1-2021

Parallelizing Dijkstra's Algorithm

Mengqing He
mhe@stcloudstate.edu

Parallelizing Dijkstra's Algorithm

by

Mengqing He

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in

Computer Science

December, 2020

Starred Paper Committee:
Jie H. Meichsner, Chairperson
Andrew A. Anda
Bryant A. Julstrom

Abstract

Dijkstra's algorithm is an algorithm for finding the shortest path between nodes in a graph. The algorithm published in 1959 by Dutch computer scientist Edsger W. Dijkstra, can be applied on a weighted graph. Dijkstra's original algorithm runtime is a quadratic function of the number of vertices.

In this paper, I will investigate the parallel formulation of Dijkstra's algorithm and its speedup against the sequential one. The implementation of the parallel formulation will be performed by Message Passing Interface (MPI) and Open Multi-Processing (OpenMP). The results gained indicated that the performance of MPI and OpenMP to be significantly better than sequential for a higher number of input data scale. And the smaller number of processors/threads give the fastest result for MPI and OpenMP implementation. However, the results show that the average speedup achieved by parallelization is not satisfied. The parallel implementation of Dijkstra's algorithm may not be the best option.

Keywords: Dijkstra's algorithm; graph; parallel computing; MPI; OpenMP; performance

Acknowledgement

I would like to thank my advisor Dr. Meichsner for offering a lot of valuable help and suggestions to my paperwork. Without her help, I cannot finish this paper smoothly. I would also like to thank the committee members Dr. Anda and Dr. Julstrom for sharing their valuable time and advice on my paper research work.

Table of Contents

List of Algorithms

List of Tables

List of Figures

Figure Page

Chapter 1: Serial Dijkstra's Algorithm

1.1 Introduction

A graph consists of a set of vertices or nodes, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph [1]. These pairs are known as edges, arcs, or lines for an undirected graph and as arrows, directed edges, directed arcs, or directed lines for a directed graph. Graphs are implemented as data structures by the adjacency list and adjacency matrix. In this paper, we talk about an undirected and non-negative weighted graph. Figure 1(a) is an undirected graph with non-negative weights. Figure 1(b) is an adjacency list representation of the undirected graph in Figure 1(a). Similarly, Figure 1(c) is an adjacency matrix representation of the graph in Figure 1(a).

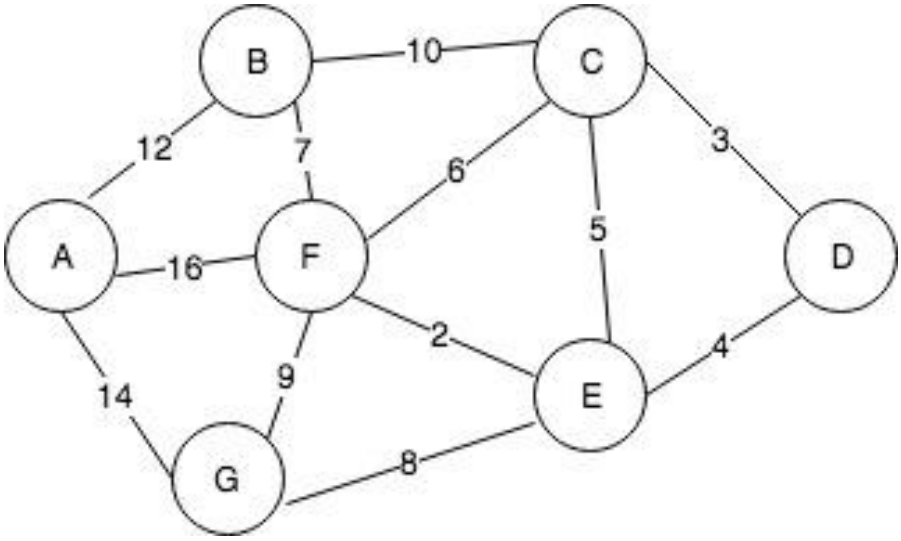

Figure 1(a). The undirected graph *G* with 7 vertices, 12 edges and non-negative weight.

