Parallel Implementation of Bisector ϵ-Approximation Shortest Path Algorithm for Concurrent Queries

By

Hua Ye

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

March 1, 2005

Abstract

One of the most important factors in a parallel implementation of the shortest path computation, is how to partition the original graph to achieve better efficiency. Multi-dimensional Fixed Partition scheme introduced by Nussbaum offers an efficient partitioning solution. As an extension, SWP (Steiner Weighted Partitioning) re-partitions a page according to the weight of the page. This weight roughly represents the computation load in that page. Using this SWP, we have implemented a parallel application for the bisector $\epsilon - \beta$-approximation scheme.

Also, we introduce the concurrent PSP (parallel shortest path) computation, which handles concurrent shortest path queries simultaneously and leads to a higher speedup. The experimental results show that the parallel implementations using SWP are efficient and effective. We also show that this concurrent parallelization can achieve higher speedup and efficiency than the batch-mode parallelization.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor, Dr. Jörg-Rüdiger Sack and my co-supervisor, Dr. Doron Nussbaum, for their guidance and support throughout this research.

I would also like to thank the visiting professor, Dr. Lyudmil Aleksandrov for his ideas, suggestions and comments during the development of this thesis.

Also, I would like to thank my defense committee: Dr. Alan Williams from School of Information Technology and Engineering (SITE), University of Ottawa, Dr. Mark Lanthier from School of Computer Science, and the chair Dr. Tony White from School of Computer Science, for taking their time reading my thesis, and for their corrections and comments.

Besides, I would like to thank all members of the PARADIGM group at Carleton University for their ideas and suggestions. Especially for Ms. Hua Guo and Mr. Robert Kane, thank you so much for your countless help.

This research is supported by NSERC and SUN Microsystems.

Last but not least, I would like to dedicate this thesis to my parents and my wife who have always given me love, trust and support.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

1.1 Motivation 1

1.2 Problem definition 2

1.3 Contribution summary 4

1.4 Thesis organization 6

2 Background 7

2.1 Euclidean and weighted shortest path 7

2.2 Shortest path approximation 11

2.2.1 Definition and overview 11

2.2.2 Different Steiner points placement schemes 15

2.2.2.1 Lanthier et al.'s schemes 15

2.2.2.2 Bisector ϵ-approximation algorithm 16

2.3 Overview of parallel computing 21

2.3.1 Parallel computing definition 21

2.3.2 Parallel machines 21

2.3.3 Parallel performance 24

2.4 Performance factors of PSP computation 25

2.4.1 Algorithmic factors 25

2.4.2 Data-related factors 28

2.4.3 Machine-related factors 29
2.4.4 Implementation factors 29
2.5 Partitioning schemes 30
2.5.1 Planar graph partitioning 30
2.5.2 Multi-dimensional Fixed Partitioning 32
2.5.2.1 MFP partitioning 32
2.5.2.2 Mapping the MFP pages 34
2.5.2.3 MFP Algorithm 36
2.6 Termination determination in PSP 38

3 Concurrent PSP using SWP **42**
3.1 Bisector $\epsilon - \beta$-approximation 42
3.2 Steiner Weighted Partitioning 49
3.2.1 SWP overview 49
3.2.2 SWP Weighting scheme 50
3.2.3 SWP Algorithm 52
3.2.4 Advantages and disadvantages 54
3.3 Concurrent PSP implementation 55
3.3.1 Concurrent PSP 55
3.3.1.1 Motivation 55
3.3.1.2 Solution 57
3.3.2 Application design 59
3.3.3 Boundary Notification Model 64
3.3.3.1 Definitions and basic strategy 64
3.3.3.2 *Relax(u)* procedure 67
3.3.3.3 *HandleNodeExtracted(msg)* procedure 68
3.3.3.4 Conclusion 71

4 Experimental Results and Analysis **72**
4.1 Environment 72
4.2 Impact of SWP threshold 75
4.2.1 SWP partitioning results 75

4.2.2 Speedup and efficiency 77
4.2.3 Heap operation statistics 83
4.3 Results on different ϵ and β values 86
4.3.1 Speedup and efficiency 86
4.3.2 Length accuracy 92
4.4 Concurrency and query distribution 97
4.4.1 Test sets 97
4.4.2 Random distribution 101
4.4.3 Clustered query distribution case 1 106
4.4.4 Clustered query distribution case 2 111
4.4.5 Concurrency tests conclusion 111

5 Conclusion **117**

Bibliography **119**

List of Tables

2.1 Different SP alogrithms on Polyhedral Surfaces 13

4.1 SWP with different thresholds, africa5k, $\epsilon = 0.1$ 75
4.2 Comparison of SWP and MFP (3x3 processors, africa5k, $\epsilon = 0.1$) . . 76
4.3 Results on different SWP thresholds, africa5k, $\epsilon = 0.1$ 78
4.4 Number of heap operations (africa5k, ϵ=0.1) 83
4.5 Number of heap operations: detail information (africa5k, ϵ=0.1) . . . 84
4.6 Results on different ϵ and β (africa5k, 50 random queries) 87
4.7 Results on different ϵ and β with single processor (africa5k, 50 random queries) 93
4.8 Results with different concurrency thresholds 102
4.9 Results with different concurrency thresholds (50 case1 clustered queries) 107
4.10 Results with different concurrency thresholds (50 case2 clustered queries) 112

List of Figures

2.1 Characteristics of a weighted shorted path 10
2.2 Steiner points placement on angle bisector 17
2.3 Discrete paths 19
2.4 Expansion of the active border showing that processors may sit idle during a shortest path computation. Shaded area means computation is in progress. (Reproduction of Figure 2 of [28]) 34
2.5 Over-partitioning allows all processors to get involved quickly in the computational and remain involved longer, thus reducing idle time. (Reproduction of Figure 3 of [28]) 35
2.6 MFP page-processor mapping example: 3 levels 36

3.1 Impact of β on number of Steiner points: k 48
3.2 Examples of the weight of a TIN vertex u and the weight of a page . 51
3.3 System architecture: Host/Workers 60
3.4 pspWorker control flow diagram 61
3.5 pspWorker State Chart and Message Sequence Chart 63
3.6 An example of boundary node u is extracted 68

4.1 Processing Time Distribution Diagrams, africa5k, $\epsilon = 0.1$ 81
4.2 Speedup Diagrams, africa5k, $\epsilon = 0.1$ 82
4.3 Heap operations distribution among processors on the Beowulf machine 85
4.4 Speedup Diagrams, Sunfire, africa5k 89
4.5 Speedup Diagrams, Beowulf, africa5k 90
4.6 Timing on different ϵ and β values, africa5k 91

4.7 Impact of β, africa5k 96
4.8 Query distribution demonstration 100
4.9 Results with different concurrency thresholds (50 random distributed queries) 104
4.10 Results with different concurrency thresholds (500 random distributed short queries) 105
4.11 Results with different concurrency thresholds (long-paths-dominated, case1) 108
4.12 Results with different concurrency thresholds (balanced-distance, case1) 109
4.13 Results with different concurrency thresholds (short-paths-dominated, case1) 110
4.14 Results with different concurrency thresholds (long-paths-dominated, case2) 114
4.15 Results with different concurrency thresholds (balanced-distance, case2) 115
4.16 Results with different concurrency thresholds (short-paths-dominated, case2) 116

Chapter 1

Introduction

1.1 Motivation

Shortest path computation is one of the fundamental problems in computational geometry as well as other domains, such as geographical information system (GIS), network routing, and robotics. Approximation algorithms are suitable for terrain (geometric) shortest path computation because high-quality paths are favored over optimal paths that are "hard" to compute. Also geographic models are sometimes approximations of reality [6].

The approximation algorithms discretize the continuous domain by the placement of Steiner points as well as their interconnected edges. A shortest path computed over those Steiner points is called a discrete shortest path. Although approximation algorithms gain speed by reducing accuracy of the solution, the execution of their sequential implementations may still be slow because of the large number of vertices

in the derived graph, (original graph with the placement of Steiner points), if high accuracy is required. Furthermore, client/server technology and web technology have been widely applied. This allows users in different geographic locations to access the system at the same time and to submit their requests (e.g. shortest path queries) concurrently. Therefore, processing speed is a key issue that we address. This suggests the use of parallelization.

Another motivation for parallel shortest path computation is that the data for the computation might be too big to fit into the main memory of a single sequential computer. This causes additional overhead through the need to use external memory. Parallel implementations use a divide-and-conquer approach to distribute the data, which reduces this overhead.

One of the most important factors for the parallel terrain shortest path computation, is the partitioning strategy for the original graph. A "good" partitioning can increase the speedup by reducing the idle time of the processors. Thus in this thesis we study the parallelization of shortest path computations and focus in particular on aspects of data partitioning.

1.2 Problem definition

Let s and t be two vertices on a given, possibly non-convex, polyhedron $\mathcal{P}$, in $\Re^3$, consisting of n triangular faces $(f_1, f_2, ...f_n)$ on its boundary, where each face $f_i \in \mathcal{P}$ $(1 \leqslant i \leqslant n)$, is associated with a positive weight w_i. 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 the path $c(\Pi(s,t))$ is the sum of the Euclidean lengths of all segments multiplied by the corresponding face weight ([27]).

$$c(\Pi(s,t)) = \sum_{i=1}^{k}(|s_i| \times w(f(s_i))),$$

where k is the number of segments of the path; $|s_i|$ is the Euclidean length of the ith segment s_i; $f(s_i)$ is the face that contains segment s_i and $w(f(s_i)))$ is the weight of face $f(s_i)$.

In modelling the world, geographical information systems (GISs) frequently use Triangular Irregular Networks, or TINs, to represent surfaces of the earth. A TIN is a triangulation of a point set representing a geographical region. Each sample point has x, y coordinates and a z value (or surface value), which typically represents the height of the vertex. Topological relationships between points and their proximal triangles are also included.

We denote by G the TIN graph presenting the surface $\mathcal{P}$. Steiner points are placed on $\mathcal{P}$ and new edges (also on $\mathcal{P}$) are added among them. This derived graph is called G^*. Different Steiner points placement schemes and methods of introducing them have been proposed (see [5, 6, 7, 8, 27, 37]). An approximation of a shortest path is a discrete path computed in G^*. A partitioning over surface $\mathcal{P}$ generates several subgraphs G_s, whose union is G^*. The subgraph can be either connected or unconnected depending on which partitioning algorithm we are using. The intersection part of those unions are called *boundaries*. A One-to-One query $Q(s,t)$ is a query for the discrete shortest path between the source vertex s and the destination vertex t.

Given p processors: $P_1..P_p$, parallel computing solves a problem by dividing the problem into q ($q > p$) sub-problems and assigning each sub-problem to a processor. The parallel computation is called fine-grained if $p = q$, otherwise it is called coarse-grained. We denote by T_s as the time it takes to solve a problem by the best sequential algorithm; T_p as the time it takes a parallel algorithm to solve the problem using p processors. The *speedup*, s is defined as, $s = \frac{T_s}{T_p}$. The *efficiency* of a parallel algorithm is measured as $\frac{s}{p} \times 100\%$.

In the parallel shortest path computation, we partition the graph into p parts and assign each partition to each of the p processors. The parallel shortest path (PSP) computation cannot achieve very high speedup and efficiency due to the inherent sequential nature of the shortest path computation. However, a good partitioning scheme can let us get higher speedup and efficiency.

If a query arrives at the system when other queries are still running, we call this query a *concurrent* query. In previous work, concurrent queries are handled in a batch mode, i.e., processed one after another. In our thesis, the concurrent queries are handled concurrently, i.e., processed at the same time.

1.3 Contribution summary

We present a vertex-based partitioning algorithm extended from the Multidimensional Fixed Partition (MFP) scheme [34]. We call it Steiner Weighted Partitioning (SWP) because the partitioning is based on the "weights" of the TIN vertices, which roughly reflects the computation load. A weight of a TIN vertex is basically the number of Steiner points surrounding that vertex.

We implement the parallel ϵ-approximation shortest path algorithm, which was introduced by Aleksandrov et al. ([7, 8]). This is the first implementation (both sequential and parallel) for this bisector scheme ϵ-approximation shortest path algorithm. In our implementation, we introduce a variation regarding the placement of the Steiner points on the bisectors. Our variation significantly reduces the number of Steiner points. The length of the output path is within a small factor of the length of the ϵ-approximation shortest path. We use Dijkstra's shortest path algorithm ([12]) in the shortest path computation instead of the more efficient algorithm introduced in ([7, 8]), since our main purpose is on the performance of our partitioning scheme.

In our parallel implementation, we also introduce a model for the node-updates along the partition boundary. This model distributes the computational load for "relaxation" of the boundary nodes on different processors. It uses a small size message to reduce the communication load elegantly. Also, it can reduce the possibility of useless re-computation.

Our implementation can simultaneously process One-to-One concurrent queries with their priorities. We have done experiments with and without concurrency. The comparison of the experimental results shows that we can significantly increase efficiency when using the concurrency implementation.

Our experiments also show that when using the SWP, we obtain reasonable speedup and efficiency for PSP computations.

1.4 Thesis organization

Chapter 2 presents the history and previous work on terrain shortest path computation, followed by the review of parallelization of shortest path computation: parallel terminology, performance factors, and different partitioning schemes. In Chapter 3, we first describe our variation on the placement of Steiner points on bisectors; then we discuss the algorithm of SWP (Steiner Weighted Partitioning); later we exhibit the implementation of concurrent parallel shortest path computation. Chapter 4 shows the experimental results of the concurrent parallel shortest path computation using SWP. The last chapter gives the conclusion and suggests future work.

Chapter 2

Background

In this chapter, we review the history of sequential terrain shortest path computation as well as the parallel shortest path (PSP) computation. We begin with a general introduction to shortest path (Section 2.1) followed by the discussion of several approximation shortest path algorithms (Section 2.2). Later, we give a short introduction to parallel computing in Section 2.3, and discuss performance factors that affect the PSP computation in Section 2.4. After, we review different partitioning schemes for the PSP computation (see Section 2.5). In Section 2.6, we will discuss the termination determination that is an important issue in PSP computation.

2.1 Euclidean and weighted shortest path

There are many variations of shortest path problems in geometric domains. They can be categorized by the space dimensionality, the distance metric, the nature of the

surface, the type of the query, and the constraints imposed on the shortest path computation, etc. Shortest path problems in graphs have been well studied and general solutions with efficient implementations are readily available. Particular problems, which are well studied, include computing shortest path between two points inside a polygon and amidst polyhedral obstacles, reporting shortest paths on the surface of a convex or a non-convex polyhedron. In 2.5-D and 3-D domains, the complexity of the problems make them still intriguing research topics. Mitchell provides a survey on geometric shortest path problems in Chapter 15 of [38].

Let s and t be two vertices on a given, possibly non-convex, polyhedron $\mathcal{P}$, in $\Re^3$, consisting of n triangular faces $(f_1, f_2, ...f_n)$ on its boundary, where each face $f_i \in \mathcal{P}$ $(1 \leqslant i \leqslant n)$, is associated with a positive weight w_i. 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 the path $||\Pi(s,t)||$ is the sum of the Euclidean lengths of all segments multiplied by the corresponding face weight ([27]).

$$||\Pi(s,t)|| = \sum_{i=1}^{k}(|s_i| \times w(f(s_i))),$$

where k is the number of segments of the path; $|s_i|$ is the Euclidean length of the ith segment s_i; $f(s_i)$ is the face that contains segment s_i and $w(f(s_i)))$ is the weight of face $f(s_i)$.

Actually, the weighted shortest path can be treated as the generalization of the Euclidean shortest path. The Euclidean (unweighted) shortest path is a special case in which

all the faces on the surface have equal weight, e.g. 1.0.

Property 2.1 and Property 2.2 show two features of a Euclidean shortest path.

Property 2.1 *(Sharir and Schorr [39]) A Euclidean shortest path $\pi(s,t)$ may cross a face at most once. The total number of segments on the Euclidean shortest path is thus $O(n)$.*

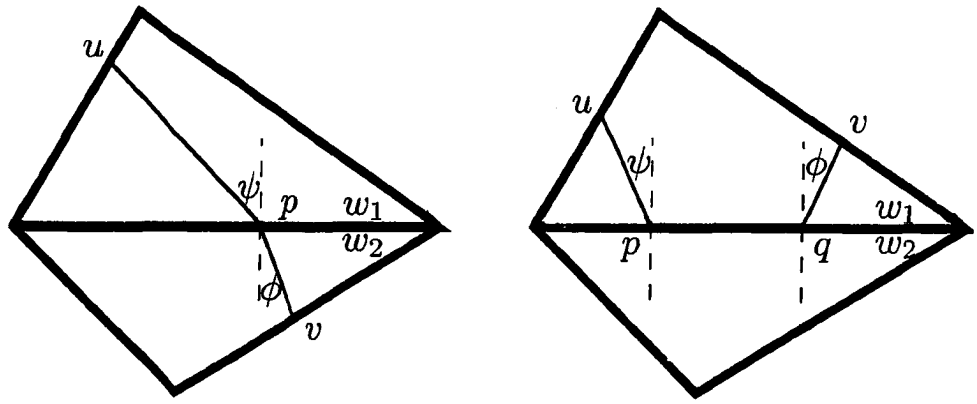
Property 2.2 *(Sharir and Schorr [39]) A Euclidean shortest path does not bend in the interior of a face.*

A weighted shortest path behaves differently and it does not hold Property 2.1 and Property 2.2, because it may contain "face-crossing" segments or "edge-using" segments([27]), and it may cross one face more than once (see Property 2.3 to Property 2.5 for details). A "face-crossing" segment is a segment that crosses a face joining two points on its boundary. The "edge-using" segment is a sub-segment of an edge([8]).

Property 2.3 *(Mitchell and Papadimitriou [33]) A weighted shortest path obeys Snell's Law [30] of refraction in which the path bends at the edges of $\mathcal{P}$. The amount by which it bends depends on the relative weights of the two faces adjacent to the edge (see Figure 2.1a).*

Snell's Law of Refraction: The path of a light ray passing through a boundary e between regions f and f', with indices of refraction α_f and $\alpha_{f'}$, obeys the relationship that $\alpha_f \sin\theta = \alpha_{f'} \sin\theta'$, where θ and θ' are the angles of incidence and refraction respectively.

The fact that light obeys Snell's Law comes from the fact that light seeks the path



(a) Face-crossing (b) Edge-using

Figure 2.1: Characteristics of a weighted shorted path

of minimum time. The index of refraction for a region is proportional to the speed with which light can travel through that region ([33]). The light path is a physical analogy to the weighted shortest path, where the index of refraction is an analogy to the weight of the surface. Therefore, weighted shortest paths must also obey Snell's Law.

In Figure 2.1a, the shortest path bends at point p. w_1 and w_2 are the weights of these two adjacent faces. By Snell's Law, $w_1 \sin\psi = w_2 \sin\phi$.

Property 2.4 *(Mitchell and Papadimitriou [33]) A weighted shortest path may travel along (i.e., critically use) an edge that is cheaper and then reflect back into the face (see Figure 2.1b).*

In Figure 2.1b, the shortest path $\Pi(u, v)$ uses the edge segment (p, q). At the bending points: p and q, Snell's Law still applies. We have,

$$w_1 sin\psi = w_2 sin\frac{\pi}{2} = w_1 sin\phi$$

$$\Rightarrow \psi = \phi = \arcsin\frac{w_2}{w_1}$$

Property 2.5 *(Mitchell and Papadimitriou [33]) A weighted shortest path may cross each face $\Theta(n)$ times. A weighted shortest path may consist of $\Theta(n^2)$ segments.*

In the weighted setting, shortest paths do not bend inside a face, however, segments of the paths may be "edge-using" as stated in the Property 2.4. The weight the "edge-using" segment is the smaller weight of the two adjacent faces separated by this edge.

2.2 Shortest path approximation

In this section, we introduce the definition of shortest path approximation and briefly review the previous work on different approximation shortest path algorithms. Then we introduce different discretization schemes followed by the discussion of the Bisector ϵ-approximation algorithm that we implement.

2.2.1 Definition and overview

Optimal terrain shortest paths are "hard" to compute, especially in weighted scenarios (see [10] for the NP-hardness proof on the unweighted optimal path problem

in 3-D space). There is no known solution for the non-approximate shortest paths on the weighted non-convex polyhedral surfaces at the time of this writing. On the other hand, the geographic models themselves are also an approximation of reality. Accuracy can be compromised to a certain degree depending on the application and its usage. Thus, most of the known research work has focused on approximation algorithms.

The shortest path approximation algorithms first discretize the continuous surface $\mathcal{P}$ (which is presented by a TIN graph G) by placing additional points (called Steiner points) on surface $\mathcal{P}$, and adding edges among Steiner points and the TIN vertices. The derived graph G^* after the discretization is also called approximation graph. The approximation algorithms then compute the shortest path on this approximation graph G^* by forcing the path using the vertices and edges in G^*. A discrete path is a path in the approximation graph: G^*. With the right placement of Steiner points, the approximate shortest path is the shortest discrete path computed from G^*. It is an approximate solution of the real shortest path on surface $\mathcal{P}$. The approximation path is called an ϵ-approximation, $\epsilon \geq 0$, of the real shortest path, if its cost is at most $(1 + \epsilon)$ times the cost of the real shortest path. The ratio of the cost of the approximation path over the cost of the shortest path is called the *approximation ratio.*

The Steiner points placement scheme is a key factor in a shortest path approximation. Different placement schemes result in different levels of accuracy, different running times, and different levels of implementation complexity. Typically the more Steiner points that are used, the more accurate the discrete path is. But with more Steiner points, the algorithm will run slower. Several Steiner points placement schemes are

Polyhedral Surface	Cost Metric	Approximation Ratio	Time Complexity	Reference
Convex	Euclidean	1	$O(n^3 \log n)$	[39]
Non-convex	Euclidean	1	$O(n^2 \log n)$	[32]
Non-convex	Euclidean	1	$O(n^2)$	[11]
Non-convex	Euclidean	1	$O(n \log^2 n)$	[26]
Convex	Euclidean	2	$O(n)$	[21]
Convex	Euclidean	$1+\epsilon$	$O(n \log \frac{1}{\epsilon} + \frac{1}{\epsilon^3})$	[3]
Convex	Euclidean	$1+\epsilon$	$O(n + \frac{\log n}{\epsilon^{1.5}} + \frac{1}{\epsilon^3})$	[18]
Convex	Euclidean	$1+\epsilon$	$O(\frac{n}{\sqrt{\epsilon}} + \frac{1}{\epsilon^4})$	[2]
Non-convex	Euclidean	$1+\epsilon$	$O(n^2 \log n + \frac{n}{\epsilon} \log \frac{1}{\epsilon} \log \frac{n}{\epsilon})$	[19]
Non-convex	Euclidean	$7(1+\epsilon)$	$O(n^{5/3} \log^{5/3} n)$	[41]
Non-convex	Euclidean	$15(1+\epsilon)$	$O(n^{8/5} \log^{8/5} n)$	[41]
Non-convex	Weighted	Additive	$O(n^3 \log n)$	[27]
Non-convex	Weighted	$1+\epsilon$	$O(n^8 \log \frac{n}{\epsilon})$	[33]
Non-convex	Weighted	$1+\epsilon$	$O(\frac{n}{\epsilon^2} \log n \log \frac{1}{\epsilon})$	[5]
Non-convex	Weighted	$1+\epsilon$	$O(\frac{n}{\epsilon} \log \frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \log n))$	[6]
Non-convex	Weighted	$1+\epsilon$	$O(\frac{n}{\epsilon} \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$	[37]
Non-convex	Weighted	$1+\epsilon$	$O(\frac{n}{\sqrt{\epsilon}} \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$	[7, 8]

Table 2.1: Different SP alogrithms on Polyhedral Surfaces

discussed in Section 2.2.2.

The following paragraphs give an brief overview of the algorithms for computing shortest paths on polyhedral surfaces. Table 2.1 lists different shortest path algorithms (both non-approximate and approximate) on polyhedral surfaces.

The shortest path problem on polyhedral surfaces was first posed by Sharir and Schorr ([39]), who provided an $O(n^3 \log n)$ solution for the convex Polyhedral Surface. Mitchell et al. improved it to $O(n^2 \log n)$ ([32]), and also extended its application on non-convex polyhedral surfaces. Chen and Han ([11]) further improved the time complexity to $O(n^2)$. The current best non-approximation algorithm for the Euclidean shortest path on non-convex polyhedral surfaces (Suggested by Kapoor ([26]))

has a near-linear time complexity of $O(n \log^2 n)$.

The second group in Table 2.1 shows the approximate algorithms for Euclidean shortest paths. Hershberger and Suri ([21]) presented a simple linear time algorithm for 2-approximate Euclidean shortest paths on a convex polyhedron. Har-Peled ([18]) extended this result to an ϵ-approximation shortest path algorithm that runs in $O(n + \frac{\log n}{\epsilon^{1.5}} + \frac{1}{\epsilon^3})$. Agarwal et al. provided another ϵ-approximation algorithm that runs in $O(n \log \frac{1}{\epsilon} + \frac{1}{\epsilon^3})$ ([3]) and later improved it to $O(\frac{n}{\sqrt{\epsilon}} + \frac{1}{\epsilon^4})$ ([2]). All of these algorithms are not easily extended to non-convex polyhedra because they rely heavily on the properties of convex polyhedra. For the Euclidean SP on non-convex polyhedral surfaces, Varadarajan et al. provide a $O(n^{5/3} \log^{5/3} n)$ algorithm that can produce an approximate path with length no more than $7(1 + \epsilon)$ in $O(n^{5/3} \log^{5/3} n)$ time and an approximate path with length no more than $15(1 + \epsilon)$ in $O(n^{8/5} \log^{8/5} n)$ time ([41]). Har-Peled ([19]) presented an ϵ-approximation algorithm that runs in $O(\frac{n}{\sqrt{\epsilon}} + \frac{1}{\epsilon^4})$ time.

For weighted shortest paths on non-convex polyhedral surface, Mitchell et al. ([33]) first presented an $O(n^8 \log \frac{n}{\epsilon})$ time ϵ-approximate algorithm using "continuous Dijkstra" paradigm. Lanthier et al. ([27]) and Aleksandrov et al. ([5, 6]) adopted a natural approach to discretize the surface by placing the Steiner points on the TIN edges (see Section 2.2.2 for details). Lanthier et al. ([27]) produce an approximate shortest path $\Pi'(s,t)$, such that

$$||\Pi'(s,t)|| \leq \beta(||\Pi(s,t)|| + W|L|),\ \beta > 1$$

where W is the maximum weight among all face weights of the surface and $|L|$ is the Euclidean length of the longest edge of the surface. It runs in $O(n^3 \log n)$

time. The ϵ-approximation algorithm presented by Aleksandrov et al. ([6]) runs in $O(\frac{n}{\epsilon}\log\frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}}+\log n))$ time. Reif and Sun ([37]) improve it to $O(\frac{n}{\epsilon}\log\frac{n}{\epsilon}\log\frac{1}{\epsilon})$. Recently, Aleksandrov et ([7, 8]) present another ϵ-approximation algorithm that runs in $O(\frac{n}{\sqrt{\epsilon}}\log\frac{n}{\epsilon}\log\frac{1}{\epsilon})$ time. This algorithm is called the bisector ϵ-approximation algorithm in our thesis, because it places the Steiner points on the angle bisectors of the triangle faces. The detail of this algorithm will be discussed in Section 2.2.2.2.

2.2.2 Different Steiner points placement schemes

In this section, we introduce several Steiner points placement schemes including the scheme for the Bisector ϵ-approximation algorithm that we implement.

2.2.2.1 Lanthier et al.'s schemes

Lanthier et al. presented a very simple approximation solution ([27]) by placing Steiner points on every edge in G. They provide the following schemes for the placement of the Steiner points: FIXED scheme and INTERVAL scheme. They also provide a scheme (SPANNER scheme) for the interconnections among the Steiner points.

FIXED Scheme: In FIXED Scheme, the Steiner points are placed on the edges with an equal number of Steiner points on each edge. Then in each face (triangle), a nearly complete graph is built among the Steiner points and the three end points of the triangle. The approximate shortest path $\Pi'(s,t)$ can be computed in $O(n^5)$ time such that, $||\Pi'(s,t)|| \leq ||\Pi(s,t)|| + W|L|$, where W is the maximum

weight among all face weights of the surface and $|L|$ is the Euclidean length of the longest edge of the surface.

