fCost )
      ;
  Geometry* getGeometryFromCell( Vector2I v2C ) const;
  pthread_mutex_t _mutW;
  barrier_t _barT;
  barrier_t _barP;
  barrier_t _barW;
  Queue< GVertex* > _q;
  unsigned int _iX, _iY;  //! number of cell columns, rows
  std::vector< std::vector< Geometry* > > _pgList;
};

#endif
```

B.4 Message passing Lanthier implementation

```
/* test_mpilanthier
 * Copyright (C)2004 Alistair K Phipps.  gspf@alistairphipps.com.
 *
 * Test program that does MPI distributed memory parallel Lanthier algorithm
 */

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <mpi.h>
#include "../core/timer.h"
#include "../core/log.h"
#include "../core/exception.h"
#include "../core/geometry.h"
#include "../core/gvertex.h"
#include "../core/face.h"
#include "../core/edge.h"
#include "../core/queue.h"
#include "../core/utility.h"

using namespace std;

int main( int argc, char * argv[] )
{
  try
  {
    Timer t;
    t.reset();
    int rank, size;
    unsigned int nx, ny;  // number of rows / cols in MFP split
    MPI::Init( argc, argv );
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    if( argc != 10 )
      throw Exception( "9 args required: tininputprefix.txt numcols numrows sourcex
          sourcey sourcez targetx targety targetz" );
    nx = Utility::strToUI( argv[2] );
    ny = Utility::strToUI( argv[3] );

    Vector3F vSource;
    vSource._x = Utility::strToD( string( argv[4] ) );
    vSource._y = Utility::strToD( string( argv[5] ) );
    vSource._z = Utility::strToD( string( argv[6] ) );
    Vector3F vTarget;
    vTarget._x = Utility::strToD( string( argv[7] ) );
    vTarget._y = Utility::strToD( string( argv[8] ) );
    vTarget._z = Utility::strToD( string( argv[9] ) );

    if( nx * ny != size )
      throw Exception( "numcols * numrows does not match MPI world size" );
    unsigned int cellx = rank % nx;
    unsigned int celly = rank / nx;
    LOG( "Started, rank: " << rank << " (cell " << cellx << ", " << celly << ") of "
        << size << endl );
```

```
ostringstream oss;
oss << argv[1] << "." << cellx << "." << celly;
ifstream ifTin( oss.str().c_str() );
if( !ifTin )
  throw Exception( oss.str() + " could not be opened for reading" );
LOG( rank << " reading geometry: " << oss.str() << endl );
Geometry g;
ifTin >> g;
ifTin.close();
LOG( rank << " read geometry: " << oss.str() << endl );
LOG( rank << " number of vertices: " << g._vList.size() << endl );
LOG( rank << " number of edges: " << g._eList.size() << endl );
LOG( rank << " number of faces: " << g._aList.size() << endl );
t.stop();
double timeLoad = t.getSecs();
t.reset();
// build geometry lookup tables
g.buildLists();
// create a buffer for buffered mode sends
LOGD( rank << " creating sending buffer" << endl );
unsigned char *buf = new unsigned char[500000];
LOGD( rank << " attaching sending buffer" << endl );
MPI::Attach_buffer( buf, 500000 );

// initialise costs to "infinity" and settled to false (settled unused)
for( vector< Edge >::iterator i = g._eList.begin(); i != g._eList.end(); i++ )
{
  for( vector< double >::iterator j = i->_fCostList.begin(); j != i->_fCostList.
      end(); j++ )
  {
    *j = 999999999;
  }
  for( vector< bool >::iterator j = i->_bSettledList.begin(); j != i->
      _bSettledList.end(); j++ )
  {
    *j = false;
  }
}
enum ETag { TAG_COST, TAG_DONE, TAG_TERMINATE, TAG_UPDATE };

// structure representing done token (that we currently hold)
struct
{
  bool held;
  int originator;
  int timesround;
} done;
// processor 0 holds it initially
if( rank == 0 )
{
  done.held = true;
  done.originator = 0;
  done.timesround = -1;
}
else
  done.held = false;

// pack structures we send/receive as arrays
#pragma pack(1)
// structure representing done token that we send/receive
struct
{
  int originator;
  int timesround;
} donetok;

// structure representing cost token that we send/receive
struct
{
  double fMaxCost;
  int originator;
} costtok;

// structure representing cost update that we send/receive
struct
{
  unsigned int ie;
  unsigned int is;
  float fCost;
} update;
// end of packed structures
#pragma pack(0)

double fLocalCost = 99999999.0; // cost we've got to on the queue
double fMaxCost = 99999999.0; // cost others have got to (max we simulate up to)
bool bTerminate = false;  // set when we receive TERMINATE

Queue< GVertex* > q; // queue of unsettled verts

t.stop();
double timeInit = t.getSecs();
t.reset();
// find source vertex
bool bFound = false;
GVertex *pv;
for( unsigned int i = 0; i < g._vList.size() && !bFound; i++ )
{
  if( g._vList[i] == vSource )
  {
    bFound = true;
    pv = new GVertexV( i, &g );
    pv->setCost( 0.0 );
    // put first vertex into queue of unsettled vertices
    q.push( pv );
    // don't bother sending out cost update as all nodes are setting the cost in
        here!
  }
}
t.stop();
double timeFindSrc = t.getSecs();
t.reset();

LOGD( rank << " entering queue loop" << endl );

Timer t2;
double timeIdle = 0.0;
while( !bTerminate )  // we explicitly exit out of this loop
{
  MPI::Status status; // status object filled in by probes

  // if we have an incoming message... process them first
  while( !bTerminate && MPI::COMM_WORLD.Iprobe( MPI::ANY_SOURCE, MPI::ANY_TAG,
      status ) )
  {
    LOGD( rank << " received message from " << status.Get_source() << ": " );
    // if we have an incoming cost token...
    if( status.Get_tag() == TAG_COST )
    {
      LOGD( " cost token" << endl );
      // receive the message
      MPI::COMM_WORLD.Recv( &costtok, sizeof( costtok ), MPI_BYTE, MPI::
          ANY_SOURCE, TAG_COST, status );
      // if it wasn't originally sent by us...
      if( costtok.originator != rank )
      {
        // don't bother sending it on if it's higher than our current max cost (
            which we've already sent a token out for)
        if( costtok.fMaxCost >= fMaxCost )
        {
          LOGD( rank << " received max cost of " << costtok.fMaxCost << ",
              greater than current max cost of " << fMaxCost << ": suppressing
              token" << endl );
        }
        else
        {
          LOGD( rank << " updating max cost from " << fMaxCost << " to " <<
              costtok.fMaxCost << " and relaying token" << endl );
          // update our max cost
          fMaxCost = costtok.fMaxCost;
          // send on token
          MPI::COMM_WORLD.Bsend( &costtok, sizeof( costtok ), MPI_BYTE, ( rank +
              1 ) % size, TAG_COST );
        }
      }
    }
    // if we have an incoming done token...
    if( status.Get_tag() == TAG_DONE )
    {
      LOGD( rank << " done token" << endl );
      // receive the message
      MPI::COMM_WORLD.Recv( &donetok, sizeof( donetok ), MPI_BYTE, MPI::
          ANY_SOURCE, TAG_DONE, status );
      // hold onto done token
      LOGD( rank << " holding onto done token with originator " << donetok.
          originator << ", timesround " << donetok.timesround << endl );
      done.held = true;
      // if we have no work to do and no other messages pending, then store this
          token for forwarding
      if( ( q.empty() || ( fLocalCost > fMaxCost ) ) && ( !MPI::COMM_WORLD.Iprobe
          ( MPI::ANY_SOURCE, MPI::ANY_TAG, status ) ) )
      {
        done.originator = donetok.originator;
        done.timesround = donetok.timesround;
        // if it was originated by us, it has gone round
        if( done.originator == rank )
          done.timesround++;
        // if was originally sent by us and has now gone round twice...
        if( ( done.originator == rank ) && ( done.timesround == 2 ) )
        {
          LOGD( rank << " sending terminate" << endl );
          // send terminate to next processor
          MPI::COMM_WORLD.Bsend( NULL, 0, MPI_BYTE, ( rank + 1 ) % size,
              TAG_TERMINATE );
          // we only terminate when we receive TERMINATE ourselves
        }
      }
      else // else reset the done token to our details so when it is sent, a new
          round begins
      {
        done.originator = rank;
        done.timesround = 0;
      }
    }
    // if we have an incoming cost update...
    if( status.Get_tag() == TAG_UPDATE )
    {
      LOGD( rank << " cost update" << endl );
      // receive message
      MPI::COMM_WORLD.Recv( &update, sizeof( update ), MPI_BYTE, MPI::ANY_SOURCE,
           TAG_UPDATE, status );
      // create GVertex from the message
      GVertex *pv;
      if( update.is == 3000000 )
      {
        pv = new GVertexV( update.ie, &g );
      }
      else
      {
        pv = new GVertexE( update.ie, update.is, &g );
      }
      // if it has a better cost than our local one, update our local one
      if( update.fCost < pv->getCost() )
```

```
          {
            LOGD( rank << " cost update has caused actual update" << endl );
            pv->setCost( update.fCost );
            // if this vertex is already on our queue (could be in any position),
                then update its position appropriately
            // FIXME: make this faster (have ref from ie, is to GVertex pointer?)
            // FIXME: also this will mem leak as the old vertex that this one
                replaces gets lost
            q.change( pv );
          }
        }
        // if we have an incoming terminate message...
        if( status.Get_tag() == TAG_TERMINATE )
        {
          LOGD( rank << " terminate received" << endl );
          // receive the message
          MPI::COMM_WORLD.Recv( NULL, 0, MPI_BYTE, MPI::ANY_SOURCE, TAG_TERMINATE,
              status );
          // send terminate to next processor, unless we were the originator
          if( !( done.held && done.originator == rank) )
            MPI::COMM_WORLD.Bsend( NULL, 0, MPI_INT, ( rank + 1 ) % size,
                TAG_TERMINATE );
          // exit while() loop
          bTerminate = true;
        }
      }
      // if we've received a terminate message, don't try to process our queue
      if( !bTerminate )
      {
        // if our queue is empty or our local cost has reached the max cost we go up
            to
        if( q.empty() || ( fLocalCost > fMaxCost ) )
        {
          LOGD( rank << " has nothing to process" << endl );
          // if we hold the done token but we haven't yet sent terminate
          if( done.held && !(done.timesround == 2) )
          {
            // send done token to next processor
            donetok.originator = done.originator;
            donetok.timesround = done.timesround;

            LOGD( rank << " sending done token" << endl );
            MPI::COMM_WORLD.Bsend( &donetok, sizeof( donetok ), MPI_BYTE, ( rank + 1
                ) % size, TAG_DONE );
            // we no longer hold the token
            done.held = false;
          }
          LOGD( rank << " waiting for incoming message" << endl );
          // wait for incoming message
          t2.reset();
          MPI::COMM_WORLD.Probe( MPI::ANY_SOURCE, MPI::ANY_TAG, status );
          t2.stop();
          timeIdle += t2.getSecs();
        }
        else // else we have stuff to process locally
        {
          LOGD( rank << " has local processing to do" << endl );
          // remove vertex from top of queue
          GVertex *pu = q.pop();

          LOGD( rank << " Element with smallest cost: " );
          LOGD( rank << "       " << pu->asVector2I() << ", cost: " << pu->getCost()
              << endl );

          fLocalCost = pu->getCost();
          // if the cost we've got to on our queue is still less than the max cost we
              're going up to... continue
          if( fLocalCost < fMaxCost )
          {
            // if the vertex we picked is the target vertex, update maxcost (as there
                's no point in finding paths that are longer than this one we've just
                 found)
            if( pu->getPosition() == vTarget )
            {
              fMaxCost = pu->getCost();
              // send cost token to update other nodes' maxcost
              costtok.fMaxCost = pu->getCost();
              costtok.originator = rank;
              LOGD( rank << " sending cost token with value " << costtok.fMaxCost <<
                  endl );
              MPI::COMM_WORLD.Bsend( &costtok, sizeof( costtok ), MPI_BYTE, ( rank +
                  1 ) % size, TAG_COST );
            }
            else  // else process adjacent vertices
            {
              vector< GVertex* > adjacent = pu->getAdjacent();
              for( vector< GVertex* >::iterator ppv = adjacent.begin(); ppv !=
                  adjacent.end(); ppv++ )
              {
                LOGD( rank << "     adjacent element: " );
                LOGD( rank << "       " << (*ppv)->asVector2I() << ", cost: " << (*
                    ppv)->getCost() << endl );

                double fW = pu->getFaceWeightTo( **ppv );
                double duv = ~((*ppv)->getPosition() - pu->getPosition()) * fW;
                if( (*ppv)->getCost() > pu->getCost() + duv )
                {
                  // set cost to new value via u
                  if( (*ppv)->getCost() == 999999999 )  // reached for first time,
                      add to Q
                  {
                    LOGD( rank << " Pushing vertex onto Q" << std::endl );
                    (*ppv)->setCost( pu->getCost() + duv );
                    q.push( *ppv );
                  }
                  else  // already in Q, but need to re-order vertex according to new
                       cost
                  {
                    LOGD( rank << " *CHANGING* cost of vertex on Q" << std::endl );
                    (*ppv)->setCost( pu->getCost() + duv );
                    q.change( *ppv );
                  }
                  LOGD( rank << "        cost updated to: " << (*ppv)->getCost() <<
                      endl );

