NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A225 399

DTIC
ELECTE
AUG 17 1990
S B D

THESIS

SOLVING THE WEIGHTED REGION
LEAST COST PATH PROBLEM
USING TRANSPUTERS

by

Ivan Garcia

December 1989

Thesis Advisor: Man-Tak Shing

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Field	Content
1a. REPORT SECURITY CLASSIFICATION	UNCLASSIFIED
1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
3. DISTRIBUTION / AVAILABILITY OF REPORT	Approved for public release; distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	
5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION	Naval Postgraduate School
6b. OFFICE SYMBOL (If applicable)	Code 52
7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code)	Monterey, CA 93943-5000
7b. ADDRESS (City, State, and ZIP Code)	
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	
8b. OFFICE SYMBOL (If applicable)	
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)	
10. SOURCE OF FUNDING NUMBERS	PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification)	SOLVING THE WEIGHTED REGION LEAST COST PATH PROBLEM USING TRANSPUTERS (UNCLASSIFIED)
12. PERSONAL AUTHOR(S)	Garcia, Ivan
13a. TYPE OF REPORT	Master's Thesis
13b. TIME COVERED	FROM ____ TO ____
14. DATE OF REPORT (Year, Month, Day)	December 1989
15. PAGE COUNT	73
16. SUPPLEMENTARY NOTATION	The views expressed in this thesis are those of the author and do not reflect the official policy of the Department of Defense or the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Path planning, Weighted regions, Transputers, Distributed computing, Parallel processing, Parallel algorithms, Processor farm.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The weighted region least cost path problem involves finding the minimal cost path between a source point and a goal point through a plane that has been subdivided into weighted regions. In this thesis, we investigate a new parallel approach which seeks to take advantage of the distributed, asynchronous computing environment provided by the INMOS Transputer. The algorithm consists of a family of local, asynchronous, iterative and parallel procedures. The program is implemented on a network of transputers using a parallel version of the C programming language and tested on various maps of triangulated regions. Results were favorable in terms of producing a near optimum path and reduced processing time.

Field	Content
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS
21. ABSTRACT SECURITY CLASSIFICATION	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	Man-Tak Shing
22b. TELEPHONE (Include Area Code)	(408)646-2634
22c. OFFICE SYMBOL	Code 52Sh

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-

Approved for public release; distribution is unlimited.

Solving the Weighted Region Least Cost Path Problem Using Transputers

by

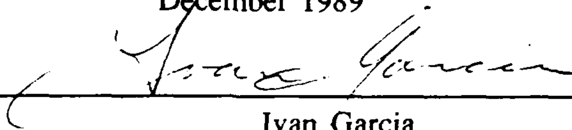
Ivan Garcia
Captain, United States Marine Corps
B.S., University of New Mexico, 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

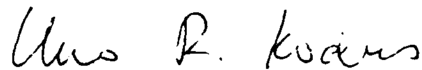
NAVAL POSTGRADUATE SCHOOL
December 1989

Author: 

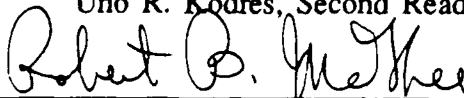
Ivan Garcia

Approved by:

Man-Tak Shing, Thesis Advisor



Uno R. Kodres, Second Reader



Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

The weighted region least cost path problem involves finding the minimal cost path between a source point and a goal point through a plane that has been subdivided into weighted regions. In this thesis, we investigate a new parallel approach which seeks to take advantage of the distributed, asynchronous computing environment provided by the INMOS Transputer. The algorithm consists of a family of local, asynchronous, iterative and parallel procedures. The program is implemented on a network of transputers using a parallel version of the C programming language and tested on various maps of triangulated weighted regions. Results were favorable in terms of producing a near optimum path and reduced processing time.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

TABLE OF CONTENTS

I. INTRODUCTION 1

II. BACKGROUND 3

A. PRELIMINARIES 3

1. Dijkstra's Algorithm 3

2. Snell's Law of Refraction 3

3. Mapping Terrain 6

B. PAST RESEARCH 9

C. THE INMOS TRANSPUTER 10

1. Transputer Architecture 11

2. Communicating Sequential Processes 15

III. THE LAIPPS ALGORITHM 16

A. PARALLELIZATION 16

B. BASIC LAIPPS PROCEDURES 18

1. General Description of the Procedures 18

2. Formal Description of the Procedure 21

IV. EXPERIMENT DESCRIPTION AND RESULTS 25

A. HARDWARE CONFIGURATION FOR THE EXPERIMENT 25

B. PROGRAMMING LANGUAGE 26

1. The OCCAM Programming Language 26

2. 3L's Parallel C Programming Language 27

C. PROCESSOR FARM APPROACH TO PARALLELIZATION 28

D. PARTITIONING THE PROBLEM 29

1. The Master Task 30

2. The Worker Task 33

E. EXPERIMENTAL DESIGN AND RESULTS 33

1. Initial Conditions and Convergence Criteria 33

2. Experiment Description and Results 34

V. CONCLUSIONS AND RECOMMENDATIONS 41

A. CONCLUSIONS 41

B. RECOMMENDATIONS FOR FURTHER RESEARCH 41

APPENDIX A - HEADER FILE SOURCE CODE 43

APPENDIX B - QUEUE DATA STRUCTURE SOURCE CODE 45

APPENDIX C - MASTER TASK SOURCE CODE 47

APPENDIX D - WORKER TASK SOURCE CODE 59

LIST OF REFERENCES 64

INITIAL DISTRIBUTION LIST 66

I. INTRODUCTION

The Weighted Region Least Cost Path problem involves finding the minimal cost path between a source point and a goal point through a plane that has been subdivided into weighted regions. For the purposes of this thesis, the cost of a line segment through a region will be considered to be the product of the Euclidian distance of the line segment and the unit cost incurred while traversing that region, and the cost of a path equals the sum of the cost of the line segments forming the path. The least cost path problem is of significant importance in the field of robotics where navigation through varied terrain is a requirement and thus is of continuing interest to researchers. In this thesis, we first introduce the principles on which most of the current research is based and survey various approaches which have been taken to solve the least cost path problem. Then we investigate a new parallel approach which seeks to take advantage of the distributed, asynchronous computing environment provided by the INMOS *Transputer*.

The algorithm which is implemented in this thesis was developed by Smith, Peng, and Gahinet [Ref. 1]. It consists of a family of local, asynchronous, iterative and parallel procedures (LAIPPs) which are designed to solve the least cost path problem for a triangulated plane. We have implemented the LAIPPs on a network of transputers using a parallel version of the C programming language, 3L's Parallel C, and studied the performance of those procedures empirically. In [Ref. 1:p. 2], Smith

et al have only developed and tested the LAIPPs using procedural parallelization on a single processor, our goal here is to ascertain whether the LAIPPs are in fact implementable on a multiprocessor system operating in parallel.

The remainder of this thesis is organized as follows. Chapter II introduces the reader to background information and the underlying theory upon which the least cost path algorithms presented here are based. Several approaches for solving the least cost path problem are then presented, followed by a description of the transputer network on which the LAIPPs are implemented. In Chapter III, a description of the LAIPPs algorithm implemented in this thesis will be provided. Chapter IV provides a description of the actual procedures and tools used to realize the implementation as well as the results of the experiment. Conclusions and recommendations for further research are offered in Chapter V.

II. BACKGROUND

A. PRELIMINARIES

In this section, we introduce the basic principles on which most of the existing approaches are based.

1. Dijkstra's Algorithm

The approach taken by most algorithms for solving the least cost path problem presented in this thesis are based on an algorithm developed by Dijkstra [Ref. 2] in 1959. The algorithm, as described by Aho, Hopcroft and Ullman in [Ref. 3:pp. 204,205] is presented in Figure 1.

2. Snell's Law of Refraction

The second principle which underlies all of the algorithms described in this thesis is Snell's Law of Refraction. This principle is utilized because it is a means by which local optimality can be achieved. Snell's Law states that the path of a ray of light passing through a boundary, *i*, between regions c and c' with indices of refraction a_c and $a_{c'}$ obeys the relationship that

$$a_c \sin \theta = a_{c'} \sin \theta'$$

where θ and θ' are the angles of incidence and refraction respectively (Figure 2). Snell's Law is implied by Fermat's Principle of optics which states that light follows the path of minimum time between two points. The index of refraction for a region

```
PROCEDURE Dijkstra;

{ Let 1 be the source vertex and C[i,j] be the length of the edge
joining vertices i and j.

    The procedure works by maintaining a set S of vertices whose shortest distance
from the source is already known. Initially, S contains only the source vertex. At each
step, we add to S a remaining vertex v whose distance from the source is as short as
possible. Assuming all arcs have nonnegative costs, we can always find a shortest path
from the source to v that passes only through vertices in the set S. Call such a path
special. At each step of the algorithm, we use an array D to record the length of the
shortest path to each vertex. Once S includes all vertices, all paths are "special", so
D will hold the shortest distance from the source to each vertex.
}

BEGIN
    S := {1};

    FOR (i := 2 TO n) DO
        D[i] := C[1,i];  { initialize D }

    FOR (i := 1 TO n-1) DO BEGIN
        choose a vertex w in V-S such that
            D[w] is a minimum;
        add w to S;

        FOR (each vertex v in V-S) DO
            D[v] := min(D[v], D[w] + C[w,v]);
    END;
END;  { Dijkstra }
```

Figure 1. Dijkstra's Algorithm

is proportional to the speed with which light can travel through that region. Therefore by treating regions as optical media and the cost index of a region like the reciprocal of the speed of light through the region, the principles of optics, as defined by Snell's Law and Fermat's Principle, can be applied to solve the least cost path problem.

Let us consider two cases where the least cost path between two points, s and p, crosses an edge i. The first case involves finding the least cost path where the

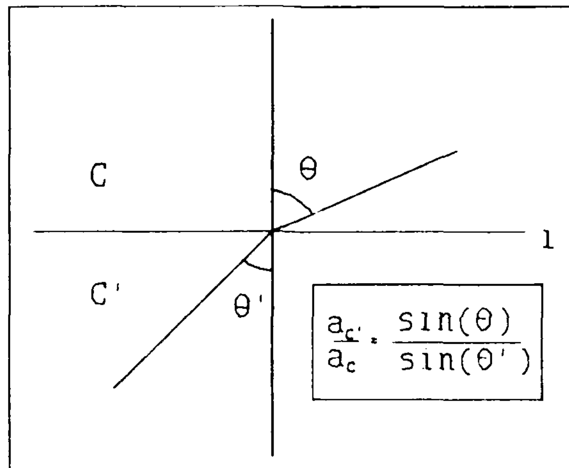


Figure 2. Snell's Law of Refraction

optimal crossing point on an edge is unconstrained. As shown in Figure 3, Snell's Law holds for this case. Now consider the case where there is a constraint for the optimal crossing point to lie within a subsegment of the edge (refer to Figure 4). In this case it is clear that if the unconstrained optimal path lies outside the constraint points, the locally optimal path must then pass through the nearest constraint point. Given the high likelihood of encountering edge constraints within an arbitrarily

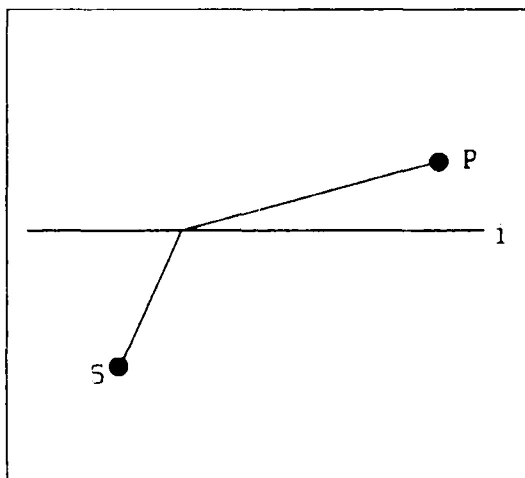


Figure 3. Unconstrained Path

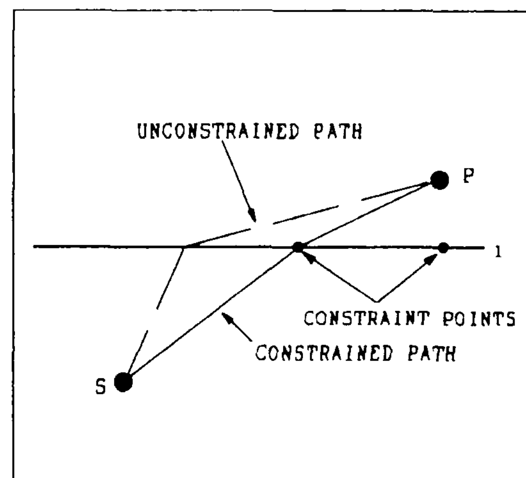


Figure 4. Constrained Path

triangulated plane, the second generalization of Snell's Law is incorporated into the procedures developed in [Ref. 1:p. 5].