INTERVAL Scheme: The INTERVAL Scheme forces the intervals between adjacent Steiner points on an edge to be of equal length, approximately $\frac{|L|}{m+1}$, where m is the number of Steiner points. The connectivity is built in the same way as in the FIXED scheme. The interval scheme allows a significant decrease in the number of Steiner points placed while maintaining the same path accuracy as with the FIXED scheme.

SPANNER Scheme: SPANNER Scheme is used to reduce the number of edges added among the Steiner points that are generated by the FIXED Scheme or INTERVAL Scheme. The cost of the produced approximate shortest path is $||\Pi'(s,t)|| \leq \beta(||\Pi(s,t)|| + W|L|)$, $\beta > 1$, where W is the maximum weight among all face weights of the surface and $|L|$ is the Euclidean length of the longest edge of the surface. The path can be computed in $O(n^3 \log n)$ time.

2.2.2.2 Bisector ϵ-approximation algorithm

Aleksandrov et al. have done extensive work on ϵ-approximations of shortest paths ([4, 5, 6, 7, 8]). We implement their best algorithm ([8]) and parallelized it to see how well it performs when combined with the Steiner Weighted Partitioning (SWP) scheme. So here after, we discuss how it places the Steiner points on the angle-bisectors of the triangles, and how it adds connectivity among the Steiner points. Figure 2.2(a) and (b) show the Steiner points placements on a bisector. We use the same method to place Steiner points on the other bisectors. $E(v)$ (the thick outer

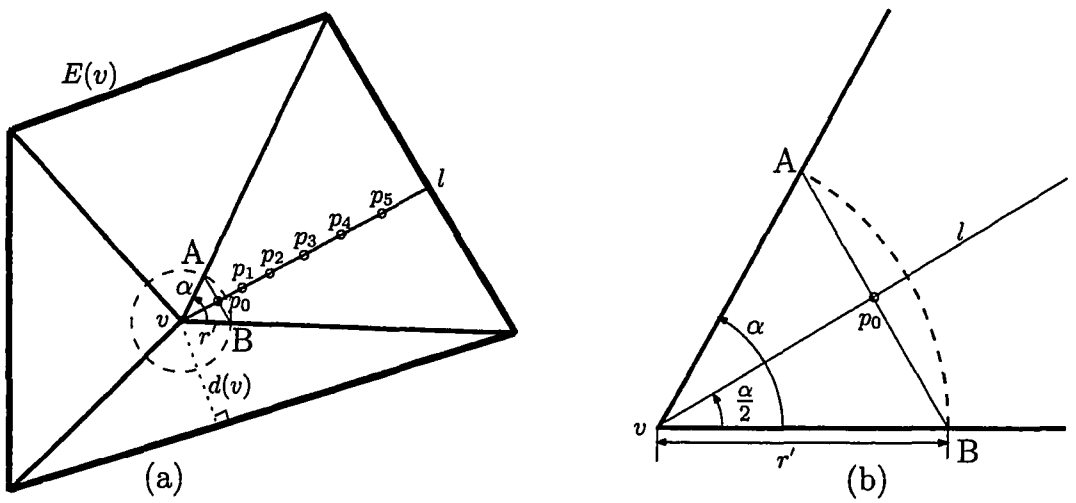


Figure 2.2: Steiner points placement on angle bisector

polygon in Figure 2.2(a)) is the set of edges of all the triangles incident to v minus the edges incident to v. The distance $d(v)$ is defined as the minimum Euclidean distance from v to the edges in $E(v)$. Line l is the bisector of α (angle noted by the arrow). Let $r(v) = \frac{d(v)}{7}$. The value 7 is chosen because it satisfies the theorem that, using the bisector ϵ-approximation scheme, the cost of the approximation shortest path is at most $(1+\epsilon)$ times of the cost of the actual shortest path. (For more details, see Theorem 3.2 in [8].) The radius of the dashed circle: r', can be obtained by,

$$r' = \epsilon \times r(v) = \epsilon \times \frac{d(v)}{7} \tag{2.1}$$

Segment AB in the figure is one of the edge of vertex-vicinity of v; its intersection with l is the first Steiner point placed on l, named p_0. As illustrated in Figure 2.2(b),

we now have,

$$|vp_0| = r' \cos\frac{\alpha}{2} = \frac{\epsilon d(v)}{7} \cos\frac{\alpha}{2} \tag{2.2}$$

The other Steiner points are placed along l such that,

$$|vp_i| = \lambda^i \times |vp_0| \quad for\ i = 1, ..., k \tag{2.3}$$

$$\lambda = (1 + \sqrt{\frac{\epsilon}{2}} \sin\frac{\alpha}{2}) \tag{2.4}$$

Because we do not place Steiner points p_i outside the triangle that contains the bisector l, we are sure that $|vp_i| \leqslant |l|$, where $|l|$ is the length of l. From Equation 2.3, we have

$$k = \lfloor \log_\lambda \frac{|l|}{|vp_0|} \rfloor \tag{2.5}$$

They also prove that,

$$NSP < C(P)\frac{n}{\sqrt{\epsilon}} \log_2 \frac{2}{\epsilon} \tag{2.6}$$

NSP is the total number of Steiner points; $C(P) = 4.83\Gamma logL$; Γ is the average of the reciprocals of the sinuses of angles in surface $\mathcal{P}$, i.e., $\Gamma = \frac{1}{3n}\sum_{i=1}^{3n} \frac{1}{\sin\alpha_i}$; L is the maximum of the ratios $2|l(v)|/r(v)$ among all vertices $v \in \mathcal{P}$. For a given $\mathcal{P}$, Γ and L are all constants, therefore, $C(P)$ is a constant only relevant to the surface $\mathcal{P}$. Then we can use Equation 2.6 to estimate the number of the Steiner points.

For the derived Graph $G^* = (V^*, E^*)$, we have already produced V^* by the Steiner points placement. Now, we need to add edges to form E^*. We say two bisectors are neighbors if the angles they split share a common edge of $\mathcal{P}$. We build edges between any pair of Steiner points p and q lying on neighbor bisectors.

There are three scenarios for the node pair (p, q): p and q are on different triangles; p

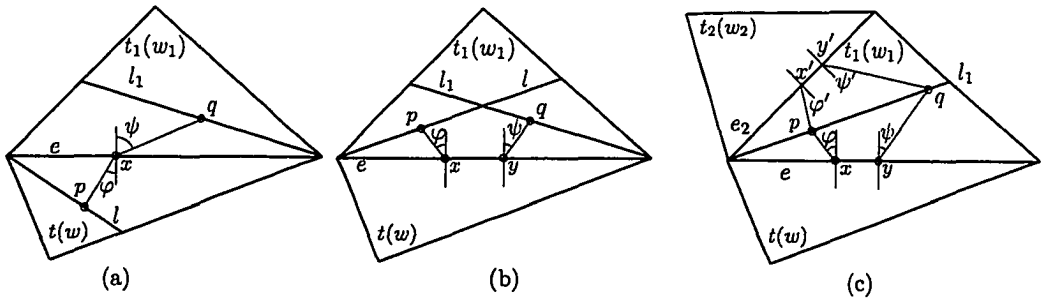


Figure 2.3: Discrete paths

and q are on different bisectors on the same triangle; p and q are on the same bisector. Figures 2.3 illustrates these three scenarios of the discrete paths in G^* separately.

p and q are on different triangles (Figure 2.3(a)): A pair of neighbor bisectors l and l_1 lying on different triangles. The shortest path between p and q bends at a point x on e, so that the angles φ and ψ satisfy the Snell's law: $w \sin\varphi = w_1 \sin\psi$. The cost of the edge (p, q) in G^* is equal to $w|px| + w_1|xq|$.

p and q are on different bisectors on the same triangle (Figure 2.3(b)): We only add an edge between p and q when $w < w_1$ and there exists two bend points x and y on e, which must satisfy $\varphi = \psi$ and $\sin\varphi = \sin\psi = w/w_1$. The cost of the edge (p, q) in G^* is

$$||pq|| = ||px|| + ||xy|| + ||xq|| = w_1|px| + w|xy| + w_1|yq| = w_1(|px| + |yq|) + w|xy|$$

p and q are on the same bisector (Figure 2.3(c)): The situation is quite similar to the previous one, except that there are two candidates for the shortest path between p and q, one crossing e and the other crossing e_2. We only take

the shorter one into G^*.

The following Theorem gives the upper bound of the number of nodes and edges inside the derived graph G^*.

Theorem 2.1 (Lemma 3.1 in [8]) *The number of nodes in G^* is $O(\frac{n}{\sqrt{\epsilon}}log\frac{1}{\epsilon})$ and the number of edges is $O(\frac{n}{\epsilon}log^2\frac{1}{\epsilon})$.*

If we apply Dijkstra's algorithm on G^* using Fibonacci heaps, it runs in $O(|E^*| + |V^*|log|V^*|)$ time. We can estimate $|E^*|$ and $|V^*|$ from Theorem 2.1, then we have,

Theorem 2.2 *Let $\mathcal{P}$ be a weighted polyhedral surface with n triangular faces and $\epsilon \in (0,1)$. When using Dijkstra's algorithm, shortest paths from a vertex v_0 to all other vertices of $\mathcal{P}$ can be approximated within a factor of $(1+\epsilon)$ in $O(\frac{n}{\epsilon}log\frac{n}{\epsilon}log\frac{1}{\epsilon})$ time ([8]).*

The paper [8] also provides an algorithm other than Dijkstra's to obtain better time complexity as shown in the following theorem. For simplification, our experiments focus on the parallelization, and we use Dijkstra's algorithm.

Theorem 2.3 (Theorem 4.2 in [8]) *Let $\mathcal{P}$ be a weighted polyhedral surface with n triangular faces and $\epsilon \in (0,1)$. Shortest paths from a vertex v_0 to all other vertices of $\mathcal{P}$ can be approximated within a factor of $(1+\epsilon)$ in $O(\frac{n}{\sqrt{\epsilon}}log\frac{n}{\epsilon}log\frac{1}{\epsilon})$ time.*

2.3 Overview of parallel computing

2.3.1 Parallel computing definition

In the sequential machine (Von Neumann machine), a processor reads instructions and data from memory, manipulates the data, and writes the results back into memory. The data manipulation is accomplished by executing the instructions in sequence one at a time. This design is geared for one processor only. However, more than one processor can be used to manipulate the data simultaneously.

The precise definition of parallel computing is still in debate. Roughly, "parallel computing" is a computer system in which interconnected processors perform concurrent or simultaneous execution of two or more processes ([31]).

2.3.2 Parallel machines

Flynn's taxonomy ([15]) of parallel machines is widely used. He classified computers with respect to the number of their instruction streams and data streams: SISD, SIMD, MISD and MIMD.

SISD: Single Instruction stream/ Single Data stream. Conventional single-processor computers are classified as SISD systems.

SIMD: Single Instruction stream/ Multiple Data streams. In SIMD machines, all processors execute the same instruction stream on a different piece of data. This approach can reduce both hardware and software complexity but is only appropriate for specialized problems characterized by a high degree of regularity.

MISD: Multiple Instruction streams/ Single Data stream. Very few machines are in this category. It might be designed for a fail-safe environment: each processor performs the same operation and checks the results with other processor(s) to ensure correctness.

MIMD: Multiple Instruction streams/ Multiple Data streams. Multiple processors are fetching their own instructions and the data specified by those instructions. This category covers most of the parallel machines used today.

A further categorization of parallel systems is whether the memory is shared or distributed.

The parallel machines with shared memory are also called PRAMs (Parallel Random Access Machines) in theoretical computer science for theoretical studies of parallel algorithms. A PRAM system has several processors accessing the common memory and exchanging data through it. Any pair of processors can communicate in constant time. Depending on whether two processors can access (read/write) the same memory simultaneously, PRAMS are categorized into the following models:

1. The exclusive-read, exclusive-write (EREW) PRAM, where it disallows simultaneous reading or writing on same memory location
2. The concurrent-read, exclusive-write (CREW) PRAM, where simultaneous reading on same memory location is allowed, but not simultaneous writing
3. The concurrent-read, concurrent-write (CRCW) PRAM, where both simultaneous reading and writing are allowed

There is a problem with the PRAM: the scalability of the system. The switch between the processors and memory can create a bottleneck of the throughput: it becomes more expensive or slower. The balance between memory size and number of processors becomes an additional configuration issue.

An SMP(Symmetric Multi-Processing) machine is a computer system that has two or more processors connected in the same cabinet, managed by one operating system, sharing the same memory, and having equal access to input/output devices. One advantage of SMP systems is its scalability; additional processors can be added as needed. But it has the disadvantage that once one processor is down, the whole system is down. SunFire 6800 is an example of SMP machine.

In a distributed memory system, processors have their own memory segments, and a processor can only access its own memory. Because there is no common bus or switch between the processors and memory, each processor can fully use its bandwidth to access its own memory. But when a processor needs information that is located in another processor's memory, it must communicate with the other processor via message passing. This kind of organization has good scalability because there is no inherent limit on the number of the processors; the size of the system is now constrained only by the interconnection among the processors. The drawback of distributed memory machines is the difficulty of the interprocess communication. Messages need to be well designed. Overhead should be expected on the construction/deconstruction and the delivery of those messages.

A computer cluster is a group of connected computers that work together as a unit. It involves connecting different systems, possibly of different architectures, together to form a parallel machine. This is typically done through a special software interface

and high speed network connections or switches. Special programming through the Message Passing Interface (MPI) library is often required. Linux clusters and Beowulf clusters are common examples.

2.3.3 Parallel performance

Runtime performance is typically a good indicator of the usefulness of a parallel algorithm. It is measured by speedup and efficiency. We denote by T_s as the time it takes to solve a problem by the best sequential algorithm; T_p as the time it takes a parallel algorithm to solve the problem using p processors. The *Speedup*, s is defined as, $s = \frac{T_s}{T_p}$. The *Efficiency* of a parallel algorithm is measured as $\frac{s}{p} \times 100\%$.
It is a common practice to use $s = \frac{T_1}{T_p}$ to measure the speedup, where T_1 is the amount of time it takes a parallel algorithm to solve the problem using a single processor. There are several reasons for choosing T_1 instead of T_s. First, an implementation of the best sequential algorithm may not be available. The implementation of the best sequential algorithm may be a very time-consuming task and the difference between T_1 and T_s may be only a constant factor. Second, the best sequential algorithm may be a theoretical one. Namely, the asymptotic linear time (Big - Oh) may be better but the size of the input data must be very large which makes such an algorithm impractical.
Ideally, we expect to get a speedup $s \approx p$, but in reality, there are a few factors that will introduce the inefficiency. These include the insufficient concurrency, the delays due to the communication, overhead incurred in synchronization, and control among the processors. Also, many algorithms might be *inherently sequential* (a.k.a *sequential in nature*) that implies a low speedup when parallelized (e.g. a BFS search

on a tree, or Dijkstra's shortest path algorithm).

2.4 Performance factors of PSP computation

There are many factors affecting the performance of a PSP implementation. Lanthier et al. presented a good summary of these factors in [28]. They categorized the factors as shown in Section 2.4.1 to 2.4.4. Most of the research has been focusing on analyzing the effects of different parts of the factors in isolation and making various assumptions on the rest of the factors. Such difference makes it difficult to perform a thorough comparison and also isolates the impact of each factor on the overall performance.

2.4.1 Algorithmic factors

- **Shortest path paradigm:**

According to the number of source and destination points, shortest path problems can be categorized into one-to-one, one-to-few, one-to-all, few-to-all, and all-to-all problems. Label-setting algorithms (typically Dijkstra's algorithm), and Label-correcting algorithms (typically Bellman-Ford algorithm) are two kinds of algorithms usually applied to solve the shortest path problem. In label-setting algorithms, the final optimal shortest path distance from the source node to the destination node is determined once the destination node is scanned ([16]). Therefore, when it is only necessary to compute a one-to-one shortest path using a label-setting algorithm, this algorithm can be terminated as soon as the destination node is scanned. There is no need to exhaust all nodes on the entire network. In contrast, a label-correcting algorithm

treats the shortest path distance estimates of all nodes as temporary and converges to the final one-to-all optimal shortest path distances until its final step. Thus using a label-correcting algorithm, a one-to-one shortest path cannot be determined before the shortest paths from the source node to all other nodes are determined.

There are many papers addressing the PSP issues. Many of them aimed at the one-to-all and few-to-all shortest path problems (see [1, 9, 24, 25, 36, 40]). Bertsekas et al. [9] claimed that when only a few destinations are used, the label-setting algorithms are better suited than the label-correcting algorithms. Similar comments can be found in [20] and [35].

Hribar et al. [22] have shown that the label-correcting algorithms have inconsistent performance for their networks. They claim that there is no one shortest path algorithm that is best for all shortest path problems, nor for problems with similar data set sizes. However, they attempt to give an indication where the label-setting algorithms outperform label-correcting algorithms. The indicator is based on the expected number of iterations per node. Later, they [24] demonstrated that label-setting algorithms perform best for distributed memory parallel machines when large data sets (16k to 66k vertices) are used.

- **Shortest path queue distribution:** Either Label-correcting or Label-setting algorithms can be implemented using queues. There are variations in the number and size of queues, and in the order that items are removed. Single-queue means all processors operate on a single queue. Multiple-queue means each processor has its own queue.

Bertsekas et al. [9] experimented with single vs. multiple queues for the label-correcting algorithms in the shared memory settings. They show that their multiple-queue strategy is better due to the reduction in queue contention. Their experiments also show that the asynchronous algorithms outperform the synchronous algorithms. Träff ([40]) compared a synchronous strategy with an asynchronous strategy using a label-setting algorithm. He found that synchronous strategy would perform poorly in systems where the cost of communication is high.

• **Communication frequency:** The number of messages and frequency of communication steps that each processor performs throughout the algorithm execution is also an important performance issue. In some algorithms (e.g. [24]), a processor first empties its queue in a single step and sends the boundary vertices to the neighbor process(s). Another approach is whenever a processor computes the vertices on the boundary, it immediately sends the boundary vertices to the neighbors. The first approach results in a few big messages passing among the processors and in terms of the message delivery, it is more efficient than the delivery of the more small messages generated by the second approach. However, the first approach may cause some processors to sit idle and wait for the other processors to empty their queues. In contrast, the second approach may let the processors get involved more quickly. Another advantage of the first approach is that, much of the work done by a processor may be discarded by the messages coming from other processors. In such a case, emptying the local queue first and then sending out messages might not be an effective choice.

• **Termination detection:** The termination detection frequency defines the number of times that the algorithms check for termination. With higher frequency detection,

the algorithm will spend more time on determining whether the shortest path computation can be terminated to avoid unnecessary computation. The algorithm can speed less time on the termination detection with lower frequency but it might speed more time on the unnecessary computation. Hribar and Taylor ([24]) state that high detection frequencies are best when a large number of processors are used and low frequency otherwise.

• **Cost function:** The cost function used in the shortest path calculation can also affect the performance. When a more computationally-intensive cost function is used, the percentage of computation time increases, which often leads to more efficient use of the processors.

2.4.2 Data-related factors

There is a strong correlation between the size of the data sets and the overall performance. Most of the researchers have tried to understand how an algorithm scales with large amounts of data in comparison to small amounts of data.

The relative locations of sources and destinations is another important factor. If a path between a source and a destination lies on one processor, then we may not benefit from parallelization because only one processor gets involved in the computation and other processors stay idle. If the path is long enough, then more processors get involved in the computation and a speedup should be achievable.

The data partitioning plays another key role in the performance. Hribar et al. ([23]) state that determining good decompositions is crucial for all parallel applications. For their algorithm, which is 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 vertices. Another another important issue is the processor mapping. The mapping from partitions to processors can be arbitrarily selected, or determined by certain schemes. Different schemes may result in different levels of communication among the processors. In Section 2.5, we will discuss the partitioning and mapping schemes in more detail.

2.4.3 Machine-related factors

One of the reasons why it is often difficult to compare results from different researchers is the difference of the parallel machine architectures and configurations that they are using. Parallel machines have different number of nodes (e.g., fine or coarse grained), different CPU types and speed, different interconnection topologies and strategies.

2.4.4 Implementation factors

There are many implementation-specific factors: the message parsing libraries, numeric computation libraries, etc. For example, one may choose different message passing libraries such as MPI, MPICH, PVM, etc. Even at the geometric primitive level such as calculating the distance from a point to a line, different implementation approaches will lead to different runtime performance.

2.5 Partitioning schemes

There are two major categories of partitioning: domain-driven partitioning and data-driven partitioning. A domain-driven partitioning is a decomposition according to the domain itself regardless to the location or connectivity of the data. In contrast, a data-driven partitioning is a decomposition based on the nature of the data. Partitioning using a K-d tree and partitioning by using a planar graph separator are just two examples of data-driven partitioning. In the following subsections, we will discuss different partitioning schemes related to parallel shortest path computation, such as the planar graph partitioning ([4, 17]) and the MFP ([34]).

2.5.1 Planar graph partitioning

A graph is said to be *embedded in a surface* S when it drawn on S so that no two edges intersect. A graph is *planar* if it can be embedded in a plane. A *graph separator* is a small set of vertices whose removal will divide the graph into roughly equal parts. Lipton and Tarjan first formally studied the graph partitioning problem in [29]. They showed that in any n-vertex planar graph with nonnegative vertex weights summing to $w(G)$, there exists a separator of no more than $\sqrt{8n}$ vertices, whose removal divides the graph into two components none of whose weight is more than $\frac{2}{3}w(G)$. This theorem forms the basis of all known separator theorems for planar graphs ([17]). Lipton and Tarjan later extended this theorem to graph bisections by showing that in any n-vertex planar graph with nonnegative vertex weights summing to $w(G)$, there exists a separator of no more than $\sqrt{8n}(1-\sqrt{2/3})$ vertices whose removal divides the graph into two components none of whose weight exceeds $\frac{1}{2}w(G)$.

In [13], Djidjev improved the bound on the size of the separator to $\sqrt{6n}$ compared to $\sqrt{8n}$, the bound proved by Lipton and Tarjan. He also generalized (in [14]) Lipton and Tarjan's results to planar graphs with costs and weights on vertices, where costs are used to evaluated the size of the separator and weights are used to evaluate the sizes of the components. He stated that in a planar graph G with nonnegative vertex weights and costs, there exists a vertex set of total cost $\sqrt{8}\sqrt{\sum_{v\in V}(c(v))^2}$ whose removal leaves no component of weight exceeding $\frac{2}{3}w(G)$, where $c(v)$ denotes the cost of a vertex v ([14]). It states that in any planar graph G with nonnegative vertex weights and costs, there exists a 1/2-vertex separator of G of cost $O(\sqrt{\sum_{v\in V}(c(v))^2})$, which can be found in $O(|G|)$ time.

Most of the graph-separator partitioning algorithms are only for graph bisection, i.e. vertices of a graph are separated into two halves. The bisection is applied recursively on each half of vertices until the partitions are small and numerous enough. In contrast, Alexandrov et al. [4] introduce an approach for separating a graph into any number of pieces and it does not apply recursive bisection.

Let $G = (V, E)$ be an n-vertex connected planar graph with real-valued positive weights $w(v)$ and costs $c(v)$ associated with its vertices. A set of vertices (edges) S is called a *t-separator* if their removal from G leaves no component of weight exceeding $tw(G)$. *Cost of t-separator* is the sum of the costs of vertices (edges) in the separator. The other requirement in terms of quality of such a separator is that the cost of a separator must be as small as possible.

In ([4, 17]), Alexandrov et al. construct a vertex t-separator of G and design an algorithm for it. They also develop a "cost-reducing technique" to reduce the cost of the separator. Then they design an algorithm to obtain an edge t-separator from vertex

t-separator. Their vertex t-separator theorem states that for any $t \in (0,1)$, there exists a t-separator S_t such that $c(S_t) \leq 4\sqrt{2\sigma(G)/t}$ where $\sigma(G) = \sum_{v \in V} (c(v))^2$. S_t can be found in $O(n)$ time in addition to the time for computing an SSSP (Single Source Shortest Path) tree in G.

2.5.2 Multi-dimensional Fixed Partitioning

The Multi-dimensional Fixed Partitioning (MFP) is a spatial data structure that was introduced by Nussbaum ([34]). It is a recursive spatial data structure tailored for parallel computation. In the sections to follow, we will introduce the basic idea of MFP partitioning, followed by the description of page-to-processor mapping and the discussion of the MFP algorithm.

2.5.2.1 MFP partitioning

In contrast to a data-driven partition, MFP is a domain-driven partition and a generalization of other domain partition structures such as octree and quadtree. It generates the decomposition according to the number of processors involved instead of a fixed dimensional grid.

MFP is geared towards a logical mesh or a logical torus computer. In a 2-D mesh network with $p = R \times C$ processors, processors are connected in a grid of R rows and C columns. A 2-D torus network is a 2-D wrapped-around mesh which has a connection between two processors at the end of each row or column of the grid.

MFP partitioning is a process of a recursive decomposition of the domain. Let $\mathcal{P}$ be the surface with n triangular faces and let the Minimum Bounding Rectangle (MBR),

which is denoted by $MBR(x_{min}, y_{min}, x_{max}, y_{max})$, be the domain size in which $\mathcal{P}$ is embedded. MFP partitions the surface $\mathcal{P}$ among $p = R \times C$ processors (R denotes the number of rows and C denotes the number of columns). For ease of description, we can image that the processors are arranged in a grid-like topology; and p_{rc} represents the processor in row r, column c of the mesh where p_{00} is the bottom left.

The top level (level 0) represents all of $\mathcal{P}$ and the first level of MFP divides $\mathcal{P}$ into an $R \times C$ pages such that, each page is of equal size: $(x_{max} - x_{min})/C \times (y_{max} - y_{min})/R$. Vertices, edges, and faces are stored in the page that contains them. Special care is given for vertices on page boundaries, which guarantees that every vertex in $\mathcal{P}$ is inside only one page. For example, we can determine each vertex on the page boundaries to be inside the page on the left and/or upper side. Faces, which span multiple pages (called border faces or border triangles), are shared by the pages which they cross.

If we stop the partitioning at the first level and assign each of the p pages to one processor (e.g. as shown in Figure 2.4), it is possible that we would have large processor idle time when used with a parallel shortest path algorithm. So we over-partition $\mathcal{P}$ by re-partitioning the page into sub pages. When there are a number of levels in the MFP structure, the resulting mapping assigns several pages to each processor. It can let all processors get involved the computation quickly, hence reducing the idle time of the processors (as shown in Figure 2.5).

We define the weight of a page (or tile size) as the number of the TIN vertices that are contained in this page. MFP re-partitions a page when the weight of this page is greater than some threshold. Lanthier et al. ([28]) did not show exactly how to choose such a threshold, but only stated that, the "best" threshold depends on factors

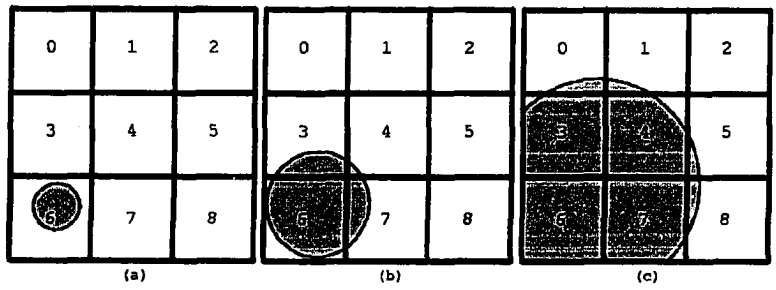


Figure 2.4: Expansion of the active border showing that processors may sit idle during a shortest path computation. Shaded area means computation is in progress. (Reproduction of Figure 2 of [28])

such as terrain size, number of processors, and the degree of clusterization. The size of the pages depends largely on a variety of operations that are required to be performed on the data. They claim that too fine and too coarse over-partitioning both have negative effects on the parallel speedup. If the grid is too fine, communication time among processors will increase. If it is too coarse, idle time of the processors will increase since a processor may have too much work to do before it can pass the computation to other processors. Their experimental results strongly support these claims.

2.5.2.2 Mapping the MFP pages

Each page in the MFP data structure is mapped onto a processor. The assignment of pages onto the processors is termed *page mapping* or *mapping* in short.