                  // send cost update msg to all processors in ppv's shared list
                  if( (*ppv)->getType() == GVertex::GVERTEX_V )
                  {
                    // find all (cellid, vertexid) for cells that share this vertex
                    vector< Vector3I > v3CellList = g._vSharedList[(*ppv)->asVector2I
                        ()._x];
                    for( unsigned int i = 0; i < v3CellList.size(); i++ )
                    {
                      update.ie = v3CellList[i]._z;
                      update.is = 3000000;
                      update.fCost = (*ppv)->getCost();
                      LOGD( rank << " sending cost update to " << v3CellList[i]._x +
                          v3CellList[i]._y * nx << endl );
                      MPI::COMM_WORLD.Bsend( &update, sizeof( update ), MPI_BYTE,
                          v3CellList[i]._x + v3CellList[i]._y * nx, TAG_UPDATE );
                    }
                  }
                  if( (*ppv)->getType() == GVertex::GVERTEX_E )
                  {
                    // find all (cellid, edgeid) for cells that share this edge
                    vector< Vector3I > v3CellList = g._eSharedList[(*ppv)->asVector2I
                        ()._x];
                    for( unsigned int i = 0; i < v3CellList.size(); i++ )
                    {
                      update.ie = v3CellList[i]._z;
                      update.is = (*ppv)->asVector2I()._y;
                      update.fCost = (*ppv)->getCost();
                      LOGD( rank << " sending cost update to " << v3CellList[i]._x +
                          v3CellList[i]._y * nx << endl );
                      MPI::COMM_WORLD.Bsend( &update, sizeof( update ), MPI_BYTE,
                          v3CellList[i]._x + v3CellList[i]._y * nx, TAG_UPDATE );
                    }
                  }
                }
              }
            }
          }
        }
      }