3. Mapping Terrain

Several methods can be used to model the terrain to be traversed. We will examine three such models and explain why the weighted region has been chosen as the model for the least cost path problem. One approach, called Dijkstra's wavefront propagation technique, to solving the least cost path problem has been to make the problem discrete by modelling the terrain to be traversed using a grid graph [Ref. 4:pp. 174-176]. A grid graph is produced by simply laying a grid over the terrain to be traversed. The resulting squares are then treated as pixels with the fineness of the grid determined by the length of the side of the squares. Dijkstra's algorithm [Ref. 2] is then used to find minimum cost paths in the grid graph. There are two drawbacks to this approach. First, to accurately capture the content of a simple map may require an extremely fine grid. Second, the requirement that the paths remain on the grid graph introduces digitization bias. This is illustrated in Figures 5 and 6 where it is plainly visible that the requirement to remain on the grid in the digitized map shown in Figure 5 results in a path which is longer than the optimal path shown in Figure 6.

A second approach is to build a graph in which regions have a corresponding node and region boundaries correspond to edges. By placing nodes at the center of the regions and then connecting the nodes of adjacent regions with an edge, we can then assign costs to these edges based on the weighted distance between the nodes (Figure 7). A search of this graph for the shortest paths results in the

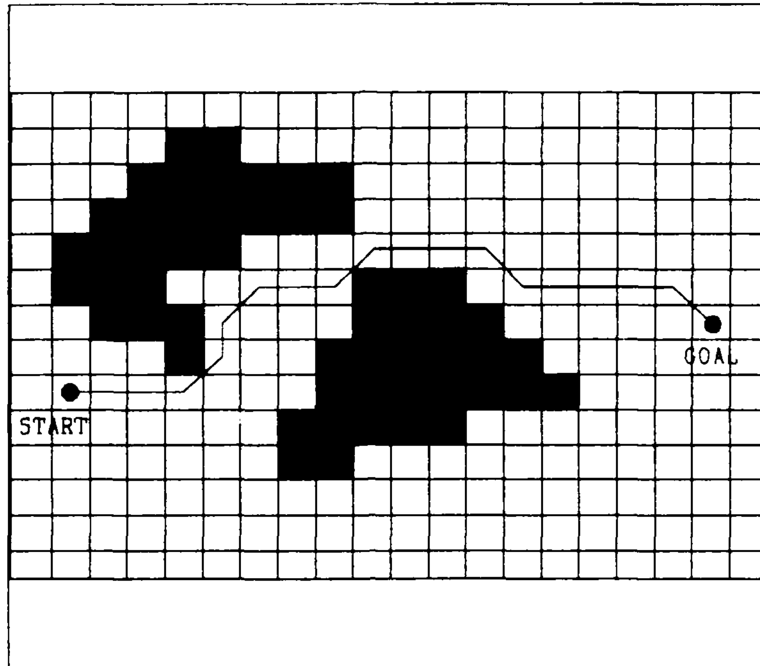


Figure 5. Path Between Obstacles Showing Digitization Bias

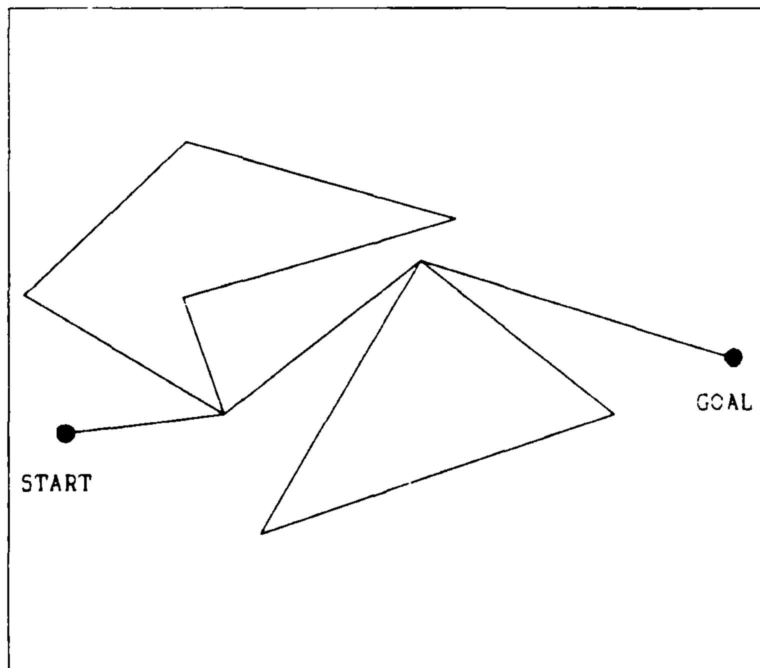


Figure 6. Optimal Path Between Obstacles

production of a "region path" from the source to the destination. A region path being a sequence of regions through which a good path should pass. Some post processing is then performed to make the path locally optimal at the region boundaries. However, this approach does have a major drawback, in that the shortest region path does not have any relationship to the global optimal path and hence there is no guarantee that the resulting shortest region path is globally optimal, or that it even approaches the global optimal path.

The research presented in this thesis makes use of a terrain map, or plane, which has been divided into regions. Each region is associated with a weight that corresponds to the unit cost of traversing through the region. The map is divided into

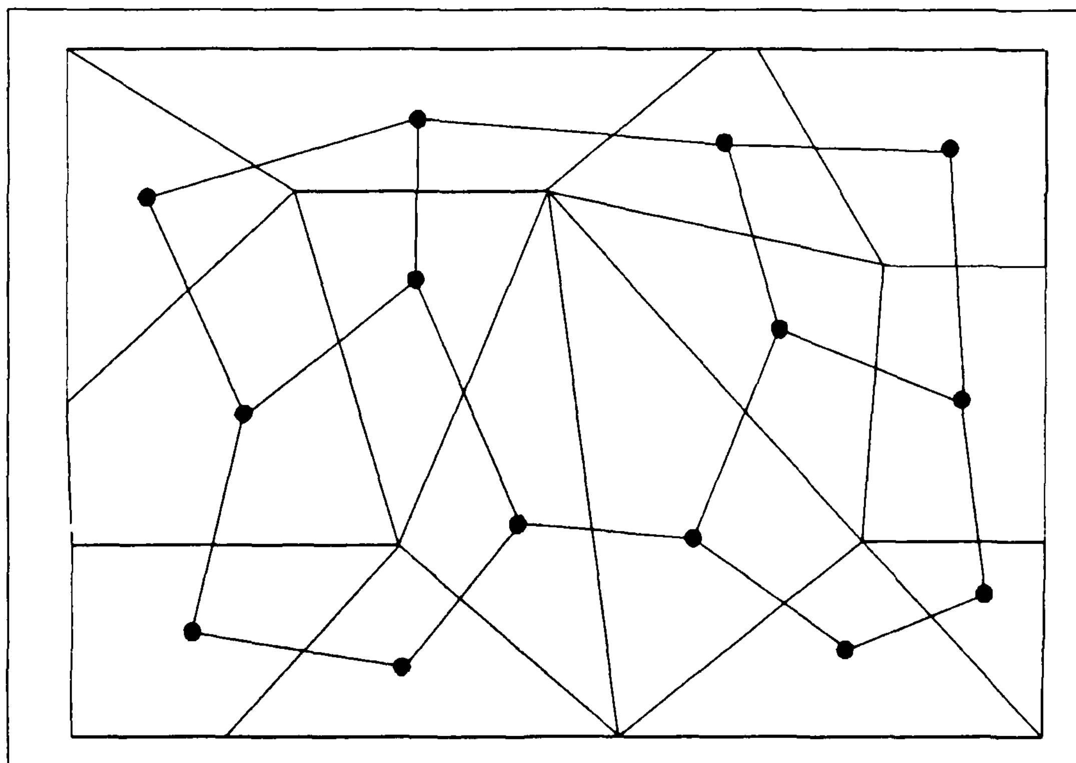


Figure 7. Region Graph

regions according to the type of ground cover or an index based on the difficulty or risk of traversing through the region. Solving the least cost path problem using this type of mapping was introduced in [Ref. 5] (discussed below) and is referred to as the weighted region problem.

B. PAST RESEARCH

Mitchell and Papadimitriou [Ref. 5] have proposed a sequential algorithm which finds paths that lie within a user specified percentage tolerance ε of the globally optimal path. This algorithm uses what the authors call a continuous Dijkstra technique to solve the weighted region problem. This technique imagines a "signal" emanating from the source across the adjacent region to the edges bordering that region. The signal then propagates further from any point it reaches, propagation taking place only when the first signal is received. The first time a signal reaches a point x, it is marked with the time it received the signal; this equates to the minimum distance from the source to x. Analysis of this algorithm shows that it runs in $O(n^7L)$ and requires space $O(n^3)$, where n is the number of boundary edges in the subdivided plane, and L is the precision of the problem instance.

Although the foregoing algorithm provides a polynomial worst case time complexity, it is not useful in practice and has never been implemented. In [Ref. 6] Richbourg, Rowe, Zyda and McGhee propose an A* search strategy whose performance is much better for the average case. They also show that this search strategy requires less computation than other procedures based on Dijkstra's wave front propagation techniques.

Mitchell [Ref. 7] also considered a special case of the weighted region problem where the cost for traversing a region f, α_f, is taken from the set $\{0,1,+\infty\}$, while the cost of traversing along a boundary i, α_i, may take on any nonnegative number and the cost of crossing an edge, ξ_i, can be either 0 or $+\infty$ (where $\xi_i = +\infty$, implies that i is an obstacle). The approach taken by Mitchell, which is a combination of the concept of the visibility graph and the local optimality properties of shortest paths, provides an exact solution to the problem in polynomial time.

Smith, Peng and Gahinet [Ref. 1] presented a family of local, asynchronous, iterative and parallel procedures (LAIPP) which find least cost paths through a triangulated plane. This approach is an attempt to make parallel the computation of the locally optimal paths by placing these LAIPPs on a network of processors. Each processor is associated with a boundary edge on the plane and each processor is only allowed to communicate with its four neighbors in order to exchange information about the location of the points where the least cost path crosses the edge. This is the algorithm on which work for this thesis is based and it will be more fully explained in Chapter III.

C. THE INMOS TRANSPUTER

The Aegis Modelling Project in the Computer Science Department of the Naval Postgraduate School is engaged in researching advanced computer architectures. Current research in the Project is centered on the application and evaluation of distributed computing architectures using a product of INMOS Ltd. called *Transputer*. Because the Transputer is already available for use and because it is specifically

designed for use in a distributed computing environment, it is chosen as the hardware on which the experiment is to be implemented.

A Transputer is a microprocessor that contains its own local memory and four communication links on a single chip. The links provide the capability for point to point connections between Transputers. The Transputer was designed specifically to implement the concept of communicating sequential processes (CSP) defined by C.A.R. Hoare [Ref. 8] and to be used as a building block for distributed computing systems. The CSP concept describes the interactions between programs that execute in parallel and will be described below. Although the Transputer is available in several different types, this discussion will be confined to the T414 and the T800 families. The reader is directed to [Ref. 9] for a more thorough description of Transputers.

1. *Transputer Architecture*

Both the T414 and T800 Transputers consist of a single chip processor which is composed of a 32-bit microprocessor, four communication links with direct memory access controller (DMA), 2 Kbytes of on-chip random access memory (RAM), a configurable memory interface, and two timers (refer to Figure 8). In addition, the T800 Transputer has a 64-bit floating point unit and an additional 2 Kbytes of on-chip memory for a total of 4 Kbytes of RAM.

The central processor consists of a 32-bit arithmetic logic unit which is controlled by a microcoded sequence controller. It utilizes six registers, three of which form a push down evaluation stack, a register which points to the next instruction to be executed in the running process, another which points to the beginning of the data

for the currently running process and an operand register. The processor is able to execute several concurrent processes via time slicing. Process switching does not degrade the performance of the Transputer as the entire switching operation is completed in less than one microsecond. Processes which are waiting for communication or a timer do not consume any processor time. And finally, processes can be run at one of the two priority levels.

There are four communication links in the Transputer. A DMA block transfer mechanism is used to transfer messages between memory and other Transputers. With a link speed of 20 Mbits per second, the T800 links are capable of transmitting unidirectionally at a rate of 1.7 Mbytes per second and at a rate of 2.3 Mbytes per second bidirectionally. Because the link interfaces and the central processor are able to operate concurrently, data can be transmitted on all links while processing continues.

The static on-chip RAM is accessible by both the central processor and the communication links at a maximum data rate of up to 80 Mbytes per second. Although the Transputer has only 2 Kbytes of on-chip memory, it is capable of addressing and directly accessing up to 4 Gbytes of linear address space that appears as one contiguous block. Memory off chip is accessible at a maximum rate of 26 Mbytes per second, all timing, control and dynamic RAM refresh signals being provided by a configurable memory controller.