The simplest page-to-processor mapping might be the trivial mapping: pages in the first row are mapped to processor 1 to C (C is the number of columns) sequentially; then pages in the second row are mapped to processor $(C+1)$ to $(2 \times C)$ .... Another

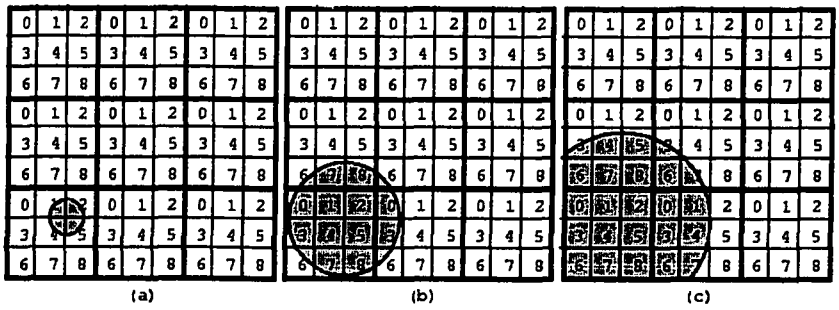


Figure 2.5: Over-partitioning allows all processors to get involved quickly in the computational and remain involved longer, thus reducing idle time. (Reproduction of Figure 3 of [28])

example of a mapping scheme may be a randomized mapping, in which pages are randomly mapped to processors.

The mapping scheme is important because it directly affects the communication between the processors. A good mapping scheme should: (1) minimizing processor hops and (2) avoiding bottlenecks. MFP mapping addresses both criteria. We describe now how processors are mapped using the MFP mapping. Let $page_{ij}$ be the page at row i, column j in its parent page, where $page_{00}$ is the bottom left page; p_{rc} be the the processor mapped with the parent page; and *level* is the level of $page_{ij}$ in the MFP tree. If *level* is even, a backward mapping is used: assigns $page_{ij}$ to processor $p_{((R+r-i) mod R)((C+c-j) mod C)}$; if *level* is odd, a forward mapping is used: assigns $page_{ij}$ to processor $p_{((r+i) mod R)((c+j) mod C)}$.

Figure 2.6 shows an example of a mapping for the 3-level MFP tree. Different colors represent different processors. Once this mapping is done, the partitions for the processors can be easily generated by just collecting those pages. For example, the partition for processor "white", is the union of white areas in the figure.

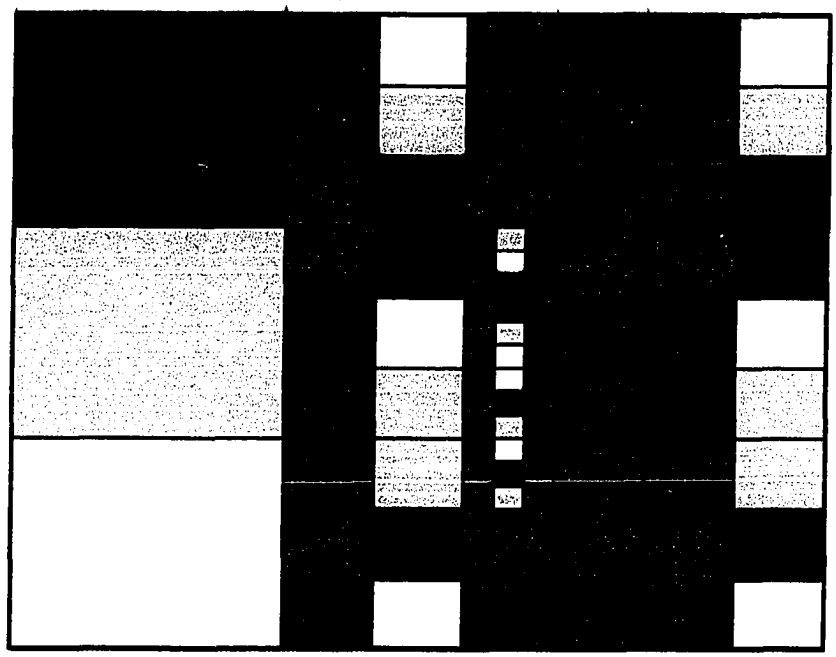


Figure 2.6: MFP page-processor mapping example: 3 levels

In [34], Nussbaum also compared MFP mapping with trivial mapping and randomized mapping. He claims that MFP mapping outperforms the others both in dilation and congestion in torus environment. He also claims that even in environments that use cluster computers, the MFP scheme has low node congestion and high serialization factor when compared with the other schemes, which in return reduces the competition for resources in the switch.

2.5.2.3 MFP Algorithm

In the previous sections, we have described the basic idea of MFP including its partitioning and mapping. In this section, we will discuss the MFP Algorithm.

Algorithm 2.1 MFP a TIN

```
1: Q ← ∅
2: create the root of MFP tree ( page_00 );
3: page_00.processor ← p_00;
4: page_00.weight ← |V|;
5: load V into page_00.points;
6: Q.insert(page_00);
7: while Q ≠ ∅ do
8:    page p ← Q.removeFirst()
9:    split p and create its children pages;
10:   for all u ∈ p.points do
11:      p.points.remove(u);
12:      find the child page pchild where u locates;
13:      pchild.points.insert(u)
14:   end for
15:   for all child page pchild do
16:      pchild.weight ← pchild.points.size
17:      compute pchild.processor according the mapping scheme
18:      if pchild.weight > threshold then
19:         Q.insert(children pages)
20:      end if
21:   end for
22: end while
```

In Algorithm 2.1, Q is a queue used to establish the recursive decomposition of the pages. Q is holding a list of pages that need to split. The time complexity of this algorithm is linear time O(n) ([34]), where n is the number of TIN triangles.

2.6 Termination determination in PSP

In the sequential Dijkstra algorithm, the one-to-one query ends when the destination point is extracted from the heap. The parallel implementation of the multi-heap Dijkstra's algorithm has different termination criteria. There are two problems in PSP termination determination:

First, in the multi-heap Dijkstra's implementation, each worker processor has its own local heap, the destination point extracted from one of the local heaps cannot guarantee that the shortest path computation is finished, because the heap in the other processor might contain a node whose distance to the source is shorter, and this node might eventually indicate a shorter path to the destination. In such a case, the processor, which extracts the destination node, has to wait until the termination criteria are met.

Therefore, the extraction of the destination from the heap indicates that we have computed a path from the source to the destination, but this path might not be the shortest. One advantage however is that, we now have an upper bound of the cost of the shortest path. Based on the fact that the cost of every node on the shortest path must be less than the length of the shortest path, nodes with costs greater than the computed upper bound are definitely not our concern. We use a MAXCOST message to limit the computation by removing those nodes from the heap. It works by telling

every processor: "I know the shortest path is shorter than MAXCOST, please take out the nodes whose costs are greater than MAXCOST from the heap and from now on, do not put into the heap any node with cost greater than MAXCOST". The following describes how we use MAXCOST:

- In each processor, we assign a variable named MAXCOST with initial value: INFINITE_LARGE.
- When a processor extracts destination from its heap, it updates its local MAXCOST with the cost of the destination (i.e. the length of the computed path), and sends a message to all the other processors to let them update their local MAXCOST variable. This message is called the MAXCOST message, and it contains the latest MAXCOST value.
- During the computation, nodes with cost greater or equal to MAXCOST are discarded: they are removed from the heap and not allowed to enter again. The computation would then be stopped while the extracted node has cost greater than MAXCOST.

Second, in an asynchronous message delivery system as we are using, there are *on-the-way* messages which might be problematic in the status determination. For example, a parallel shortest path algorithm is running on two processors: p_a and p_b. The heap in p_b is empty and its status is DONE. p_a extracts the last non-target node u from its heap (the next extracting node will be the target node). The node u needs to relax the nodes in p_b, therefore, p_a sends an Update-message to p_b. Remember the message is sent asynchronously, which means, p_a can extract the next node in its heap (the target), when the message is still on the way to p_b. If p_a has finished the operation

on the target node before the p_b receives this message, the heaps in p_a and p_b are now both empty, and their status both marked as DONE. It is apparent that the query is incomplete. Therefore, *each heap being empty*, cannot guarantee that the query is complete.

To address these problems, a 2-Round DONE token termination detection was introduced by Nussbaum ([34]) and Lanthier ([28]). It can be used in any distributed/parallel applications to safely determine the global completion of the job. Here we only describe it briefly.

- First, we logically construct a ring topology using all the processors.
- A DONE token (one for each query), which records the origin processor and the number of passing rounds (counter), is passed through this ring.
- The DONE token is only passed (without any change) from one processor p_i to the next processor in the ring p_j when p_i has done the local job and its status is DONE; otherwise, p_i will hold the token until it has done the local job.
- The token is reset (both the origin processor and the number of rounds), when its holder processor is working or the status of its holder has been changed since the last time the processor held the token.
- When a processor receives the DONE token, whose origin processor is itself and its current status is DONE, increases the counter by 1 and passes it again only if the counter is less than 2. If the counter is 2, we can say that the query is completed.

We apply this 2-round DONE token termination detection in our parallel shortest path computation implementation. We define the processor status as WORKING if its heap is not empty, otherwise the processor status is DONE. A processor with a DONE token will hold the token until the processor has finished its computation (its local heap becomes empty), then it modifies its status to DONE, passes it to the next processor in the ring topology. After the token is sent, if the processor receives other update-messages to re-start the computation, the DONE token is reset the next time it is received by the processor. That is why we choose 2 as the number of rounds. The first round is to check whether other processors are DONE; the second round is to prevent the status changes by the "on-the-way" messages.

There is a precondition for this mechanism: in the receiver processor, the messages from one processor must be handled in the same order as they are sent out. For example, processor p_a sends an update-message m_u and a DONE token message m_d to processor p_b in this particular order. Then p_b must handle m_d after it has handled m_u. Otherwise, the DONE token is passed faster than designed, and will eventually lead to an early termination before the computation is done. The single-threaded message delivery systems using TCP/IP (i.e. MPI) can guarantee the messages with same signature(sender, receiver and communicator) can be received in the same order as they are sent out. We only need to handle the received messages with caution according to their order, especially when we are using a message selector.

Chapter 3

Concurrent PSP using SWP

In this chapter, we first introduce the Bisector $\epsilon - \beta$-approximation, which is a variation of the Bisector ϵ-approximation scheme. We have introduced different performance factors of parallel shortest path computation in Section 2.4. In our parallel implementation, we focus on the data partitioning scheme (one of the data-related factors). We present the Steiner Weighted Partitioning in Section 3.2. In Section 3.3, we discuss the design of a concurrent parallel shortest path computation.

3.1 Bisector $\epsilon - \beta$-approximation

In Section 2.2.2.2, we have introduced the bisector ϵ-approximation shortest path algorithm. Equations 2.2 to 2.4 show how to place Steiner points on a bisector. Equation 2.5 computes the number of Steiner points on a bisector $k = \lfloor \log_\lambda \frac{|l|}{|vp_0|} \rfloor$, where $\lambda = (1 + \sqrt{\frac{\epsilon}{2}} \sin \frac{\alpha}{2})$, and α is the angle that the bisector splits. If α is a very

small acute angle ($\alpha \to 0$), then $\lambda \to 1$, which in return, makes k huge ($k \to \infty$). This implies that the number of Steiner points grows rapidly.

For example, when $\epsilon = 0.01$, $\alpha = 1°$, $\lambda = 1 + \sqrt{\frac{0.01}{2}} \sin \frac{1°}{2} = 1.000617059$. By the definition of $|l|$ and $d(v)$ in Section 2.2.2.2, $|l| \geq d(v)$. So we have,

$$\frac{|l|}{|vp_0|} = \frac{|l|}{\frac{cd(v)}{7} \cos \frac{\alpha}{2}} = \frac{7}{\epsilon \cos \frac{\alpha}{2}} \times \frac{|l|}{d(v)} \geqslant \frac{7}{\epsilon \cos \frac{\alpha}{2}} = \frac{7}{0.01 \cos \frac{1°}{2}} \approx 700.03.$$

Therefore, from Equation 2.5, we have,

$$k = \log_\lambda \frac{|l|}{|vp_0|} = \frac{\lg \frac{|l|}{|vp_0|}}{\lg \lambda} \geqslant \frac{\lg 700.03}{\lg 1.000617059} \approx 10,620.$$

When $\epsilon = 0.01$, $\alpha = 0.1°$, $\lambda = 1 + \sqrt{\frac{0.01}{2}} \sin \frac{0.1°}{2} = 1.0000617067$. Similaly, we have,

$$\frac{|l|}{|vp_0|} \geq \frac{7}{0.01 \cos \frac{0.1°}{2}} \approx 700.0003, \quad k \geq \frac{\lg 700.0003}{\lg 1.0000617067} \approx 106,168.$$

The number of Steiner points greatly affects the speed of the shortest path algorithm. Therefore, if there are small acute angles in a TIN, the running time of the algorithm is considerably longer when compared with another TIN (without small acute angles). In our implementation, we generalize Equation 2.4, by introducing a parameter $\beta = \sin \frac{\alpha^*}{2}$, where the critical angle $\alpha^* \in [0°, 180°]$, $\beta \in [0, 1]$ and letting

$$\lambda' = \left(1 + \sqrt{\frac{\epsilon}{2}} \times max\left(\sin \frac{\alpha}{2}, \beta\right)\right). \tag{3.1}$$

Because $\beta = \sin \frac{\alpha^*}{2}$, Equation 3.1 means that when we compute λ, the smaller angle α will be replaced by the pre-determined critical angle α^*. More specifically, if $\alpha^* =$

$0°$, ($\beta = 0$), then Equation 3.1 behaves exactly the same as Equation 2.4. When $\alpha^* = 180°, \beta = 1.0$, $\lambda' = 1 + \sqrt{\frac{\epsilon}{2}}$, which means λ' is irrelevant to the angle. λ' is a monotonically increasing function with respect to β ($0 \leq \beta \leq 1$). Thus, as λ' increases, the number of Steiner points which are placed on a bisector decreases.

For example, if we substitute Equation 2.4 with Equation 3.1, and use $\beta = 1$, then we obtain, $\lambda' = 1 + \sqrt{\frac{0.01}{2}} = 1.0707107$, resulting in,

when $\alpha = 1°$, $k' = \log_{\lambda'} \frac{|l|}{|vp_0|} = \frac{\lg \frac{|l|}{|vp_0|}}{\lg \lambda'} \geqslant \frac{\lg 700.03}{\lg 1.0707107} \approx 96$.

when $\alpha = 0.1°$, $k' = \log_{\lambda'} \frac{|l|}{|vp_0|} = \frac{\lg \frac{|l|}{|vp_0|}}{\lg \lambda'} \geqslant \frac{\lg 700.0003}{\lg 1.0707107} \approx 96$.

Compared with the original values: 10,620 ($\alpha = 1°$) and 106,168 ($\alpha = 0.1°$), k drops dramatically.

For better illustrating β's impact on the number of Steiner points on a single bisector, we let $\epsilon = 0.01$ and $|l| = d(v)$. Therefore, $\frac{|l|}{|vp_0|} = \frac{700}{\cos \frac{\alpha^*}{2}}$, and $\lambda' = \left(1 + \sqrt{\frac{0.01}{2}} \times max\left(\sin \frac{\alpha}{2}, \beta\right)\right)$. Using equation $k' = \log_{\lambda'} \frac{|l|}{|vp_0|}$, we can draw figures based on different α values and see how k' changes with different β values. Figure 3.1(a), (b) and (c) show the number of Steiner points on the bisector with $\alpha = 1°, 30°$ and $120°$ respectively. The figures show that, k remains the same before β reaches $\sin \frac{\alpha}{2}$, and k decreases when β increases.

More generally, if we denote $A = \frac{|l|}{|vp_0|}$, the ratio of k' over k is

$$\frac{k'}{k} = \frac{\log_{\lambda'} A}{\log_{\lambda} A} = \frac{\frac{\log A}{\log \lambda'}}{\frac{\log A}{\log \lambda}} = \frac{\log \lambda}{\log \lambda'} = \frac{\log\left(1 + \sqrt{\frac{\epsilon}{2}} \sin \frac{\alpha}{2}\right)}{\log\left(1 + \sqrt{\frac{\epsilon}{2}} \times max\left(\sin \frac{\alpha}{2}, \beta\right)\right)}.$$

When $\alpha \geq \alpha^*, \sin \frac{\alpha}{2} \geq \beta$, therefore, $k' = k$.

When $\alpha < \alpha^*$, $\sin \frac{\alpha}{2} < \beta$, therefore, $k' = k \times \frac{\log\left(1 + \sqrt{\frac{\epsilon}{2}} \sin \frac{\alpha}{2}\right)}{\log\left(1 + \sqrt{\frac{\epsilon}{2}} \times \beta\right)}$.

Because $\beta \leq 1$, $k' \geq k \times \frac{\log(1+\sqrt{\frac{\epsilon}{2}}\sin\frac{\alpha}{2})}{\log(1+\sqrt{\frac{\epsilon}{2}})}$. So we have the lower bound of k' as,

$$k' \geq k \times \frac{\log(1+\sqrt{\frac{\epsilon}{2}}\sin\frac{\alpha}{2})}{\log(1+\sqrt{\frac{\epsilon}{2}})} = \frac{\log A}{\log(1+\sqrt{\frac{\epsilon}{2}})}.$$

By this β-variation, the total number of Steiner points is reduced, however, the precision is worse: we cannot guarantee ϵ-approximation with this β. However, this parameter can largely reduce the number of Steiner points, while experimentally keeping the length of the path still very close to the length of ϵ-approximation shortest path. (See Section 4.3.2 for experimental results.)

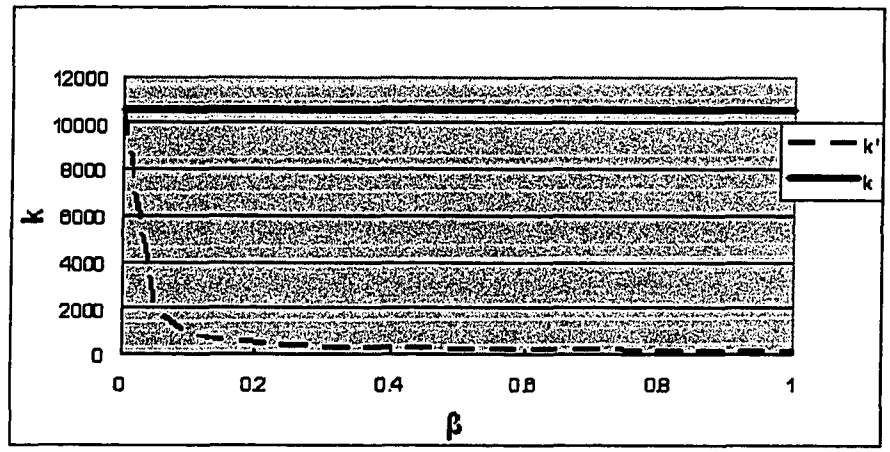
From the previous discussion, we know that the impact will only occur on the triangles whose angle is smaller than the critical angle α^*. Error would be introduced when the path crosses the affected triangles. Shorter paths have less opportunities to cross the affected triangles than longer paths. However, if short paths cross the affected triangles, the effect of the error would be greater when compared with long paths. Long paths will be less affected because the affected segments would only be a small portion of the whole path length.

If we determine the critical angle α^*, we can obtain β by $\beta = \sin\frac{\alpha^*}{2}$. So the problem of how to choose β now becomes how to choose the critical angle α^*. When we choose larger α^*, β is larger, and fewer Steiner points will be placed. This will reduce the runing-time of the approximation shortest path computation. However, the trade-off is that we are also taking more risks of having worse length accuracy. Given the specific terrain with a specific ϵ value, we can estimate the total number of Steiner points by Equation 2.6, or we can even compute the actual total number of Steiner points using Equation 2.5. Using the total number of Steiner points, we can estimate

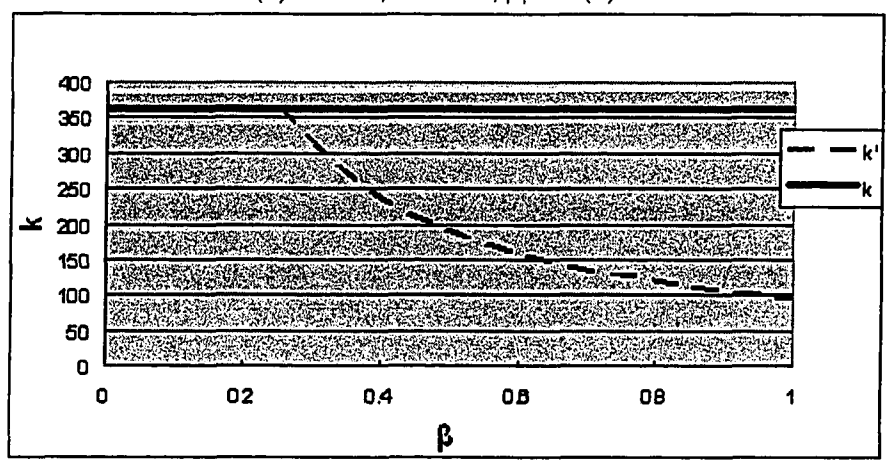
the runing-time for the approximation shortest path computation on the specific parallel machine. The bisector ϵ-approximation scheme may introduce huge number of Steiner points, typically, which may result in a very long execution of the PSP. The β parameter allows the user to determine whether to gain speed by increasing the risk of obtaining less accurate approximation of the shortest path. From our experiments (see Section 4.3.2 for details), the loss in the path length accuracy is within a small factor (smaller than 0.3%) of the length of the ϵ-approximated SP.

Unfortunately, there is no exact formula for choosing α^*. The type of shortest path queries, the type of terrain angles, and the angles distribution are three major factors one must consider when choosing α^*. Because the percentage accuracy loss for long queries is less than that for short queries, when the queries are long, one can choose a larger α^*, otherwise one may have to choose a smaller α^*. From the previous examples in this section, we know that a small angle is the main cause for producing a huge number of Steiner point. For example, given $\epsilon = 0.01$ and $|l| = d(v)$, the number of Steiner points on a bisector is 1,067 for $\alpha = 10°$, 10,620 for $\alpha = 1°$ and 106,168 for $\alpha = 0.1°$. Therefore, if the angles on the terrain are all proportional to each other, probably one maynot gain much from the β scheme. But if there are a few very small angles, by choosing α^* one can dramatically reduce the number of Steiner points on the bisectors which split those small angles. The distribution of small angles on the terrain is also an important factor for choosing α^*. If the small angles are clustered, then the probabilities of obtaining a path with a larger error in comparison to the approximated shortest path (using the bisector ϵ scheme) is higher. But if the small angles are uniformly distributed, the loss of length accuracy when applying β is relatively smaller.

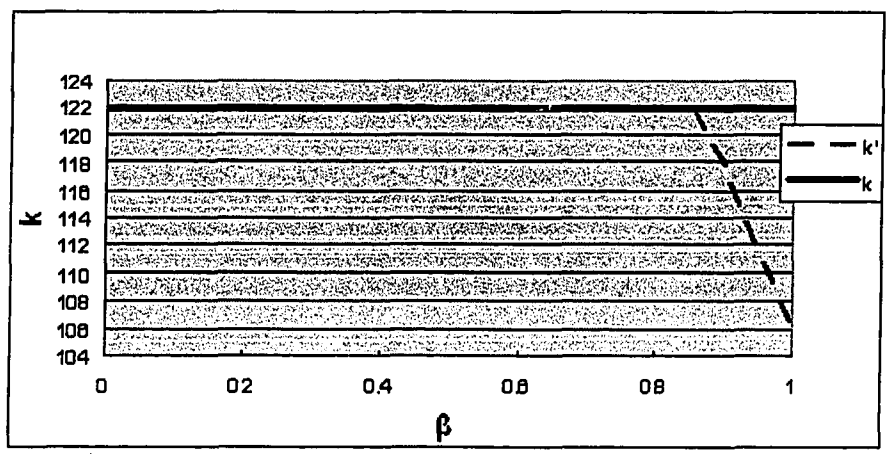
Although we cannot offer precise solution for choosing α^*, we can build a tool to help the decision. We can first sort the terrain angles from minimum to maximum. For each angle, we compute the number of Steiner points on the bisector that splits the angle. Then by choosing the α^* and recomputing the number of Steiner points of the angles that are smaller than α^*, one can observe the reduction in the total number of Steiner points (because only the angles smaller than critical angle will be affected). One can estimate the runing-time using the total number of Steiner points. The tool can also provide the percentage of angles which are affected. At last, one can choose α^* which satisfies both the acceptable running-time and the acceptable risk of worse length accuracy.



(a) $\alpha = 1°, \epsilon = 0.01, |l| = d(v)$



(a) $\alpha = 30°, \epsilon = 0.01, |l| = d(v)$



(c) $\alpha = 120°, \epsilon = 0.01, |l| = d(v)$

Figure 3.1: Impact of β on number of Steiner points: k

3.2 Steiner Weighted Partitioning

In this section, we will present the Steiner Weighted Partitioning (SWP) scheme for parallel shortest path computations. SWP is an extension of the Multi-dimensional Fixed Partitioning (MFP) scheme that we introduced in Section 2.5.2. In the following, we start with an overview of SWP, followed by the introduction of the SWP weighting scheme, the SWP algorithm, and the comparison with MFP.

3.2.1 SWP overview

Partitioning is an important performance factor of PSP computation. It can impact the computational load as well as the communication load. The computational load of a processor largely depends on the number of nodes inside the partition that the processor controls. The term *nodes* mentioned here after include both the TIN vertices and the Steiner points.

The MFP implementations define the weight of a partition as the number of TIN vertices inside the partition. When the number of the Steiner points inserted is linear proportional to the number of TIN vertices such as in FIXED Steiner points placement scheme, the defined weight of a partition reflects the number of nodes inside the partition. However, when using the bisector ϵ-approximation scheme, the Steiner points are inserted unevenly. Hence, the number of TIN vertices does not accurately reflect the number of nodes inside a partition. For example, there might be a page with 10 TIN vertices having 100 Steiner points inserted and another page with 10 TIN vertices having 100,000 Steiner points inserted. These two pages have the same number of vertices (10), but have a huge difference in their computational

load.

To address this problem, SWP defines the weight of a partition as the number of nodes inside the partition, which more accurately represents the computational load in the partition. Our hypothesis is that partitioning with respect to the number of nodes can make the nodes more evenly distributed into partitions.

The SWP weighting scheme and SWP algorithm will be separately discussed in the following sections.

3.2.2 SWP Weighting scheme

As we have introduced previously, the weight of a page is defined to be the number of nodes inside the page. The SWP weighting scheme is tailored for the bisector ϵ-approximation PSP. It shows an efficient way to obtain the weight of a page in $O(m)$ time where m is the number of TIN vertices inside the page.

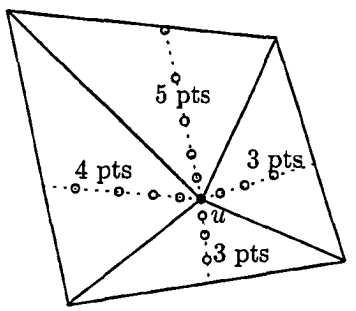
We say a node is *physically inside* a page if the geographic coordinate of the node falls into the geographic range of the page. We need to handle the special cases in which the vertex is on the boundary segments of the pages, to make sure that each vertex is included in only one page. For example, if the vertex lies on a page's boundary of the south side or west side, the vertex is included in only this page. If we associate a node with a unique TIN vertex, the node is said to be *logically inside* a page when the associated TIN vertex is *physically inside* the page.

Counting the number of Steiner points *physically inside* a page requires a heavy-load computation on the location (coordinates) of each Steiner point. In order to efficiently calculate the number of nodes inside a page, we count the number of the nodes *logically inside* the page instead of the nodes *physically inside* the page. By

defining the *master vertex of a node*, we build the logic association between a node and a unique TIN vertex as follows:

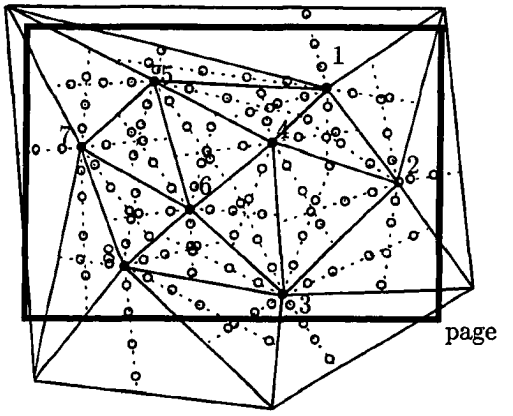
Master vertex of a node: The master vertex (or master for short) of a TIN vertex is itself; the master of a Steiner point is the TIN vertex from which its incident bisector starts. (see Section 2.2.2.2 for the placement of the Steiner points in the bisector ϵ-approximation scheme)

Now we define the *weight of a TIN vertex* u as the number of nodes whose master vertex is this TIN vertex. We denote it as W_u. Figure 3.2(a) shows an example of vertex u and its associated weight W_u. The points on the bisectors are Steiner points. TIN vertex u is the master of those points.



