EUCLIDEAN DISTANCE TRANSFORM OF 3D IMAGES ON L.A.R.P.B.S. MODEL

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

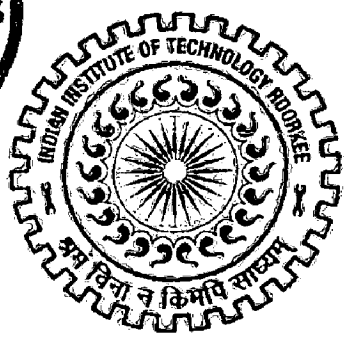
MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

PIYUSH DHAR DIWAN



DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)

JUNE, 2007

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled **"EUCLIDEAN DISTANCE TRANSFORM OF 3D IMAGES ON L.R.B.P.B.S. MODEL"** towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology** in **Computer Science Engineering** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India) is an authentic record of my own work carried out during the period from July 2006 to June 2007, under the guidance of **Dr. Padam Kumar, Professor, Department of Electronics and Computer Engineering, IIT Roorkee.**

I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.

Date: 08-06-2007
Place: Roorkee

PDiwan
PIYUSH DHAR DIWAN

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 08-06-2007
Place: Roorkee

PKumar
Dr. Padam Kumar
Professor
Department of Electronics and Computer Engineering
IIT Roorkee – 247 667

ACKNOWLEDGEMENTS

I would like to extend my heartfelt gratitude to my guide **Dr. Padam Kumar**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for his able guidance, regular source of encouragement and assistance throughout this dissertation work. It is his vision and insight that inspired me to carry out my dissertation in the upcoming field of Parallel Computing. I would state that the dissertation work would not have been in the present shape without his umpteen guidance and I consider myself fortunate to have done my dissertation under him.

I also extend my sincere thanks to **Dr. D. K. Mehra**, Professor and Head of the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee for providing facilities for the work.

I also wish to thank all my friends for their valuable suggestions and timely help.

Finally, I would like to say that I am indebted to my parents for everything that they have given to me. I thank them for the sacrifices they made so that I could grow up in a learning environment. They have always stood by me in everything I have done, providing constant support, encouragement and love.

PIYUSH DHAR DIWAN

ABSTRACT

Euclidean Distance Transform (EDT) of 2-D and 3-D images is one of the useful tools in various image processing algorithms. It is one of the basic operations in image processing and computer vision fields and essentially used in expanding, shrinking, thinning, segmentation, clustering and computing of images, object reconstruction, etc. It converts a binary image consisting of black and white pixels to a representation where each pixel has the Euclidean distance of the nearest black pixel. Many sequential and parallel algorithms have been developed for Euclidean Distance Transform computation of 2-D and 3-D images on various computational platforms. The objective of this dissertation work is to develop a time-optimal and scalable algorithm for EDT computation of 3-D images.

In this dissertation work, an efficient and scalable parallel algorithm has been designed and implemented for computing EDT of **3-D** images, on **L**inear **A**rray with **R**econfigurable **P**ipelined **B**us **S**ystem (**LARPBS**) multiprocessor model, which is a recently proposed architecture based on optical buses. This work is the extension of the algorithm for 2-D EDT computation on LARPBS architecture which is given by *Chen, Pan and Xu.*

The algorithm computes the EDT of a 3-D image represented by N × N × N binary matrix in $\mathbf{O(N^2 \log N /(a(n) * b(n)))}$ time using $\mathbf{N^2 * a(N) * b(N)}$ processors where a(N) and b(N) are the parameters and their values can be selected between 1 and N. By selecting different values for a(N) and b(N), time complexity and number of processors required can be altered which makes the algorithm more flexible and scalable. This algorithm has been implemented and tested on the multiprocessor cluster available at the Institute Computer Center. The performance has been analyzed and compared with other EDT algorithms given on various parallel computing platforms.

CONTENTS

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

The **Euclidean Distance Transform** computation of binary images is a basic operation in image processing and computer vision fields. Some other basic operations of image processing like image expansion, shrinking, thinning, segmentation, clustering, object reconstruction, etc require EDT of binary images as an intermediate tool [1]. In a 3-dimensional binary image array, consisting of 1s and 0s, 1-voxels are referred as ***foreground*** or ***black*** voxels while 0-voxels are referred as ***background*** or ***white*** voxels. Often, we are interested in the shape and position of the black voxels relative to each other. The extraction of such information from a binary image can be simplified considerably by using a number of computational techniques. Some of the most important ones include the Medial Axis Transform (MAT) introduced by Blum [2] and the Distance Transform (DT) introduced by Rosenfeld and Pfaltz [3], [4].

The **E**uclidean **D**istance **T**ransform of a binary image array, consisting of 1 and 0 voxels, transforms it to another array where each voxel has a value or coordinates that represents the distance or location to the nearest 1 voxel [6]. A great deal of work has been done on EDT computation techniques of 2-D as well as 3-D images. A Sequential *Brute Force* exhaustive approach for EDT computation of a **3-D** binary array would have inherent time complexity of *O (n^6)* on a sequential, uni-processor machine. To optimize this time, many parallel algorithms have been given so far on various multiprocessor architectures with different time complexities and different number of processors used.

The algorithm derived in this dissertation is an extension to the algorithm for 2-D EDT computation on LARPBS model and it is given by *Chen, Pan and Xu* [1]. This computes EDT of an N × N × N binary image array, with respect to *black* voxels on LARPBS model. LARPBS is a multiprocessor model consisting of Processor Arrays with Reconfigurable Pipelined Optical Bus System. It has become the focus of interest for implementation of many efficient parallel algorithms, since it limits the communication latency and provides concurrent message transmission through the bus system [7].

1.2 STATEMENT OF THE PROBLEM

The Euclidean Distance Transform of a 3-D image array, consisting of black and white voxels, transforms it to another array where each voxel has a value or coordinates that represents the distance or location to the nearest black voxel. The aim of this dissertation work is to design, implement and analyze the performance of an efficient and scalable parallel algorithm for the computation of Euclidean Distance Transform of 3-Dimensional binary images on LARPBS multiprocessor architecture.

The work towards the solution of this problem can be divided as following:

1. Design of EDT computation algorithm of 3-D binary images on LARPBS model as an extension of the algorithm for 2-D binary images.
2. Implementation of the above algorithm on the available multiprocessor cluster.
3. Time benefit analysis of the proposed algorithm on the basis of complexity and scalability.
4. Comparison of relative performances of this parallel algorithm with other existing algorithms for EDT computation.

1.3 ORGANISATION OF THE DISSERTATION

This report is divided into six chapters including this introductory chapter. The rest of this thesis is organized as follows.

Chapter 2 gives the background about the EDT algorithms and LARPBS model. It also discusses about the related work, research gaps and motivation.

Chapter 3 discusses the proposed algorithm for 3-D EDT computation and also its complexity and scalability analysis.

Chapter 4 provides implementation details like hardware and software specification for the proposed algorithm on the multiprocessor cluster architecture. It also includes the performance evaluation and comparison metrics.

Chapter 5 provides the implementation results and evaluates performance of the proposed algorithm. It also provides a comparative analysis with other algorithms.

Chapter 6 concludes the dissertation and gives some suggestions for future work.

CHAPTER 2
BACKGROUND

2.1 BASIC INFORMATION ABOUT EUCLIDEAN DISTANCE TRANSFORM

A three dimensional binary image is represented by a 3-D binary array in which the ***black*** voxels are represented by 1s and the ***white*** voxels are represented by 0s. The *distance transform* of binary images was first introduced by *Rosenfeld and Pfalz* [3]. It has been very useful metric information for binary images in various image processing algorithms and application. EDT of a 3-D image is defined as the transform of original array in which every element has the value equals to the distance of the corresponding voxel in the original array from its nearest 1-voxel.

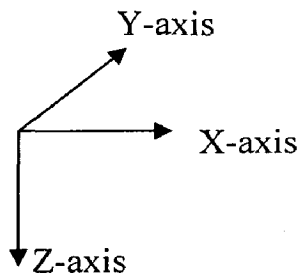


Figure 2.1. Direction of X, Y and Z- axis

Let us consider that an N × N × N binary image is represented by a 3-D matrix A where,

$A = \{(i, j, k): a(i, j, k) = 0 \text{ or } 1\}$

Also let B is a set of triples (x, y, z) containing the positions of all the 1-voxels in A.

The ***Euclidean Distance*** of any voxel a(i, j, k) is defined by [7]:

$$d^2_{i,j,k} = \min_{(x, y, z) \in B} \{ (i-x)^2 + (j-y)^2 + (k-z)^2 \}$$

For all i, j, k = 0.....N-1.

The nearest foreground voxel from any voxel a(i, j, k) is denoted by F(i, j, k) and the distance between these two voxels $d_{i,j,k}$ is the EDT of the voxel a(i, j, k). It is clear hear that the Euclidean Distance of all voxels in set B, i.e. all 1-voxels in the given binary image array will be ***zero***.

Let's see an example of 2-D, 4 × 4 binary image array A and its EDT array F:

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \quad F = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

Since every voxel in a 2-D or 3-D image array posseses its own Euclidean Distance, these values can be determined parallaly by partitioning the whole array into smaller subarrays. Hence, to exploit the parallelizability of 2-D or 3-D EDT computation, many parallel algorithms have been suggested on different multiprocessor architectures which varies in their computational time complexities and number of processors required.

Before going into the algorithm, let us put some focus on the basic properties of EDT which will be used in the algorithm.

Lemma 1. *[8],[9] Let A = (i, j, k) and B =(p, j, k) be two voxels on the same column and also same plane with p < i. Let F(i, j, k)= (x, y, z) and F(p, j, k)=(u, v, w), then u ≤ x namely F(i , j, k) is below or on the same row as F(p, j, k).*
Lemma 1 is extended to more general cases in Lemma 2.

Lemma 2. *[8],[9] Let $(i_1, j, k), (i_2, j, k) \ldots\ldots\ldots\ldots (i_r, j, k)$ be voxels on the same column and same plane and $i_1 < i_2 < \ldots < i_r$. Suppose $F(i_t, j, k) = (x_t, y_t, z_t)$ for t = 1, 2.....r then $x_1 \leq x_2 \leq \ldots \leq x_r$.*

Note: If we denote the nearest 1-voxel of a(i, j ,k) in the i^{th} row as RF(i, j, k), then we have the following lemma.

Lemma 3.*[1] If we denote the distance from voxel (i, j, k) to its nearest foreground pixel as* d_{ij}, *and denote the distance between voxels (i, j, k) and (f, g, h) as D[(i, j, k),(f, g, h)] then we have* $d_{ij} = min_{0 \le p \le N-1} D[(i, j, k), RF(p, j, k)]$ *namely, F(i, j, k) can be selected from the set RF(p, j, k) where* $0 \le p \le N-1$.

Lemma 4. *[1] Let* $(i_1, j, k), (i_2, j, k) \ldots\ldots\ldots\ldots (i_r, j, k)$ *be voxels on the* j^{th} *column,* k^{th} *plane, and* $i_1 < i_2 < \ldots < i_r$. *Let* $F(i_t, j, k) = RF(g_t, j, k)$ *,where* $t = 1,2, \ldots\ldots r$. *Then,*

1. $g_1 \le g2 \le \ldots \le g_r$ *and*

2. For every $i \in (i_{t-1}, i_t)$, *F (i, j, k) can be found in RF* (g_{t-1}, j, k), *RF* $(g_{t-1}+1, j, k) \ldots\ldots RF (g_t, j, k)$.

Note: Lemma 1 to 4 will be used in 2-D EDT computation within a single 2-D plane. While Lemma 5 and 6 will be used in 3-D EDT computaion of complete 3-D array.

Lemma 5. *[6]* Let Q(x_Q, y_Q, z_1) and P(x_P, y_P, z_2) be 2 voxels with same X and Y cordinates and $z_1 \le z_2$ i.e. Q is above P in the direction of Z-axis. Let F_Q and F_P be the nearest 1-voxels of Q and P with coordinates (x_{FQ}, y_{FQ}, a) and (x_{FP}, y_{FP}, b) respectively. Then $b \ge a$ i.e. F_P is below F_Q.

Lemma 6. *[6]* Let Q(x_Q, y_Q, z_1) and P(x_P, y_P, z_2) be 2 voxels with same X and Y cordinates and $z_1 \le z_2$ i.e. Q is above P in the direction of Z-axis. Let $F_Q(T_r)$ be the nearest 1-voxel of voxel Q on plane r. Then,

If $F_Q = F_Q(T_r)$, $0 \le r \le N-1$, then $F_P \in F_Q(T_w)$ where $F_Q(T_w) = F_P(T_w)$ for $r \le w \le N-1$.

2.2 "LINEAR ARRAY WITH RECONFIGURABLE PIPELINED BUS SYSTEM" MULTIPROCESSOR ARCHITECTURE

Processor array with reconfigurable pipelined buses is a recently proposed architecture for efficient computations of various parallel algorithms. These systems allow concurrent transmission of multiple messages across the bus in a pipelined fashion and the bus can be reconfigured dynamically under program control to support different algorithmic requirements [7]. Figure 2.2 illustrates the optical bus system in an *n*- processor system.

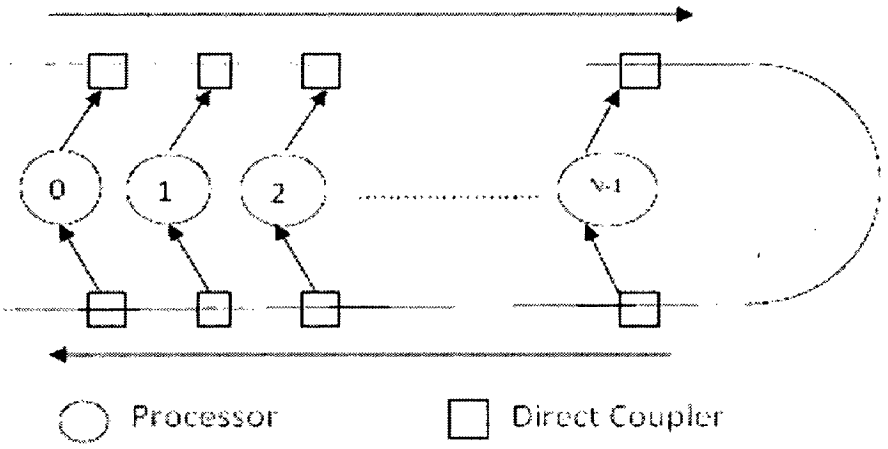


Figure 2.2. A linear optical bus system of *n* processors.

In multiprocessor systems the communication diameter i.e. the maximum distance between processors grows with the size of the system and the interconnection network used for inter-processor communication [10]. Hence, increasing the size of these networks may not contribute in optimization of the time complexities of the parallel algorithms designed for them. Processor arrays with reconfigurable buses overcome this bottleneck by partitioning the bus into many segments, and all segments can be reconfigured as a single global bus [11]. The pipelined optical bus system uses optical waveguides instead of electrical signals to transfer messages among processors [7]. Fiber optics communications offer a combination of high bandwidth, low error probability, and gigabit transmission capacity. This design integrates the advantages of both optical transmission and electronic computation [7]. The detailed Bus organization of LARPBS system is illustrated in Figure 2.3.