As previously mentioned, the Transputer is able to support the running of concurrent processes. This is accomplished by a scheduler which maintains a list of active processes for each priority level and a timer for each of the two priority levels.

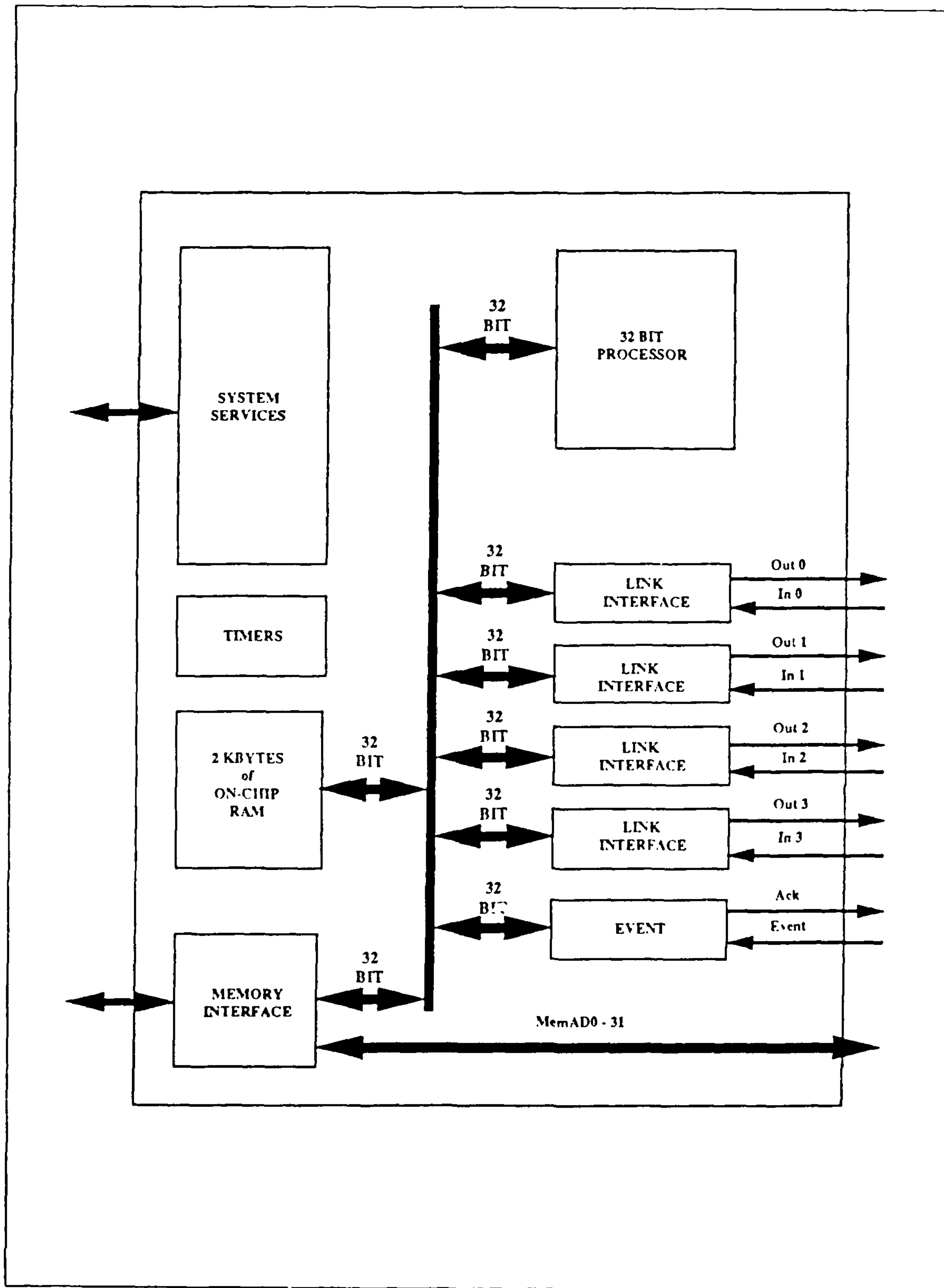


Figure 8. T414 Transputer Architecture

An active process is one that is either executing or ready to execute. An inactive process is one which is either waiting for some communication to complete or waiting for a programmed time delay. Selection of the next process to run is based on the following rules:

- High priority processes run to completion or until they must wait for communication or a programmed time delay. If a low priority process is executing when a high priority process is added to the high priority active process list, the low priority process is preempted by the high priority process. [Ref.9:p. 36]

- Low priority processes will execute only when there are no active high priority processes. Unlike high priority processes however, low priority processes do not execute until completion. Instead they execute until they must wait for a communication, wait for a programmed time delay, or the process has been running for between one and two timeslice periods. Each timeslice lasts for 5120 clock cycles (about 1 millisecond at the 5 MHz clock frequency). A process that is preempted after using up its timeslice is placed back in the low priority list. [Ref. 9:p. 37]

Note that the intention of having two priority levels for processes is to allow those high priority tasks, which must be executed when they are invoked, to preempt a currently executing low priority process and execute to completion. It is important that the high priority tasks have a very short execution time (less than one timeslice period). Otherwise the low priority processes, which should be the computation intensive processes, will not be given fair access to the processor.

The floating point unit on the T800 is a 64-bit floating point unit that adheres to the ANSI-IEEE 754-1985 floating point standard. It is able to perform operations concurrently with and under the control of the central processor. Three floating point numbers in IEEE format can be manipulated on a three deep evaluation

stack internal to the floating point unit. All data transfers between the floating point unit and memory are performed under the control of the central processor.

2. Communicating Sequential Processes

Parallel programming by its very nature implies that there are concurrently executing processes. These processes can themselves be composed of collections of processes which may have internal concurrency. The Transputer directly implements the concept of communicating sequential processes. Under this model, a program is viewed as a collection of processes (called tasks in Parallel C). A process is a sequence of instructions which has its own disjoint data space and can communicate with other processes. However, interprocess communication can only occur by message passing using explicit communication channels.

In order for the concurrent processes to communicate, message passing must be synchronized. This is achieved by having a process which needs to communicate, explicitly name the receiving process as the destination of its output and the receiving process must in turn name the sending process as the source for its input. This allows the value to be output by the source process to be copied into the destination process. Note that the synchronization imposes a requirement that an output (input) command must be delayed until the corresponding input (output) command in the other process is ready to be executed.

III. THE LAIPPS ALGORITHM

The work done in this thesis is based upon the procedures presented by Smith *et al* in [Ref. 1]. Recall that these procedures defined the method by which the weighted region least cost path problem could be solved by partitioning the problem in such a manner that the computations could then be performed in parallel and asynchronously. Here, we will present in detail the approach taken and the assumptions made by Smith *et al* to achieve parallelization.

A. PARALLELIZATION

The most basic assumption made in [Ref. 1] was that of defining the shapes of the weighted regions. In order to allow the simple development of LAIPPs, the shape of the regions is assumed to be triangular and each region is associated with a unit cost for traversing that region. Such an assumption is not too restrictive since we can easily convert any planar partition into a triangulated one by adding non-intersecting diagonals to the regions. The following assumptions are also made:

- each edge of every triangle in the plane has a processor associated with it (refer to Figure 9).

- each processor can only communicate with its immediate neighbors (refer to Figure 9).

- processors execute asynchronously until a termination condition is met.

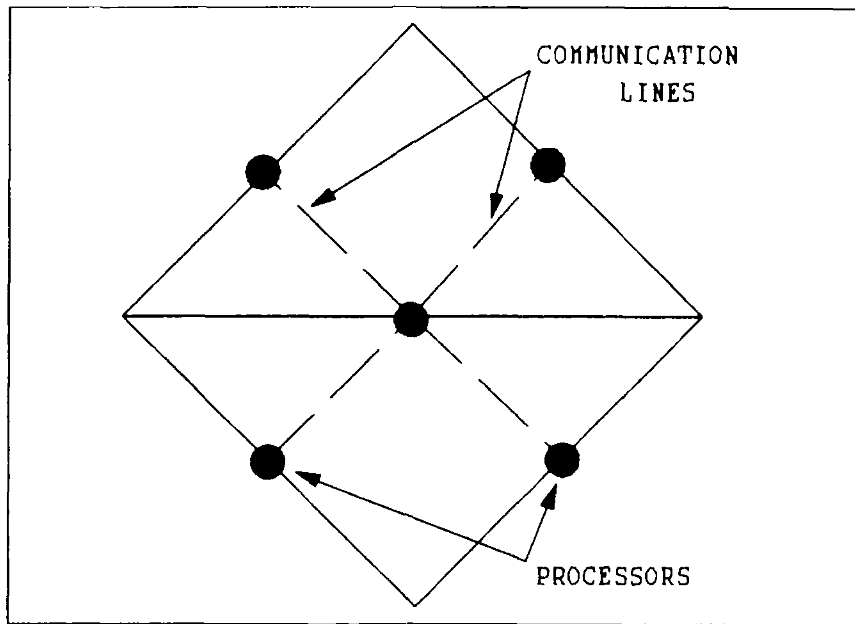


Figure 9. Processor Placement

- processors perform the computation of a relatively simple task

By assuming that each processor has been mapped exclusively to an unique edge, it can be shown that each processor can compute a minimum value:

$$C(i) = C(j) + c_{ij}$$

where j is one of the immediate neighbors of edge i, $C(j)$ is an estimated cost of reaching edge j from the start point s, and c_{ij} is the cost of reaching i from j. Computing this cost, $C(i)$, for every edge i, a global least cost path can be found from the start point, s, to any edge i, $i = 1,...,n$ in a finite number of iterations.

In order to compute a weighted region least cost path using the above procedures, each processor must be able to maintain a record of:

- the locations of a fixed number of points through which paths linking the start point with the destination point cross the associated edge. These we shall call *path points*.

- The minimal costs of reaching each of its path points from both the start point and the destination point. These costs are termed the s-values and the e-values, respectively, of a given path point. The summation of tl .se two costs provides an estimate of the cost associated with a path from s to e via the corresponding path point. This estimate is referred to as p-values of the corresponding path point.

B. BASIC LAIPPS PROCEDURES

1. General Description of the Procedures

The basic procedures described in [Ref. 1] keep track of four path points on each edge. It should be noted that these procedures do not find paths which incorporate reentrant critical angle crossings like the one shown in Figure 10. Figure 11 is a graphic representation of a given edge i its immediate neighbors and the corresponding path points. There are two basic heuristics on which the LAIPPs are based:

- An optimal path from s to e passing through edge i must go through one of the four pairs of edges (k,l), (k,m), (l,n) or (m,n).

- By having each edge processor apply local procedures iteratively and asynchronously, an improvement to the best path from s to e should be produced.

Hence, processor i uses a total of four path points to keep track of the four paths from s to e that cross edge i and the four edge pairs (k,l), (k,m), (l,n) and (m,n). Inputs to the processor on edge i are provided by the four processors associated with each of its immediate neighbors. Initial locations, s- and e-values of the four path points, p_{x_j} $(x \in \{k,l,m,n\}$, j=1,...,4), are known to each processor prior to starting the LAIPPs. (See Section IV.E for details.) Based on the path points on i's four