(a) Weight of vertex u

$W_u = 1 + 3 + 3 + 5 + 4 = 16$



(b) Weight of a page

$W_{page} = \sum_{i=1}^{8} W_i$
$= 15 + 16 + 22 + 13 + 15 + 15 + 16 + 18$
$= 130$

Figure 3.2: Examples of the weight of a TIN vertex u and the weight of a page

Finally, we define *weight of a page* (denoted by W_{page}), as the sum of the weight of

TIN vertices that are *physically inside* this page as follows:

$$W_{page} = \sum_{v \in page} W_v,$$

where v is any TIN vertex that is *physically inside* this page. An example is shown in Figure 3.2(b). The thick frame is the MBR of the page. Dotted segments are the bisectors.

3.2.3 SWP Algorithm

Algorithm 3.1 shows the SWP algorithm, which is quite similar to the MFP algorithm (Algorithm 2.1). They differ from each other only in the way of counting the weight and the criteria of over-partitioning termination.

Lines 5-9 in Algorithm 3.1 replace lines 4-5 in Algorithm 2.1 by adding the computation for the weight of every TIN vertex. By Equation 2.5, we can compute the number of Steiner points on a bisector in $O(1)$ time. There are $3n$ bisectors in the TIN where n is the number of triangles, so the computation for the weight of every TIN vertex can be done in $3n \times O(1) = O(n)$ time.

Lines 22-27 determine whether a page should be further split. A page with only one vertex inside should not be further split. If a page contains only one vertex that has a weight greater than the re-partition threshold, no matter how we re-partition, this vertex still remains in one page with its weight still greater than the threshold.

The other parts for the partitioning including the page creation, and page-processor mapping are all the same as MFP. It is easy to see that the SWP algorithm has linear runtime with respect to the number of triangles.

Algorithm 3.1 SWP a TIN

```
1: Q ← ∅
2: create the root of MFP tree ( page_00 );
3: page_00.processor ← p_00;
4: page_00.weight ← 0;
5: for all v ∈ V do
6:    Compute v.weight;
7:    page_00.weight ← page_00.weight + v.weight;
8:    page_00.points.insert(v)
9: end for
10: Q.insert(page_00);
11: while Q ≠ ∅ do
12:    page p ← Q.removeFirst()
13:    split p and create its children pages;
14:    for all u ∈ p.points do
15:       p.points.remove(u);
16:       find the child page pchild where u locates;
17:       pchild.points.insert(u)
18:       pchild.weight ← pchild.weight + u.weight
19:    end for
20:    for all child page pchild do
21:       compute pchild.processor according the mapping scheme
22:       if pchild.points.size ⩽ 1 then
23:          continue;
24:       end if
25:       if pchild.weight > threshold then
26:          Q.insert(children pages)
27:       end if
28:    end for
29: end while
```

3.2.4 Advantages and disadvantages

In this sub-section, we will discuss the advantages and disadvantages of SWP over MFP.

The only difference between SWP and MFP is on their weighting scheme to repartition a page. MFP actually uses the number of TIN vertices in a page as the weight of the page. Instead, SWP determines the weight of a page to be the number of nodes in the page. As we discussed earlier in Section 3.2.1, when the number of the inserted Steiner points is linear proportional to the number of TIN vertices (e.g. using FIXED scheme), SWP will produce exactly the same partitions as MFP. However, when we place Steiner points using bisector ϵ-approximation scheme, the geometry features of the TIN heavily affect the number of the Steiner points inserted. These geometry features include angles of the triangles and the distance from a vertex to its surrounding polygons. In this case, because the number of vertices might not be linear proportional to the number of nodes, it cannot accurately reflects the computational load in the partition. MFP might produce pages which contains just a few vertices but many Steiner points. SWP solves this problem because it will repartition a page whose weight (number of nodes) is more than the threshold, no matter how many vertices are inside that page. In extreme cases when only one vertex has huge weight, SWP will eventually isolate this vertex in a single page, and distribute the surrounding vertices into other pages. Therefore, by restraining the number of nodes in a page, SWP distribute the nodes more evenly among the partitions.

The SWP algorithm is more complicated when compared with MFP algorithm, because it has the overhead of computing the weight of each vertex. Also, to compute the vertex weight, we need to know the parameter(s) for the shortest path algorithm,

e.g., the parameter ϵ for the bisector ϵ-approximation algorithm. For different parameter setting, we have to run SWP separately. On the contrary, MFP partitioning does not need to know the parameters in advance. MFP can offer a once-and-for-all partitioning solution, which saves the runtime of pre-computation for the partitioning in parallel shortest path computation. However, we should notice that, the runtime for the SWP partitioning is very small when compared with the runtime for shortest path queries using the bisector ϵ-approximation algorithm. The benefit from SWP in PSP might be greater than the saving of the runtime for the partitioning.

3.3 Concurrent PSP implementation

In Section 3.3.1, we first introduce the concept of concurrent parallel shortest path computation, followed by the design for the concurrent PSP implementation in Section 3.3.2. In Section 3.3.3, we discuss a model for node information exchange among the processors.

3.3.1 Concurrent PSP

3.3.1.1 Motivation

As we have mentioned earlier, it is difficult to efficiently parallelize an algorithm that is sequential in nature. Unfortunately, the shortest path problem is sequential in nature. Let $\pi(s,u)$ denote the shortest path from source vertex s to any vertex u, and $||\pi(s,u)||$ denote the length of the path $\pi(s,u)$. Using Dijkstra's algorithm, a shortest path $\pi(s,u)$ can only be computed after the completion of computation of

shortest path $\pi(s,v)$, where v is any vertex that satisfies $||\pi(s,v)|| < ||\pi(s,u)||$.

The effect of this sequential nature on the performance of PSP is that it results in a large amount of processor idle time, during which a processor is doing nothing but waiting for the results from another processor. Although by choosing a good partitioning scheme, we can let the processors get involved quickly to reduce the idle time of the processor, the idle time is still noticeable, especially when the number of processors increases. From Lanthier et al.'s ([28]) experimental results, we can clearly see that the amount of idle time increases when the number of processors grows. They roughly obtained a speedup of $\sqrt{p}$ for PSP computation, where p is the number of processors. Hence the efficiency, which is roughly $\frac{1}{\sqrt{p}}$, degrades quickly when p grows.

If a shortest path query q_i is accepted by the application when the application is still processing other queries, we call these queries *concurrent queries.* A parallel shortest path application usually deals with concurrent queries in batch mode (i.e., one query at a time). Before the running query is finished, other arriving queries have to be put into a waiting queue.

The concurrent queries are waiting, and some processors are sitting idle while other processors are busy. Is it possible to utilize those idle times on other concurrent queries, so that we can get higher speedup and efficiency? This brings us the idea of concurrent PSP. *Concurrent PSP* is a parallel shortest path computation that can process concurrent queries simultaneously to obtain higher speedup and efficiency.

Another issue we should consider is the priority of the queries. Normally, we apply a "first-come, first-serve" strategy so that the earlier query has the higher priority in the computation. We cannot simply use the multi-threading or the multi-processing to duplicate the instance of the single query solution, because neither multi-threading

nor the multi-processing can guarantee the execution priority of the query.

3.3.1.2 Solution

Here is the basic idea for the solution:

- Allow processors to hold several queries at the same time.
- When the processor has locally computed a query and its previous queries, the processor can start the computation of the next query right away. The processor becomes idle only if all the queries it holds have been locally computed.

The major tradeoff in the implementation of this multi-tasking solution is between computation speed and memory utilization. The more queries a processor holds at the same time, the more memory is required. Practically, due to the memory allocation problem, we have to limit the number of queries that a processor can process at the same time (named **concurrency threshold**), which means the application cannot process an un-limited number of queries concurrently. A trivial estimation for the upper bound of the concurrency threshold is,

$$concurrency\ threshold \leqslant \frac{MAPP}{MNPQ}.$$

In this equation, $MAPP$ is the size of memory that can be allocated for this application on each processor; $MNPQ$ is the size of memory that is needed for a shortest-path query on each processor (e.g., memory space for nodes and heaps).
In implementations of the sequential Dijkstra's algorithm, shortest-path heap is often used for the shortest path computation. The top element of the shortest-path heap

represents the vertex that is closest to the source vertex than any other vertex in the heap. In our implementation, each query is wrapped by a task in each processor. Each query has a priority that is usually based on the query id (i.e., sequence number of the query). We say a query has higher priority if its query id is smaller. Each task has a shortest-path heap which is only used for the wrapped query. We say a task is a *working task* if and only if its shortest-path heap is not empty, otherwise the task is an *idle task*.

Every worker processor has a heap (called *task-heap*) of tasks. The task-heap is applied to manage priority of the tasks (queries). A task is inserted into the task-heap after it has been initiated and removed after it has finished the query. The task's position in the heap is adjusted when the task's status is changed (e.g., between working and idle).

The goal for the design of the task-heap is to obtain, in constant time, the working task that has the highest priority among all the working tasks. To build such a task-heap, we must define a way to compare the priority of two tasks. The comparing function is shown in Function 3.2:

We can see that, the task priority is a combination of the task's status and the query's priority. The top element of task-heap is the task with the highest priority. It can be accessed in $O(1)$ time. We always only process the task with the highest priority. After the task with the highest priority is locally done (its shortest-path heap is empty), its priority will drop because it is now an idle task. The task-heap is then adjusted by bringing up the task with current highest priority to the top of the task-heap. While an idle task with higher query priority turns to working again after it receives any node-update message, the task-heap is adjusted again to bring this task up because it

Function 3.2 ComparePriority(task1, task2)

```
1: rvalue ← UNDEFINED
2: if task1.status = task2.status then
3:   if task1.query.id < task2.query.id then
4:     rvalue ← Task1IsHigher
5:   else if task1.query.id > task2.query.id then
6:     rvalue ← Task2IsHigher
7:   else
8:     rvalue ← IsSame
9:   end if
10: else
11:   if task1.status = WORKING then
12:     rvalue ← Task1IsHigher
13:   else
14:     rvalue ← Task2IsHigher
15:   end if
16: end if
17: return rvalue
```

now has higher task priority. The processor will become idle only when all the tasks are idle. Therefore, with concurrent PSP, the possibility of a processor becoming idle is reduced, which is exactly what we want to achieve.

3.3.2 Application design

A basic model for the PSP computation can be illustrated by the following steps [28]:

- Partition the original graph into different partitions, for example, one partition for each processor
- Locally run the Dijkstra algorithm in each processor

- Processors communicate with each other to update the cost and the path information at the partition boundary
- When the shortest path is found, report the length and the path.

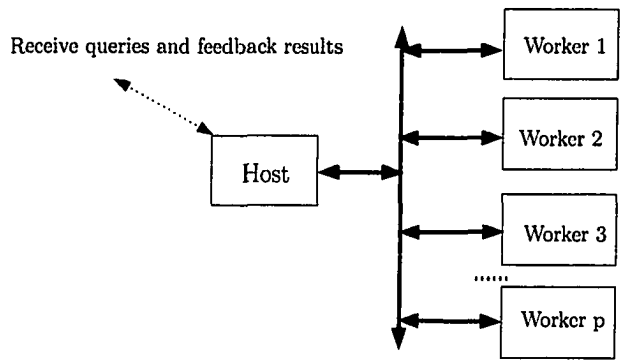


Figure 3.3: System architecture: Host/Workers

Naturally, we choose the Host/Workers structure as shown in Figure 3.3. A host (named pspHost) is a process that acts as the coordinator between the "inside" and "outside" of the PSP system. It receives the query requests from "outside", dispatches them to the "inside" workers, and feeds back the results after the query is done. Each worker (named pspWorker) is running on one processor. Workers receive the queries from the host, communicate with each other to compute the shortest path and then send back the results to the host.

The brief control-flow for the pspWorker is illustrated in Figure 3.4. Roughly, this is a combination of Dijkstra's algorithm (see right side of Figure 3.4) and the message handling process (see left side of Figure 3.4). During the Dijkstra's iterations, if there is a message coming in, the message is handled between two iterations (i.e., a single extract_min/relax iteration cannot be interrupted by the message).

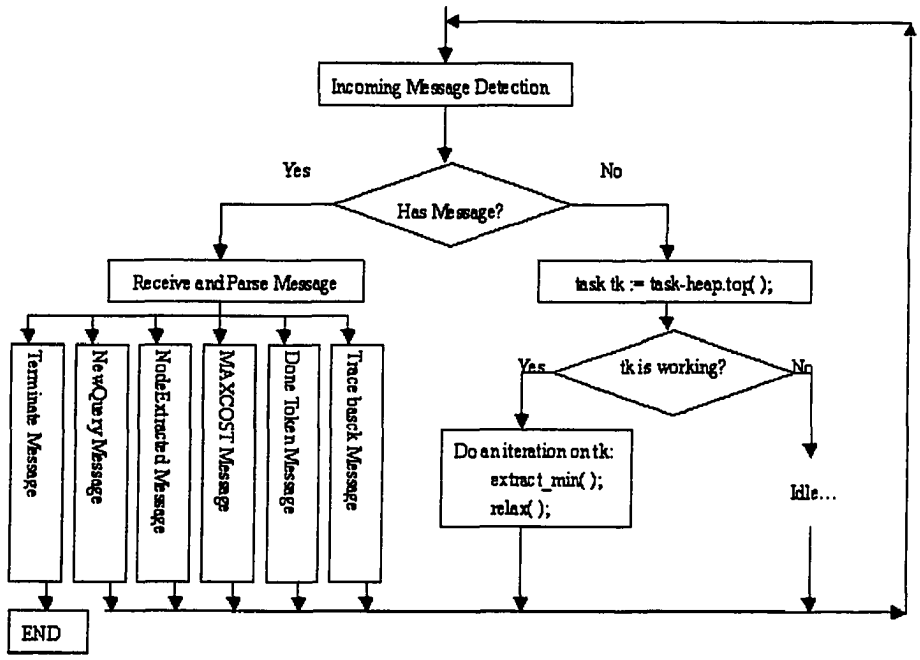
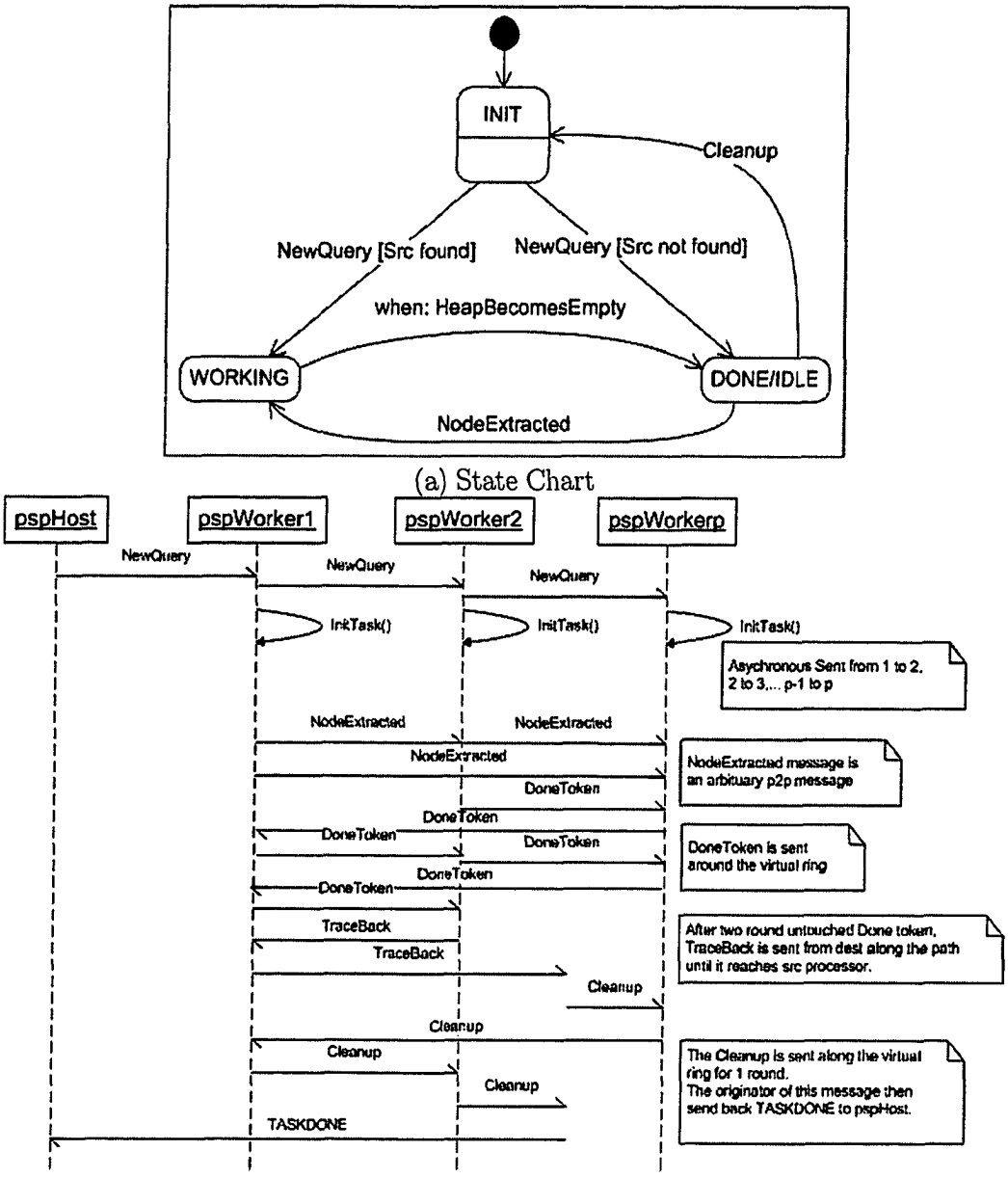


Figure 3.4: pspWorker control flow diagram

Figure 3.5(a) shows the State Chart for a task. There are three basic states for each task: INIT, WORKING and DONE. When a task receives the NewQuery message, it searches for the source and destination point in its local partition. If the source is found, it puts the source into the shortest-path heap and turns into WORKING state; otherwise, it turns into DONE state. When the shortest-path heap is empty, we say the shortest path computation is locally done, and the task turns into DONE state. When the task is in DONE state and the task receives a NodeExtracted message that notifies it that a node on the boundary is extracted from other processor, the node is then put into the shortest-path heap and the task is put back into a WORKING state (see Section 3.3.3 for details).

Figure 3.5(b) shows the Message Sequence Chart. It briefly illustrates the messages corresponding among the processors. When pspHost receives a query from the client, it sends a NEWQUERY message to the first processor. Then this message is passed from one processor to another until it arrives in the last processor. If the shortest path is computed across the partitions, the NodeExtracted message is sent among the relevant processors (details in Section 3.3.3). The DoneToken messages are used to determine the termination of the shortest path computation, as described in in Section 2.6. When a DoneToken has implied the finishing of computation, the shortest path trace-back is started from the destination point to the source point so that the segments of the shortest path are delivered back to the user. After the trace-back work is done, a Cleanup message is generated and passed through all the pspWorker processors to do the cleanup work. The last pspWorker processor then notifies the pspHost that this task is done and ready for the next query.



(a) State Chart

(b) Message Sequence Chart

Figure 3.5: pspWorker State Chart and Message Sequence Chart

3.3.3 Boundary Notification Model

In the parallel shortest path computation, when the computation reaches the partition boundary, the node-updates (i.e., the cost and parent information) should be passed from one processor to the other relevant processors, so that the computation can continue seamlessly. In this section, we describe the BNM (Boundary Notification Model) to accomplish this update task. It is suitable for the scenario, in which one node can relax many other nodes, especially when applying the bisector ϵ-approximation scheme ([7]).

3.3.3.1 Definitions and basic strategy

We start this section with some definitions.

- **Master processor of a node:** As we defined earlier, a node could be either a TIN vertex or a Steiner point. The master vertex of a node is a unique TIN vertex that is logically associated with a node with respect to the SWP weighting scheme (see Section 3.2.2). After the SWP partitioning, each TIN vertex is "physically inside" a partition, which is mapped to a single processor. In this way, each TIN vertex is associated with a unique processor that is called the master processor of a vertex. Further, the master processor of a node is the master processor of the node's master vertex.

- **Boundary:** A partitioning scheme will provide each processor with a partition, which is composed of part(s) of the original TIN. The partition associated with processor i is denoted by G_i^*. The intersection of G_i^* and G_j^* is called the

boundary between partition i and partition j, denoted by $B_{i,j}$ ($0 \leqslant i, j \leqslant p$; $i \neq j$; p is the number of processors).

- **Boundary node:** A node is a boundary node, if and only if it belongs to any of the boundary $B_{i,j}$ ($0 \leqslant i, j \leqslant p$; $i \neq j$; p is the number of processors).

- **Local/remote node:** From the definition of master processor of a node, we know a node has a unique master processor. A node in partition i is called **local node** of partition i, if its master processor is processor i. In contrast, a node in partition i is called **remote node** of partition i, if its master processor is not processor i. A boundary node would be a local node in one partition, and be a remote node in the other partitions. For example, node u belongs to a boundary $B_{i,j}$, if u's master processor is i, then in partition i, u is a local node; but in partition j, u is a remote node.

- **NodeExtracted message:** It is a message that notifies other processors about the extracted node, when a local boundary node is extracted from its local shortest-path heap. This message is only generated when the computation reaches the boundary. Through it, the shortest computation is forwarded to other processors.

In this model, we first introduce a rule for the relaxation:

Relaxation rule: Processor i only relaxes the nodes whose master processor is i.

When this rule is applied in the boundary, the computation related to the relaxation of the boundary nodes is distributed to their master processors. In Section 3.3.3.2,

we discuss this rule further and show the ***Relax(u)* procedure**, which relaxes the adjacent nodes of node u and generates the node-update messages.

There are two things that we need to consider when we design the node-update method:

1. We want to reduce the re-computation since it reduces the efficiency. The node-updates along the boundary may cause a rippling of node-updates throughout the partition, thereby causing a re-computation of previously computed costs to many non-border nodes of the partition as well ([28]). In the sequential Dijkstra's algorithm, a node is extracted only once from the heap. But in the parallel multi-heap Dijkstra's implementation, a node could be re-inserted into the heap and re-extracted from the heap many times because of the node-updates.

2. We should consider the volume of the communication caused by the node-update messages. Lanthier et al. used a "Fixed Steiner points" scheme in their parallel shortest path implementations ([28]). Their node-update messages not only contained the information of the extracted node, but also contained the information of the nodes relaxed by the extracted node. The advantage of this mode is that, on the message receiver side, we can directly update the relaxed nodes that have already computed on the sender side. Unfortunately, unlike the "Fixed Steiner points" scheme, the bisector-ϵ approximation scheme will introduce thousands of, or in some extreme cases, millions of Steiner points in one single relax operation. This makes the implementation for the previous mode impractical: the node-update message becomes too large to be created or sent. For example, in the lam-MPI implementation, the sending buffer has a limited

size, not to mention that, the consecutive buffer allocation for such a huge size will have problem. Therefore in our implementation, the node-update message (also named as NodeExtracted message) only contains the information of the extracted node itself. The size of the message is then small and fixed, which avoids the problem of message size limit as discussed previously.

Later in Section 3.3.3.3, we discuss how to handle the received NodeExtracted message. After that, we make a conclusion for the BNM model.

3.3.3.2 *Relax(u)* procedure

As discussed in Section 2.2.2.2, we say two bisectors are neighbors if the angles they split share a common edge of the TIN. Edges are added between nodes on two neighbor bisectors (see [7, 8] for details). In a TIN, the bisector has at most 7 neighbors including itself. Using the SWP weighting scheme, the master vertex of all the nodes on a bisector t is the TIN vertex (denoted by a) from which t starts. Therefore, the nodes on the bisector t have the same master processor: the master processor of a. We now use an example for better understanding of the relaxation rule. As illustrated in Figure 3.6, the boundary node u is an extracted node on bisector t_0. t_0 has 7 neighbor bisectors: $t_0..t_6$. In the sequential implementation, the relaxation will include nodes on all the 7 neighbor bisectors $t_0..t_6$. In a parallel implementation of this example, we assume processor 1 is the master processor of nodes: P_1 and P_2, and processor 2 is the master processor of node P_3. Then while processor 1 extracts u from its local heap, it will only relax nodes on bisectors t_0, t_1, t_3, t_4 and t_5 because the master processor of the nodes on these bisectors is processor 1. While processor 2

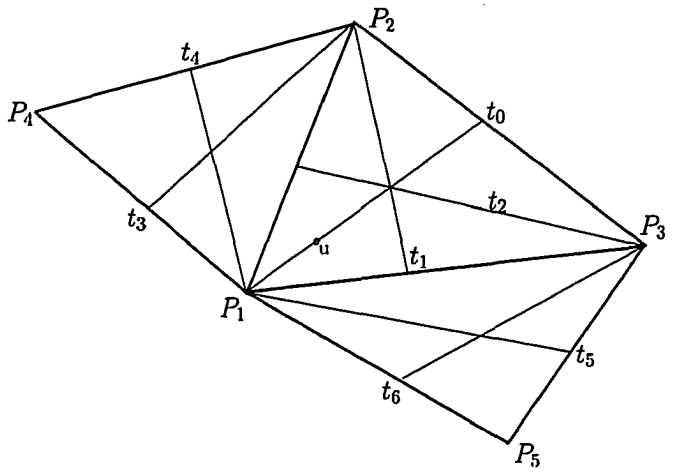


Figure 3.6: An example of boundary node u is extracted

extracts u from its local heap, it will only relax nodes on bisectors t_2 and t_6 because the master processor of the nodes on these bisectors is processor 2. By this rule, we distribute the relaxation computation to different processors.

The pseudo-code of the procedure is shown in Procedure 3.3.

When a node in the boundary is extracted from the heap, its following "relax" operations are distributed to related processors with respect to the relaxation rule: *Processor i only relaxes the nodes whose master processor is i.* This accords with the SWP weighting scheme, in which the weight of a partition reflects the computational load on the mapped processor.

3.3.3.3 *HandleNodeExtracted(msg)* procedure

Procedure 3.4 is invoked when a NodeExtracted message (denoted by msg), is received from other processor. In Section 3.3.3.1, we have introduced that msg contains the

Procedure 3.3 Relax(u)

```
1: t_0 ← u's incident bisector
2: S ← ∅ {S is a set of processors}
3: for each neighbor bisector (denoted by t_i) of t_0 do
4:    x ← t_i's start vertex
5:    p_x ← x's master processor
6:    if p_x = this processor then
7:       for each node v on t_i do
8:          if v.cost > u.cost + cost(u, v) then
9:             v.cost ← u.cost + cost(u, v)
10:            v.parent ← u;
11:            update v's position in heap
12:         end if
13:      end for
14:   else
15:      if u is local node then
16:         Add p_x into S
17:      end if
18:   end if
19: end for
20: if u is local node then
21:   Prepare NodeExtracted message (msg), msg.node ← u
22:   for each processor pr ∈ S do
23:      Send msg to pr
24:   end for
25: end if
```

information of the extracted node (denoted by *msg.node*). From the definition of local node and remote node, we know that, *msg.node* represents a remote node (also denoted by u) on the receiver side, because it is a local node on the sender side.

Procedure 3.4 HandleNodeExtracted(msg)

```
1: Find the node u represented by msg.node
2: if u.cost > msg.node.cost then
3:   u.cost ← msg.node.cost
4:   if u is not in local heap then
5:     insert u into heap
6:   else
7:     update u's position in the heap
8:   end if
9: end if
```

The receiver processor does not perform the "relax" operations at the time it receives the node-update message. Instead, it puts this node into its local heap and waits for its retrieval from the queue. This "late-update" strategy has the following benefits:

- First, the processor can receive and process the node-update messages quickly by postponing the "heavy" computation for relaxation. Thus in such a single-threaded message-driven system, the message throughput is increased and the communication is more efficient.

- It reduces the probability of useless re-computation. In a parallel shortest path computation, the cost of the extracted node is just locally minimum (minimum on the sender side).

