Parallel implementation of geometric shortest path algorithms*

Mark Lanthier Doron Nussbaum Jörg-Rüdiger Sack

Abstract

In application areas such as GIS, the Euclidean metric is often less meaningfully applied to determine a shortest path than metrics which capture, through weights, the varying nature of the terrain (e.g., water, rock, forest). Considering weighted metrics however increases the run-time of algorithms considerably suggesting the use of a parallel approach. In this paper, we provide a parallel implementation of shortest path algorithms for the Euclidean and weighted metrics on triangular irregular networks (i.e., a triangulated point set in which each point has an associated height value). To the best of our knowledge, this is the first parallel implementation of shortest path problems in these metrics. We provide a detailed discussion of the algorithmic issues and the factors related to data, machine, implementation determining the performance of parallel shortest path algorithms. We describe our parallel algorithm for weighted shortest paths, its implementation and performance for single-source and multiple-source instances. Our experiments were performed on standard architectures with different communication/computation characteristics, including PCs interconnected by a cross-bar switch using fast ethernet, a state-of-the-art Beowulf cluster with gigabit interconnect and a shared-memory architecture, SunFire.

1 Introduction

Shortest path problems are among the fundamental problems studied in computational geometry and other areas including graph algorithms, geographical information systems (GIS) and robotics. Most of the existing algorithms for realistic shortest path problems are either very complex and/or have very large time and space complexities. Hence they are unappealing to practitioners and pose a challenge to theoreticians. This coupled with the fact that geographic models are approximations of reality anyway and high-quality paths are favored over optimal paths that are "hard" to compute, approximation algorithms are suitable and necessary. In our ongoing pursuit to tackle these problems we have proposed simple and efficient sequential algorithms, that approximate the cost of the shortest (weighted and unweighted) path within an additive or multiplicative factor, see [5, 6, 7, 24, 25]. Their possible practicality has been demonstrated in [25]. In GIS, usually the instance of the given geometric configuration is in the gigabytes to terabytes range and hence even efficient sequential algorithms, in reality, will run extremely slow due to actual machine parameters. For example, the size of main memory (caches) is typically a few hundred megabytes (up to gigabyte range). Due to virtual memory effects and absence of external memory aware algorithms for any of these problems, the actual run time is orders of magnitude slower than on smaller problem instances that fit into main memory.

We now formally state the problems studied. Let s and t be two vertices on a given terrain (or possibly non-convex polyhedron) $\mathcal{P}$, in $\Re^3$, consisting of n triangular faces on its boundary, each face, f_i has an associated weight $w_i > 0$. A Euclidean *shortest path* $\pi(s,t)$ between s and t is defined to be a path with minimum Euclidean length among all possible paths joining s and t that lie on the surface of $\mathcal{P}$. A *weighted shortest path* $\Pi(s,t)$ between s and t is defined to be a path with minimum cost among all possible paths joining s and t that lie on the surface of $\mathcal{P}$. The *cost* of a path is $\sum d_i w_i$ where d_i is the length of the path through face f_i. The weighted shortest path problem is computationally harder than the Euclidean instance; e.g., the first algorithm (see Section 2) for weighted shortest paths even in $2-d$ had a time complexity proportionally to n^8 (additional terms are ignored here).

*Research supported by NSERC, NCE GEOIDE and SUN Microsystems. School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, Ontario K1S5B6, Canada. {lanthier,nussbaum,sack}@scs.carleton.ca

The high time-complexities and the large problem-sizes motivate the study of parallel algorithms for shortest paths, in particular for weighted shortest paths computations. In terms of scalability, the shortest path problem itself appears to be inherently sequential. Adamson and Tick [1] state that no known one-to-all graph shortest path algorithms are scalable. Thus the aim of most implementations is to achieve a "reasonable" efficiency - namely, efficiency levels of 25%-60% with 2-16 processors has been achieved. The goal in this research is to investigate the various factors that influence scalability for geometric shortest path problems and to design, implement, and experimentally study an approximation algorithm for computing a weighted shortest path $\Pi'(s,t)$ between two points s and t on the surface of a polyhedron $\mathcal{P}$, and demonstrate its practicality. We study both the single-source and multiple-source settings. Our parallel algorithm is based on a new, efficient way to partition the data in a manner that lends itself well to solving the class of propagation-type problems which include the shortest path problem. (Propagation-type problems are problems on images/maps (raster or vector) where events (e.g., fires, diseases) start at one or more location(s) and then propagate (spread) through the image according to some propagation function.)

The remainder of this paper is organized as follows. In Section 2 we give a brief survey of previous work on shortest path problems in sequential and parallel settings. Therein, we discus how our work compares to the previous work in terms of the performance-related factors. Following this, in Section 3, we describe our algorithm in detail. Then in Section 4, we discuss the experiments that were conducted in order to assess the performance of the algorithm and present the results observed pertaining to particular performance-related factors. Finally, we conclude with Section 5.

2 Related Work and Our Approach:

We briefly discuss here some of the work that has been carried out for solving shortest path problems in different scenarios. We then discuss the performance factors of parallel shortest path algorithms and their implementations.

2.1 Previous Shortest Path Work:

Shortest path problems in computational geometry can be categorized by various factors which include the dimensionality of the space, the type and number of objects or obstacles (e.g., polygonal obstacles), and the distance measure used (e.g., Euclidean or weighted distances). Several research articles, including surveys (see [27, 30, 31]), have been written presenting the state-of-the-art in this active field. Here we discuss those contributions which relate more directly to our work; these are in particular stated in 2 and 3-dimensional weighted scenarios.

In two dimensions several variations of shortest path problems have been studied over the last two decades. This includes computing Euclidean shortest paths and answering shortest path queries between two points inside a simple polygon and amidst polygonal obstacles. Of particular interest is the *weighted region problem* (WRP) introduced in [29]; it is a natural generalization of the shortest path problem in polygonal domains. A planar triangulated subdivision is given consisting of n faces, where each face has a positive non-zero weight. Using an algorithm based on the "continuous Dijkstra's method" Mitchell and Papadimitriou [29] provide an approximation algorithm to compute a (weighted) ε-short path; it runs in $O(n^8 \log(nNW/w\epsilon))$ time using $O(n^4)$ space, where N is the largest integer coordinate of any vertex of the triangulation and W (w) is the maximum (minimum) weight of any face of the triangulation.

Lanthier et al. [25] have described several algorithms for the WRP problem; the cost of the approximation is no more than the shortest path cost plus an (additive) factor of $W|LongestEdge|$, where $LongestEdge$ is the longest edge and W is the largest weight among all faces. As their experimental analysis shows, these algorithms are also of practical value. Aleksandrov et al. provide ϵ-approximation algorithms in $O(\frac{n}{\varepsilon^2} \log n \log \frac{1}{\varepsilon})$ time in [6] and in $O(\frac{n}{\varepsilon} \log \frac{1}{\varepsilon}(\frac{1}{\sqrt{\varepsilon}} + \log n))$ time see [7]. Sun and Reif [37], use Aleksandrov et al.'s discretization scheme [7] in the design of an algorithm whose run time is $O(\frac{n}{\varepsilon} \log \frac{n}{\varepsilon} \log \frac{1}{\varepsilon})$. Most recently Aleksandrov et al. developed an $O(\frac{n}{\sqrt{\varepsilon}} \log \frac{n}{\varepsilon} \log \frac{1}{\varepsilon})$ time algorithm for this problem [8]. While the

earlier work [29] uses $O(n^4)$ space the latter work improves further on the storage requirement over that presented in [25, 7, 37]. More precisely, the number of Steinerpoints and the time complexity is reduced by $\sqrt{\varepsilon}$ over [7, 37]. Assume e.g., that ε is 1/100 then the number of Steinerpoints is reduced by a factor of 1/10 which is significant when considering geographical data where n may represent several million points. (The ϵ-approximation scheme has been implemented and experimentally verified at the Max-Planck Institute for Computer Science, Germany [40].) Lanthier et al. [24] also present approximation algorithms for anisotropic shortest path problems where vehicle characteristics are taken into consideration.

Given a set of pairwise disjoint polyhedra in $\Re^3$ and two points s and t, the problem of computing a shortest path between s and t that avoids the interiors of the polyhedra is NP-Hard [11]. The shortest path problem amidst (disjoint) convex polyhedra can be solved in time exponential in the number of polyhedral objects as was shown by Sharir [36]. For two polyhedral obstacles with a total of n vertices, Baltsan and Sharir [9] presented an $O(n^3 \log n)$ time shortest path algorithm. The NP-hardness and the large time complexities of 3-d shortest paths algorithms even for special problem instances have motivated the search for approximate solutions to shortest path problems. Recently, there have been several results on approximation algorithms for computing Euclidean shortest paths on (convex) polyhedral surfaces. Hershberger and Suri [17] presented a simple linear time algorithm that computes a 2-approximation to the shortest path. Har-Peled et al. [14] extended this result to provide an algorithm to compute an ε-approximation of the shortest path; it runs in $O(n \min\{1/\varepsilon^{1.5}, \log n\} + 1/\varepsilon^{4.5} \log(1/\varepsilon))$ time. Agarwal et al. [3] provided an improved algorithm that runs in $O(n \log \frac{1}{\varepsilon} + \frac{1}{\varepsilon^3})$ time. Har-Peled [15] provided an algorithm to solve a variation of this problem which computes, in $O(\frac{\log n}{\varepsilon^{1.5}} + \frac{1}{\varepsilon^3})$ time, an ε-approximation for a path between two points on the surface of a convex polytope, after a linear time preprocessing. All of these algorithms heavily exploit the properties of convex polyhedra. Varadarajan and Agarwal [4] provided an algorithm that computes a path on a, possibly non-convex, polyhedron that is at most $7(1+\varepsilon)$ times the shortest path length; it runs in $O(n^{5/3} \log^{5/3} n)$ time. They also presented a slightly faster algorithm that returns a path which is at most $15(1+\varepsilon)$ times the shortest path length. Recently, Aleksandrov et al. [7] provided an algorithm for the shortest path problem in a subdivision of $\Re^3$ consisting of n convex regions where each face, f_i is associated with a weight, $w_i > 0$. They provided the first polynomial time approximation scheme; it runs in $O(\frac{n}{\varepsilon^3} \log \frac{1}{\varepsilon}(\frac{1}{\sqrt{\varepsilon}} + \log n))$ time. Moreover, the algorithms of [25] apply for Euclidean shortest path problems on polyhedral surfaces as well. We note that the Chen and Han's algorithm [12] can compute an exact Euclidean shortest path between two points on a polyhedral surface in $O(n^2)$ time and very recently Kapoor [23] has proposed an $O(n \log^2 n)$ time algorithm.

The parallel algorithm proposed here attempts to maintain the simplicity of our sequential algorithms while providing a decrease in the running time as well as allowing larger terrains to be processed. The problem is again reduced to solving the shortest path between vertices of a graph. Although there already exist algorithms for computing shortest paths in graphs in parallel, these algorithms often assume a large number of processors (e.g., $O(n)$ [35]) and sometimes assume constraints on the data such as being evenly distributed [1, 10, 38] and/or being sparse [1]. More recent work has aimed at providing implementations of distributed algorithms that obtain "reasonable" performance and in analyzing the factors that affect the performance [1, 10, 38].

We illustrate how our algorithm's performance varies with some of these factors as well. We use a general spatial indexing storage structure, called *Multidimensional Fixed Partition (MFP)* described in [32]). The structure implicitly achieves a form of load balancing as well as allows the processor idle time to be reduced in cases where the terrain has dense clusters of data. In addition, the structure can also be used for other types of propagation applications besides shortest path such as visibility, weighted Voronoi diagrams etc. (A Voronoi diagram of a point set S is a partitioning of the plane in which each point in S is associated with the region of points to which it is the nearest neighbor among all points in S.)

2.2 Performance Factors of Parallel Algorithms:

The runtime performance of a parallel algorithm depends on a variety of factors determined by the algorithm, data, machine and implementation. We will discuss some of these factors as well as the relevant literature in

this section. The number of factors and their interaction make it difficult to perform a thorough comparison and isolate the impact of each factor on the overall performance. Much of the existing research has been geared towards analyzing the effects of one or two factors in isolation. To achieve this, various assumptions are often made on the other factors. For example, one study may examine the effects of algorithmic factors while making assumptions on data and machine related factors such as assuming efficient load balancing on a specific machine architecture. Another study may assume a large number of processors (i.e., $O(n)$) and a very specific architecture to focus exclusively on different partitioning strategies. As a result, it is difficult to compare results of one study to those from another.

Before discussing the performance factors, we should indicate that our algorithm uses the Multiple Instruction Multiple Data (MIMD) model with distributed memory. Our processors also use asynchronous communication, in that they each compute at their own rate on a different portion of the terrain data.

