Parallel Graph Algorithms

By

WISSAM ITANI

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Project Advisor : Dr. NASHAT MANSOUR

Department of computer Science

LEBANESE AMERICAN UNIVERSITY

June 2000

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the project of

Wissam Itani

Candidate for the *Master of Science* degree*.

date 27/6/2020

*We also certify that written approval has been obtained for any proprietary material contained therein.

Acknowledgments

Appreciation is due to several people for their assistance, directly or indirectly, in the completion of this project. I wish to acknowledge the support and patience of my advisor, Dr. Nashat Mansour. Dr. May Abboud reviewed my work and provided me with valuable advises. I am grateful for the cooperation given me by the staff of Computer Center and the Library.

Finally, a special note of thanks to my family and friends for their continuing encouragement.

To my parents

Abstract

One way the early use of parallel computers has resembled the early use of sequential computers has been the emphasis on numerical algorithms. However, as the field of parallel algorithms matures, the emphasis can be expected to shift to nonnumerical algorithms because more and more problems being solved on computers are symbolic in nature. This document examines a number of parallel algorithms developed to solve problems in graph theory. These problems relate to shortest paths in graphs and minimum-cost spanning trees.

Chapter **four** is devoted to Moore's algorithm thus, it presents two parallel algorithms, one for solving the single-source shortest path (SSSP) problem and the second solve the all pairs shortest path (APSP) problem. Our focus in chapter **five** is on Dijkstra's algorithm; first, we describe the sequential SSSP algorithm, then we present a parallel formulation for solving APSP problems. Chapter **six** presents Prim's graph algorithm for computing the minimum spanning tree.

Table Of Contents

.

List of Figures

Introduction

Ever since conventional serial computers were invented, their speed has steadily increased to match the needs of emerging applications. However, the fundamental physical limitation imposed by the speed of light makes it impossible to achieve further improvements in the speed of such computers indefinitely. Recent trends show that the performance of these computers is beginning to saturate. A natural way to circumvent this saturation is to use an ensemble of processors to solve problems.

A cost-performance comparison of serial computers over the last few decades shows an interesting evolutionary trend. Figure 1.1 represents typical cost-performance curves of serial computers over the past three decades.

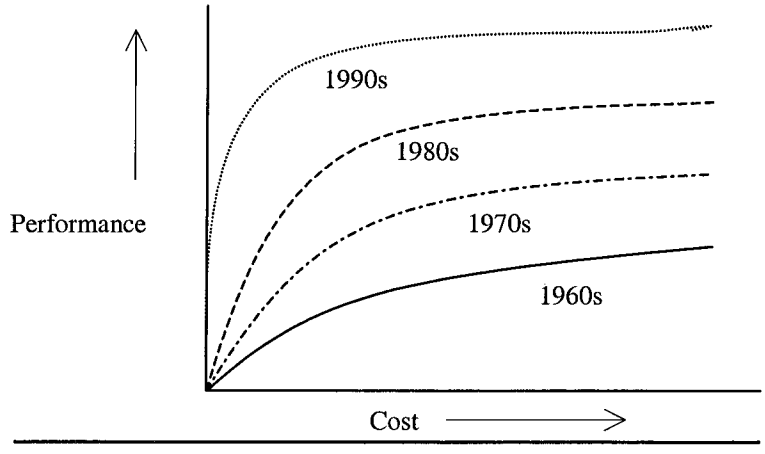


Figure 1.1 Cost versus performance curve and its Evolution over the decades

At the lower end of each curve, performance increases almost linearly (or even faster than linearly) with cost. However, beyond a certain point, each curve starts to saturate, and even small gains in performance come at an exorbitant increase in cost. Furthermore, this transition point has become sharper with the passage of time, primarily as a result of advances in very large scale integration (VLSI) technology. It is now possible to construct very fast, low-cost processors. This increases the demand for and production of these processors, resulting in lower prices.

Currently, the speed of off-the-shelf microprocessors is within one order of magnitude of the speed of the fastest serial computers. However, microprocessors cost many orders of magnitude less. This implies that, by connecting only a few microprocessors together to form a parallel computer, it is possible to obtain raw computing power comparable to that of the fastest serial computers. Typically, the cost of such a parallel computer is considerably less.

Furthermore, connecting a large number of processors into a parallel computer overcomes the saturation point of the computation rates achievable by serial computers. Thus, parallel computers can provide much higher raw computation rates than the fastest serial computers as long as this power can be translated into high computation rates for actual applications.

1.1 Parallel Computing

This section illustrates some important aspects of parallel computing by drawing an analogy to a real-life scenario.

Consider the problem of stacking (reshelving) a set of library books. A single worker trying to stack all the books in their proper places cannot accomplish the task faster than a certain rate. We can speed up this process, however, by employing more than one worker. Assume that the books are organized into shelves and that the shelves are grouped into bays. One simple way to assign the task to the workers is to divide the books equally among them. Each worker stacks the books one at a time. This division of work may not be the most efficient way to accomplish the task, since the workers must walk all over the library to stack books. An alternate way to divide the work is to assign a fixed and disjoint set of bays to each worker .As before, each worker is assigned an equal number of books arbitrarily. If a worker finds a book that belongs to a bay assigned to him or her, he or she places that book in its assigned spot. Otherwise, he or she passes it on to the worker responsible for the bay it belongs to. The second approach requires less effort from individual workers.

The preceding example shows how a task can be accomplished faster by dividing it into a set of subtasks assigned to multiple workers. Workers cooperate, pass the books to each other when necessary, and accomplish the task in unison. Parallel processing works on precisely the same principles. Dividing a task among workers by assigning them a set of books is an instance of ***task partitioning.*** Passing books to each other is an example of ***communication*** between subtasks.

Problems are parallelizable to different degrees. For some problems, assigning partitions to other processors might be more time-consuming than performing the processing locally. Other problems may be completely serial. For example, consider the task of digging a post hole. Although one person can dig a hole in a certain amount of time, employing more people does not reduce this time. Because it is impossible to partition this task, it is poorly suited to parallel processing. Therefore, a problem may have different parallel formulations, which result in varying benefits, and all problems are not equally amenable to parallel processing.

1.2 Parallel Algorithms

Designing algorithms is a fundamental aspect of computer science. Every branch of the field is concerned at some point or another with designing a method--that is, an algorithm, later translated into a program – to solve a problem. Algorithms for conventional computers are known as sequential or serial algorithms. They are familiar to anyone who has taken a first course in computing. Algorithms for parallel computers, or parallel algorithms (i.e., methods for solving computational problems on parallel computers) are perhaps less familiar.

Analysis of parallel algorithms A number of criteria are commonly used in evaluating the goodness of an algorithm. The most important of these are the algorithm's running time, how many processors it uses, and the total number of steps it performs. A less widely used, but no less important, criterion is the algorithm's probability of success in completing the, in those situations where

such a criterion is meaningful. These four criteria and the techniques employed in measuring them and interpreting the results of such evaluations are referred to collectively as *algorithms analysis*.

1.3 Issues in Parallel Computing

To use parallel computing effectively, we need to examine the following issues:

Design of Parallel Computers It is important to design parallel computers that can scale up to a large number of processors and are capable of supporting fast communication and data sharing among processors. This is one aspect of parallel computing that has seen the most advances and is the most mature.

Design of Efficient Algorithms A parallel computer is of little use unless efficient parallel algorithms are available. The issues in designing parallel algorithms are very different from those in designing their sequential counterparts. A significant amount of work is being done to develop efficient parallel algorithms for a variety of parallel architectures.

Methods for Evaluating Parallel Algorithms Given a parallel computer and a parallel algorithm running on it, we need to evaluate the performance of the resulting system. Performance analysis allows us to answer questions such as *How fast can a problem be solved using parallel processing?* and *How efficiently are the processors used?*

Parallel Computer Languages Parallel algorithms are implemented on parallel computers using a programming language. This language must be flexible enough to allow efficient implementation and must be easy to program in. New languages and programming paradigms are being developed that try to achieve these goals.

Parallel Programming Tools To facilitate the programming of parallel computers, It is important to develop comprehensive programming and tools. These must serve the dual purpose of shielding users from low-level machine characteristics and providing them with design and development tools such as debuggers and simulators.

Portable Parallel Programs Portability is one of the main problems with current parallel computers. Typically, a program written for one parallel computer requires extensive modification to make it run on another parallel computer. This is an important issue that is receiving considerable attention.

Automatic Programming of Parallel Computers Much work is being done on the design of parallelizing compilers, which extract implicit parallelism from programs that have not been parallelized explicitly. Such compilers are expected to allow us to program a parallel computer like a serial computer. We speculate that this approach has limited potential for exploiting the power of large-scale parallel computers.

1.4 Previous Work

So far, most of the papers and researches on parallel computing were done on PRAM model (a theoretical machine that consists of a number of identical processors, a common memory and a memory access unit. This model is used for theoretical research). Although, researches in multiproceesing exist extensively, there isn't much work about multicomputer programming. Usually, and in the limit of our knowledge, differences are found between theoretical analysis and experimental results; no profound theoretical analysis beyond correctness proofs are given, the experimental results yield only quite limited speedup. In the following, we present some of the work done on parallel graph algorithms.

Single-Source Shortest Path Problem. The single-source shortest path (SSSP) problem has so far resisted fast efficient parallel solutions. The work of a parallel algorithm is given by the product of its running time and the number of processing units (PUs) p. There is no parallel O(n log n + m) work PRAM algorithm for general digraphs with non-negative edge weights.

For random digraphs, simple criteria are proposed [Crauser 98] which divide Dijkstra's sequential SSSP algorithm into a number of phases, such that the operations within a phase can be done in parallel.

Meyer and Sanders [Meyer 99] gave an algorithm which is a generalization of Dial's algorithm and the Bellman-Ford algorithm. It is best suited for random

directed graphs with uniformly distributed edge weights. As a side effect of this algorithm, it gives a simple linear time sequential algorithm for a large class of not necessarily random directed graphs with random edge weights.

Minimum Spanning Tree (MST). Many PRAM algorithms for computing the MST exist. The algorithm due to Awerbuch and Shiloach [Awerbuch 87] runs in O(log n) time on an (n + m)-processor PRIORITY CRCW PRAM, where the priority of a processor is determined by the weight of the edge assigned to it. This is a very powerful contention resolution rule. Johnson and Metaxas [Johnson 95] gave an algorithm running in $O(\log^{3/2} n)$ time on an (n + m)-processor EREW PRAM. Karger [Karger 93] was the first to use randomization in parallel MST algorithms. He gave an EREW PRAM algorithm that requires O(log n) time and uses $(m/\log n + n^{1+\varepsilon})$ processors. A randomized sequential linear-time algorithm is given in [Karger 95]. A parallel version of this algorithm, running in O(log n) time and performing linear work on a CRCW PRAM, is decribed in [Cole 96].

Connected Components. The problem of finding the connected components of a graph has broad importance in both computer and computational science. In computer vision, for example, edge detection and object recognition depend on connected components. Connected components algorithms have also advanced the study of physical phenomena, including properties of magnetic materials near critical temperatures. However, the problem offers a unique challenge for parallel computing. Sequential solutions are well understood and commonly used as an application of depth-first and breadth-first search. Parallel solutions have received a great deal of attention from theorists, and have proven difficult. Algorithms such as Shiloach-

Vishkin obtain good results with the CRCW PRAM model [Shiloach 82], which assumes uniform memory access time and arbitrary bandwidth to any memory location, but the inherent contention in the algorithm makes even EREW solutions much more challenging [Chong 93]. Implementation of the theoretical work has been restricted to shared-memory machines and SIMD machines with very slow processors.

Graph Coloring. Graph coloring is an important problem with applications in register allocation, scheduling, and the efficient computation of sparse Jacobian matrices. Finding a coloring or determining that no valid coloring exists often involves a large search which is computationally intensive. Jones and Plassmann [Jones 93] presented an asynchronous parallel algorithm that produces a coloring of a given graph, aiming at using few colors, but without any guarantee that the number of colors used is the smallest possible. Laxmikant et.al [Laxmikant 95] presented a set of parallel graph coloring heuristics and described their implementation in an environment supporting machine-independent parallel programming. The heuristics are intended to provide consistent, monotonically increasing speedups as the number of processors is increased.

1.5 Objectives And Organization Of The Project

The objective of this project is to present parallel computing applied on graph theory. For this purpose, first, we picked up some important and fundamental sequential algorithms developed to solve problems in graph theory. These problems relate to

minimum–cost spanning trees, and shortest paths in graphs. Next, we analyzed the parallelizable sections in each of these algorithms, and then by applying parallel techniques on these algorithms we formulated the corresponding parallel graph algorithms. Then, we implemented these algorithms using Visual C++ combined to the MPI Library (provides message-passing routines). Finally, we tested our code and we evaluated the corresponding experimental results. In this document, we concentrated upon the use of multiple computers that communicate between themselves by sending messages; hence the term message-passing parallel programming. The computers we use are of different capacities and speeds but must be interconnected by a network, and a software environment must be present for intercomputer message passing.

The remainder of this document is divided into 7 chapters. We begin with chapter **two** by presenting graph theory, emphasizing graph concepts and graph representations. Chapter **three** exposes the programming language used for implementing our algorithms. Chapter **four** is devoted to Moore's algorithm thus, it presents sequential algorithm for solving single-source shortest path (SSSP) problem, and two parallel algorithms for solving the (SSSP) problem and the all pairs shortest path (APSP) problem. Our focus in chapter **five** is on Dijkstra's algorithm; first, we describe the sequential SSSP algorithm, then we present a parallel formulation for solving APSP problems. Chapter **six** presents Prim's graph algorithm for computing the minimum spanning tree. Chapter **seven** discuss the experimental results. Finally chapter **eight** gives concluding remarks.

Graphs

2.1 Graph Theory

It is probably fair to say, and has been said before by many others, that graph theory began with Euler's solution in 1735 of the class of problems suggested to him by the Konigsberg bridge puzzle. But had it not started with Euler, it would have started with Kirchhoff in 1847, who was motivated by the study of electrical networks; had it not started with Kirchhoff, it would have started with Cayley in 1857, who was motivated by certain applications to organic chemistry, or perhaps it would have started earlier with the four-color map problem, which was posed to De Morgan by Guthrie around 1850. And had it not started with any of the individuals named above, it would almost surely have started with someone else, at some other time. For one has only to look around to see "real-world graphs" in abundance, either in nature (trees, for example) or in the works of man (transportation networks, for example). Surely someone at some time would have passed from some real-world object, situation, or problem to the abstraction we call graphs, and graph theory would have been born.