  The receiver processor might still compute on another node that has smaller cost. Therefore, if we do the relaxation operations as soon as we receive the node-update event, we have to do the same relaxation operations again, once

this node is relaxed by the node(s) in the current local heap. Then the previous relaxation operations are completely wasted. To get better runtime performance, we should avoid such useless re-computation.

On the contrary, if the receiver processor is idle or it is processing nodes with larger cost, while it receives the node-update message, because we insert the extracted node (denoted by u) into the local heap, from the control flow diagram (Figure 3.4) we can see that, after the processor has handled the received messages, it will extract u from the heap before other nodes because its cost is smallest. Then the relaxation of u is done cleanly.

3.3.3.4 Conclusion

The BNM modal we have presented is suitable for the parallel shortest path computation with ϵ-approximation scheme. The relaxation rule can help us distribute the computation work on relaxation of boundary nodes into different processors respectively. The "late-update" strategy discussed in Section 3.3.3.3 is good for reducing re-computation.

Chapter 4

Experimental Results and Analysis

4.1 Environment

We have implemented and run this PSP application on both a SunFire 6800 Cluster and a Pentium Xeon-based Beowulf Cluster. The machine configurations are:

SunFire 6800 Cluster

- 1 node with 20 - 900 MHz UltraSPARC III Cu Processors
- 20 GB of memory per 20 processor node
- 1.3 TB disk storage on Sun StorEdge[tm] T3 Disk array
- Sun Grid Engine v5.3 Enterprise Edition with Sun MPI v5.0

Beowulf Cluster

- Total of 128 processors: 32 nodes with dual 1.7 GHz Xeon Processors with

1 GB RAM per node, 32 nodes with dual 2.0 GHz Xeon Processors with 1.5 GB RAM per node

- 60 GB of disk storage per node
- The 1.7 GHz nodes use Intel Pro 1000 XT NICs
- The 2.0 GHz nodes use on-board GigE interfaces.
- All nodes are interconnected via a Cisco 6509 switch using Gigabit ethernet
- 8 nodes are equipped with high end Graphics cards
- Redhat 7.3, Sun Grid Engine Enterprise Edition Scheduler with LAM-MPI v6.5.9

The processor configurations are 1x1(1 processor), 2x2(4 processors), 3x3(9 processors), 4x4(16 processors).The dataset we are using is:

- "africa5k": 5000 vertices, 9799 faces(triangles) and 14798 edges.

We choose only this terrain because the SP application using the bisector ϵ-approximation scheme is a very time consuming application. As shown in Table 4.6, even a single test on the Sunfire machine may take 1,094,903.81 seconds (12.67 days). It would be interesting to work on other terrains in the future to examine the performance stability.

First, we have done experiments with different SWP thresholds on a fixed ϵ value. We have stated that different thresholds can result in different partitions. We will analyze the time distributed with respect to computation, communication and idle times. We can observe how the performance varies with different partitions in our experiments. We will analyze the computational load of each processor by counting

the heap operations on each of them. We can see how the computation is distributed among the processors.

We also did our experiments on different ϵ values: 0.1, 0.05 and 0.01, with different parameter β values: 0, 0.5, 0.707107 and 1. In Section 3.1, we discussed that larger parameter β can reduce the number of Steiner points. We will see how the ϵ and β values affect the number of the Steiner points, and how the number of the Steiner points affects the running-time of the application. We also study how the β affects the accuracy of the shortest path.

We did experiments with different *concurrency thresholds* (the number of queries that can be executed at the same time) to see how the concurrency contributes to the speedup. We will also study the relationship between the performance and query distribution(evenly distributed or clustered).

threshold	2x2 processors		3x3 processors		4x4 processors	
	tree level	page #	tree level	page #	tree level	page #
100	6	3184	5	13889	4	26071
200	6	3184	5	13889	4	26071
500	6	3178	5	12545	4	24211
2500	6	1450	5	3281	4	3451
5000	5	757	4	681	3	2881
10000	5	310	3	593	3	676
20000	4	196	3	457	2	241
50000	3	55	2	73	2	211
100000	3	43	2	73	2	151
150000	3	22	2	65	2	46
180000	2	13	2	57	1	16

Table 4.1: SWP with different thresholds, africa5k, $\epsilon = 0.1$

4.2 Impact of SWP threshold

4.2.1 SWP partitioning results

Table 4.1 exhibits the configuration of the SWP page tree (levels, page numbers) according to the processor configurations and the threshold values. We can see when threshold increases, the page number decreases and the tree level also decreases. This is because a page can contain more nodes when the SWP threshold is larger, hence the over-partitioning with larger threshold is coarser. We also notice that sometimes, different thresholds can cause same partitioning. For example, in 3x3 processor configuration, the partitioning using threshold 50000 and 100000 are the same. The reason is when we partition with threshold 100000, each page has weight less than 50000. Therefore, any threshold between 50000 to 100000 cannot further trigger the page re-partitioning.

We also run SWP and MFP separately with different thresholds and record the number of the partition tree, the number of pages, and the standard deviation of number of nodes in partitions. The results are shown in Table 4.2. After either SWP or MFP partitioning, we can obtain p (p equals to the number of processors) partitions. For

threshold ratio	SWP				MFP			
	threshold	tree level	page #	stdev	threshold	tree level	page #	stdev
0.002	3205	4	2265	10924	10	4	2249	15909
0.004	6411	3	617	7408	20	3	593	9026
0.006	9616	3	601	9499	30	3	577	11274
0.008	12822	3	569	12521	40	3	561	12518
0.010	16027	3	529	13231	50	3	521	16123
0.012	19233	3	473	15752	60	3	465	17218
0.014	22438	3	409	17677	70	3	393	24607
0.016	25643	3	289	21162	80	3	305	26202
0.018	28849	3	201	23979	90	3	249	17194
0.020	32054	3	121	13096	100	3	145	22436
0.030	48082	2	73	16659	150	2	73	16659

Table 4.2: Comparison of SWP and MFP (3x3 processors, africa5k, $\epsilon = 0.1$)

each partition, we count the number of nodes in that partition and at last, compute the standard deviation of number of nodes in partitions (column 5 and 9 in Table 4.2). This standard deviation value represents how good the nodes are distributed among the partitions. The value is smaller if the nodes are more evenly distributed among the partitions. The SWP threshold and the MFP threshold have different meaning. To make a fair comparison, we choose their thresholds by the same threshold ratio. In SWP, the threshold ratio is equal to the threshold over the total number of nodes in the whole graph. In MFP, the threshold ratio is equal to the threshold over the total number of TIN vertices in the whole graph.

From Table 4.2, first we find that with the same threshold ratio, SWP and MFP generate the same level of partitioning, quite similar number of pages. This is because the chosen terrain (africa5k) has similar density of TIN vertices and Steiner points. Therefore, the partitioning with respect to the number of nodes (i.e. SWP) does not have large difference with the partitioning with respect to the number of TIN vertices (i.e. MFP). However, the difference is still exist. In most cases, the standard deviation of nodes in partitions is better in SWP than in MFP, which means the nodes are more evenly distributed among the processors.

4.2.2 Speedup and efficiency

To analyze the runtime performance of our parallel shortest path computation, we specify the following terms with respect to the processing time:

- Computation time: It represents the time that the processor spends on computation on its own heap, including the time for generating the result paths.
- Communication time: It represents the time spent on sending and receiving messages from other processors, as well as the time on probing the incoming messages.
- Idle time: It is the amount of time that the processor remains idle while it is performing neither computation nor communication.
- TOTAL time: It is the sum of previous three times. It starts counting after the processor has loaded the partition and is ready for the queries. It ends after the result of the last query has been output.

In Section 2.5.2, we discussed the impact of SWP threshold theoretically. For 2x2, 3x3, 4x4 processor configurations, we picked several threshold candidates and ran our application. We counted the time spent on computation, communication as well as the idle time. We drew the time distribution diagrams and the speedup diagrams. After the comparison of those diagrams, we found that they had almost the same features. Therefore, in this thesis, we only show two of them and explain their features. Table 4.3 (a) and (b) show the processing time distribution as well as the speedup and efficiency on the Sunfire machine and the Beowulf machine correspondingly. Figure 4.1(a) and (b) show the processing time distribution on the Sunfire machine and the

processor and WMFP threshold	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency
1x1_10000	0.01	0	75984.05	75984.06	1	100%
2x2_500	448.95	29.33	22802.44	23280.72	3.26	81.6%
2x2_2500	355.84	11.44	21710.29	22077.57	3.44	86.04%
2x2_5000	271.84	16.12	21500.16	21788.12	3.49	87.19%
2x2_20000	191.51	227.17	23016.13	23434.81	3.24	81.06%
2x2_50000	147.28	499.43	24213.81	24860.52	3.06	76.41%
2x2_150000	125.05	735.84	24676.73	25537.62	2.98	74.38%
2x2_180000	107.92	3590.09	25050.46	28748.47	2.64	66.08%
3x3_200	430.17	26.81	11001.12	11458.1	6.63	73.68%
3x3_2500	299.31	26.44	11253.46	11579.21	6.56	72.91%
3x3_5000	308.08	65.79	11294.69	11668.55	6.51	72.35%
3x3_10000	200.31	53.34	11241.16	11494.81	6.61	73.45%
3x3_20000	230.91	123.99	12450.08	12804.99	5.93	65.93%
3x3_50000	263.2	420.61	12223.63	12907.45	5.89	65.41%
3x3_150000	290.32	754.1	13629.33	14673.76	5.18	57.54%
4x4_100	1457.19	70.44	7546.17	9073.8	8.37	52.34%
4x4_2500	487.11	65.09	7267.32	7819.53	9.72	60.73%
4x4_5000	466.86	94.41	7262.97	7824.25	9.71	60.7%
4x4_10000	377.04	256.8	8436.38	9070.22	8.38	52.36%
4x4_20000	208.22	195.43	7564.43	7968.08	9.54	59.6%
4x4_100000	223.99	578.65	11103.03	11905.68	6.38	39.89%
4x4_180000	200.06	2893.84	7325.75	10419.65	7.29	45.58%

(a) Sunfire machine

processor and WMFP threshold	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency
1x1_10000	0.02	0	17068.44	17068.46	1	100%
2x2_500	958.3	17.21	5396.74	6372.25	2.68	66.96%
2x2_2500	727.51	6.72	5065.04	5799.27	2.94	73.58%
2x2_5000	579.67	7.82	4907.35	5494.83	3.11	77.66%
2x2_20000	472.01	69.54	5056.12	5597.67	3.05	76.23%
2x2_50000	372.38	129.92	5125.19	5627.48	3.03	75.83%
2x2_150000	306.1	101.05	5110	5517.16	3.09	77.34%
2x2_180000	264	747.74	5025.19	6036.93	2.83	70.68%
3x3_200	729.93	12.78	2693.63	3436.33	4.97	55.19%
3x3_2500	619.2	12	2664.12	3295.32	5.18	57.55%
3x3_5000	393.04	19.92	2514.63	2927.59	5.83	64.78%
3x3_10000	380.84	18.52	2520.64	2920	5.85	64.95%
3x3_20000	402.43	32.38	2763.19	3198	5.34	59.3%
3x3_50000	230.53	103.06	2586.48	2920.08	5.85	64.95%
3x3_150000	259.68	148.61	2961.67	3369.97	5.06	56.28%
4x4_100	694.28	27.6	1871.03	2592.91	6.58	41.14%
4x4_2500	472.54	21.08	1649.38	2143	7.96	49.78%
4x4_5000	471.73	29.22	1676.91	2177.85	7.84	48.98%
4x4_10000	356.04	67.56	1847.77	2271.37	7.51	46.97%
4x4_20000	218.18	53.72	1572.59	1844.49	9.25	57.84%
4x4_100000	259.02	198.53	1601.6	2059.15	8.29	51.81%
4x4_180000	131.63	652.75	1504.94	2289.32	7.46	46.6%

(b) Beowulf machine

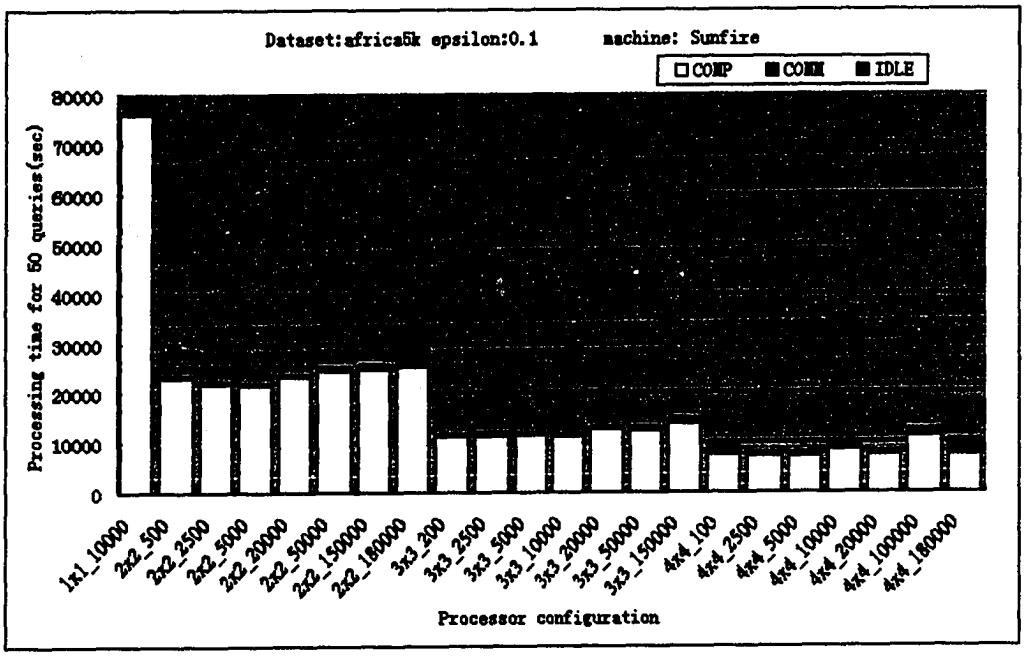
Table 4.3: Results on different SWP thresholds, africa5k, $\epsilon = 0.1$
time unit: second

Beowulf machine correspondingly. Figure 4.2(a) and (b) show the speedup diagram on the Sunfire machine and the Beowulf machine. In each diagram, the last number in the processor configuration is the SWP threshold. For example, 3x3_10000 means SWP threshold is 10000 and the application runs with 9 processors.

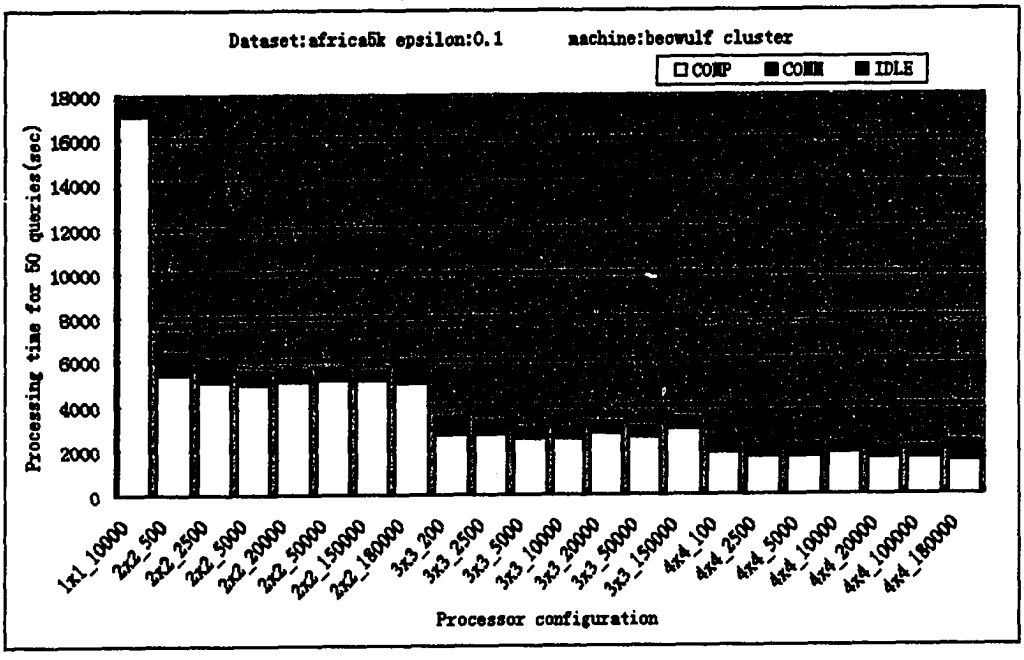
First, from Figure 4.1, we can see when the threshold is small, the communication time increases (the blue portion in the figure), because the regions assigned to each processor are too small to work with and more time is spent carrying out inter-processor communication. On the other hand, when the threshold is large, the idle time increases (the red portion in the figure), because the regions assigned to each processor are too big to work with and the processor has to too much work to do before it can carry them to the other processors.

When we compare Figure 4.1(a) and Figure 4.1(b), the most apparent difference is the percentage of the communication time denoted by the blue portion. The percentage of the communication time on Beowulf machine is larger than that on Sunfire machine. This is because the Sunfire machine uses the shared memory and its message passing is actually the memory copy, much faster than the message passing on the Beowulf machine that uses the Gigabit ethernet. The other difference between these two figures is the total time spent. The total time on the Beowulf is roughly one fourth to one third of the total time on the Sunfire because Beowulf has more powerful CPUs. From the speedup diagrams in Figure 4.2, we can see that it runs faster on the Beowulf machine due to the processors with higher speed. However, the speedup on Beowulf is lower than that on Sunfire, because the Sunfire machine uses the shared memory, and reduces the efficiency lost on communication. We also observe that when the SWP tree is around 4 levels in 2x2, and around 3 levels in 3x3 and 4x4, we can get

the best performance. This is a good hint for us to choose the threshold. If we are using 4 processors, we can choose a SWP threshold so that the SWP tree has 4 levels. If we are using 9 or 16 processors, we can choose a threshold so that that the SWP tree has 3 levels.

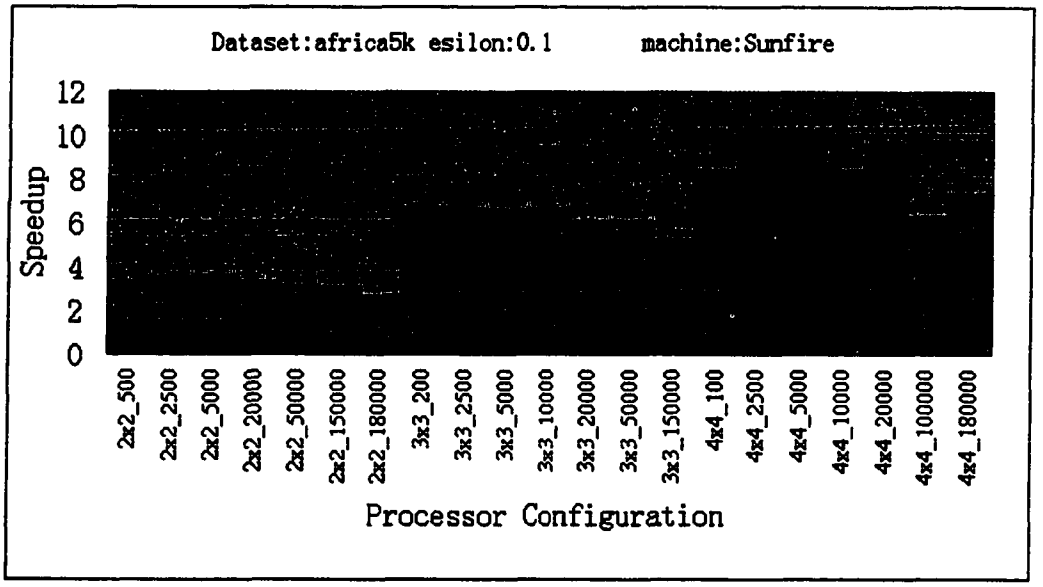


(a) Sunfire machine

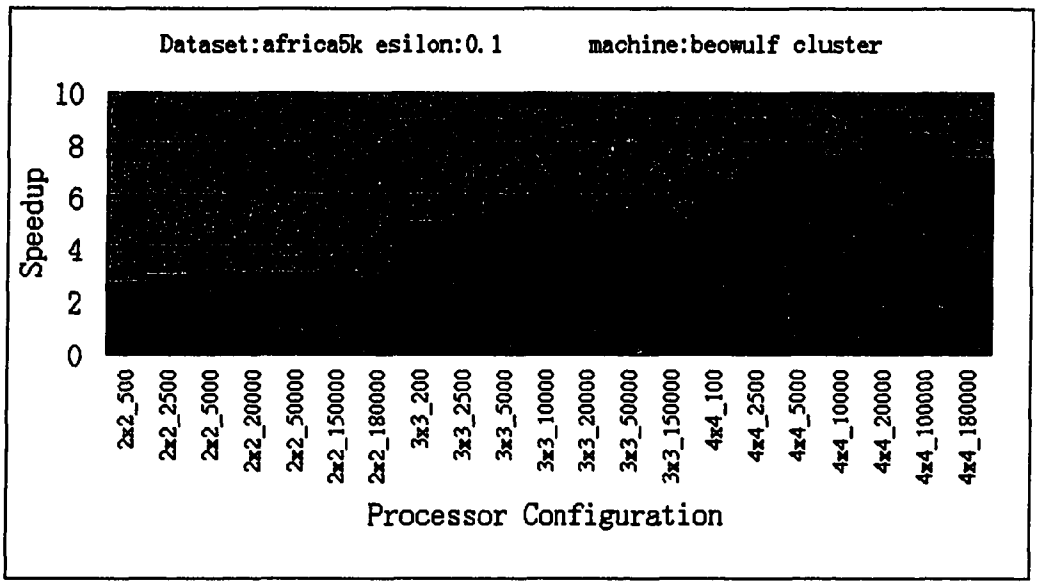


(b) Beowulf machine

Figure 4.1: Processing Time Distribution Diagrams, africa5k, $\epsilon = 0.1$



(a) Sunfire machine



(b) Beowulf machine

Figure 4.2: Speedup Diagrams, africa5k, $\epsilon = 0.1$

Processor configuration	Sunfire		Beowulf	
	NHO_p	$\frac{NHO_p}{NHO_1}$	NHO_p	$\frac{NHO_p}{NHO_1}$
1x1	50942592	1.00	50942592	1.00
2x2	55717573	1.09	56792131	1.11
3x3	63258358	1.24	65435556	1.28
4x4	71063523	1.40	70912849	1.39

Table 4.4: Number of heap operations (africa5k, ϵ=0.1)

4.2.3 Heap operation statistics

In this sub-section, we analyze the number of heap operations on the processors, to see (a) how much over-computation it costs; (b) the computation load distribution among the processors.

There are three basic heap operations:

- INSERT: INSERT operation is called while the propagation reaches a new node.
- UPDATE: While a node is relaxed, UPDATE operation is called.
- EXTRACT_MIN: This operation extracts the node in the heap with the minimum cost, and adjusts the heap.

We denote NHO as the total number of heap operations (including all of the three mentioned operations). We count NHO for a single query on each processor and sum them to get NHO_p. NHO_1 is the NHO with only 1 processor (sequential case). Table 4.4 summaries the NHOs of different processor configurations. The ratio $\frac{NHO_p}{NHO_1}$ can present how much is the over-computation. We can find that when using 4x4 processors, nearly 40% of the computation is wasted by the over-computation.

Table 4.5 shows the NHO distribution over processors. It lists the number of heap operations on each processor. Using the data in Table 4.4 and 4.5, we draw Figure 4.3, which visually demonstrates the heap operations among the processors on the

Processor config		Sunfire	Beowulf	Processor config		Sunfire	Beowulf
1x1	1	50942592	50942592	4x4	1	5654573	5461521
				4x4	2	4376926	4348932
2x2	1	14551434	14472843	4x4	3	4700265	4593551
2x2	2	13777434	14175986	4x4	4	4622159	4630927
2x2	3	13953166	14297448	4x4	5	3784954	3895394
2x2	4	13435539	13845854	4x4	6	4304492	4172768
				4x4	7	4057781	3899893
3x3	1	7052992	7149403	4x4	8	4405638	4553990
3x3	2	6924950	7102531	4x4	9	4306274	4431364
3x3	3	7063389	7322892	4x4	10	4223783	4343245
3x3	4	6557379	6911166	4x4	11	4015199	4302163
3x3	5	7198285	7320324	4x4	12	4045747	4194419
3x3	6	7319186	7487799	4x4	13	4820239	4745236
3x3	7	6770749	7385669	4x4	14	4100236	3965365
3x3	8	7459436	7595099	4x4	15	4426878	4413471
3x3	9	6911992	7160673	4x4	16	5218379	4960610

Table 4.5: Number of heap operations: detail information (africa5k, ϵ=0.1)

Beowulf machine. In this figure, the data of the reference line is computed by $\frac{NHO_1}{p}$, which means the ideal number of heap operations on each processor when NHO is evenly distributed among processors. The margin between the actual data line and the reference line represents the over-computation of each processor. From Figure 4.3, we can see that the computational load (represented by the NHO) is nicely distributed among the processors.

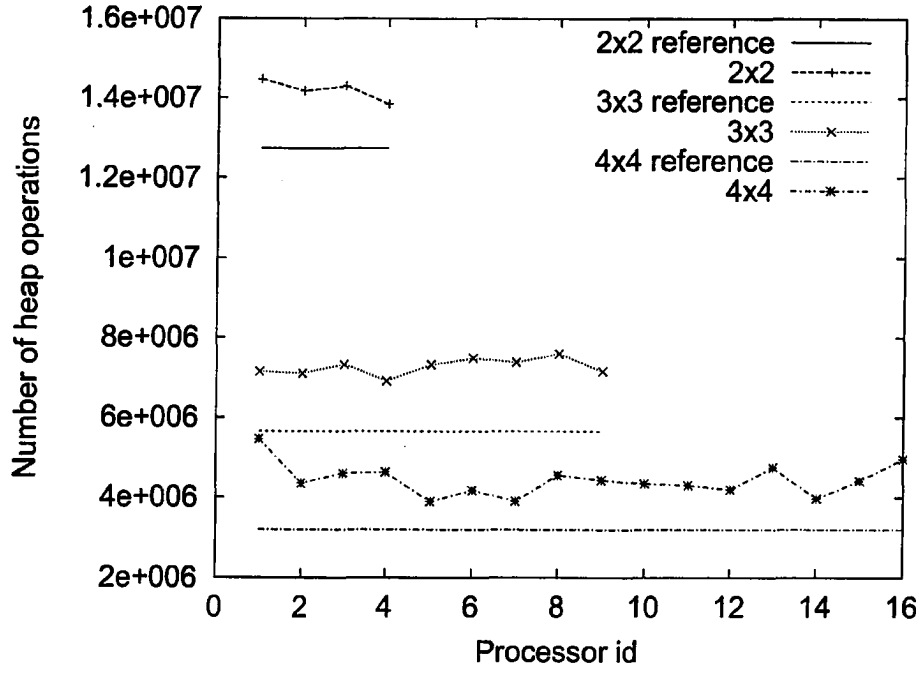


Figure 4.3: Heap operations distribution among processors on the Beowulf machine

4.3 Results on different ϵ and β values

4.3.1 Speedup and efficiency

Table 4.6 lists the time, speedup and efficiency for the experiments on different ϵ and β values. In these experiments, the concurrency threshold is fixed to 10. The "time" column in the table represents the total time in seconds for 50 random queries. The best speedup efficiency for 2x2, 3x3, 4x4 on Sunfire are 85.9%, 76.3% and 71.4%, respectively. The best speedup efficiency for 2x2, 3x3, 4x4 on Beowulf are 88.1%, 79.2% and 65.4%, respectively. From Table 4.6, we can read that, when $\epsilon = 0.01$ and $\beta = 0$, 50 shortest path queries will cost us 1,094,903.81 seconds (12.67 days) on the Sunfire machine. The running time is reduced to 135,537.3 seconds (1.57 days) when using 16 processors. This contrast really makes our parallel works worthwhile.