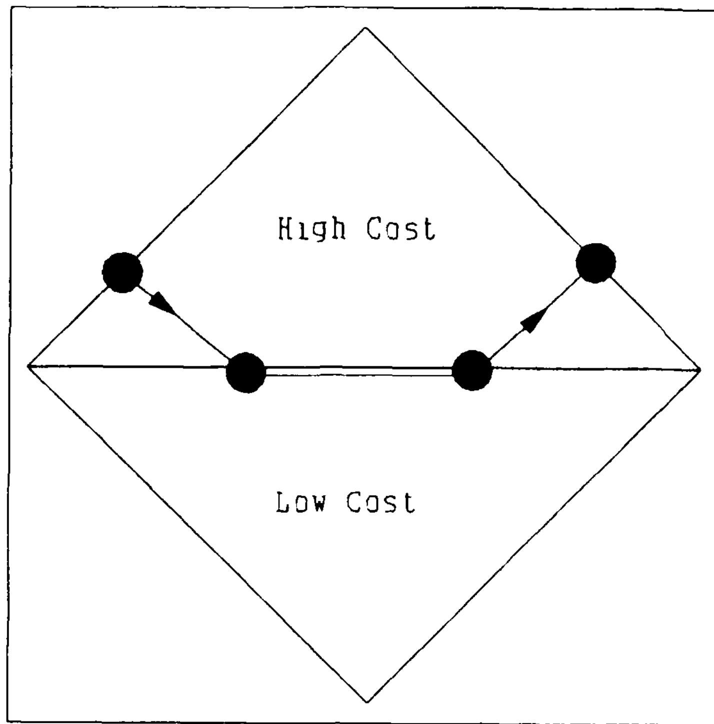


Figure 10. Reentrant Critical Angle Crossings

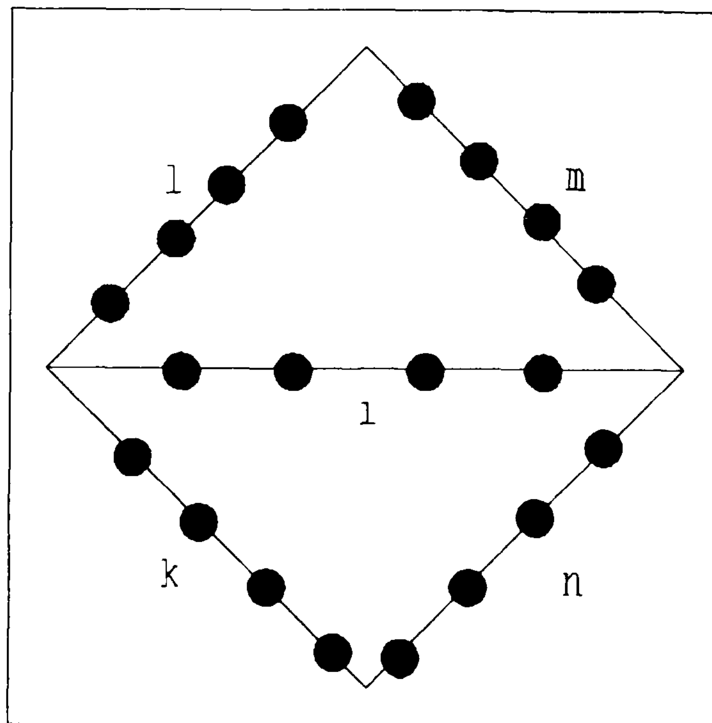


Figure 11. Given Edge and Its Associated Regions and Neighbors

immediate neighbors, the processor on edge i recomputes the position of each of its four path points to ensure the local enforcement of the generalized form of Snell's Law.

The path point P_{i_1}, whose corresponding path traverses the edge pairs (k,l) is updated in the following manner. Each of the 16 path point pairs, with one path point from edge k and one path point from edge l in each pair, must be considered. That is, we must consider the pairs $\{(P_{k_1},P_{l_1}),\ldots,(P_{k_4},P_{l_4})\}$. The generalized form of Snell's Law is applied to each of these 16 path point pairs (with the constraint imposed by the two end points of edge i) to locate the places where they cross edge i. Using the s- and e-values of the path points of each path point pair and the corresponding crossing point on edge i, we compute the cost (p-value) of the two s-e paths, one path crossing the edges in the order k, i, l and the other path crossing the three edges in the order l, i, k. The cheaper cost of the two is chosen as the p-value, p^*, of the new crossing point. The crossing point, P^*, that yields the lowest p-value is then selected from the 16 possible crossing points (p^*'s). If P^* is strictly lower than the p-value previously computed for P_{i_1}, then P_{i_1} will be replaced by P^* and the s- and e-values of P_{i_1} are revised to reflect the new cost of the path through edges i, k and l. Similar computations are performed by processor i to update the other three path points. Processor i then informs its immediate neighbors of the locations of the revised path points and their associated s- and e-values.

Similar procedures are executed by other processors asynchronously and these procedures iterate until some termination condition is met. The termination

condition used in [Ref. 1] was "...if the magnitude of change between successive p-values at all processors fell below a small threshold, processing ceased".

2. Formal Description of the Procedure

Prior to formally describing the LAIPPs procedure in Figure 12, we present the following definitions.

- Functions RIGHT and LEFT return the neighboring edges on the respective sides of a given edge.

- P is a function which returns a path point on a given edge.

- FERMAT is a procedure that takes the two path points on adjacent edges as input and computes the location of the crossing point on an edge based on the generalized form of Snell's Law.

- EUCL is a procedure which computes the Euclidean distance between two points.

- ENDPT is a function that returns the end points of an edge.

- RCOST and LCOST are functions that return the cost of traversing the respective region on the right or left of a given edge.

- rt_pt and lt_pt are two given points and i is a given edge.

- *eps* is a user specified value that determines when to terminate the execution of the FERMAT procedure.

In addition to the above functions, a set of flags are used to denote whether the s- and e-values of a specific point on a given edge have been revised.

```
PROCEDURE LAIPPS;

BEGIN
input:
   set of edges, EDGE, and associated region costs;

initialize:
   location of the four path points,
   e- and s-values, and flags for every edge;

REPEAT
   choose an edge i ∈ EDGE;

   DO (∀rt_edge ∈ RIGHT(i)) and (∀lt_edge ∈ LEFT(i)) BEGIN

      FOR ((∀rt_pt ∈ P(rt_edge)) and (∀lt_pt ∈ P(lt_edge)) and
         (FLAG(i,rt_edge,rt_pt) or FLAG(i,lt_edge,lt_pt))) BEGIN

         np = FERMAT(rt_pt,lt_pt,i);
         rt_dist = rt_cost * EUCL(rt_pt,np);
         lt_dist = lt_cost * EUCL(lt_pt,np);
         cs_rt = (s-value of rt_pt) + rt_dist;
         ce_rt = (e-value of rt_pt) + rt_dist;
         cs_lt = (s-value of lt_pt) + lt_dist;
         ce_lt = (e-value of lt_pt) + lt_dist;

         IF ((cs_rt + ce_lt) > (cs_lt + ce_rt)) THEN BEGIN
            cs = cs_lt;
            ce = ce_rt;
            END
         ELSE BEGIN
            cs = cs_rt;
            ce = ce_lt;
            END;

         IF ((s-value for pairs(rt_edge,lt_edge))
            + (e-value for pairs(rt_edge,lt_edge)) > (cs + ce))
            THEN BEGIN
            s-value for pairs(rt_edge,lt_edge) = cs;
            e-value for pairs(rt_edge,lt_edge) = ce;
            path_point for pairs(rt_edge,lt_edge) = np;
            flag for pairs(rt_edge,lt_edge) = set;
            END;
         END;
      END;
   END; {DO}
UNTIL (no change on every edge);
END; {LAIPPS}
```

Figure 12. Four Point LAIPPs Procedure

The state of these flags determines whether further computation is necessary at a given edge, *i.e.*, if no flags are set at any of the points on all four of the neighboring edges, there is no need to recompute the points on the given edge.

The method for selecting which edge to process next is not specified. The authors in [Ref. 1] suggested the following two options from the many available:

- Cyclically iterate through a list which has been placed in some initial order (perhaps random). The ordering being subject to change after each cycle through the list.

- A sequence of random choices from EDGE which simulates a random, asynchronous process.

To enforce the generalized form of Snell's Law, the procedure FERMAT (Figure 13) computes a point, np, on an edge i of bounded length that minimizes the p-value of the point. This point is computed based on the positions of two path points, one on each of two of its adjacent edges (on opposite sides of edge i). The algorithm is based on the application of the Golden Ratio procedure for finding the minimum value of a convex function [Ref. 10:pp. 510,511].

```
PROCEDURE Fermat(rt_pt,lt_pt,i);

BEGIN
a = ENDPT(i,start_point);
b = ENDPT(i,end_point);

IF ((a.x) == (b.x)) THEN  { to handle the vertical edge }
   swap the x_coord with the y_coord of both end points;

IF (a.x > b.x) THEN
   make b the start_point and a the end_point of the line segment;

dx = b.x - a.x;
k = (b.y - a.y) / dx;

WHILE ( dx > eps) DO

   BEGIN
   ca.x = dx * 0.382 + a.x;
   cb.x = dx * 0.618 + a.x;
   ca.y = dx * 0.382 * k + a.y;
   cb.y = dx * 0.618 * k + a.y;
   d1 = (RCOST(i) * EUCL(ca,rt_pt) + LCOST(i) * EUCL(ca,lt_pt));
   d2 = (RCOST(i) * EUCL(cb,rt_pt) + LCOST(i) * EUCL(cb,lt_pt));

   IF ( d1 < d2 ) THEN
      b = cb;

   ELSE
      a = ca;

   dx = b.x - a.x;
   END;

return a;
END; {Fermat}
```

Figure 13. FERMAT Procedure

IV. EXPERIMENT DESCRIPTION AND RESULTS

A. HARDWARE CONFIGURATION FOR THE EXPERIMENT

The configuration of the hardware for this experiment is illustrated in Figure 14. The host computer was a 80286 based microcomputer with a Transputer board installed. Empirical tests were conducted on a single Transputer as well as on networks of five, nine and 13 Transputers; the additional Transputers being added in groups of four as

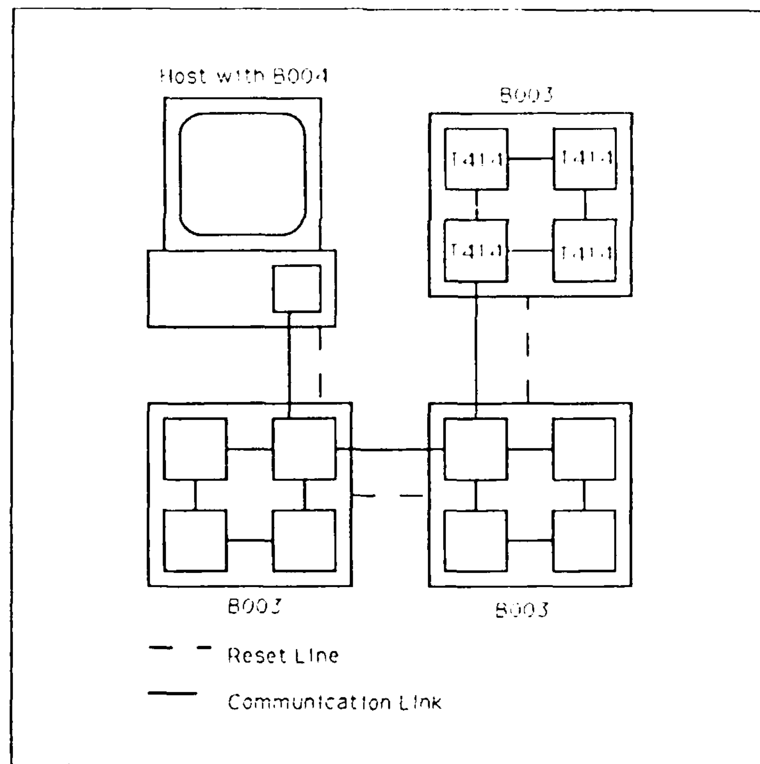


Figure 14. Hardware Configuration

they were mounted on IMS B003 Transputer evaluation boards, each of which has four T414 Transputers. The Transputer that is linked to the host always acts as the root processor of the network and the worker tasks were loaded from this processor.

Code for the Transputer was written, compiled and linked on the host and then subsequently loaded and tested on the root Transputer. All the code for this thesis was developed in this manner, which allowed for separate testing of the different tasks without the need to actually have a network in place for testing.

B. PROGRAMMING LANGUAGE

There are several languages which can be used to write programs for use on the Transputer. Among these are OCCAM, 3L's Parallel Pascal and Parallel C. Because of availability, only two of the languages were considered for coding the LAIPPs on the Transputer network. These two languages are OCCAM and Parallel C.

1. The OCCAM Programming Language

Of the languages available, OCCAM most closely implements the concept of communicating sequential processes. It is a language which was developed explicitly for use with the Transputer and hence supports concurrency and communication channels. Of particular interest are the two constructs described below that enable the language to support parallel processes:

- PAR: This construct has the effect of allowing the processes within its bounds to execute in parallel.

- ALT: This construct is used to allow a processor to select only one of several guarded processes for execution. The process whose guard is first found to be true is executed.

Although OCCAM is the language specifically developed for use with the Transputer, we decided against using it because OCCAM offers only one abstract data structure, the array. Such a limitation would have caused all edge data to be defined as a collection of multidimensional arrays. This makes the addressing of specific fields of the abstract data structure very cumbersome and the resulting code very difficult to write and read.

2. 3L's Parallel C Programming Language

The other programming language available was Parallel C. This is a parallel version of C which incorporates, with a few exceptions, all the attributes of C as defined by Kernighan and Ritchie [Ref. 11]. This commonality with sequential C, the ability to build abstract data types, and the ease of defining and using channels for communication drove us to select 3L's Parallel C as the programming language to implement the LAIPPs.