    }
    t.stop();
    double timeAlgo = t.getSecs();
```

```
    t.reset();
    LOG( rank << ": Final list..." << endl );
    for( unsigned int i = 0; i < g._eList.size(); i++ )
    {
      for( unsigned int j = 0; j < g._eList[i]._fCostList.size(); j++ )
      {
        LOG( rank << ": SP... edge index: " << i << ", SP index: " << j << ", cost: "
             << g._eList[i]._fCostList[j] << ", settled: " << g._eList[i].
            _bSettledList[j] << endl );
      }
    }
    LOGD( rank << ": Finalizing" << endl );
    MPI::Finalize();
    // free buffer (Finalize detaches it)
    LOGD( rank << " freeing sending buffer" << endl );
    delete[] buf;
    LOGD( rank << ": Exiting" << endl );
    t.stop();
    double timeFinal = t.getSecs();
    LOG( rank << ": Loading time: " << timeLoad << endl );
    LOG( rank << ": Structure initialisation time: " << timeInit << endl );
    LOG( rank << ": Time to find source vertex: " << timeFindSrc << endl );
    LOG( rank << ": Algorithm time: " << timeAlgo << endl );
    LOG( rank << ": ... of which we were idle: " << timeIdle << endl );
    LOG( rank << ": Finalisation time: " << timeFinal << endl );
    LOG( rank << ": Total processing time (structure init + algorithm): " << timeInit
         + timeAlgo << endl );
  }
  catch( Exception& e )
  {
    cerr << e.what() << endl;
  }
  return 0;
}
```

Appendix C

Graphs of Results

This chapter contains the set of primary result graphs, including ones that are not referred to within the text. Note that no attempt has been made to remove outlying data. All machines used were public-access and occasionally there were issues with them, which have caused a couple of strange timings, which are shown here to demonstrate the issues faced. A complete set of timings in the form of the captured output of program runs is available on DICE in */home/v1aphipp/PROJECT/data/final-complete-timing-data.tar.gz*.

C.1 Algorithm time vs number of faces, per algorithm, system, queue