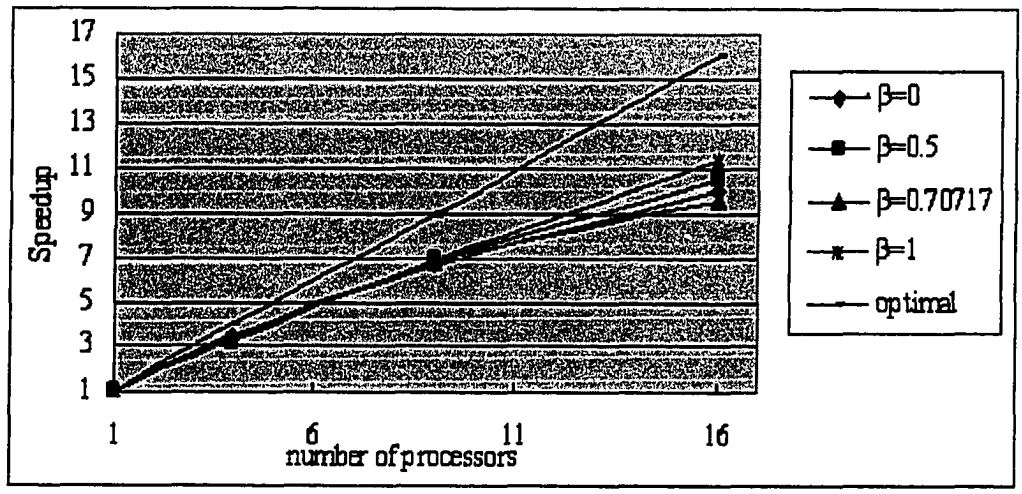
By using the data in Table 4.6, we draw speedup diagrams Figure 4.4 (on Sunfire machine) and Figure 4.5 (on Beowulf machine). Figure 4.4 (a), (b) and (c) show the speedup diagrams on Sunfire for different ϵ values: 0.1, 0.05 and 0.01 respectively. Figure 4.5 (a), (b) and (c) show the speedup diagrams on Beowulf for different ϵ values: 0.1, 0.05 and 0.01 respectively. From these diagrams, we find the speedups for our concurrent PSP are in a reasonably good range. We cannot observe obvious relationships between the speedup and either the parameter ϵ or the parameter β. However, we can observe a trend between the speedup, β and ϵ when analyzing the 4x4 configuration. Here the speedup range between the values of β decreases as ϵ decreases. We conjecture that this is due to load increasement on each processor as ϵ decreases. The β parameter reduces the number of Steiner points by a constant amount. Therefore, we expect the running time of each test to decrease by a constant

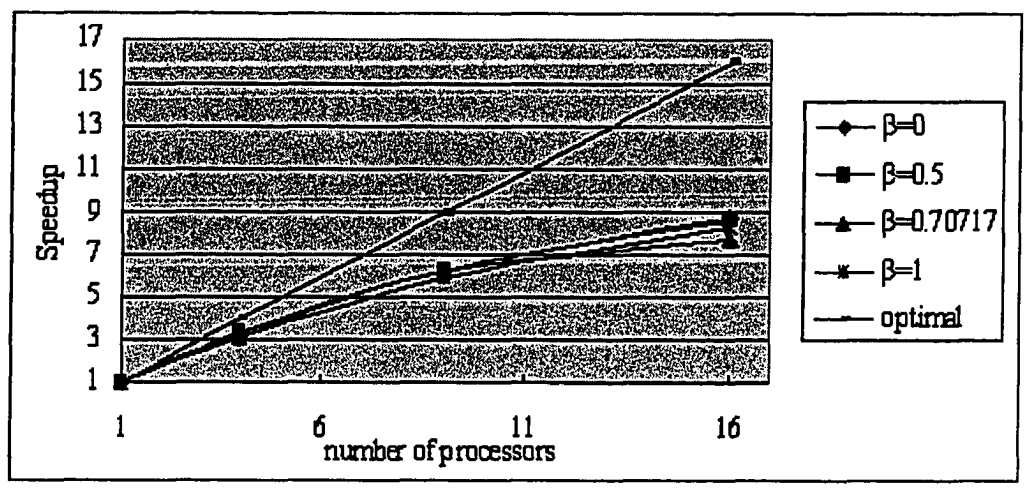
ϵ	β	processor configuration	Beowulf			Sunfire		
			time	speedup	efficiency	time	speedup	efficiency
0.1	0	1x1	16324.49	1	100.00%	57865.1	1	100.00%
0.1	0	2x2	5932.36	2.75	68.79%	16834.89	3.44	85.93%
0.1	0	3x3	3013.91	5.42	60.18%	8677.81	6.67	74.09%
0.1	0	4x4	2325.32	7.02	43.88%	5773.62	10.02	62.64%
0.1	0.5	1x1	9369.52	1	100.00%	32152.36	1	100.00%
0.1	0.5	2x2	3404.37	2.75	68.81%	9819.55	3.27	81.86%
0.1	0.5	3x3	1718.78	5.45	60.57%	4687.96	6.86	76.21%
0.1	0.5	4x4	1176.29	7.97	49.78%	3033.5	10.60	66.24%
0.1	0.707107	1x1	6454.52	1	100.00%	20231.32	1	100.00%
0.1	0.707107	2x2	2263.03	2.85	71.30%	5937.85	3.41	85.18%
0.1	0.707107	3x3	1162.29	5.55	61.70%	2969.41	6.81	75.70%
0.1	0.707107	4x4	723.32	8.92	55.77%	2109.78	9.59	59.93%
0.1	1	1x1	3589.25	1	100.00%	10954.74	1	100.00%
0.1	1	2x2	1295.84	2.77	69.25%	3393.38	3.23	80.71%
0.1	1	3x3	760.78	4.72	52.42%	1594.96	6.87	76.31%
0.1	1	4x4	429.93	8.35	52.18%	959.1	11.42	71.39%
0.05	0	1x1	55588.4	1	100.00%	149237.62	1	100.00%
0.05	0	2x2	15774.59	3.52	88.10%	44791.28	3.33	83.30%
0.05	0	3x3	7831.75	7.10	78.86%	24023.17	6.21	69.02%
0.05	0	4x4	6169.86	9.01	56.31%	17169.72	8.69	54.32%
0.05	0.5	1x1	27669.52	1	100.00%	81631.02	1	100.00%
0.05	0.5	2x2	8837.92	3.13	78.27%	24401.17	3.35	83.63%
0.05	0.5	3x3	4438.38	6.23	69.27%	13117.82	6.22	69.14%
0.05	0.5	4x4	3271.68	8.46	52.86%	9445.35	8.64	54.02%
0.05	0.707107	1x1	16209.39	1	100.00%	51405.33	1	100.00%
0.05	0.707107	2x2	5540.64	2.93	73.14%	16211.74	3.17	79.27%
0.05	0.707107	3x3	2849.08	5.69	63.21%	8259.42	6.22	69.15%
0.05	0.707107	4x4	2271.56	7.14	44.60%	6640.62	7.74	48.38%
0.05	1	1x1	8441.47	1	100.00%	27798.55	1	100.00%
0.05	1	2x2	3006.44	2.81	70.19%	8810.49	3.16	78.88%
0.05	1	3x3	1630.56	5.18	57.52%	4659.69	5.97	66.29%
0.05	1	4x4	1041.17	8.11	50.67%	3337.92	8.33	52.05%
0.01	0	1x1	384127.93	1	100.00%	1094903.81	1	100.00%
0.01	0	2x2	113853.44	3.37	84.35%	382723.11	2.86	71.52%
0.01	0	3x3	53872.54	7.13	79.23%	197058.24	5.56	61.74%
0.01	0	4x4	38244.96	10.04	62.77%	135537.3	8.08	50.49%
0.01	0.5	1x1	193012.05	1	100.00%	603483.54	1	100.00%
0.01	0.5	2x2	62414.84	3.09	77.31%	209498.72	2.88	72.02%
0.01	0.5	3x3	29015.6	6.65	73.91%	97684.11	6.18	68.64%
0.01	0.5	4x4	19489.9	9.90	61.89%	69653.94	8.66	54.15%
0.01	0.707107	1x1	121380.87	1	100.00%	389740.97	1	100.00%
0.01	0.707107	2x2	37183	3.26	81.61%	121477.21	3.21	80.21%
0.01	0.707107	3x3	20250.03	5.99	66.60%	61778.33	6.31	70.10%
0.01	0.707107	4x4	11605.59	10.46	65.37%	43117.46	9.04	56.49%
0.01	1	1x1	62093.79	1	100.00%	199138.8	1	100.00%
0.01	1	2x2	18684.06	3.32	83.08%	62799.17	3.17	79.28%
0.01	1	3x3	10347.59	6.00	66.68%	34346.84	5.80	64.42%
0.01	1	4x4	6101.01	10.18	63.61%	22222.4	8.96	56.01%

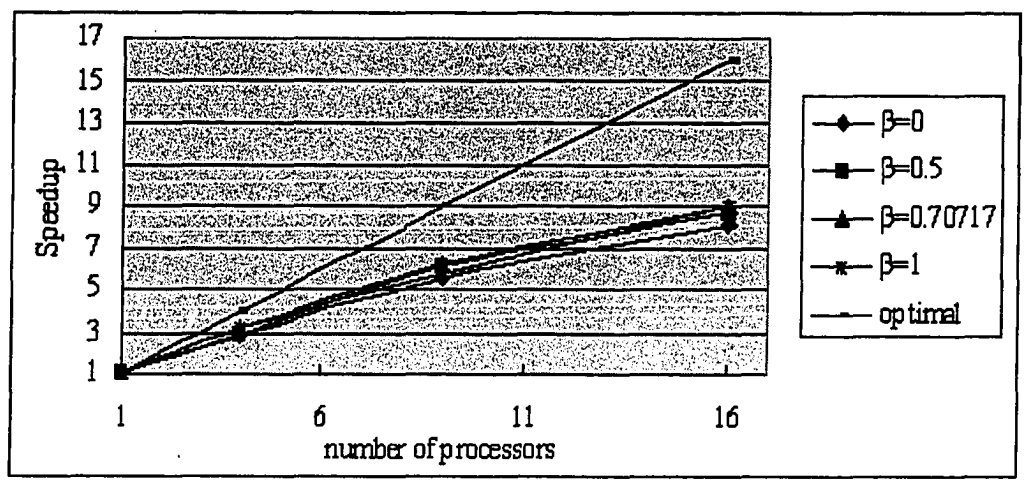
Table 4.6: Results on different ϵ and β (africa5k, 50 random queries)
time unit: second

factor, which means that the relative change in the speedup is the same. In the case when $\epsilon = 0.1$, the amount of work on each processor is very low and therefore the range of speedup is larger. In the case when $\epsilon = 0.01$, there are 10 times more Steiner points. Thus when β reduces the number of Steiner points, the effect of speed reduction is constant. This is strongly evident in the 2x2 cases when an increasement in β reduces the computation time by a constant amount regardless of ϵ. This is an observation that must be substantiated in future work by examing more data sets and different type of queries.

Figure 4.6 shows the timing for the dataset:"africa5k" on the Sunfire machine (Figure 4.6a) and the Beowulf machine (Figure 4.6b). In these figures, there are four points on each vertical data line. From top to bottom, they are representing the total time for 50 queries by 1x1, 2x2, 3x3 and 4x4 processors, respectively. The x-axis is for the number of Steiner points generated from the TIN according to different ϵ and parameter β values. These timing diagrams clearly show that, when the total number of Steiner points increases, the application is more time-consuming. If we connect the top points on every vertical data line (these points represent the time spent when using single processor), we can see a curve more like quadratic than $O(nlog(n))$. This is mainly because our short path algorithm implementation uses normal heap instead of the Fibonacci heap.

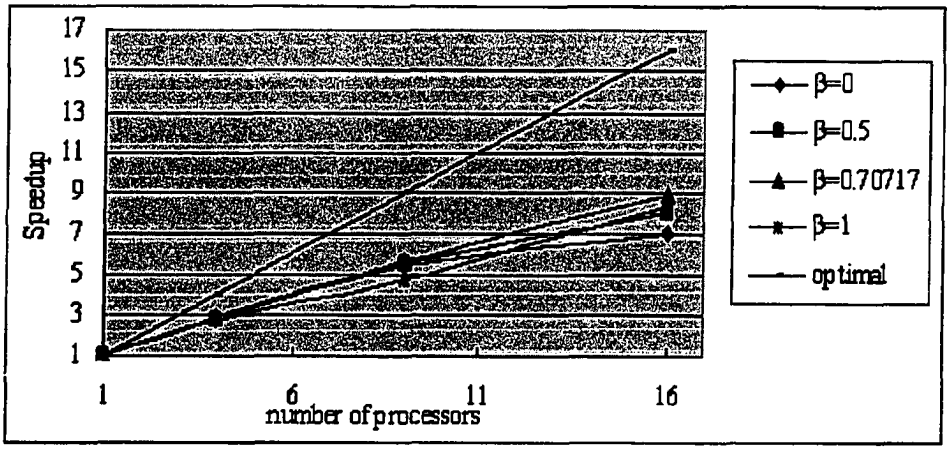




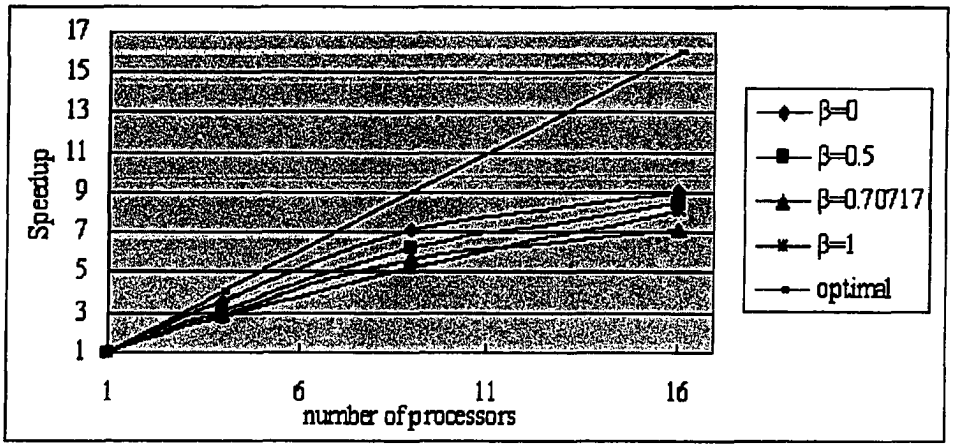


(c) Speedup diagram: Sunfire, africa5k, ϵ=0.01

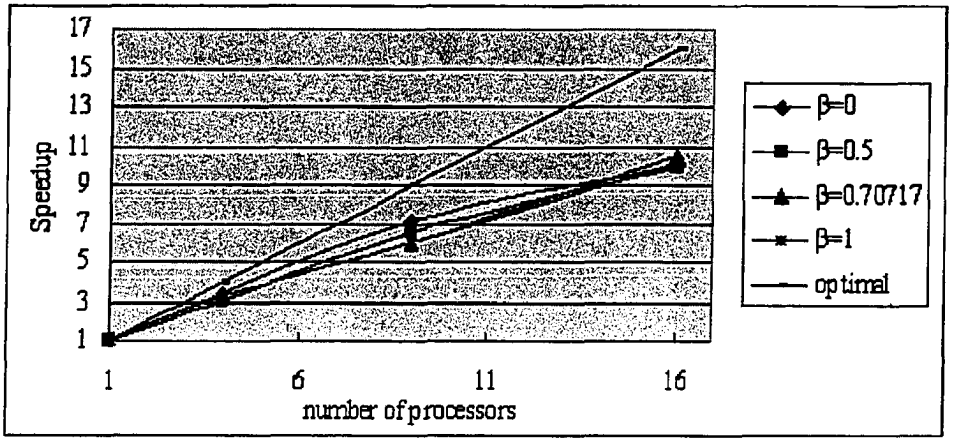
Figure 4.4: Speedup Diagrams, Sunfire, africa5k



(a) Speedup diagram: Beowulf, africa5k, ϵ=0.1

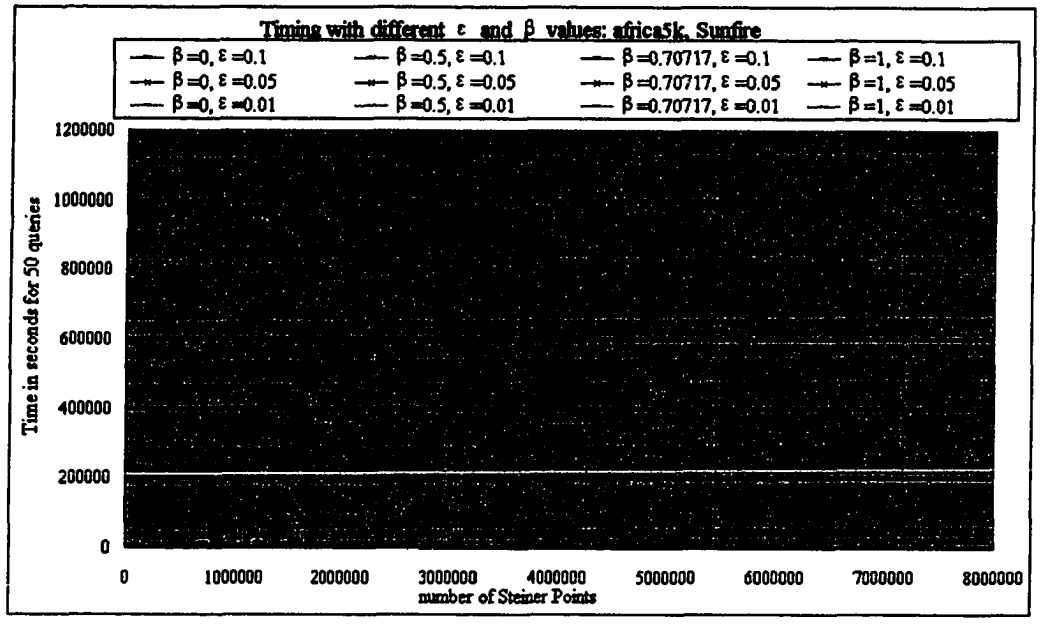


(b) Speedup diagram: Beowulf, africa5k, ϵ=0.05

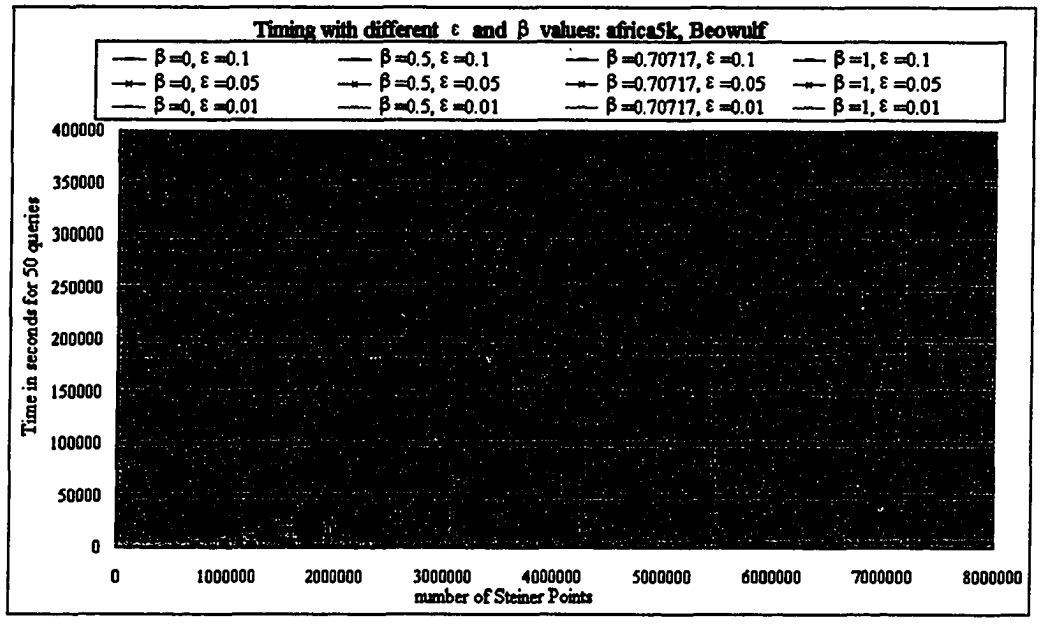


(c) Speedup diagram: Beowulf, africa5k, ϵ=0.01

Figure 4.5: Speedup Diagrams, Beowulf, africa5k



(a) Sunfire machine



(b) Beowulf machine

Figure 4.6: Timing on different ϵ and β values, africa5k

4.3.2 Length accuracy

In Section 3.1, we have discussed the influence of parameter β on the approximation accuracy. In the experiments, we measure the length of the generated path and compare the average length of the paths with different β and ϵ values. Tables 4.7 summarizes the results on single processor with different ϵ and β values. The third column in Table 4.7(a) lists the numbers of Steiner points generated according to the ϵ and β values. The fourth column in Table 4.7(a) is the total time (in seconds) it takes a processor to finish those queries on the Beowulf machine. The fifth column in Table 4.7(a) is the total time (in seconds) it takes a processor to finish those queries on the Sunfire machine. The last column in Table 4.7(a) is the average length of the computed approximation shortest paths for the 50 queries. In different test rounds with same queries, we can always get the same approximated shortest paths. This means our PSP implementation can consistently compute the correct shortest paths as in sequential implementation.

From Table 4.7(a), we can calculate some corresponding ratios as shown in Table 4.7(b). In Table 4.7(b), the length ratio (the 6th column) is computed as $\frac{L_\beta}{L_0}$, where L_0 is the average length of the computed paths for 50 queries when $\beta = 0$, and L_β is the average length of the computed paths for these 50 queries with parameter β. Table 4.7(c) offers another view of the length ratio by listing the minimum, maximum and average values of $\frac{L_{\beta i}}{L_{0i}}$, where $L_{\beta i}$ denotes the length of the path for the ith query using β variation, and L_{0i} denotes the length of the path for the ith query without β variation (i.e. $\beta = 0$). Therefore,

$$average(\frac{L_{\beta i}}{L_{0i}}) = \frac{1}{m}\sum_{i=1}^{m}\frac{L_{\beta i}}{L_{0i}},$$

ϵ	β	Steiner points #	time(Beowulf)	time(Sunfire)	avg. length
0.1	1.0	714157	3589.25	10954.74	25876.3977
0.1	0.707107	974366	6454.52	20231.32	25859.8339
0.1	0.5	1225810	9369.52	32152.36	25853.1695
0.1	0	1597717	16324.49	57865.10	25846.6442
0.05	1.0	1119840	8441.47	27798.55	25861.2532
0.05	0.707107	1532017	16209.39	51405.33	25853.3436
0.05	0.5	1942317	27669.52	81631.02	25848.1477
0.05	0	2539195	55588.40	149237.62	25844.8775
0.01	1.0	3076005	62093.79	199138.80	25847.2056
0.01	0.707107	4274518	121380.87	389740.97	25844.8799
0.01	0.5	5459143	193012.05	603483.54	25844.0532
0.01	0	7169460	384127.93	1094903.81	25843.3147

(a) time unit: second

ϵ	β	Steiner points ratio	speed ratio (Beowulf)	speed ratio (Sunfire)	average length ratio
0.1	1	44.70%	4.55	5.28	1.00115
0.1	0.707107	60.98%	2.53	2.86	1.00051
0.1	0.5	76.72%	1.74	1.80	1.00025
0.1	0	100.00%	1	1	1
0.05	1	44.10%	6.59	5.37	1.00063
0.05	0.707107	60.33%	3.43	2.90	1.00033
0.05	0.5	76.49%	2.01	1.83	1.00013
0.05	0	100.00%	1	1	1
0.01	1	42.90%	6.19	5.50	1.000151
0.01	0.707107	59.62%	3.16	2.81	1.000061
0.01	0.5	76.14%	1.99	1.81	1.000029
0.01	0	100.00%	1	1	1

(b) some ratios

ϵ	β	$\min(\frac{L_{\beta i}}{L_{0i}})$	$\max(\frac{L_{\beta i}}{L_{0i}})$	$\text{average}(\frac{L_{\beta i}}{L_{0i}})$
0.1	1	1.00019	1.0022	1.0012
0.1	0.707107	1.00012	1.0027	1.00059
0.1	0.5	1.000010	1.0011	1.00025
0.05	1	1.00012	1.0028	1.00070
0.05	0.707107	1.000019	1.00091	1.00030
0.05	0.5	1.000014	1.0012	1.00016
0.01	1	1.000029	1.00054	1.00016
0.01	0.707107	1.0000035	1.00045	1.000080
0.01	0.5	1.0000012	1.00015	1.000029

(c) length ratio (minimum, maximum and average)

Table 4.7: Results on different ϵ and β with single processor (africa5k, 50 random queries)

where m is the number of queries. This value is not the same with the 6th column in Table 4.7(b), which is computed by

$$average\ length\ ratio = \frac{\sum_{i=1}^{m} L_{\beta i}}{\sum_{i=1}^{m} L_{0i}}.$$

We can observe from Table 4.7(c) that, even the maximum length ratio is very close to 1.0. It means that the length of the output path is within a small factor of the length of the ϵ-approximation shortest path.

The speed ratio in Table 4.7(b) is computed by $\frac{T_0}{T_\beta}$, where T_β is the total time spent with parameter β, and T_0 is the total time spent when $\beta = 0$.

The number of Steiner points ratio is computed by $\frac{NSP_\beta}{NSP_0} \times 100\%$, where NSP_β is the number of Steiner points with parameter β, and NSP_0 is the number of Steiner points when $\beta = 0$.

Using the data in Table 4.7(b), we construct three diagrams in Figure 4.7, which show the impact of parameter β on the length accuracy, the number of Steiner points, and speed respectively.

Figure 4.7(a) shows that in our experiments, the length of computed shortest paths with parameter β (denoted by L_β) is very close to the length of the ϵ-approximation shortest paths (denoted by L_0). For example, when $\epsilon = 0.1$, the difference between L_β and L_0 is within 0.12%.

Recall that $\lambda = \left(1 + \sqrt{\frac{\epsilon}{2}} \times max\left(\sin\frac{\alpha}{2}, \beta\right)\right)$. When $\beta = 0$, it satisfies the ϵ-approximation, which means: $|SP|_0 \leqslant (1+\epsilon)|SP|$, where $|SP|_0$ denotes the length of the shortest path computed by the algorithm, $|SP|$ denotes the length of the actual shortest path.

From Table 4.7, we can see, even when β is set to 1, the length is still very close to

the ϵ-approximation path length. From the last row of Table 4.7a, we know that,

$$|SP|_0 \leqslant (1+\epsilon)|SP|, \; here \, |SP|_0 = 25843.3147, \epsilon = 0.01.$$

$$\Longrightarrow |SP| \geqslant \frac{|SP|_0}{1+\epsilon} = 25587.4403$$

Now, in the case $\epsilon = 0.01, \beta = 1$, the current approximation ratio is,

$$1+\epsilon' = \frac{|SP|_1}{|SP|} \leqslant \frac{25847.2056}{25587.4403} = 1.0102 \Longrightarrow \epsilon' \leqslant 0.0102.$$

In the case $\epsilon = 0.05, \beta = 1$, the current approximation ratio is,

$$1+\epsilon' = \frac{|SP|_1}{|SP|} \leqslant \frac{25861.2532}{25587.4403} = 1.0107 \Longrightarrow \epsilon' \leqslant 0.0107.$$

In the case $\epsilon = 0.1, \beta = 1$, the current approximation ratio is,

$$1+\epsilon' = \frac{|SP|_1}{|SP|} \leqslant \frac{25876.3977}{25587.4403} = 1.0113 \Longrightarrow \epsilon' \leqslant 0.0113.$$

Therefore, the experimental results show that when $\beta = 1$, the approximation of the shortest path is still very good.

From Figure 4.7(b), we also notice that when $\beta = 1$, the total number of Steiner points on surface $\mathcal{P}$ is less than half of the number when $\beta = 0$. From Figure 4.7(c), we understand that the application is running faster with less Steiner points involved.