In Parallel C, communication is achieved by using a run-time library package that provides access to the Transputer's communication channels. Recall that a parallel program, based on the concept of communicating sequential processes, is composed of a collection of tasks, each executing in parallel. In Parallel C, a task is syntactically equivalent to a sequential C program, having a main function and zero or more additional functions. In addition to the functions as they are normally viewed in C, tasks can have other segments of code, called threads, which execute in parallel within the task. This allows several layers of parallelism to exist, not only in the

multiprocessor system, but in a single processor as well. Once compiled and linked the tasks must be mapped onto the available processors and they remain static throughout the execution of the program, *i.e.*, tasks cannot be dynamically distributed throughout the system.

As mentioned above, communication between tasks is achieved via the use of message passing functions provided in the run-time library. Communication can be directed to a specific task or to one of several identical tasks executing on several processors. The latter method, called the "processor farm" approach [Ref. 12:pp. 63-72], is used in this thesis to achieve parallelization and will be discussed in detail in the next section.

C. PROCESSOR FARM APPROACH TO PARALLELIZATION

Recall that when using Parallel C, it is necessary to specify exactly which processor will execute which task(s). This is accomplished through the use of a configuration file. This file is used to name processors, tasks and channels and then to map the channels and tasks onto specific processors within the network. When the work to be executed is divisible into two tasks, one of which can be viewed as a controlling task and the other as a "worker" task, a processor farm approach can be used to map the tasks onto the multiprocessor network. The processor farm approach relieves the programmer from having to map each individual task onto a specific processor in the system. Parallel C provides a tool called a flood-filling configurer, which will automatically load any network of Transputers with the master and/or worker tasks as well as the underlying communication tasks.

The controlling task, referred to as the master task, is loaded onto the Transputer connected to the host computer, and its function is to send and receive work packets to and from the worker tasks in the network. A worker task is a computationally intensive task which receives the data sent to it by the master task in the work packets and performs the computation. The worker task is loaded onto all Transputers in the network including the root Transputer.

The processor farm approach also relieves the programmer from having to worry about specific communication channels and the number of processors in the network. Communication channels are defined and serviced by a task called *frouter*, which is automatically loaded on each processor by the flood-fill configurer [Ref. 12:p. 68]. Additional processors can be added to the network without the need to rewrite the program to identify the additional processors or channels.

Based on the principle of parallelization presented in [Ref. 1], that "each processor carry out a relatively simple computational task", we decided to use the processor farm approach in the implementation of the LAIPPs. Adhering to this principle, the program is divided into two tasks which are described in the next section.

D. PARTITIONING THE PROBLEM

In [Ref. 1], one of the assumptions included in the basic principles of parallelization was that there be a processor located, or associated, with each edge of every triangle. This was clearly not a viable option to pursue due to the limited resources of the Aegis Laboratory. It was therefore decided, early on, to attempt to

partition the problem in a manner such that the computations for any edge could be performed on any available processor.

The problem was partitioned with the processor farm concept in mind and resulted in having one controlling task which dynamically controls the processing of edges and a second task which computes the new path points of an edge using the Fermat procedure previously discussed. The controlling task is executed on the root processor and will henceforth be referred to as the master task. The second task (the worker task) is loaded onto every processor in the network, including the root processor.

1. The Master Task

The master task has three threads (parallel processes) which execute concurrently. The main thread inputs the edge data, sets all flags, initializes the path points on the edges and sets their s- and e-values to infinity, and begins the execution of the other two threads. (The input edge data is formatted so that the source point resides on the first edge and the goal point on the last edge.) The other two threads basically send and receive packets to and from the worker tasks on the other processors in the network. Data is shared among the threads by the use of semaphores. (Refer to Figure 15.)

Note that [Ref. 1] offered no definitive method to determine the order in which edges are to be selected for processing; we therefore elected to use a first-in first-out queue to maintain the list of edges which need to be considered for processing. Initially all edges, except the edges containing the source and destination

```
PROCEDURE Main; { Master Task };

BEGIN
    read edge data;
    initialize path points, flags and counters;
    place all edges in queue to await processing;
    start parallel execution of send and receive threads;

    WHILE ( NOT done ) DO  release processor to other processes;

    perform Dijkstra's search to find least cost path;
    output results;
END;  {Main}
```

Figure 15. Master Task Algorithm

points, are placed in the queue. An edge is removed from the head of the queue and a determination is made as to whether it can be processed. The determination of whether an edge can be processed is based on three criteria:

- whether the edge has already been farmed out to a processor,
- whether any of its neighbors is presently farmed out, and
- whether the edge contains the source or goal points or is an external boundary.

If the edge is already being processed, then no further action is necessary. The edge is discarded and the procedure is repeated with the next edge removed from the queue. If, however, one of the edge's neighbors prevents farming out the edge, then the edge is placed at the end of the queue for reconsideration later. This procedure is continued until the queue is empty and there are no edges being processed out in the processor farm. The algorithm for the send thread appears in Figure 16.

```
PROCEDURE Send;      { Send Thread }

BEGIN
  LOOP
    IF (not empty queue) THEN

      BEGIN
      remove edge from head of queue;

      IF (edge does not contain source or destination) THEN

        IF ((edge is not being processed) AND
            (no neighbor of edge is being processed)) THEN


          BEGIN
          set appropriate flags;
          build work packet;
          send work packet to idle processor;
          END;

        ELSE IF (edge is not being processed) THEN
          place edge at tail of the queue;
    END;
  END LOOP;
END;   { Send }
```

Figure 16. Send Thread Algorithm

The Receive thread waits for any work packets being returned from the processor farm. When a packet arrives, it is unbundled, the edge data is updated with the results received in the work packet, flags are reset appropriately and counters updated. Additionally, if any of the path points on the edge just received have been modified then the four neighbors of the edge are placed in the queue so that they will be considered for processing based on the revised path points. The algorithm for this thread appears in Figure 17.

```
PROCEDURE Receive;        { Receive Thread }

BEGIN
  LOOP

    WHILE ( NOT packet arrived);

    unbundle packet;
    IF (any path point has been modified) THEN
      enqueue neighbors;

    reset flags and counters;
  END LOOP;
END;  { Receive }
```

Figure 17. Receive Thread Algorithm

2. The Worker Task

The worker task has only one thread (main) and computes the location of the new path points based on the location of the path points on the neighboring edges. It is straightforward in its execution and employs the Fermat procedure to recompute the new path points. The task waits for a work packet to arrive, then calls the Fermat procedure four times (once for each of the four possible edge pairs). The results are then placed in a results packet and returned to the master task.

E. EXPERIMENTAL DESIGN AND RESULTS

1. Initial Conditions and Convergence Criteria

In the experiments described below it was necessary to initialize the s- and e-values and the path points for each edge in the partition. Since the path points are recomputed and can be moved anywhere between the edge's end points, the path points

can be initialized to lie anywhere on the edge. For our experiments all four path points on every edge were initialized to a position at the center of the edge. The only exceptions being the two edges which contain the source and destination points. For the edges containing the source or the destination points, the four path points were set to the actual coordinates of those points and recomputation of these points was prevented by performing a check in the send thread. The only requirement for the initial s- and e-values of all the path points was that they exceed the optimal values. This was easily accomplished by initializing these values to a large value (such as the maximum value of the integer type), except the s-value of the source point and the e-value for the destination point which were both set to zero to ensure convergence.

As mentioned in a previous section, the Fermat procedure continually recomputed the path points until a convergence criteria, *eps*, was met. The value of *eps* determines the degree of accuracy to which the p-value is computed. The smaller the value of *eps*, the greater the amount of computation is needed before the Fermat procedure terminates. Because our main goal was to determine the feasibility of the LAIPPs in a truly parallel network of processors, we decided not to make the value of *eps* very small. For this experiment the value of *eps* was set to 0.0001.

2. Experiment Description and Results

In order to test our implementation of the LAIPPs, three sets of experiments were designed. The first set of experiments took the form of a two dimensional chain of triangular regions of uniform cost having 41 edges as shown in Figure 18. The second and third sets of experiments both involve triangulations of a

hexagon having non-uniform region costs. The second experiment involves a hexagon with 42 edges and the third involves a hexagon with 156 edges (refer to Figures 19 and 20, respectively). The region costs for the large hexagon were randomly generated and assigned. The region costs for the small hexagon, however, were intentionally biased in order to determine whether the program would produce a path through a particular series of lower cost regions. The program was executed 5 five times for each combination of the two smaller maps and the number of Transputers in the network (a total of 20 times for each map). The program for the large hexagon was executed only five times with a network of 13 Transputers due to time constraints.

The results of all the experiments were encouraging. All the paths that were calculated crossed the edges in close proximity to each other. Deviations between the computed path points for each of the computed paths were within ±0.1 of a unit distance of each other. The resulting paths can be seen in Figures 18 through 20. Table 1 contains the results of the experiments for the linear chain of triangles, Tables 2 and 3 contain the results for the small and large hexagons, respectively.

The data show that using the processor farm approach, the addition of more processors to the network reduces the total time to compute the near optimal path. The results tend to show, however, that after the number of processors in the network reaches a certain point, the expected decrease in processing time is not guaranteed and may increase. Looking at Table I, there is clearly a decrease in processing time when going from a network having five processors to nine. However, the next step from nine processors to 13 increases processing time. Table II shows that when going from

Number of Processors	Edges Processed	Total Processing Time	Average Time per Edge
1	1991	515 min 56 sec	15.55 sec
5	3693	276 min 7 sec	4.50 sec
9	1458	64 min 50 sec	2.59 sec
13	3091	280 min 21 sec	5.40 sec

Table I. Results For Linear Chain

Number of Processors	Edges Processed	Total Processing Time	Average Time per Edge
1	1053	206 min 37 sec	11.77 sec
5	852	34 min 29 sec	2.43 sec
9	759	27 min 11 sec	2.14 sec
13	734	27 min 33 sec	2.25 sec

Table II. Results For Small Hexagon

Number of Processors	Edges Processed	Total Processing Time	Average Time per Edge
13	6426	103 min 45 sec	0.97 sec

Table III. Results for Large Hexagon

a network of five to nine processors, there is only a small improvement in performance and taking the next step to 13 processors produces no noticeable improvement.

These results may be attributable to several factors such as the shape or form of the triangulated plane or the initial ordering of the edges in the queue. Table I shows the timing results for processing a chain of triangular regions and Table II the results of processing regions grouped together to form a hexagon. The results for these two experiments both improve to a point but the addition of more processors beyond

this point causes an increase in processing time for the chain of regions where the hexagon's results show no change. Further examination of the two experiments reveals that the first experiment has 20 edges which are external border edges while the second experiment has only 12. Since edges that are external borders are not processed, there is a higher probability in the second experiment that the edge at the head of the queue can be processed immediately resulting in improved processing speed.

There may be other factors which may also influence processing time, such as the method used to select the next edge for processing. Or perhaps the total number of edges in the experiment, or the method used to determine which edges are placed in the queue to await processing. In this thesis, for example, all the neighbors of an edge which has had one or more of its path points modified are put in the queue. These are only a few factors which may have an affect on processing time and need to be further investigated.