During the last few decades, Combinatorial Optimization and Graph Theory have experienced a particularly fast development. There are various reasons for this fact; one is, for example, that applying combinatorial ways of deduction has become more and more common. However, two developments on the "outside" of mathematics may have been more important: First, a lot of problems in Combinatorial

Optimization arose immediately from everyday practice in engineering and management, for example determining shortest or most reliable paths in traffic or communication networks, maximal or compatible flows or shortest tours, planning connections in traffic networks, coordinating projects or supply and demand problems. Second, practical instances of these tasks, which belong to Operations Research, have becvome accessible by the development of more and more efficient computer systems. Furthermore, Combinatorial Optimization problems are also important for Complexity Theory, an area in the common intersection of Mathematics and Theoretical Computer Science which deals with the analysis of algorithms.

Today graph theory is a vast and somewhat sprawling subject, embracing as it does applications in many diverse areas: physics, chemistry, engineering, operations research, genetics, economics, psychology, and sociology, to name some. Dozens of books and proceedings of conferences on graph theory have appeared, mostly within the last fifteen years, and the number of journal articles dealing with graphs that have appeared in this time interval must number in the thousands. Today there are journals devoted exclusively to graph or network theory, and other journals, devoted exclusively to combinatorial mathematics, in which many, if not most, of the papers that appear are about graphs.

2.2 Graph Concept

Fundamentally, a graph is an extremely simple object: a finite set of vertices and a finite set of edges that connect pairs of vertices. In a directed graph, each edge has an orientation and goes from one vertex to another. An undirected graph is an ordered

pair (V,E) consisting of a set V of vertices and a set E of edges, each edge being an unordered pair of distinct vertices. For example, the edge involving vertices x and y will be denoted by {x,y}. when no confusion is possible, we will use xy as a shorthand for {x,y}. vertices x and y are said to be adjacent if xy is an edge. We also say that the edge xy is incident with x and y. as usual, we shall write G = (V,E) to denote a graph G with vertex-set V and edge-set E. A subgraph of G is a graph G' = (V',E') such that V'$\in$ V and E'$\in$ E. Given a set A of vertices of G, the subgraph induced by A is denoted by G(A) = (A,E(A)) where E(A) = {xy | x $\in$ A and y $\in$ A}. For a vertex x in G, the degree d(x) stands for the number of edges incident with x. In Fig. 2.1, for example, the degree of vertex 6 is 3.

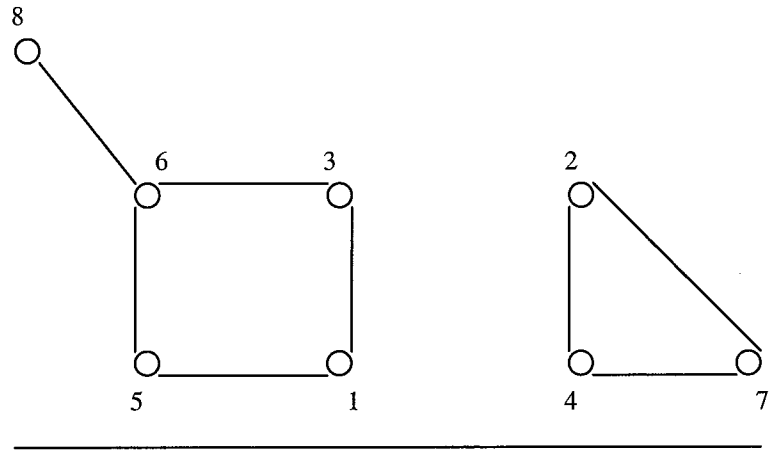


Figure 2.1 a Graph G

A path is a sequence $v_0,v_1,\ldots,v_p$ of distinct vertices of G with $v_{i-1} \in E$ for all i ($1 \leq i \leq p$). A graph is connected if there is a path between any pair of its vertices. The Connected components of G are the maximal connected subgraphs of G. For example,

the Graph G in Fig. 2.1 has two connected components, induced by {1,3,5,6,8} and {2,4,7}, respectively. A cycle of length p+1 is a sequence $v_0,v_1,\ldots,v_p$ of distinct vertices of G such that $v_{i-1}v_i \in E$ for all i ($1 \leq i \leq p$) and $v_pv_0 \in E$.

A graph G = (V,E) is biconnected if the graph obtained from G by removing an arbitrary vertex along with all edges incident to it is still connected. For example, the subgraph induced by {2,4,7} in Fig. 2.1 is biconnected. A vertex v of G is said to be a cut vertex if the graph induced by V – {v} is not connected. For example, vertex 6 is a cut vertex in the graph induced by {1,2,5,6,8}. An edge {x,y} of G is said to be a bridge if the graph obtained from G by removing the edge {x,y} is not connected.

A graph that contains no cycle is termed acyclic. A (legal) coloring of a graph is an assignment of integers to the vertices of the graph in such a way that adjacent vertices always receive distinct integers (i.e., colors).

A clique in a graph is a set of pairwise adjacent vertices. For example, {2, 4, 7} is a clique in the graph featured in Fig. 2.1. A set of pairwise nonadjacent vertices is said to be independent. Referring again to Fig. 2.1, the set {2, 3, 5, 8} is an independent set. A clique cover for a graph G is a set of cliques that contain all the vertices of the graphs. A clique cover is minimal when it contains the fewest possible cliques that together cover all the vertices. For example, {6, 8}, {6, 3}, {5, 1}, and {2, 4, 7} constitute a clique cover. A set D of vertices of a graph is said to be dominating if every vertex outside D is adjacent to a vertex in D. for example {1, 2, 5, 6} is a dominating set in the graph of Fig. 2.1.

A matching in a graph is a set of edges sharing no common vertex. For example, the set {{6,8},{1,5},{2,7}} is a matching in the graph of Fig. 2.1. consider a graph G along with a matching M in G. It is customary to call the edges in M matched and the edges outside M free. Vertices not incident with matched edges are referred to

as exposed. A path P: $x_1, x_2,\ldots,x_p$ is called alternating if the edge $\{x_i, x_{i+1}\}$ is free for all odd values of I and matched for all even values of i. If both x_1 and x_p are exposed vertices, then the path P is termed augmenting.

A tree is a connected acyclic graph. When dealing with trees, it is customary to refer to vertices as nodes. In a tree, every two nodes are connected by a unique path. When a distinguished node termed the root of the tree is selected, the tree is said to be rooted. For a node u of a tree, its ancestors are all the nodes on the unique path to the root; the descendaents of u are the nodes in the subtree of the original tree rooted at u. a leaf in a tree is a node of degree 1. If u is a node of a rooted tree T, then T_u represents the subtree of T rooted at u and containing all the descendents of u in T. Given any two nodes x and y of a rooted tree T, the lowest common ancestor of x and y, denoted lca(x,y), is the unique node z of T that is an ancestor of both x and y and such that no descendent of z is an ancestor of both x and y.

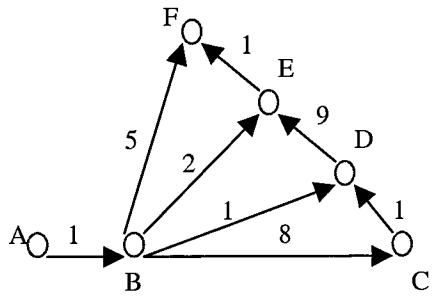
2.3 Graph Representation

There are two basic ways that a graph can be represented in a program:

- Adjacency matrix – a two-dimensional array, a, in which a[i][j] holds the weight associated with the edge between vertex i and vertex j if one exists
- Adjacency list – for each vertex, a list of vertices directly connected to the vertex by an edge and the corresponding weights associated with the edges

Both methods are shown in Figure 2.2. The adjacency list is implemented as a linked list. The order of the edges in the adjacency list is arbitrary.

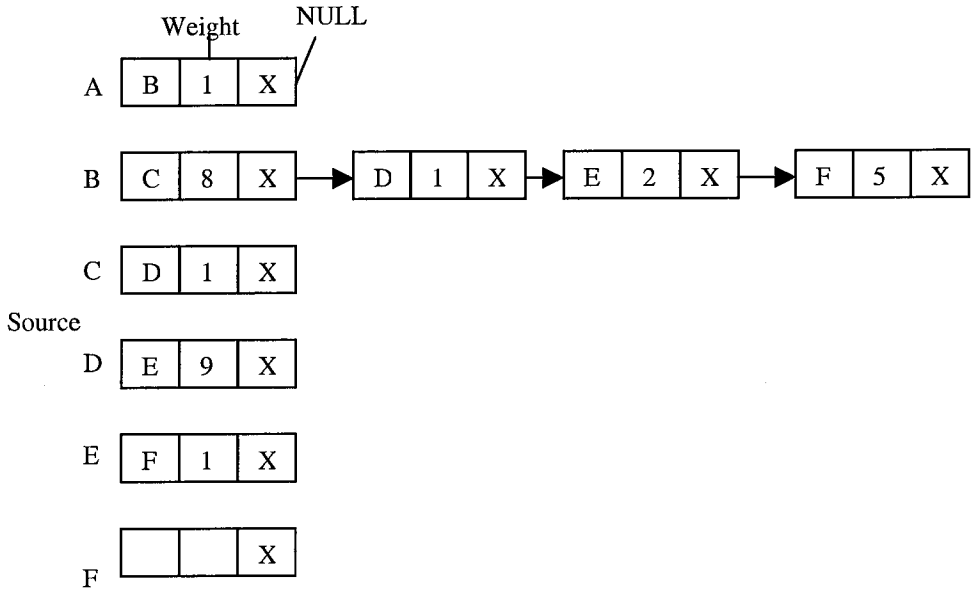
The particular method chosen will depend upon characteristics of the graph and the structure of the program. For sequential programs, normally the adjacency matrix is used for dense graphs, graphs where there are many edges from each vertex. The adjacency list is used for sparse graphs, graphs where there are few edges from each vertex. The difference is based upon space (storage) requirements. An adjacency matrix has an $O(n^2)$ space requirement and an adjacency list has an O(nv) space requirement, where there are v edges from each vertex and n vertices in all. In general, v will be different for each vertex, and therefore the upper bound on the space requirement of an adjacency list is given by $O(nv_{max})$. Accessing the adjacency list is slower than accessing the adjacency matrix, as it requires the linked list to be traversed sequentially, which potentially requires (v steps).



(a)

	A	B	C	D	E	F
A	∞	10	∞	∞	∞	∞
B	∞	∞	8	13	24	51
C	∞	∞	∞	14	∞	∞
D	∞	∞	∞	∞	9	∞
E	∞	∞	∞	∞	∞	17
F	∞	∞	∞	∞	∞	∞

(b) Adjacency



(c) Adjacency list

Figure 2.2 Representing a graph

MPI Programming Model

MPI is a standard specification for interfaces to a library of message-passing routines. The goals of MPI are efficient communication, portability, and rich functionality.

Although several message-passing libraries exist on different systems (for example, Intel Paragon's NX, Connection Machine's CMMD, or PVM), MPI has emerged as a favorite for the following reasons :

- Support for full asynchronous communication – Process communication can overlap process computation.
- Group membership – Processes may be grouped based on context.
- Synchronization variables that protect process messaging – When sending and receiving messages, synchronization is enforced by source and destination information, message labeling, and context information.
- Portability – All implementations are based on a published standard that specifies the semantics for usage.

The MPI library contains 128 routines. These routines provide a set of functions that support point-to-point communications, collective operations, process groups and communication contexts, process topologies, and data type manipulation. See "Appendix A. Basic MPI library routines" for more information.

Although the MPI library contains a large number of routines to choose from, you can design a number of applications by only using a limited number of basic routines. As your application grows in complexity, you can introduce other routines from the library.

In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor. However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as multiple program multiple data (MPMD) to distinguish it from the SPMD model in which every processor executes the same program.

Processes can use point-to-point communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications. A group of processes can call collective communication operations to perform commonly used global operations such as summation and broadcast. MPI's ability to probe for messages supports asynchronous communication. Probably MPI's most important feature from a software engineering viewpoint is its support for modular programming. A mechanism called a communicator allows the MPI programmer to define modules that encapsulate internal communication structures. These modules can be combined by both sequential and parallel composition.

Algorithms that create just one task per processor can be implemented directly, with point-to-point or collective communication routines used to meet communication requirements. Algorithms that create tasks in a dynamic fashion or

that rely on the concurrent execution of several tasks on a processor must be further refined to permit an MPI implementation.

Parallel Moore's Shortest Path

4.1 Single Source Shortest Path Problem: Sequential Algorithm

In a single-source shortest-path problem we must find the shortest path from a specified vertex s, the source, to all other vertices in a weighted, directed graph. Let weight (u, v) represent the length of the edge from u to v; if no such edge exists, then weight (u, v) = ∞.

A sequential algorithm, developed by Moore (1959), solves this problem. In Moore's single-source shortest-path algorithm, distance(v) is initially assigned the value ∞, for all v ∈ V-{s}. The distance from s to itself is, of course, zero. A queue contains vertices from which further searching must be done; initially it contains s. As long as the queue remains nonempty, the vertex u from the head of the queue is removed, and all edges (u, v) ∈ E are examined. If distance(u) + weight(u ,v) < distance(v), a new shorter path to v has been found (through u). In this case distance(v) is revised and v is added to the tail of the queue, if it's not already in the queue. The algorithm continues this process until the queue is empty. Moore's algorithm is shown in the following.

SHORTEST PATH (sequential)

```
Parameter   n          {number of vertices in a graph}
Global      distance   {element i contains distance from s to i}
            s          {source vertex}
            weight     {contains weight of every edge}
1.    Begin
2.       For i = 1 to n  do
3.          INITIALIZE(i)
4.       end for
5.       insert s into the queue
6.       while the queue is not empty do
7.          SEARCH( )
8.       end while
9.    end
```

SEARCH( )

```
Local  new.distance  {distance to v if pass through u}
       u             {examined edge leaves this vertex}
       v             {examined edge enters this vertex}
10.   Begin
11.      dequeue vertex u
12.      For every edge{u, v} in the graph  do
13.         new.distance = distance(u) + weight({u,v})
14.         if new.distance < distance(v)  then
15.            distance(v) = new.distance
```

```
16.                 if v is not in the queue  then
17.                     enqueue vertex v
18.                 end if
19.             end if
20.         end for
21.     end
```

Procedure INITIALIZE, called in line 3 of the algorithm, initializes the distance of every nonsource vertex to ∞ and the distance of s to zero. The 'For' loop in lines 12 to 20 corresponds to the search for shorter paths to vertices directly reachable from vertex u.

4.2 Single Source Shortest Path Problem: Parallel Formulation

In the sequential algorithm, a queue is used to save the vertices for later consideration. However, there is no reason these queue entries must be considered in any particular sequence. In fact. there is no reason why queued vertices could not be all considered in *parallel.* This observation is the key to formulating a parallel version of this shortest path algorithm. The vertex queue can be transformed into a *work* Pool of vertex numbers. A group of parallel *Worker* processes can remove vertices from the Work Pool. process them and add new vertices to the Work Pool. When the Work Pool is empty, the algorithm terminates. This is the concept of **Centralized Work Pool**. The work pool can be partitioned and distributed across the local memories of

the multicomputer. This parallel programming technique is called **Decentralized Work Pool**.