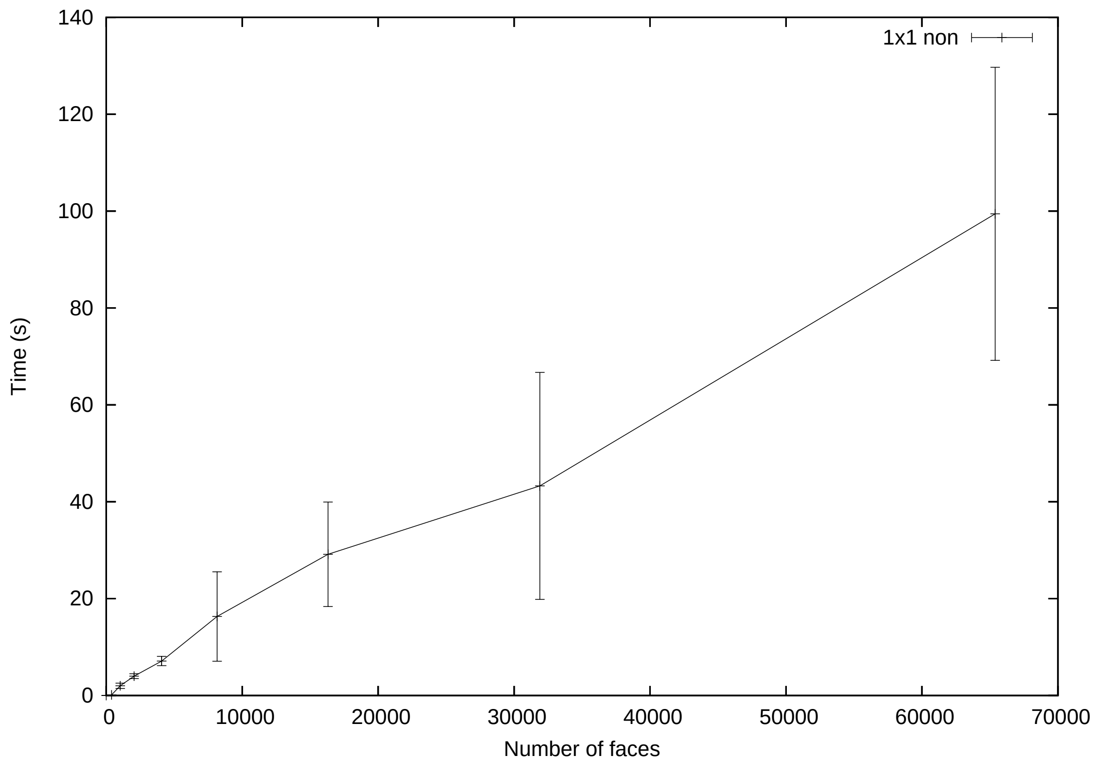

Figure C.1: Algorithm Time, Sequential Dijkstra, 16-node Beowulf, Sorted Queue

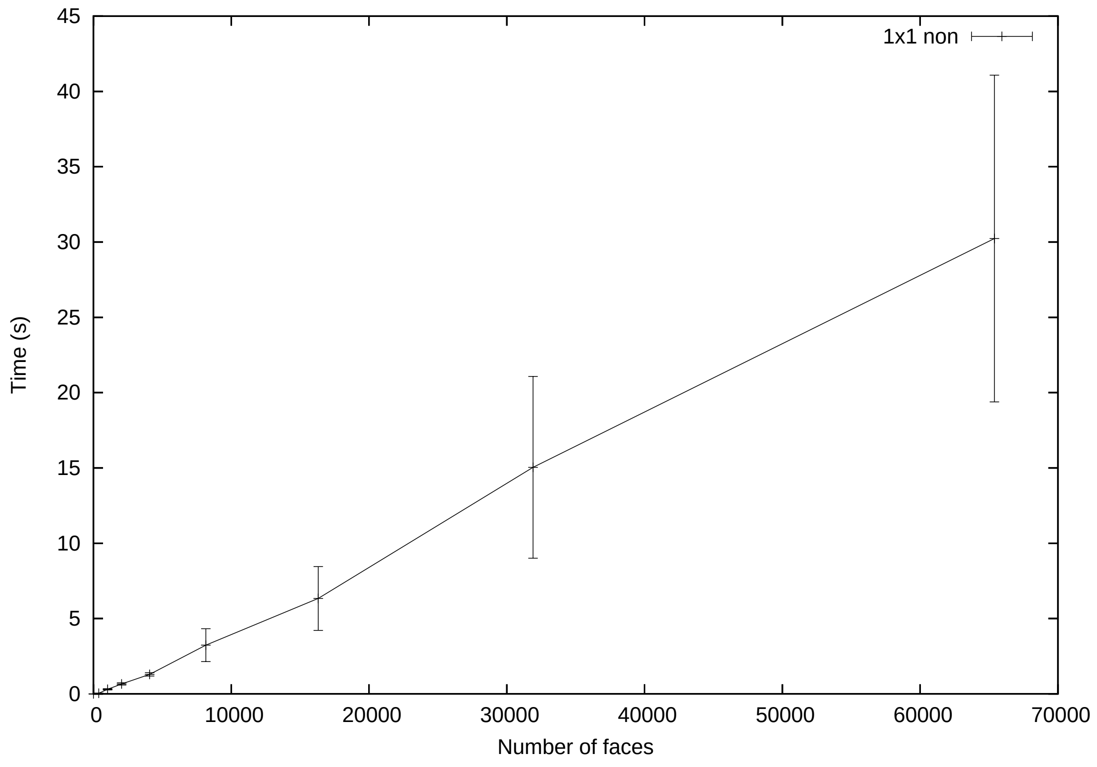

Figure C.2: Algorithm Time, Sequential Dijkstra, 16-node Beowulf, Unsorted Queue

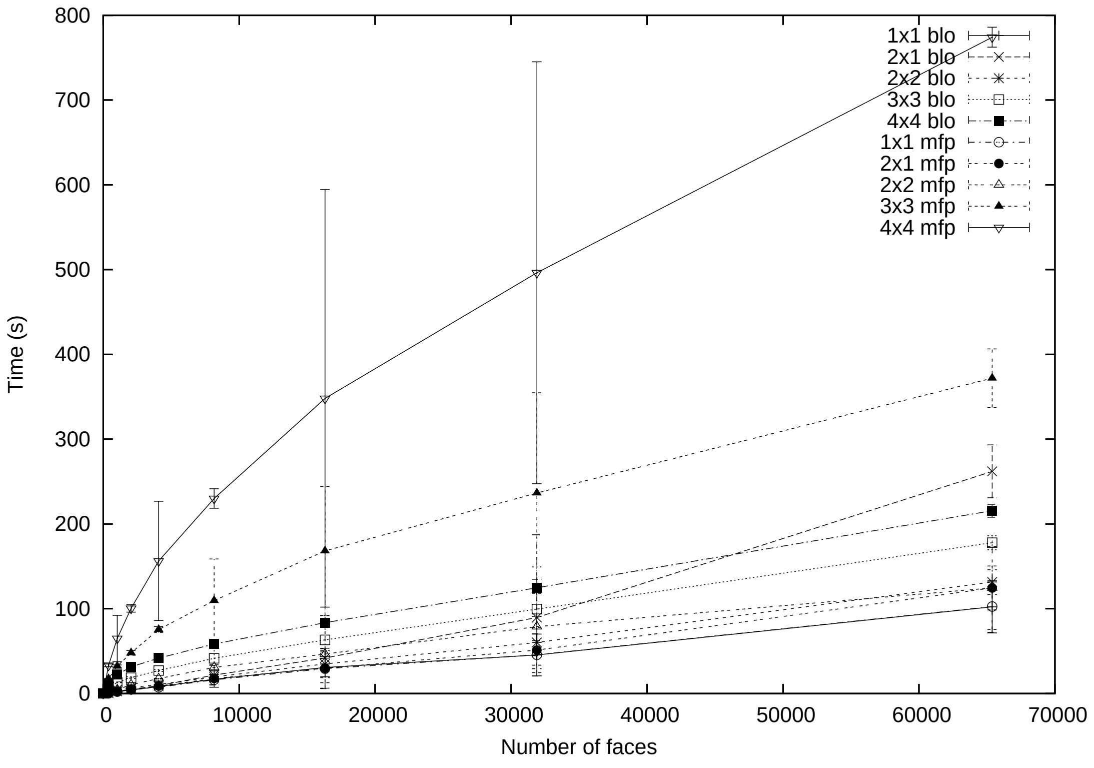

Figure C.3: Maximum Node Algorithm Time, MPI Dijkstra, 16-node Beowulf, Sorted Queue

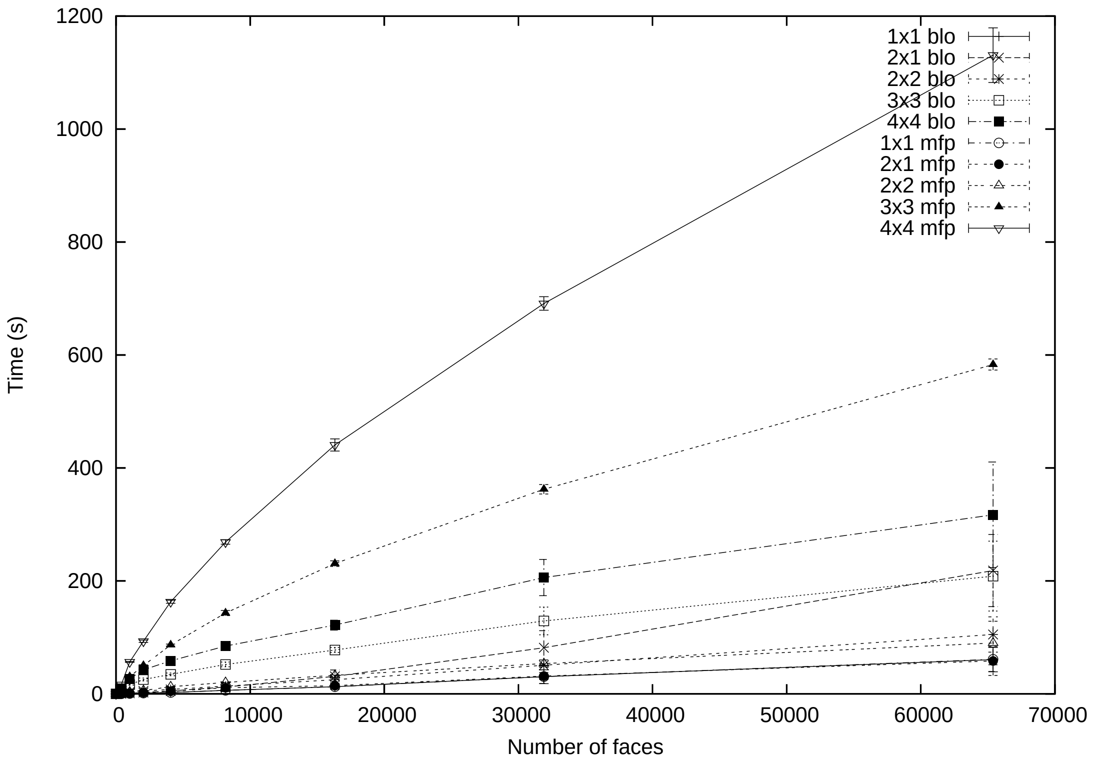

Figure C.4: Maximum Node Algorithm Time, MPI Dijkstra, 16-node Beowulf, Unsorted Queue

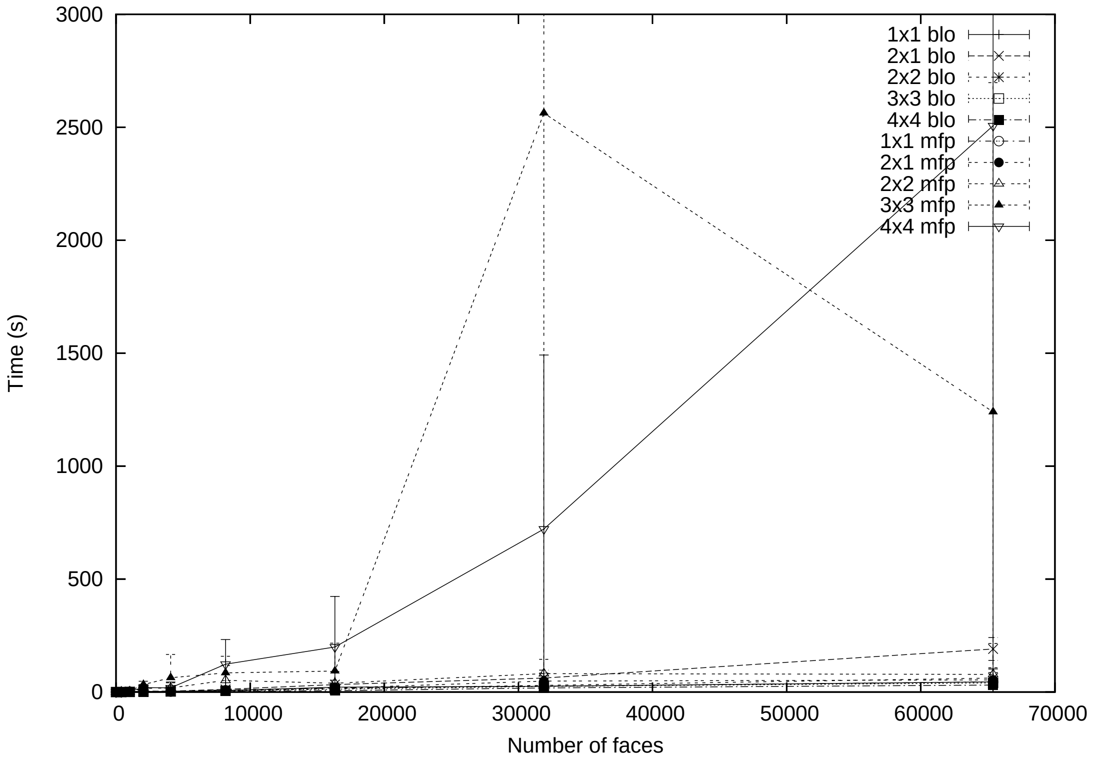

Figure C.5: Maximum Node Algorithm Time, MPI Lanthier, 16-node Beowulf, Sorted Queue

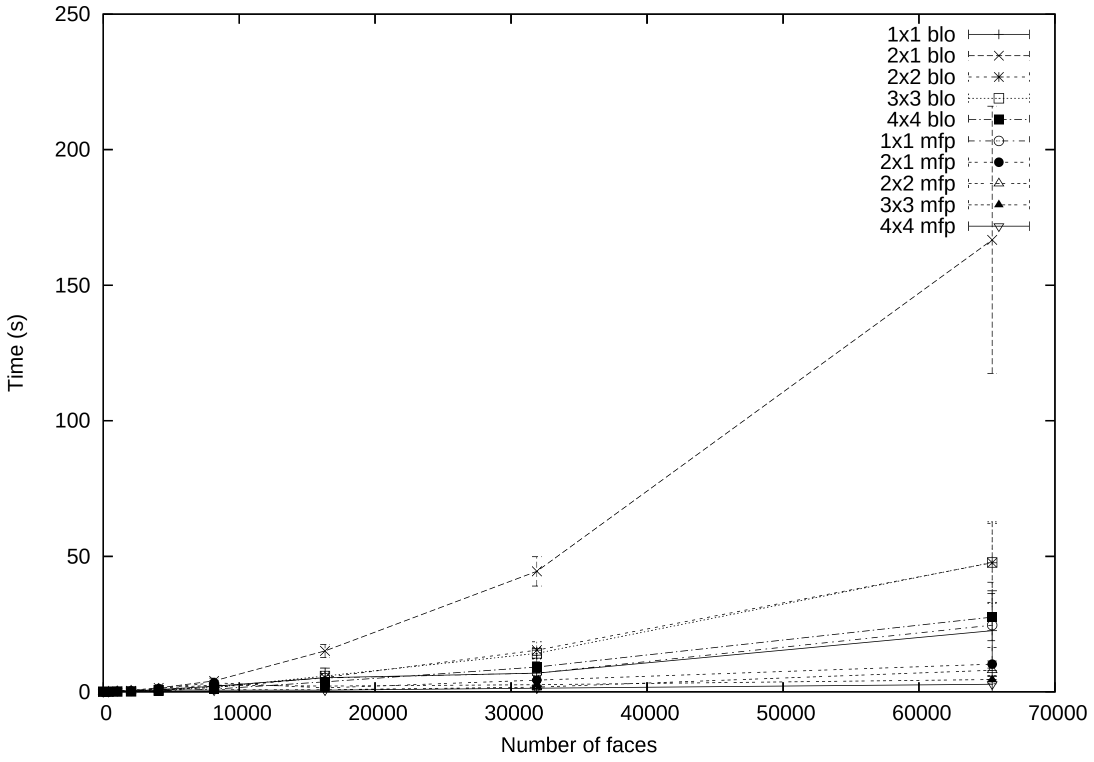

Figure C.6: Maximum Node Algorithm Time, MPI Lanthier, 16-node Beowulf, Unsorted Queue

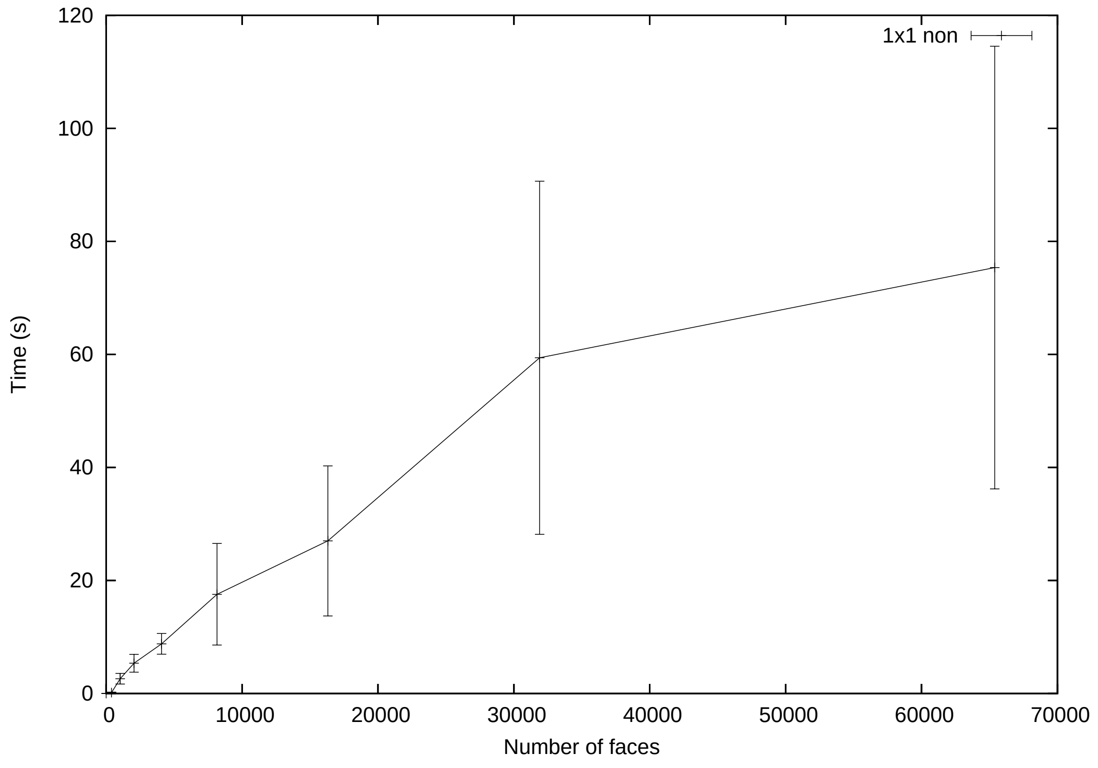

Figure C.7: Algorithm Time, Sequential Dijkstra, 4-way SMP, Sorted Queue

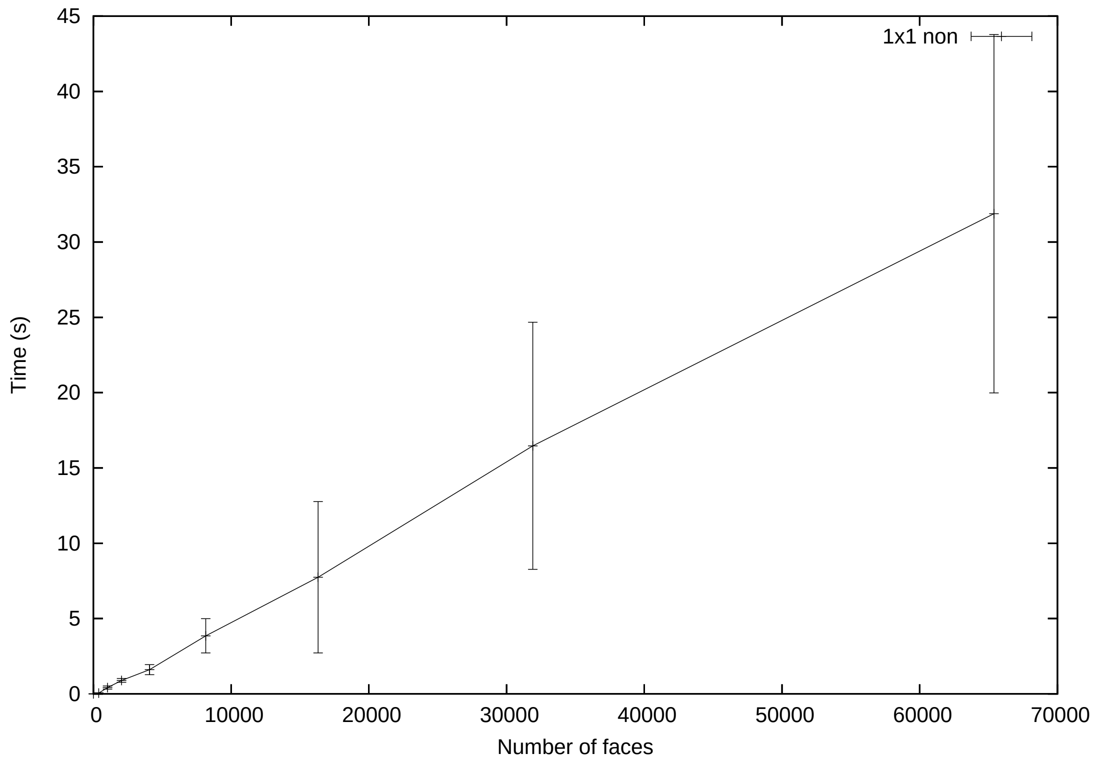

Figure C.8: Algorithm Time, Sequential Dijkstra, 4-way SMP, Unsorted Queue