2.2.1 Algorithmic Factors:

- **Shortest path paradigm**: Much of the published results on parallel distributed shortest paths are aimed at the one-to-all (also few-to-all) shortest path problem (see [1, 10, 20, 21, 35, 38]). A study by Gallo and Pallottino [13] has shown that the most efficient label-correcting[1] algorithms are faster than the most efficient label-setting[2] methods in the serial case for the one-to-all problem when small, sparse graphs are used. Note that graphs with an average node degree of six or less are typically considered to be sparse. Bertsekas et al. [10] have suggested that the one-to-all label-correcting algorithms for sparse graphs also provides an advantage in the parallel shared memory setting. They do state however that when only a few destinations are used (i.e., one-to-one), then the label-setting algorithms are better suited (see also [16, 34]).

  In contrast, Hribar et al. [18] have shown that the label-correcting algorithms have inconsistent performance for their networks. They state that there is no one shortest path algorithm that is best for all shortest path problems, nor for problems with similar data set sizes. They conclude that generalizations cannot be made about the best algorithm for all networks. They do however, attempt to give an indicator where the label-setting algorithms outperform label-correcting algorithms which is based on the expected number of iterations per node. In a later study by Hribar et al. [20], they give experiments that demonstrate that label-setting algorithms perform best for distributed memory parallel machines when large data sets (16k - 66k vertices) are used. They experimented with five transportation networks corresponding to actual traffic networks as well as random grid networks.

- **Shortest path queue distribution**: Besides the categories of label-correcting and label-setting, there are variations of each that differ in the number and size of queues, and in the order that items are removed. Bertsekas et al. [10] experimented with single vs. multiple queues for the label-correcting algorithms in the shared memory setting. Their multiple queue version assigns a local queue to each processor, however, the data in the queue has no particular relationship (such as spatial) with that processor. They show that the multiple queue strategy is better, due to the reduction in queue contention. Their experiments also investigated synchronous vs. asynchronous algorithms and their results show that the asynchronous algorithms always outperformed the synchronous ones. Träff [38] has compared a synchronous strategy with an asynchronous strategy using a label-setting algorithm and he indicates that synchronous strategies will perform poorly in systems where there the cost of communication is high.

  Adamson and Tick [1] compared five algorithms that use a single queue for all processors which is either sorted or unsorted. They compare partitioned and non-partitioned algorithms. The partitioned algorithms assign vertices to each processor according to some modulus mapping function, whereas the unpartitioned version assigns vertices to processors arbitrarily. They show that the unsorted queue

[1] Bellman-Ford algorithm is a label-correcting algorithm. While it takes constant time per iteration, the number of iterations may vary.

[2] Dijkstra's algorithm is a label-setting algorithm; it has fixed number of iterations.

works best for the unpartitioned data with a small number of processors and that the sorted queues are better for the partitioned data when a higher number of processors are used. They have two varying parameters of granularity and eagerness which specify the size of the partitions and frequency of queue extraction requests, respectively. They were able to obtain a best efficiency of 44% with 16 processors.

- **Communication frequency**: The number and frequency of communication steps that each processor performs throughout the algorithm execution is also an important performance issue. Some algorithms (e.g., [20]) allow each processor to empty out their local queues as a common step and then all boundary node information is communicated to all adjacent processors. A different strategy would be to communicate shared information as soon as it is available (e.g., [32, 38]). The advantage of the first strategy is that only a few large messages need to be communicated, whereas the second strategy would have many more smaller messages being communicated which could cause runtime delays. The use of few large messages is a standard approach when the bulk-synchronous processing (Valiant [39]) model of computation is used. However, when data is unbalanced among processors, the first approach may have some processors sitting idle while other are still emptying their queue. In addition, much of the work done by a processor may necessarily be discarded as a result of new information coming from its adjacent processor after its queue has been emptied. A further disadvantage of the first approach is that it requires synchronization which can lead to a performance decrease as shown by Träff [38].

- **Termination detection**: One algorithmic factor in parallel shortest path computations that is often overlooked is that of *termination detection frequency*. This defines the amount of times that the algorithm checks for termination. With a high frequency detection, a processor will spend much of its time checking whether or not the algorithm has terminated, whereas a low detection frequency may check less but processors may do more computational work than is necessary. Hribar and Taylor [21] analyzed the impact of the detection frequency on the performance of a synchronous label-correcting shortest path algorithm. Their analysis indicates that the optimal frequency for detecting termination depends on the number of processors and the size of the problem. They state that high detection frequencies are best when a large number of processors are used and low frequency otherwise. To validate their claim, they tested with 4 variations of termination detection which varied with respect to the time interval between communication steps as well as the frequency of synchronization.

- **Cost function**: The final algorithm-related factor that we will discuss is that of the cost function. The cost function used in the shortest path calculation can also affect the performance. If a simple cost function is used such as the Euclidean metric, then little time is spent on computation and a higher percentage of time is spent on communication. When a more computationally-intensive cost function is used, such as anisotropic metrics, then the percentage of compute time increases, which often leads to more efficient use of the processors. Most of the existing research has focussed on weighted graphs with no discussion as to how the weights were obtained. To our knowledge, there has not been any papers that analyzed the effects of the cost function.

2.2.2 Data-Related Factors:

We now turn our discussion towards data-related factors. Clearly, the amount of data is of paramount importance since there is a strong correlation between the size of the data sets and the overall performance. Most researchers provide a comparison of different data set sizes in order to better understand how the algorithm will scale with large amounts of data in comparison to small amounts of data. Adamson and Tick [1] state that the amount of parallelism is dependent on the average degree (connectivity) of the graph. Most of the research has concentrated on sparse graphs (e.g., transportation networks) where the average degree is about six. In practice, it is important to choose the "right" algorithm that has a good performance for the type of data being used.

The distribution of the data plays another key role in the performance. Different data partitioning approaches have been proposed, and they differ in the number of partitions made, the sizes of the partitions,

the amount of shared data between adjacent processors and the topological order of partitions (i.e., if they are recursively partitioned). Hribar et al. [19] state that determining good decompositions is crucial for all parallel applications. For their algorithm, as applied to transportation networks, they show that the best decomposition minimizes the number of connected components, diameter and number of partition boundaries, but maximizes the number of boundary nodes. Hribar et al. [20] have shown for transportation network graphs that distributing the data among the processors has roughly half the execution time as replicating the graph on all processors. Even with distributed data, if the amount of work (i.e., data) given to one processor, p_i, is significantly less than that of another processor, p_j, then processor p_i will most likely sit idle while the processor p_j will be busy processing the data. Basically, the more time a processor spends idle, the less efficient a parallel algorithm will be.

Since the shortest path problem on terrains requires traversal between sleeves of faces, our partitioning strategy attempts to maintains the spatial relationship among the data in that connected groups of faces are distributed to each processor. Thus, a significant amount of processing can be accomplished before requiring communication with another processor.

The relative locations of sources and targets is another factor of significant importance. If the source and target points, and the path between them lie on one processor, then the algorithm may not use more than one processor and there will be no benefits from the parallelization. If they are far away on different processors, then more processors get involved in the computation and speedup should be noticeable. Of course, when they lie in different processors, there is communication required. There has not been much research in determining the effects of source/target locations. Instead, most of the previous experimental work has concentrated on the one-to-all and all-to-all problems.

2.2.3 Machine-Related Factors:

One of the reasons why it is often difficult to compare and contrast results from different researchers is that the machine architecture is never quite the same. The machines differ in the number of nodes (e.g., fine or coarse grained), the type and speed of processors (e.g., Power PC, Sparc, Pentium), the interconnection topology (e.g., ring, mesh, hypercube, network of workstations), the interconnection strategy (e.g., ethernet (fast/slow), crossbar, dedicated links), the amount of internal and virtual memory per processor (including levels of caches) and more importantly, the accessibility of memory to each processor (i.e., shared or distributed memory). All of these differences can significantly affect the runtime performance of any parallel algorithm. (For a discussion of PRAM algorithms see e.g., the recent work of [28] which presents a study of the average-case complexity of the parallel single-source shortest-path (SSSP) problem on a PRAM.

One of the significant differences between machines is that of the accessibility of memory to each processor. For example, some algorithms are meant to be run on a shared memory machine (see [1, 2, 10]) in which all processors have access to a common memory. This is useful if all processors need to access common data structures such as a priority queue used in label setting algorithms. The other choice is to have a distributed memory machine (see [38, 19, 21]) in which each processor has its own local memory. Any processor requiring access to memory or data from another processor must do so through message-passing.

2.2.4 Implementation-Related Factors:

The last set of factors to be examined are implementation-specific factors. For example, two programs implemented by different people are likely to have variations in styles and efficiency, even at the geometric primitive level such as calculating the slope of a line or intersecting two line segments. There are issues such as numerical stability and correctness that can be related to performance as well. In addition to the programmers' implementation discrepancies, there are efficient and inefficient implementations of libraries from vendors. For message passing strategies alone, there are choices as to which libraries to use such as Trollius, MPI, MPICH and PVM. In addition, there are different vendors that produce versions with their own characteristics which can lead to versions which are slower than others. Hence, the use of software from one vendor to another can lead to significant fluctuations in runtime performance. Moreover, the cost of

communication versus cost of computation can be affected by the choice of libraries. We have chosen MPI for our implementation.

3 The Parallel Algorithm

Our parallel shortest path algorithm is designed for a distributed memory MIMD architecture. In order to maximize the utilization of the processors the algorithm distributes its data among the processors and allows each processor to process the data asynchronously. We discuss the data partitioning and data assignments (data mapping) in Section 3.1.

The asynchronous operation entails that each processor processes the data independently from other processors as fast as possible. Messages are exchanged between processors as needed in order to ensure that data processing can continue. During propagation, the active border for a given processor will often reach its partition boundary. (The active border is the set of points currently being worked on). At this time, the cost of some node v_i that is shared with its neighbor partition is updated. This adjacent processor obtains the updated cost for v_i and continues processing within its partition. Eventually, each processor will have exhausted its priority queue and the algorithm halts. At this point, a target query can be presented and the resulting path can be computed. In fact, the implementation provides a mechanism for detecting when the target(s) has been reached and it allows the simulation to halt in that case. The simulation itself can be broken down into three steps: preprocessing, executing a shortest path algorithm (Dijkstra's algorithm), tracing back the path. Each of these steps is explained in more detail in the subsections to follow.

Our algorithm implementation is based on Dijkstra's shortest path algorithm. In the sequential version, when the processor exhausts its local queue or it processed the target, the processor can conclude that processing is complete. In the parallel version, when processor p_i processed the target it cannot conclude termination because there may still be another processor that is currently working on another (possibly shorter) path.

Therefore, in this case, global cooperation is required to determine whether the algorithm has terminated. Since our algorithm is asynchronous it cannot update and access global data. To overcome this difficulty, the algorithm stores all required global data implicitly in each processor and uses global tokens to update global data as needed. This prevents hot-spot contention during access of data.

Our parallel algorithms for weighted terrains are based on the ideas proposed by Lanthier et. al in [25]. Their approach to solving the weighted shortest path problem is to discretize the input polyhedron in a natural way, by placing new points, called *Steiner* points, along the edges of the polyhedron. They construct a graph G containing the Steiner points as vertices and edges as the interconnections between Steiner points that correspond to segments which lie completely in the triangular faces of the polyhedron. The geometric shortest path problem on polyhedra is thus stated as a graph problem so that the existing efficient algorithms and their implementations for shortest paths in graphs can be used. They also analyze the approximation quality of their algorithm theoretically and experimentally. Their algorithms produce nearly optimal paths.

The algorithm is described for the special case in which there is a single source and a single target. The following variations are also possible with the algorithm as well:

- Single Source / Multiple Targets: (ONE-TO-ALL or ONE-TO-ANY)
- Multiple Sources / Single Target: (ALL-TO-ONE or ANY-TO-ONE)
- Multiple Sources / Multiple Targets: (ANY-TO-ANY or ANY-TO-ALL)

These variations require only a simple modification of the algorithm to store arrays of sources or targets to be used.

3.1 Preprocessing Phase:

We describe here our partitioning phase of the algorithm which uses the *Multidimensional Fixed Partition (MFP)* scheme of [22, 32, 33]. The MFP scheme divides the data spatially, takes care of clustering problems

and provides a natural mapping to processors. It is a domain driven partition (in contrast to data driven) and can be seen as a generalization of other known domain partition structures such as quadtree and octree. The MFP differs from those in the shape of the partitioning grid in that it generates decompositions according to the number of available processors as opposed to a fixed dimensional grid. The tree represents a decomposition of the data into levels, where lower levels represent a recursive (refined) partitioning of the level above it. We first describe the single-level partitioning for the MFP and then discuss the multi-level partitioning.

During preprocessing, the terrain $\mathcal{P}$ with n triangular faces is partitioned among $p = R \times C$ processors (R denote the row and C denote the column, seen for ease of description as having a grid-like interconnection topology). We say that processor p_{rc} is in row r, column c of the mesh (where p_{00} is the bottom left). Each level of decomposition in the MFP tree consists of p sub-partitions. The top level represents all of $\mathcal{P}$ and is denoted as level 0. Each further level of partitioning is created by dividing a partition from the previous level into an $R \times C$ grid of cells whose boundaries are defined by horizontal and vertical cut lines.