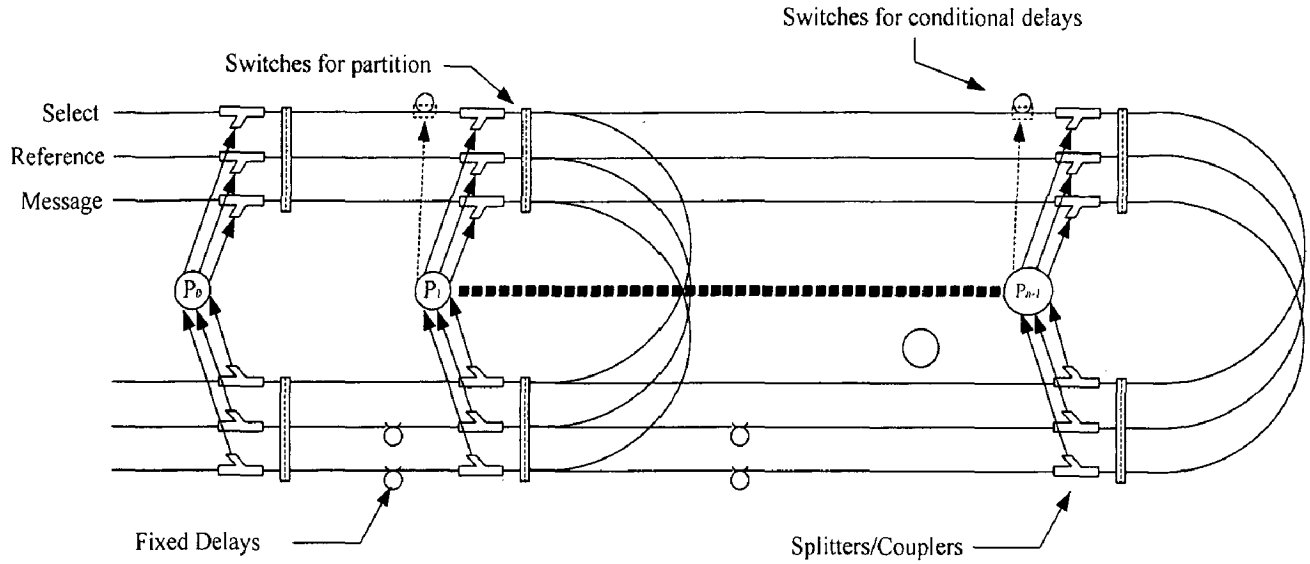


Figure 2.3. LARPBS - Linear Array with a Reconfigurable Pipelined Bus [10].

An optical bus has two other important characteristics: *unidirectional propagation and predictable propagation delay.* These advantages of using waveguides enable synchronized concurrent accesses of an optical bus in a pipelined fashion [10]. Such systems support massive simultaneous communications and so they are appropriate for communication intensive operations such as broadcasting, one-to-one communication, multicasting, compression, split, and many other irregular communication patterns [10].

Many parallel algorithms have been proposed on LARPBS model to solve problems such as sorting, selecting, matrix computation, and computational geometry etc [11]. Even many parallel algorithms on PRAM can be transformed into LARBPS algorithms using the results for PRAM simulation on the LARPBS model [11].

In terms of architecture, **L**inear **A**rray with **R**econfigurable **P**ipelined **B**us **S**ystem (LARPBS) is an array of N processors P_1, P_2 ..... P_N connected by an *optically pipelined* bus. A *bus cycle* is the end-to-end propagation delay on the bus. The time complexity of any algorithm is determined in terms of time steps, when a single time step comprises of one bus cycle and one local computation step [10]. Following operations, on the LARPBS model will be used in the algorithm [11]:

- ***Broadcasting*:** A source processor P_i sends a message to all the other N-1 processors P_1, P_2, ..., P_{i-1}, P_{i+1},....., P_N. All the messages are sent simultaneously.
- ***Multicasting*:** A source processor sends a message to a subset of the N processors.
- ***One-to-One Communication*:** A subset of processors P_{i1}, P_{i2} , ... P_{im} are senders and another subset of processors P_{k1}, P_{k2}, ... P_{km} are the receivers. Processor P_{ij} sends a message to processor P_{kj} for $1 \leq j \leq m$. All these messages are sent simultaneously.
- ***Multiple Multicasting*:** There are g disjoint groups of receiving processors, $G_k = \{P_{jk,1}, P_{jk,2} .....\}$, $1 \leq k \leq g$. Also, there are g senders P_{i1}, P_{i2}.....P_{ig}. Processor P_{ik} broadcasts a message to all the processors in G_k, for $1 \leq k \leq g$.
- ***Integer Summation*:** Each processor P_i holds an integer value I_i of bounded magnitude and precision. This operation computes the sum of these integer values and move the sum to the first processor.

Lemma 2.1 *[14] The minimum value of* n *data items can be computed on the LARPBS in O(1) time by using* n *processors if each item is of bounded magnitude and precision.*

Lemma 2.2 *[15] One-to-one communication, broadcasting, multicasting, multiple multicasting, ordered compression, binary prefix sum and summation of integers of bounded magnitude and precision can be done in O(1) time on the LARPBS model.*

2.3 LITERATURE SURVEY AND PREVIOUS WORK

Parallel algorithms for EDT computation have been one of the focuses of interest for researchers in recent years. Several parallel algorithms have been given to compute EDT of 2-D and 3-D images so far. These algorithms tend to divide the overall computation into various parallel subtasks. Different algorithm uses different number of processor to execute these subtasks in parallel and then by gathering the results from all the processors, the actual EDT of given image array is computed. The implementations of such algorithms are based upon various multi-processor architectures like PRAM etc. Let us put some focus on the work done so far on various algorithms for EDT computation.

Yamada [16] was the first to propose an algorithm to compute the exact EDT of an $N \times N$ binary image. The running time of his algorithm was $O(N^3)$. Later, Kolountzakis and Kutulakos [17] proposed an $O(N^2 \log N)$ time algorithm. Chen and Cheung [8] presented an optimal $\Theta(N^2)$ time sequential algorithm for this problem. Since it is desirable to compute the distance transform even faster for many real-time applications, several parallel algorithms have been developed for this problem on different parallel architectures. Fujiwara *et al.* [18] presented two work-optimal algorithms with running times $O(\log N)$ on an $N^2/\log N$ processor EREW PRAM and in $O(\log N/\log \log N)$ time on an $(N^2 \log \log N)/\log N$ processor Common CRCW PRAM.

With the increasing prevalence of 3-D voxel images, it is useful to consider the distance transform of a 3-D digital image array. Saito and Toriwaki [19] presented several EDT algorithms based on the scan approach for an n-dimensional image array. For the 3-D EDT problem, Toriwaki's EDT algorithm takes $O(N^4)$ time complexity. Lee *et al.* [6] have also presented the algorithm for 3-D EDT computation with $O(\log^2 N)$ time complexity on EREW PRAM model. The previous best algorithm for computing the EDT on the LARPBS is by Pan *et al.* [20]. Their algorithm runs either in $O(\log N \log \log N)$ time on an N^2-processor LARPBS or in $O(\log \log N)$ time on an N^3-processor LARPBS.

A summery of the parallel algorithms for 2-D and 3-D EDT computation given so far on different architectures is listed in Table 2.1.

Table 2.1

Summery of Parallel Algorithms for EDT computation

Algorithm	Time Complexity	Number of Processors	Computing Architecture
Chen and Pan	O(log n log log n)	$n^2/(\log \log n)$	LARPBS
Datta et al.	O(log log n)	$O(n^{2+\epsilon})$	LARPBS
Pan et al.	O(log log n)	n^3	LARPBS
Pan et al.	O(log n log log n)	n^2	LARPBS
Datta et al.	O(1)	$O(n^3)$	REMESH
Pan and Li	O(1)	$O(n^4)$	REMESH
Datta et al.	$O((\log \log n)^2)$	$O(n^{2+\epsilon} / (\log \log n))$	PRAM CRCW
Hayashi et al.	O(log log n)	$O(n^2 / (\log \log n))$	PRAM CRCW
A. Fujiwara et al.	O(log n / log log n)	$n^2 \log \log n / \log n$	PRAM CRCW
Pavel and Akl	O(log n)	n^2	PRAM EREW
Lee at al.	$O(\log^2 n)$	n^3	PRAM EREW
Chen and Chuang	$O(n^2/p + n \log n)$	p	PRAM EREW
Chen and Chuang	O(n log n)	n/ log n	PRAM EREW
A. Fujiwara et al.	O(log n)	$n^2/\log n$	PRAM EREW

Now let us move our focus towards the work done so far on Processor Array architectures with reconfigurable optical buses like LARPBS model. During the past decade, several optical bus parallel models have been proposed, together with a suite of basic and advanced algorithms. The *Linear Array with a Reconfigurable Pipelined Bus System* (LARPBS) is one of the recently proposed multiprocessor architectures based on optical buses.

The Linear Array with Reconfigurable Pipelined Bus System (LARPBS) was first published in 1996 [21]. This is one of 10 distinct fiber-based optical bus models that appeared between 1990 and 1998 [21]. Of the ten models, LARPBS appears the most popular based upon not only the number of publications that strongly relate to this model but also the extent of algorithm design, model extension and derived models from LARPBS [21].

The related work on practical implementations of Optical Buses includes feasibility study of power budget and scalability of it with multiple processor systems [10]. The majority of later work on LARPBS mostly concern algorithm design and model refinements.

The main motivation of the development and implementation of optical interconnections is to overcome the bottlenecks that electrical data buses produce due to their relatively low bandwidth [10]. Optical buses allow high bandwidth and pipelined message transmission facilities and these have been frequently used for design and performance analysis of fast parallel algorithms for many different problems in recent years.

2.4 RESEARCH GAPS AND MOTIVATION

We have seen that a great deal of work has been already done on the algorithms for EDT computation. Most of the available algorithms for EDT computation have been designed and implemented for 2-D EDT computation on various multiprocessor platforms like EREW, CRCW PRAM etc. But only a few other algorithms discuss the parallel computation approach for 3-D EDT computation on these architectures. These algorithms vary in their time complexities, number of processors used and the scalability factor. Most of them are time optimal but not enough scalable for higher dimensions and different computational architecture. They have been designed for specific architecture and posses fixed time complexities. This lack of scalability is one of the research issues.

Another major issue in the specified architectures is the Inter-processor Communication overhead. Several strategies have been suggested for reducing this overhead. Processor Arrays with Optical Buses is one of the recent architectures which tend to optimize the inter-processor communication. LARPBS model is based on this architecture. This model provides high speed data transmission through reconfigurable fiber optics buses. The algorithm given by *Chen et al.* [1] discusses a parallel approach for 2-D EDT computation on LARPBS model. This algorithm is scalable enough and resolves the communication overhead issues well, but there is no further discussion available for 3-D EDT computation either in this literature or in its related work. Also the algorithms which are designed for 3-D EDT computation, none of them are based upon LARPBS architecture.

Above discussion shows that there is an empty space for an efficient and scalable 3-D EDT parallel algorithm for of 3-D binary images which would possibly overcome the above issues. This space has derived the motivation for extending the 2-D EDT parallel algorithm on LARPBS model for 3-D images as well.

CHAPTER 3
PROPOSED ALGORITHM

3.1 BRIEF DESCRIPTION OF THE ALGORITHM

This parallel algortihm can be broadly devided into 2 phases:

1. **Phase I: Plane Phase**

   In this phase Euclidean Distance of every voxel on its own plane is computed using 2-D EDT algorithm for LARPBS model [1]. Input to this phase is a 3-D binary array which is stored at the participating processors. This algorithm runs N times in parallel in this phase for N 2-D arrays of the whole 3-D array. This phase uses total $n^2*a(n)*b(n)$ number of processors where $1 \leq a(n) \leq n$ and $1 \leq b(n) \leq n$. These processors are divided into n sets of $n*a(n)*b(n)$ processors. Every set computes the 2-D EDT for each plane. This phase consists of 2 subphases which are:

   a. Row major Phase: Every 2-D plane array is scanned Row wise parallely by group of processors. Each Row is further divided into groups and for each voxel in each group, their Row-wise closest 1-voxel i.e. RF(i, j, k) is determined parallely by processor assigned to the group.

   b. Column major phase: Every 2-D plane array is scanned Column wise parallely by group of processors. Some row indices are selected to partition every column. Then using the results obtained in Row major phase with Lemma 3 and 4, the Euclidean Distance for each voxel, on the same plane is computed.

2. **Phase II: Vertical Phase**

   After having calculated the Euclidean Distance on the same plane for each voxel, this phase integrates the results of Phase I for each plane with other plane's results to compute the actual EDT for each voxel in the complete 3-D array. This is done by assigning N^2 vertical columns parallely in the 3-D array to N^2 group of processors. Every group of processors is further subdivided into subgroups. These subgroups calculates the EDT of the voxels present in the veritcal column.

3.2 THE PARALLEL ALGORITHM FOR 3-D EDT COMPUTATION

Computaional Plateform: LARPBS system.

Initial Input: A 3-D binary martix of order N × N × N consisting 1s and 0s.

Final Output: A 3-D matrix containing Euclidean Distance Transform of corresponding Input matrix.

Phase I: Plane Phase

Row Major Phase

For ***n*** sets of *n*a(n)*b(n)* processors having k = 0 to n-1

Parbegin

For n sets of *a(n)*b(n)* processors having i = 0 to n-1

Parbegin

1. Divide *n* voxels of each row into *a(n)*b(n)* groups. One processor contains group of q = *n/(a(n)*b(n))* voxels.
2. Find leftmost and rightmost 1-voxel in each row. Let them be ***lf*** and ***rf*** respectively.
3. All voxels left to *lf* will have RF(i,j,k) = lf and similerly all the voxels right to *rf* will have RF(i,j,k) = *rf*.
4. Let t^{th} group of voxels is contained by processor PE(t) and its leftmost 1-voxel is lf(t) and lb(t) is number of 0-voxel at the left of lf(t). Similerly its rightmost 1-voxel is rf(t) and rb(t) is number of 0-voxels at the right of rf(t).

   lb(t) = lf(t) - t*q +1;

   rb(t) = (t+1)*q- rf(t);

   Every Processor collects lf(t), lb(t), rf(t), rb(t) values and sends lf(t) and lb(t) to PE(t-1) and gets rf(t-1) and rb(t-1) from PE(t-1). Similerly sends rf(t) and rb(t) to PE(t+1) and gets lf(t+1) and lb(t+1) from PE(t+1).

   Note: All these messages can be comuunicated simultaneously without any significant delay in LARPBS model.

5. If lb(t) < rb(t-1) then

   For all voxels left to lf(t) at PE(t), set RF value to voxel [i, lf(t), k].