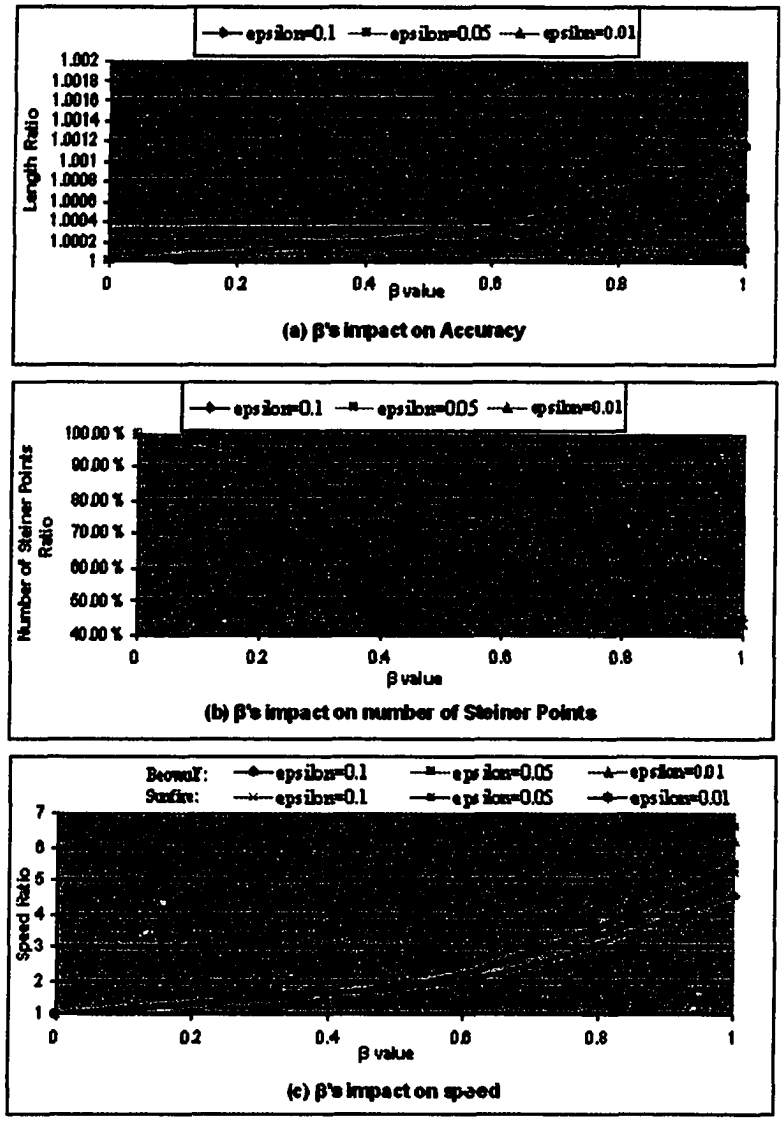


Figure 4.7: Impact of β, africa5k

4.4 Concurrency and query distribution

In Section 3.3, we have discussed that concurrent PSP is expected to handle concurrent queries more efficiently. The concurrency does not mitigate the computational load. Instead, it utilizes the idle time on the processors to let the computation start earlier, then consequently, reduce the total time for the concurrent queries.
In the following sub sections, we start with the introduction of the test sets. Then we will show the test results on each scenario.

4.4.1 Test sets

The following experiments in this section are executed on the Beowulf machine. The chosen dataset is africa5k, $\epsilon = 0.1$, $\beta = 1$.
To study the performance on different query distributions, we prepare three scenarios: random query distribution, clustered query case 1, clustered query case 2. In tests with random query distribution, we have done experiments on 50 randomly generated query pairs, as well as on 500 randomly generated short queries. We define the terms 'long', 'median' and 'short' differently in case 1 and case 2.
In clustered query case 1, distance is measured by shortest path length. Given a source point, we perform a one-to-all shortest path computation and sort the points (TIN vertices as well as Steiner points) according to the length of their shortest paths from the source point. A 'long'/'median'/'short' query is a query from the source to a point in the longer/middle/shorter 1/3 of the points.
In clustered query case 2, distance is measured by the 'hops' of the shortest path (number of faces crossed by the path). Given a source point, we find its located

face, then we run a BFS search on the faces, and count the distance from the source face. The faces are sorted according to their distance to the source face. A 'long'/'median'/'short' query is a query from the source to a point located in the further/middle/nearer 1/3 of the faces.

In each clustered query case, we randomly pick 5 source points, from each source point, we randomly pick a 'short' queries, b 'median' queries and c 'long' queries. We choose 3 test sets for each of the case:

- Long-paths-dominated test set: In a long-paths-dominated test set (5127), for each source point, we pick 1 'short' query, 2 'median' queries and 7 'long' queries.
- Balanced-distance test set: In a balanced-distance test set (5433), for each source point, we pick 4 'short' queries, 3 'median' queries and 3 'long' queries.
- Short-paths-dominated test set: In a short-paths-dominated test set (5721), for each source point, we pick 7 'short' queries, 2 'median' queries and 1 'long' query.

Therefore, we have 50 queries starting from 5 different sources in each of the clustered case. Figure 4.8 demonstrates the query distribution in each test case. Figure 4.8(a) and (d) are long-paths-dominated test sets; Figure 4.8(b) and (e) are Balanced-distance test sets; Figure 4.8(c) and (f) are short-paths-dominated test sets. Figure 4.8(g) and (h) represent random (evenly) distribution test set with 50 queries and 500 short queries respectively. The black boxes in the figure represent the Minimum Bounding Rectangle of the TIN map: africa5k. Red points represent the source points and black points represent the destination points. The lines just represent the queries and they are not the actual shortest paths.

Let T_a denote the time when a query is accepted by the system, and T_f denote the time when the query is finished with generated path information. The response time $t_{ri} = T_f - T_a$, reflects how long the user has to wait after the ith query is submitted. The average response time t_{rav} is a good indicator of the system performance. $t_{rav} = \frac{1}{m}\sum_{i=1}^{m} t_{ri}$, where m is the total number of queries. In our experiments, we also compared average response time in the following tests to show how the concurrency contributes to the performance.

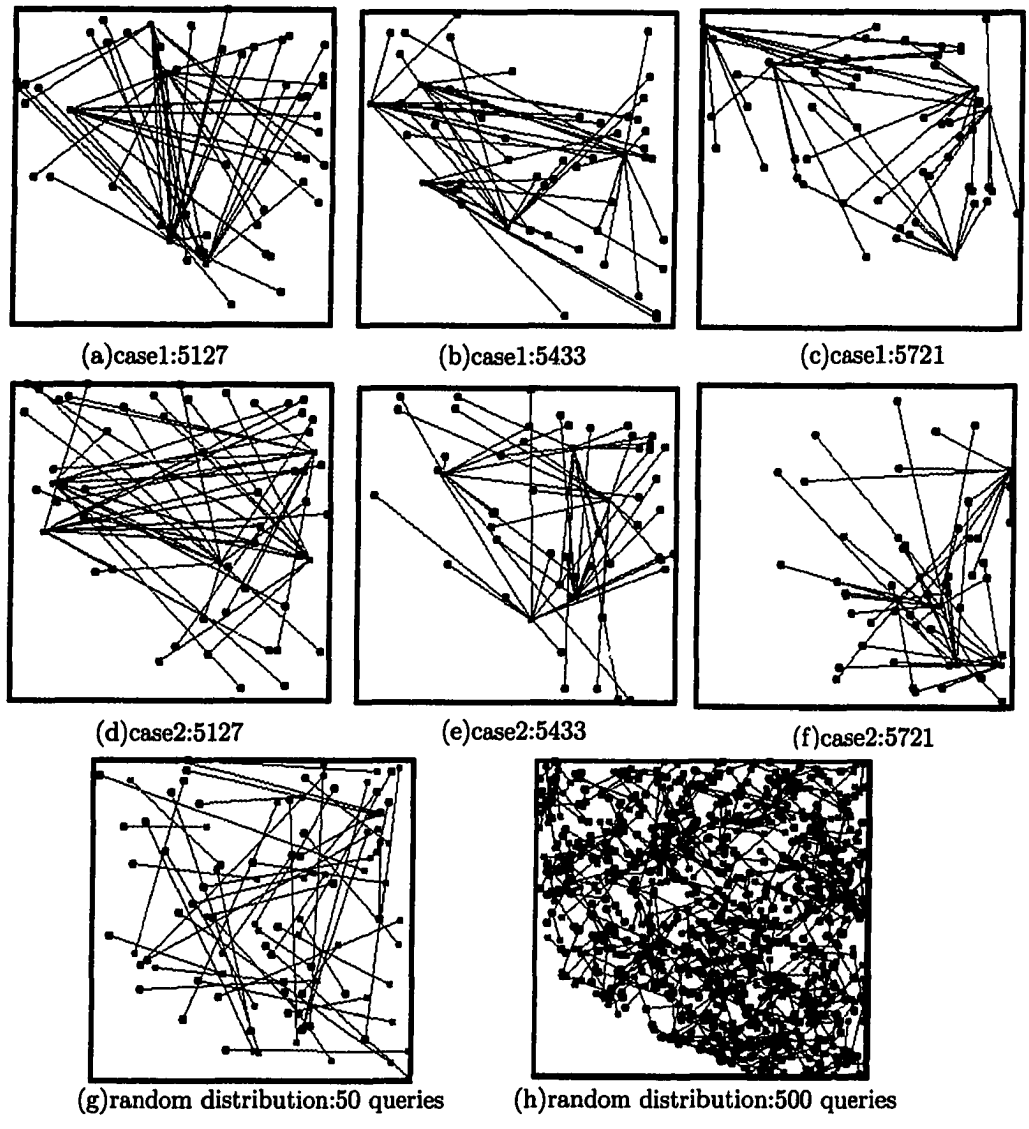


Figure 4.8: Query distribution demonstration

Red points represent source points. Black points represent destination points. A line represents a query from source point to destination point. In case 1, distance is measured by shortest path length; in case 2, distance is measured by the 'hops' of the shortest path. 'case1:5127' means there are 5 source points; from each source point, there is 1 'short' query, 2 'median' queries and 7 'long' queries. The terms 'short', 'median' and 'long' are measured according to case1 scenario: distance is measured by shortest path length.

4.4.2 Random distribution

We used the queries test sets as shown in Figure 4.8(g) and (h) with different concurrency thresholds: 1, 5, 20 and 50. The results are shown in Table 4.8 (a) and (b) respectively. Using these results data, we make three kind of diagrams: timing distribution diagram, average response time diagram and speedup diagram, to illustrate the impact of concurrency threshold. Figure 4.9 contains the diagrams for the test set of 50 random queries. Figure 4.10 contains the diagrams for the test set of 500 random short queries.

Figure 4.9 and Figure 4.10 all show a big difference between the test with concurrency threshold =1 and the tests with concurrency threshold $\geqslant 5$. From the speedup diagrams Figures 4.9(c) and Figures 4.10(c), we can clearly observe the speedup improvement while handling queries concurrently. For the 50 queries, the efficiency goes up around 8% in 2x2, around 13% in 3x3 and around 10% in 4x4. For the 500 short queries, the efficiency goes up around 18% in 2x2, around 23% in 3x3 and around 15% in 4x4.

The decrease of the response time shown in Figure 4.9(c) and Figure 4.10(c) also reflects this performance enhancement.

Another important phenomenon is that, the performance differs very little no matter if the concurrency threshold is 5, 10 or 20. This seems to be odd, however, Figure 4.9(a) gives us a good explanation. First, it shows a dramatic drop on the idle time from the test of threshold 1 to the test of threshold 5. The idle time has been decreased to a very small amount, from which larger threshold cannot benefit. It also shows that the computation time and the communication time remain almost the same. Therefore, the performance enhancement comes from the decrease of the idle

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	3264.64	3264.64	1	100.00%	1592.59
50	2x2	132.16	3.35	1059.39	1194.9	2.73	68.30%	577.47
50	3x3	85.46	25.53	571.84	682.83	4.78	53.12%	333.16
50	4x4	82.59	5.9	347.19	435.69	7.49	46.83%	211.42
20	1x1	0.01	0	3263.2	3263.2	1	100.00%	1594.03
20	2x2	132.45	2.33	1058.32	1193.11	2.74	68.38%	579.50
20	3x3	82.72	22.88	553.43	659.03	4.95	55.02%	332.15
20	4x4	81.94	8.7	343.34	433.98	7.52	47.00%	212.88
5	1x1	0.01	0	3260.74	3260.75	1	100.00%	1587.52
5	2x2	131.29	14.92	1049.57	1195.77	2.73	68.17%	581.27
5	3x3	85.63	41.36	566.75	693.74	4.7	52.23%	337.34
5	4x4	82.25	17.9	340.3	440.45	7.4	46.27%	212.82
1	1x1	0.03	0	3255.88	3255.91	1	100.00%	1591.68
1	2x2	133.42	164.59	1036.51	1334.52	2.44	60.99%	650.08
1	3x3	79.89	318.6	500.1	898.59	3.62	40.26%	431.35
1	4x4	76.26	179.91	305.19	561.36	5.8	36.25%	278.40

(a) 50 random distributed queries

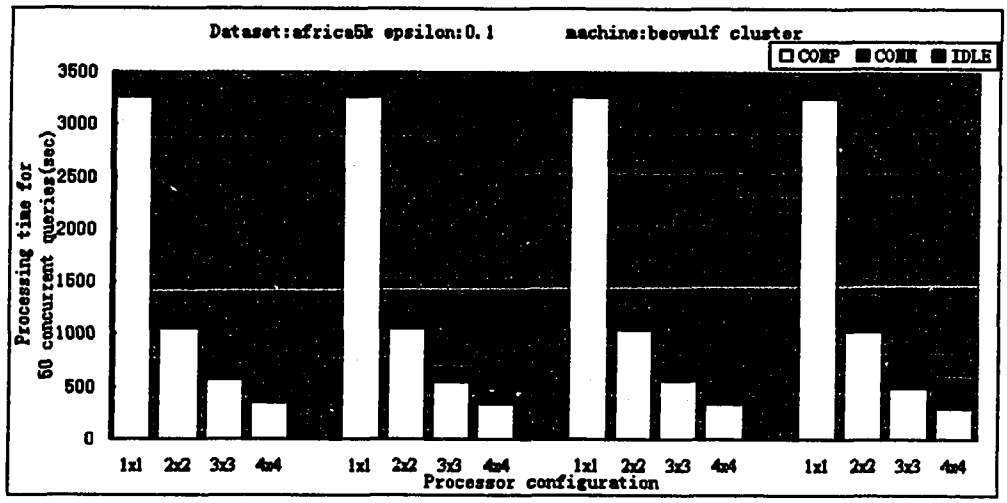
Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.06	0	3090.33	3090.39	1	100%	1517.07
50	2x2	154.32	1.99	1126.33	1282.63	2.41	60.24%	643.62
50	3x3	108.13	8.79	646.46	763.38	4.05	44.98%	379.09
50	4x4	92.31	6.44	352.12	450.87	6.85	42.84%	221.96
20	1x1	0.06	0	3077.31	3077.37	1	100%	1511.51
20	2x2	152.52	1.81	1121.1	1275.42	2.41	60.32%	643.07
20	3x3	107.56	17.77	638.41	763.73	4.03	44.77%	376.07
20	4x4	93.32	11.98	354.47	459.77	6.69	41.83%	228.61
5	1x1	0.06	0	3073.67	3073.73	1	100%	1510.33
5	2x2	152.96	55.41	1105.5	1313.88	2.34	58.49%	666.12
5	3x3	104.05	141.31	592.34	837.69	3.67	40.77%	414.48
5	4x4	90.12	90.89	329.56	510.57	6.02	37.63%	254.44
1	1x1	0.26	0	3085.73	3085.98	1	100%	1516.79
1	2x2	156.24	591.63	1057.34	1805.21	1.71	42.74%	903.78
1	3x3	102.29	935.68	531.35	1569.32	1.97	21.85%	771.71
1	4x4	87.35	704.77	295.33	1087.45	2.84	17.74%	544.72

(b) 500 random distributed short queries

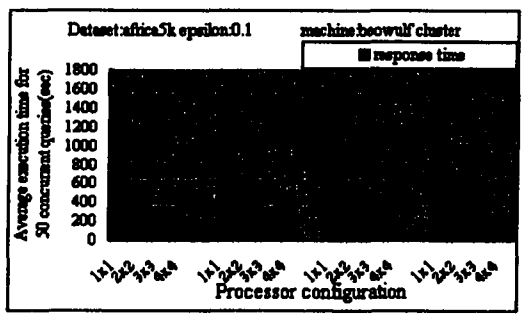
Table 4.8: Results with different concurrency thresholds
time unit: second

time. While all the processors are already very busy with 5 concurrent queries, we cannot expect more performance enhancement by increasing the concurrency threshold.

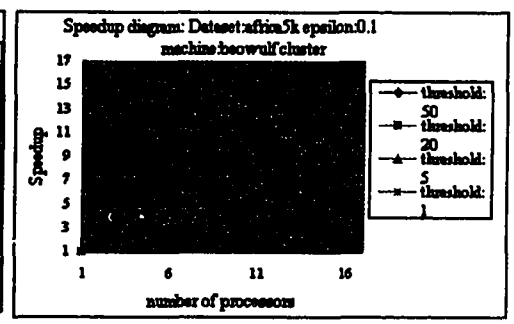
Comparing Figure 4.9(c) with Figure 4.10(c), we find that the efficiency on short queries is lower, especially when threshold is 1 (without concurrency). This is because the path of the short query might not even cross the partitions, so other processors can not help with the computation. We also see that, concurrency can reduce such efficiency gap. For example, when threshold=1, the efficiency difference on 4x4 processors on these two test sets is (36.25%-17.74%)=18.51%; when threshold=20, the efficiency difference is (47.00%-41.83%)=5.17%. Therefore, using concurrency, we can achieve good performance even with short queries.



(a) timing



(b) Response time



(c) speedup

Figure 4.9: Results with different concurrency thresholds (50 random distributed queries)

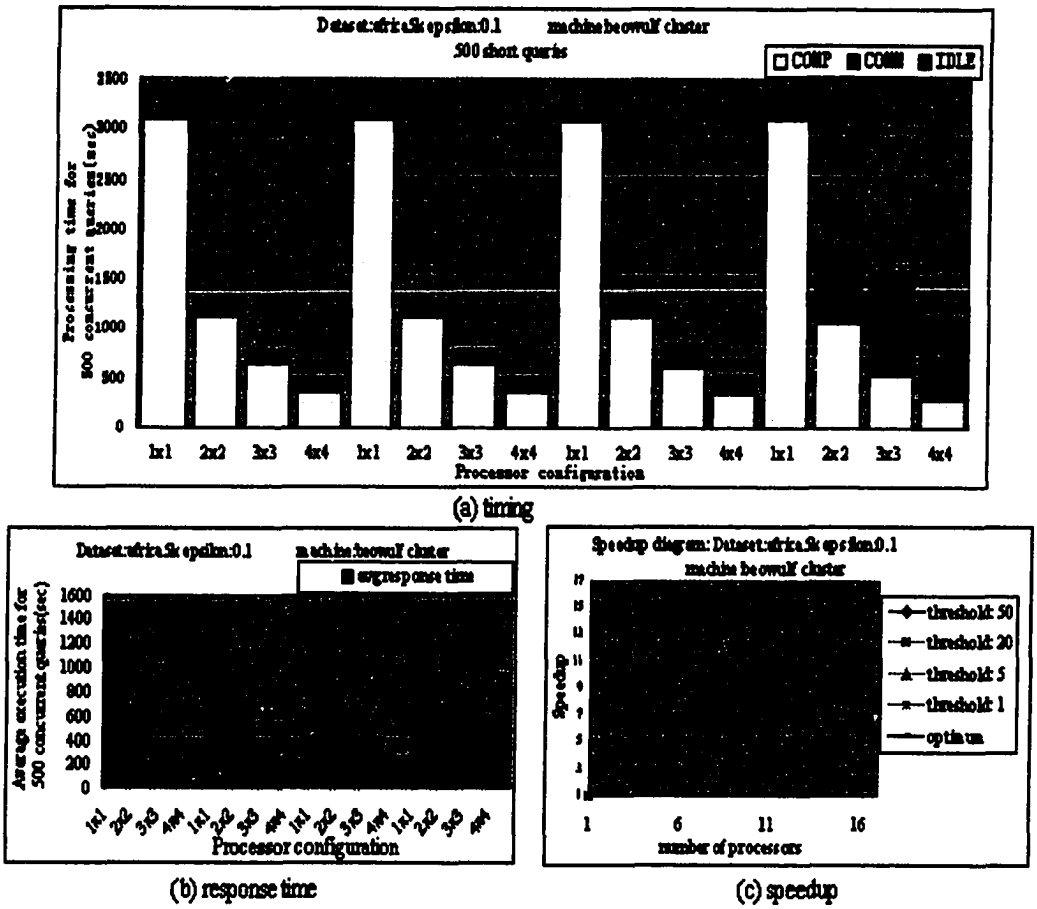


Figure 4.10: Results with different concurrency thresholds (500 random distributed short queries)

4.4.3 Clustered query distribution case 1

We used the test sets as shown in Figure 4.8(a),(b),(c) with different concurrency thresholds: 1, 5, 20 and 50. The results are shown in Table 4.9(a), (b), (c) respectively. Figure 4.11 shows the results for long-paths-dominated test set. Figure 4.12 shows the results for balanced-distance test set. Figure 4.13 shows the results for short-paths-dominated test set. The results exhibit similar features as discussed in the previous subsection: the speedup is increased when the concurrency threshold is greater than 1; but it does not go higher when the idle time has dropped to a very small value. When we compare the figures in Figure 4.8, Figure 4.12 and Figure 4.13, we can find that the running time for long queries is longer than that for short queries, however, the speedup for long queries is higher than that for short queries. This is because the long queries have more chances to cross more partitions which belong to different processors. Hence more processors can be able to get involved so that we can archive higher speedup.

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	4278.65	4278.66	1	100.00%	1810.82
50	2x2	170.56	3.47	1385.93	1559.96	2.74	68.57%	664.41
50	3x3	117.3	27.31	782.9	927.51	4.61	51.26%	397.66
50	4x4	107.45	10.24	447.64	565.33	7.57	47.30%	244.76
20	1x1	0.01	0	4296.05	4296.06	1	100.00%	1815.08
20	2x2	170.74	9.47	1384.83	1565.04	2.75	68.63%	665.48
20	3x3	119.04	24.11	801.86	945	4.55	50.51%	401.51
20	4x4	108.15	10.55	451.32	570.02	7.54	47.10%	248.97
5	1x1	0.01	0	4273.95	4273.95	1	100.00%	1807.33
5	2x2	169.31	12.57	1369.62	1551.48	2.75	68.87%	663.90
5	3x3	116.94	46.86	776.8	940.61	4.54	50.49%	406.56
5	4x4	107.08	16.76	441.12	564.96	7.57	47.28%	245.82
1	1x1	0.03	0	4255.72	4255.75	1	100.00%	1801.57
1	2x2	170.93	189.6	1355.31	1715.84	2.48	62.01%	733.19
1	3x3	113.62	328.16	714.44	1156.22	3.68	40.90%	499.93
1	4x4	103.9	167.73	416.92	688.55	6.18	38.63%	302.72

(a) 5127: long-paths-dominated

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	2924.47	2924.47	1	100.00%	1111.66
50	2x2	114.34	5.85	932.64	1052.83	2.78	69.44%	406.89
50	3x3	80.31	17.15	547.65	645.1	4.53	50.37%	253.28
50	4x4	73.85	8.22	313	395.08	7.4	46.26%	156.43
20	1x1	0.01	0	2932.93	2932.93	1	100.00%	1111.42
20	2x2	114.05	11.28	925.83	1051.16	2.79	69.75%	406.10
20	3x3	79.87	19.75	543.62	643.24	4.56	50.66%	250.68
20	4x4	74.22	10.52	314.63	399.37	7.34	45.90%	157.13
5	1x1	0.01	0	2920.61	2920.61	1	100.00%	1110.20
5	2x2	113.63	24.79	916.82	1055.23	2.77	69.19%	407.13
5	3x3	79.12	42.11	532.78	654.01	4.47	49.62%	255.82
5	4x4	73.97	17.12	310.09	401.18	7.28	45.50%	157.26
1	1x1	0.03	0	2921.37	2921.4	1	100.00%	1109.97
1	2x2	119.18	184.93	923.45	1227.55	2.38	59.50%	481.27
1	3x3	77.13	281.83	486.49	845.45	3.46	38.39%	336.44
1	4x4	72.73	153.28	291.15	517.16	5.65	35.31%	212.34

(b) 5433: balanced-distance

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	1940.93	1940.94	1	100.00%	793.67
50	2x2	79.05	10.42	649.3	738.77	2.63	65.68%	307.67
50	3x3	49.41	9.89	330.78	390.08	4.98	55.29%	158.89
50	4x4	49.9	4.57	206.23	260.71	7.44	46.53%	109.07
20	1x1	0.01	0	1932.23	1932.23	1	100.00%	786.91
20	2x2	79.49	11.76	652.61	743.86	2.6	64.94%	307.24
20	3x3	49.36	12.66	331.5	393.52	4.91	54.56%	160.28
20	4x4	49.96	6.57	205.63	262.15	7.37	46.07%	109.69
5	1x1	0.01	0	1936.09	1936.1	1	100.00%	788.45
5	2x2	77.73	15.78	641.65	735.16	2.63	65.84%	305.54
5	3x3	50.28	25.52	334.02	409.82	4.72	52.49%	166.81
5	4x4	49.84	14.8	203.31	267.95	7.23	45.16%	113.18
1	1x1	0.03	0	1934.78	1934.81	1	100.00%	788.98
1	2x2	82.92	137.83	647.12	867.87	2.23	55.73%	361.64
1	3x3	45.63	201.34	284.74	531.71	3.64	40.43%	223.63
1	4x4	48.77	138.8	189.02	376.59	5.14	32.11%	163.23

(c) 5721: short-paths-dominated

Table 4.9: Results with different concurrency thresholds (50 case1 clustered queries) time unit: second

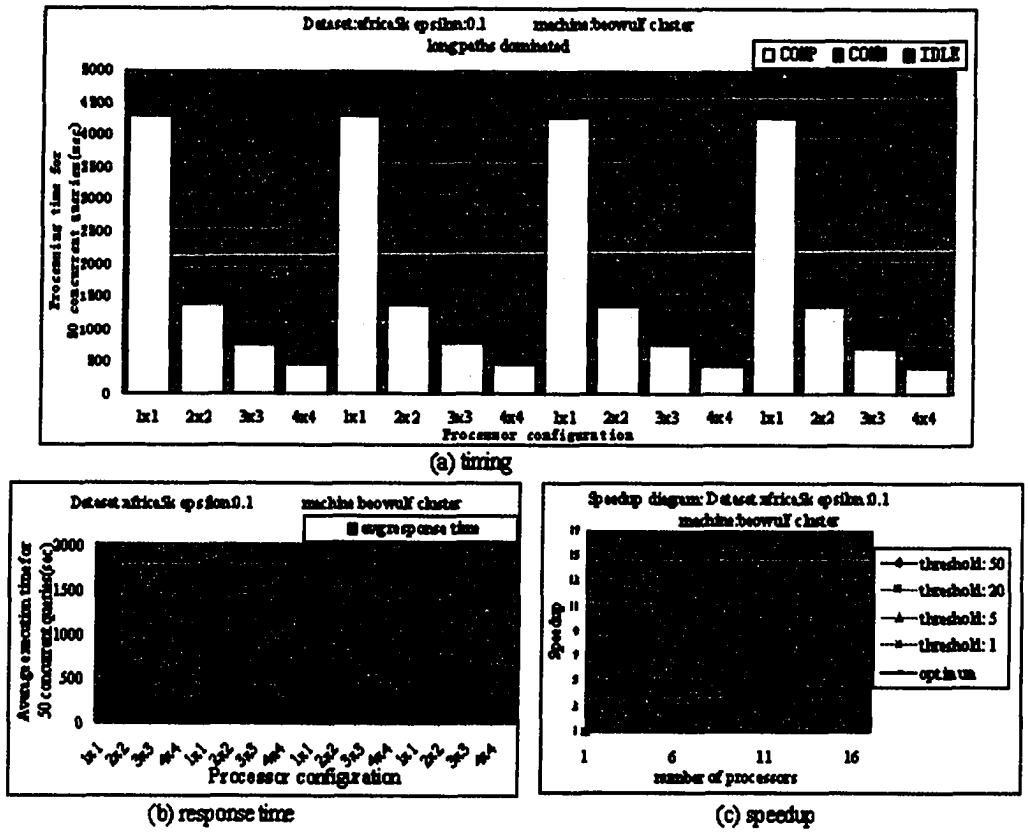


Figure 4.11: Results with different concurrency thresholds (long-paths-dominated, case1)