More formally, let the smallest enclosing bounding box of $\mathcal{P}$ be defined by (x_{min}, y_{min}) and (x_{max}, y_{max}). Divide $\mathcal{P}$ into an $R \times C$ grid such that each grid cell defines an equal area of the terrain defined by $((x_{max} - x_{min})/C) \times ((y_{max} - y_{min})/R)$. For example, Figure 1 shows how two terrains are divided into 3 x 3 grids when 9 processors are used. Each face of $\mathcal{P}$ is assigned to each processor whose corresponding grid cell it is contained in or intersects with. Using this technique, faces of $\mathcal{P}$ that intersect the boundary of two adjacent

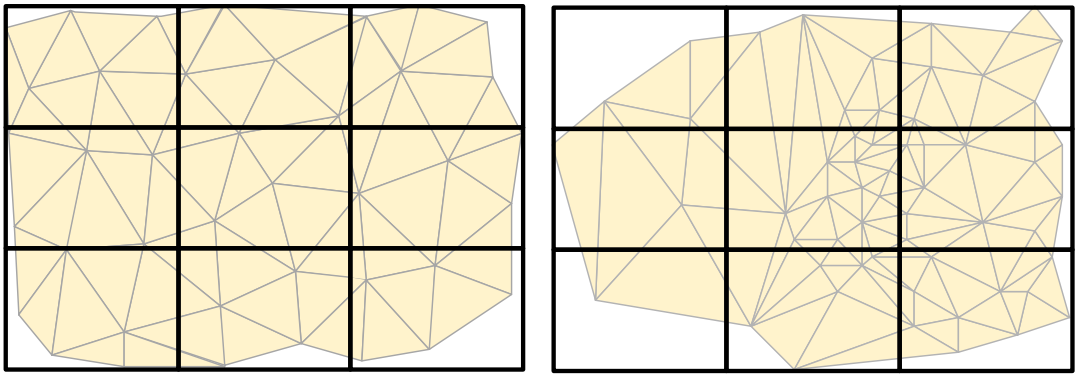

Figure 1: Dividing non-clustered and clustered terrains into equal-area 3 x 3 grid.

grid cells are shared between the two partitions. We call the shared faces *border faces* or *border triangles*.

This single-level partitioning of the MFP scheme (as well as most other partitioning techniques) are susceptible to large processor idle times when used with a parallel shortest path algorithm. Consider the example of Figure 2 in which a wavefront propagates across nine processors beginning at processor 6. A processor does work only when a portion of the active border lies within its region of $\mathcal{P}$. While the active border expands within processor 6, all other processors sit idle. Eventually, the active border will reach processors 3, 4 and 7. Still, many processors sit idle. It is easily seen that processor 2 remains idle until the propagation is near completion; thus it is barely used and is therefore wasted. Moreover, once the active border has left the region assigned to processor 6, it also becomes idle while the other processors take over.

One way of avoiding this idling problem is to assign to each processor, more than one connected portion of the terrain. That is, we can *over-partition* $\mathcal{P}$ which involves a recursive decomposition of the partitions. The result is that each processor will have scattered portions of $\mathcal{P}$. It is therefore more likely that the active border will be processed by many processors throughout its growth. Figure 3 shows how the MFP tree re-partitions each partition one level further to two levels. Observe that in the example, the active border expands quickly to allow four processors to participate in the calculations. Soon afterwards, all processors get involved and are continually used throughout the propagation.

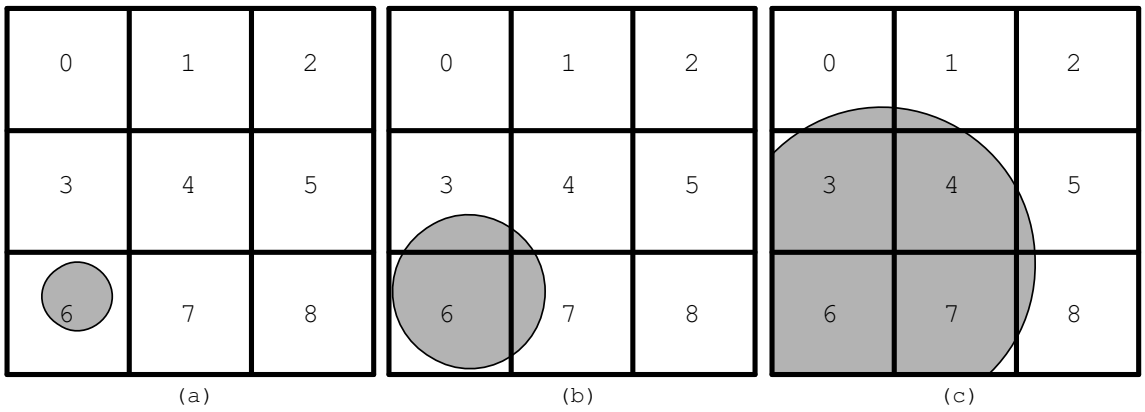

Figure 2: Expansion of the active border showing that processors may sit idle during a shortest path computation.

To produce multiple levels of partitioning, it is necessary to choose a threshold for the maximum number of vertices that a single partition can hold before being re-partitioned. If the number of vertices in a partition (called the *tile size*) exceeds the threshold, it is re-partitioned further. Deciding on the "best" threshold depends on factors such as terrain size, number of processors and the degree of clusterization. If we make the grid too fine, then the regions assigned to each processor are too small to work with and most of the processor time is spent carrying out interprocessor communication. At the other extreme, if we make the grid too coarse, a processor may be assigned a large amount of data thus, forcing the processor to spend too much time processing the data before it can forward some of the work to other processors. This reduces parallelism and ultimately results in a running time equivalent to that of a sequential algorithm. There is a specific number of decomposition levels that permits a nice tradeoff between processor work and communication. For typical terrain data, we expect that no more than four levels of recursion are required and that two to three levels are adequate for achieving good results for a variety of applications (for a detailed discussion of the MFP and applications see [32, 33]).

An additional factor is data clustering, that is, if we partition the entire terrain equally, some parts of data may be sparse, while other parts may be quite dense. This can result in a deficiency in parallelism throughout shortest path propagation since the algorithm would slow down at dense clusters. The MFP scheme helps to avoid clustering problems by re-partitioning those larger partitions that are dense. This results in partitions of different sizes that cover different amounts of spatial area. In order to determine when to re-partition, a threshold is chosen as the maximum number of vertices allowed per partition. If a partition exceeds this threshold, it is re-partitioned and a new level is formed in the tree. Figure 4 shows a 3-level partitioning using MFP for nine processors. Notice the refined partitioning in the dense areas. When partitioning with multiple levels, each processor must know which other processors it shares a partition boundary with in order to allow propagation over shared boundaries.

The method by which a processor determines which other processors it needs to communicate with is known as a *mapping scheme*. Two important issues that must be considered are (1) minimizing processor *hops* and (2) avoiding *bottlenecks*. The first of these issues is important in a parallel machine in which processors are connected to each other by localized communication links, such as a mesh or torus (i.e., a mesh interconnection scheme with wraparound connections) topology. Thus the mapping scheme should allow messages to be sent to processors that are nearby (perhaps in the next row or column of the grid, as opposed to two or three rows/columns away). This will also help in reducing the congestion in the network as there will be less messages "in transit" at any time. For machines in which a single shared "bus" is used

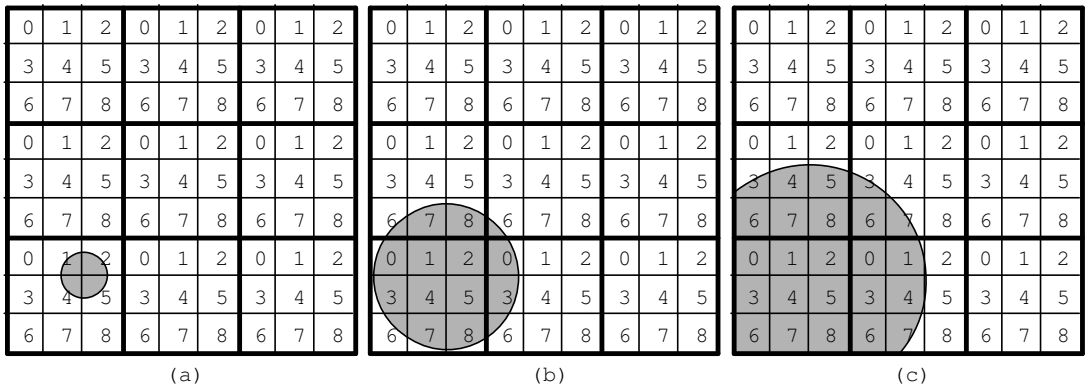

Figure 3: Over-partitioning allows all processors to get involved quickly in the computations and remain involved longer, thus reducing idle time.

between all processors, this is not an issue since there are no intermediate processors. The second issue is always relevant. If the mapping scheme is poor, then it may be the case that certain processors are "favored" over others as being a receptacle for messages which can lead to bottlenecks. A good mapping scheme should treat all processors equally in terms of how much communication and work they are to perform.

The MFP scheme handles both of these issues. The mapping is done with respect to the levels of the tree (i.e., levels of recursive partitioning). We say that unpartitioned data is at level 0, while a single partitioning is at level 1, and so on. Consider re-partitioning the data in processor p_{rc} to create one more level. The data must be re-partitioned again by splitting it up into R rows and C columns which are smaller partitions (called *cells*). The data for each of these newly created cells must now be re-assigned to the $p = R \times C$ processors. Let $cell_{ij}$ be the cell at row i, column j of the grid of cells where $cell_{00}$ is at the bottom left. The *MFP mapping scheme* is as follows. If a re-partition of the cells will represent an even level number in the MFP tree, a *backward mapping* is used, otherwise a *forward mapping* is used. A forward mapping assigns $cell_{ij}$ to processor $p_{((r+i)modR)((c+j)modC)}$ and a backward mapping assigns $cell_{ij}$ to processor $p_{((R+r-i)modR)((C+c-j)modC)}$ (for details see [32, 33]). Figure 5 shows an example of how the MFP scheme maps a partition recursively for a 3x3 mesh of processors. The example shows three levels of recursive partitioning, assuming a uniform distribution of data. The actual partition scheme would only re-partition those partitions which are dense. Notice that the mapping scheme tends to promote groupings of similar processor assignments which reduces the amount of overall communication during shortest path computation. Another benefit of the MFP mapping scheme is that it does not "favor" any particular processor with respect to communication time.

3.2 Running the Simulation:

In this section we describe the execution of the algorithm which assumes that the data has already been partitioned.

3.2.1 Algorithm Outline

Each processor begins with an initialization step in which the processor loads its partition data from disk into memory and initializes the cost to each vertex and initializes its own priority queue. The source vertex is given the cost of zero, while all other vertices are assigned a cost of infinity. Two additional LOCAL_COST and MAX_COST variables are maintained locally for each processor (they are used in the termination detection phase). The LOCAL_COST variable maintains the cost of the last node which has been removed from the

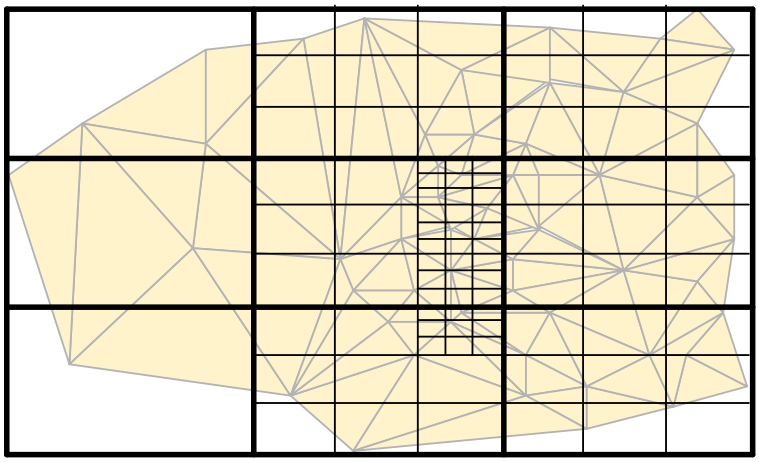

Figure 4: A 3-level MFP partition for a 3x3 processor configuration.

queue. The MAX_COST variable represents the maximum cost that the program will simulate up to. That is, once all nodes have a LOCAL_COST exceeding the MAX_COST, the processor will stop processing data. The MAX_COST has the initial value of infinity.

After initialization, the processor enters into a loop which does the actual shortest path propagation. The algorithm differs from Dijkstra's algorithm in that it must inform adjacent processors of vertex costs whenever the active border crosses a partition boundary. In addition, the condition for deciding when to stop the processing is slightly more complicated and involves passing around tokens to all processors. The pseudo-code for the algorithm is shown in Figure 6. Note that Q is the priority queue and TARGET is the target vertex.

It is easily seen that the bottom portion (i.e., last lines 23-36) is quite similar to that of Dijkstra's algorithm. There is added code for sending updated costs to adjacent processors when a vertex is processed on a partition boundary (i.e., shared between two processors). If a processor receives a cost for a vertex from an adjacent processor, it must check to see if this cost is better than its own locally stored cost for that vertex. If so, this vertex is updated to have this new smaller cost and this change may cause a re-ordering of the queue. Processing then continues as before by removing the vertex with minimum cost (which may now be this newly updated vertex).