To implement decentralized work pool, the most difficult issue is the termination detection. A new and complex distributed termination algorithm is required. Another important area of consideration for distributed work pool is the method of distributing the work items. The Work Pool is completely partitioned among the processes, and therefore each work item must be sent directly to a specific process. For distributed work pool it is often the case that the processes are distinguishable, and it is necessary for certain work items to go to specific processes. This requirement results from the ***data partitioning*** that is often necessary in multicomputer algorithms.

In this section we will consider first the issue of centralized work pool, next the decentralized work pool and data partitioning, then the termination detection algorithm and finally we will look at the parallel shortest path algorithm.

4.2.1 Centralized Work Pool

The first parallel implementation to consider uses a centralized work pool holding the vertex queue, vertex_queue[] as tasks. Each slave takes vertices from the vertex queue and returns new vertices in the manner illustrated in Figure 4.1. For the slaves to identify edges and compute distances, they need access to both the structure holding the graph weights (adjacency matrix or adjacency list) and the array holding the current minimum distances, dist[]. If this information is held by the master process, messages will need to be sent to the master to access the information. This could lead to a very significant communication overhead. Since the structure holding the graph weights is

fixed, this structure could be copied into each slave. For now, let us assume that the distance array, dist [], is held centrally and simply copied with the vertex in its entirety. Instead, individual requests for distances could also be made. The code could be of the form

Master

```
while (vertex_queue() != empty) {
   recv(P_ANY, source = P_i) ;              /* request task from slave */
   v = get_vertex_queue() ;
   send(&v, P_i);                           /* send next vertex and */
   send(&dist, &n, P_i);                    /* current dist array */
        .
   recv(&j, &dist[j], P_ANY, source = P_i); /* new distance */
   append_queue(j, dist[j]);                /* append vertex to queue */
                                            /*and update distance array */
};
recv(P_ANY, source = P_i);                  /* request task from slave */
send(P_i, termination_tag);                 /* termination message*/
```

Slave (process i)

```
send(Pmaster);                      /* send request for task */
recv(&v, Pmaster, tag);             /* get vertex number */
if (tag != termination_tag) {
  ( recv(&dist, &n, Pmaster);       /* and dist array */
   for (j = 1; j < n; j++)          /* get next edge */
     if (w[v] [j] ! = infinity) {   /* if an edge */
        newdist_j = dist [v] + w[v] [j];
        if (newdist_j < dist[j] ) {
           dist[j] = newdist_j;
           send(&j, &dist[j], Pmaster); /* add vertex to queue */
        }                               /* send updated distance */
```

```
        }
    }
```

Clearly, the vertex number and distance array could be sent in one message. Notice also that individual slaves may have distances that are not exactly the same because they are being updated continually by different slaves.

The master waits for requests from any slave but must respond to the specific slave that makes a request. In our pseudocode notation, source = P_i is used to indicate the source of the message. In an actual programming system, the source could be identified by making each slave send its identification (possibly as a unique tag). In the case of MPI, the actual source of the message can be found by reading the status word returned by the MPI_Recv( ) routine.

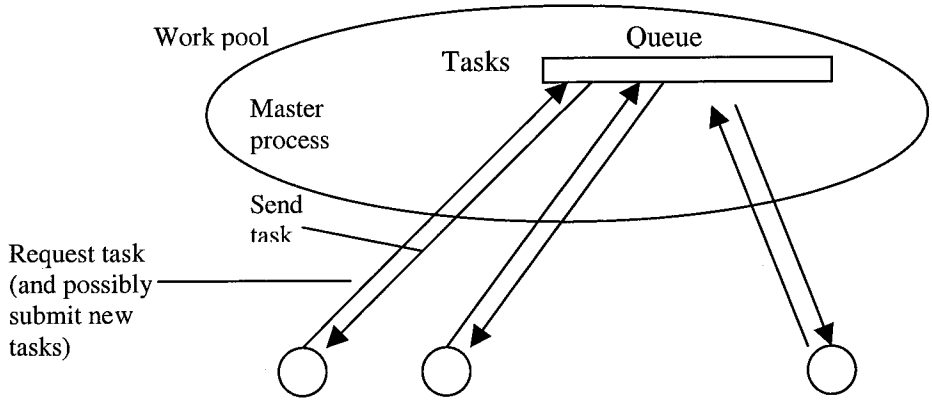


Figure 4.1 Centralized work pool

4.2.2 Decentralized Work Pool

One of the distributed work pool approaches can be applied to our problem. The task queue, in our case vertex_queue[], could also be distributed. A convenient approach is to assign slave process i to search around vertex i only and for it to have the vertex queue entry for vertex i if this exists in the queue. In other words, one element of the queue is reserved specifically to hold vertex i. and this entry is in process i. The array dist[] will also be distributed among the processes so that process i maintains the current minimum distance to vertex i. Process i also stores an adjacency matrix/list for vertex i, for the purpose of identifying the edges from vertex i.

With our arrangements, the algorithm can proceed as follows. The search will be activated by a coordinating process loading the source vertex into the appropriate process. Referring to figure 2.2, vertex *A* is the first vertex to search. The process assigned to vertex *A* is activated. This process will immediately begin searching stored value and replace if the currently stored value is larger. In our case, the process responsible for vertex *B* will be contacted with the distance to vertex *B*. In this fashion, all minimum distances will be updated during the search. If the contents of d[i] changes, process i will he reactivated to search again. Message passing will distribute across many of the slaves processes, rather than be focused on the master process.

A code segment for the slave processes might take the form

Slave (process i)

```
recv (newdist, P_ANY);
if(newdist < dist)     {
  dist = newdist;
vertex_queue = TRUE;                    /* add to queue */
}else vertex_queue = = FALSE;
```

```
        if (vertex_queue = = TRUE)          /* start searching around vertex
*/
          for (j = 1; j < n; j++)           /* get next edge */
            if (w[j] != infinity) {
              d = dist + w[j];
              send(&d, Pj);                 /* send distance to proc j */
            }
```

A mechanism is necessary to repeat the actions and terminate when all processes are idle. The mechanism must cope with messages in transit. The simplest solution is to use synchronous message passing, in which a process cannot proceed until the destination has received the message. We will consider, in section 4.2.3 a more powerful method of identifying the unique parent that receives an acknowledgment last. The method described in this section is impractical for a large graph if one vertex is allocated to each processor. In our parallel implementation of Moore's shortest path algorithm we will allocate a group of vertices to each processor instead of allocating one vertex.

4.2.3 Distributed Termination Detection Algorithm

So far, we have considered distributing the tasks. Now let us look at how to terminate these distributed tasks. First we will examine the termination conditions.

4.2.3.1 Termination Conditions

When a computation is distributed, recognizing that the computation has come to an end may be difficult unless the problem is such that one process reaches a solution. In general, distributed termination at time t requires the following conditions to be satisfied [Bertsekas 89]:

- Application-specific local termination conditions exist throughout the collection of processes, at time t.
- There are no messages in transit between processes at time t.

The subtle difference between these termination conditions and those given for a centralized system is having to take into account messages in transit. The second condition is necessary for the distributed termination system because a message in transit might restart a terminated process. One could imagine a process reaching its local termination condition and terminating while a message is being sent to it from another process. The first condition is usually relatively easy to recognize. Each process can send a message to the master when its local termination conditions are satisfied. However, the second condition is more difficult to recognize. The time that it takes for messages to travel between processes will not be known in advance, One could conceivably wait a long enough period to allow any message in transit to arrive. This approach would not be favored and would not permit portable code on different architectures.

4.2.3.2 Message Acknowledgments

Bertsekas and Tsitsiklis [Bertsekas 89] describe a distributed termination method using request and acknowledgment messages. The method is very general, mathematically sound, and with messages being in transit when a process is about to terminate locally. Bertsekas and Tsitsiklis give formal mathematical arguments in detail. The method is illustrated in Figure 4.2. Each process is in one of two states:

1. Inactive
2. Active

Initially, without any task to perform, the process is in the inactive state. Upon receiving task from a process, it changes to the active state. The process that sent the task to make it enter the active state becomes its "parent." If the process passes on a task to an inactive process, it similarly becomes the parent of this process, thus creating a tree of processes each with a unique parent. An active process could potentially receive more tasks from other active processes while it is in the active state, but these other processes are not parent of the process. Hence, the computation itself need not be a tree structure. On every occasion when a process sends a task to another process, it expects an acknowledgment message from that process. On every occasion when it receives a task from a process, it immediately sends an acknowledgment message, except if the process it receives the task from is a parent process. It only sends an acknowledgment message to its parent when it is ready to become inactive. It becomes inactive when

- Its local termination condition exists (all tasks are completed).
- It has transmitted all its acknowledgments for tasks it has received.
- It has received all its acknowledgments for tasks it has sent out.

it is ready to become inactive. It becomes inactive when.

The last condition means that a process must become inactive before its parent process. When the first process becomes idle, the computation can terminate.

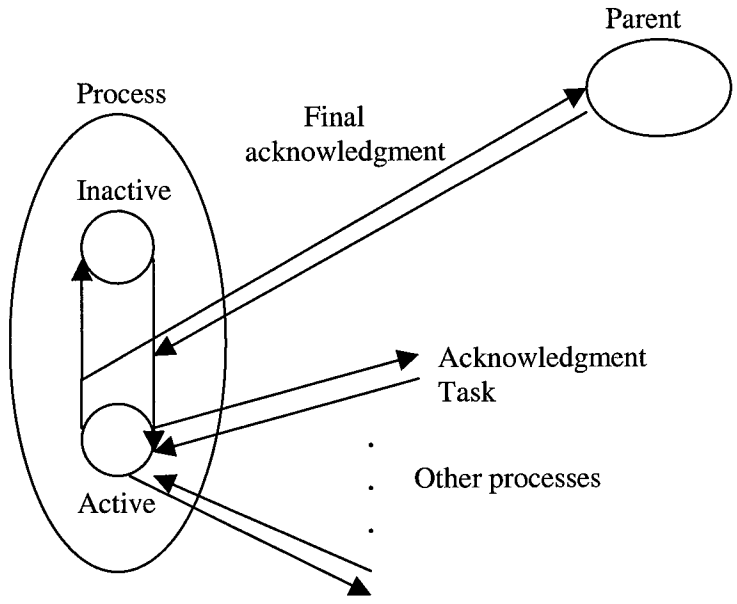


Figure 4.2 Termination using message acknowledgements

4.2.4 Pseudocode For Parallel Moore's Algorithm

In the previous sections we presented the design of the shortest path algorithm, and we saw the different phases of this algorithm. Now we will look at the pseudocode that presents our algorithm in its final stage.

P_SSSP_ Moore ( )

```
If  (Master process)
{
        GetGraph  from input file
        Broadcast(graph)
        Send(node=source,distance=0) to the corresponding slave process
        P_source
        Increment ack_count
```

```
                Wait for acknowledgment from P_source
                Receive results from slaves
                Combine results
                Finalize

        }
        /* end master process */

        else
        if (Slave process)
        {
                GetGraph from Master process
                While (work_queue is empty and Process not finish)
                {
                        GetWorkQueue( )
                }
                While (work_queue is not empty and Process is not finish)
                {
                while ( j < number of columns )        /* number of columns = 6 */
                {
                        if (edge[queue element][j]  exists)
                        {
                        newdist[j] = dist[queue element] + weight[queue element][j]
                        if newdist[j] < dist[j]
                        {
                        dist[j] = newdist[j]
                        send ( node = j, distance = dist[j] )     to corresponding process
                        increment ack_count
                        }
                    }
                    increment  j
                } /* end while */

                While (work_queue is empty and Process not finish)
                {
                        GetWorkQueue( )
                }
           }
           send result to Master process
     }
    /* end slave process */

GetWorkQueue ( )
{
Repeat
        If process in Active state AND work_queue is empty AND all
        acknowledgments received
        {
                Acknowledge current parent
                Go into Idle state
```

```
        }
        receive message from any source
        if message is acknowledgment then decrement ack_count
Until the message in not an acknowledgment

If process in Active state then
{
        Send acknowledgment to source of the work message
}
Else
{
        Go into idle Active state
        Set new parent from source of the work message
}
Return "work" to calling process
}
/* end of GetWorkQueue */
```

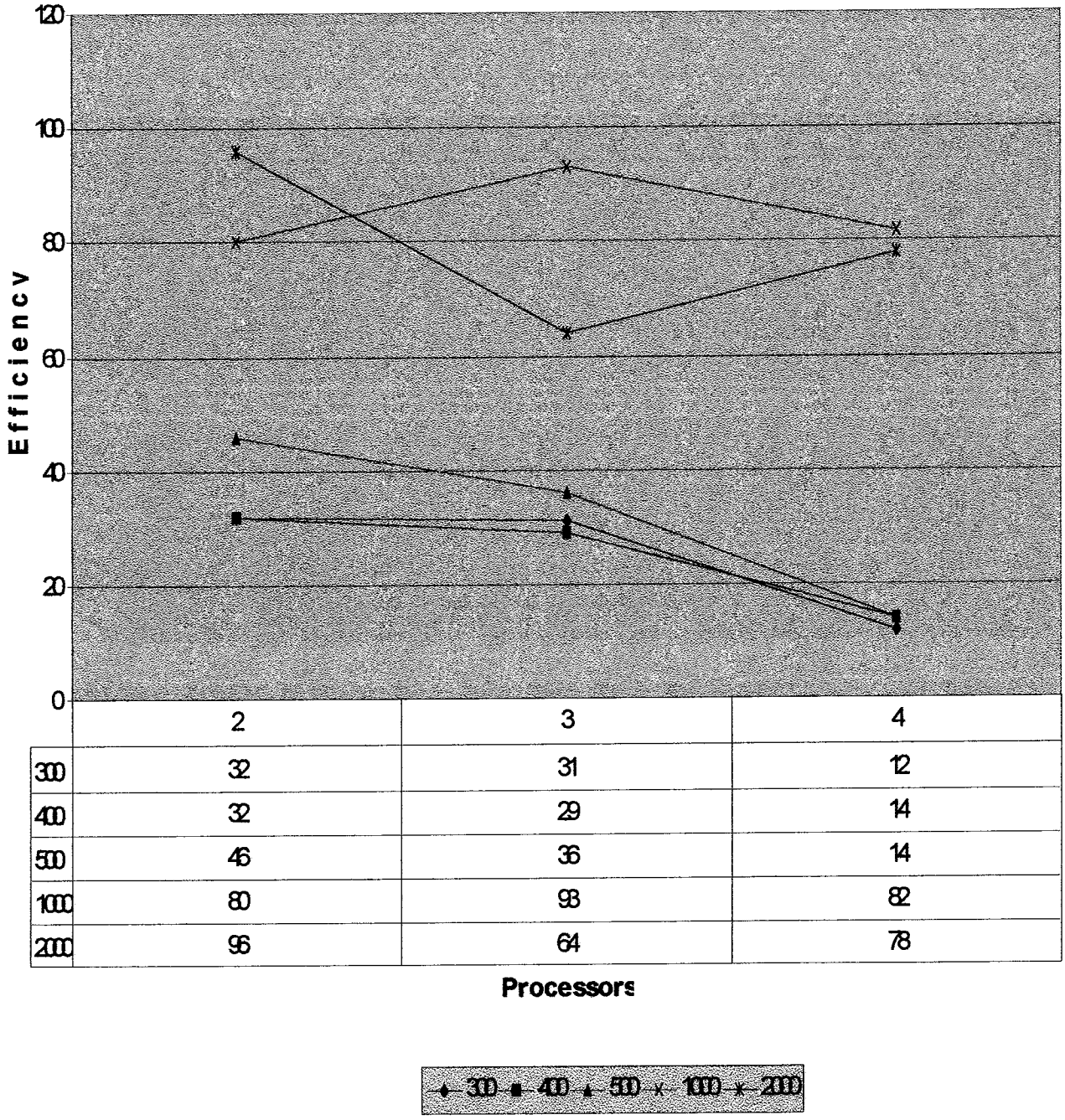
4.2.5 Experimental Results

We have implemented the algorithm described in section 4.2.4 for sparse graphs. The algorithm computes the shortest path from a single source to all sources for a connected directed graph.