   Else

   For initial [lb(t)-rb(t-1)]/2 voxels at PE(t), set RF value as [i,rb(t-1),k] and for rest voxels at the left of lf(t), set RW value as [i, lf(t),k]. Similerly set RF values for the voxels rigth to rf(t) at PE(t).

6. For all other voxels between lf(t) and rf(t) at PE(t), set their RF value according to their closeness from adjecent left and right foreground voxels.

Parend.

Parend.

Column Major Phase

For ***n*** sets of *n*a(n)*b(n)* processors having k = 0 to n-1

Parbegin

For n sets of *a(n)*b(n)* processors having j = 0 to n-1

Parbegin

1. Divide *a(n)*b(n)* processors into b(n) gropus of a(n) proecessors per group. $PE_j(p,q)$ denotes q^{th} processor in p^{th} group in j^{th} column for j = 0 to N-1, p= 0 to b(n)-1,q = 0 to a(n)-1. Every such processor contains n/a(n) number of processors.
2. Send voxel (t*a(n)+l, j,k) and its RF value, to processor $PE_j(0,l)$ where t= 0 to n/a(n)-1 and l=0 to a(n)-1.
3. Broadcast n/a(n) RF values stored at $PE_j(0,l)$,l=0....a(n)-1 to PEj(r,l),r=0....b(n)-1.
4. Divide voxels in the j^{th} column into *b(n)* groups. Let g = n/b(n). Then select b(n) pivot voxels to compute their *F* and *d* values. First, send voxel (tg, j, k) to $PE_j(t,0)$, t= 0...b(n)-1.
5. For t=0 .....b(n)-1

   Broadcast voxels (tg, j, k) stored at $PE_j(t,0)$ to PEj(t,s), s = 0 ..... a(n)-1.

6. For t = 0 ..... b(n)-1

   For l = 0 ..... a(n)-1

   Compute the Distance from voxel (tg, j, k) to *n/a(n)* RF values stored in $PE_j(t, l)$ and find the minimum which is called *Local Minimum.*

7. Now divide a(n) processors $PE_j(t, s)$,s =0 to a(n), into smaller groups, to find the minimum from a(n) local minima obtained in previous step and store them $PE_j(t, 0)$. These values are the F and d values for voxels (tg, j, k).

   Let x_g be the row index of F(tg, j, k) and nnumber of RF values corresponding to the voxels in t^{th} group be $L^t = x_{(t+1)g} - x_g$.

8. Now repartition the j^{th} group of a(n)*b(n) processors in to b(n) groups where t^{th} group will have $[L^t*a(n)*b(n)]/n$ processors. Here

9. $\sum_{t=0}^{b(n)-1} L^t*a(n)*b(n)/n \leq a(n) * b(n)$ since $\sum_{t=0}^{b(n)-1} L^t \leq n$. Further partition these $[L^t*a(n)*b(n)]/n$ processors into b(n) subgroups of $[L^t*a(n)]/n$ processors which will keep the voxels and their RF value of t^{th} group of voxels in j^{th} column. Here q^{th} processor in p^{th} subgroup and t^{th} group is denoted by $PE_{jt}(p, q)$.

10. Distribute RF values to $PE_{jt}(0,l)$, l= 0 to $[L^t*a(n)]/n - 1$. Each processor will have n/a(n) RF values.

11. For l= 0 to $[L^t*a(n)]/n - 1$

    Broadcast n/a(n) RF values from $PE_{jt}(0,l)$ to $PE_{jt}(p,l)$, p= 0 ....b(n)-1.

12. Let g= n/b(n) and $h=n/[b(n)]^2$. Divide these g voxels into b(n) subgroups and get the b(n) pivot voxels to compute their F and d values. Send pivot voxel (tg + ph, j, k) to processor $PE_{jt}(p,0)$,p = 0 ..... b(n)-1.

13. For p = 0 .....b(n)-1

    Broadcast the pivot pixel (tg + ph, j, k) stored at $PE_{jt}(p,0)$ to $PE_{jt}(p,s)$, s = 0...a(n)-1.

14. For p = 0 .....b(n) - 1

    For s = 0 .....a(n) - 1

    Compute the distance between pivot voxels (tg + ph, j, k) and n/a(n) RF values which are stored at $PE_{jt}(p,s)$ and find out the minimum as local minimum.

15. Divide the processors $PE_{jt}(p,s)$, s = 0..... $[L^t*a(n)]/n - 1$ into smaller subgroups to get the minimum form $[L^t*a(n)]/n$ local minima.These values are the *F* and *d* value of pivot voxels (tg+ph, j, k), t, p = 0....b(n)-1.
16. Now voxels in j^{th} column are further partitioned into $[b(n)]^2$ subgroups by these pivot voxels.

Repeat steps 8 to 15 recursively until all the voxels gets the F and d value at the corresponding processor assigned to them.

Parend

Parend

This completes the EDT algorithm for 2-D planes. After this phase, ED on the same plane for every voxel has been calculated. Figure 3 illustrates the operation of Phase I.

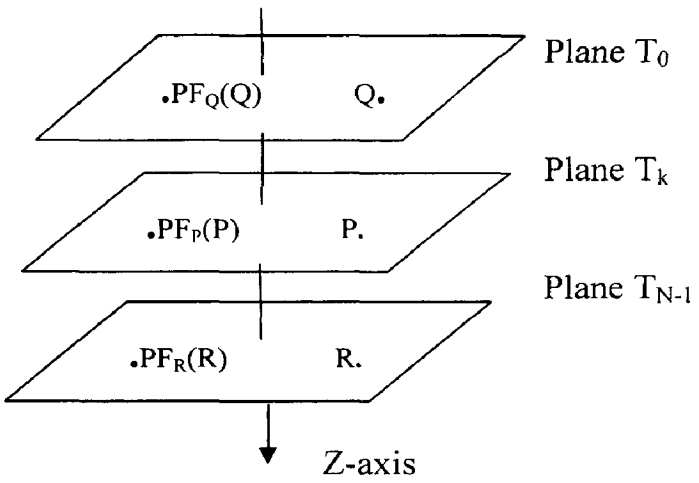


Figure 3.1. Illustration of Completion of Phase I.

Phase II: Vertical Phase

This phase integrates these results obtained in phase I for estimating the actual 3-D EDT for each voxel. In this phase, the processors are reconfigured into N^2 groups of *a(n)*b(n)* processors. Each group processes one vertical column of N voxels parallely.

Let g = n/(a(n)*b(n)) denotes number of voxels per processor and $PE_{ij}(p)$ denotes the p^{th} processor in $(i,j)^{th}$ group. Let PF_k(i, j, k) denotes the ED of voxel (i, j, k) on the same plane T_k computed for every voxel in Phase I.

In this phase, following facts are considered to reduce the search region for computation of F(i, j, k):

1 On the basis of Lemma 5 and 6, it can be deduced that, for any 2 voxels Q and P with same X and Y–coordinates and Q above P in Z-axis direction, if F_Q is known first then the possible region for F_P would be starting from F_Q to the end of image array in Z-axis direction. It is impossible for F_P to be located in the region between the starting of image and F_Q. So it is unnecessary to search for nearest 1-voxel in this region. Hence this is avoided in this phase.

2 Secondly, since the distance to nearest 1-voxel on the same plane i.e. $PF_k(i, j, k)$ is already known, so for computing F(i, j, k) of each voxel, those planes can also be ignored which have Z-direction distance from the current plane to be more than $PF_k(i, j, k)$.

Let RG_k denotes the region in which F(i, j, k) is expected to lie. This region is restricted by the two factors mentioned above.

Steps:

1. Assign N, $PF_k(i, j, k)$, $0 \leq k \leq N-1$ values to group of *a(n)*b(n)* processors. Each processor will have g = n/(a(n)*b(n)) $PF_k(i, j, k)$ values.
2. For i = 0 to N-1 and j = 0 to N-1

   Parbegin

   1. Start with the middle voxel of the column having k = N/2.
   2. Compute the expected region RG_k for plane T_k using the first fact mentioned above.
   3. Ignore those planes T_h from the expected region RG_k which satisfy the condition: $| t - k | \geq PF_k(i, j, k)$
   4. For all $h \in RG_k$
      a. Broadcast $PF_h(i, j, h)$ value from processor $PE_{ij}(h/g)$ to processor $PE_{ij}(k/g)$.
      b. Set $F(i, j, k) = \{ (x, y, h) \mid \min_{h \in RG_k} \{ (k-h)^2 + PF_h(i, j, h)^2 \} \}$

      $$d^2_{i,j,k} = (x - i)^2 + (y - j)^2 + (h - k)^2$$

   5. Divide the column into 2 parts:
      a. T_0 or the last plane before T_k for which EDT has been computed upto T_{k-1}.
      b. T_k to either T_{N-1} or first plane after T_k for which EDT has been computed.

      Recursively repeat the whole process in parallel for these 2 subparts of the original column.

   Parend.

3.3 COMPLEXITY AND SCALABILITY ANALYSIS

The estimation of time complexity and number of processor used in the above algorithm is as follows:

- **Phase I: Plane Phase**

In this phase first, each plane is scanned Row mojor. LARPBS premitive operations like multicasting, extraction etc are performed in this phase which takes O(1) time. Then every processors computes RF values for n/(a(n)*b(n)) voxels so this takes O(n/(a(n)*b(n))) time if n > a(n)*b(n) otherwise it takes O(1) time. In Column major scan of each plane, every step takes O(n/a(n)) time while there are [log n/ log b(n)] steps in the recusion. So the complexity of column major scan is O((n log n)/(a(n)* log b(n))). So the complexity of Phase I will be O(n/(a(n)*b(n))) + O((n log n)/ (a(n)* log b(n))). Assuming n > a(n)*b(n), the overall complexity of this phase will also be ***O(n log n/(a(n)*log b(n)))***.

- **Phase II: Vertical Phase**

In this phase N^2 groups of a(N)*b(N) processors process one vertical column of N voxels in parallel. At each step we devide the vertical column into 2 regions and recursively compute the EDT of the regions. In the worst case the region to be considered could have N voxels, and the whole column is scanned in log N passes. So the total complexity of computing EDT of one vertical column is O(N log N) provided every processor would compute EDT for 1 voxel. But since here every processor computes EDT for N/(a(N)*b(N)) voxels, so the complexity is increased to **$O(N^2 \log N/(a(n) * b(n)))$**.

Considering both phases, the overall complexity would be:

$O(N \log N/(a(N)*\log b(N))) + O(N^2 \log N /(a(n) * b(n)))$

Or effectively it will be ***$O(N^2 \log N /(a(n) * b(n)))$***.

Estimation of total number of processors used:

- **Phase I:**
  - **Row Major Subphase:** N groups of a(N) * b(N) processor for n rows in each plane, and total N planes.
  - **Column Major Subphase:** N groups of a(N) * b(N) processor for N columns in each plane, and total N planes.

  Total $= N^2 * a(N) * b(N)$ processors in both subphases.

- **Phase II :**

N^2 groups of a(N) * b(N) processor for N^2 vertical columns in the whole 3-D array.

Total $= N^2 * a(N) * b(N)$ processors.

So altogether this algorithm needs $N^2 * a(N) * b(N)$ number of processors to complete.

Scalability of the Algorithm : In this algorithm, the time complexity and number of processors required have not been made strictly fixed but rather two parameters a(N) and b(N) have been used to alter these attributes of the algorithm. The values of the two parameters can be selected between 1 to N. By choosing different values for a(N) and b(N), the time complexity and number of processors needed, can be varied, which makes the algorithm more flexible and scalable. For example let us take a(N) = N, and b(N) = N, then the overall time complexity of the algorithm will be $O(N^2 \log N/(N*N)) = O(\log N)$ and the total number of processor required will be $N^2 * N * N = N^4$. Similerly the algorithm can be scaled to different time complexities and different number of processors available in the multiprocessor system.

CHAPTER 4
IMPLEMENTATION

4.1 SYSTEM SPECIFICATION

The scalable parallel algorithm for 3-D EDT computation discussed in the last chapter has been designed for LARPBS multiprocessor model. But the research work done so far on LARPBS model, concludes that the practical implementation and the feasibility analysis of this model are still undergoing processes. So the algorithms designed for this architecture have been usually implemented on different multiprocessor systems. To emulate LARPBS system on other architectures, the inter-processor communication overheads are just considered to be in the scale of LARPBS architecture and the parallel processing of the algorithms can be performed on any other multiprocessor systems. For the implementation of the 3-D EDT algorithm designed in this work, the same strategy has been followed. The implementation of the parallel algorithm has been done on the multiprocessor cluster available in the Institute Computer Center. This had been one of the major challenges in this dissertation work to emulate the LARPBS model on the available multiprocessor cluster. This cluster has the following system configuration which has been used as the implementation platform:

- **Hardware Specification:**
  - HP DL 140G2 6-CPU Cluster, which are mapped onto 45 Processing Elements.
  - Xeon i386 processor.

- **Software Specifications:**
  - LAM/MPI.
  - Red Hat Enterprise Linux ES release 3 (Taroon) – Kernel 2.4.21-4.ELsmp.

Here it is important to note that, since the inter processor communication operations like broadcasting, multicasting etc with negligible time requirement which are specific to LARPBS model, are not provided by the cluster platform, so these operations have been implemented through the primitives provided by MPI and LAM on the HP cluster and the time overhead have been taken in the order of LARPBS model. Detailed discussion about MPI primitives is given in next section.

4.2 MESSAGE PASSING INTERFACE (MPI)

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. MPI enables developers to efficiently program "tightly coupled" algorithms which require nodes to communicate during the course of a computation [23]. MPI is a library of subprograms that can be called by a C program. The foundation of the library is a small group of functions that can be used to achieve parallelism by *message passing* [24]. A message passing function is simply a function that explicitly transmits data from one process to another. Message passing is a powerful and very general method of expressing parallelism.

Message passing can be used to create extremely efficient parallel programs, and it is currently the most widely used method of programming parallel computers. The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and abstractions are built upon lower level message passing routines the benefits of standardization are particularly apparent [24]. This standard is intended to allow users to write portable message passing programs. The standard includes [23]:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Bindings for Fortran and C
- Environmental Management and inquiry
- Profiling interface

MPI provides many features intended to improve performance on scalable parallel computers with specialized inter-processor communication hardware. Thus, it is expected that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard UNIX inter-processor communication protocols will provide portability to workstation clusters and heterogeneous networks of workstations.

Each MPI program must have MPI_Init() and MPI_Finalize() functions. The first and the last MPI statements in an MPI program are MPI_Init() and MPI_Finalize() respectively [22]. All processes must initialize MPI by calling MPI_Init() function and finalize MPI by calling MPI_Finalize() function. The calling syntax for these functions are

```
err = MPI_Init(&argc, &argv)
err = MPI_Finalize()
```

where err is the error number.