Recall that the vertices in the graph are Steiner points placed on the edges of polyhedron P, and the edges are the segments joining pairs of Steiner points in a face. In order to reduce the graph storage overhead, the edges which are internal to the terrain faces are not explicitly stored. However, the terrain edges themselves (called *arcs*) are stored. The coordinates of Steiner points along each arc are not stored, but their costs are kept in an array associated with the arc. For each arc that crosses a partition boundary, it is stored in both partitions. Whenever the cost changes for any of the endpoints or Steiner points along this arc, the whole arc (along with the Steiner point costs) is sent to the adjacent processor. The adjacent processor then decides which costs need to be updated.

3.2.2 Termination detection

The algorithm uses tokens which are sent to adjacent processors when the target is processed. A *cost token* is used to keep track of the maximum cost that the processor is allowed to process to. Once a processor extracts the target from its queue, it updates its local MAX_COST to be this cost and sends a cost token indicating this target cost to an adjacent processor. This cost token is passed from processor to processor in round-robin fashion. When a processor p_i receives this token message from another processor, it stores the

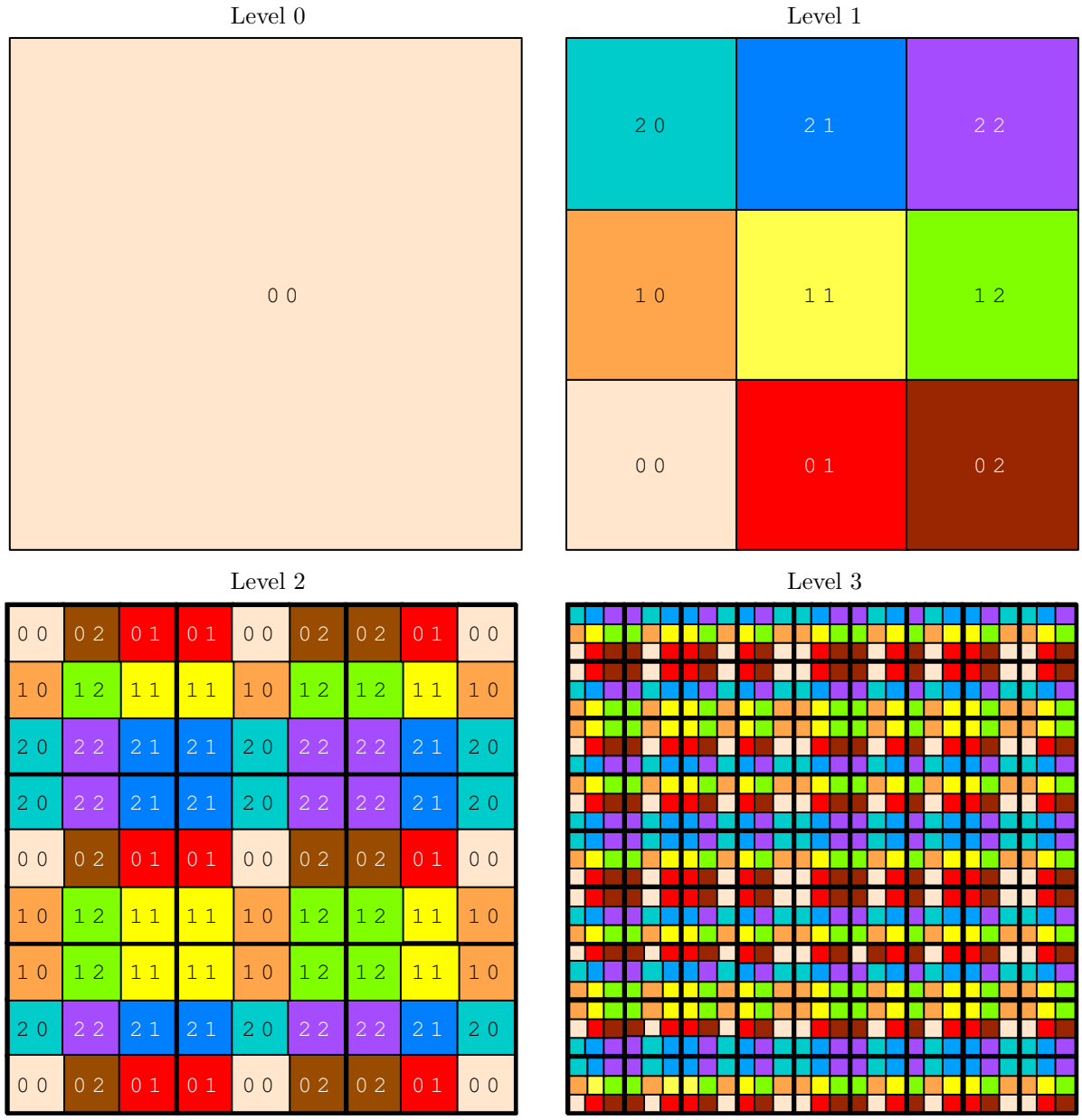

Figure 5: Levels 0, 1, 2 and 3 of the MFP mapping scheme for a 3x3 mesh of processors.

token's cost as its MAX_COST. If there are no more nodes with cost less than this new MAX_COST (i.e., LOCAL_COST exceeds MAX_COST) in the queue of p_i, then p_i halts and waits for messages (lines 18-21).

In addition to the cost token, an additional *done token* is passed around to determine which processors have completed their computations. The done token originates from the processor that first reaches the target node. Upon receiving this token, if a processor has more work to do, it keeps the token. Once this processor has no more work to do, a new round of sending the done token message starts. If the done token was circulated twice through all the processors (and it was not restarted) then the algorithm concludes that the search for the shortest path is complete.

```
1   MAX_COST <- INFINITY ;  LOCAL_COST <- 0;
2   WHILE(TRUE) DO {
3       IF (there is an incoming message) THEN {
4           IF (message is a cost token with cost C) THEN {
5               MAX_COST <- C;
6               IF (this vertex was not the originator of the token) THEN
7                   Send cost token to an adjacent vertex; }
8           IF (message is a done token) THEN {
9               IF (this processor was the originator of the token and
10                  the token has gone around twice) THEN {
11                  Send a TERMINATE message to an adjacent processor;
12                  Quit: ``no more work to do''; }
13              ELSE  Store the done token; }
14          IF (message is an updated cost C for a vertex V and C < cost(V)) THEN
15              cost(V) <- C;
16          IF (message is TERMINATE) THEN
17              Quit: ``all processors have finished''; }
18      IF ((Q is empty) or (LOCAL_COST > MAX_COST)) THEN {
19          IF (this processor has the done token) THEN
20              Send done token to an adjacent processor;
21          Wait for an incoming message; }
22      ELSE {
23          Vmin <- vertex with minimum cost;
24          Remove Vmin from Q;
25          LOCAL_COST <- cost(Vmin);
26          IF (LOCAL_COST < MAX_COST) THEN {
27              IF (Vmin = TARGET) THEN {
28                  MAX_COST <- cost(Vmin);
29                  Send cost token to adjacent processor with cost(Vmin); }
30              ELSE {
31                  FOR (each edge E incident to Vmin) DO {
32                      V <- vertex at other end of E than Vmin;
33                      IF (cost(V) > (cost(Vmin) + cost(E))) THEN {
34                          cost(V) <- cost(Vmin) + cost(E);
35                          IF (V is a shared vertex with processor p) THEN
36                              Send cost(V) to p; }}}}}}
```

Figure 6: Pseudo-code for algorithm on each processor.

A counter is maintained within the token indicating the number of processors that had no work to do when the token arrived. Once this token has gone around twice with a count equal to that of the number of processors, a TERMINATE message is then sent to all processors to halt the processing and begin a traceback (construction) of the path.

Note that unlike Dijkstra's algorithm, if the queue becomes empty at any time, the algorithm does not stop. Also note that some minor details have been left out from the pseudo-code. For example, when a vertex is updated, the vertex must also store which vertex led to that one. That is, each vertex must keep a pointer to the parent vertex in shortest path tree. The union of all these pointers implicitly represents a shortest path tree.

3.2.3 Trace back

Once processing has completed, the cost to the target is known and all that remains is to trace the path back from the target to the source. All processors receive a traceback message. The processor on which the target lies begins the traceback by piecing together the graph edges backwards from the target. The path is repeatedly traced back within the processor that contains the target by accessing the vertex before it in an implicit shortest path tree. The traceback continues until either the source is reached, or until a partition border is reached. In the later case, the processor must package up all path information obtained so far and pass it to the adjacent processor which must "take over" the traceback. Eventually, the path will be traced back to the source and this path is reported.

4 Experimental Results

In this section, we describe the experiments that we conducted in order to show the practicality of our algorithms. For a more in-depth look at our experimental results, see [26]. The objectives of the experiments were to investigate the performance characteristics of the algorithm by observing the effects when parameters are varied. The tests were run on real terrain data (see [24, 25]). The focus in this paper is on timing results, since path accuracy and relevance in the sequential setting have already been established in [24, 25].

As discussed in section 2.2, there are many factors that can affect the performance of a parallel algorithm. In addition to these factors, there are some which are inherent for our particular algorithm (e.g., the number of Steiner points used and the weighted cost function). To provide a complete report on the effects of all of these parameters would require testing all combinations of parameters for many iterations. However, if not enough variations of data and parameters are used, it is not possible to make an adequate assessment of their effects on the algorithm's performance. We have chosen a compromise in that we use an accumulative experimentation approach. That is, we test one set of data for a specified parameter set, then modify the parameters in stages to show the effects at each stage.

In terms of algorithmic performance factors, our algorithm uses a distributed queue label-setting strategy with round-robin token passing for termination detection. Our research presented here does not compare different labelling algorithms, queueing strategies, termination detection techniques nor variations of the cost function. Instead, our experiments focused on the data-related factors pertaining to partitioning, terrain size and the relative location of source/target pairs. Although we did not address any particular machine-related issues, we do show the performance results of our algorithm running on three machines of different architectures. Lastly, there is only one detail that we address pertaining to implementation-related factors: the effect that communication speed has on the performance as different message-passing software is used.

The remainder of this section is organized as follows. We start by describing in Section 4.1 the test environment, and in Section 4.2 the experimental test data and procedures. We then describe our performance results in Sections 4.3 and 4.4 in terms of speedup and efficiency. We conclude with a discussion on the impact of communication and the impact of number of sources on the overall performance in Sections 4.5 and 4.6, respectively.

4.1 Test Environment

We conducted the tests on a variety of platforms so as to examine the sensitivity level of our algorithms and data structures to different parallel models. Although our implementation is primarily designed for MIMD architecture using the Message Passing Interface (MPI) standard, we also executed our algorithms on a shared memory architecture. The experiments were performed on a variety of platforms including

- **Network of Workstations -** consists of sixteen 166Mhz Pentium computers that connected via a dedicated 100 megabit crossbar switch. Each processor has 96MB of internal memory except processor 0 which has 128MB. This machine is referred to as ASP in the text.

- **A Beowulf Cluster -** consists of 64 1.8 Ghz Pentium P4 Xeon computers (arranged as Dual P4s for each of 32 nodes) and connected via a dedicated Cisco 6509 switch. Each node has 512 MB RAM and a Gigabit copper ethernet connection.
- **Symmetric MultiProcessing Architecture (SMP) -** The SMP architecture is a SunFire 6800 server. It consists of 20 Sun UltraSPARC III 750MHz processors with 20 GB of RAM. The RAM is shard by all processors.

4.2 Test Data and Procedures:

We ran our tests on various terrains that differed in size and height variation. The terrains are represented as triangulated irregular networks (TIN). Since many partitioning schemes are susceptible to performance differences when clusterization occurs in the data, we also attempted to test terrains with different clustering characteristics. Table 1 shows the specifications of the terrains that were tested. The terrains were all

NAME	#VERTICES	#FACES
Africa	5,000	9,799
Sanbern	8,000	15,710
Madagascar15k	15,000	29,582
America40k	40,000	79,658
Madagascar50k	50,000	99,268
BigTin	1,000,000	1,495,000

Table 1: Terrains used for testing the performance of the algorithm.

constructed from cropped portions of actual Digital Elevation Model data. Once cropped, the obtained grided terrain was then simplified through the removal of vertices. The vertices chosen to be eliminated were those such that when removed, the volume difference of the terrain was minimal. Figure 7 shows top-down view snapshots of the terrains tested. Notice the left-sided clusterization of the America40k terrain.

In order to obtain proper speedup comparisons, the tests were run many times for a variety of processor configurations. The configurations were obtained by varying the number of rows and columns of processors. The configurations used were 1x1, 2x1, 2x2, 3x2, 3x3, 4x3, 4x4. Note that as mentioned earlier, our algorithm requires mid-range granularity of the data. That is, if we partition the data too fine, the processors end up doing too much communication with neighbors. Thus, in order to make efficient use of a larger number of processors than 4x4, we would need larger data sets than the ones tested here.