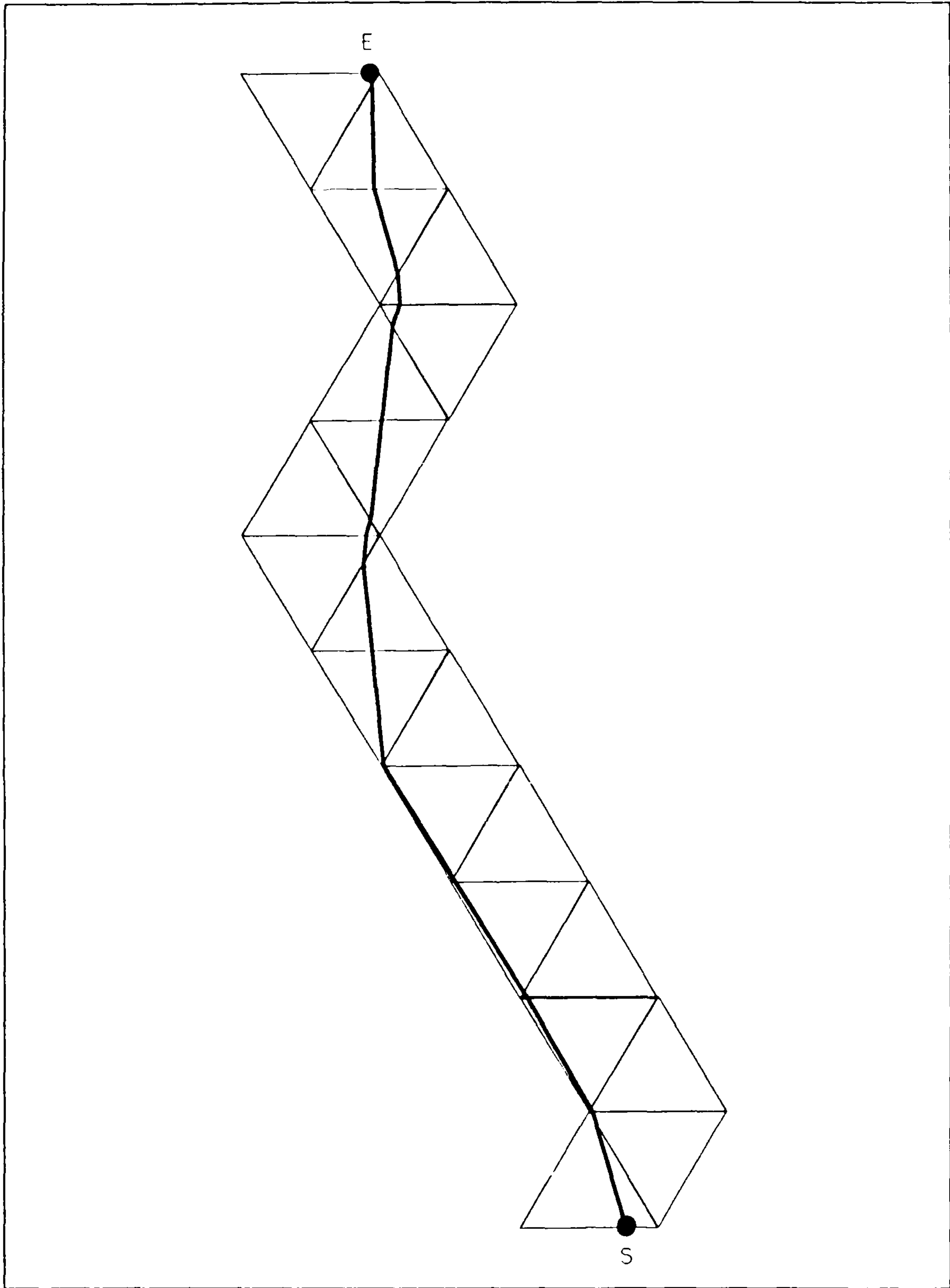


Figure 18. Chain of Triangular Regions

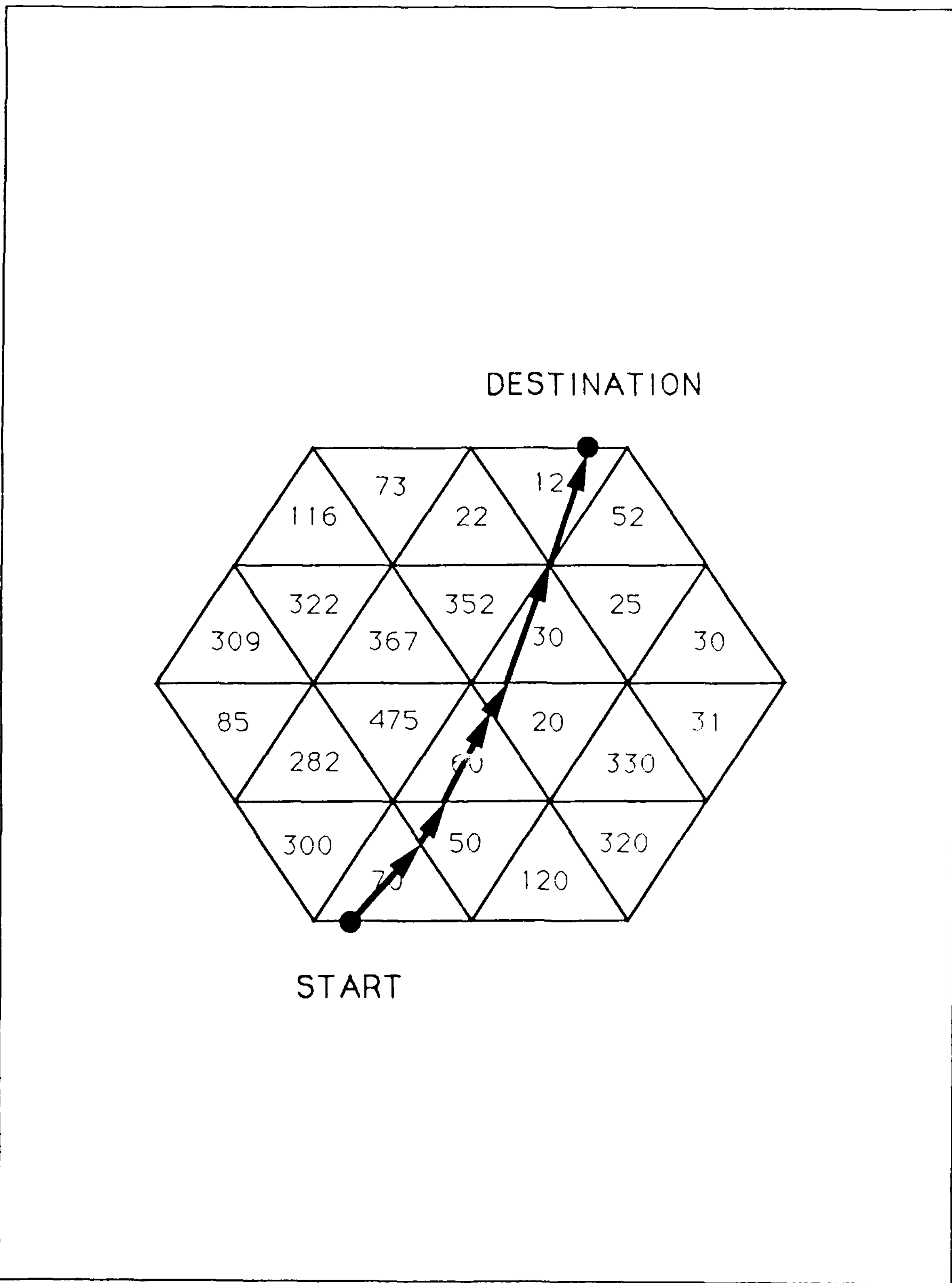


Figure 19. Triangulation of Small Hexagon

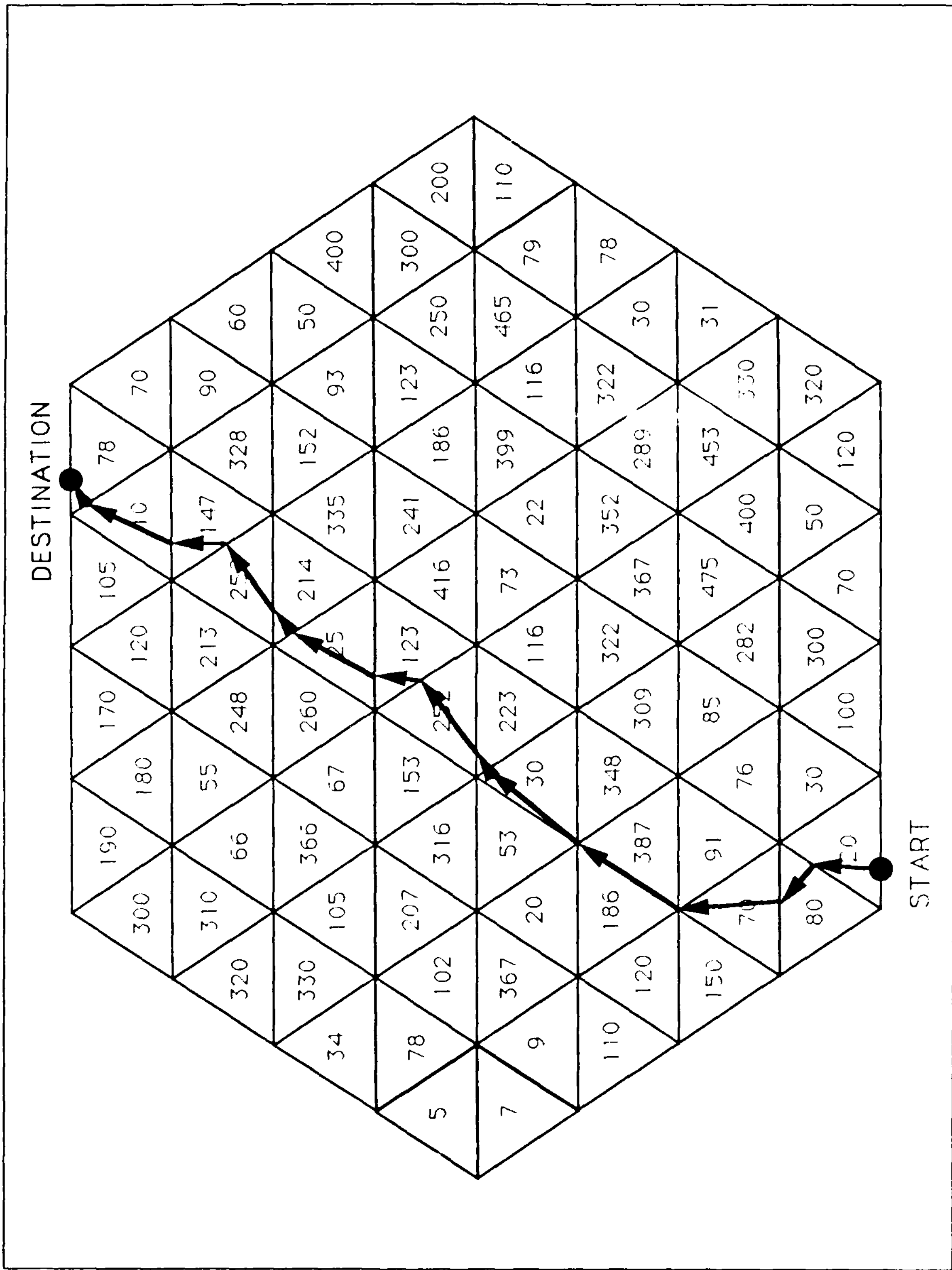


Figure 20. Triangulation of Large Hexagon

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

We have shown that the weighted region least cost path problem can be solved in a distributed computing environment using the basic concepts of the LAIPPs presented in [Ref. 1]. The problem was partitioned in a manner that allowed any edge which was available for processing, to be processed by any idle processor. We have also shown that the program will compute a path that is very close to a least cost path using a map with biased region weights.

The results of the testing , however, have shown that, for the procedures implemented here, simply adding more processors does not in itself translate to a decrease in processing time. In our experiments it was shown that the addition of processors past a point did not improve performance and may in fact cause degradation of performance. This point appears to be dependent on the number of edges in the map and possibly the layout of the map.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

Due to the limitations placed on us by the library functions provided with 3L's Parallel C, it is not possible to use a mix of Transputer types in a processor farm. It is recommended that procedures be developed and implemented to allow the mixing of Transputer types within the processor farm. It is also recommended that fault

tolerant code be implemented to avoid deadlock which may be caused by lost messages or bad communication links.

This thesis was the first step in implementing the LAIPPs, the next logical step would be to implement a program which allows reentrant paths in the computation of the least cost path. Another area which needs to be implemented involves finding the least cost path for non-isotropic weighted regions [Ref 13]. Timing studies should be conducted using various sizes and geometries to determine if there is an optimal size for the network of Transputers which produces the best performance in terms of processing time. Perhaps this optimum network size is dependent on the shape and/or size of the problem set and needs to be further investigated.

The final recommendation seeks to determine whether these procedures can be implemented on a heterogenous network of conventional processors, such as the ISI workstations located in the Artificial Intelligence Laboratory at the Naval Postgraduate School. The results should then be compared with those obtained when using a network of Transputers of equal size.