In basic message passing the processors coordinate their activities by explicitly sending and receiving messages. For example, at its most basic, the Message Passing Interface (MPI) provides functions for sending a message and receiving a message [23]. The process of sending and receiving is illustrated in Figure 4.1 [25] and the syntaxes are given below:

```
int  MPI_Send(void*          buffer,        /* in */
              int            count,         /* in */
              MPI_Datatype   datatype,      /* in */
              int            destination,   /* in */
              int            tag,           /* in */
              MPI_Comm       communicator   /* in */
             )
int  MPI_Recv(void*          buffer,        /* out */
              int            count,         /* in */
              MPI_Datatype   datatype,      /* in */
              int            destination,   /* in */
              int            tag,           /* in */
              MPI_Comm       communicator   /* in */
              MPI_Status*    status         /* out */
             )
```

A ***Broadcast*** is a collective communication in which a single process sends the same data to every process to the communicator. Each system that runs MPI has a broadcast function MPI_Bcast. The syntax of MPI_Bcast is given below:

```
int MPI_Bcast ( void*          message     /*in / out */,
                int            count       /*in */,
                MPI_Datatype   datatype    /*in */,
                int            root        /* in */,
                MPI_Comm       comm.       /* in */
              )
```

This simply sends a copy of data in message on the process with the rank *root* to each process on the communicator *comm*. It is called by all the process in the communicator with the same arguments for *root* and *comm*. Hence a broadcast message can not be received by function MPI_Recv.

The current version of MPI assumes that processes are statically allocated, i.e., the number of processes is set at the beginning of program execution, and no additional processes are created during execution. Each process is assigned a unique integer rank in the range 0, 1, . . . , $p - 1$, where p is the number of processes. These processes are assigned to the participating processors by the master processor which also integrates the results from all the computing processors and is responsible for final results.

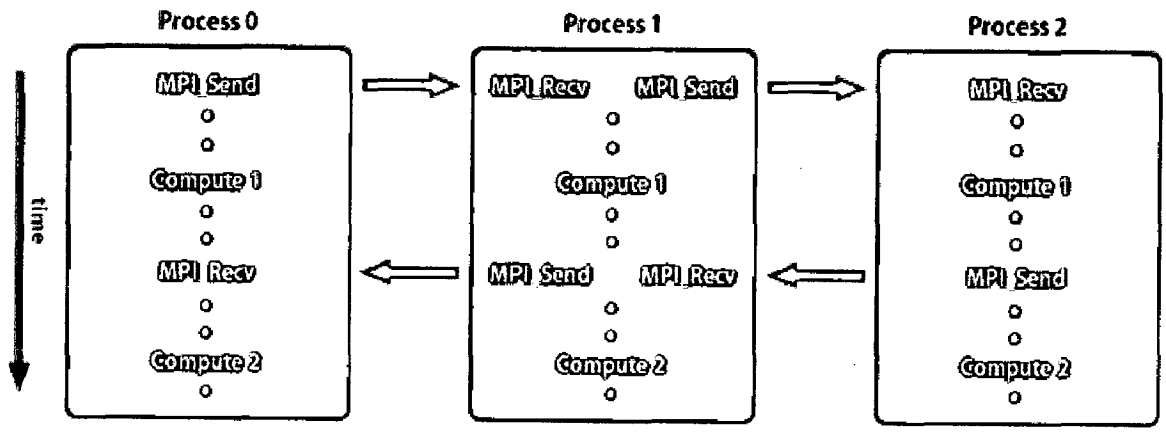


Figure 4.1: Illustration of MPI_Send and MPI_Recv functions by multiple Processors.

4.3 IMLEMENTATION DETAILS

As mentioned in earlier chapters, we have extended *Chen's* approach [1] for the 2D-EDT problem for an N ×N × N, 3-D image array. *Chen et al.* [1] have proposed the algorithm and architecture for the 2D-EDT problem. For an N × N 2D image array, *Chen's* algorithm consists of two main passes, the row scan, and the column scan. First, the rows are scanned in parallel by group of processors then every column is scanned and then, the results are merged. After the column scan, the 2-D EDT results are obtained. A great advantage of *Chen's* algorithm is the ability to scale the 2-D EDT computation with the variations in the number of processors available and also in the time complexity of the overall process. The benefit of this property is that their algorithm is easily parallelized and implemented on different architectures with different time requirement for EDT computation.

The extended algorithm for 3-D binary image arrays continues even after the row and column scan of each 2-D plane in the whole 3-D array. After having calculated the individual EDTs at each 2-D plane, the results for each voxel, are integrated to find out the actual 3-D EDT for each voxel. The 3-D EDT algorithm also possesses the properties of the original 2-D algorithm.

The algorithm discussed in the last chapter has been implemented in 2 versions: sequential and parallel. The performance of both versions of the 3-D EDT algorithm has been compared and analyzed with the results of parallel algorithms proposed by *Lee et al.* [6] (denoted by 3DEDT_LEE), *Yamada* [16] (denoted as 3DEDT_YD) and *Saito and Toriwaki* [19] (denoted 3DEDT_SCAN). Detail discussion about both the versions is given in the subsequent subsections.

4.3.1 Sequential Implementation

The sequential version has been implemented on a simple Pentium-IV machine using C++ language. In this version, the EDT of an N × N × N 3-D binary image array has been calculated by simply following the approach discussed in our algorithm but in sequential manner. In this version, it has been considered that all the parallel processing mentioned in the original algorithm is performed sequentially on a uni-processor machine.

The input has been taken from a file which stores the 3-D binary array containing 1s and 0s. This file is read into a 3-D matrix in the program. Then the given 3-D binary voxel array of size N × N × N is divided into N planes of N × N binary voxels. For each 2-D plane separately, EDT values are calculated in the function *plane_phase ( )*. To calculate the values of 2-D EDT for each voxel at the same plane, the same strategy is followed which is mentioned in the parallel algorithm i.e. scanning 2-D plane row wise then column wise and integrating both results. But this process is performed sequentially one by one for each plane.

After having calculated the 2-D EDT values for each plane individually, function *vertical_phase( )* is called separately for each vertical column of 3-D binary array which compares the 2-D EDT values for each voxel on their own planes, with the distances of other 1-voxels on different planes, to get the actual 3-D EDT values for each voxel. This function is called recursively for each vertical column in such fashion that at each run it divides the whole vertical column into parts and calculates the actual EDT of the dividing voxel. Finally, the computation of 3-D EDT for each voxel gets over by function *vertical_phase ( )* for each vertical column and the output values are stored in the 3-D array which contains the Euclidean Distances for each voxel.

This sequential version of 3-D EDT algorithm has been implemented just for the sake of the performance comparison with the sequential versions of some other 3-D EDT algorithms. The performances have been compared on the basis of the increasing number of 1-voxels in the input array and different values of N. The performance evaluation and comparison results will be discussed in the next chapter.

4.3.2 Parallel Implementation

The parallel algorithm has been implemented on the HP cluster using MPI, for a 3-D binary voxel array of size 128 × 128 × 128. The inter processor communication involved in the algorithm is specific to LARPBS model but they have been implemented using the available MPI primitives for the cluster architecture which has been used as the platform to emulate the LARPBS model. We have compared the performance of this algorithm with the performances of other parallel 3-D EDT algorithms. The comparison is based upon time complexities, computational time and scalability etc. Performance of the algorithm has been analyzed with different values of the parameters a(N) and b(N), thereby altering the number of processors used. Performance comparison results will be discussed in the next chapter.

The implementation of parallel algorithm is decomposed into 2 phases. Initially the input file containing the input 3-D binary array is available at shared memory of the cluster. The computation is initiated by the master processor by reading the input array and distributing it to the participating processors. The processing of both phases has been explained in subsequent subsections.

a. Implementation of Phase I:

This phase can also be called Plane Phase, as in this phase we compute Euclidean Distance for every voxel on its own plane by following the approach of 2D-EDT algorithm on LARPBS model [1]. Initially we have taken total 32 processors for the complete computation but the code has also been run with different number of processors. These processors are subdivided into different sets of processors. Every set computes the 2-D EDT for each plane. The master processor distributes the input 3-D binary array to the participating processors according to allocation strategy mentioned in the algorithm. Now, this phase is run for each of N 2-D arrays from the whole 3-D array. Each plane is scanned through 2 subphases which are explained below:

I) Row major scan:

The processors assigned for one plane are further subdivided into N sets to process each row of the 2-D array. Every processor from these subgroup gets some voxels at each row. The value of leftmost 1-voxel denoted as ***lf*** and right most 1-voxel denoted as ***rf*** are calculated at each row. Now voxels at each row are scanned by the assigned processer. The processors start determining the RF values for each voxel i.e. the distance of nearest 1-voxel on the same row, according to the procedure explained in the algorithm. Once the RF values for each voxel from all the rows are known, this scan is over. The processors assigned for each row contains the RF values of the voxels assigned to them. Now each 2-D array is scanned columnwise.

Note: All these messages between the processors are supposed to be transfered simultaneously without any significant delay in LARPBS model but the cluster architecture involves some delay in this process. This delay has not been taken into consideration in the performance evaluation in order to maintain the standards of LARPBS model.

II) Column major scan:

Once the row major scan is over, the processors are redistributed through the columns of each 2-D array into subgroups. These subgroups of processors are assinged the group of voxels from each column so there are N such subgroups for N columns. The group of voxels in each column are assigned to these processors by following the way explained in the algorithm. Row indices are selected in order to partition each column. After this assignment of voxels to group of processors, the processors transfer the RF values of the old voxel sets to other groups according to the procedure explained in the algorithm and compare those values to the newly obtained values in each column. In this process the set of proccessors are reconfigured for a couple of times. At the end of this process the processors choose the minimum value among all the distances of 1-voxels from the voxel under consideration and sets as the EDT for that voxel at the same plane. After this scan, the processing of phase I completes and the processors contain the values of 2-D EDT for the assigned voxels on their own plane. In the next phase i.e. Vertical phase these results are integrated with the results obtained by scanning the vertical columns of the 3-D array to obtain the actual 3-D EDT transform.

b. Implementation of Phase II:

This phase can also be called Vertical Phase since the 3-D array is scanned vertically i.e. each vertical column is scanned by the group of processors. So there are N^2 such groups of processors formed for N^2 columns. Now on the basis of the properties explained in the algorithm, the search region is reduced for each voxel. Once the search region is decided for each voxel, the processors are reassigned with the 2-D EDT values determined in phase I for different number of voxels.

Now each vertical column is scanned and partitioned according to the algorithm and the processors associated with each voxel determine the distances of other 1-voxels which are not on the same plane. Then comparison between these distances is performed to get the actual Euclidean Distance for a particular voxel. The corresponding processor sends this result to the master processor which stores these values into the output array. The partition process of each vertical column is performed recursively and at each step we calculate the EDT values of different number of voxels. This whole process runs for all the N^2 columns parallely by different group of processors. They all send their results to the master processor as and when the compute the EDT for their assigned voxels. Once the result is collected from all the participating processors, the master processor displays the output matrix which is the Euclidean Distance Transfer of the original input matrix.

4.4 PERFORMANCE EVALUATION METRICS

The algorithm for 3-D EDT computation has been implemented in 2 versions sequential and Parallel. The performances of both versions have been evaluated on basis of different factors. In case of sequential version, the major evaluation metrics are *different numbers of 1-voxels in the 3-D input matrix* and *different sizes of the 3-D input matrix*. Its performance has been compared with the sequential versions of other EDT algorithm on different platforms on these 2 factors. As this version is implemented and run on a single machine, so there is no factor of inter processor communication overhead involved. Complete execution is performed sequentially and the total execution time is compared with the same of the implementations of other algorithms.

The parallel version of the algorithm has been implemented on HP cluster architecture which emulates the performance of LARPBS model. Inter processor communication overhead is one of the major factors which affects the performance of the algorithm. Here we have supposed this overhead to be in the order of the LARPBS model in order to maintain the computation time of the algorithm. Apart from this factor some other factors which affect the performance are listed below:

1. Different sizes of 3-D input matrix i.e. different values of N.
2. Different values of parameter a(N) and b(N) taken between 1 and N.
3. Variable Number of processors used.
4. Variable Time Complexities.

In the next chapter we will discuss the performance evaluation of both versions on the basis of these metrics along with the comparative analysis of the performances of our algorithm with other 3-D EDT algorithms.

CHAPTER 5
RESULTS AND DISCUSSION

In this chapter, we will discuss and analyze the performance results of both version of implementation of the 3-D EDT algorithm designed in this dissertation work compare.

5.2 PERFORMANCE EVALUATION AND COMPARISION

5.1.1 Performance of Sequential Version

The sequential version has been implemented on a single Pentium-IV machine using C++ language. In this version, the EDT of an N × N × N 3-D binary image array has been calculated by following the approach discussed in our algorithm but in sequential manner. The execution time of this version is compared with that of 3 sequential versions of other 3-D EDT algorithms with the change in various parameters.

The performance of the sequential versions of our 3-D EDT algorithm has been compared and analyzed with the performances of the algorithms proposed by *Lee et al.* [6] (denoted by 3DEDT_LEE), *Yamada* [16] (denoted as 3DEDT_YD) and *Saito and Toriwaki* [19] (denoted as 3DEDT_SCAN).

Fig. 5.1 shows the performance comparisons for the 3D-EDT sequential algorithms with the increasing number of 1-voxels Vn × Vn in the input binary image matrix of size 128 × 128 × 128 where the 1-voxel distribution is uniform. As we can see from Fig. 5.1, program 3DEDT_YD is very time-consuming when the number of 1-voxels is sparse because the propagation distance is long and requires more iteration. By increasing the number of 1-voxels, the running time of program 3DEDT_YD converges to a stable time. The running time of program 3DEDT_YD is very sensitive to the number and the distribution of 1-voxels. The program 3DEDT_SCAN runs much faster and more stably than program 3DEDT_YD. interestingly however, 3DEDT_LEE shows an even stable and better performance than the other two.

Our implementation shows comparatively better performance than 3DEDT_YD and 3DEDT_SCAN. For less number of 1- voxels it gives a little worse performance than 3DEDT_LEE but for higher number of 1-voxels it improves and shows a better performance.