All tests use the *fixed placement scheme* of [25], i.e. by placing an equal number of Steiner points per edge, and then interconnecting all of them within a face. It has been shown previously that the *interval placement scheme* and *sleeve-based placement scheme* had near identical timing in the sequential setting. Therefore, it was unnecessary to test all placement scheme variations of [25] since they differed mainly in path accuracy and not in runtime. Varying the number of Steiner points does not have a significant effect on the efficiency of our parallel algorithm. With every linear increase (i.e., m) in the number of Steiner points per edge, there is a quadratic increase (i.e., m^2) in the number of arcs of its incident faces. With larger terrains, the use of more Steiner points should result in a higher percent of computation time, thereby attaining better speedup. (The interested reader is referred to [32] for details.) Since we have observed in [25] that typically 6 Steiner points were sufficient to achieve good accuracy, we continued the remainder of our tests with this same number of Steiner points.

Our tests include both one-to-one and one-to-all variations. For the one-to-one tests, we compute a set of 50 random vertex pairs, each pair is called a *session*. For each session, we generate overall statistics that include path cost and overall run time. Most of our graphs show average information which is obtained by summing all the results per session and dividing by the number of sessions. We also show minimum and maximum results where they present some more insight.

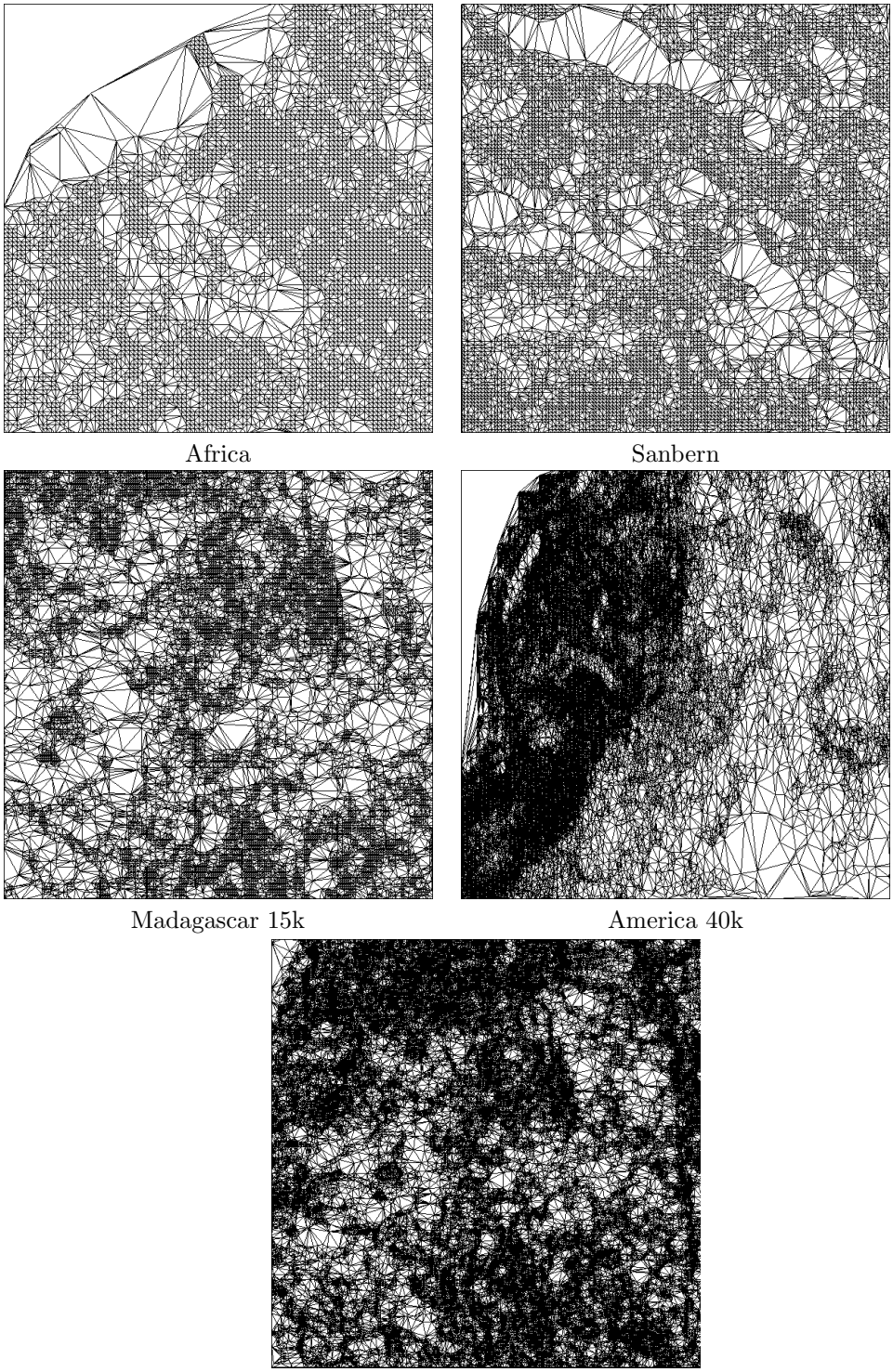

Figure 7: Top-down view snapshots of the terrains tested. The BigTin data set is not shown due to density of the data. However, it is shaped like a sausage spanning from the top left corner to the bottom right corner of its enclosing rectangle.

The *overall run time* is the time (in seconds) elapsed from when the session first starts until the final path is returned. The timing results presented here include the time required to compute the path itself, not just to produce the cost [1]. In addition to these, we also generated timing information per processor that included

- Initialization time
- Session initialization time
- Total run time
- Compute time
- Idle time
- Communication time

The *compute time* represents the amount of time that the processor spends doing computations with respect to its own queue (including the time for generating the final path). The *idle time* is the total amount of time that the processor remains idle while it is not performing any computation or communication. The *communication time* is the amount of time that the processor spends on communicating with other processors. The *total run time* for a processor is the sum of these three times. The communication time in our experiments is determined by subtracting the compute and idle times from the total run time [2].

Lastly, we generate (per processor) information pertaining to *causality errors*. Causality errors are one of the major issues in parallel simulation. They occur when a lack of synchronization causes a processor to work only with local information and thus, to incorrectly determine the course of the simulation. This happens when a processor does not wait for global synch up-to-date data, but rather processes the locally stored data (which is outdated) and advances its local simulation time usually ahead of the global simulation time. Causality errors can lead to poor performance of the parallel computer. In cases where the output effects of the executed program are not uniformly distributed across the spatial area, some processors may do extra work in vain. This is especially important and evident when the computation part is expensive, in which case, precious computing cycles are wasted. The information pertaining to causality errors includes:

- Number of messages sent to adjacent processors
- Number of messages received from adjacent processors
- Number of processed vertices
- Number of vertices received from adjacent processors
- Number of vertex updates
- Number of vertices inserted into the queue
- Number of vertices re-inserted into the queue

With this causality information, we are able to estimate the amount of computation overhead from having each processor maintain its own queue as opposed to a global queue as in the sequential setting.

[1] The overall run time was computed using the time() function in C which rounds off to the nearest number of seconds.
[2] These time were computed using the clock() function in C which rounds off to the nearest number of milliseconds.

4.3 Speedup and Efficiency

In this paper, we measure the speedup as $s = \frac{T_1}{T_p}$, where T_i is the time it takes to solve the problem using the parallel algorithm on i processors, $1 \leq i \leq p$. This measure of speedup is used extensively in the literature (see e.g., [1, 10]); we also use it when evaluating our experiments. Also, the *efficiency* is defined here as $\frac{s}{p}$ when p processors are used.

We tested several types of shortest path queries: one-to-one (point to point), one-to-all and few-to-all. In order to obtain a good representation of the performance we executed a large number of random queries for each query type. The graphs presented in this section are the average of the query results.

Figure 8 shows the speedup and efficiency of executing one-to-one queries on three different data sets on the Beowulf. For the America40k we obtained a speed up of 2, 2.8 and 3.6 for the 2x2, 3x3 and 4x4 configurations, respectively. Similar speedups were also observed for the Madagarscar50k data set.

When we examined the efficiency levels we observed an interesting phenomena. For the America40k we obtained an efficiency rate of roughly 50%, 32%, and 23% for the 2x2, 3x3 and the 4x4 configurations, respectively. For the Madgascar50k we obtained an efficiency rate of 58%, 34%, and 25% for the 2x2, 3x3 and the 4x4 configurations, respectively. This trends shows that for these data sets, the expected speedup is roughly $\sqrt{p}$ where p is the number of processors. Similar behavior was observed when the query was executed on other types of hardware within a constant factor. Figure 9 shows the execution results on the SunFire. As can be seen in section 4.5, similar results were obtained on the ASP computer with a fast communication implementation (see Figure 12). Notice that the SunFire (i.e., shared memory machine) results show better speedup than the Beowulf. This is because the communication speed is faster on the SunFire, as data is transferred between processors through shared memory, as opposed to being passed over a network switch (as on the Beowulf). Although, the SunFire is able to process the queries faster, we can see a similar trend to within a constant factor. The reason for this behavior is the number of processors that are simultaneously active. The processors that are active are those on the active border of the simulation. Data sets such as the America40k and the Madagascar50k are not big enough to take advantage of the MFP partition and mapping. The MFP mapping assigns different parts of the data sets to different processors and thus it overloads the processors.

Overloading the processors with data to be processed reduces the amount of time that a processor waits for work. The MFP partitioning and mapping provides a natural way of distributing the data to the processors such that each processor can work on different parts of the data simultaneously. To show the impact of partitioning on the overall performance we conducted a series of tests with different partitioning sizes and thus with different levels of overloading. In Figure 12 we observe the impact of processor overloading. By examining the columns of the fast communication we can see that as the tile size threshold is reduced, the speed-up increases. This is evident even with small data sets (America40k and Madgascar50k).

When larger data sets are used, such as the BigTin, the overloading has a greater impact and we can see a significant improvement in the performance. In Figure 8 the 2x2 configuration has achieved a speedup of almost 4 (99% efficiency) and the 3x3 and 4x4 configuration roughly achieved a speedup of 4.5 (52% efficiency) and 6 (38% efficiency) respectively. In the case of the 4x4, the speedup of the MFP is not fully exploited because of the shape of the data. The sausage-like shape does not allow the 16 processors to take full advantage of the MFP structure.

4.4 Measuring the Amount of Over-Processing and Re-processing:

One measure of efficiency of a parallel shortest path algorithm is the number of vertices that are processed. Hribar et al. [20] use a similar strategy in that they measure the total number of updates. Since faces (and hence graph vertices) are shared between adjacent partitions, the cost to these vertices may change many times if a shortest path crosses the boundary frequently. More importantly, the cost updates along border vertices may cause a rippling of cost updates throughout the partition, thereby causing a re-computation of previously computed costs to many non-border vertices of the partition as well. As a result, the cost to any particular vertex may be re-computed often. In a sequential label-setting algorithm, once a vertex

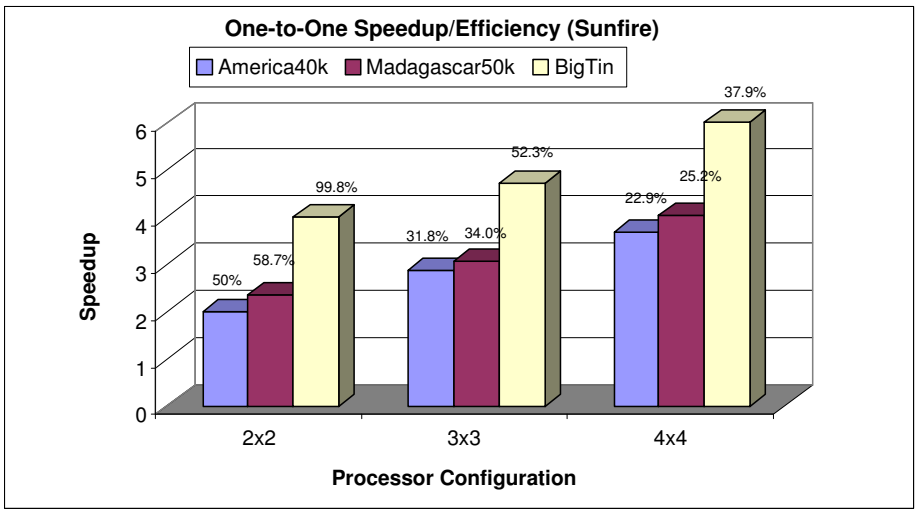

Figure 8: Graph showing speedup and efficiency of the one-to-one query using different processor configurations. The queries were run on the Beowulf computer. The processors configuration is shown on the x-axis and the obtained speedup on the y-axis. The efficiency is shown as a percentage above the corresponding data set.