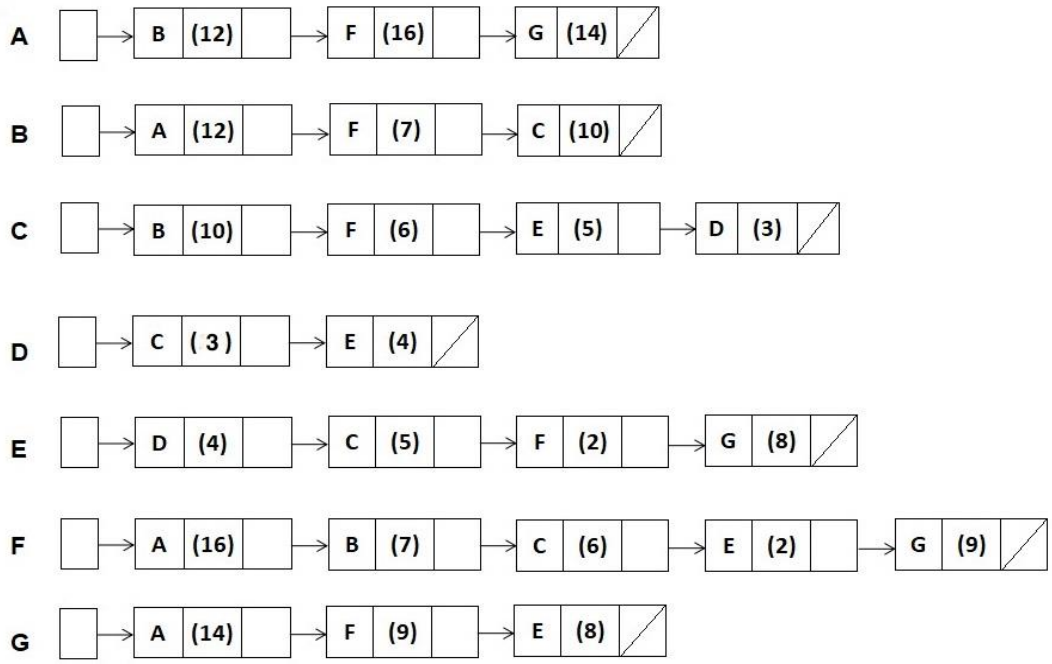

Figure 1(b). The adjacency list representation of *G.*

	A	B	C	D	E	F	G
A	0	12	∞	∞	∞	16	14
B	12	0	10	∞	∞	7	∞
C	∞	10	0	3	5	6	∞
D	∞	∞	3	0	4	∞	∞
E	∞	∞	5	4	0	2	8
F	16	7	6	∞	2	0	9
G	14	∞	∞	∞	8	9	0

Figure 1(c). The adjacency matrix representation of *G.*

Suppose we have a given weighted graph $G = (V, E, w)$, where V is the set of vertices in this graph and E is the set of edges that connect with vertices, w is the set of weights of these edges. The single source shortest paths problem is to find the shortest paths from a vertex $s \in V$ to all other vertices in V [2]. A shortest path from vertex s to vertex v is a minimized-weight path. Depending on the application, edge weights may represent time, cost, penalty, loss, or any other quantity that accumulates additively along a path and is to be minimized.

Dijkstra's algorithm is a greedy algorithm. A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Dijkstra's algorithm incrementally finds the shortest paths from s to the other vertices of G. It always chooses an edge to a vertex that appears closest.

There are several variants of Dijkstra's algorithm [3]; the original variant found the shortest path between two specific vertices, but a more common variant fixes a single vertex as the source vertex and finds shortest paths from the source to all other vertices in the graph, producing a shortest-path tree [4].

Dijkstra's algorithm can solve the single source shortest path problem on a graph. For a given source vertex in the graph, the algorithm finds the shortest path between the vertex and every other vertex. The solution to the shortest path problem is not unique. If it exists several paths from source vertex to the specific vertex, Dijkstra's algorithm will choose one path arbitrary. In particular, it depends on the order in which we traverse the vertices in each iteration.

1.2 Pseudo Code

```
/* V: set of vertices in the graph;
```

```
* E: set of edges in the graph;
* w: set of weights of these edges;
* s: source vertex;
* /
```

```
1.  procedure DIJKSTRA_SINGLE_SOURCE_SP (V, E, w, s)
2.  begin
3.      S := {s};       // S holds the vertices that the shortest path has been found
4.      for all v ∈ U do       // U = V - S
5.          if (s, v) exists set d[v] := w(s, v);     // d[v] holds the min weight from s to v
6.          else set d[v] := ∞
7.      while S ≠ V do
8.      begin
9.          find a vertex u such that d[u] := min{d[v] | v ∈ U};
10.         S := S ∪ {u};
11.         for all v ∈ U do
12.             d[v] := min{d[v], d[u] + w(u, v)};   // update min weight of other vertices
13.     endwhile
14. end DIJKSTRA_SINGLE_SOURCE_SP
```

Algorithm 1. Dijkstra's sequential single-source shortest paths algorithm [2].