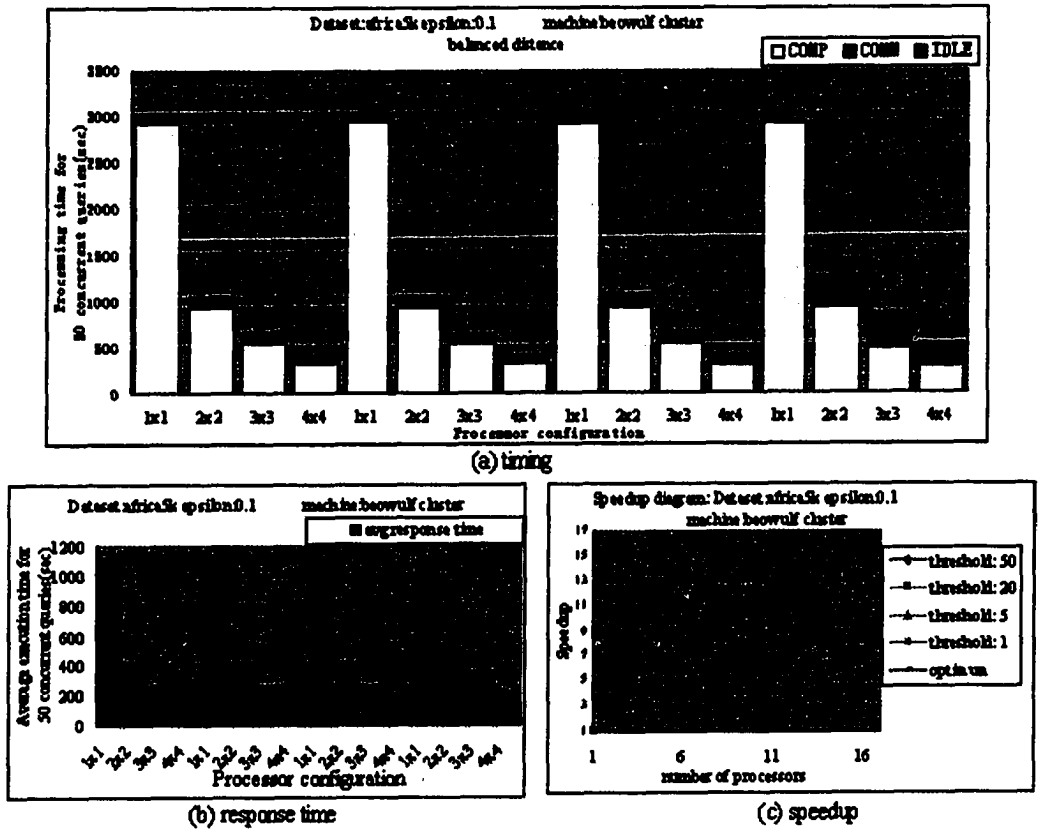


Figure 4.12: Results with different concurrency thresholds (balanced-distance, case1)

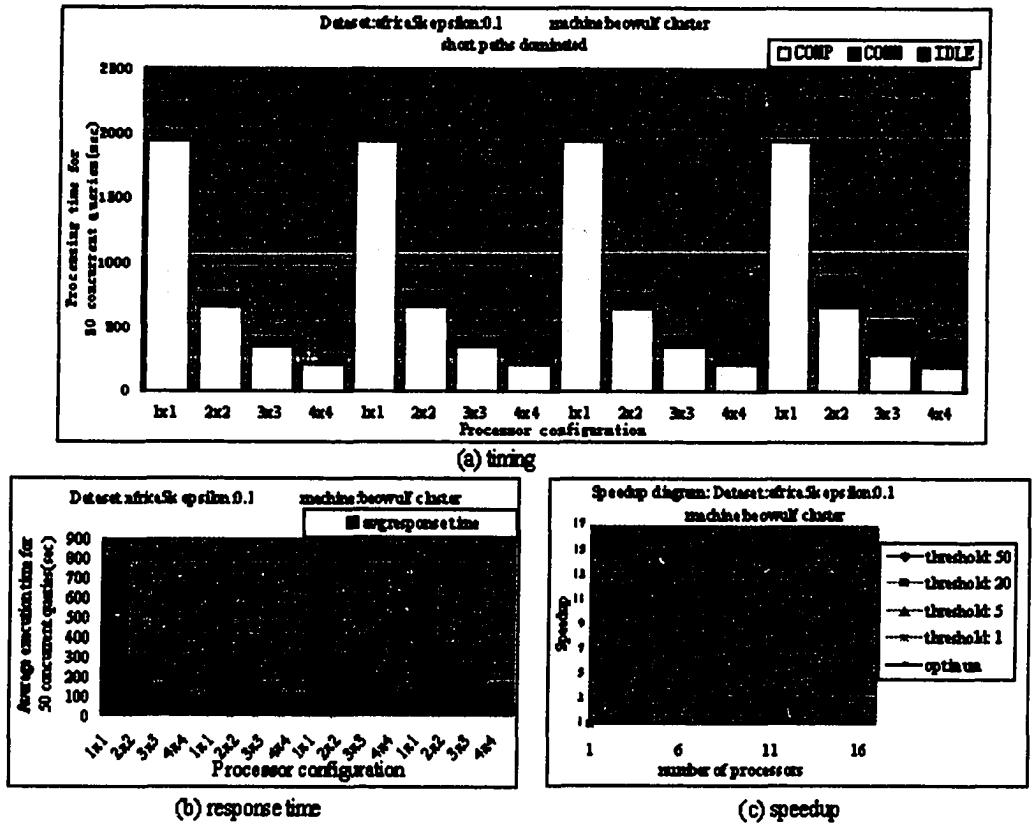


Figure 4.13: Results with different concurrency thresholds (short-paths-dominated, case1)

4.4.4 Clustered query distribution case 2

We used the test sets as shown in Figure 4.8(d),(e),(f) with different concurrency thresholds: 1, 5, 20 and 50. The results are shown in Table 4.10(a), (b), (c) respectively. Figure 4.14 shows the results for long-paths-dominated test set. Figure 4.15 shows the results for balanced-distance test set. Figure 4.16 shows the results for short-paths-dominated test set. Again, we can still observe the features discussed before. When the idle time has been reduced to a small amount, increasing the concurrency threshold will not improve the performance.

4.4.5 Concurrency tests conclusion

The results on the test sets with various query distributions have shown quite similar features. We can conclude that the multi-tasking design is effective to improve the speedup and efficiency by handling concurrent queries at the same time. This performance enhancement comes from the reduction of the processor idle time. Therefore, when the processors have been fully-loaded with the queries (i.e. the idle time is already very small), then increasing the concurrency threshold would not have much effect on the system performance.

The tests on random queries have shown that, using concurrency, we can achieve good performance even with short queries and clustered queries. The performance over evenly distributed queries is better than that over clustered queries, because the clustered queries tend to heavily burden only part of the processors. This can be observed from the portion of idle time when comparing the timing figures of long queries and short queries. The portion of idle time is smaller in the computation of

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	4456.92	4456.93	1	100.00%	2120.29
50	2x2	185.11	8.97	1473.79	1667.87	2.67	66.81%	785.19
50	3x3	115.52	53.34	769.84	938.7	4.75	52.76%	444.84
50	4x4	108.64	14.48	458.34	581.46	7.67	47.91%	279.63
20	1x1	0.01	0	4467.35	4467.36	1	100.00%	2124.77
20	2x2	184.65	13.41	1477.11	1675.16	2.67	66.67%	786.13
20	3x3	115.98	50.99	773.39	940.36	4.75	52.79%	447.93
20	4x4	108.26	13.24	458.97	580.47	7.7	48.10%	279.37
5	1x1	0.01	0	4460.81	4460.82	1	100.00%	2121.66
5	2x2	183.09	30.65	1460.24	1673.98	2.66	66.62%	785.56
5	3x3	115.01	75.6	760.53	951.15	4.69	52.11%	455.01
5	4x4	107.93	21.46	451.9	581.29	7.67	47.96%	282.32
1	1x1	0.03	0	4460.81	4460.84	1	100.00%	2121.31
1	2x2	182.39	208.12	1433	1823.51	2.45	61.16%	855.30
1	3x3	112.45	335.27	708.8	1156.51	3.86	42.86%	566.94
1	4x4	103.9	190.3	416.87	711.07	6.27	39.21%	346.12

(a) 5127: long-paths-dominated

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	2811.84	2811.85	1	100.00%	1327.62
50	2x2	114.88	12.31	892.05	1019.23	2.76	68.97%	461.23
50	3x3	71.64	45.03	470.12	586.79	4.79	53.24%	251.77
50	4x4	72.04	7.35	298.03	377.42	7.45	46.56%	176.10
20	1x1	0.01	0	2806.38	2806.39	1	100.00%	1323.89
20	2x2	113.41	31.1	891.9	1036.41	2.71	67.70%	472.82
20	3x3	70.17	52.6	455.2	577.97	4.86	53.95%	253.93
20	4x4	71.52	10.82	295.62	377.97	7.42	46.41%	177.68
5	1x1	0.01	0	2803.64	2803.65	1	100.00%	1322.70
5	2x2	113.97	64.93	893.37	1072.27	2.61	65.37%	500.75
5	3x3	71.7	82.92	460.87	615.49	4.56	50.61%	275.60
5	4x4	70.72	20.14	288.47	379.33	7.39	46.19%	182.18
1	1x1	0.03	0	2799.57	2799.6	1	100.00%	1321.40
1	2x2	120.59	178.6	917.22	1216.41	2.3	57.54%	574.73
1	3x3	76.33	283.22	461.62	821.17	3.41	37.88%	374.58
1	4x4	70.67	143.42	276.44	490.53	5.71	35.67%	239.07

(b) 5433: balanced-distance

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	1635.86	1635.86	1	100.00%	778.23
50	2x2	76.97	11.1	580.76	668.82	2.45	61.15%	314.16
50	3x3	53.45	14.95	342.57	410.97	3.98	44.23%	203.04
50	4x4	45.05	8.09	186.46	239.6	6.83	42.67%	113.80
20	1x1	0.01	0	1631.89	1631.89	1	100.00%	776.46
20	2x2	75.71	13.97	570.91	660.6	2.47	61.76%	311.66
20	3x3	51.14	26.55	326.73	404.42	4.04	44.83%	199.40
20	4x4	45.53	11.92	189.73	247.19	6.6	41.26%	115.09
5	1x1	0.01	0	1633.96	1633.97	1	100.00%	777.53
5	2x2	75.95	37.46	565.93	679.33	2.41	60.13%	326.47
5	3x3	49.59	75.16	312.96	437.7	3.73	41.48%	225.04
5	4x4	44.16	22.25	182.55	248.96	6.56	41.02%	116.02
1	1x1	0.03	0	1631.93	1631.96	1	100.00%	776.86
1	2x2	76.66	116.81	557.33	750.8	2.17	54.34%	370.37
1	3x3	49.79	233.27	296.2	579.27	2.82	31.30%	302.44
1	4x4	42.36	141.28	166.08	349.7	4.67	29.17%	169.39

(c) 5721: short-paths-dominated

Table 4.10: Results with different concurrency thresholds (50 case2 clustered queries)

time unit: second

long queries than in that of short queries. The performance over long queries is better than that over short queries, because long queries may cross more pages belongs to different processors, hence get more benefits from parallelization. From the above timing figures we can also find that when the concurrency threshold is 5, the idle time becomes very small even in a clustered query distribution. This can be attributed to the SWP partitioning. SWP uses the over-partitioning to let all the processors get involved quickly.

When the number of processor grows, concurrent PSP can let the efficiency degrade slower than the non-concurrent PSP. We know that if the efficiency degrade too fast, which means we may not even get higher speedup when adding more processors into computation. A slower efficiency degradation may suggest that we can still obtain higher speedup when using more processors.

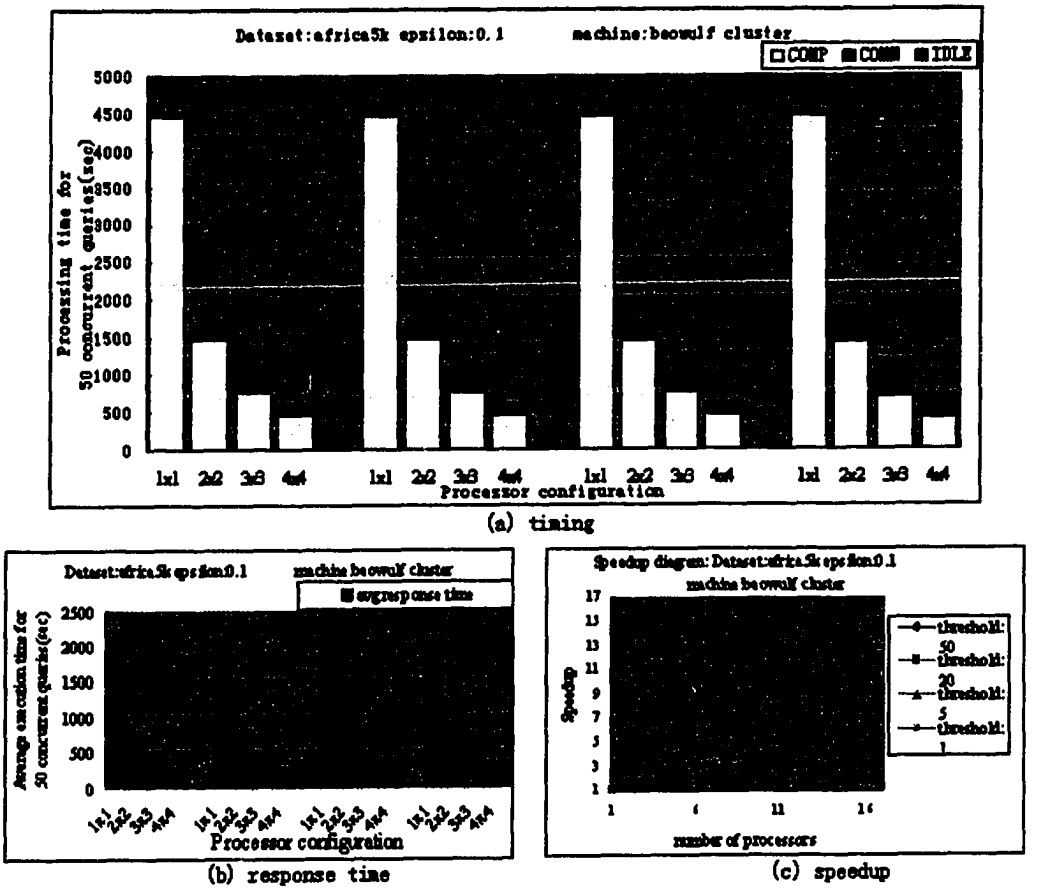


Figure 4.14: Results with different concurrency thresholds (long-paths-dominated, case2)

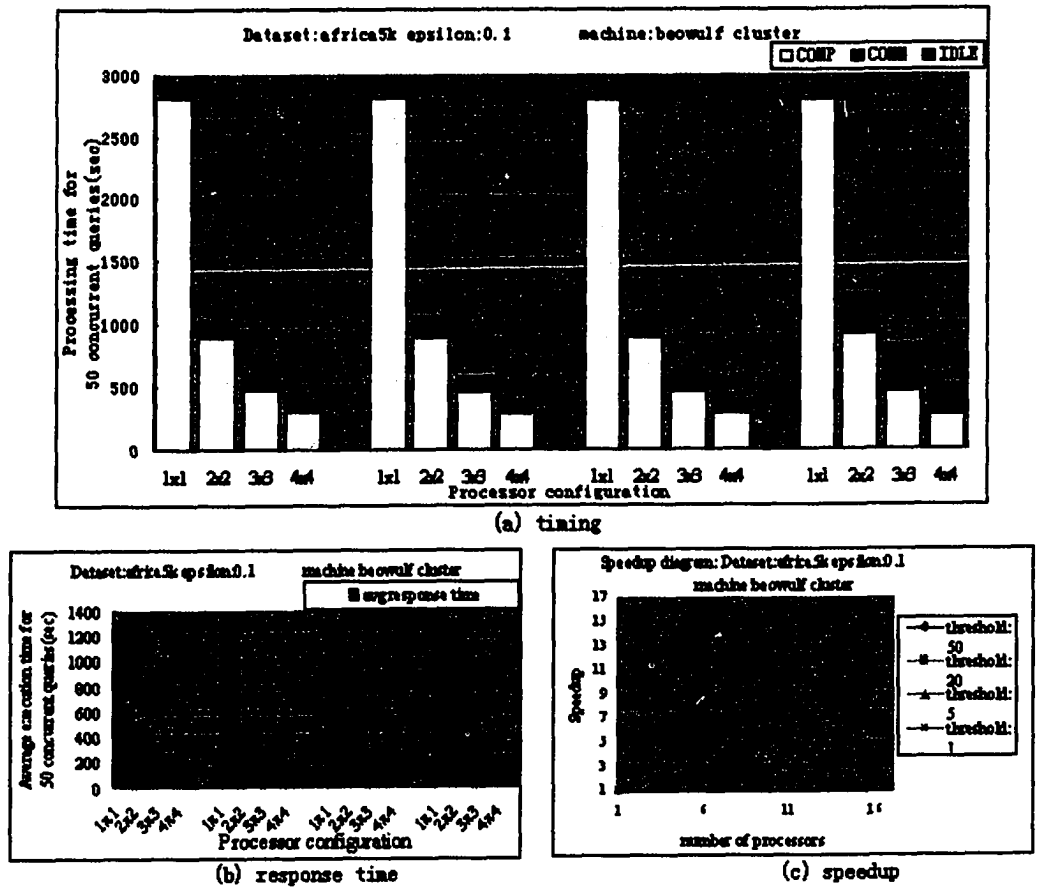


Figure 4.15: Results with different concurrency thresholds (balanced-distance, case2)

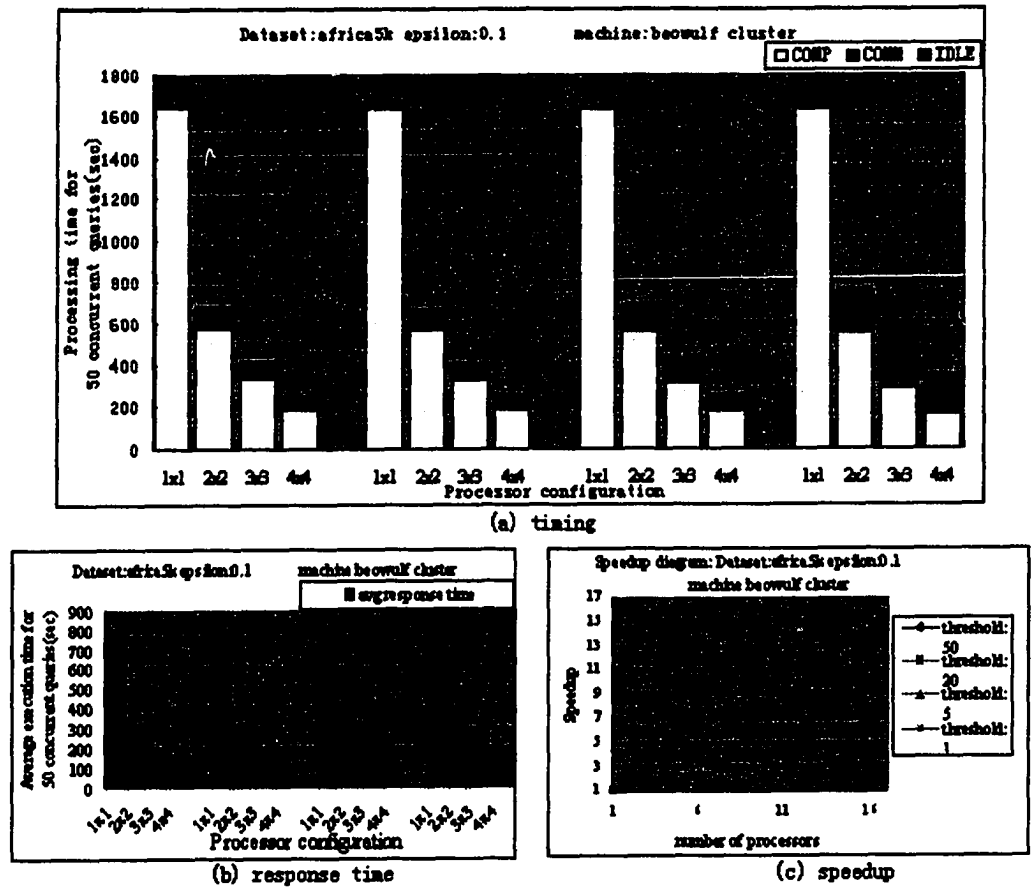


Figure 4.16: Results with different concurrency thresholds (short-paths-dominated, case2)

Chapter 5

Conclusion

The ultimate goal of our research is to compute terrain shortest path more efficiently with allowable length accuracy. We first present the bisector $\epsilon - \beta$-approximation scheme which is a variation of bisector ϵ-approximation scheme. The bisector $\epsilon - \beta$-approximation scheme makes the shortest path computation more efficiently by reducing the number of Steiner points placed on the original TIN graph. This results in a trade-off between the accuracy of the approximation shortest path and the running-time. In our experiments, the application using bisector $\epsilon - \beta$-approximation scheme runs 3-6 times faster than that using bisector ϵ-approximation scheme. However, the length of the output $\epsilon - \beta$-approximated shortest path is within a small factor (maximum 0.28%) of the length of the ϵ-approximated shortest path. To choose the parameter β, one must consider the type of shortest path queries and the data set. We designed a method that helps the user choose the suitable β value which satisfies both the length accuracy and the application running-time.

We also present a partitioning weighting scheme: SWP, which is a small extension of

MFP, and apply it to parallel terrain shortest path computation. SWP determines whether to partition a page according to its weight which roughly represents the computational load of the page. It is suitable for scenarios in which the distribution of Steiner points is different with the distribution of TIN vertices. For example, using the bisector ϵ-approximation scheme, the Steiner points may be heavily clustered in the area which TIN vertices are sparse.

We introduce the idea of concurrent PSP which utilizes the processor idle time to compute concurrent queries simultaneously. We implemented SWP as well as the parallel concurrent bisector $\epsilon - \beta$-approximation shortest path application. From the concurrency experiments, we confirm that concurrent PSP is effective in increasing the speedup and efficiency by reducing the processor idle time. We observed a good performance of our parallel algorithm that yields good speedup and efficiency levels. By using concurrent PSP, the efficiencies are 85.9%, 76.2% and 71.4% for processor 2x2, 3x3, 4x4 respectively on Sunfire; 88.1%, 79.2% and 65.4% for processor 2x2, 3x3, 4x4 respectively on Beowulf. These efficiencies are much higher than that without concurrency (61.0%, 40.3%, and 36.3% for processor 2x2, 3x3, 4x4 respectively on Beowulf). Moreover, we observed fair performance while dealing with clustered queries.

Future work

SWP can also work with other approximation shortest path schemes such as FIXED, INTERVAL scheme, etc. Because of the limit of time, we now only works on one terrain. Experiments can be done in other terrains to verify the performance stability. It would be interesting to study how SWP deals with TIN modifications (for example, the insertion or removal of a vertex).

Bibliography

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

[2] P.K. Agarwal, S. Har-Peled, and M. Karia. Computing approximate shortest paths on convex polytopes. *Algorithmica*, 33:227–242, 2002.

[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 (JACM)*, 44:567–584, 1997.

[4] L. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari. Partitioning planar graphs with costs and weights. In *The 4th Workshop on Algorithm Engineering and Experiments (ALENEX 02)*, pages 98–110, 2002.

[5] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.R. Sack. An epsilon-approximation algorithm for weighted shortest paths on polyhedral surfaces. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 1998.

[6] L. Aleksandrov, A. Maheshwari, and J.R. Sack. Approximation algorithms for geometric shortest path problems. In *the 32nd Annual ACM Symposium on Theory of Computing*, pages 286–295, 2000.

[7] L. Aleksandrov, A. Maheshwari, and J.R. Sack. An improved approximation algorithms for computing geometric shortest paths. In *Foundations of Computation Theory*, pages 246–257, 2003.

[8] L. Aleksandrov, A. Maheshwari, and J.R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM (JACM)*, 52:25–53, January 2005.

[9] D.P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *Jounral of Optimization Theory and Applications*, 88(2):297–320, 1996.

[10] J. Canny and J.H Reif. New lower bound techniques for robot motion planning problems. In *Proceedings of the 28th IEEE Symp. on Foundations of Computer Science*, pages 49–60, 1987.

[11] J. Chen and Y. Han. Shortest paths on a polyhedron. In *Proceedings of 6th ACM Symposium on Computational Geometry*, pages 360–369, 1990. Full version appeared in International Journal of Computational Geometry and Application 6: 127-144, 1996.

[12] Cormen, Leiserson, Rivest, and Stein, editors. *Introduction to Algorithms.* McGraw-Hill, 2nd edition, 2002.

[13] H.N. Djidjev. On the problem of partitioning planar graphs. *SIAM J. on Algebraic and Discrete Methods*, 3:229–240, 1982.

[14] H.N. Djidjev. Partitioning planar graphs with vertex costs: Algorithms and applications. *Algorithmica*, 28(1):51–57, 2000.

[15] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing C-21*, (9):948–960, Sep. 1972.

[16] G. Gallo and S. Pallottino. Shortest paths algorithms. *Annals of Oper. Res.*, (13):3–79, 1988.

[17] H. Guo. Partitioning planar graphs with costs and weights. Master's thesis, Carleton University, 2002.

[18] S. Har-Peled. Approximate shortest paths and geodesic diameters on convex polytopes in three dimensions. In *Discrete and Computational Geometry 21*, pages 217–231, 1999.

[19] S. Har-Peled. Constructing approximate shortest paths maps in three dimensions. *SIAM Journal on computing*, 28:1182–1197, 1999.

[20] R.V. Helgason and D. Stewart. One-to-one shortest path problem: An emperical analysis with the two-tree dijkstra algorithm. *Computational Optimization and Application*, 2:47–75, 1993.

[21] J. Hershberger and S. Suri. Practical methods for approximating shortest paths on a convex polytope in r3. In *6th ACM-SIAM Symposium on Discrete Algorithms*, pages 447–456, 1995.

[22] M. Hribar, V. Taylor, and D. Boyce. Choosing a shortest path algorithm. Technical Report CSE-95-004, Nortwestern University, 1995.

[23] M. Hribar, V. Taylor, and D. Boyce. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *13th Annual conference on Intel Supercomputers User Group*, Albuquerque, NM, 1997.

[24] 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.

[25] M. Hribar, V. Taylor, and D. Boyce. Reducing the idle time of parallel shortest path algorithms. Technical Report CPDC-TR-9803-016, Northwestern University, 1998.

[26] S. Kapoor. Efficient computation of geodesic shortest paths. In *31st ACM STOC*, pages 770–779, 1999.

[27] M. Lanthier, A. Maheshwari, and J.R. Sack. Approximating weighted shortest paths on polyhedral surfaces. *Algorithmica*, 30(4):527–562, 2001.

[28] M. Lanthier, D. Nussbaum, and J.R. Sack. Parallel implementation of geometric shortest path algorithms. *Parallel Computing*, 29:1445–1479, 2003.

[29] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36(2):177–189, 1979.

[30] L.A. Lyusternik. Shortest paths: Variational problems. *Macmillan, New York*, 1964.

[31] G. McDaniel, editor. *IBM Dictionary of Computing.* McGraw-Hill, 1994.

[32] J.S.B. Mitchell, D.M. Mount, and C.H. Papadimitriou. The discrete geodesic problem. *SIAM Journal on Computing*, 16:647–668, August 1987.

[33] J.S.B. Mitchell and C.H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of ACM*, 38:18–73, January 1991.

[34] D. Nussbaum. *Parallel spatial modelling.* PhD thesis, Carleton University, 2001.

[35] L. Polymenakos and D.P. Bertsekas. Parallel shortest path auction algorithms. *Parallel Computing*, 20:1221–1247, 1994.

[36] K.V.S. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13:235–257, 1992.

[37] J.H Reif and Z. Sun. An efficient approximation algorithm for weighted region shortest path problem. In *4th workshop on Algorithmic Foundations of Robotics(WAFR2000)*, pages 191–203, 2000.

[38] J.R. Sack and J. Urrutia, editors. *Handbook of Computational Geometry.* 2000.

[39] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. *SIAM Journal on Computing*, 15:193–215, 1986.

[40] J.L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.

[41] K.R. Varadarajan and P.K. Agarwal. Approximating shortest paths on nonconvex polyhedron. In *38th IEEE FOCS*, 1997. Full version in SIAM Journal on Computing 30(4):1321-1340, 2000.