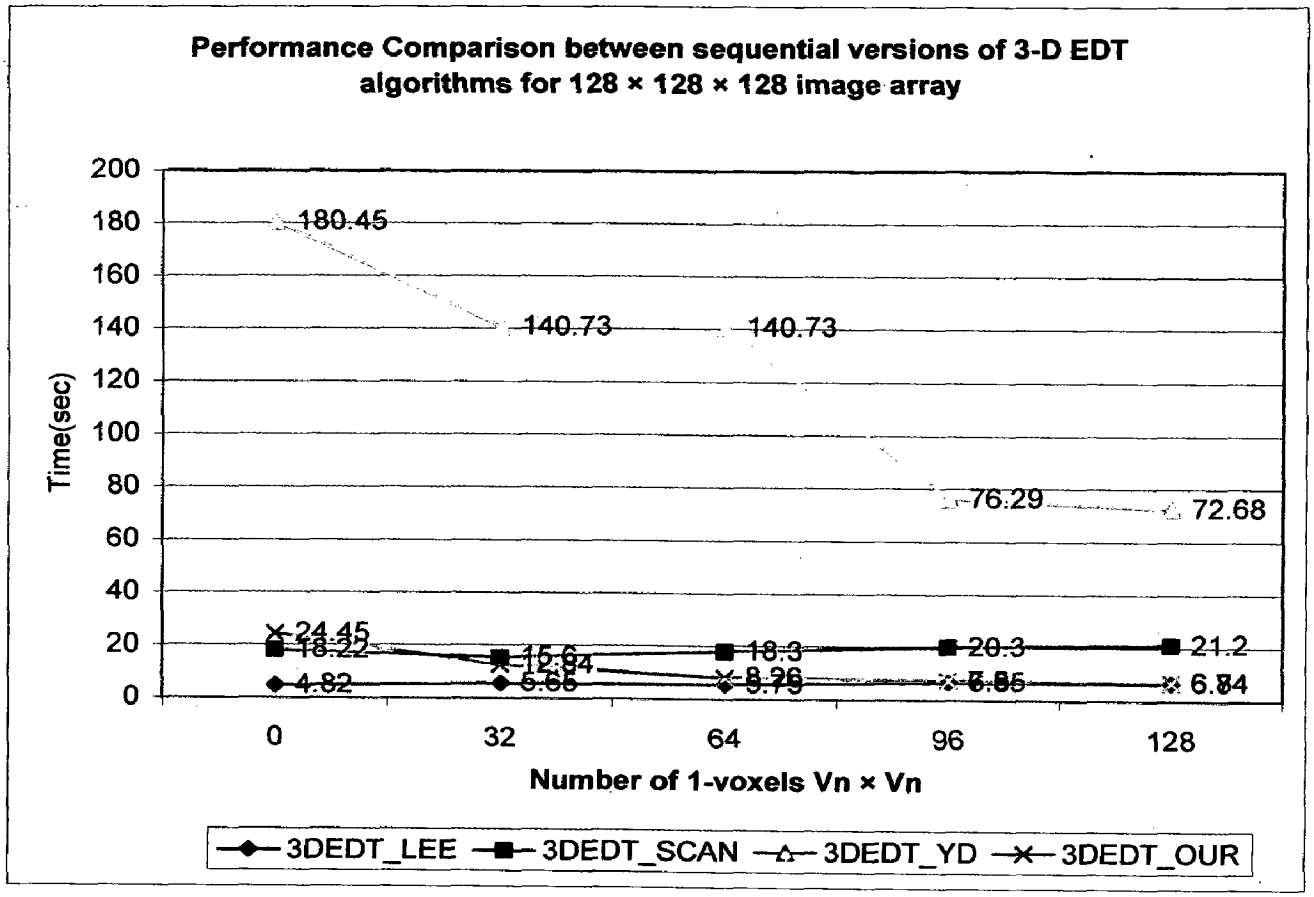


Figure 5.1 Performance Comparison between sequential versions of 3-D EDT algorithms with increasing number of 1-voxels.

Fig. 5.2 shows the performance comparisons of the above mentioned 3D-EDT sequential algorithms with the increasing size of the voxel image array. Without being affected with the size N, our sequential implementation runs with almost same performance as 3DEDT_LEE algorithms performance. Here also we can see that 3DEDT_YD still exhibits the poorest performance, while 3DEDT_SCAN shows an average performance among the all the other algorithms.

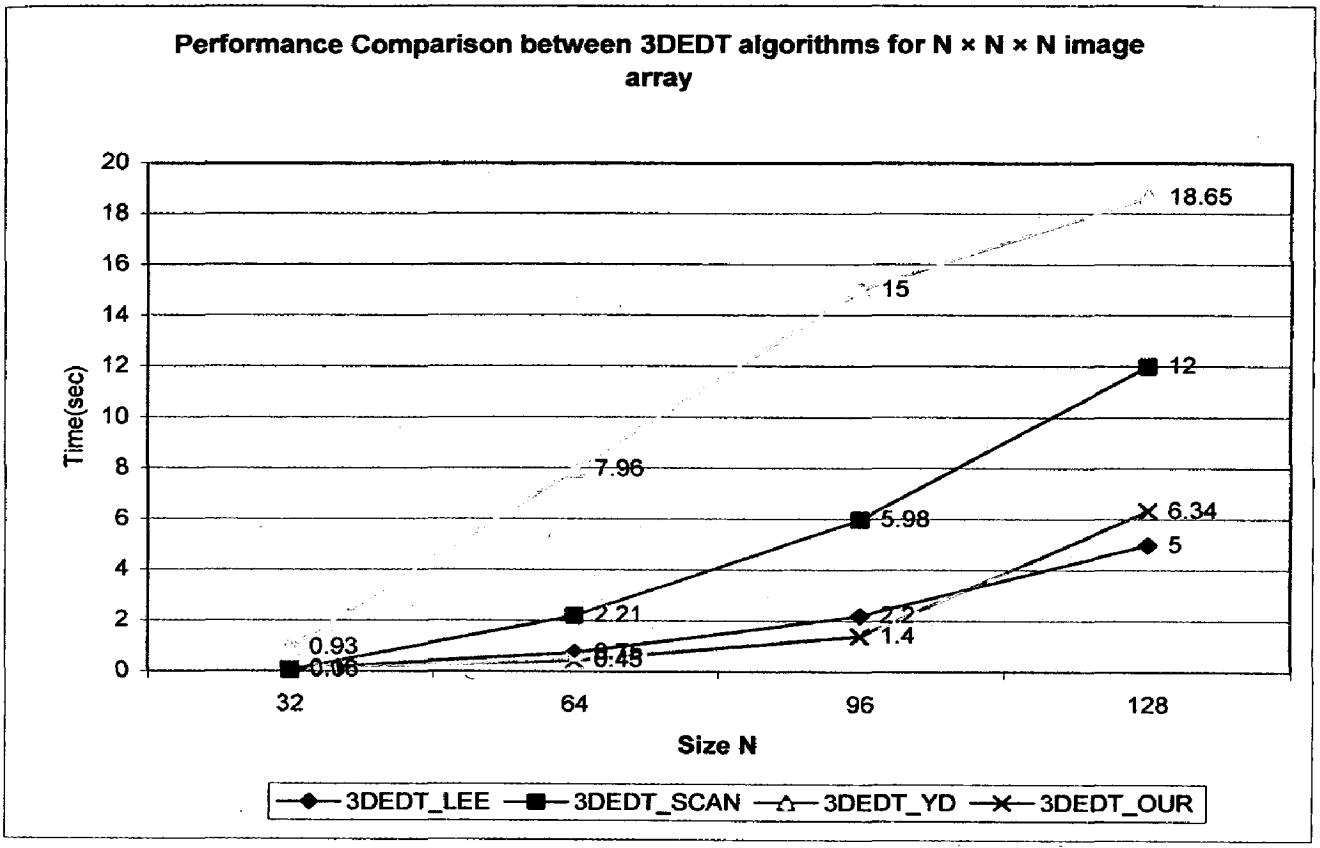


Figure 5.2 Performance Comparison between sequential versions of 3-D EDT algorithms with increasing size of input array.

Run Time Ratio (RTT) can be defined as the ratio of the time taken by any other algorithm to that of our algorithm. Table 5.1 shows the run time ratio of our sequential implementation with other 3 sequential algorithms for various values of N.

Table 5.1 RTT comparison between the sequential algorithms for different values of N.

EDT Algorithm(X)	**Time Complexity**	**RTT = X / (3DEDT_OUR)**			
		N=32	**N=64**	**N=96**	**N=128**
3DEDT_LEE	$O(N^3)$	0.6	1.66	1.57	0.78
3DEDT_SCAN	$O(N^4)$	0.6	4.91	4.27	1.89
3DEDT_YD	$O(N^3)$	9.3	17.68	10.71	2.94

5.1.2 Performance of Parallel Version

Here, the parallel 3D-EDT algorithm has been implemented the HP cluster architecture using MPI (Message Passing Interface) to emulate the LARPBS model. For a 128 × 128 × 128 3-D voxel image array. Here it should be noted that the delay involved in the inter processor communication in LARPBS model is negligible as we have seen in chapter 2. While the HP cluster system is non-shared memory architecture and the data exchange time exceeds the processor computation time. So in order to maintain the overhead as per the LARPBS model standards, we have ignored the actual delay introduced because of inter processor communication and considered only the actual computational time.

For each sub image array, the computation loading is different for each processor and it is dependent on the particular 1-voxel pattern that is loaded. The running time on the cluster is bounded by the worst computation time for each sub image array. So, for each phase, we sum up the worst computation time of each processor.

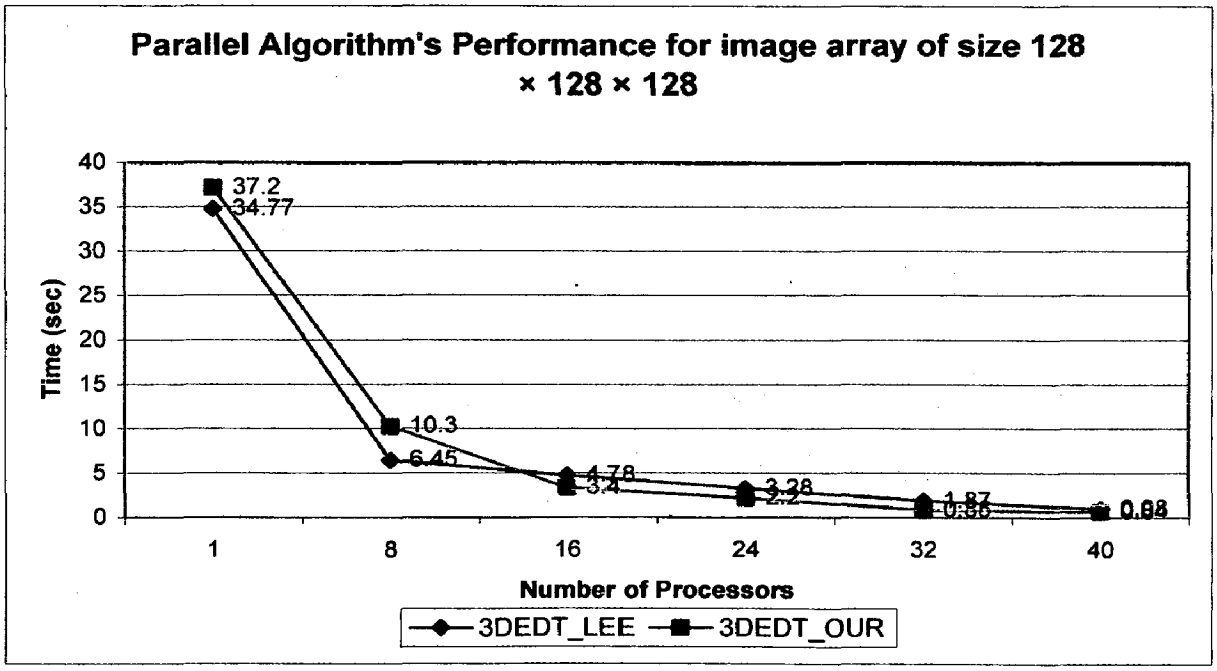


Figure 5.3 Performance Comparison of Parallel 3-D EDT algorithms with increasing number of processors used.

Fig. 5.3 shows the parallel algorithm's running time with different number of processors used, and it is compared with *Lee et al.* [6] (denoted by 3DEDT_LEE), to obtain the speed up curve of the proposed parallel algorithm. From Fig. 5.3, we see that both performances are quite close but with higher number of processors our algorithm does a little better than 3DEDT_LEE. On the other hand with less number of processors 3DEDT_LEE performs well than our algorithms.

Finally, we have analyzed the performance of the parallel MPI program running on HP cluster on the basis of the data exchange rate, actual computational time and the total running time of the program. This analysis has been shown in Figure 5.4. From this figure it is clear that the contribution of data exchange time in total running time is less significant than the actual computational time since our algorithm exhibit quite a small amount of data exchange which makes the inter processor communication overhead negligible and improves the overall performance.

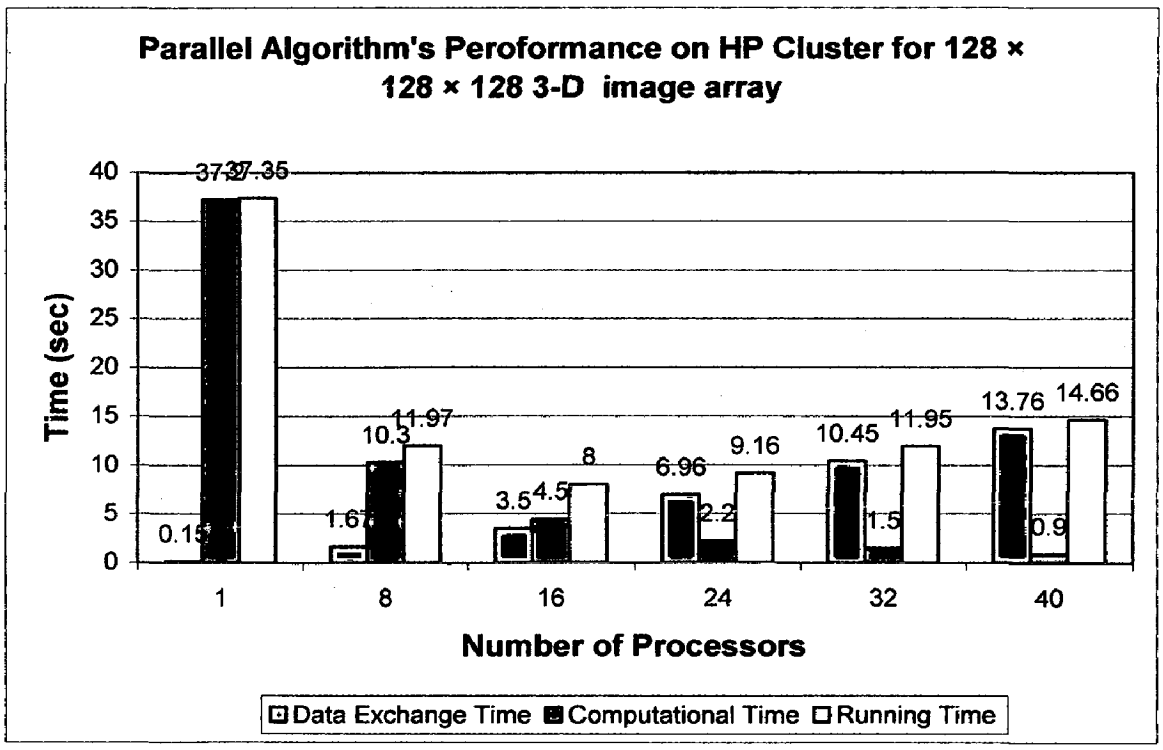


Figure 5.3 Performance of 3-D EDT parallel MPI program running on HP Cluster.