We investigated the algorithm on a cluster of workstations, using 2 to 5 computers. Our programs are written in the programming language Visual C++ 6.0 and make use of the Coimbra University MPI-library (WMPI1.3).

Experiments were done on random graphs. For each of the n vertices a set of edges (containing 1 to 6 edges), is randomly chosen. The edges are assigned a random (integer) weight from {1,…,maxW-1), where (maxW –1) denotes the maximum allowable edge weight. The random numbers are generated by the C-function srand( ), using always '1' as the seed number to generate the same graph for each execution of the program.

The behavior of the algorithms was tested on graphs of size n = {500, 1000, 2000), and maxW = 60. All results state the average of at least 5 runs.



	2	3	4
300	32	31	12
400	32	29	14
500	46	36	14
1000	80	93	82
2000	96	64	78

Figure 4.3 Efficiency Of P_SSSP_MOORE Algorithm

4.2.6 Observations And Assessment

As we can see in Figure 4.3, Algorithm P_SSSP_MOORE performs badly for small graph sizes. For graph size less than 1000 nodes, the efficiency felt to 12 percent. In other terms for 88 percent of the execution time the computers are idle. However, for graphs with more than 1000 nodes, the efficiency goes to 96 percent, which means that the computers are idle for only 4 percent of the execution time. The communication overhead justifies these results. In small graphs the computation/communication ratio is very small which explains the very low efficiency, while in large graphs this ratio is high which leads to lessen the communication effects on the efficiency of the algorithm.

4.3 All Pairs Shortest Path Problem

4.3.1 Description Of The Problem

The all pairs shortest path problem can be stated as follows:

Given: A directed graph G = (V, E) where V is the set of vertices of G and E is the set of edges of G, the number of vertices being n, and a weight matrix W representing the weights of edges between each pair of vertices.

Find: For every pair of vertices, determine the shortest path along the edges of G. Let this solution be represented by an n x n matrix A, where an element $A_{i,j}$ of this matrix represents the shortest path from vertex i to vertex j.

4.3.2 Moore's All Pairs Shortest Path Algorithm

As we saw previously Moore's sequential algorithm is an algorithm for the single-source shortest path problem. An application of this algorithm for each vertex considered as a source in turn results in the solution to the all pairs shortest path problem.

Hence, in terms of the result matrix A, as each source vertex is considered in turn, the corresponding row of the matrix is updated for each application of the basic algorithm at a time.

The parallel version of Moore's algorithm for the all pairs shortest path problem (APSP) represent a distinct approach to the single source shortest path problem (SSSP), presented in previous sections, in terms of communication. The parallel implementation of

the APSP algorithm use the all-peers paradigm. In the all-peers paradigm, all tasks participate in the computation and are referred to as worker tasks in this scenario. Each of these tasks decides what portion of the problem it will solve, based on some common algorithm. In general, all tasks control the computation as a whole, communicating with each other as necessary both for computation and control purposes. Only one of the peer tasks may still read in the input data and distribute it to the rest of its peers, and then collect the results in the end.

Parallel Moore's APSP solution uses concurrent applications of the sequential algorithm for distinct sets of vertices, with each set assigned to different processors in the cluster. The collection of results from these tasks constitutes the all pairs solution. Each task needs the entire graph for its calculation. One of the peer tasks (coordinator) has the input graph W, and it broadcasts this graph to all the slave tasks then it executes like any other task. A process, upon receiving the input graph W, computes the shortest paths for its subset of vertices and sends its partial result back to the coordinator. The coordinator collects all the partial results and forms the complete result. The peer tasks each executes Algorithm P_APSP_MOORE.

P_APSP_MOORE

1. Broadcast graph G;

2. p = number of tasks;

3. m = my instance id;

4. for(i=m; i<graph size; i+=p) {

sequential moore;

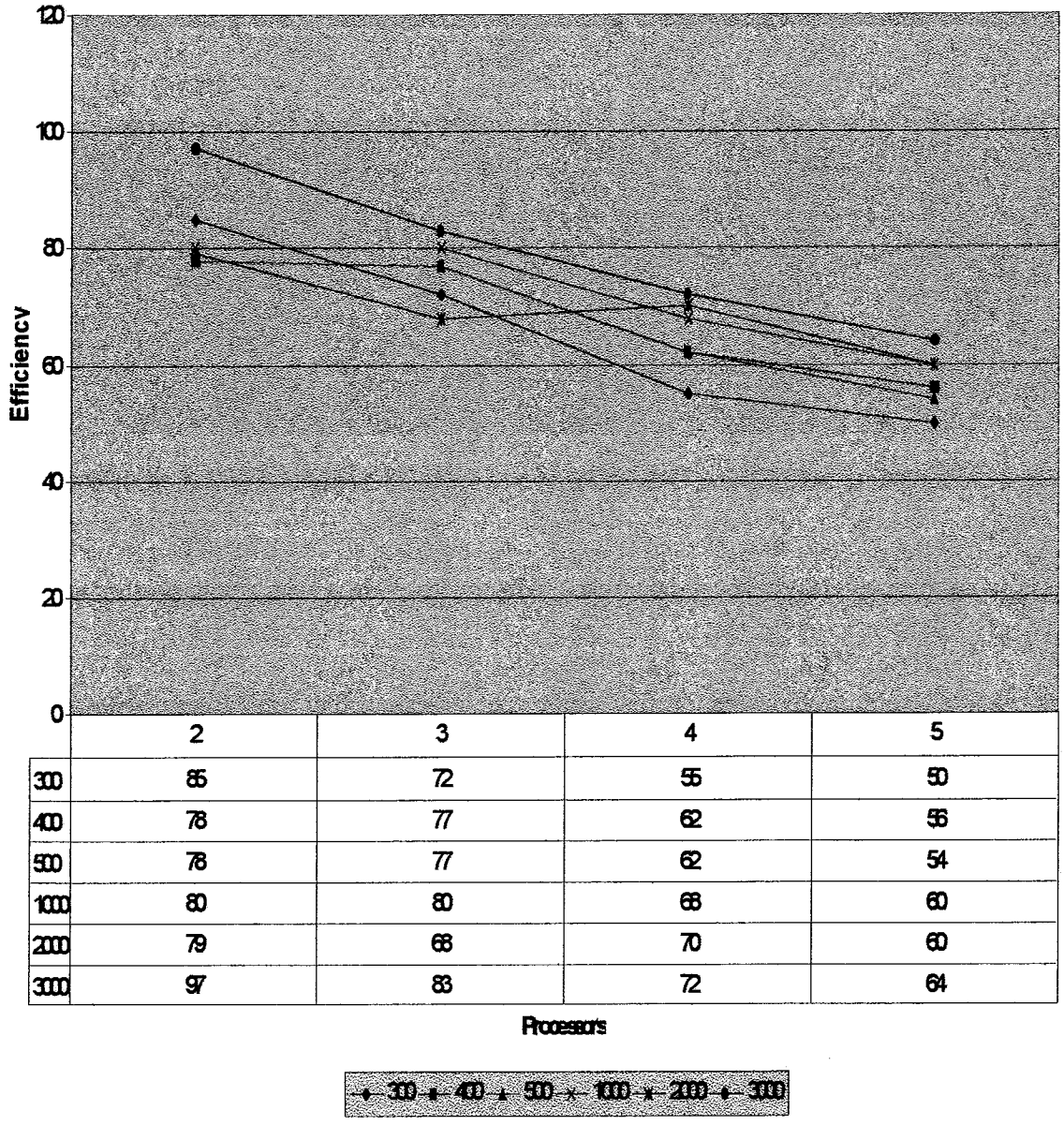
}

5. Coordinator Gather results from all tasks ;

4.3.3 Experimental Results

Our experiments were run on a cluster of computers with 5 machines (one 300 MHZ, two 333 MHZ and one 400 MHZ, four of them with 64 Mbytes of RAM and one with 128 Mbytes). In measuring algorithm time we exclude the times required for loading and broadcasting the initial graph, and for initializing the data structures. All our times were CPU times measured in seconds with MPI_Wtime( ) command.

In the following, we report the results of our experiments with Moore's (APSP) algorithm :



	2	3	4	5
300	85	72	55	50
400	78	77	62	55
500	78	77	62	54
1000	80	80	68	60
2000	79	68	70	60
3000	97	83	72	64

Figure 4.4 Efficiency Of P_APSP_MOORE Algorithm

4.3.4 Observations And Assessment

The efficiency of our code is shown in Figure 4.4. We observe that the efficiency is influenced by two parameters : number of processors and graph size. Efficiency is low in case of small graph size. Efficiency also decrease if number of processors increase. In the other hand, efficiency is high for big graphs and it increases when number of processors increase. This means that efficiency is inversely proportional to number of processors and proportional to graph size.

The explanation of these observations is that a processor is more active when the amount of data allocated to it for processing is bigger. So when the graph size is big or the number of processors is small, the data share of each processor will be relatively high which leads to more efficiency and which explains the above observations.

Dijkstra's Shortest Path Algorithm

5.1 Sequential Algorithm

Dijkstra's algorithm finds shortest paths from the source node s to all other nodes in a graph with nonnegative arc lengths. The algorithm uses two sets of nodes, S and C. at every moment the set S contains those nodes that have already been chosen; as we shall see, the minimal distance from the source is already known for every node in S. the set C contains all the other nodes, whose minimal distance from the source is not yet known, and which are candidates to be chosen at some later stage. Hence we have the invariant property $N = S \cup C$. At the outset, S contains only the source itself; when the algorithm stops, S contains all the nodes of the graph and our problem is solved. At each step we choose the node in C whose distance to the source is least, and add it to S.

We shall say that a path from the source to some other node is special if all the intermediate nodes along the path belong to S. At each step of the algorithm, an array D holds the length of the shortest special path to each node of the graph. At the moment when we add a new node v to S, the shortest special path to v is also the shortest of all the paths to v. When the algorithm stops, all the nodes of the graph are

in S, and so all the paths from the source to some other node are special. Consequently the values in D give the solution to the shortest path problem.

For simplicity, we assume that the nodes of G are numbered from 1 to n, so N = {1,2,...,n}. We can suppose without loss of generality that node 1 is the source. Suppose also that a matrix L gives the length of each directed edge: L[i,j]>= 0 if the edge (i,j)∈ A, and L[i,j] = ∞ otherwise. Here is the algorithm.

```
Function Dijkstra(L[1..n,1..n]):array[2..n]
    Array D[2..n]
    {initialization}
    C ← {2,3,...,n}  {S = N \ C exists only implicitly}
    For i ← 2 to n do D[i] ← L[1,i]
    {greedy loop}
    repeat n-2 times
        v ← some element of C minimizing D[v]
        C ← C \ {v}  {and implicitly S ← S ∪ {v} }
        For each w ∈ C do
            D[w] ← min(D[w], D[v] + L[v,w])
    Return D
```

The algorithm proceeds as follows on the graph in Figure 5.1.

Step	v	C	D
Initialization	--	{2,3,4,5}	[50,30,100,10]
1	5	{2,3,4}	[50,30,20,10]
2	4	{2,3}	[40,30,20,10]
3	3	{2}	[35,30,20,10]

Clearly D would not change if we did one more iteration to remove the last element of C. this is why the main loop is repeated only n-2 times.

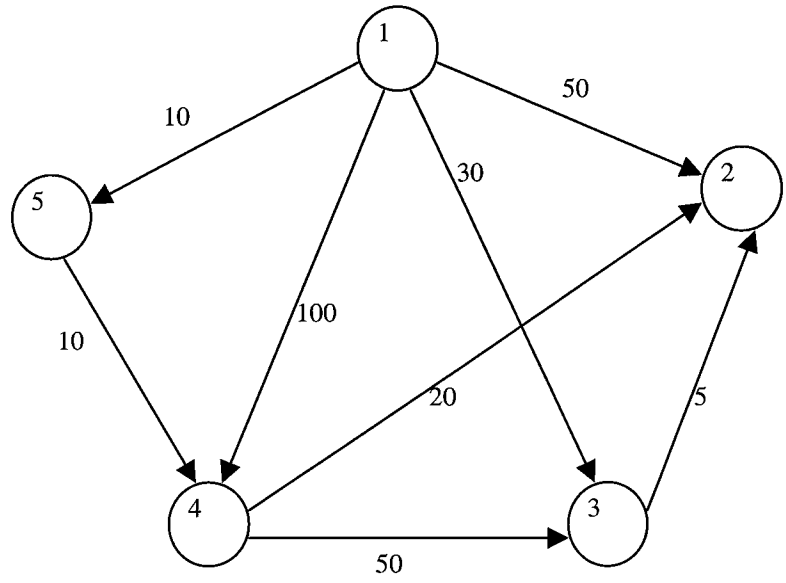


Figure 5.1 A directed graph

5.2 Analysis Of The Algorithm

Suppose Dijkstra's algorithm is applied to a graph with n nodes and a edges. Using the representation suggested up to now, the instance is given in the form of a matrix L[1 . .n, 1 . .n]. Initialization takes a time in O(n). In a straightforward implementation, choosing v in the repeat loop requires all the elements of C to be examined, so we look at n-1, n-2, ...,2 values of D on successive iteration, giving a total time in $\theta(n^2)$. The inner for loop does n – 2, n – 3, ...,1 iterations, for a total in $\theta(n^2)$. The time requires by this version of algorithm is therefore in $\theta(n^2)$.

If $a << n^2$, we might hope to avoid looking at the many entries containing ∞ in the matrix L. With this in mind, it could be preferable to represent the graph by an array of n lists, giving for each node its direct distance to adjacent nodes. This allows us to save time in the inner for loop, since we only have to consider those nodes w adjacent to v ; but it still takes a time in $\Omega(n^2)$ to determine in succession the n – 2 values taken by v.