From the pseudo code, the time complexity is at line 7~line 12. In the graph (*V, E, w*), *V* is all vertices in the graph and *E* presents all edges. The first level loop at line 7, the time is *O* ($|V|$). At line 9, get the best vertex, cost time *O* ($|V|$). The second level loop at line 11, the time is *O* ($|E|/|V|$). The total time complexity is $O\left(|V| * (|V| + |E|/|V|)\right) = O(|V|^2 + |E|) \rightarrow O(|V|^2) \rightarrow O(n^2)$.

1.3 Description

The main feature of Dijkstra's algorithm is to extend the outer layer (the breadth-first search idea) around the source vertex until it reaches the end vertex.

When calculating the shortest path in the Graph *G*, we specify the starting vertex *s* (that is, starting from the source vertex *s*). In addition, two sets *S* and *U* are introduced. The role of *S* is to record the vertices and the corresponding shortest path length for which the shortest path has been found. The set *U* is used to record the vertices and the distance from the vertices to the source vertex *s* which the shortest path has not been found. Initially, there is only the source vertex *s* in *S*; *U* contains vertices other than *s*, and the path of the vertex in *U* is the path from source vertex to this vertex. Then, find the shortest path for this vertex from *U* and add it to *S*, update the vertex and the corresponding path in *U*. Then, find the shortest vertex of the path from *U* and add it to *S*, update the vertex and the corresponding path in *U* … repeat the operation until all the vertices have been traversed.

(1) Initially, *S* only contains the starting vertex *s*; *U* contains other vertices except *s*, and the distances. The distance is the weight from the starting vertex *s* to the vertices in *U*. For example, the distance of the vertex *v* in *U* is ∞ if *s* and *v* are not adjacent.

(2) Select the shortest vertex *u* from *U* and add vertex *u* to *S*; meanwhile, remove vertex *u* from *U*;

(3) Update the distance from each vertex in *U* to the source vertex. The reason why the distance of the vertices in *U* is updated is that in the previous step *u* is the vertex of the shortest path, so that the distance of other vertices can be updated by *u*; for example, the distance of (*s*, *v*) may be greater than the distance (*s*, *u*) + (*u*, *v*).

(4) Repeat steps 2 and 3 until all the vertices have been traversed.

Simply looking at the above theory may be difficult and misunderstood. The algorithm can be illustrated by an example Figure 2(a). We would like to compute the distances from source vertex *D* to other vertices.

1) Choose source vertex *D*

   S = {*D* (0)}

   U = {*A* (∞), *B* (∞), *C* (3), *E* (4), *F* (∞), *G* (∞)}

   S is the set of vertices that the shortest path has been calculated.

   U is the set of vertices that the shortest path has not been calculated.

   C (3) means the distance from vertex *C* to *D* is 3.

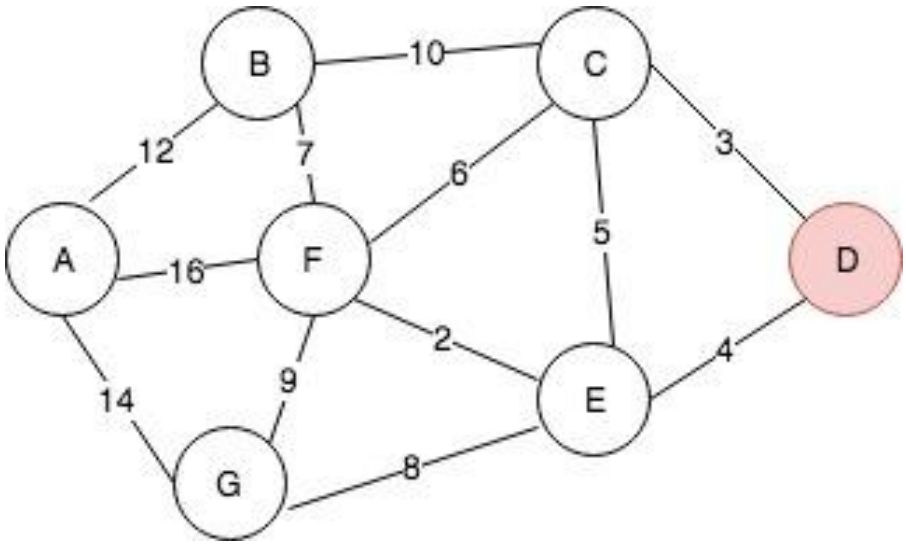

Figure 2(a). Choose source vertex *D*.