APPENDIX A - HEADER FILE SOURCE CODE

```
/*
    Author:     Ivan Garcia
    Date:       10 July 1989
    Title:      LAIPP.H

    Last Modified:  1 Sep 1989

    This file contains the formats for the packets used to communicate between
    the master and worker tasks.
*/


#include    "d:/tc2v0/math.h"
#include    "d:/tc2v0/sema.h"
#include    "d:/tc2v0/net.h"


#define     MODIFIED        1       /* modified flag is set */
#define     NOT_MODIFIED    0       /* modified flag is not set */
#define     IN_USE          1       /* edge is in use */
#define     NOT_IN_USE      0       /* edge is not in use */
#define     FOUR            4
#define     TWO             2
#define     TRUE            1
#define     FALSE           0
#define     SET             1
#define     UNSET           0


typedef struct  coord_structure
{
    float   x, y;
}   COORD;


typedef struct  path_pt_structure
{
    float   s_value, e_value;
    COORD   path_pt_coord;
    int     flag;
}   PATH_PT;
```

```
typedef struct  edge_structure
{
    PATH_PT path[FOUR];                  /* 4 path points per edge */
    int     right[TWO], left[TWO];       /* 2 adjacent edges, rt & lt */
    COORD   start_coord, end_coord;      /* 2 end points of edge */
    int     right_cost, left_cost;

}   EDGE;

/*      Graphic description of the EDGE structure
 *
 *   path    0       1       2       3
 *        ------- ------- ------- -------
 *       |       |       |       |       |
 *       |       |       |       |       |      type PATH_PT
 *        ------- ------- ------- -------
 *    right  0       1
 *        ------- -------
 *       |       |       |
 *       |       |       |                      type INT
 *        ------- -------
 *    left   0       1
 *        ------- -------
 *       |       |       |
 *       |       |       |                      type INT
 *        ------- -------
 *    start_coord     end_coord
 *        -------         -------
 *       |       |       |       |
 *       |       |       |       |              type COORD
 *        -------         -------
 *    right_cost      left_cost
 *        -------         -------
 *       |       |       |       |
 *       |       |       |       |              type INT
 *        -------         -------
 *
 *
 */

typedef    struct    command_structure {
    int      num;        /* id's the edge when returned to master task */
    EDGE     e;
    PATH_PT n[4][4];
}   COMMAND;

typedef    struct    reply_structure
{
    int      num;        /* id's the edge when returned to master task */
    EDGE     e;
} REPLY;
```

APPENDIX B - QUEUE DATA STRUCTURE SOURCE CODE

```
/*
 *      Name:  que.c
 *      Author: I. Garcia
 *      Date:   15 Aug 1989
 *      Source: que.c
 *      Revised:21 Aug 1989
 *
 *    Implementation of a queue data structure.
 *
 */



typedef struct {
  int      data;
  struct node
           *next;
} node;


static  node  *head,
              *tail;



int      empty_queue()
{

  /* Return true if empty, false otherwise. */
  if ( head == NULL )
    return (TRUE);
  else
    return (FALSE);
}
```

```
qadd(p_data)
int     p_data;  {

  node  *new_node;

  /*  Add the data element to the tail of the queue. */

  new_node = (node *) malloc(sizeof(node));
  new_node->data = p_data;
  new_node->next = NULL;

  if ( empty_queue() )
  {
    head = tail = new_node;
  }
  else
  {
    tail->next = new_node;
    tail = new_node;
  }
}



int     dequeue()
{

  /*  Remove the item at the head of the queue.
   *  This function does not check for an empty queue.  The user must
   *  perform this test prior to calling this function.
   *  Otherwise results are not guaranteed.
   */

  int     p_data;
  node     *front_node;

  front_node = head;
  p_data = front_node->data;

  if ( head->next == NULL )
    head = tail = NULL;

  else
    head = head->next;

  free(front_node);
  return (p_data);
}
```

APPENDIX C - MASTER TASK SOURCE CODE

```
/*          Author: Ivan Garcia
            Modified: 29 Sep 1989
                      3 Nov 1989

 * The user of this program should ensure that the Source coordinates
 * lie on the edge entered as the first edge and that the Sink
 * coordinates lie on the edge of the LAST edge entered from the data
 * file.
 */

#include    <stdio.h>
#include    <par.h>
#include    <thread.h>
#include    <time.h>
#include    "laipp.h"
#include    "que.c"

#define     MAX_EDGES       160
#define     MAX_NUM         1048576
#define     MAP             "series3"

static int      use_flag[MAX_EDGES];      /* Indicates edge is being
                                           * processed.
                                           */

static int      users[MAX_EDGES];         /* How many other edges are
                                           * currently using this edge's
                                           * data. Range is 0 to 4.
                                           */

static int      m_flag[MAX_EDGES];        /* Flag is set when this edge has
                                           * been referenced by all its
                                           * neighbors simultaneously.
                                           * i.e. users[i] reaches 4.
                                           * This prevents any of its
                                           * neighbors from recomputing path
                                           * points before edge 'i' has
                                           * itself been recomputed.
                                           */

static EDGE     edge[MAX_EDGES];            /* Pointer to array of edges.*/

static SEMA     memory_free;              /* Memory Access semaphore. */

static int      done,
                num_of_edges,
                iterations,           /* Number of edges sent
                                       * for processing
```

```
                                        */
            edges_out;



/*
 *          I N C R _ U S E R S
 *
 *      Increments the counter which is used to prevent an edge
 * from never being processed.  Once the counter reaches 4 the m_flag
 * is set.
 *
 */
incr_users(i)
int     i; {

    int     k;

    for (k=0; k<4; k++)
    {
        if ( ++users[edge[i].right[k]] == 4)
            m_flag[edge[i].right[k]] = SET;
    }

    /* No need to protect these edges against starvation. */
    users[0] = users[1] = users[num_of_edges] = 0;

}



/*                  D E C R _ U S E R S
 *
 *      Decrease the value of the users entry for the neighbors of i.
 *  This value is not allowed to decrease below 0.
 */
decr_users(i)
int     i;
{
    int     k;

    for (k=0; k<4; k++)
    {
        if ( --users[edge[i].right[k]] < 0 )
            users[edge[i].right[k]] = 0;
    }

}
```

```
/*
 *              N E I G H B O R S _ F R E E
 *
 *      Check the neighbors of edge i to assure that they are not being
 * starved.
 *
 */
neighbors_free(i)
int     i;
{
    int     k;

    for ( k=0; k<4; k++ )
    {
        if (
            (m_flag[edge[i].right[k]] == SET) ||
            (use_flag[edge[i].right[k]] == IN_USE)
           )
            return(FALSE);
    }

    return(TRUE);
}




send()
{
    int         empty,              /* Status flag of queue */
                free,               /* Status flag for free neighbors */
                processing,         /* Flag set when edge is processing */
                s_edge,
                j;
    COMMAND     c;


    /* Find an edge whose neighbor has been modified and which is not
     * in use.  Send that edge to a worker and mark it as in use.
     */
    for(;;)
    {
        sema_wait(&memory_free);
        empty = empty_queue();
        sema_signal(&memory_free);

        if ( !empty )
        {
            sema_wait(&memory_free);
            s_edge = dequeue();
            sema_signal(&memory_free);

            if ( (s_edge <= 1) || (s_edge == num_of_edges) )
            {
```

```
        /*  There is no need to process edges 0,
         *  1 and the target edge. Reset use flag,
         *  discard edge and continue.
         */
        sema_wait(&memory_free);
        use_flag[s_edge] = NOT_IN_USE;
        sema_signal(&memory_free);
        continue;
    }

    sema_wait(&memory_free);
    free = neighbors_free(s_edge);
    processing = use_flag[s_edge];
    sema_signal(&memory_free);

    /* If edge is not being processed and neighbors are free,
     * then farm out the edge for processing.
     */
    if ( (!processing) && (free) )
    {

        /* Build the packet. */
        c.num = s_edge;


        sema_wait(&memory_free);
        incr_users(s_edge);
        use_flag[s_edge] = IN_USE;
        c.e = edge[s_edge];
        edges_out++;
        iterations++;
        for ( j=0; j<4; j++ )
        {
            c.n[0][j] = edge[edge[s_edge].right[0]].path[j];
            c.n[1][j] = edge[edge[s_edge].right[1]].path[j];
            c.n[2][j] = edge[edge[s_edge].left[0]].path[j];
            c.n[3][j] = edge[edge[s_edge].left[1]].path[j];
        }
        sema_signal(&memory_free);

        /*  Send the packet.  */
        net_send( sizeof(COMMAND), &c, 1 );
        par_printf("Sent: %d\n",c.num);

    }
    else
    {
        /* A neighbor is being processed so replace edge in queue
         * for later processing.
         */
        if ( !processing )
        {
            sema_wait(&memory_free);
            qadd(s_edge);
            sema_signal(&memory_free);
```

```
                }
            }
        }

        /* Are we done? */
        sema_wait(&memory_free);
        if ((empty_queue()) && (edges_out == 0))
        {
            done = TRUE;
        }
        sema_signal(&memory_free);

    }
}




/*
 *  Title:      RECV()
 *  Author:     Ivan Garcia
 *  Date:       19 July 1989
 *  Modified:   20 July 1989
 *
 *  This thread waits until a worker task returns a completed packet
 * and then writes the results to memory.  It then resets the appropriate
 * use flags to NOT IN USE.
 *
 */
recv()
{
    REPLY       r;

    int         empty,              /* Status of queue flag */
                ready,              /* Signals end of message */
                i,
                j,
                k;

    for (;;) {

        /* Wait here until packet arrives. */
        net_receive(&r, &ready);

        /* Unpack the packet */
        k = r.num;

        if (
            ( r.e.path[0].flag == MODIFIED ) ||
            ( r.e.path[1].flag == MODIFIED ) ||
            ( r.e.path[2].flag == MODIFIED ) ||
            ( r.e.path[3].flag == MODIFIED )
           ) {
```

```
            sema_wait(&memory_free);
            for ( i=0; i<4; i++ )
            {
                qadd(edge[k].right[i]);
            }

            sema_signal(&memory_free);
        }


        sema_wait(&memory_free);
        edge[k] = r.e;
        edges_out--;
        decr_users(k);
        use_flag[k] = NOT_IN_USE;
        m_flag[k] = UNSET;
        sema_signal(&memory_free);
        par_printf("Recv'd: %d\n",k);
    }
}



init_paths(j)
EDGE     *j;
{
    float   dx,
            dy,
            sx,                         /* Start x coordinate */
            sy;                         /* Start y coordinate */

    int     i;

    sx = j->start_coord.x;
    sy = j->start_coord.y;

    /*
     * Determine the difference between the start and end coordinates.
     */
    dx = ( (j->end_coord.x) - (j->start_coord.x) ) / 2;
    dy = ( (j->end_coord.y) - (j->start_coord.y) ) / 2;

    for (i=0; i<4; i++){
        j->path[i].path_pt_coord.x = sx + dx;
        j->path[i].path_pt_coord.y = sy + dy;
        j->path[i].flag = MODIFIED;
        j->path[i].s_value = MAX_NUM;
        j->path[i].e_value = MAX_NUM;
    }

    return;
}
```

```
struct  min
{
    COORD    coord;
    float    s_val,
             e_val;
    int      edge;
};

typedef struct min MIN_STRUCT;




/*                    F I N D _ M I N
 *
 *      Finds the point on edge "num" whose sum of the e and s values is
 *  least of the four points on the edge.
 *
 */
MIN_STRUCT      find_min(num)
int     num;
{
    int      i;
    MIN_STRUCT
             min_pt;

    min_pt.coord = edge[num].path[0].path_pt_coord;
    min_pt.e_val = edge[num].path[0].e_value;
    min_pt.s_val = edge[num].path[0].s_value;
    min_pt.edge = num;

    for (i=1; i<4; i++)
    {
        if ( (edge[num].path[i].e_value + edge[num].path[i].s_value)
              < (min_pt.s_val + min_pt.e_val) )
        {
            min_pt.s_val = edge[num].path[i].s_value;
            min_pt.e_val = edge[num].path[i].e_value;
            min_pt.coord = edge[num].path[i].path_pt_coord;
        }
    }

    return (min_pt);
}



main()
{
```

```
FILE    *fp;

int     e_num,
        start_time,
        stop_time,
        i,                      /* Counter */
        j;                      /* Counter */

COORD   Source,
        Sink;

MIN_STRUCT     min_pt,
               nu_min_pt;

float   dist,
        d1;



struct trail                          /* Structure used to store the */
{                                       /* least cost path constructed */
    int      edge;
    COORD    coord;
    float    dist;
    struct   trail     *next;
};

typedef struct trail     TRAIL;

TRAIL    *start,
         *current;


/* Initialize the semaphores. */
sema_init(&memory_free, 1);             /* Memory free */

if (( fp = fopen(MAP, "r" )) == NULL )
{
    printf(" Cannot open %s.\n", MAP);
    exit(0);
}

if ( fscanf( fp, "%d", &num_of_edges) != EOF )
    printf(" There are %d edges in the database.\n\n", num_of_edges);

else
{
    printf(" ERROR: EOF WHERE DATA EXPECTED.\n\n");
    exit(0);
}


/* Read Source and Sink coordinates. */
fscanf(fp, "%f %f %f %f", &Source.x, &Source.y, &Sink.x, &Sink.y);

i = 1;
```

```
    /* Read the information for each edge.*/
    while(fscanf( fp, "%f %f %f %f %d %d %d %d %d %d",
                &edge[i].start_coord.x,
                &edge[i].start_coord.y,
                &edge[i].end_coord.x,
                &edge[i].end_coord.y,
                &edge[i].right_cost,
                &edge[i].left_cost,
                &edge[i].right[0],
                &edge[i].right[1],
                &edge[i].left[0],
                &edge[i].left[1]) !=EOF )
    {
        i++;
    }

/*
 * Initialize the flags which indicate that an edge is in use (use_flag[])
 * Initialize all s and e values to a large value.
 */

    for (i = 0; i < (num_of_edges+1); i++){
        use_flag[i] = NOT_IN_USE;
        m_flag[i] = UNSET;
        users[i] = 0;
        init_paths(edge[i]);            /* Initialize path points */
    }

    /* Set all path points of edge 1 to the coordinates of the Source.
     *
     *  Set all coordinates of edge "num_of_edges" to the coordinates
     * of the Sink.
     *
     *  The user of this program should ensure that the Source coordinates
     * lie on the edge entered as the first edge and that the Sink
     * coordinates lie on the edge of the LAST edge entered from the data
     * file.
     *
     */

    for ( i=0; i<4; i++ )
    {
        edge[1].path[i].path_pt_coord.x = Source.x;
        edge[1].path[i].path_pt_coord.y = Source.y;
        edge[1].path[i].s_value = 0;
        edge[1].path[i].e_value = MAX_NUM;

        edge[num_of_edges].path[i].path_pt_coord.x = Sink.x;
        edge[num_of_edges].path[i].path_pt_coord.y = Sink.y;
        edge[num_of_edges].path[i].s_value = MAX_NUM;
        edge[num_of_edges].path[i].e_value = 0;
    }

    /* Enqueue all edges for processing except for edges 1 and
     * num_of_edges.
```

```
 */
for ( i=1; i<num_of_edges; i++ )
  {
    qadd(i);
  }

done = FALSE;
iterations = 0;
edges_out = 0;

close(fp);

printf(" \n Input of data successful.\n\n");

start_time = clock();

/* Create the send and receive threads. */
thread_create(send, 10000, 2, 0, 0);
thread_create(recv, 10000, 2, 0, 0);

i = 1;

for (;;)
{
    sema_wait(&memory_free);
    j = done;
    sema_signal(&memory_free);
    if (j)
        break;

    if ((i++ % 1000) == 0)
    {
        par_printf(".");
        i = 1;
    }

    thread_deschedule();
}

stop_time = clock();


par_printf("\n**** Begin postprocessing\n\n");
/* Post-processing begins here. */

for (i=2; i<num_of_edges; i++)
    use_flag[i] = NOT_IN_USE;


/* Begin building a linked list with the first node having the
 * Source coordinates.
 */
start = (TRAIL *) malloc (sizeof(TRAIL));
start->coord = Source;
start->edge = 1;
start->next = NULL;
```

```
current = start;

use_flag[0] = IN_USE;
use_flag[1] = IN_USE;


/* Set the first neighbor of the current edge as the next path point
 * of the least cost path. Then compare this point's s and e values
 * with those of the other three neighbor edges. If the new neighbor's
 * minimum path point has smaller s and e values then that point becomes
 * the next path point.
 */
e_num = 1;
for (;;)
{
    if (e_num == num_of_edges)
        break;
    min_pt.edge = edge[e_num].right[0];
    min_pt = find_min(min_pt.edge);
    for (i=1; i<4; i++)
    {
        if (use_flag[edge[e_num].right[i]] == NOT_IN_USE)
        {
            nu_min_pt = find_min(edge[e_num].right[i]);
            if ( nu_min_pt.e_val < min_pt.e_val)
            {
                min_pt = nu_min_pt;
                min_pt.edge = edge[e_num].right[i];
            }
        }
    }

    /* Append this new point to the linked list. */
    current->next = (TRAIL *) malloc (sizeof(TRAIL));
    current = current->next;
    current->coord = min_pt.coord;
    current->edge = min_pt.edge;
    current->dist = min_pt.s_val + min_pt.e_val;
    current->next = NULL;
    e_num = min_pt.edge;
    use_flag[e_num] = IN_USE;

}

par_printf("*** PRINTING RESULTS\n\n");

if ( (fp = fopen("results","w")) == NULL )
{
    par_printf("Cannot open results. Writing to stdout.\n");
    fp = stdout;
}

par_fprintf(fp,"\n\nThe optimal path found for %s follows.\n\n", MAP);
par_fprintf(fp,"EDGE          X-COORD               Y-COORD                    DISTANCE\n");
par_fprintf(fp,"==========================================================================\n\n");
current = start;
```

```
    while (current)
    {
        par_fprintf(fp,"%4d      %15.10f      %15.10f      %15.10f\n",
                current->edge,
                        current->coord.x,
                        current->coord.y,
                        current->dist);
        current = current->next;
    }

    start_time = stop_time - start_time;

    par_fprintf(fp,"\n\n %d edges were processed.",iterations);
    par_fprintf(fp,"\n\nOverall processing time = %d minutes %d seconds.",
                start_time/60, start_time % 60);

    close(fp);

    while ( !empty_queue() )
    {
        par_printf("edge %d\n",dequeue());
    }


    par_printf(" Que is %s\n", empty_queue() ? "empty" : "NOT empty");
    par_printf(" There are %d edges out processing.",edges_out);

    par_printf(" Job completed normally\n");

}
```