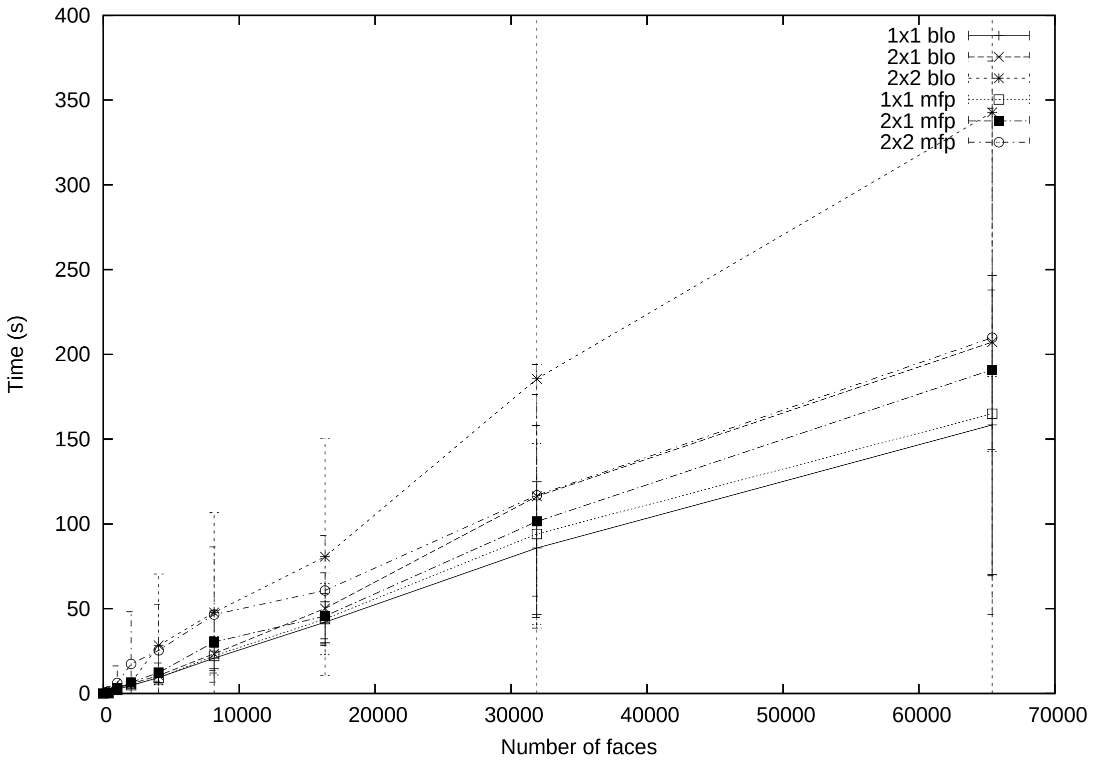

Figure C.9: Maximum Thread Algorithm Time, PThread Dijkstra, 4-way SMP, Sorted Queue

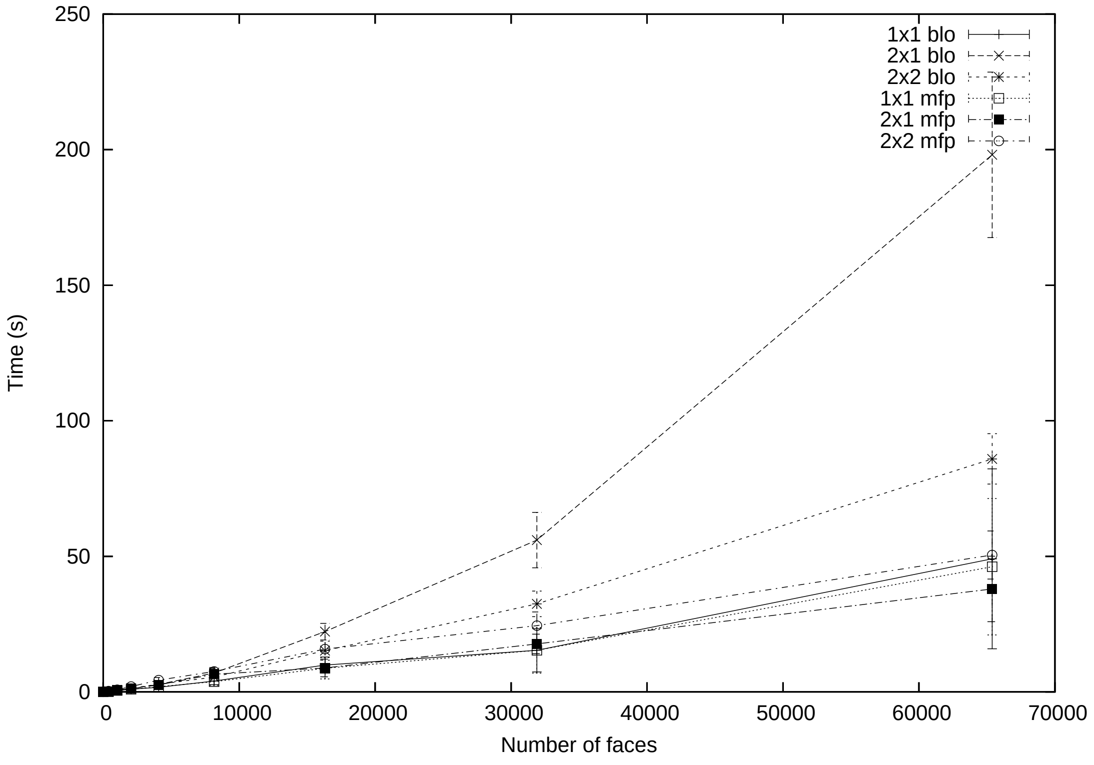

Figure C.10: Maximum Thread Algorithm Time, PThread Dijkstra, 4-way SMP, Unsorted Queue

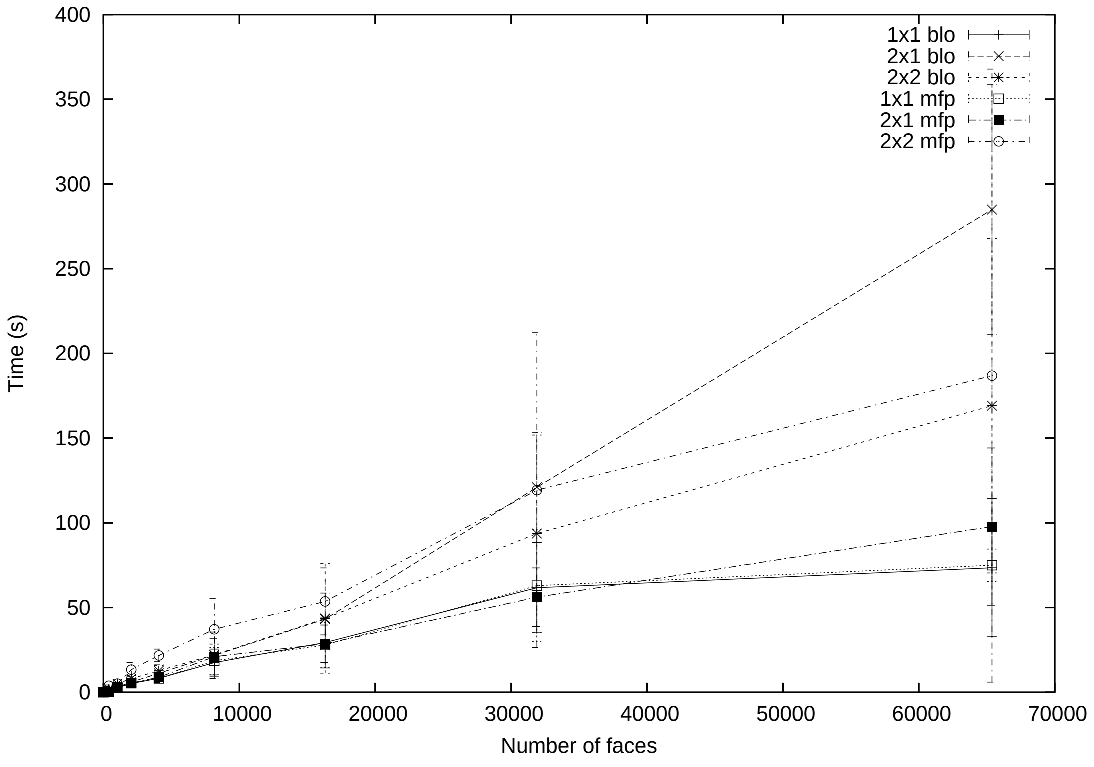

Figure C.11: Maximum Thread Algorithm Time, MPI Dijkstra, 4-way SMP, Sorted Queue

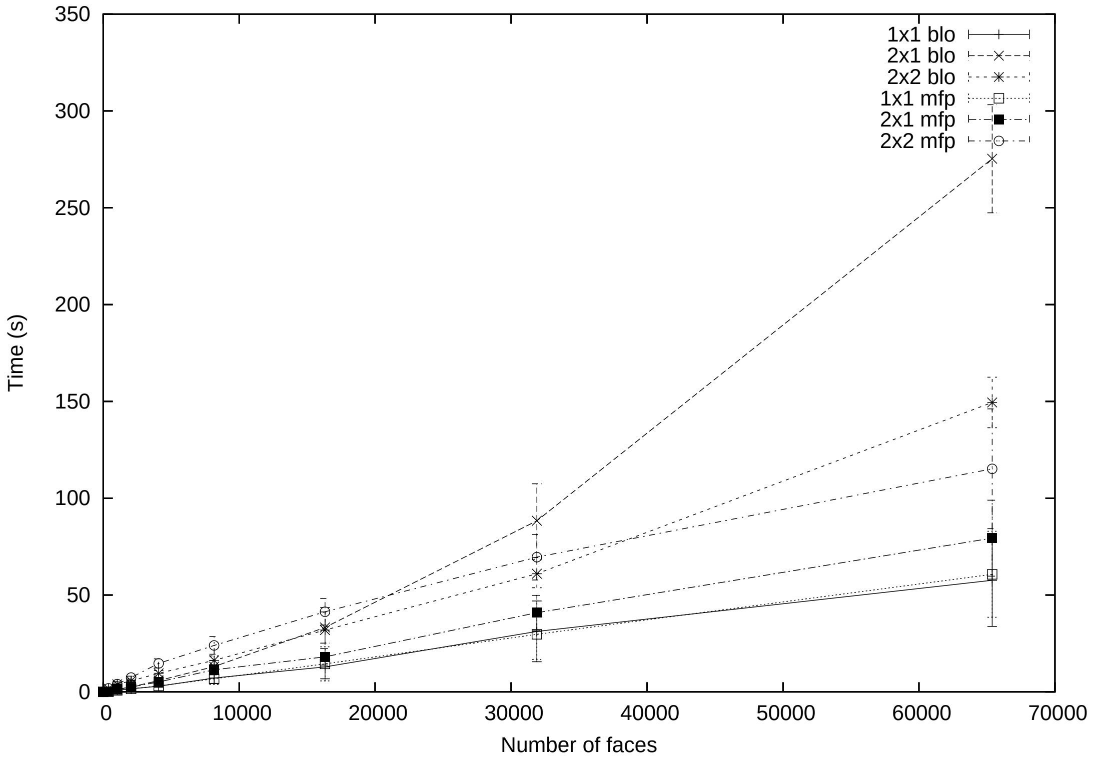

Figure C.12: Maximum Thread Algorithm Time, MPI Dijkstra, 4-way SMP, Unsorted Queue

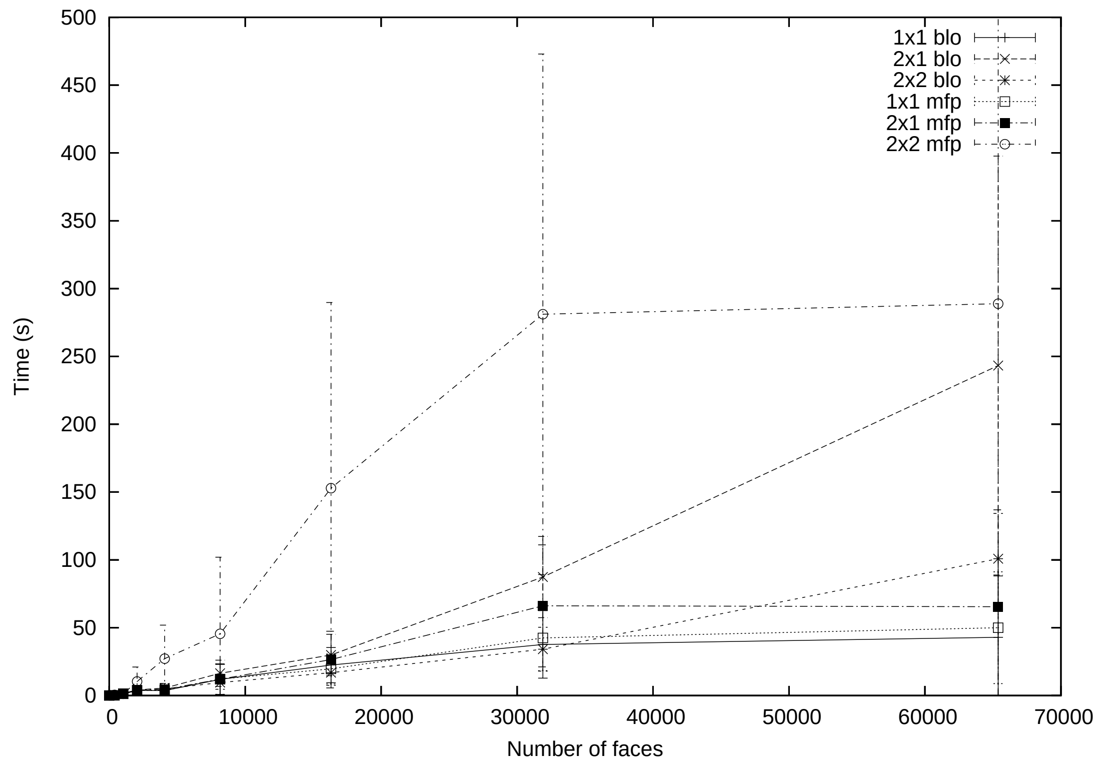

Figure C.13: Maximum Thread Algorithm Time, MPI Lanthier, 4-way SMP, Sorted Queue

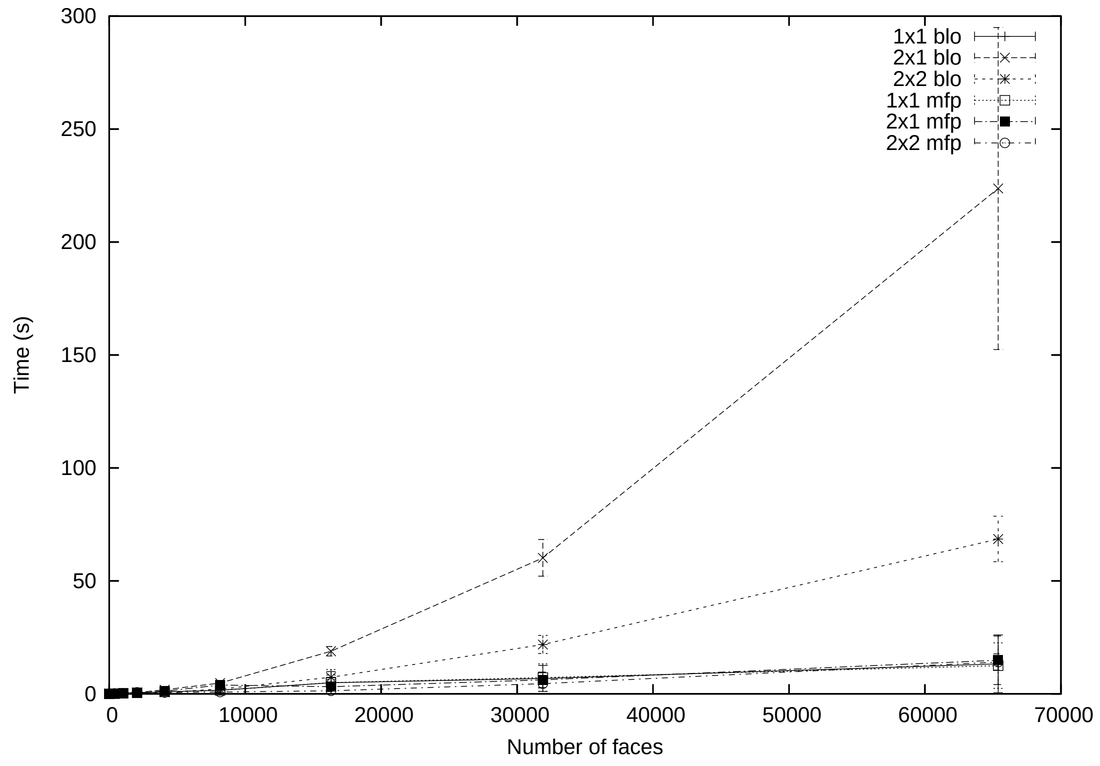

Figure C.14: Maximum Thread Algorithm Time, MPI Lanthier, 4-way SMP, Unsorted Queue

C.2 Algorithm time vs number of faces, per system, processor configuration, queue

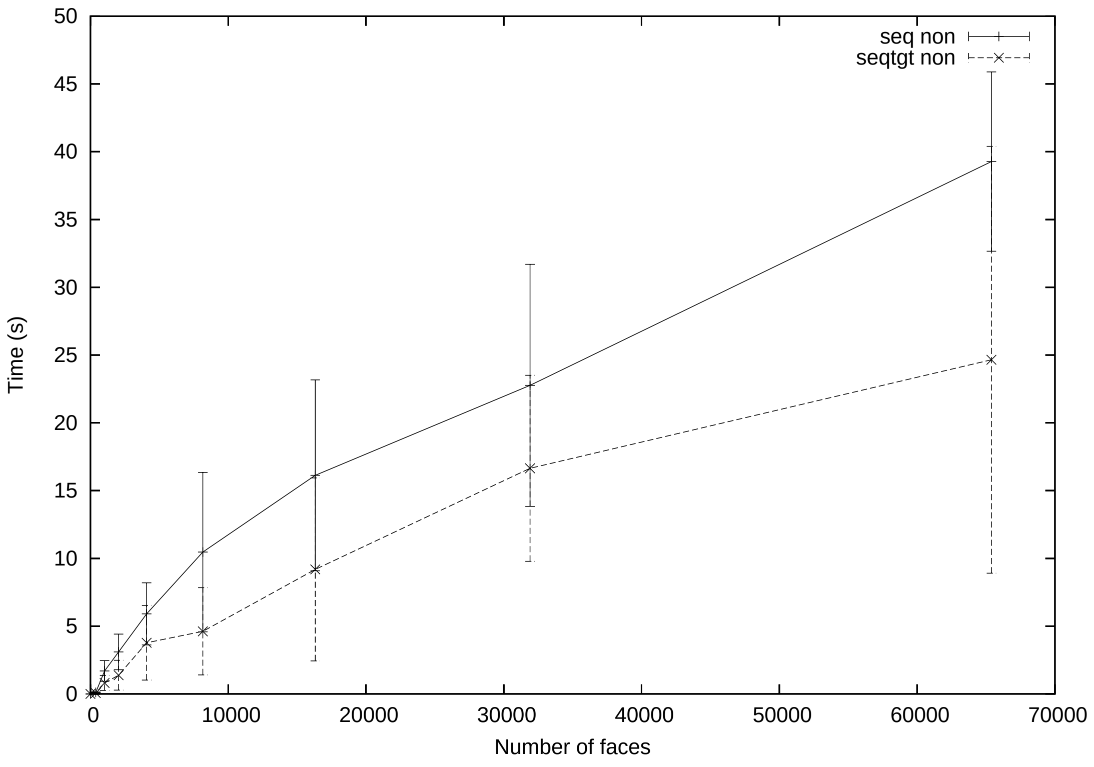

Figure C.15: Maximum Thread Algorithm Time, Uniprocessor, 1x1, Sorted Queue

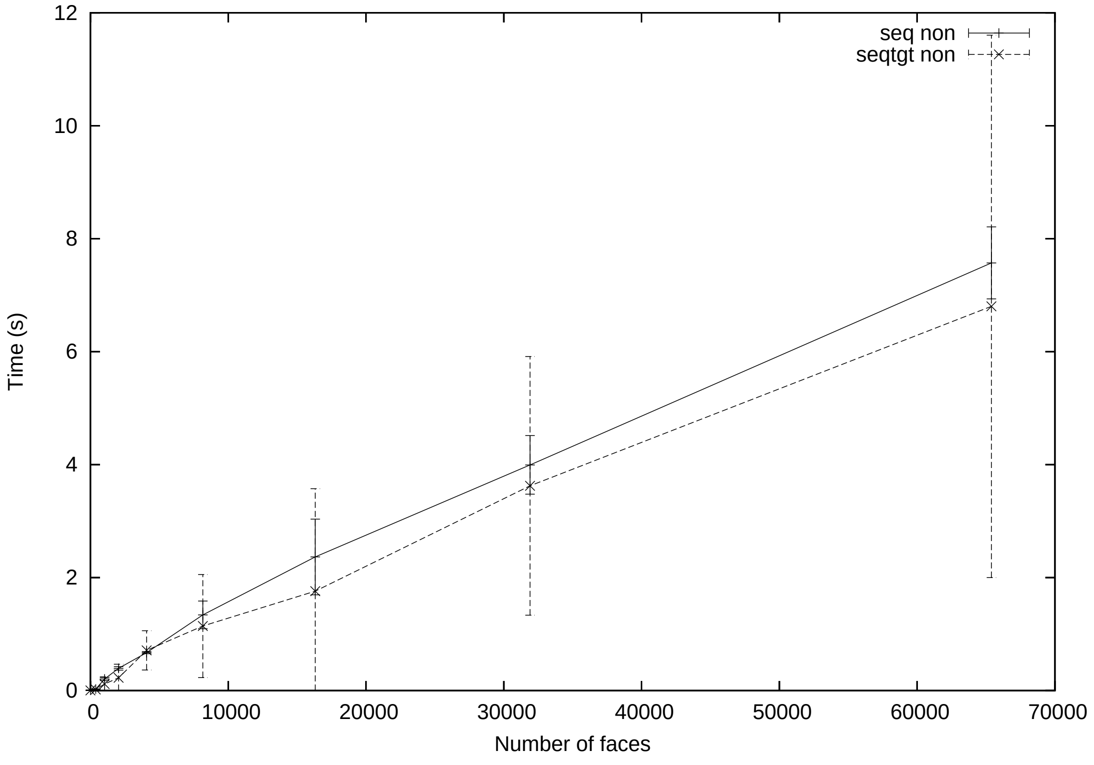