5.2 COMPLEXITY AND SCALABILITY COMPARISON

We have discussed in the chapter 3 that the complexity of our algorithm is $O(N^2 \log N / (a(n) * b(n)))$ and the total number of processor used in both phases are $N^2 * a(n) * b(n)$. This is clear that by selecting different values for parameters a(N) and b(N), we can make our algorithm more flexible and also scalable to different architectures on the basis of the number of processors available for computation. Also we can achieve, different time complexities by varying the values of these parameters. Table 5.2 gives the summery about the various time complexities and number of processors required by the selection of various values for a(N) and b(N). The list does not end here but can be extended for some more values of these parameteres.

Table 5.2 Different algorithms derivable from this framework

a(N)	b(N)	Time Complexity	Number of Processors
N	N	O(log N)	N^4
N	Constant r	O(N log N)	rN^3
N/ log N	log N	O(N log N)	N^3
N/ log N	Constant r	O(N $\log^2$ N)	rN^3 / log N
1	log N	O(N^2)	N^2 log N
1	Constant r	O(N^2 log N)	rN^2
N	$N^{0.5}$	O($N^{0.5}$ log N)	$N^{3.5}$
log N	Constant r	O(N^2)	N^2 log N

The performance of our algorithm has been compared with 3 other 3DEDT algorithms. If we talk about the complexities of these algorithms, Yamada's algorithm takes O (N^3) time in the worst case. It requires N iterations to converge in the worst case. During each iteration, it takes O (N^2). So the sequential time to scan the whole image array, in the worst case would be O (N^3). Saito and Toriwaki [19] presented several EDT algorithms based on the scan approach for an n-dimensional image array. For the 3D-EDT problem, Saito and Toriwaki's EDT algorithm also takes O (N^4) time complexity. Our sequential implementation on the other hand takes O (N^3) time to scan the whole 3-D array.

5.3 DISSCUSSION

We have implemented the proposed algorithm sequentially and compared the performance of our implementation with those proposed by Lee [6], Yamada [16] and Toriwaki [19]. Based on the comparison, the algorithm presented in this paper exhibits a better performance to the other three algorithms. We also implemented the parallel algorithm which runs on an HP DL cluster and compared it with Lee's parallel implementation. Its performance demonstrates near to that of Lee's. The latter, however, takes too much data exchange time.

The performance improvements of our implementation of the 3D EDT algorithm can be achieved through proper selection of a(N) and b(N). To get the higher speed, the time complexity $\mathbf{O\ (N^2\ log\ N\ /\ (a\ (n)\ *\ b\ (n)))}$ must be minimized, i.e., **a (n) * b (n)** should reach its minimum. Therefore, we must choose both parameters as small as possible. To get the highest efficiency, the time-processor cost O (N^4 log N) must be minimized, but since it is independent of a (N) and b (N) so variation in these parameters will not affect the efficiency of the algorithm.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

A scalable parallel algorithm for 3-D EDT computation on LARPBS model has been designed and implemented in this dissertation work. The algorithm is an extension to 2-D EDT algorithm on the same model. The algorithm computes the 3-D EDT with time complexity $O(N^2 \log N /(a(n) * b(n)))$ and $N^2*a(n)*b(n)$ number of processors. LARPBS model has been chosen for this algorithm since it gives the benefit of low communication time between processors and high speed data transfer. The operations like multicasting, broadcasting etc which involve heavy interprocessor communication, can be performed in constant time on this model. This algorithm takes the advantage of these features and results in low communication overhead.

After the introductory chapter 1, in chapter 2 we discussed about the basic information and some properties of Euclidean Transform. Some useful lemmas were given which are helpful in understanding the wrokflow and lgic of the algorithm. We also dicussed about the architechture of LARPBS model, its properties, workflow, interprocessor communication mechanism etc. Rest of the chapter dicussed about the work done so far in these areas and the research gaps which gave the motivation for this work.

We derived the parallel algorithmm for 3-D EDT computation as an extension to the 2-D EDT algorithm for the same LARPBS model in chapter 3. Along with the algorithm, its complexity and scalability analysis has also been given in this chapter.

The detailed discussion about the implementations platform, implementation tool i.e. LAM/MPI was done in Chapter 4. The sequential and parallel implementation of the 3-D EDT algorithm were discussed in detail. This chapter also gives a breif information about the metrics used for the performance analysis and comaparison.

Finally Chapter 5 gives the detailed performance analysis and comaprative study between the other 3-D EDT algorithms for other platform with our algorithm and both implentations of it. The basis of the performance and comaparison are the metrics given in the last chapter.

6.2 SUGGESSION FOR FUTURE WORK

One of the major challenge faced during this dessertation work is the emulatioon of LARPBS model on the available HP cluster architecture. So for future work in this area involves the developement of some tools which could effciently emulate the performance of LARPBS model through other multiprocessor models sothat the algorithms specifically designed for LARPBS can be implemented, evaluated and analyzed, on other architectures as well.

A second intiatve which could be taken is incorporation of some efficient task scheduling or task-to-processors mapping approaches for various multiprocessor architectures with the derived algorithm would further optimize the perfomance of the algorithm by reducing communication overheads. This is seen that during the design of the parallel algorithms, the hardware architectures are ususally not considered. When the algorithm is designed for any specific model than by considering the hardware architecture of that model we can opt some efficient task scheduing strategy for the given multiprocessor platform, which may result into further optimization of the algorithm. In future these issues could be taken into research efforts.

REFERENCES

[1] Ling Chen, Yi Pan, and Xiao-hua Xu, *"Scalable and Efficient Parallel Algorithms for Euclidean Distance Transform on the LARPBS Model"*, IEEE Transactions on Parallel and Distributed Systems, Vol. 15, No. 11, November 2004.

[2] H. Blum, *"A Transformation for Extracting New Descriptors of Shape"*. Models for the Perception of Speech and Visual Form, Mass.: MIT Press, W. Wathen-Dunn, ed., pp. 362-380, 1967.

[3] A. Rosenfeld and J.L. Pfalz, *"Sequential Operations in Digital Picture Processing"*. ACM, vol. 13, pp. 471-494, 1966.

[4] A. Rosenfeld and J.L. Pfalz, *"Distance Function on Digital Pictures"*. Pattern Recognition, vol. 1, pp. 33-61, 1968.

[5] A. Rosenfeld and J. L. Pfalz, *"Distance function on digital pictures and Pattern Recognition"*.

[6] Yu-Hua Lee, Shi-Jinn Horng, and Jennifer Seitzer, *"Parallel Computation of the Euclidean Distance Transform on a Three-Dimensional Image Array"*, IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 3, March 2003.

[7] Yi Pan Mounir Harndi, *"Quicksort on A Linear Array With A Reconfigurable Pipelined Bus System"*.

[8] L. Chen and H. Y. H. Chuang, *"A fast algorithm for Euclidean distance maps of a 2-D binary image"*. Inform. Process. Lett., **51** (1994), pp. 25-29.

[9] L. Chen, *"An Optimal Algorithm for Complete Euclidean Distance Transform"*, Chinese J. of Computer, vol. 18, no. 8, pp. 611-616, 1995.

[10] Ren´e Rold´an a, Brian J. d'Auriol, *"A Preliminary Feasibility Study of the LARPBS Optical Bus Parallel Model"*.

[11] Amitava Datta, Subbiah Soundaralakshmi, *"Fast and Scalable Algorithms for the Euclidean Distance Transform on the LARPBS"*.

[12] Z. Guo, R. Melhem, R. Hall, D. Chiarulli, and **S.** Levitan, *"Array Processors with Pipelined Optical Busses"*, Journal of Parallel and Distributed Computing, 1, 2, 3, pp. 269-282 (1991).

[13] A. Fujiwara, T. Masuzawa, and H. Fujiwara, *"An Optimal Parallel Algorithm for the Euclidean Distance Maps of 2-D Binary Images"*, Information Processing Letters, vol. 54, pp. 277-282, 1995.

[14] K. Li, Y. Pan and M. Hamdi, *"Solving graph theory problems using reconfigurable pipelined optical buses"*, Parallel Computing, 26 (2000), pp. 723-735.

[15] K. Li, Y. Pan and S. Q. Zheng, *"Fast and processor efficient parallel matrix multiplication algorithms on a linear array with a reconfigurable pipelined bus system"*, IEEE Trans. Parallel and Distributed Systems, 9, (8), (1998), pp. 705-720.

[16] Yamada H., *"Complete Euclidean distance transformation by parallel operation"*. Proc. 7th International Conference on Pattern Recognition, 1984, pp. 69-71.

[17] M. N. Kolountzakis and K.N. Kuatulakos, *"Fast Computation of the Euclidean Distance Maps for the Binary Image"*, Information Processing Letters, vol. 43, pp. 181-184, 1992.

[18] A. Fujiwara, T. Masuzawa and H. Fujiwara, *"An optimal parallel algorithm for the Euclidean distance maps of 2-D binary images"*. Inform. Process. Lett. **54**, (1995), 295-300.

[19] T. Saito and J. Toriwaki, *"New Algorithms for Euclidean Distance Transformation of an n-Dimensional Digitized Picture with Applications,"* Pattern Recognition, vol. 27, pp. 1551-1565, 1994.

[20] Y. Pan, Y. Li, J. Li, K. Li and S.-Q. Zheng, *"Computing distance maps efficiently using an optical bus"*, Proc. IPDPS 2000 Workshop on Parallel and Distributed Computing in Image Processing, Video Processing and Multimedia (PDIVM 2000), LNCS **1800**, pp. 178-185.

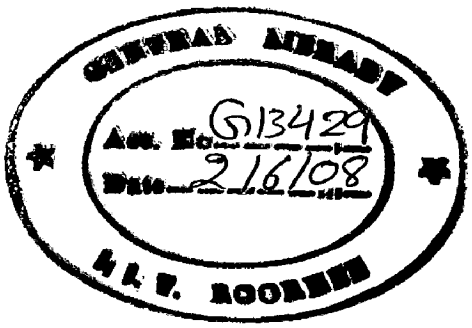
[21] Yi Pan and Keqin Li. *"Linear array with a reconfigurable pipelined bus system — concepts and applications."* In H.R. Arabnia, editor, Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA'96), Vol. III, pages 1431–1441, Sunnyvale, California, USA, August 1996.

[22] Y. Pan, Y. Li, J. Li, K. Li, and S.-Q. Zheng, *"Efficient Parallel Algorithms for Distance Maps of 2D Binary Images Using an Optical Bus"*, IEEE Trans. Systems, Man, and Cybernetics—Part A: Systems and Humans, vol. 32, no. 2, pp. 228-236, Mar. 2002.

[23] MPI information Available at website,
http://www-unix.mcs.anl.gov/mpi/ , *Last Accessed May 25, 2007*

[24] Peter S. Pacheco, *"Parallel Processing With MPI"*, Morgan Kaufmann Publishers.

[25] LAM/MPI information Available at:
http://www.lam-mpi.org/using/docs/7.1.3-user.pdf



APPENDIX

SOURCE CODE LISTING

Sequential Implementation of 3D EDT Algorithm in C++.

----------Edt.h----------

```
/* Euclidean distance transform */
#ifndef EDT_H
#define EDT_H

#include <algorithm>
#include "image.h"
#define INF 1E20

/* edt of 1d function using squared distance */
static float *dt(float *f, int n)
{
 float *d = new float[n];
 int *v = new int[n];
 float *z = new float[n+1];
 int k = 0;
 v[0] = 0;
 z[0] = -INF;
 z[1] = +INF;
 for (int q = 1; q <= n-1; q++)
{  float s  = ((f[q]+square(q))-(f[v[k]]+square(v[k])))/(2*q-2*v[k]);
  while (s <= z[k]) {
   k--;
   s  = ((f[q]+square(q))-(f[v[k]]+square(v[k])))/(2*q-2*v[k]);
```

```
  }
   k++;
   v[k] = q;
   z[k] = s;
   z[k+1] = +INF;
  }

  k = 0;
  for (int q = 0; q <= n-1; q++) {
   while (z[k+1] < q)
    k++;
   d[q] = square(q-v[k]) + f[v[k]];
  }

  delete [] v;
  delete [] z;
  return d;
}

/* dt of 2d function using squared distance */
static void dt(image<float> *im)
{
  int width = im->width();
  int height = im->height();
  float *f = new float[std::max(width,height)];

  // transform along columns
  for (int x = 0; x < width; x++)
{
   for (int y = 0; y < height; y++)
   {
```

```
      f[y] = imRef(im, x, y);
    }
    float *d = dt(f, height);
    for (int y = 0; y < height; y++) {
      imRef(im, x, y) = d[y];
    }
    delete [] d;
  }

  // transform along rows
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      f[x] = imRef(im, x, y);
    }
    float *d = dt(f, width);
    for (int x = 0; x < width; x++) {
      imRef(im, x, y) = d[x];
    }
    delete [] d;
  }

  delete f;
}

/* dt of binary image using squared distance */
static image<float> *dt(image<uchar> *im, uchar on = 1)
 {
  int width = im->width();
  int height = im->height();
  image<float> *out = new image<float>(width, height, false);
  for (int y = 0; y < height; y++) {
```

```
  for (int x = 0; x < width; x++)
 {
   if (imRef(im, x, y) == on)
       imRef(out, x, y) = 0;
   else
       imRef(out, x, y) = INF;
  }
 }
 dt(out);
 return out;
}
#endif
```

----------image.h----------

```
/* a simple image class */

#ifndef IMAGE_H
#define IMAGE_H
#include <cstring>

template <class T>

class image
{
 public:
  /* create an image */
  image(const int width, const int height, const bool init = true);

  /* delete an image */
  ~image();
```

```
  /* init an image */
  void init(const T &val);

  /* copy an image */
  image<T> *copy() const;

  /* get the width of an image. */
  int width() const { return w; }

  /* get the height of an image. */
  int height() const { return h; }

  /* image data. */
  T *data;

  /* row pointers. */
  T **access;

 private:
  int w, h;
};

/* use imRef to access image data. */
#define imRef(im, x, y) (im->access[y][x])

/* use imPtr to get pointer to image data. */
#define imPtr(im, x, y) &(im->access[y][x])

template <class T>
image<T>::image(const int width, const int height, const bool init) {
```

```
  w = width;
  h = height;
  data = new T[w * h];  // allocate space for image data
  access = new T*[h];   // allocate space for row pointers

  // initialize row pointers
  for (int i = 0; i < h; i++)
    access[i] = data + (i * w);

  if (init)
    memset(data, 0, w * h * sizeof(T));
}

template <class T>
image<T>::~image() {
  delete [] data;
  delete [] access;
}

template <class T>
void image<T>::init(const T &val) {
  T *ptr = imPtr(this, 0, 0);
  T *end = imPtr(this, w-1, h-1);
  while (ptr <= end)
    *ptr++ = val;
}


template <class T>
image<T> *image<T>::copy() const {
  image<T> *im = new image<T>(w, h, false);
```

```
 memcpy(im->data, data, w * h * sizeof(T));
 return im;
}

#endif
```