APPENDIX D - WORKER TASK SOURCE CODE

```
/*          Title:  Lw.c
 *          Author: Ivan Garcia
 *          Date:   14 Aug 1989
 *          Revised:21 Aug 1989
 *
 *          Source: LW.c
 *
 *      This is the worker task for finding the least cost path between
 *  two points on a weighted triangulated plane.
 *
 *      Included in this task are the functions eucl and fermat.
 *
 *      5 Oct 1989: Modified to handle the case of a vertical edge.
 *
 */
#include  "laipp.h"
#include  <math.h>

int       errno;

static    EDGE      e;                      /* Subject edge */


/*  Eucl returns the euclidean distance between the two points
 *  p1 and p2.
 */
float   eucl(p1,p2)

COORD   p1,
        p2;
{
        float  x, y;

        x = p1.x - p2.x;
        y = p1.y - p2.y;

        return  sqrt((x * x) + (y * y));

}


/* fermat:      computes the location of a point on an edge minimizing
the p-value of the point given 2 points on adjaceent edges. */

fermat( rt_pt, lt_pt, subj_edge, np )
```

```
COORD   rt_pt,
        lt_pt,
        *np;

EDGE    subj_edge;
{
   COORD          a,
                  b,
                  ca,
                  cb;

   float          lx,
                  ly,
                  k,
                  d1,
                  d2,
                  eps = 0.0001;

   a = subj_edge.start_coord;      /* get coord of start_pt of edge */
   b = subj_edge.end_coord;        /* get coord of end_pt of edge   */

   if ( a.x == b.x )               /* Vertical case */
   {
      ca.x = a.x;                  /* Swap x and y coordinates */
      ca.y = b.x;
      a.x  = a.y;
      b.x  = b.y;
      a.y  = ca.x;
      b.y  = ca.y;
   }

   if ( a.x > b.x )                /* Ensure b.x is larger than a.x */
   {                               /* so that k below is always positive.*/
      ca = a;
      a  = b;
      b  = ca;
   }

   lx = b.x - a.x;
   k = (b.y - a.y) / lx;

   while ( lx > eps ) {

      ca.x = lx * 0.382 + a.x;
      cb.x = lx * 0.618 + a.x;
      ca.y = lx * 0.382 * k + a.y;     /* Use similar triangles to compute */
      cb.y = lx * 0.618 * k + a.y;     /* y coordinates for 2 points */


      d1 = subj_edge.right_cost * eucl(ca, rt_pt) +
      subj_edge.left_cost * eucl(ca,lt_pt);

      d2 = subj_edge.right_cost * eucl(cb, rt_pt) +
      subj_edge.left_cost * eucl(cb,lt_pt);

      if ( d1 < d2 )
```

```
            b = cb;

        else
            a = ca;

        lx = b.x - a.x;

    }

    *np = ca;

}       /* fermat */



comput_pt( rn, ln, i)
PATH_PT     rn[],
            ln[];
PATH_PT     *i;{

    int      rp,
             lp;

    float    rt_d,                   /* Euclidean dist to rt point. */
             lt_d,                   /* Euclidean distance to lt point. */
             cs,                     /* Weighted cost from s to np. */
             ce,                     /* Weighted cost from e to np. */
             cs_rt,                  /* Right and left points' costs. */
             ce_rt,
             cs_lt,
             ce_lt;

    COORD    r_pt,
             l_pt,
             np;                     /* New point */

    /* For each point of right neighbor i */
    for ( rp=0; rp<4; rp++ ){

      /* For each point in left neighbor j */
      for ( lp=0; lp<4; lp++ ){

        /* get pt rp of right edge r */
        fermat(rn[rp].path_pt_coord, ln[lp].path_pt_coord, e, &np);

        /* Compute weighted distances between points */
        rt_d = e.right_cost * eucl(r_pt, np);
        lt_d = e.left_cost * eucl(l_pt, np);

        /* Determine new s and e values between these points. */
        cs_rt = rn[rp].s_value + rt_d;
        ce_rt = rn[rp].e_value + rt_d;
        cs_lt = ln[lp].s_value + lt_d;
```

```
        ce_lt = ln[lp].e_value + lt_d;

        /* Determine the optimal cost and direction for this point. */
        if (( cs_rt + ce_lt) > (cs_lt + ce_rt)){
          cs = cs_lt;
          ce = ce_rt;
        }
        else{
          cs = cs_rt;
          ce = ce_lt;
        }

        /* Compare this new optimum
         * with the optimum costs currently in points k and j.
         */
        if (( i->s_value + i->e_value ) > ( cs + ce )){
            i->s_value = cs;
            i->e_value = ce;
            i->path_pt_coord = np;
            i->flag = MODIFIED;
        }
      }
    }
}



main(){

  int      j,
           i,
           f;                              /* End of packet flag */

  COMMAND    i_p;                          /* Incoming edge packet. */
  REPLY      o_p;                          /* Outgoing packet. */

  PATH_PT    m[4][4];                      /* Neighbors' path points. */



  for (;;){

    /* Wait here for edge packet to arrive. */
    net_receive( &i_p, &f );

    /* Break out the packet. */
    e = i_p.e;
    o_p.e = i_p.e;
    o_p.num = i_p.num;

    for ( i=0; i<4; i++ )
        for ( j=0; j<4; j++ )
        {
            m[i][j] = i_p.n[i][j];
        }
```

```
        for (i=0; i<4; i++)
            o_p.e.path[i].flag = NOT_MODIFIED;

        comput_pt( m[0], m[2], &o_p.e.path[0] );
        comput_pt( m[0], m[3], &o_p.e.path[1] );
        comput_pt( m[1], m[2], &o_p.e.path[2] );
        comput_pt( m[1], m[3], &o_p.e.path[3] );

        net_send(sizeof(REPLY), &o_p, 1);

    }
}
```

LIST OF REFERENCES

1. Smith, T.R., G. Peng, P. Gahinet,"A Family of Local, Asynchronous, Iterative and Parallel Procedures For Solving the Weighted Region Least Cost Path Problem", Technical Report, Department of Computer Science, University of California at Santa Barbara, 20 April 1988.

2. Dijkstra, E.W., "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik*, 1 (1959), pp. 269-271.

3. Aho, A.V., Hopcraft, J.E., Ullman, J.D., *Data Structures and Algorithms*, Addison-Wesley Publishing Co., 1983.

4. Mitchell, J.S.B., "An Algorithmic Approach to Some Problems in Terrain Navigation", *Artificial Intelligence*, 37 (1988), pp. 171-201.

5. Mitchell, J.S.B., Papadimitriou, C.H., "The Weighted Region Problem", Technical Report, Department of Operations Research, Stanford University, CA, October 1985.

6. Richbourg, R.F., Rowe, N.C., Zyda, M.J. and McGhee, R., "Solving Global Two-Dimensional Routing Problems Using Snell's Law and A* Search", *Proceedings IEEE International Conference on Robotics and Automation*, Raleigh, NC (1987), pp. 1631-1636.

7. Mitchell, J.S.B., "Shortest Paths Among Obstacles, Zero-Cost Regions, and Roads", Technical Report, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, December 1987.

8. Hoare, C.A.R., "Communicating Sequential Processes", *Communcations of the ACM*, 21,8 (August 1978), pp. 666-677.

9. INMOS Ltd., *Transputer Reference Manual*, January 1987.

10. Knuth, D.E., *The Art of Computer Programming, Vol. 3 / Sorting and Searching*, Addison-Wesley Publishing Co., 1973.

11. Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Prentice-Hall Inc., 1978.

12. 3L Ltd., *Parallel C Users Guide*, 1988.

13. Ross, R.S., *Planning Minimum-Energy Paths in an Off-Road Environment with Anisotropic Travel Costs and Motion Constraints*, Ph. D. Dissertation, Naval Postgraduate School, Monterey, CA, June 1989.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	1
4.	Man-Tak Shing, Code 52Sh Department of Computer Science Naval Postgraduate School Monterey, California 93943	25
5.	Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
6.	Captain Ivan Garcia Marine Corps Tactical Systems Support Activity Marine Corps Base Camp Pendleton, California 92054	2
7.	Lieutenant Commander Jeffrey M. Schweiger, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
8.	AEGIS Modelling Laboratory, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
9.	Captain Jonathan E. Hartman, Code 37 Naval Postgraduate School Monterey, California 93943	1