Figure C.16: Maximum Thread Algorithm Time, Uniprocessor, 1x1, Unsorted Queue

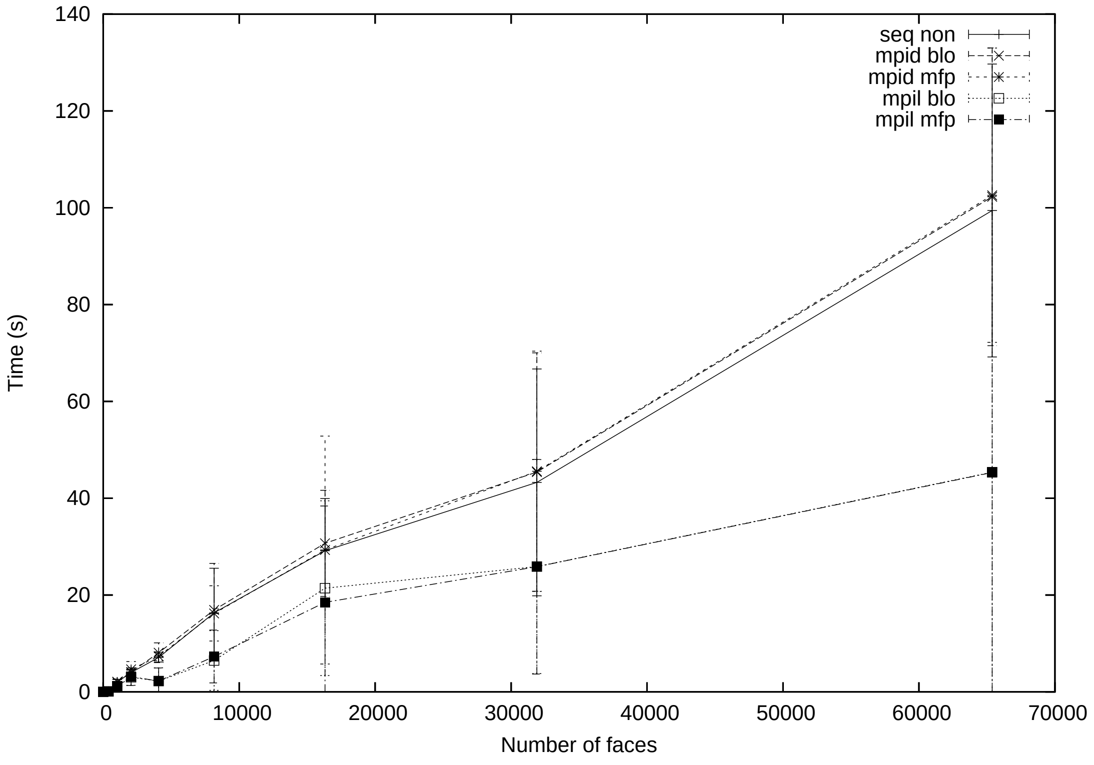

Figure C.17: Maximum Thread Algorithm Time, 16-node Beowulf, 1x1, Sorted Queue

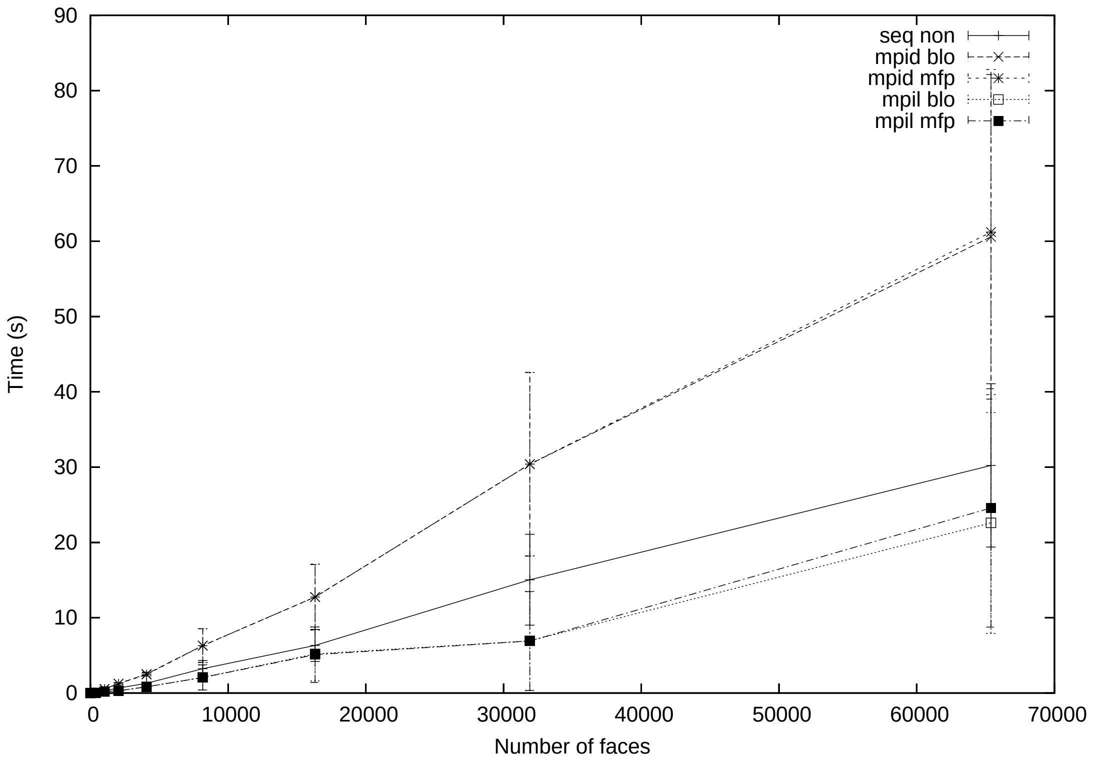

Figure C.18: Maximum Thread Algorithm Time, 16-node Beowulf, 1x1, Unsorted Queue

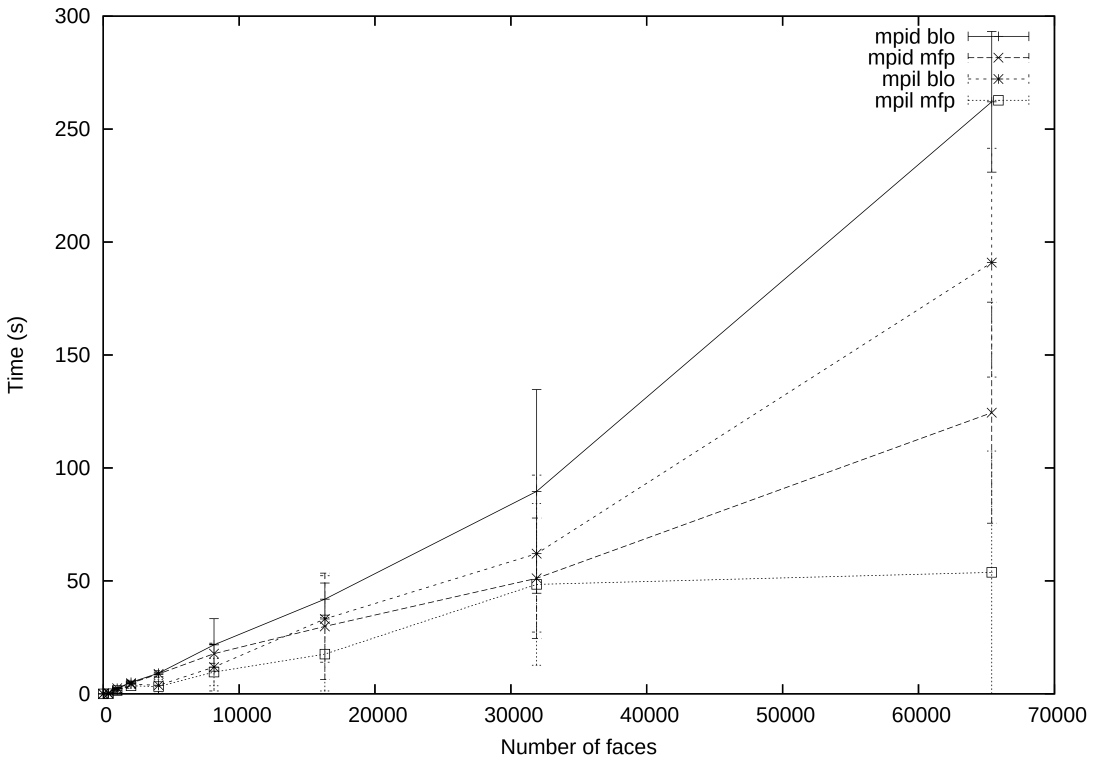

Figure C.19: Maximum Thread Algorithm Time, 16-node Beowulf, 2x1, Sorted Queue

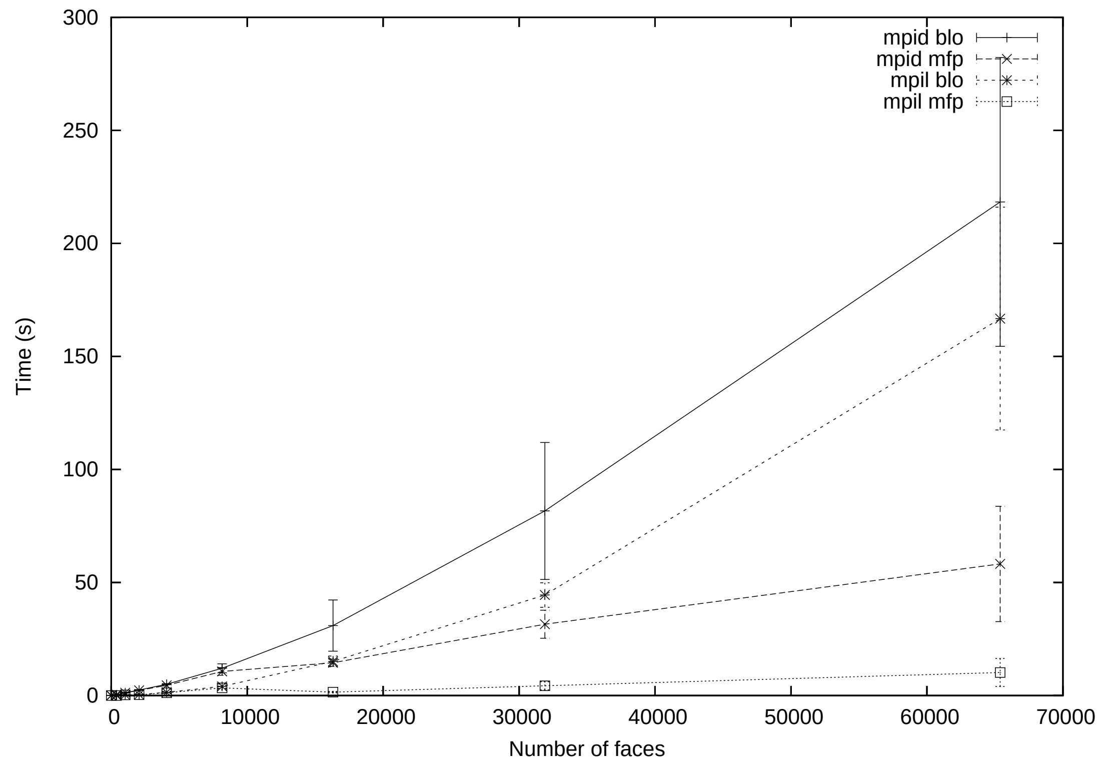

Figure C.20: Maximum Thread Algorithm Time, 16-node Beowulf, 2x1, Unsorted Queue

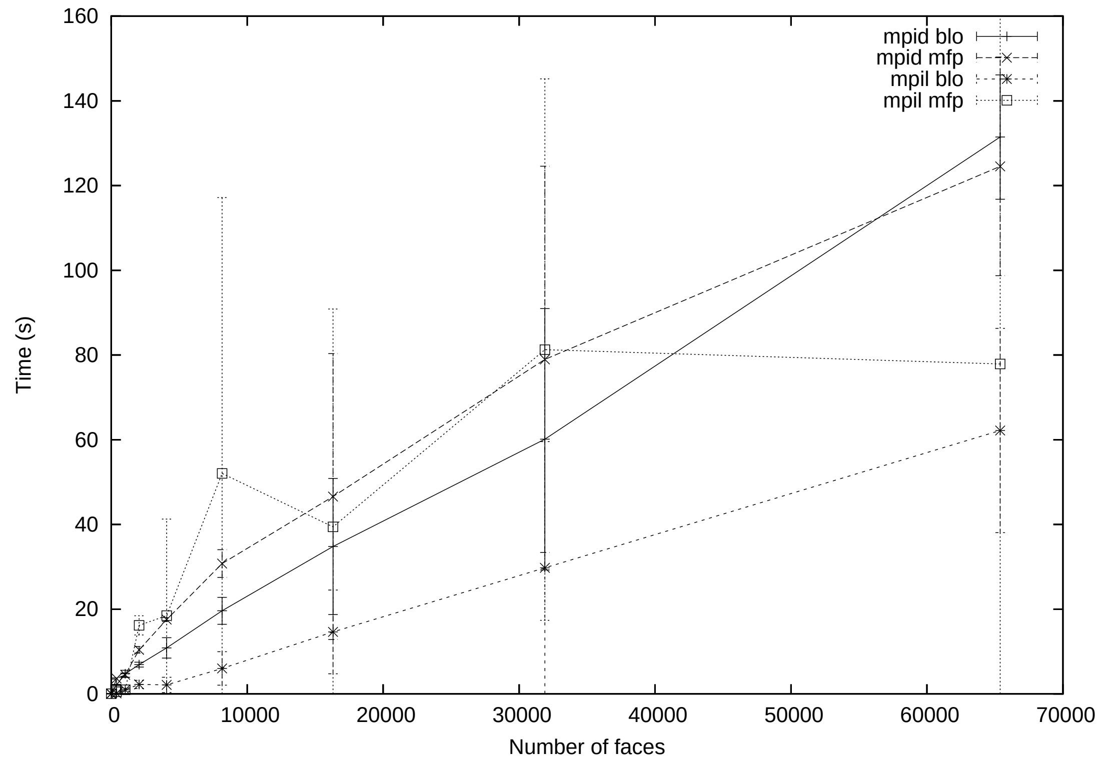

Figure C.21: Maximum Thread Algorithm Time, 16-node Beowulf, 2x2, Sorted Queue

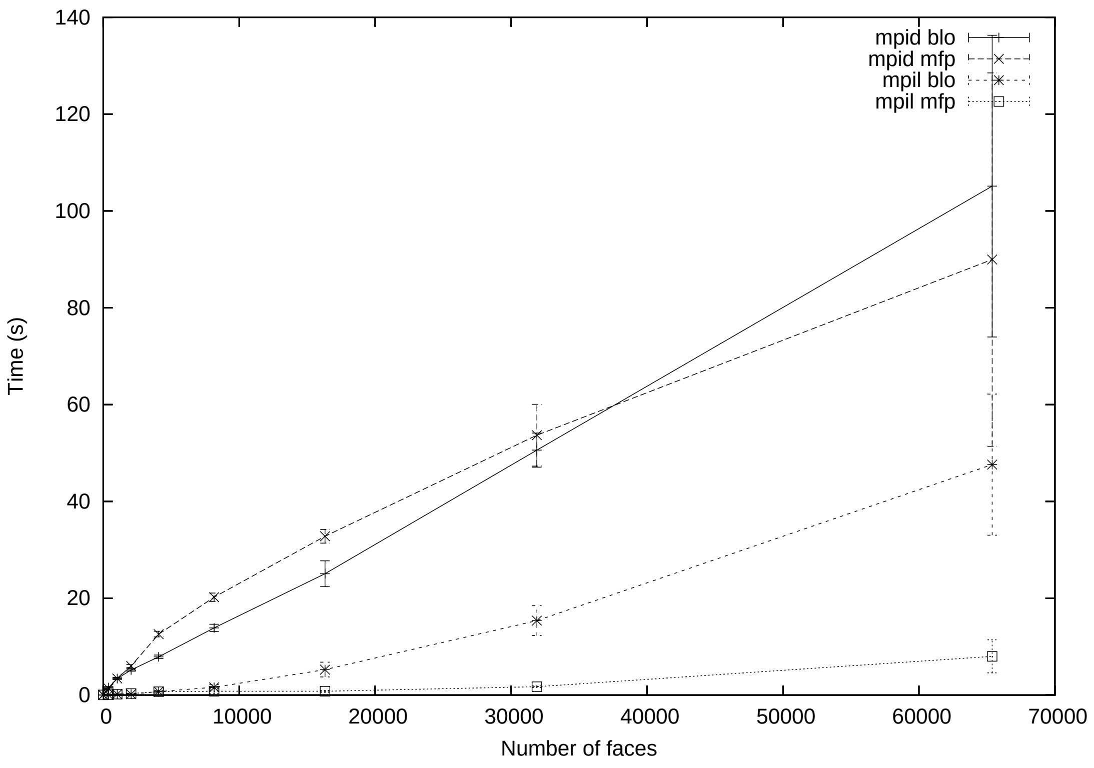

Figure C.22: Maximum Thread Algorithm Time, 16-node Beowulf, 2x2, Unsorted Queue

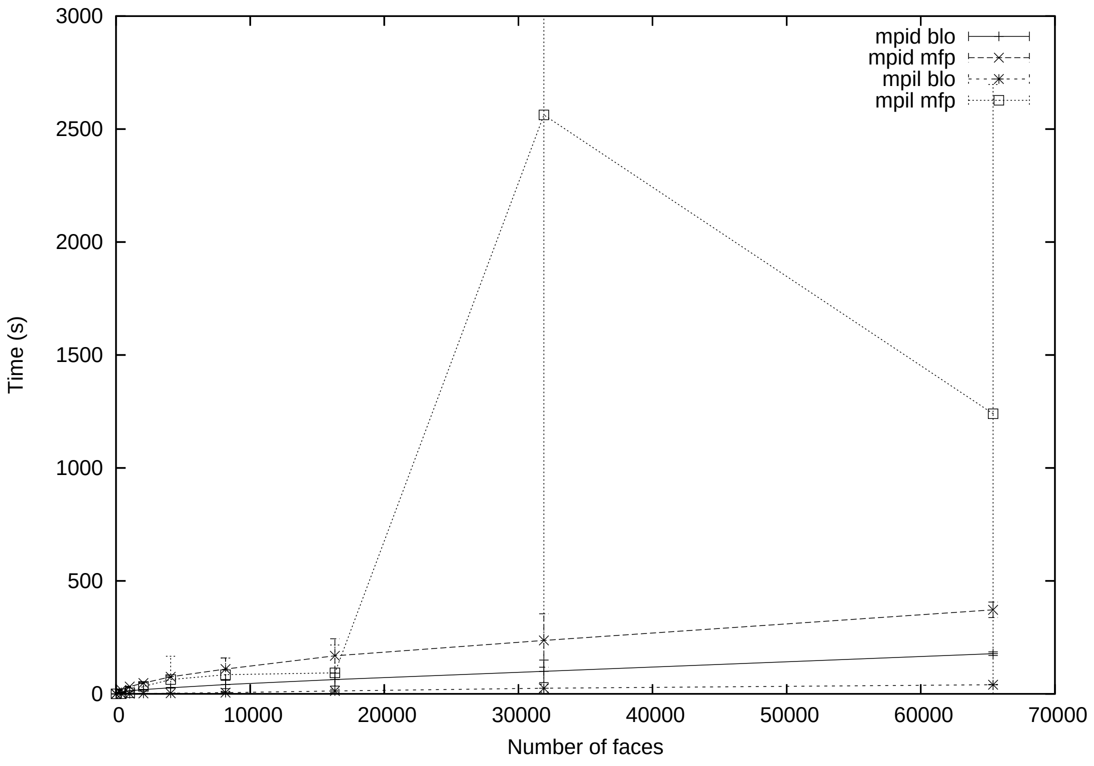

Figure C.23: Maximum Thread Algorithm Time, 16-node Beowulf, 3x3, Sorted Queue

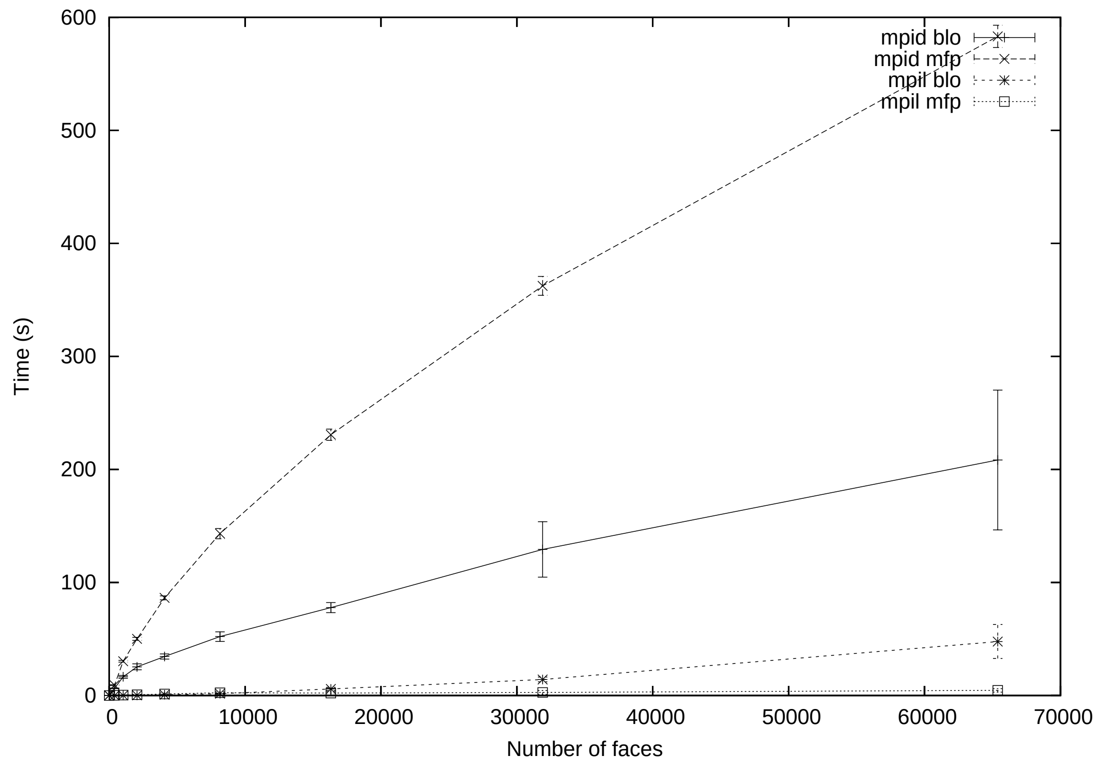