5.3 Parallel Dijkstra's Algorithm For (APSP) Problem

In Section 5.1 we presented Dijkstra's algorithm for finding the shortest paths from a vertex v to all the other vertices in a graph. This algorithm can also be used to solve the all-pairs shortest paths problem by executing the single-source algorithm on each processor, for each vertex v. We refer to this algorithm as Dijkstra's all-pairs shortest

paths algorithm (APSP). Since the complexity of Dijkstra's single-source algorithm is $\theta(n^2)$, the complexity of the all pairs algorithm is $\theta(n^3)$.

5.3.1 Parallel Formulation

Dijkstra's all-pairs shortest paths problem can be parallelized in two distinct ways. One approach partitions the vertices among different processors and has each processor compute the single-source shortest paths for all vertices assigned to it. We refer to this approach as the **source-partitioned formulation.** Another approach assigns each vertex to a set of processors and uses the parallel formulation of the single-source algorithm (Section 5.1) to solve the problem on each set of processors. We refer to this approach as the **source-parallel-formulation.** For our implementation we are interested in the source-partitioned formulation; the following section discuss and analyze this approach.

5.3.2 Source-Partitioned Formulation

The source-partitioned parallel formulation of Dijkstra's algorithm uses n processors. Each processor P_i finds the shortest paths from vertex v_i to all vertices by executing Dijkstra's sequential single-source shortest paths algorithm. It requires no interprocessor communication. Thus, the parallel run time of this formulation is given by

$$T_p = \theta(n^2)$$

Since the sequential run time is $W = \theta(n^3)$, the speed up and efficiency are as follows: $S = \frac{\theta(n^3)}{\theta(n^2)}$

$$E = \theta(1)$$

It might seem that, due to the absence of communication, this is an excellent parallel formulation. However, that is not entirely true. The algorithm can use at most n processors. If the number of processors available for solving the problem is small (that is, $n = \theta(p)$) , then this algorithm has good performance. However, if the number of processors is greater than n, other algorithm will eventually outperform this algorithm because of its poor scalability. Here is the parallel algorithm.

P_APSP_DIJKSTRA

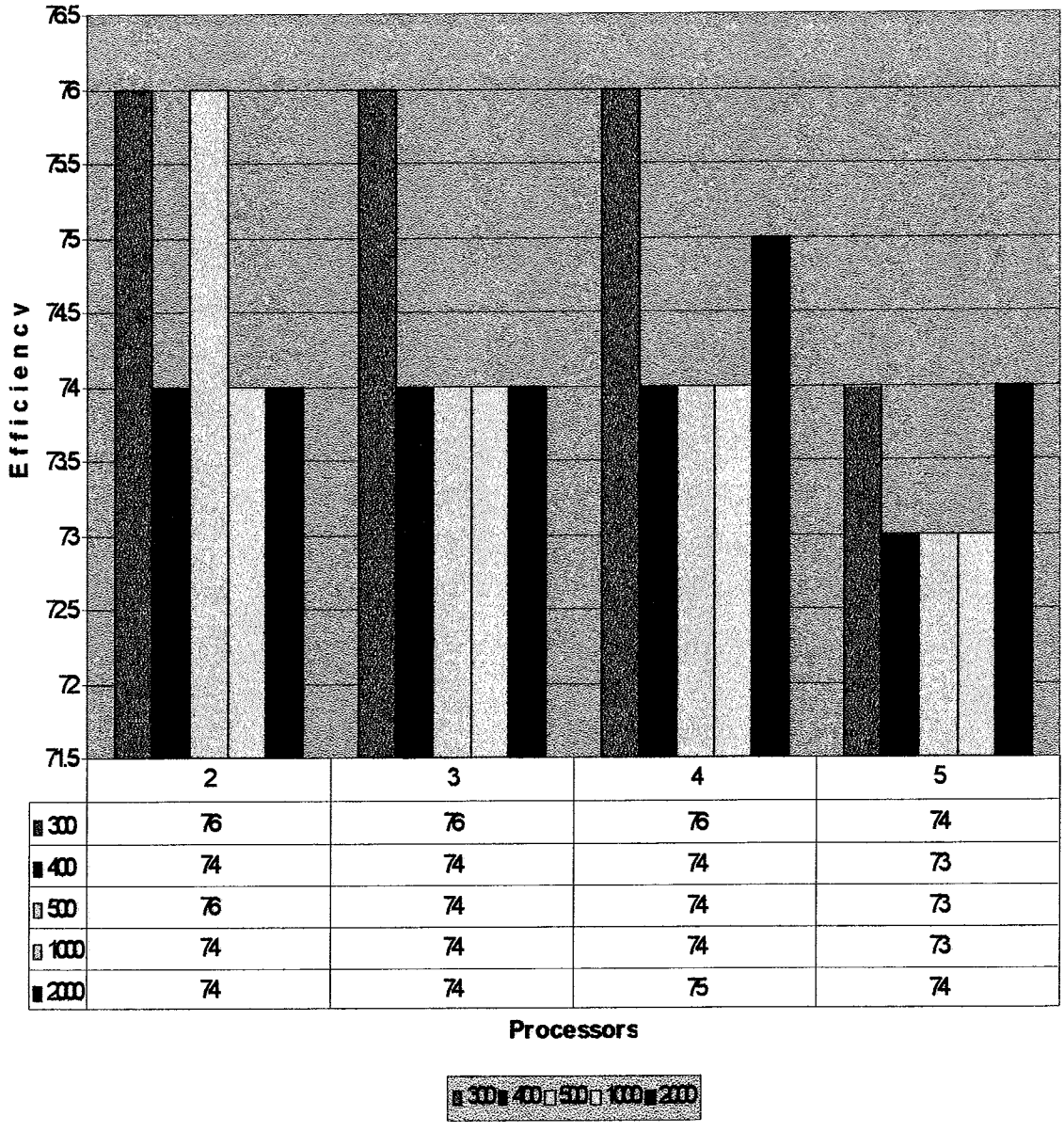
1. Broadcast graph G;
2. p = number of tasks;
3. m = my instance id;
4. for(i=m; i<graph size; i+=p) {

        sequential moore;

   }
5. Coordinator Gather results from all tasks ;

5.4 Experimental Results

Our experiments on P_APSP_DIJKSTRA were done in the same environment as the previous experiments and with the same type of input graphs. Here are the results of the experiments



	2	3	4	5
300	76	76	76	74
400	74	74	74	73
500	76	74	74	73
1000	74	74	74	73
2000	74	74	75	74

Figure 5.2 Efficiency Of P_APSP_DIJKSTRA Algorithm

5.5 Observations And Assessment

We can see in Figure 5.2 that the efficiency is varying very slightly with graph size and with number of processors. With efficiency between 73 and 76 percent we get acceptable speedup figures. But we saw in section 5.3.2 that theoretically, efficiency should be 100 percent. The difference between experimental results and theoretical performance is due, in our case, to the heterogeneity of the multicomputer system used. The existence of computers slower than others leads to an execution time equal to that of the slower computer, which causes the faster computers to be inactive for a fraction of total execution time, and then the average efficiency of these computers will drop from 100% (theoretical) to around 75% (experimental results).

Minimum Spanning Tree: Prim's Algorithm

6.1 Sequential Algorithm

A ***spanning tree*** of an undirected graph G is a subgraph of G that is a tree containing all the vertices of G. In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph. A ***minimum spanning tree*** (MST) for a weighted undirected graph is a spanning tree with minimum weight. Many problems require finding an MST of an undirected graph. For example, the minimum length of cable necessary to connect a set of computers in a network can be determined by finding the MST of the undirected graph containing all the possible connections. Figure 6.1 shows an MST of an undirected graph.

If G is not connected , it cannot have a spanning tree. Instead, it has a ***spanning forest.*** For simplicity in describing the MST algorithm, we assume that G is connected. If G is not connected, we can find its connected components and apply the algorithm on them. Alternatively, we can modify the MST algorithm to output a minimum spanning forest.

Prim's algorithm for finding an MST is a greedy algorithm. The algorithm begins by selecting an arbitrary starting vertex. It then grows the minimum spanning tree by choosing a new vertex and edge that are guaranteed to be in the minimum spanning tree. The algorithm continues until all the vertices have been selected. Here

is the algorithm.

```
1.   Algorithm PRIM_MST ( V, E, w, r )
2.   begin
3.       V_T = {r};
4.       d[r] = {0;
5.       for all v ∈ (V – V_T) do
6.           if edge (r, v) exists set d[v] = w(r, v)
7.           else set d[v] = ∞ ;
8.       while V_T != V do
9.       begin
10.          find a vertex u such that d[u] = min{d[v] | v ∈ (V -
         V_T)} ;
11.          V_T = V_T ∪ {u} ;
12.          for all v ∈ (V - V_T) do
13.              d[v] = min{d[v], w(u,v)};
14.      end while
15.  end PRIM_MST
```

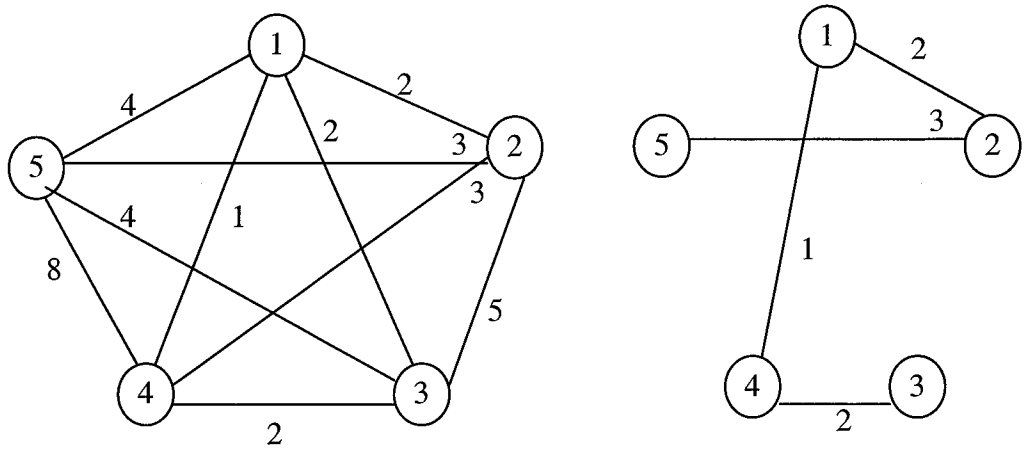


Figure 6.1 An undirected graph and its minimum spanning

Let G = *(V, E, w)* be the weighted undirected graph for which the minimum spanning tree is to be found, and let $A = (a_{i,j})$ be its weighted adjacency matrix. Prim's algorithm is shown in page 45 (PRIM_MST). The algorithm uses the set V_T to hold the vertices of the minimum spanning tree during its construction. It also uses an array *d*[1..n] in which, for vertex $v \in (V - V_T)$, *d*[v] holds the weight of the edge with the least weight from any vertex in V_T to vertex v. Initially, V_T contains an arbitrary vertex *r* that becomes the root of the MST. Furthermore, *d*[r]=0, and for all v such that $v \in (V - V)$, *d*[v] = *w(r, v)* if such an edge exists; otherwise d[v] = ∞. During each iteration of the algorithm, a new vertex u is added to V_T such that $d[u] = \min\{d[v] \mid v \in (V - V_T)\}$. After this vertex is added, all values of d[v] such that $v \in (V - V_T)$ are updated because there may now be an edge with a smaller weight between vertex v and the newly added vertex u. The algorithm terminates when $V_T = V$. Upon termination of Prim's algorithm, the cost of the minimum spanning tree is $\Sigma_{v \in V} d[v]$.

In algorithm PRIM_MST, the body of the while loop (lines 10-13) is executed n-1 times. Both the computation of $\min\{d[v] \mid v \in (V - V_T)\}$ (line 10), and the for loop (lines 12 and 13) execute in O(n) steps. Thus, the overall complexity of Prim's algorithm is $\theta(n^2)$.

6.2 Parallel Formulation

Prim's algorithm is iterative. Each iteration adds a new vertex to the minimum spanning tree. Since the value of d[v] for a vertex v may change every time a new vertex u is added in V_T, it is impossible to select more than one vertex to include in

the minimum spanning tree. Thus, different iterations of the while loop cannot be performed in parallel. However, each iteration can be performed in parallel as follows.

Let p be the number of processors, and let n be the number of vertices in the graph. The set V is partitioned into p subsets using the block-striped partitioning. Each subset has n/p consecutive vertices, and the work associated with each subset is assigned to a different processor. Let V_i be the subset of vertices assigned to processor P_i for i = 0,1,...,p-1. Each processor P_i stores the part of the array d that corresponds to V_i (that is, processor P_i stores d[v] such that $v \in V_i$). Each processor P_i computes $d_i[u] = \min\{d_i[v] \mid v \in (V - V_T) \cap V_i\}$ during each iteration of the while loop. The global minimum is then obtained over all $d_i[u]$ by using the single-node accumulation operation and is stored in processor P_0. Processor P_0 now holds the new vertex u, which will be inserted into V_T. Processor P_0 broadcasts u to all processors by using one-to-all broadcast. The processor P_i responsible for vertex u marks u as belonging to set V_T. Finally each processor updates the values of d[v] for its local vertices.

When a new vertex u is inserted into V_T, the values of d[v] for $v \in (V - V_T)$ must be updated. The processor responsible for v must know the weight of the edge (u, v). Hence, each processor P_i needs to store the columns of the weighted adjacency matrix corresponding to set V_i of vertices assigned to it.

The computation performed by a processor to minimize and update the values of d[v] during each iteration is $\theta(n/p)$. the communication performed in each iteration is due to the single-node accumulation and the one-to-all broadcast.

6.3 Sparse Graphs Consideration

The parallel algorithm in the previous section is based on an algorithm for dense-graph problems. However, we have yet to address the parallel algorithm for sparse graphs. Recall that a graph G = (V, E) is sparse if |E| is much smaller than $|V|^2$.

Any dense-graph algorithm works correctly on sparse graphs also. However, if the sparseness of the graph is taken into account, it is usually possible to obtain significantly better performance. An important step in developing sparse-graph algorithms is to use an adjacency list instead of an adjacency matrix. This change in representation is crucial, since the complexity of adjacency-matrix-based algorithms is usually $\Omega(n^2)$, independent of the number of edges. Conversely, the complexity of adjacency-list-based algorithms is usually $\Omega(n + |E|)$, which depends on the sparseness of the graph.

6.4 Parallel Algorithm Description

In the following we'll present the algorithm for computing minimum spanning tree in parallel.