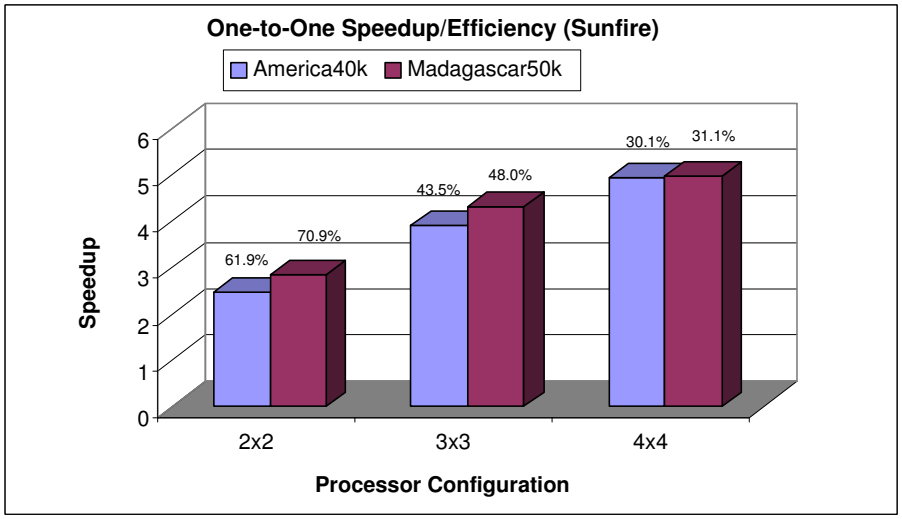

Figure 9: Graph showing speedup and efficiency of the one-to-one query using different processor configurations. The queries were run on the SunFire computer. The processors configuration is shown on the x-axis and the obtained speedup on the y-axis. The efficiency is shown as a percentage above the corresponding data set.

is removed from the priority queue, it never gets updated again. The parallel distributed algorithm is less

efficient in terms of the number of vertices processed. We call this amount of additional computation as *over-processing*. We also define the *re-processing count* of a vertex to be the number of times that vertex had been processed (i.e., removed from the priority queue).

When propagating over a boundary, another processor "takes over" a portion of the active border and processes in parallel to other portions of the active border on other processors. Therefore, the global active border will typically grow larger than with a sequential algorithm for any particular source/target pair. This causes vertices to be processed by the parallel shortest path algorithm that otherwise would not have been processed by a sequential shortest path algorithm. Therefore, in order to obtain a "good" estimate as to the amount of re-processing done by our algorithm, we ran tests that processed the entire terrain. By doing this, we eliminate the need to consider processed vertices that would not have even been processed by the sequential algorithm.

We analyzed the results of our one-to-all tests with six Steiner points per edge using the weighted cost function. We compute the amount of over-processing as follows:

$$\left(\frac{|V_p| - |V_1|}{|V_1|} * 100 \right) \%$$

where $|V_p|$ is the number of processed vertices when p processors are used and $|V_1|$ is the number of processed vertices when one processor is used.

The graph of Figure 10 shows that the amount of over-processing increases with the number of processors but also that the partition itself plays a role as well. For example, notice that the 4x4 partition has less over-processing than the 3x3 partitioning for the two small terrains. Notice also that the America and Africa terrains have the largest amount of over-processing. This is due to the unbalanced load across the processors because of the clustered nature of the data.

There is a significant difference in the results depending on the location of the source point. The results using a centrally located (i.e., middle) source are better than those using a source near the terrain corner. The reason for this large difference can be explained by examining the idle times. Figure 11 shows the percent of idle time for both tests using the corner and middle source points for the Madagascar50k TIN. Notice that the processors are more often idle when the corner source is used than with a middle source point because in the former case the wavefront reaches all processors later.

Hence, using a single-level partitioning the processor idle time is a significant factor that hinders the parallelization efforts of the algorithm and results in a poor performance. In some cases, many processors sit idle for up to 50% of the time. A related phenomenon has also been observed by Hribar et al. [20].

4.5 Impact of Communication:

Communication patterns and speed play an important role in the overall performance of parallel implementations. The tradeoff between amount of data, which is transmitted between processors, and the computation time that can be gained should be carefully assessed when fine tuning the implementation.

In this section we give an example of the impact that a relatively minor change in communication pattern on the overall performance. In our first prototype of the software, we observed that our results were significantly lower than expected. After investigation, we observed two behaviors that impacted the communication component of the software: communication pattern and communication performance.

In our earlier communication pattern we used a fixed-size array to store all incident arcs from a vertex. Whenever propagation reaches the boundary from a Steiner point which is shared between two partitions, this entire array is sent in a message. This is not always necessary however, since propagation from Steiner points on an arc requires, in most cases, only a few number of arcs to be updated across the boundary between processors. Sending an array of constant size had a major impact on the overall performance as it added a constant factor to the communication component of the software even though the amount of data in the fixed-size message was not large. However, this constant factor in the message size was enough to slow the system down because the underlying communication package of MPI could not handle the volume of data. In our modified version we sent only a small number of modified arcs between the processors. This

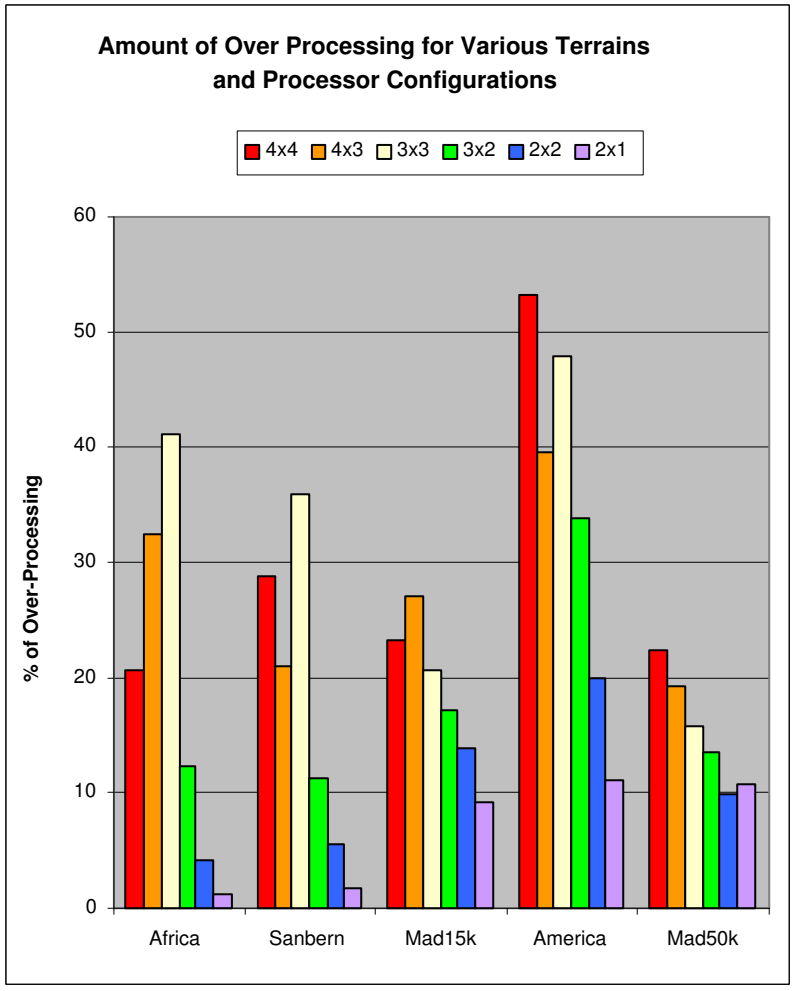

Figure 10: Graph showing over-processing for various terrains and processor configurations.

change, however, has added complexity to the design and code since it required careful bookkeeping of which Steiner point was changed and when.

The second factor that also had a significant impact on the algorithm was the poor performance of one of the functions of the underlying communication package (MPI). We switched from version 6.1 of the LAM MPI libraries to the more efficient version 6.3b libraries. In version 6.1, the MPI_Iprobe function is implemented rather inefficiently and a call to this function causes a significant delay. In the newer 6.3b version, this function is much more efficient and returns much quicker. As will be seen, the difference in this function's efficiency accounts for the difference in computation time between the slow and fast communication tests.

The graphs of Figure 12 show the differences in speedup between our original implementation (i.e., the slow communication) and our improved implementation (i.e., the fast communication). The difference in speedup between the implementations is significant for the smaller tile sizes and less significant for the larger tile sizes. The effect of increasing the communication time with the smaller tile sizes is not as noticeable when the faster communication is used. That is, the communication overhead of having many small tile sizes becomes insignificant as communication speed increases and this leads to better overall speedup. With the slower communication, the larger tile sizes (and hence the single-level partitioning) outperform the smaller

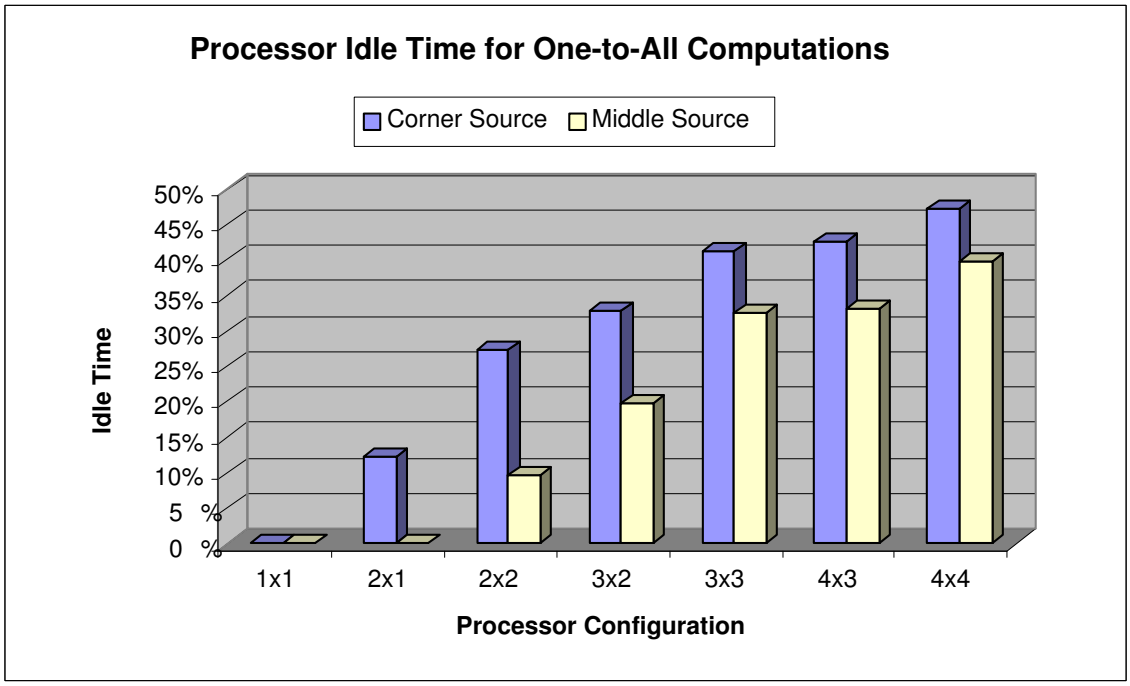

Figure 11: Graphs comparing processor idle time for the two starting source locations in the one-to-all tests for the Madagascar50k TIN.

tile sizes. Notice that the America40k terrain (which is the most clustered terrain) shows a more significant speedup than the Madagascar50k terrain as the partition size decreases (i.e., more levels of partitioning are used). This indicates that the multi-level partitioning scheme has an advantage over the single-level partitioning scheme for terrains that are more clustered. Also shown on the graphs is the maximum efficiency that is obtained from our tests. Notice that efficiencies of around 25% are obtained with the 4x4 processor configuration and this efficiency increases to up to 42% and 59% for the 3x3 and 2x2 processor configurations, respectively.

Figure 13 shows the processor usage for different processor configurations and tile sizes for the Madagascar50k and America40k terrains. The leftmost set of 12 bars in each graph represent the processor usage when the slow communication is used and the rightmost is used for the fast communication. Notice that the overall communication time is negligible when the fast communication is used. The single-level partitioning is the rightmost bar in each group of four bars. As expected, the graphs show that as the tile size decreases (i.e., more re-partitioning is used) the idle time is traded off with communication time. Since our objective is to reduce the idle time of processors, the multi-level partitioning scheme has accomplished this. This reduction is more pronounced with the America40k terrain since it is much more clustered than the Madagascar 50k terrain. With slow communication, this tradeoff is not as helpful to reduce the overall runtime. However, with the faster communication speed, the tradeoff between idle time and communication proves to be advantageous and provides a significant improvement in speedup for the smaller tile sizes (when compared to the slow communication results).