----------imconv.h----------

```
#ifndef CONV_H
#define CONV_H

#include <climits>
#include "image.h"
#include "imutil.h"
#include "misc.h"

#defineRED_WEIGHT       0.299
#define GREEN_WEIGHT    0.587
#define BLUE_WEIGHT     0.114

static image<uchar> *imageRGBtoGRAY(image<rgb> *input) {
 int width = input->width();
 int height = input->height();
 image<uchar> *output = new image<uchar>(width, height, false);

 for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
   imRef(output, x, y) = (uchar)
       (imRef(input, x, y).r * RED_WEIGHT +
        imRef(input, x, y).g * GREEN_WEIGHT +
        imRef(input, x, y).b * BLUE_WEIGHT);
```

```
    }
  }
  return output;
}

static image<rgb> *imageGRAYtoRGB(image<uchar> *input) {
  int width = input->width();
  int height = input->height();
  image<rgb> *output = new image<rgb>(width, height, false);

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      imRef(output, x, y).r = imRef(input, x, y);
      imRef(output, x, y).g = imRef(input, x, y);
      imRef(output, x, y).b = imRef(input, x, y);
    }
  }
  return output;
}

static image<float> *imageUCHARtoFLOAT(image<uchar> *input) {
  int width = input->width();
  int height = input->height();
  image<float> *output = new image<float>(width, height, false);

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      imRef(output, x, y) = imRef(input, x, y);
    }
  }
  return output;
```

```
}

static image<float> *imageINTtoFLOAT(image<int> *input) {
 int width = input->width();
 int height = input->height();
 image<float> *output = new image<float>(width, height, false);

 for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
   imRef(output, x, y) = imRef(input, x, y);
  }
 }
 return output;
}

static image<uchar> *imageFLOATtoUCHAR(image<float> *input,
                                       float min, float max) {
 int width = input->width();
 int height = input->height();
 image<uchar> *output = new image<uchar>(width, height, false);

 if (max == min)
  return output;

 float scale = UCHAR_MAX / (max - min);
 for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
   uchar val = (uchar)((imRef(input, x, y) - min) * scale);
   imRef(output, x, y) = bound(val, (uchar)0, (uchar)UCHAR_MAX);
  }
 }
```

```
  return output;
}

static image<uchar> *imageFLOATtoUCHAR(image<float> *input) {
  float min, max;
  min_max(input, &min, &max);
  return imageFLOATtoUCHAR(input, min, max);
}

static image<long> *imageUCHARtoLONG(image<uchar> *input) {
  int width = input->width();
  int height = input->height();
  image<long> *output = new image<long>(width, height, false);

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      imRef(output, x, y) = imRef(input, x, y);
    }
  }
  return output;
}

static image<uchar> *imageLONGtoUCHAR(image<long> *input, long min, long max)
{
  int width = input->width();
  int height = input->height();
  image<uchar> *output = new image<uchar>(width, height, false);

  if (max == min)
    return output;
```

```
  float scale = UCHAR_MAX / (float)(max - min);
  for (int y = 0; y < height; y++) {
   for (int x = 0; x < width; x++) {
    uchar val = (uchar)((imRef(input, x, y) - min) * scale);
    imRef(output, x, y) = bound(val, (uchar)0, (uchar)UCHAR_MAX);
   }
  }
  return output;
}

static image<uchar> *imageLONGtoUCHAR(image<long> *input) {
  long min, max;
  min_max(input, &min, &max);
  return imageLONGtoUCHAR(input, min, max);
}

static image<uchar> *imageSHORTtoUCHAR(image<short> *input,
                                       short min, short max) {
  int width = input->width();
  int height = input->height();
  image<uchar> *output = new image<uchar>(width, height, false);

  if (max == min)
   return output;

  float scale = UCHAR_MAX / (float)(max - min);
  for (int y = 0; y < height; y++) {
   for (int x = 0; x < width; x++) {
    uchar val = (uchar)((imRef(input, x, y) - min) * scale);
    imRef(output, x, y) = bound(val, (uchar)0, (uchar)UCHAR_MAX);
   }
```

```
/* basic image I/O */
#ifndef PNM_FILE_H
#define PNM_FILE_H

#include <cstdlib>
#include <climits>
#include <cstring>
#include <fstream>
#include "image.h"
#include "misc.h"

#define BUF_SIZE 256

class pnm_error { };

static void read_packed(unsigned char *data, int size, std::ifstream &f) {
 unsigned char c = 0;

 int bitshift = -1;
 for (int pos = 0; pos < size; pos++) {
  if (bitshift == -1) {
   c = f.get();
   bitshift = 7;
  }
  data[pos] = (c >> bitshift) & 1;
  bitshift--;
  }
}
```

```
 static void write_packed(unsigned char *data, int size, std::ofstream &f) {
  unsigned char c = 0;

  int bitshift = 7;
  for (int pos = 0; pos < size; pos++) {
     c = c | (data[pos] << bitshift);
     bitshift--;
     if ((bitshift == -1) || (pos == size-1)) {
         f.put(c);
         bitshift = 7;
         c = 0;
     }
  }
}

static void pnm_read(std::ifstream &file, char *buf) {
  char doc[BUF_SIZE];
  char c;

  file >> c;
  while (c == '#') {
   file.getline(doc, BUF_SIZE);
   file >> c;
  }
  file.putback(c);

  file.width(BUF_SIZE);
  file >> buf;
  file.ignore();
}
```

```
static image<uchar> *loadPBM(const char *name) {
  char buf[BUF_SIZE];

  /* read header */
  std::ifstream file(name, std::ios::in | std::ios::binary);
  pnm_read(file, buf);
  if (strncmp(buf, "P4", 2))
    throw pnm_error();

  pnm_read(file, buf);
  int width = atoi(buf);
  pnm_read(file, buf);
  int height = atoi(buf);

  /* read data */
  image<uchar> *im = new image<uchar>(width, height);
  for (int i = 0; i < height; i++)
    read_packed(imPtr(im, 0, i), width, file);

  return im;
}

static void savePBM(image<uchar> *im, const char *name) {
  int width = im->width();
  int height = im->height();
  std::ofstream file(name, std::ios::out | std::ios::binary);

  file << "P4\n" << width << " " << height << "\n";
  for (int i = 0; i < height; i++)
    write_packed(imPtr(im, 0, i), width, file);
}
```

```
  float scale = UCHAR_MAX / (float)(max - min);
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      uchar val = (uchar)((imRef(input, x, y) - min) * scale);
      imRef(output, x, y) = bound(val, (uchar)0, (uchar)UCHAR_MAX);
    }
  }
  return output;
}

static image<uchar> *imageLONGtoUCHAR(image<long> *input) {
  long min, max;
  min_max(input, &min, &max);
  return imageLONGtoUCHAR(input, min, max);
}

static image<uchar> *imageSHORTtoUCHAR(image<short> *input,
                                       short min, short max) {
  int width = input->width();
  int height = input->height();
  image<uchar> *output = new image<uchar>(width, height, false);

  if (max == min)
    return output;

  float scale = UCHAR_MAX / (float)(max - min);
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      uchar val = (uchar)((imRef(input, x, y) - min) * scale);
      imRef(output, x, y) = bound(val, (uchar)0, (uchar)UCHAR_MAX);
    }
```

```
  }
  return output;
}

static image<uchar> *imageSHORTtoUCHAR(image<short> *input) {
  short min, max;
  min_max(input, &min, &max);
  return imageSHORTtoUCHAR(input, min, max);
}

#endif
```

----------imutil.h----------

```
/* some image utilities */

#ifndef IMUTIL_H
#define IMUTIL_H

#include "image.h"
#include "misc.h"

/* compute minimum and maximum value in an image */
template <class T>
void min_max(image<T> *im, T *ret_min, T *ret_max) {
  int width = im->width();
  int height = im->height();

  T min = imRef(im, 0, 0);
  T max = imRef(im, 0, 0);
  for (int y = 0; y < height; y++) {
```

```
    for (int x = 0; x < width; x++) {
      T val = imRef(im, x, y);
      if (min > val)
          min = val;
      if (max < val)
          max = val;
    }
  }

  *ret_min = min;
  *ret_max = max;
}

/* threshold image */
template <class T>
image<uchar> *threshold(image<T> *src, int t) {
  int width = src->width();
  int height = src->height();
  image<uchar> *dst = new image<uchar>(width, height);

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      imRef(dst, x, y) = (imRef(src, x, y) >= t);
    }
  }

  return dst;
}
#endif
```

```
/* basic image I/O */
#ifndef PNM_FILE_H
#define PNM_FILE_H

#include <cstdlib>
#include <climits>
#include <cstring>
#include <fstream>
#include "image.h"
#include "misc.h"

#define BUF_SIZE 256

class pnm_error { };

static void read_packed(unsigned char *data, int size, std::ifstream &f) {
 unsigned char c = 0;

 int bitshift = -1;
 for (int pos = 0; pos < size; pos++) {
  if (bitshift == -1) {
   c = f.get();
   bitshift = 7;
  }
  data[pos] = (c >> bitshift) & 1;
  bitshift--;
  }
}
```

```
static void write_packed(unsigned char *data, int size, std::ofstream &f) {
  unsigned char c = 0;

  int bitshift = 7;
  for (int pos = 0; pos < size; pos++) {
     c = c | (data[pos] << bitshift);
     bitshift--;
     if ((bitshift == -1) || (pos == size-1)) {
          f.put(c);
          bitshift = 7;
          c = 0;
     }
  }
}

static void pnm_read(std::ifstream &file, char *buf) {
  char doc[BUF_SIZE];
  char c;

  file >> c;
  while (c == '#') {
    file.getline(doc, BUF_SIZE);
    file >> c;
  }
  file.putback(c);

  file.width(BUF_SIZE);
  file >> buf;
  file.ignore();
}
```

```
static image<uchar> *loadPBM(const char *name) {
  char buf[BUF_SIZE];

  /* read header */
  std::ifstream file(name, std::ios::in | std::ios::binary);
  pnm_read(file, buf);
  if (strncmp(buf, "P4", 2))
    throw pnm_error();

  pnm_read(file, buf);
  int width = atoi(buf);
  pnm_read(file, buf);
  int height = atoi(buf);

  /* read data */
  image<uchar> *im = new image<uchar>(width, height);
  for (int i = 0; i < height; i++)
    read_packed(imPtr(im, 0, i), width, file);

  return im;
}

static void savePBM(image<uchar> *im, const char *name) {
  int width = im->width();
  int height = im->height();
  std::ofstream file(name, std::ios::out | std::ios::binary);

  file << "P4\n" << width << " " << height << "\n";
  for (int i = 0; i < height; i++)
    write_packed(imPtr(im, 0, i), width, file);
}
```

```
static image<uchar> *loadPGM(const char *name) {
  char buf[BUF_SIZE];

  /* read header */
  std::ifstream file(name, std::ios::in | std::ios::binary);
  pnm_read(file, buf);
  if (strncmp(buf, "P5", 2))
    throw pnm_error();

  pnm_read(file, buf);
  int width = atoi(buf);
  pnm_read(file, buf);
  int height = atoi(buf);

  pnm_read(file, buf);
  if (atoi(buf) > UCHAR_MAX)
    throw pnm_error();

  /* read data */
  image<uchar> *im = new image<uchar>(width, height);
  file.read((char *)imPtr(im, 0, 0), width * height * sizeof(uchar));

  return im;
}

static void savePGM(image<uchar> *im, const char *name) {
  int width = im->width();
  int height = im->height();
  std::ofstream file(name, std::ios::out | std::ios::binary);
```

```
  file << "P5\n" << width << " " << height << "\n" << UCHAR_MAX << "\n";
  file.write((char *)imPtr(im, 0, 0), width * height * sizeof(uchar));
}

static image<rgb> *loadPPM(const char *name) {
  char buf[BUF_SIZE], doc[BUF_SIZE];

  /* read header */
  std::ifstream file(name, std::ios::in | std::ios::binary);
  pnm_read(file, buf);
  if (strncmp(buf, "P6", 2))
    throw pnm_error();

  pnm_read(file, buf);
  int width = atoi(buf);
  pnm_read(file, buf);
  int height = atoi(buf);

  pnm_read(file, buf);
  if (atoi(buf) > UCHAR_MAX)
    throw pnm_error();

  /* read data */
  image<rgb> *im = new image<rgb>(width, height);
  file.read((char *)imPtr(im, 0, 0), width * height * sizeof(rgb));

  return im;
}

static void savePPM(image<rgb> *im, const char *name) {
  int width = im->width();
```

```
  int height = im->height();
  std::ofstream file(name, std::ios::out | std::ios::binary);

  file << "P6\n" << width << " " << height << "\n" << UCHAR_MAX << "\n";
  file.write((char *)imPtr(im, 0, 0), width * height * sizeof(rgb));
}
template <class T>
void load_image(image<T> **im, const char *name) {
  char buf[BUF_SIZE];
    /* read header */
  std::ifstream file(name, std::ios::in | std::ios::binary);
  pnm_read(file, buf);
  if (strncmp(buf, "VLIB", 9))
    throw pnm_error();

  pnm_read(file, buf);
  int width = atoi(buf);
  pnm_read(file, buf);
  int height = atoi(buf);

  /* read data */
  *im = new image<T>(width, height);
  file.read((char *)imPtr((*im), 0, 0), width * height * sizeof(T));
}
template <class T>
void save_image(image<T> *im, const char *name) {
  int width = im->width();
  int height = im->height();
  std::ofstream file(name, std::ios::out | std::ios::binary);

  file << "VLIB\n" << width << " " << height << "\n";
```

```
  file.write((char *)imPtr(im, 0, 0), width * height * sizeof(T));
}

#endif
```

----------edt.cpp----------

```
/* This program calculates the 3-D EDT of the image stored in the input file */
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include "pnmfile.h"
#include "imconv.h"
#include "dt.h"

int main(int argc, char **argv) {
  if (argc != 3) {
    fprintf(stderr, "usage: %s input(pbm) output(pgm)\n", argv[0]);
    return 1;
  }

  char *input_name = argv[1];
  char *output_name = argv[2];

  // load input
  image<uchar> *input = loadPBM(input_name);

  // compute dt
  image<float> *out = dt(input);

  // take square roots
```

```
  for (int y = 0; y < out->height(); y++) {
    for (int x = 0; x < out->width(); x++) {
      imRef(out, x, y) = sqrt(imRef(out, x, y));
    }
  }

  // convert to grayscale
  image<uchar> *gray = imageFLOATtoUCHAR(out);

  // save output
  savePGM(gray, output_name);

  delete input;
  delete out;
  delete gray;
}
```

----------Mpi_initmodule.h----------

```
/* begin MODULE mpi_module */
#include "mpi.h"

INT update_bc_2( INT mp, INT m, REAL **vt, INT k, INT below, INT above )
 {
        MPI_Status status[6];  /* SGI doesn't define MPI_STATUS_SIZE */

        MPI_Sendrecv( vt[mp  ]+1, m, MPI_DOUBLE, above, 0, vt[0]+1, m,
        MPI_DOUBLE, below, 0, MPI_COMM_WORLD, status );

        MPI_Sendrecv( vt[1   ]+1, m, MPI_DOUBLE, below, 1, vt[mp+1]+1, m,
        PI_DOUBLE, above, 1, MPI_COMM_WORLD, status );

         return (0);
}

/* end MODULE mpi_module */
```