1. **Algorithm P_PRIM_MST**
2. p = process identification
3. if (p = = master)
4. broadcast graph to all processes
5. get the root vertex **r** (randomly or as a parameter)

6. broadcast root vertex to all processes

7. end if

8. $V_T = \{r\}$; d[r] = 0;

9. for all $v \in (V - V_T)$ do in parallel

10. if edge (r,v) exists set d[v] = w(r,v);

11. else set $d[v] = \infty$;

12. end for

13. while V_T != V do in parallel

14. begin

15. find a vertex u such that $d_i[u] = \min\{d_i[v] \mid v \in (V - V_T)\}$;

16. send all the min results to the master process

17. if p = = master get the min of all the min (min_u), and broadcast min_u

18. $V_T = V_T \cup \{ min_u \}$;

19. for all $v \in (V - V_T)$ do in parallel

20. $d_i[v] = \min\{d_i[v], w(min_u, v)\}$;

21. end while

22. send all the partial MST to the master process and get the complete MST.

6.5 Experimental Results

Our experiments on P_PRIM_MST were done in the same environment as the previous experiments and with the same type of input graphs. Here are the results of the experiments.

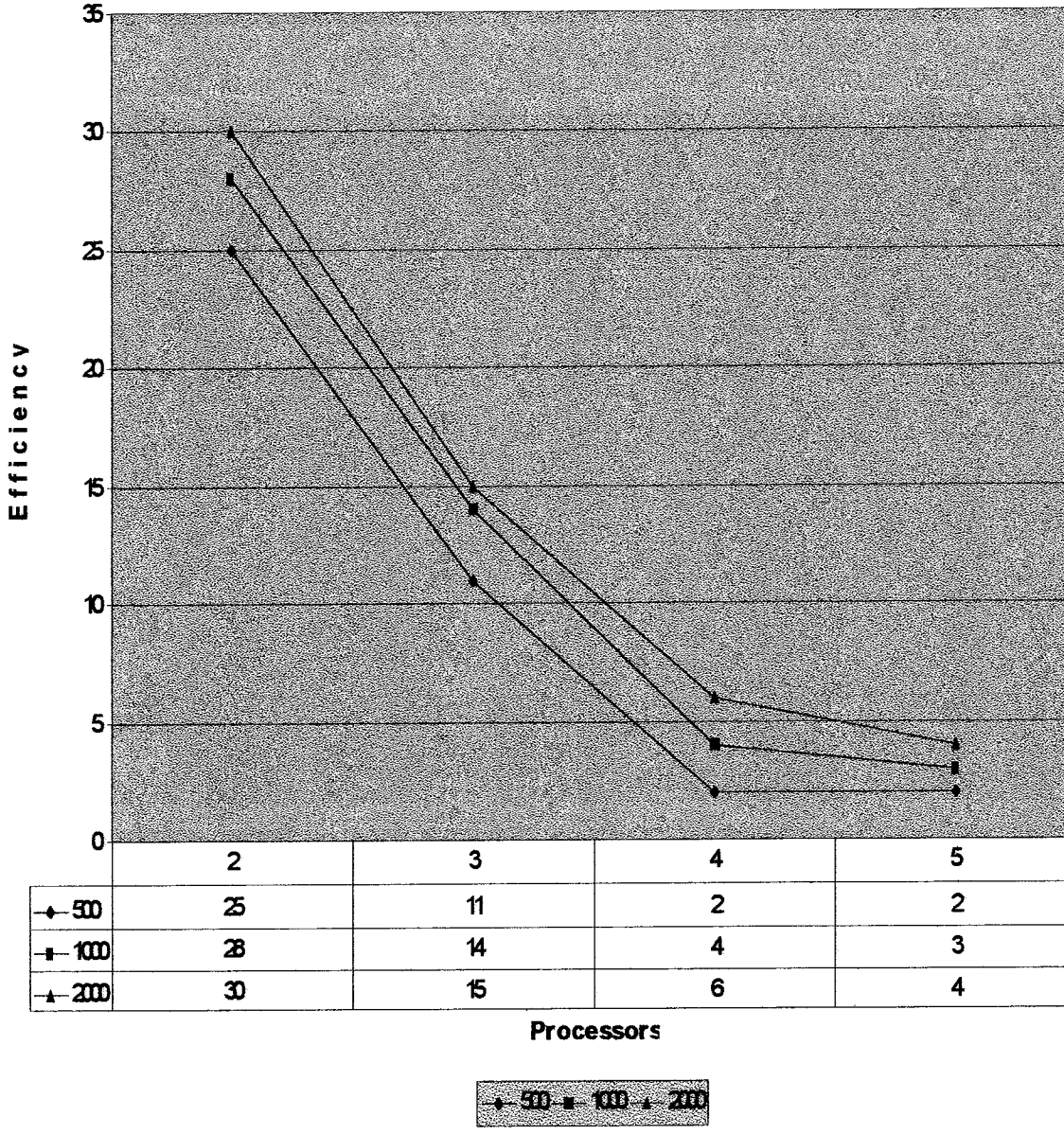


Figure 6.2 Efficiency Of P_PRIM_MST Algorithm

6.6 Observations And Assessment

As shown in Figure 6.2, performs badly for different set of parameters. But its worst performance arise with the maximum number of processors used for testing, and its best performance comes with minimum number of processors used (i.e. 2) and the biggest input graph used for testing.

Discussion

In this document, we have presented parallel algorithms to solve shortest paths and minimum spanning tree problems. We developed four different algorithms , namely, P_SSSP_MOORE, P_APSP_MOORE, P_APSP_DIJKSTRA, P_PRIM_MST. Each differs in the problem they solve or in the way they solve the problem. Two of the algorithms, that solve the all pairs shortest path problem, are embarrassingly parallel, and the two remainder are parallel algorithms with interprocess communication.

We implemented the algorithms as SIMD programs and studied their performance on random-weighted graphs. The input graphs were connected and the largest consisted of nearly 3000 nodes.

After experimentation, we conclude that the parallel shortest path algorithm P_SSSP_MOORE shows good performance for large random graphs. We expect the applicability of our approach even on non-random graphs but always for sparse graphs. In terms of scalability, this algorithm can use at most n processors.

According to the experimental results shown in sections 4.2.3 and 5.4, we can see obviously that algorithm P_APSP_MOORE outperforms algorithm P_APSP_Dijkstra. This observation is due mainly to the following reason.

Dijkstra's algorithm is very powerful for dense graphs, but for sparse graphs better algorithms exist. As we know in Moore's algorithm each vertex is added to the queue, for later consideration , x number of times. Let a be the average number of all

instances of x and n the number of vertices. The complexity of Moore's algorithm is O(a.n). Even for sparse graphs, Dijkstra's complexity still in $\Omega(n^2)$. For a << n, Moore's algorithm outperforms dijkstra's algorithm.

The fourth parallel algorithm presented in this document, Prim, performs badly. This is explained by two major factors : communication overhead and cost of synchronization especially increased by the usage of different computers with different speeds.

1. Communication overhead : referring to algorithm P_PRIM_MST in section 6.4, the while loop at line 13 contains n collective messages, with n the number of nodes. With collective messages, like Broadcast and Gather, all concerned processes are involved with sending or receiving the messages. From here we can see the high communication cost produced by the implemntation of this algorithm.
2. Cost of synchronization : we talked about collective messages, so we can induce that the synchronization between processes is a must, which means that some inactive processes should wait for active processes to finish their work, then to proceed with the message-passing process. the synchronization cost is obviously aggravated by the communication overhead and by the usage of heterogeneous set of computers.

The experimentation done on the implementation of parallel Prim's algorithm, emphasizes what is said in many parallel computing papers like [Adler 98].

"For many graph problems such as list ranking, connected components and minimum spanning tree, it seems difficult to find communication-efficient algorithms. for many of these problems, very fast (linear time) sequential algorithms exist, but efficient parallel algorithms are unknown."

Conclusions and Future Work

In this document, we have described our project on the design and implementation of parallel algorithms for some important graph problems. Our main focus was on using networked workstations and parallel computers. In shared-memory multiprocessing we are interested in techniques that allow concurrent access to shared memory, while in multicomputer programming the main issue is accessing remote memory and minimizing this remote access to lessen the communication cost on the efficiency of the multicomputer system.

Future work can tackle the design and implementation of communication-efficient parallel minimum spanning tree algorithms.

Another approach is to find scalable shortest path algorithm, in other terms they should be efficient and applicable for $p_{processors} > n_{nodes}$.

Interesting investigation can be done for the all pairs shortest path algorithm. One approach could be to run the single-source algorithm on disjoint subsets of processors instead of running one instance of it on each processor. Specifically, p processors are divided into n partitions, each with p/n processors. Each of the n single-source shortest paths problems is solved by one of the n partitions. In other words, start with parallelizing the all-pairs shortest paths problem by assigning each vertex to a separate set of processors, and then parallelize the single-source algorithm

by using the set of p/n processors to solve it. However, this formulation is of interest only if $p > n$.

Also, of further interest is to minimize the communication cost of the parallel single-source shortest path Moore's algorithm. We should investigate two different approaches of partitioning the graph between processes, one approach is breadth-first partition and the other is to assign all the edges incident to a node to one process. Further work in this respect **may lead** to the development of better algorithms to solve the single-source shortest path problem.

REFERENCES

[Adler 98] M. Adler, W. Dittrich, B. Juurlink, M. Kutylowski and I. Rieping. "Communication-Optimal parallel minimum spanning tree algorithms," 10th Annual ACM Symposium on Parallel Algorithms and Architectures, June 28 – July 2, 1998.

[Akl 97] Selim G. Akl. "Parallel computation; models and methods," Prentice Hall, 1997.

[Ahuja 93] R. K. Ahuja, T. L. Magnanti and J. B. Orlin. "Network Flows," Prentice Hall, 1993.

[Awerlbuch 87] B. Awerbuch and Y. Shiloach. "New connectivity and MSF algorithms for Shuffle-Exchange network and PRAM," IEEE Transactions on Computers, pp. 1258-1263, 1987.

[Bertsekas 89] D. P. Bertsekas and J. Tsitsiklis. "Parallel and distributed computation; numerical methods," Prentice Hall, 1989.

[Brassard 96] G. Brassard and P. Bratley. "Fundamentals of algorithmics," Prentice Hall, 1996.

[Chong 93] K. W. Chong, T. W. Lam, "Finding connected components in O(log n log log n) Time on EREW PRAM," 4th ACM-SIAM Symposium on Discrete Algorithms, pp. 11-20, 1993.

[Cole 96] R. Cole, P. N. Klein, and R. E. Tarjan. "Finding minimum spanning forests in logarithmic time and linear work using random sampling," In Annual ACM Symposium on Parallel Algorithms and Architecture, pp. 243-250, 1996.

[Crauser 98] A. Crauser, K. Mehlhorn, U. Meyer and P.Sanders. "A parallelization of Dijkstra's shortest path algorithm," http://www.mpi-sb.mpg.de , 1998.

[Johnson 95] D. B. Johnson and P. Metaxas. "A parallel algorithm for computing minimum spanning trees," Journal of Algorithms, pp. 383-410, 1995.

[Jones 93] Mark T Jones and Paul E. Plassmann. "A parallel graph coloring heuristic," SIAM J. Sci Statist. Comput., pp 654-669, MAY 1993.

[Karger 95] D. Karger, P. Klein, and R. Tarjan. "A randomized linear-time algorithm to find minimum spanning trees," Journal of the ACM, pp. 321-328, 1995.

[Karger 93] D. R. Karger. "Random sampling for minimum spanning trees and other optimization problems," In annual ACM Symposium on Foundations of Computer Science, 1993.

[Kumar 1994] V. Kumar, A. Grama, A. Gupta and G. Karypis. "Introduction to parallel computing, design and analysis of algorithms," The Benjamin/Cummings Publishing Company, Inc. 1994.

[Laxmikant 95] Laxmikant V. Kale, Ben H. Richards and Terry D. Allen. "Efficient parallel graph coloring with prioritization," http://charm.cs.uiuc.edu , 1995.

[Lester 93] Bruce P. Lester. "The art of parallel programming," Prentice Hall, 1993.

[Lumetta 96] Steven S. Lumetta, Arvind Krishnamurthy, and David E. Culler. "Towards modeling the performance of a fast connected components algorithm on parallel machines," http://HTTP.CS.Berkeley.EDU/~stevel, 1996.

[Meyer 99] U. Meyer and P. Sanders. "Δ-Stepping : A parallel single source shortest path algorithm," http://www.mpi-sb.mpg.de/sanders , 1999.

[Quinn 94] Michael J. Quinn. "Parallel computing; theory and practice," McGraw-Hill, 1994.

[Romeijn 99] H. Edwin Romeijn, Robert L. Smith. "Parallel algorithms for solving aggregated shortest-path problems," Computers & Operations Research 26 pp 941-953, 1999.

[Roosta 99] Seyed H. Roosta. "Parallel processing and parallel algorithms; theory and computation," Springer, 1999.

[Schiloach 82] Y. Schiloach, U. Vishkin, "An O(log n) parallel connectivity algorithm," Journal of Algorithms, No. 3, pp. 57-67, 1982.

[Wilkinson 99] B. Wilkinson and M. Allen. "Parallel programming; techniques and applications using networked workstations and parallel computers," Prentice Hall, 1999.

Implementation Details

In the following, we present four parallel programs corresponding to the implementation of the four parallel graph algorithms described in previous chapters.