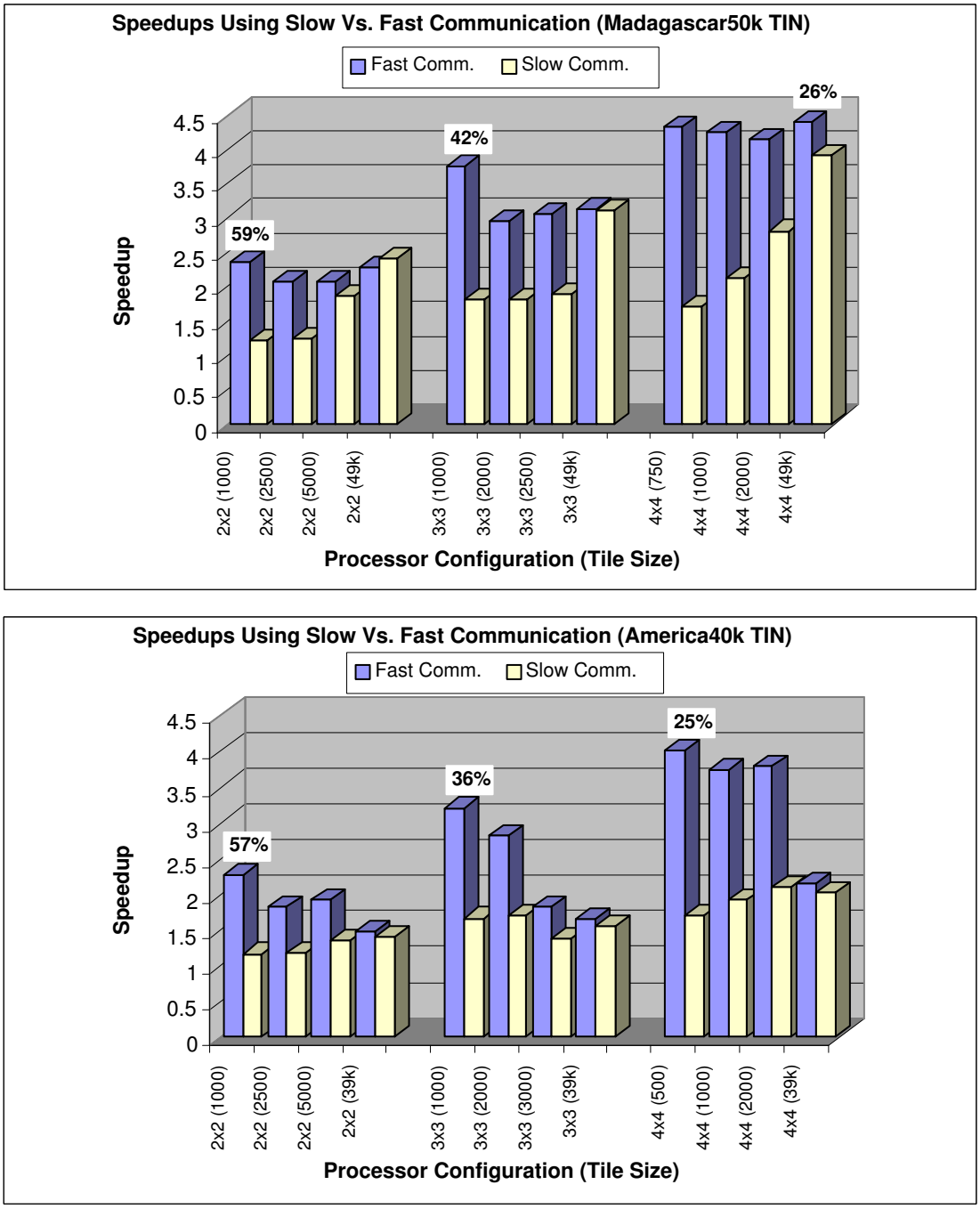

Figure 12: Differences in speedup between the slow and fast communication.

4.6 Impact Of Multiple Source Points

One of the main advantages of the multilevel partitioning scheme is that it helps to reduce causality errors. So far, all the results that were presented used a single source. To fully see the effects of causality errors, we

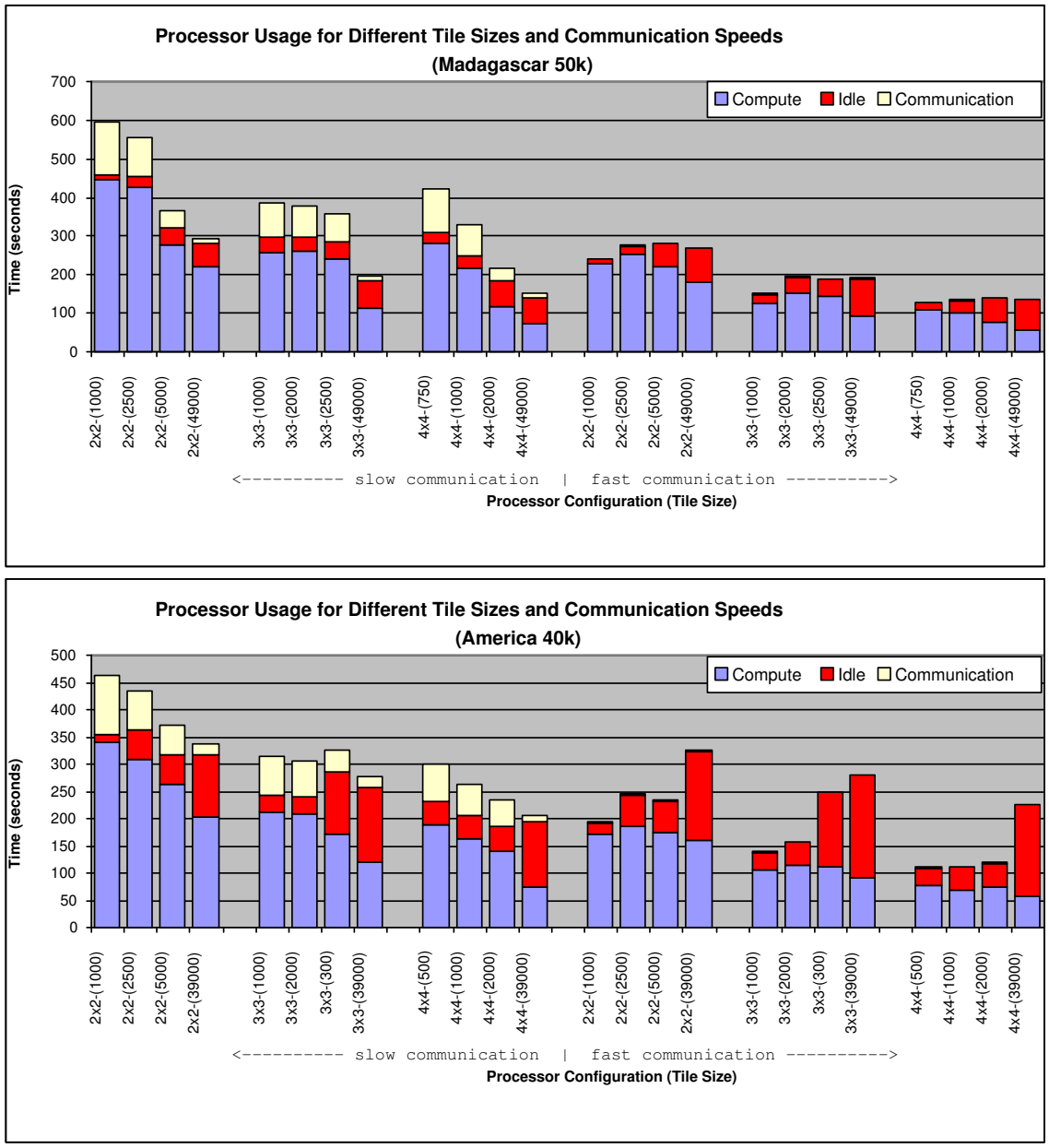

Figure 13: Graphs showing processor usage as processor configuration and tile sizes are changed for implementations with either slow or fast communication speed.

carried out tests for few-to-all shortest paths using multiple sources, each with a different starting weight. This kind of computation can be useful for facility location problems in which each facility has a different "attraction" factor and we would like to compute a weighted Voronoi diagram on the polyhedral surface.

We ran few-to-all tests on the America40k and Madagascar50k terrains. The tests used five source vertices on the terrain: four placed near the four corners of the terrain boundary respectively and one in the

center. The sources were assigned weights of 1, 50, 100, 150 and 200, respectively. We used four different partitionings which were formed using different tile sizes, as with our one-to-all tests, as well as various processor configurations of 2x2, 3x3 and 4x4. The speedups obtained are shown in Figure 14.

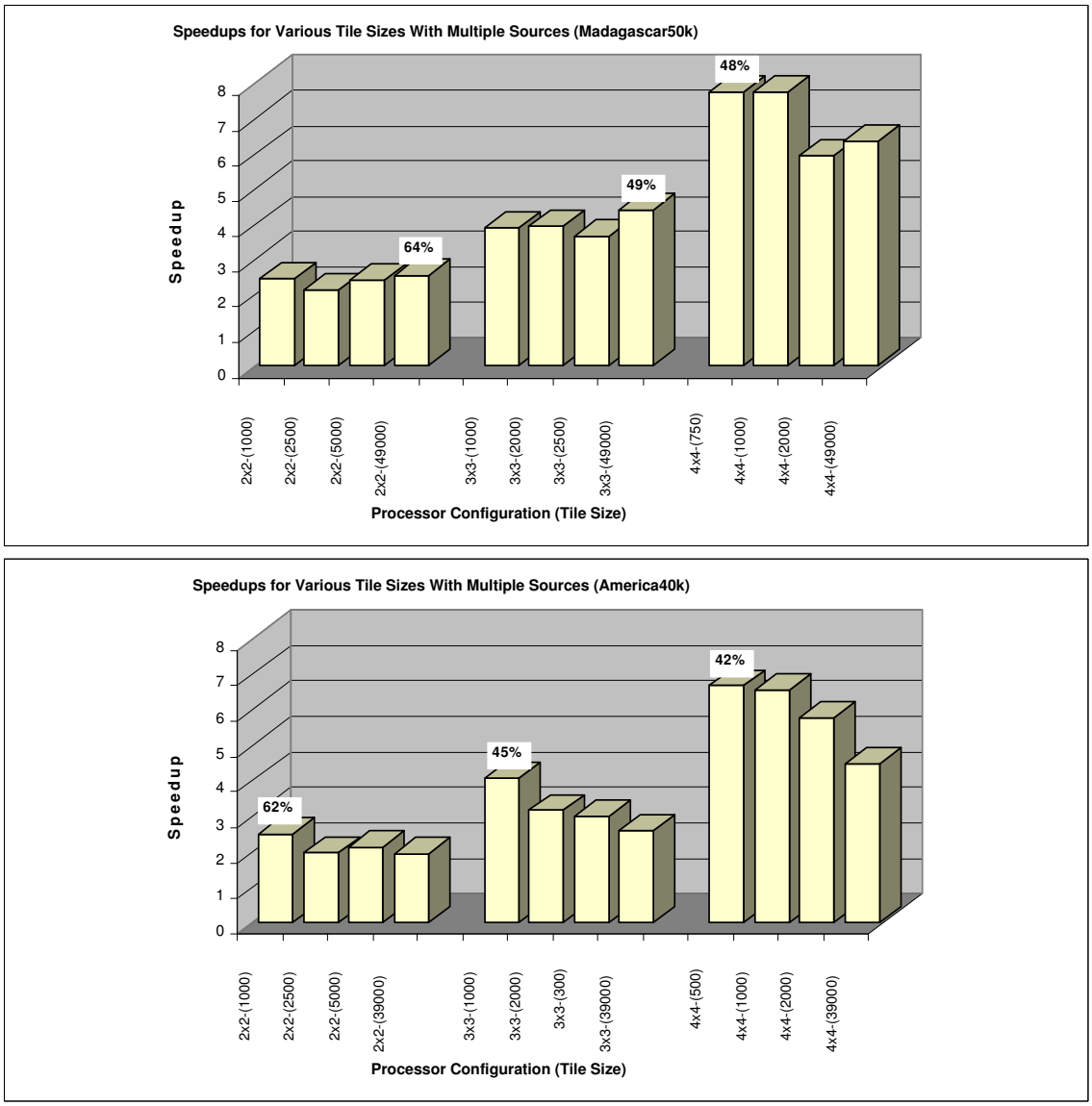

Figure 14: Speedups for few-to-all tests on America40k and Madagascar50k TINs.

The speedups are better than those of the one-to-all tests. Through these tests, the maximum efficiency has almost doubled for our tests with the 4x4 processor configuration. Increases in efficiency are also noticeable in the 3x3 and 2x2 processor configuration tests. The maximum overall efficiency obtained is 64% which is observed from the 2x2 tests. The increase in efficiency results from the more active processors that compute the data correctly and are less exposed to causality errors. As expected, the 2x2 and the 3x3 configurations in the case of the Madagascar50k data set showed a degrading performance as the data

was partitioned. This is because the multiple source points are placed in a uniform fashion on a uniformly distributed data set. Each of the 4 processors in the case of 2x2 and more than half of the 9 processors in the case of the 3x3 have a source point to start with. This significantly reduces the interaction between the processors. This is not the case when the 4x4 configuration is used because more processors are idle (only one third of the processors have a source point). However, when the data is not uniformly distributed such as the America40k we can see the benefit of the MFP. Since the partitioning and the mapping help in the propagation of the active border to multiple processors, this reduces the idle time of the processors.

Examine the results from the 4x4 tests on the America 40k terrain. Notice that the efficiency increases as the tile size decreases. If however, the tile size is too small, then the efficiency will begin to decrease due to the overhead of communication. Hence, the peak in efficiency occurs with the optimal tile size which is likely less than or around 500. It should be noted that even better overall speedups can be obtained once the optimal tile size has been determined experimentally.