Figure C.24: Maximum Thread Algorithm Time, 16-node Beowulf, 3x3, Unsorted Queue

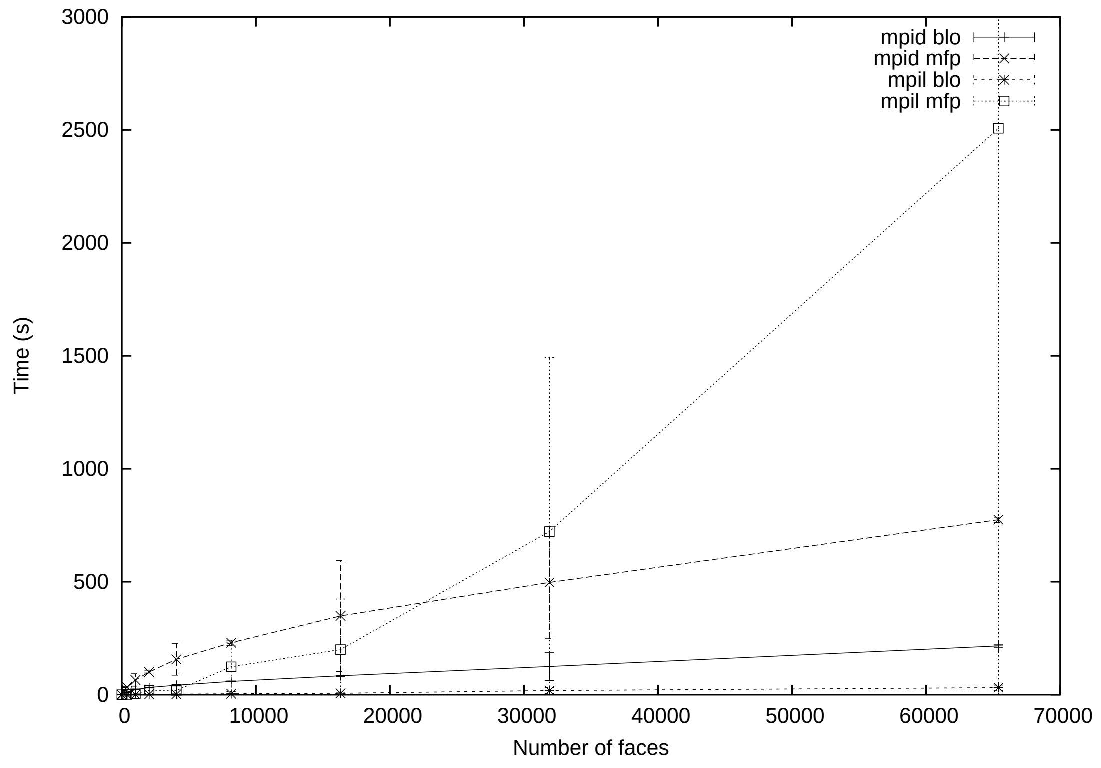

Figure C.25: Maximum Thread Algorithm Time, 16-node Beowulf, 4x4, Sorted Queue

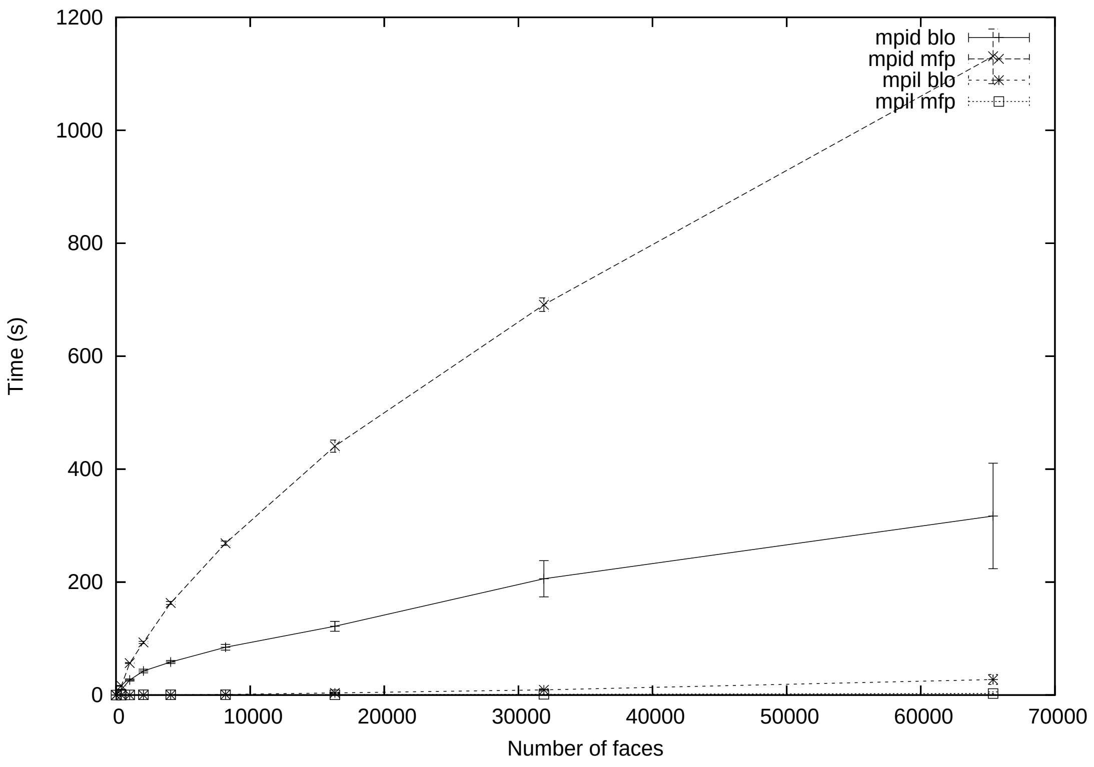

Figure C.26: Maximum Thread Algorithm Time, 16-node Beowulf, 4x4, Unsorted Queue

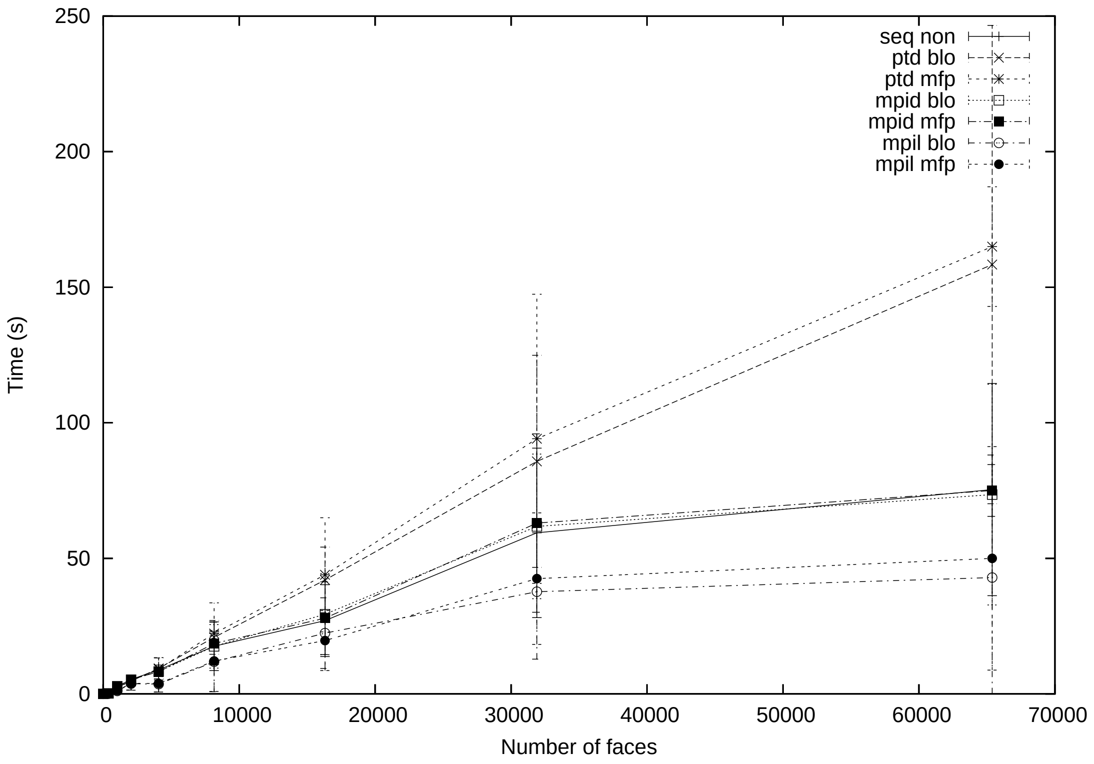

Figure C.27: Maximum Thread Algorithm Time, 4-way SMP, 1x1, Sorted Queue

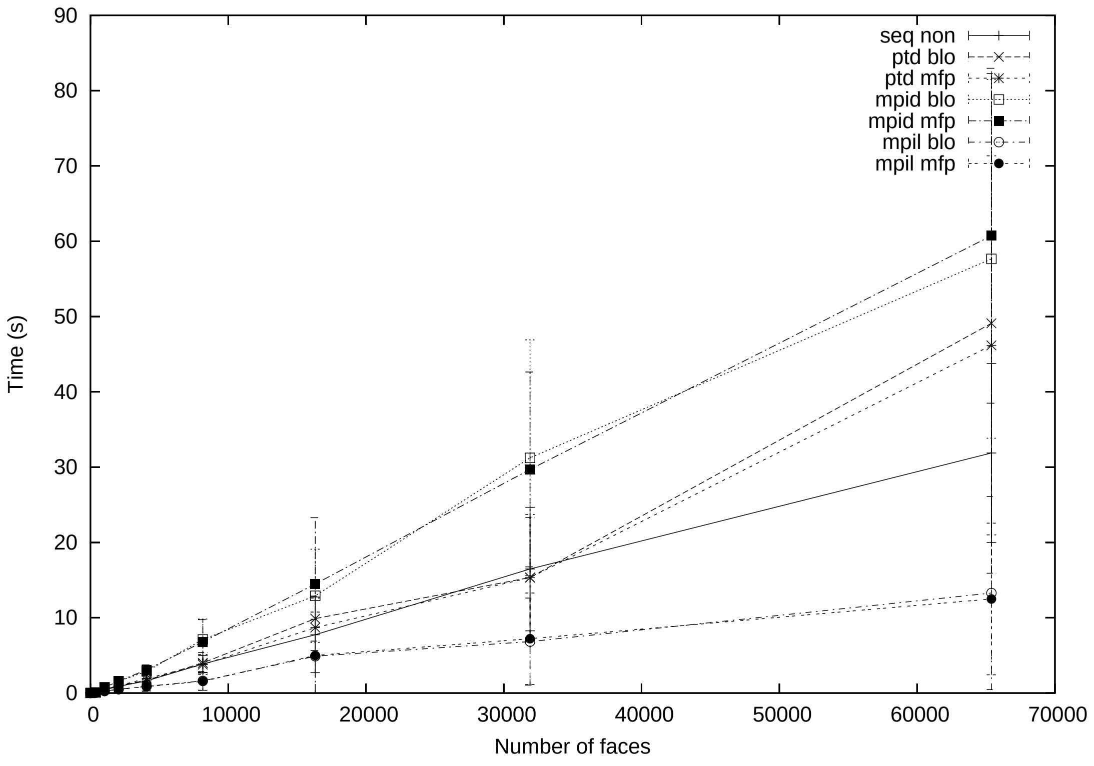

Figure C.28: Maximum Thread Algorithm Time, 4-way SMP, 1x1, Unsorted Queue

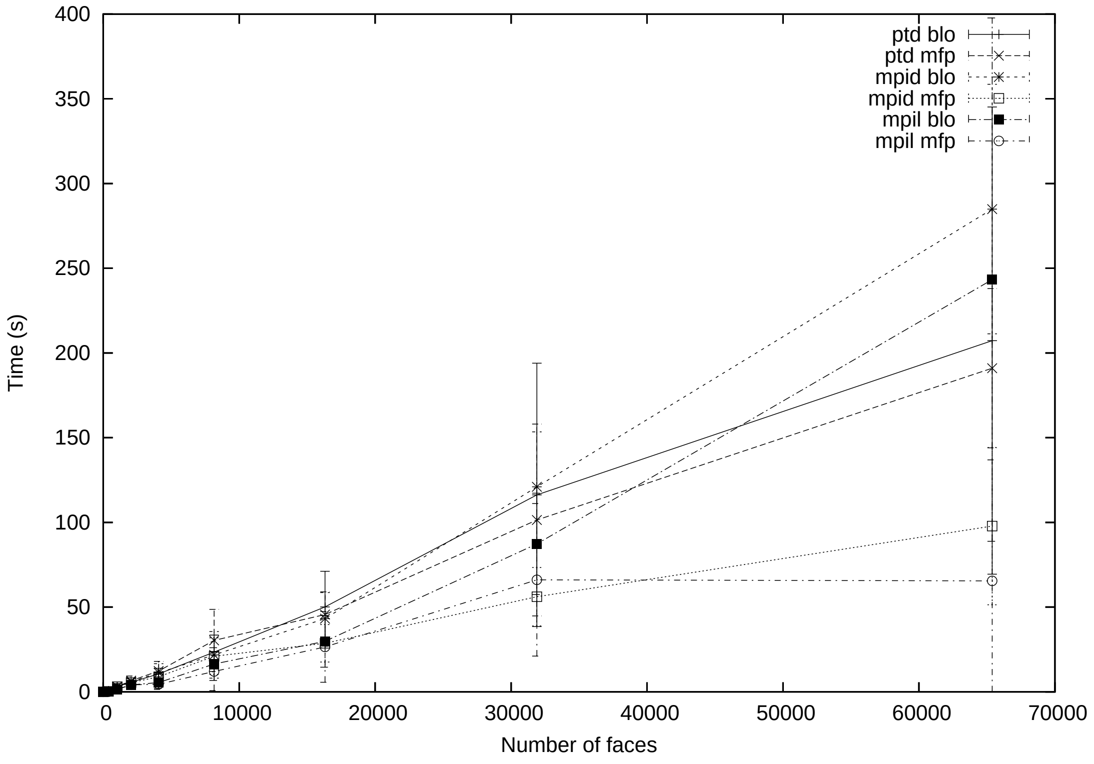

Figure C.29: Maximum Thread Algorithm Time, 4-way SMP, 2x1, Sorted Queue

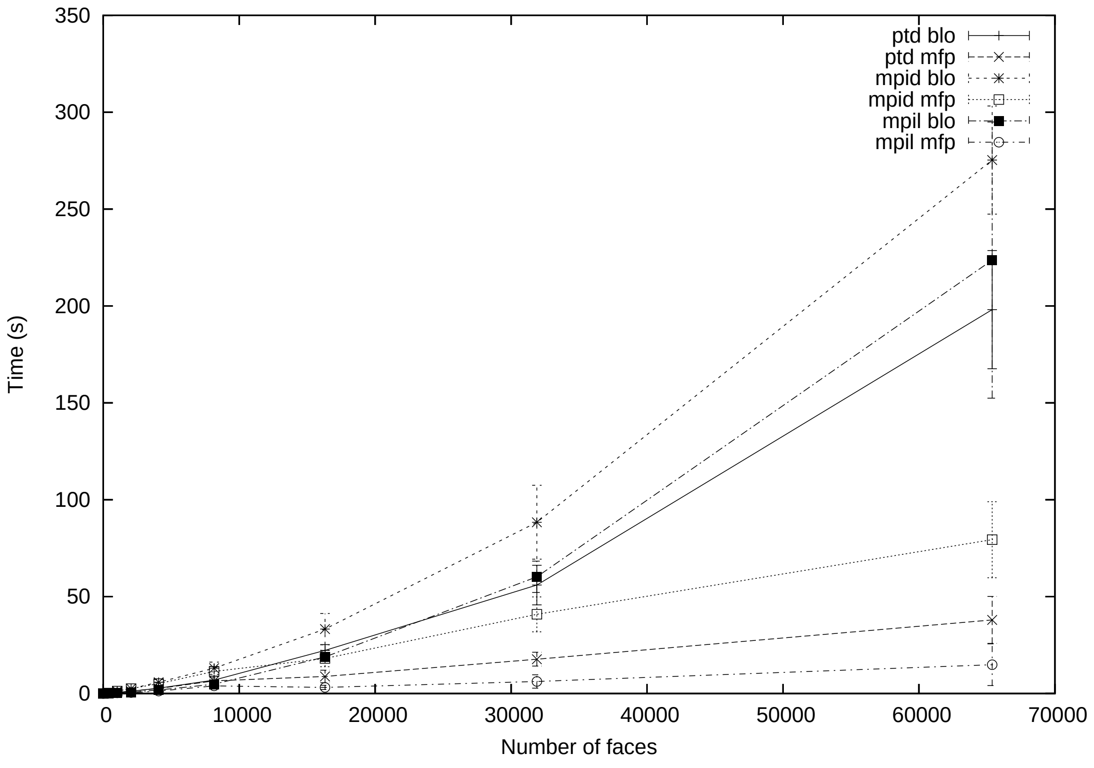

Figure C.30: Maximum Thread Algorithm Time, 4-way SMP, 2x1, Unsorted Queue

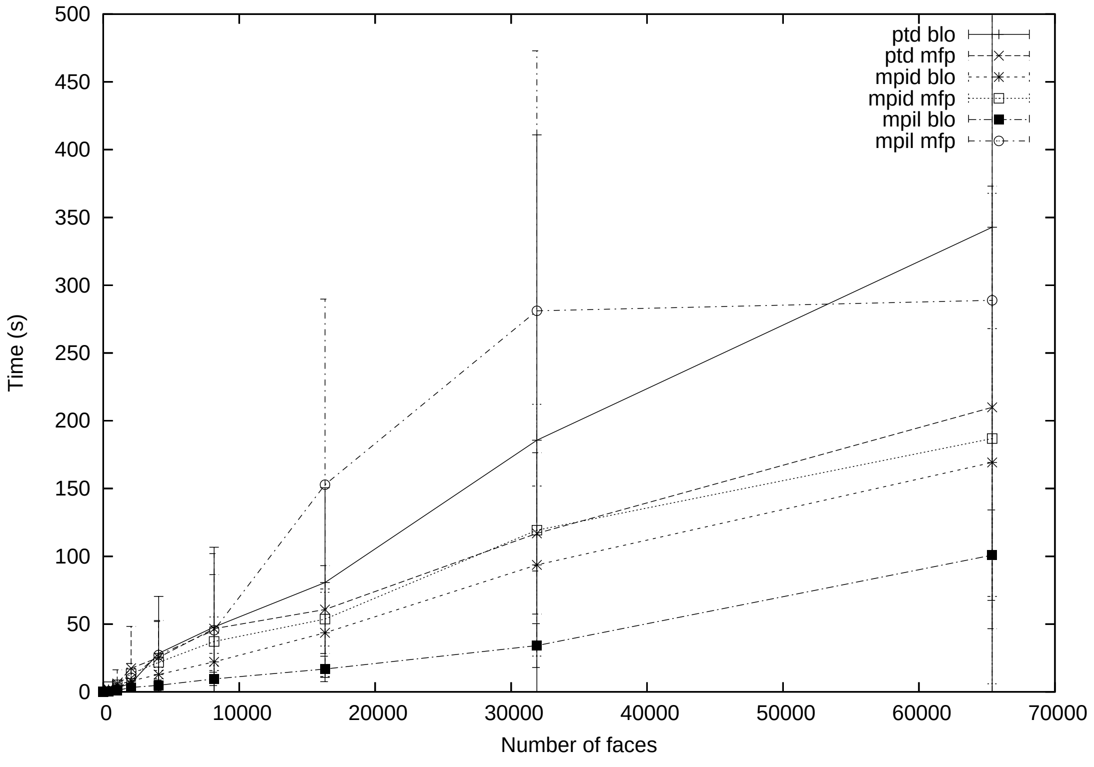

Figure C.31: Maximum Thread Algorithm Time, 4-way SMP, 2x2, Sorted Queue

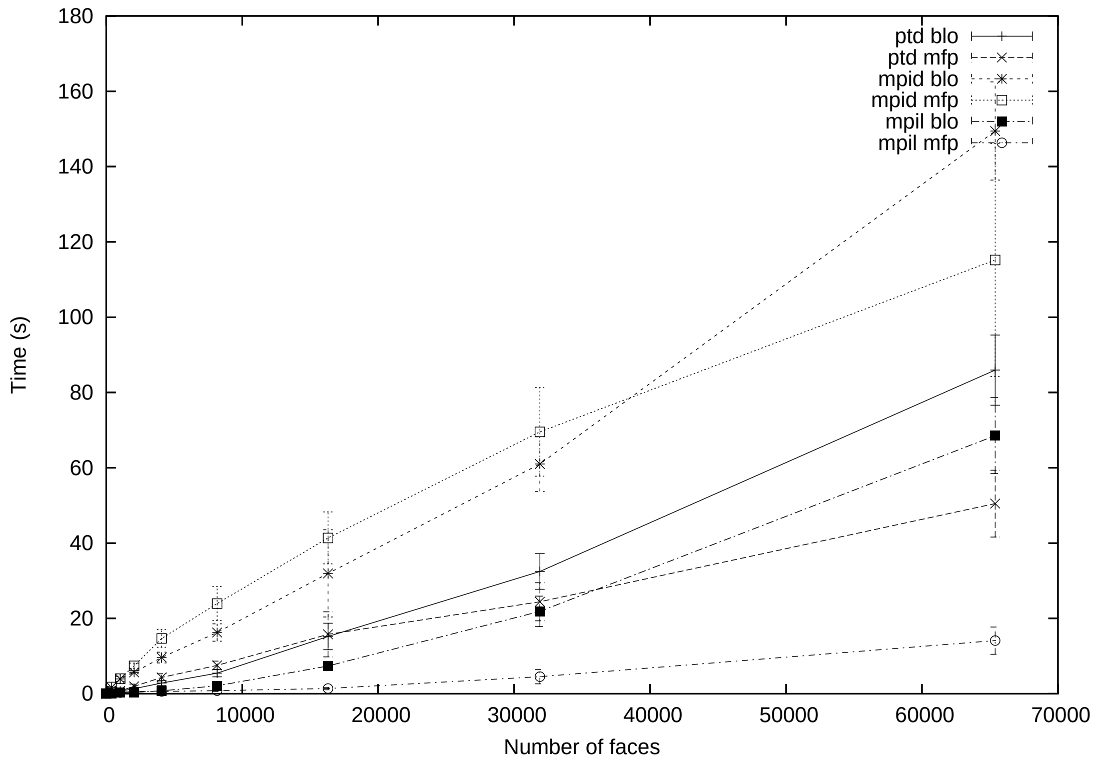

Figure C.32: Maximum Thread Algorithm Time, 4-way SMP, 2x2, Unsorted Queue