Parallel Single-Source Shortest Path Moore's Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#define ACK     -1
#define INFINITY -1
#define FINISH  -2
#define NCOL     6

int ackcount = 0, v_queue=-1;
bool active = false, stop = false;

void bzero(void *ptr, int count)
{
        memset(ptr,-1,count);
}

int* create_vector(int n)
{
        int* dist;
        dist=(int*)malloc(n*sizeof(int));
        bzero((void*)dist,n*sizeof(int));
        return(dist);
}

void init_var(int* dist, int source)
{
        dist[source] = 0;
}

void basic_mpi(int* id, int* task, int* slave)
{
        MPI_Comm_rank(MPI_COMM_WORLD, id);
```

```
        MPI_Comm_size(MPI_COMM_WORLD, task);
   *slave = *task - 1;
}

int** create_table(int n)
{
        int** tab;
        int i;
        tab=(int**)malloc(n*sizeof(int*));
        for(i=0;i<n;i++)
        {
                tab[i]=(int*)malloc(NCOL*sizeof(int));
                if (tab[i])
                        bzero((void*)tab[i],NCOL*sizeof(int));
        }
        return(tab);
}

void sr_graph(int** edge, int** weight, int n) /* send-receive graph */
{
        int i=NCOL*n, os=i*2;
        int *tmp,*ebuf,*wbuf;
        ebuf=create_vector(i);
        wbuf=create_vector(i);
        tmp=ebuf;
        for(int h=0;h<n;h++)
                for(int g=0;g<NCOL;g++)
                        *ebuf++=edge[h][g];
        ebuf=tmp;
        tmp=wbuf;
        for(h=0;h<n;h++)
                for(int g=0;g<NCOL;g++)
                        *wbuf++=weight[h][g];
        wbuf=tmp;

        MPI_Bcast(ebuf,i,MPI_INT,0,MPI_COMM_WORLD);
        MPI_Bcast(wbuf,i,MPI_INT,0,MPI_COMM_WORLD);
                for(h=0;h<n;h++)
                        for(int g=0;g<NCOL;g++)
                                edge[h][g]=*ebuf++;
                for(h=0;h<n;h++)
                        for(int g=0;g<NCOL;g++)
                                weight[h][g]=*wbuf++;
}

void get_basic_info(int* s, int* n,int arg1,char** arg2)
{
                *s=atoi(arg2[1]);
                *n=atoi(arg2[2]);
}

int get_p_dest(int dest, int n)
{
        int p;
        return(p=(dest%(n-1))+1);
}

int* get_gvertices(int p, int np,int nv)
```

```
{
        int i=0, q=0;
        int* arr_v;
        bool stp=false;
        q=(nv/(np-1))+1;
        arr_v=create_vector(q);
        while(i<q && !stp)
        {
                arr_v[i]=(p-1)+(np-1)*i;
                if (arr_v[i] > nv-1)
                        stp=true;
                i++;
        }
        return(arr_v);
}

bool empty(int queue)
{
        if (queue==-1)
                return(true);
        else
                return(false);
}

void getwork(int* triald, int* tdest, int* parent)
{
        int* buf, ack_var,n,h=0;
        MPI_Request request;
        MPI_Status  status;
        do
        {
                if(active && ackcount==0 && (empty(v_queue)))
                {
                        active  = false;
                        ack_var = -1;
                        MPI_Isend(&ack_var,1,MPI_INT,*parent,
                        -1,MPI_COMM_WORLD,&request);
                }
                n=2;
                buf=create_vector(n);

MPI_Recv(buf,2,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
                                MPI_COMM_WORLD,&status);
                if (status.MPI_TAG == ACK)
                {
                        ackcount--;
                }
        }
        while(status.MPI_TAG==ACK);

        if(status.MPI_TAG==FINISH)
                stop=true;
        else
        {
                if(active)
                {
                        ack_var = -1;
```

```
        MPI_Isend(&ack_var,1,MPI_INT,status.MPI_SOURCE,-
1,MPI_COMM_WORLD,&request);
                        }
                        else
                        {
                                active  = true;
                                *parent = status.MPI_SOURCE;
                        }
                        *tdest =buf[0];
                        *triald=buf[1];
                }
        } /* end of getwork */

        void send_work(int v_dest,int dist, int p_dest)
        {
                int* buf, n=2,h=0;
                MPI_Request request;

                buf=create_vector(n);
                buf[0]=v_dest;
                buf[1]=dist;

                MPI_Isend(buf,2,MPI_INT,p_dest,1,MPI_COMM_WORLD,
                &request);
                ackcount++;
        }

        /* random generation of Sparse Directed graph */

        int random(int max)
        {
                int x;
                return(x=rand()%max);
        }

        void randSDgraph(int **edg,int **weit,int nv,int max_w)
        {
                int i,j,v,rv,rw;

                for(i=0;i<nv;i++)
                {
                        j=0;

                        while(j<NCOL)
                        {

                                rv = random(nv);

                                if(j==0)
                                 while(rv==i)
                                        rv = random(nv);
                                else
                                {

                                while(rv!=i && (rv==edg[i][0] || rv==edg[i][1]
                                                || rv==edg[i][2] || rv==edg[i][3]
                                        || rv==edg[i][4] || rv==edg[i][5]))
```

```
                                        rv=random(nv);
                                }
                                if(rv==i)
                                {

                                for(v=j;v<NCOL;v++)
                                        {
                                                edg[i][v]=-1;
                                                weit[i][v]=-1;
                                        }
                                        j=NCOL;
                                }
                                else
                                {
                                        edg[i][j]=rv;
                                        rw=random(max_w);
                                        while(rw==0)
                                                rw=random(max_w);
                                        weit[i][j]=rw;
                                        j++;
                                }
                        } /* end while */
                } /* end for */
} /* end randSDgraph */


void main(int argc, char* argv[])
{
        int source, nbvertices, recv_ack=0, p_dest;
        int parent=-1, triald=0, tdest=0,myId=-1,nTasks=-1,nSlaves=-1;
        int **edgemat,**weightmat;
        int *distvect,*gvert,*portion;
        int ind=0,nport=0,i,max_w=60;
        double t1,t2,t3,t4,tres=0;
        MPI_Status status;

        MPI_Init(&argc,&argv);
        t1=MPI_Wtime();
        basic_mpi(&myId,&nTasks,&nSlaves);
        get_basic_info(&source,&nbvertices,argc,argv);
        distvect=create_vector(nbvertices);
        weightmat=create_table(nbvertices);
        edgemat=create_table(nbvertices);

        if(myId==0)
        {
                if(argc==4)
                {
                randSDgraph(edgemat,weightmat,nbvertices,max_w);

                        v_queue=source;
                        init_var(distvect,source);
                        sr_graph(edgemat,weightmat,nbvertices);
                        p_dest=get_p_dest(source,nTasks);
                        send_work(source,distvect[source],p_dest);
                        /* wait for acknowledment from p_dest */
MPI_Recv(&recv_ack,1,MPI_INT,p_dest,MPI_ANY_TAG,
MPI_COMM_WORLD,&status);
```

```
        /* send finish tag to all processes */
        for(i=1;i<nTasks;i++)
MPI_Send(&i,1,MPI_INT,i,-2 ,MPI_COMM_WORLD);
        /* receive all portions from slaves */
                nport=(nbvertices/(nTasks-1))+1;
                portion=create_vector(nport);
                for(i=1;i<nTasks;i++)
                {
MPI_Recv(portion,nport,MPI_INT,MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD,&status);
gvert=get_gvertices(status.MPI_SOURCE,nTasks,nbvertices);
                        while(ind<nport)
                        {
                                int a;
                                a=gvert[ind];
                                if (a!=-1)
                                distvect[a]=portion[ind];
                                ind++;
                        }
                        ind = 0;
                }
                t2 = MPI_Wtime();
                /* end receive all portions from slaves */
                for(int i=0;i<nbvertices;i++)
                {
                        if ((i%3)==0)
                printf("distvect[%d]=%d \n",i,distvect[i]);
                        else
                        printf("distvect[%d]=%d  ",i,distvect[i]);
                }
                printf("\n time = %.3lf \n",t2-t1);
        }
        else /* if argc==4 */
        {
                printf("exit");
                MPI_Finalize();
                exit(1);
        }
}
else /*if slave */
{
        sr_graph(edgemat,weightmat,nbvertices);
        while(empty(v_queue) && !stop)
        {
                getwork(&triald,&tdest,&parent);
                if(!stop)
                 if(triald<distvect[tdest] || distvect[tdest]==-1)
                 {
                        distvect[tdest]=triald;
                        v_queue=tdest;
                 }
        }
        while(!empty(v_queue) && !stop)
        {
                int j=0;
        while(j<NCOL && edgemat[v_queue][j]!=INFINITY)
                {
                int d=distvect[v_queue]+weightmat[v_queue][j];
```

```
                                        int dest=edgemat[v_queue][j];
                                    if(d<distvect[dest] || distvect[dest]==-1)
                                    {
                                        p_dest=get_p_dest(dest,nTasks);
                                            if (p_dest!=myId)
                                            distvect[dest]=d;
                                            send_work(dest,d,p_dest);
                                        }
                                    j++;
                                }
                                v_queue=-1;
                                while(empty(v_queue) && !stop)
                                {
                                 getwork(&triald,&tdest,&parent);
                                 if(!stop)
                                  if(triald<distvect[tdest] || distvect[tdest]==-1)
                                  {
                                        distvect[tdest]=triald;
                                        v_queue=tdest;
                                  }
                                }
                        }
                        /* send the portion of distvect to the master*/
                        gvert=get_gvertices(myId,nTasks,nbvertices);
                        nport=(nbvertices/(nTasks-1))+1;
                        portion=create_vector(nport);
                        while(ind<nport)
                        {
                                int a;
                                a=gvert[ind];
                                if(a!=-1)
                                portion[ind]=distvect[a];
                                ind++;
                        }
                MPI_Send(portion,nport,MPI_INT,0,MPI_ANY_TAG,
                MPI_COMM_WORLD);
                /* end of send the portion of distvect to the master */
                } /* end slave */
        MPI_Finalize();
} /*end of main*/
```

Parallel All Pairs Shortest Path Moore's Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <mpi.h>
#include <queue>
#include <list>

using namespace std ;
// Using queue with list
```

```
typedef list<int > INTLIST;
typedef queue<int>  INTQUEUE;

int** create_mat(int nl,int nc)
{
        int** tab;
        int i;
        tab=(int**)malloc(nl*sizeof(int*));
        for(i=0;i<nl;i++)
        {
                tab[i]=(int*)malloc(nc*sizeof(int));
                if (tab[i])
                        bzero((void*)tab[i],nc*sizeof(int));
        }
        return(tab);
}

int get_index(int* distind,int vertex,int nport)
{
        int i=0;
        while(i<nport)
        {
                if (distind[i] == vertex)
                        return(i);
                i++;
        }
}
int port_size(int np, int nv)
{
        div_t div_result;
        int x=0,nport;

        div_result=div(nv,np);
        if (div_result.rem!=0)
                x=1;
        return(nport = div_result.quot + x);
}

int get_port(int p,int np,int nv)
{
        div_t div_result;
        int nport;
        int x=0;

        div_result=div(nv,np);
        if (div_result.rem!=0)
                x=1;
        if (p < div_result.rem)
                return(nport = div_result.quot + x);
        else
                return(nport = div_result.quot);
}

void moore(int i,int *dist,int **edge,int **weight)
{
        int j;
        int vert_dist,vert;
        INTQUEUE q;
```

```
    // Insert items in the queue(uses list)
    q.push(i);
    dist[i]=0;

        while (!q.empty())
    {
                vert = q.front();
                q.pop();
                vert_dist=dist[vert];
                j=0;
                while(j<NCOL && edge[vert][j]!=-1)
                {
                        int x=edge[vert][j];
                if (((dist[x] > vert_dist + weight[vert][j]) || dist[x]==-1)
                                && x!=i)
                        {
                                dist[x]=vert_dist+weight[vert][j];
                                q.push(x);
                        }
                        j++;
                }
        }
}
void main(int argc, char* argv[])
{
        int r_vertex,source=0,a_ind;
        int nbvertices=0,myId=-1,nTasks=-1,nSlaves=-1;
        int end=0, start=0;
        int **edgemat,**weightmat;
        int *distvect,*distind, **distmat;
        int *recvbuf,*sendbuf,*rbuf,*mst;
        int ind=0,dpointer=0,nport,count_v,sport,ce,recv_e;
        int minv=0,mind=0, max_w=60;
        long totamount=0;
        double t1,t2;

        MPI_Init(&argc,&argv);

        basic_mpi(&myId,&nTasks,&nSlaves);
        get_basic_info(&source,&nbvertices,argc,argv);
        nport    = get_port(myId,nTasks,nbvertices);
        distvect = create_vector(nbvertices);
        sport    = port_size(nTasks,nbvertices);
        distind  = create_vector(nport);
        ce       =(nbvertices+1)*sport;
        sendbuf  = create_vector(ce);
        recv_e   = ce * nTasks;
        rbuf     = create_vector(recv_e);
        distmat  = create_mat(sport,nbvertices);
        weightmat=create_table(nbvertices);
        edgemat=create_table(nbvertices);

        if(myId==0)
        {
                randSDgraph(edgemat,weightmat,nbvertices,max_w);
                r_vertex=source;
        }
```

```
sr_graph(edgemat,weightmat,nbvertices);
t1=MPI_Wtime();

a_ind=0;
for(int f=myId;f<nbvertices;f+=nTasks)
{
        moore(f,distmat[a_ind],edgemat,weightmat);
        a_ind++;
}

f=myId;
int pas=0;
int u=0;

while(u<ce)
{
        int r;
        sendbuf[u]=f;
        u++;
        for(r=0;r<nbvertices;r++)
        {
                sendbuf[u]=distmat[pas][r];
                u++;
        }
        pas++;
        f+=nTasks;
}

MPI_Gather(sendbuf,ce,MPI_INT,rbuf,ce,MPI_INT,0,
MPI_COMM_WORLD);
t2=MPI_Wtime();
if (myId == 0)
{
        int **totmat;
        int b=0,t=0;
        ind = rbuf[b];
        b++;
        totmat = create_mat(nbvertices,nbvertices);
        if (ind<nbvertices)
        {
                while(b<recv_e)
                {
                        if (ind<nbvertices)
                        {
                                for(t=0;t<nbvertices;t++)
                                {
                                        totmat[ind][t]=rbuf[b++];
                                }
                        }
                        else
                        {
                                b=b+nbvertices;
                        }
                        ind=rbuf[b++];
                }
        }
        for(b=0;b<nbvertices;b++)
        for(t=0;t<nbvertices;t++)
```

```
                {
                        if ((t % 3)==0)
                        printf("mat[%d][%d]=%d  \n",b,t,totmat[b][t]);
                        else
                        printf("mat[%d][%d]=%d    ",b,t,totmat[b][t]);
                        }
        } /* end if master */

        printf("\n time1 = %.3lf \n",t2-t1);

        MPI_Finalize();

} /*end of main*/
```

Parallel All Pairs Shortest Path Dijkstra's Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <mpi.h>


int get_dist(int v,int* edge,int* weight)
{
        int i=0,rdist=-1;
        bool found = false;

        while(i<NCOL && !found)
        {
                if (edge[i] == v)
                {
                        found = true;
                        rdist=weight[i];
                }
                else
                i++;
        }
        return(rdist);
}


int min_out(int nVert, int *c, int *dist, int i)
{  int j,h,g;
        int v = 0;

        for (j=0; j<nVert-1; j++)
        {
                if ((dist[j+1] < dist[v]) && (c[j+1] != -10)
                        && (dist[j+1]!=-1) &&
                        (dist[v]!=-1))
                        v = j+1;
                else
                        if ((c[v] == -10)  || (dist[v]==-1))
                                v++;
        }
```

```
        return(v);
}
void dijkstra(int i,int nVert,int *dist,int **edge,int **weight)

{
        int j=0,ind,v=-1,w,g,rank;
        int *c;
        int indist;
        bool found=false;

        c = create_vector(nVert);


        while (j<nVert)
        {
                if (j==i)
                {
                 c[j] = -10;
                 dist[j]=-1;
                }
                else
                {
                        c[j] = j;
                        dist[j] = get_dist(j,edge[i],weight[i]);
                }
  j++;
        }

        for (j=0; j<nVert-2; j++)
        {
                v=min_out(nVert, c, dist, i);
                ind = c[v];
                c[v] = -10;

                if (ind > -1)
                {
                 for (w=0; w<nVert; w++)
                 {
                        indist=get_dist(w,edge[ind],weight[ind]);
                        if (indist!=-1 && w!=i)
                        if (dist[w] > dist[ind] + indist || dist[w]==-1)
                                        dist[w] = dist[ind] + indist;
                 }
                }
        }
}

void main(int argc, char* argv[])
{
        int r_vertex,source=0,a_ind;
        int nbvertices=0,myId=-1,nTasks=-1,nSlaves=-1;
        int end=0, start=0;
        int **edgemat,**weightmat;
        int *distvect,*distind, **distmat;
        int *recvbuf,*sendbuf,*rbuf,*mst;
        int ind=0,dpointer=0,nport,count_v,sport,ce,recv_e;
        int minv=0,mind=0, max_w=60;
        long totamount=0;
```