Figure 15 shows the difference in the amount of over-processing between the single source tests and multiple weighted source tests. Notice that the amount of over-processing is significantly reduced in many cases when the multiple weighted sources are used. This improvement is most noticeable with the smaller tile sizes and with larger number of processors. This indicates that the MFP partitioning scheme is better suited for weighted shortest paths with multiple sources.

5 Conclusions

In this paper we presented a discussion of the factors influencing the performance of parallel shortest path algorithms and a detailed analysis of our software for computing shortest paths in weighted terrains using parallel computers. Our presented solution overloads each processor with data in such a way that the processor can reduce its idle time and reduce inter-process communication. The overloading scheme, which is based on the multi-dimensional fixed partition (MFP), allows each processor to continue to process the data even though it may have to repeat some of the processing. The MFP structure is also applicable when solving other problems.

6 Acknowledgements

We acknowledge the contributions made by Anil Maheshwari. The authors are also grateful to Ting Wen and Tong Guo for porting and testing of the parallel code.

References

[1] P. Adamson and E. Tick, "Greedy Partitioned Algorithms for the Shortest-Path Problem", *International Journal of Parallel Programming*, Vol. 20, No. 4, 1991, pp. 271-298.

[2] P. Adamson and E. Tick, "Parallel Algorithms for the Single-Source Shortest-Path Problem", *International Conference on Parallel Processing*, 1992, pp. III-346-350.

[3] P.K. Agarwal, S. Har-Peled, M. Sharir, and K.R. Varadarajan, "Approximating Shortest Paths on a Convex Polytope in Three Dimensions", *Journal of the ACM*, Vol. 44, 1997, pp. 567-584.

[4] P.K. Agarwal and K.R. Varadarajan, "Approximating Shortest Paths on a Nonconvex Polyhedron", *Proceedings of the 38th IEEE Symp. on Foundations of Computer Science*, 1997.

[5] L. Aleksandrov, M. Lanthier, A. Maheshwari and J.-R. Sack, "An ϵ-Approximation Algorithm for Weighted Shortest Path Queries on Polyhedral Surfaces", *14th European Workshop on Computational Geometry*, Barcelona, Spain, 1998, pp. 19-21.

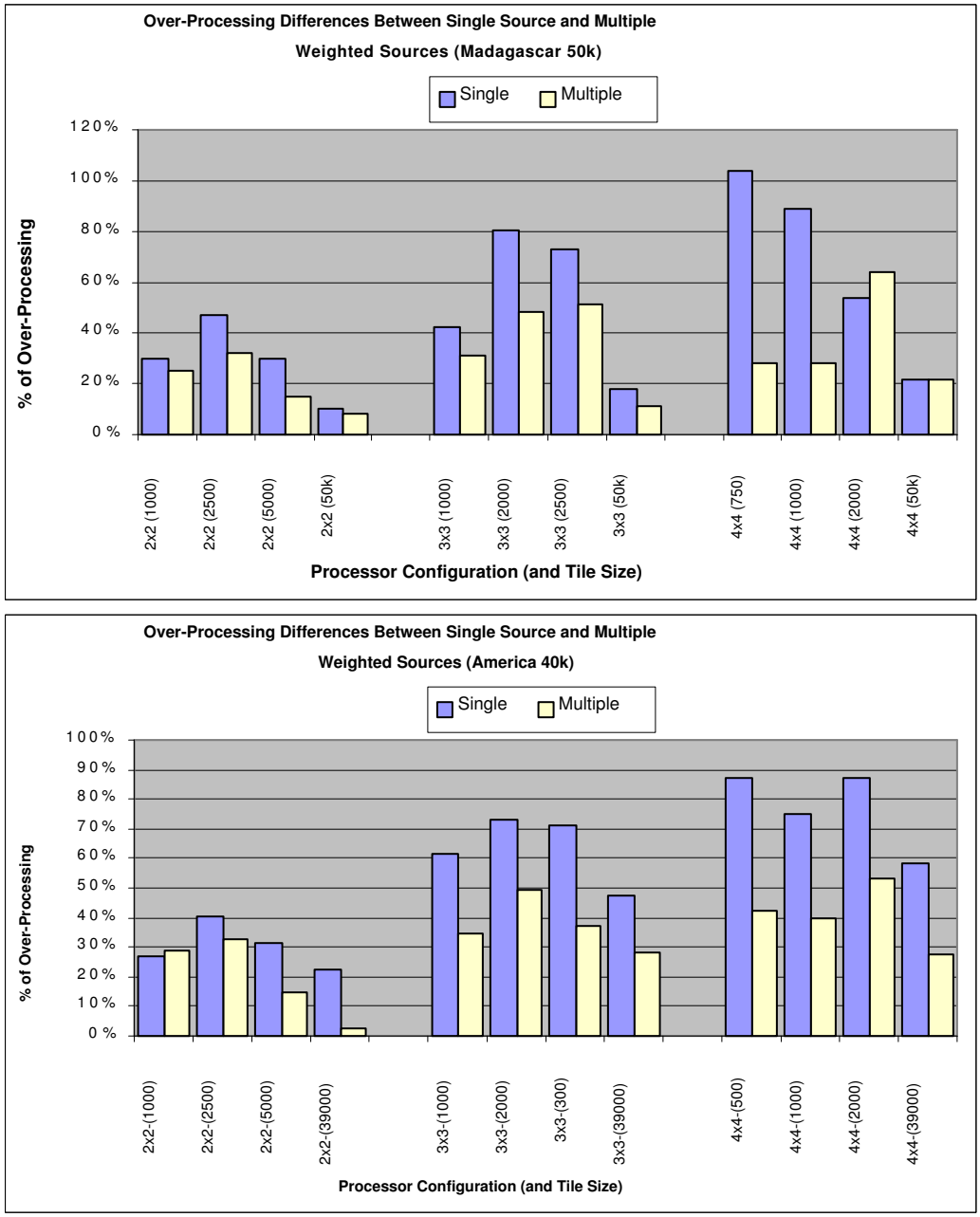

Figure 15: Comparison of over-processing between single source and multiple source tests.

[6] L. Aleksandrov, M. Lanthier, A. Maheshwari and J.-R. Sack, "An ϵ-Approximation Algorithm for Weighted Shortest Paths on Polyhedral Surfaces", *6th Scandinavian Workshop on Algorithm Theory*, LNCS 1432, Stockholm, Sweden, 1998, pp. 11-22.

[7] L. Aleksandrov, A. Maheshwari and J.-R. Sack, *Approximation algorithms for geometric shortest path problems*, Proc. 32nd ACM-STOC (Symposium on Theory of Computing), Portland, Oregon, May 2000, pp. 286–295.

[8] L. Aleksandrov, A. Maheshwari and J.-R. Sack, "An Improved Approximation Algorithm for Computing Geometric Shortest Paths", accepted for presentation at FTC'03, August 2003.

[9] A. Baltsan and M. Sharir, "On the Shortest Paths Between Two Convex Polyhedra", *Journal of the ACM*, **35**, January 1988, pp. 267-287.

[10] D.P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel Asynchronous Label-Correcting Methods for Shortest Paths", *Journal of Optimization Theory and Applications*, Vol. 88, No. 2, 1996, pp. 297-320.

[11] J. Canny and J. H. Reif, "New Lower Bound Techniques for Robot Motion Planning Problems", *Proceedings of the 28th IEEE Symp. on Foundations of Computer Science*, 1987, pp. 49-60.

[12] J. Chen and Y. Han, "Shortest Paths on a Polyhedron", *International Journal of Computational Geometry and Applications*, Vol. 6, 1996, pp. 127-144.

[13] G. Gallo and S.Pallottino, "Shortest Path Methods: A Unified Approach", *Mathematical Programming Study*, Vol. 26, 1986, pp. 38-64.

[14] S. Har-Peled, M. Sharir, and K.R. Varadarajan, "Approximating Shortest Paths on a Convex Polytope in Three Dimensions", *Proc. 12th Annual Symp. on Computational Geometry*, Philadelphia, PA, 1996, pp. 329-338.

[15] S. Har-Peled, "Approximate Shortest Paths and Geodesic Diameters on Convex Polytopes in Three Dimensions", *in Discrete & Computational Geometry*, Vol. 21, 1999, pp. 217-231.

[16] R.V. Helgason, and D. Stewart, "One-to-One Shortest Path Problem: An Emperical Analysis with the Two-Tree Dijkstra Algorithm", *Computational Optimization and Applications*, Vol. 2, 1993, pp. 47-75.

[17] J. Hershberger and S. Suri, "Practical Methods for Approximating Shortest Paths on a Convex Polytope in $\Re^3$", *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1995, pp. 447-456.

[18] M. Hribar, V. Taylor and D. Boyce, "Choosing a Shortest Path Algorithm", *Technical Report CSE-95-004*, Northwestern University, 1995.

[19] M. Hribar, V. Taylor and D. Boyce, "Performance Study of Parallel Shortest Path Algorithms: Characteristics of Good Decompositions", *13th Annual Conference on Intel Supercomputers User Group*, Albuquerque, NM, 1997.

[20] M. Hribar, V. Taylor and D. Boyce, "Parallel Shortest Path Algorithms: Identifying the Factors that Affect Performance", *Technical Report CPDC-TR-9803-015*, Northwestern University, 1998.

[21] M. Hribar, V. Taylor and D. Boyce, "Reducing the Idle Time of Parallel Shortest Path Algorithms", *Technical Report CPDC-TR-9803-016*, Center for Parallel and Distributed Computing, Northwestern University, 1998.

[22] D. Hutchinson, M. Lanthier, A. Maheshwari, D. Nussbaum, D. Roytenberg, J.-R. Sack, "Parallel Neighbourhood Modeling", *Proc. of the 4th ACM Workshop on Advances in Geographic Information Systems*, Minnesota, 1996, pp. 25-34.

[23] S. Kapoor, "Efficiently computing geodesic shortest paths", 31st ACM Symposium on Theory of Computing, 1999, pp. 770-77.

[24] M. Lanthier, A. Maheshwari and J.-R. Sack, "Shortest Anisotropic Paths on Terrains", *ICALP 99*, LNCS 1644, Prague, 1999, pp. 524-533.

[25] M. Lanthier, A. Maheshwari and J.-R. Sack, "Approximating Weighted Shortest Paths on Polyhedral Surfaces", *Algorithmica*, 30(4), 2001, pp. 527-562.

[26] M. Lanthier, "Shortest Path Problems on Polyhedral Surfaces", *Ph.D. Thesis*, Ch.5, School of Computer Science, Carleton University, Ottawa, Canada, 1999.

[27] A. Maheshwari, J.-R. Sack and H. Djidjev, "Link Distance Problems", *Handbook on Computational Geometry*, J.-R. Sack and J. Urrutia Eds., Elsevier Science, 2000, pp. 519-558.

[28] U. Meyer, "Buckets Strike Back: Improved Parallel Shortest Paths", *16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 15-19 April 2002, CD-ROM/Abstracts Proceedings, 2002.

[29] J.S.B. Mitchell and C.H. Papadimitriou, "The Weighted Region Problem: Finding Shortest Paths Through a Weighted Planar Subdivision", *Journal of the ACM*, **38**, January 1991, pp. 18-73.

[30] J.S.B. Mitchell, "Shortest Paths and Networks", *Handbook of Discrete and Computational Geometry*, J. Goodman and J. O'Rourke Eds., CRC Press LLC, Chapter 24, 1997, pp. 445-466.

[31] J.S.B. Mitchell, "Geometric Shortest Paths and Network Optimization", *Handbook on Computational Geometry*, J.-R. Sack and J. Urrutia Eds., Elsevier Science B.V., 2000, pp.633-702.

[32] D. Nussbaum, "Parallel Spatial Modeling", *Ph.D. Thesis*, School of Computer Science, Carleton University, Ottawa, Canada, 2000.

[33] D. Nussbaum, "A New Partitioning Method and its Applications for Spatial Modeling", journal version in preparation, School of Computer Science, Carleton University, Ottawa, Canada.

[34] L. Polymenakos, and D.P. Bertsekas, "Parallel Shortest Path Auction Algorithms", *Parallel Computing*, Vol. 20, 1994, pp. 1221-1247.

[35] K.V.S. Ramarao, and S. Venkatesan, "On Finding and Updating Shortest Paths Distributively", *Journal of Algorithms*, Vol. 13, 1992, pp. 235-257.

[36] M. Sharir, "On Shortest Paths Amidst Convex Polyhedra", *SIAM Journal of Computing*, **16**, 1987, pp. 561-572.

[37] Z. Sun and J. Reif, "BUSHWACK: An approximation algorithm for minimal paths through pseudo-Euclidean spaces", 12th ISAAC, LNCS 2223, 2001, pp. 160-171.

[38] J.L. Träff, "An Experimental Comparison of two Distributed Single-Source Shortest Path Algorithms", *Parallel Computing*, Vol. 21, 1995, pp. 1505-1532.

[39] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (August 1990), pp. 103-111.

[40] M. Ziegelmann, (Max-Planck Institut für Informatik, Saarbrücken, Germany) Constrained Shortest Paths and Related Problems PhD thesis, Universitt des Saarlandes, 2001.