----------initialize.h----------

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define CHAR char
#define REAL double
```

```
#define INT  int

#define OUTPUT stdout       /* output to standard out                  */
#define PLOT_FILE "plots"   /* output files base name                  */
#define INCREMENT 100       /* number of steps between convergence check  */

#define P 1                 /* define processor count for serial codes    */
#define K 0                 /* current thread number for serial code is 0 */
#define MAX_M 128           /* maximum size of Input Array        */
#include "utils.h"          /* header file of function prototype in utils.c */

#endif
```

----------allocate.c----------

```
main()
{
/********** MAIN PROGRAM ***********************************
 * Allocates the 3-D input array into the shared memory of cluster  *
***********************************************************/
    INT iter, m, mp;
    REAL gdel;
    CHAR line[10];
    REAL **u, **un;

    fprintf(OUTPUT,"Enter size of interior points, m :");
    (void) fgets(line, sizeof(line), stdin);
    (void) sscanf(line, "%d", &m);
    fprintf(OUTPUT,"m = %d\n",m);

    mp = m/P;
```

```
    u  = allocate_2D(m, mp);  /* allocate mem for 2D array */
    un = allocate_2D(m, mp);

    gdel = 1.0;
    iter = 0;

    bc(m, mp, u, K, P);
    replicate(m, mp, u, un);   /* u = un */

    while (gdel > TOL) {  /* iterate until error below threshold */
     iter++;            /* increment iteration counter */

     if(iter > MAXSTEPS) {
      fprintf(OUTPUT,"Iteration terminated (exceeds %6d", MAXSTEPS);
      fprintf(OUTPUT," )\n");
      return (0);       /* nonconvergent solution */
     }
/* compute new solution according to the Jacobi scheme */
     update_jacobi(m, mp, u, un, &gdel);

     if(iter%INCREMENT == 0) {
      fprintf(OUTPUT,"iter,gdel: %6d, %lf\n",iter,gdel);
     }
    }

    fprintf(OUTPUT,"Stopped at iteration %d\n",iter);
    fprintf(OUTPUT,"The maximum error = %f\n",gdel);

    write_file( m, mp, u, K, P );

    return (0);
```

```
}
```

----------util.c----------

```
REAL **allocate_2D(INT m, INT n) {
  INT i;
  REAL **a;

  a = (REAL **) malloc((unsigned) (m+2)*sizeof(REAL*));

/* Each pointer array element points to beginning of a row with n+2 entries*/
  for (i = 0; i <=m+1; i++) {
    a[i] = (REAL *) malloc((unsigned) (n+2)*sizeof(REAL));
  }

  return a;
}
INT write_file( INT m, INT n, REAL **u, INT k, INT p ) {
/*****************************************************
 * Writes 2D array ut columnwise (i.e. C convention) *
 * m   - size of rows m+2                            *
 * n   - size of columns n+2                         *
 * u   - scratch array                               *
 * k   - 0 <= k < p; = 0 for single thread code      *
 * p   - p >= 0; =1 for single thread code           *
 *****************************************************/
 INT ij, i, j, per_line;
  CHAR filename[50], file[53];
  FILE *fd;
/*
```

```
  prints u, 6 per line; used for matlab plots;
  PLOT_FILE contains the array size and number of procs;
  PLOT_FILE.(k+1) contains u pertaining to proc k;
  for serial job, PLOT_FILE.1 contains full u array.
*/

  (void) sprintf(filename, "%s", PLOT_FILE);

  if ( k == 0 ) {
    fd = fopen(filename, "w");
    fprintf(fd, "%5d %5d %5d\n", m+2, n+2, p);
    fclose(fd);
  }

  per_line = 6;                    /* to print 6 per line */
  (void) sprintf(file, "%s.%d", filename, k); /* create output file */
  fd = fopen(file, "w");
  ij = 0;
  for (j = 0; j <=n+1; j++) {
    for (i = 0; i <=m+1; i++) {
      fprintf(fd, "%11.4f ", u[i][j]);
      if ((ij+1)%per_line == 0) fprintf(fd, "\n");
      ij++;
    }
  }
  fprintf(fd, "\n");
  fclose(fd);
  return (0);
}
```

```
void init_array(INT m, INT n, REAL **a) {
/********* Initialize Array *********************
 * Initialize array with nx rows and ny columns  *
 ************************************************/
 INT i, j;

 for (i = 0; i <=m+1; i++) {
  for (j = 0; j <=n+1; j++) {
   a[i][j] = 0.0;          /* initialize all entries to zero */
  }
 }
}

void bc(INT m, INT n, REAL **u, INT k, INT p)
{
 INT i;
 init_array( m, n, u);                    /* initialize u to 0 */

 if (p > 1) {
  if (k == 0) {
   for (i = 0; i <=m+1; i++) {
    u[i][0] = sin(PI*i/(m+1));            /* at y = 0; all x */
   }
  }
  if (k == p-1) {
   for (i = 0; i <=m+1; i++) {
    u[i][n+1] = sin(PI*i/(m+1))*exp(-PI);  /* at y = 1; all x */
   }
  }
 } else if (p == 1) {
   for (i = 0; i <=m+1; i++) {
```

```
      u[i][  0] = sin(PI*i/(m+1));    /* at y = 0; all x */
      u[i][n+1] = u[i][0]*exp(-PI);   /* at y = 1; all x */
    }
  } else {
   printf("p is invalid\n");
  }
}

void prtarray( INT m, INT n, REAL **a, FILE *fd) {
/********** Print Array **********************
 * Prints array "a" with m rows and n columns  *
 * tda is the Trailing Dimension of Array a    *
 ***********************************************/
 INT i, j;
 for (i = 0; i <=m+1; i++) {
   for (j = 0; j <=n+1; j++) {
     fprintf(fd, "%8.2f", a[i][j]);
   }
   fprintf(fd, "\n");
 }
}

 INT i, j;
 *del = 0.0;
 for (i = 1; i <=m; i++) {
   for (j = 1; j <=n; j++) {
     unew[i][j] = ( u[i  ][j+1] + u[i+1][j  ] +
                    u[i-1][j  ] + u[i  ][j-1] )*0.25;
     *del += fabs(unew[i][j] - u[i][j]);    /* find local max error */
   }
 }
```

```
  for (i = 1; i <=m; i++) {
    for (j = 1; j <=n; j++) {
      u[i][j] = unew[i][j];
    }
  }

  return (0);
}

INT update_sor( INT m, INT n, REAL **u, REAL omega, REAL *del, CHAR redblack)
{
  INT i, ib, ie, j, jb, je;
  REAL up;

     *del = 0.0;
     if (redblack == 'r') {
/* process RED odd points ... */
       jb = 1; je = n; ib = 1; ie = m;
       for ( j = jb; j <= je; j+=2 ) {
         for ( i = ib; i <=ie; i+=2 ) {
           up = ( u[i  ][j+1] + u[i+1][j  ] +
                  u[i-1][j  ] + u[i  ][j-1] )*0.25;
           u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
           *del += fabs(up-u[i][j]);
         }
       }
/* process RED even points ... */
       jb = 2; je = n; ib = 2; ie = m;
       for ( j = jb; j <= je; j+=2 ) {
         for ( i = ib; i <= ie; i+=2 ) {
           up = ( u[i  ][j+1] + u[i+1][j  ] +
```

```
            u[i-1][j  ] + u[i  ][j-1] )*0.25;
          u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
          *del += fabs(up-u[i][j]);
        }
       }
       return (0);
     } else {
       if (redblack == 'b') {
/* process BLACK odd points ... */
         jb = 2; je = n; ib = 1; ie = m;
         for ( j = jb; j <= je; j+=2 ) {
          for ( i = ib; i <= ie; i+=2 ) {
           up = ( u[i  ][j+1] + u[i+1][j  ] +
                u[i-1][j  ] + u[i  ][j-1] )*0.25;
           u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
           *del += fabs(up-u[i][j]);
          }
         }
/* process BLACK even points ... */
         jb = 1; je = n; ib = 2; ie = m;
         for ( j = jb; j <= je; j+=2 ) {
          for ( i = ib; i <= ie; i+=2 ) {
           up = ( u[i  ][j+1] + u[i+1][j  ] +
                u[i-1][j  ] + u[i  ][j-1] )*0.25;
           u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
           *del += fabs(up-u[i][j]);
          }
         }
         return (0);
       } else {
         return (1);
```

```
      }
    }
}

INT replicate( INT m, INT n, REAL **a, REAL **b ) {
/*********************************************************
 * Replicates array a into array b                       *
 * m   - (INPUT)  size of interior points in 1st index *
 * n   - (INPUT)  size of interior points in 2st index *
 * a   - (INPUT)  solution at time N                     *
 * b   - (OUTPUT) solution at time N + 1                 *
 *********************************************************/
 INT i, j;

 for (i = 0; i <=m+1; i++) {
  for (j = 0; j <=n+1; j++) {
   b[i][j] = a[i][j];
  }
 }
 return (0);
}

INT transpose( INT m, INT n, REAL **a, REAL **at ) {
/*********************************************************
 * Transpose a(0:m+1,0:n+1) into at(0:n+1,0:m+1)         *
 * m   - (INPUT)  size of interior points in 1st index   *
 * n   - (INPUT)  size of interior points in 2st index   *
 * a   - (INPUT)  a  =  a(0:m+1,0:n+1)                   *
 * at  - (OUTPUT) at = at(0:n+1,0:m+1)                   *
 *********************************************************/
 INT i, j,k;
```

```
 for (i = 0; i <=m+1; i++) {
  for (j = 0; j <=n+1; j++) {
       (k = 0; k <=n+1; k++) {
    at[j][i][k] = a[i][j][k];
  }
 }
 return (0);
}

void neighbors(INT k, INT p, INT UNDEFINED, INT *below, INT *above) {
/*************************************************************
 * determines two adjacent threads                           *
 * k         - (INPUT)  current thread                       *
 * p         - (INPUT)  number of processes (threads)        *
 * UNDEFINED - (INPUT)  code to assign to out-of-bound neighbor *
 * below     - (OUTPUT) neighbor thread below k (usually k-1)  *
 * above     - (OUTPUT) neighbor thread above k (usually k+1)  *
 *************************************************************/
 if(k == 0) {
  *below = UNDEFINED;    /* tells MPI not to perform send/recv */
  *above = k+1;
 } else if(k == p-1) {
  *below = k-1;
  *above = UNDEFINED;    /* tells MPI not to perform send/recv */
 } else {
  *below = k-1;
  *above = k+1;
 }
}
```

```
INT main(INT argc, CHAR *argv[]) {
/***************MAIN PROGRAM ************************************
 * Gathers all the computation results from participating processors and builds the final
EDT of input 3-D array *
 ************************************************************/
    INT iter, m, mp, p, k, below, above;
    REAL omega, rhoj, rhojsq, del, delr, delb, gdel;
    CHAR line[80], red, black;
    MPI_Comm grid_comm;
    INT me, iv, coord[1], dims, periods, ndim, reorder;
    REAL **v, **vt;

    MPI_Init(&argc, &argv);          /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &k);  /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);  /* get # procs from env or */

    periods = 0; ndim = 1; reorder = 0; red = 'r'; black = 'b';

    if(k == 0) {
     fprintf(OUTPUT,"Enter size of interior points, m :\n");
     (void) fgets(line, sizeof(line), stdin);
     (void) sscanf(line, "%d", &m);
     fprintf(OUTPUT,"m = %d\n",m);
    }
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    mp = m/p;
 MPI_Sendrecv( vt[mp  ]+1, m, MPI_DOUBLE, above, 0,
            vt[0   ]+1, m, MPI_DOUBLE, below, 0,
            MPI_COMM_WORLD, status );
```

```
    v  = allocate_2D(m, mp);  /* allocate mem for 2D array */
    vt = allocate_2D(mp, m);

    gdel = 1.0;
    iter = 0;
    rhoj = 1.0 - PI*PI*0.5/((m+2)*(m+2));
    rhojsq = rhoj*rhoj;

/* create cartesian topology for matrix */
    dims = p;
    MPI_Cart_create(MPI_COMM_WORLD, ndim, &dims,
          &periods, reorder, &grid_comm);
    MPI_Comm_rank(grid_comm, &me);
    MPI_Cart_coords(grid_comm, me, ndim, coord);
    iv = coord[0];
    bc( m, mp, v, iv, p);     /* set up boundary conditions */
    transpose(m, mp, v, vt);  /* transpose v into vt */

    replicate(mp, m, vt, v);
    MPI_Cart_shift(grid_comm, 0, 1, &below, &above);
        MPI_Sendrecv( vt[1   ]+1, m, MPI_DOUBLE, below, 1,
            vt[mp+1]+1, m, MPI_DOUBLE, above, 1,
            MPI_COMM_WORLD, status );
    omega = 1.0;
    update_sor( mp, m, vt, omega, &delr, red);
    update_bc_2( mp, m, vt, iv, below, above);
    omega = 1.0/(1.0 - 0.50*rhojsq);
    update_sor( mp, m, vt, omega, &delb, black);
    update_bc_2( mp, m, vt, iv, below, above);
    while (gdel > TOL) {
     iter++;    /* increment iteration counter */
```

```
      omega = 1.0/(1.0 - 0.25*rhojsq*omega);
      update_sor( mp, m, vt, omega, &delr, red);
      update_bc_2( mp, m, vt, iv, below, above);
      omega = 1.0/(1.0 - 0.25*rhojsq*omega);
      update_sor( mp, m, vt, omega, &delb, black);
      update_bc_2( mp, m, vt, iv, below, above);
      if(iter%INCREMENT == 0) {
       del = (delr + delb)*4.0;
       MPI_Allreduce( &del, &gdel, 1, MPI_DOUBLE,
          MPI_MAX, MPI_COMM_WORLD);   /*  find global max error */
       if (k == 0) {
        fprintf(OUTPUT,"iter gdel omega: %5d %13.5f %13.5f\n",iter,gdel,omega);
       }
      }

MPI_Sendrecv( vt[mp  ]+1, m, MPI_DOUBLE, above, 0, vt[0 ]+1, m, MPI_DOUBLE,
below, 0, MPI_COMM_WORLD, status );
      if(iter > MAXSTEPS) {
       fprintf(OUTPUT,"Iteration terminated (exceeds %6d", MAXSTEPS);
       fprintf(OUTPUT," )\n");
       return (1);              /* nonconvergent solution */
      }
    }

MPI_Sendrecv( vt[1   ]+1, m, MPI_DOUBLE, below, 1,
          vt[mp+1]+1, m, MPI_DOUBLE, above, 1,
          MPI_COMM_WORLD, status );
    if (k == 0) {
      fprintf(OUTPUT,"Stopped at iteration %d\n",iter);
      fprintf(OUTPUT,"The maximum error = %f\n",gdel);
    }
```

```
    transpose(mp, m, vt, v);  /* transpose v into vt */
    write_file( m, mp, v, k, p);

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();

    return (0);
}
```