```
double t1,t2;

MPI_Init(&argc,&argv);

basic_mpi(&myId,&nTasks,&nSlaves);
get_basic_info(&source,&nbvertices,argc,argv);

nport    = get_port(myId,nTasks,nbvertices);
distvect = create_vector(nbvertices);
sport    = port_size(nTasks,nbvertices);
distind  = create_vector(nport);
ce       =(nbvertices+1)*sport;
sendbuf  = create_vector(ce);

recv_e   = ce * nTasks;
rbuf     = create_vector(recv_e);
distmat  = create_mat(sport,nbvertices);
weightmat=create_table(nbvertices);
edgemat=create_table(nbvertices);

if(myId==0)
{
        randSDgraph(edgemat,weightmat,nbvertices,max_w);
        r_vertex=source;
}

sr_graph(edgemat,weightmat,nbvertices);
t1=MPI_Wtime();
a_ind=0;
for(int f=myId;f<nbvertices;f+=nTasks)
{

dijkstra(f,nbvertices,distmat[a_ind],edgemat,weightmat);
        a_ind++;
}

f=myId;
int pas=0;
int u=0;

while(u<ce)
{
        int r;
        sendbuf[u]=f;
        u++;
        for(r=0;r<nbvertices;r++)
        {
                sendbuf[u]=distmat[pas][r];
                u++;
        }
        pas++;
        f+=nTasks;
}

MPI_Gather(sendbuf,ce,MPI_INT,rbuf,ce,MPI_INT,0,
MPI_COMM_WORLD);
t2=MPI_Wtime();
if (myId == 0)
```

```
        {
                int **totmat;
                int b=0,t=0;
                ind = rbuf[b];
                b++;
                totmat = create_mat(nbvertices,nbvertices);
                if (ind<nbvertices)
                {
                        while(b<recv_e)
                        {
                                if (ind<nbvertices)
                                {
                                        for(t=0;t<nbvertices;t++)
                                        {
                                                totmat[ind][t]=rbuf[b++];
                                        }
                                }
                                else
                                {
                                        b=b+nbvertices;
                                }
                                ind=rbuf[b++];
                        }
                }
        } /* end if master */

        printf("\n time1 = %.3lf \n",t2-t1);

        MPI_Finalize();

} /*end of main*/
```

Parallel Minimum Spanning Tree Prim's Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <mpi.h>

#define  NCOL     6

int get_index(int* distind,int vertex,int nport)
{
        int i=0;

        while(i<nport)
        {
                if (distind[i] == vertex)
                        return(i);
                i++;
        }
}
```

```
int get_port(int p,int np,int nv)
{
        div_t div_result;
        int nport;
        int x=0;

        div_result=div(nv,np);
        if (div_result.rem!=0)
                x=1;
        if (p < div_result.rem)
                return(nport = div_result.quot + x);
        else
                return(nport = div_result.quot);
}

void get_limits(int p,int np,int nv,int* lim,int* sta)
{
        div_t div_result;
        int limit,start;
        int d=0, x=0;

        div_result=div(nv,np);

        if (div_result.rem!=0)
        {
                if ((p-1)<div_result.rem)
                        start = p*(div_result.quot+1);
                else
                {
                        d=p-div_result.rem;
                        start = ((div_result.rem)*(div_result.quot+1))+
                                        (d*div_result.quot);
                }

                if (p<div_result.rem)
                        limit=start+div_result.quot+1;
                else
                        limit=start+div_result.quot;
        }
        else
        {
                start = p*div_result.quot;
                limit = start+div_result.quot;
        }
        *sta=start;
        *lim=limit;

}

void min_dist(int* distvect,int*distind,int dpointer,
                        int* minv,int* min,int nport)
{
        int mv,m,i,v,newp;
        bool found=false;
        dpointer++;
        i=dpointer;
```

```
        while(i<nport && !found)
        {
                mv=distind[i];
                if(distvect[mv]!=-1)
                {
                        found=true;
                        m=distvect[mv];
                }
                i++;
        }
        if(found)
        {
                newp=i;
                for(i=newp;i<nport;i++)
                {
                        v=distind[i];
                        if(distvect[v]<m && distvect[v]!=-1)
                        {
                                m=distvect[v];
                                mv=v;
                        }
                }
        }
        else
        {
                mv=99999;
                m=99999;
        }
        *minv=mv;
        *min=m;
} /* end min_dist */

void get_min(int* recvbuf,int* minv,int* mind,int np)
{
        int mv,md,i=3;

        mv=recvbuf[0];
        md=recvbuf[1];
        while(i<(np*2))
        {
                if(recvbuf[i]<md)
                {
                        mv=recvbuf[i-1];
                        md=recvbuf[i];
                }
                i=i+2;
        }
        *minv=mv;
        *mind=md;
}



void init_dist(int* distvect,int* distind,int rvert,int start,int end,
                int** edgemat,int** weightmat,int* d_p,int nport)
{
        int dpointer,j,i,x;
```

```
        if (rvert<start || rvert>=end)
        {
                dpointer=-1;
                j=0;
                for(i=start;i<end;i++)
                {
                        distind[j]=i;
                        j++;
                }
                for(i=0;i<NCOL;i++)
                {
                        if(edgemat[rvert][i]!=-1)
                        {
                if(edgemat[rvert][i]>=start && edgemat[rvert][i]<end)
                                {
                                        x=edgemat[rvert][i];
                                        distvect[x]=weightmat[rvert][i];
                                }
                        }
                }
        }
        else
        {
                dpointer=0;
                distind[0]=rvert;
                distvect[rvert]=0;
                i=1;
                j=start;
                while(i<nport)
                {
                        if(j!=rvert)
                        {
                                distind[i]=j;
                                j++;
                        }
                        else
                        {
                                j++;
                                distind[i]=j;
                                j++;
                        }
                        i++;
                }

                for(i=0;i<NCOL;i++)
                {
                if(edgemat[rvert][i]!=-1)
                {
                if(edgemat[rvert][i]>=start && edgemat[rvert][i]<end)
                                {
                                        x=edgemat[rvert][i];
                                        distvect[x]=weightmat[rvert][i];
                                }
                        }
                }
        }
        *d_p=dpointer;
} /* end init_dist */
```

```
void append_d(int* distvect,int* distind,int vertex,int start,int end,
                int** edgemat,int** weightmat,int* d_p,int nport)
{
        int dpointer;
        int r=vertex,v,j;
        bool found;

        dpointer=*d_p;
        dpointer++;

        if(r>=start && r<end)
        {
                int ind,temp;
                ind = get_index(distind,r,nport);
                temp= distind[dpointer];
                distind[dpointer]=r;
                distind[ind]=temp;
                dpointer++;
        }

        for(int i=dpointer;i<nport;i++)
        {
                v=distind[i];
                j=0;
                found=false;
                while(j<NCOL && !found)
                {
                if (edgemat[r][j]!=-1 && edgemat[r][j]==v)
                        {
                        found=true;
                        if(distvect[v]==-1 || weightmat[r][j]<distvect[v])
                                        distvect[v]=weightmat[r][j];
                        }
                        j++;
                }
        }
        dpointer--;
        *d_p=dpointer;
} /* end append_d */


long sum(int* distvect,int start,int end)
{
        long amt=0;
        for(int i=start;i<end;i++)
        {
                amt=amt+distvect[i];
        }
        return(amt);
}

/* random generation of Sparse Undirected graph */

int random(int max)
{
        int x;
```

```
        return(x=rand()%max);
}

int finish_index(int *vect)
{
        int i=0;

        while(vect[i]!=-1 && i<NCOL)
                i++;
        return(i);
}

void randSUgraph(int **edg,int **weit,int nv,int max_w)
{
        int i,j,ind,v,rv,rw;

        for(i=0;i<nv-1;i++)
        {
                printf("\ni=%d",i);
                j=finish_index(edg[i]);

                while(j<NCOL)
                {

                        rv = random(nv);

                        if(j==0)
                                while(rv==i)
                                        rv = random(nv);
                        else
                        {

                        while(rv!=i && (rv==edg[i][0] || rv==edg[i][1]
                                        || rv==edg[i][2] || rv==edg[i][3]
                                || rv==edg[i][4] || rv==edg[i][5]))
                                        rv=random(nv);
                        }

                        if(rv==i)
                        {
                        for(v=j;v<NCOL;v++)
                        {
                                edg[i][v]=-1;
                                weit[i][v]=-1;
                        }
                        j=NCOL;
                        }
                        else
                        {
                                ind=finish_index(edg[rv]);
                                if(ind<NCOL)
                                {
                                 edg[i][j]=rv;
                                 rw=random(max_w);
                                 while(rw==0)
                                        rw=random(max_w);
                                 weit[i][j]=rw;
                                 edg[rv][ind]=i;
```

```
                                        weit[rv][ind]=rw;
                                        j++;
                                        }
                                }
                        } /* end while */
                } /* end for */
} /* end randSUgraph */


void main(int argc, char* argv[])
{
        int r_vertex,source=0;
        int nbvertices=0,myId=-1,nTasks=-1,nSlaves=-1;
        int end=0, start=0;
        int **edgemat,**weightmat;
        int *distvect,*distind;
        int *recvbuf,*mst;
        int ind=0,dpointer=0,nport,count_v;
        int minv=0,mind=0, max_w=60;
        long totamount=0;
        double t1,t2,tres=0;

        MPI_Init(&argc,&argv);

        basic_mpi(&myId,&nTasks,&nSlaves);
        get_basic_info(&source,&nbvertices,argc,argv);

        nport    = get_port(myId,nTasks,nbvertices);
        mst      = create_vector(nbvertices);
        distvect = create_vector(nbvertices);
        distind  = create_vector(nport);
        /*initmat=create_table(nbvertices);*/
        weightmat=create_table(nbvertices);
        edgemat=create_table(nbvertices);

        if(myId==0)
        {
        randSUgraph(edgemat,weightmat,nbvertices,max_w);
        r_vertex=source;
        }

        sr_graph(edgemat,weightmat,nbvertices);
        get_limits(myId,nTasks,nbvertices,&end,&start);
        t1=MPI_Wtime();
        MPI_Bcast(&r_vertex,1,MPI_INT,0,MPI_COMM_WORLD);
        init_dist(distvect,distind,r_vertex,start,end,edgemat,
                        weightmat,&dpointer,nport);
        count_v=1;
        mst[0]=r_vertex;
        recvbuf=create_vector(2*nTasks);
        while(count_v < nbvertices)
        {
        min_dist(distvect,distind,dpointer,&minv,&mind,nport);
        int buf[2];
        bzero((void*)recvbuf,2*nTasks*sizeof(int));
        buf[0]=minv;
        buf[1]=mind;
```

```
                MPI_Gather(buf,2,MPI_INT,recvbuf,2,MPI_INT,0,
                                MPI_COMM_WORLD);

                        if(myId==0)
                        {
                                get_min(recvbuf,&minv,&mind,nTasks);
                        }

                        MPI_Barrier(MPI_COMM_WORLD);

        MPI_Bcast(&minv,1,MPI_INT,0,MPI_COMM_WORLD);

                        mst[count_v]=minv;
                        count_v++;
                        append_d(distvect,distind,minv,start,end,edgemat,
                                        weightmat,&dpointer,nport);

                }

                long amount=sum(distvect,start,end);
                MPI_Reduce(&amount,&totamount,1,MPI_LONG,MPI_SUM,
                0,MPI_COMM_WORLD);
                t2=MPI_Wtime();

                printf("mst = %ld",totamount);
                printf("\n time = %.3lf \n",t2-t1);

                MPI_Finalize();

} /*end of main*/
```

Basic MPI Routines

The following is a collection of MPI routines. The routines described here are divided into preliminaries (those for establishing the environment and related matters), basic point-to-point message passing, and collective message passing.

Preliminaries

int **MPI_Init** (argc, argv)

initializes MPI environment.

int **MPI_Finalize** ( )

Terminates MPI environment

int **MPI_Comm_rank** (communicator, rank)

Determines rank of processor in communicator

int **MPI_Comm_size** (communicator, size)

determines size of group associated with communicator.

double **MPI_Wtime** ( )

returns elapsed time from some point in past, in seconds.

POINT-TO-POINT MESSAGE PASSING

int **MPI_Send** (buf, count, datatype, dest, tag, communicator)

sends message (blocking)

int **MPI_Recv** (buf, count, datatype, source, tag, communicator, status)

Receives message (blocking)

MPI_Isend, **MPI_Irecv** : nonblocking *send* and *receive*.

Group Routines

int **MPI_Barrier** (communicator)

Blocks process until all processes have called it.

int **MPI_Bcast** (buf, count, datatype, root, communicator)

Broadcasts message from root process to all processes in communicator and itself.

int **MPI_Gather** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, communicator)

Gathers values for group processes

intMPI_Reduce

(sendbuf,recvbuf,count,datatype,operation,root,communicator)

Combines values on all processes to single value

GLOSSARY

Barrier A synchronization point. No process may continue execution beyond the barrier until all processes have reached the barrier.

Broadcast When one processor sends a value or set of values to all other processors

Complexity Measure of time or space used by an algorithm. without adjective, refers to time complexity.

Connected Component Given an undirected graph G, find the minimal set of subgraphs such that every subgraph is connected. Also known as the component labeling problem.

Cost Product of the time complexity of an algorithm and the maximum number of processors used.

CRCW PRAM Concurrent read, concurrent write PRAM

CREW PRAM Concurrent read, exclusive write PRAM

Directed graph Graph with ordered edges.

Efficiency Ratio of speedup to number of processors used.

Embarrassingly Parallel Computation Ideal computation from a parallel computing standpoint. A computation that can be divided into a number of completely independent parts, each of which can be executed by a separate processor.

EREW PRAM Exclusive read, exclusive write PRAM.

Graph Coloring Problem Determine whether vertices of a graph can be colored with c colors so that no two adjacent vertices are assigned the same color.

Multicomputer A multiple-CPU parallel computer lacking a shared memory.

Multiprocessor A shared memory multiple-CPU parallel computer.

Parallelization The process of making an algorithm parallel.

PRAM Model of parallel computation consisting of a control unit, global memory, and an unbounded set of processors, each with its own private memory.

Speedup Ratio between the time needed for the most efficient sequential algorithm to perform a computation and the time needed to perform the same computation on a machine incorporating pipelining and/or parallelism.