2) Choose vertex *C*, add vertex *C* to *S*

   After the previous operation, the distance from vertex *C* to source vertex *D* in *U* is the shortest. Therefore, *C* is added to *S* and we update the distance of the vertices in *U*. Taking the vertex *F* as an example, the distance from the previous *F* to *D* is ∞; but after adding *C* to *S*, the distance from *F* to *D* is 9 = (*F*, *C*) + (*C*, *D*).

S = {D (0), C (3)}

U = {A (∞), B (13), E (4), F (9), G (∞)}

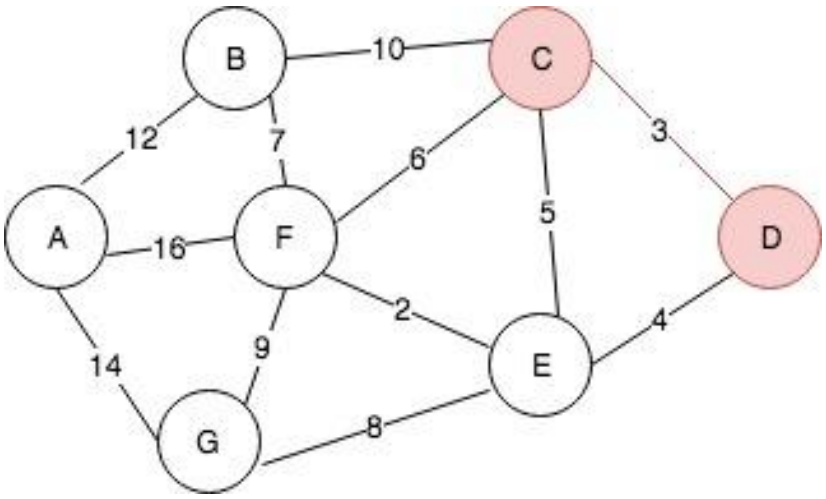

Figure 2(b). Choose vertex *C*.

3) Choose vertex *E*, add vertex *E* to *S*

After the previous operation, the distance from the vertex *E* to the source vertex *D* is the shortest. Therefore, *E* is added to *S* and we update the distance of vertices in *U*. For example, the distance from *F* to *D* is 9; but after adding *E* to *S*, the distance from *F* to *D* is 6 = (*F*, *E*) + (*E*, *D*).

S = {D (0), C (3), E (4)}

U = {A (∞), B (13), F (6), G (12)}

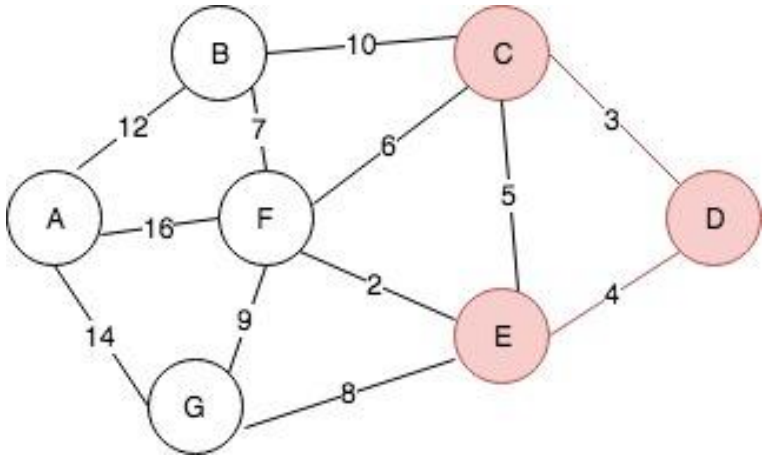

Figure 2(c). Choose vertex *E*.

4) Choose vertex *F*

S = {*D* (0), *C* (3), *E* (4), *F* (6)}

U = {*A* (22), *B* (13), *G* (12)}

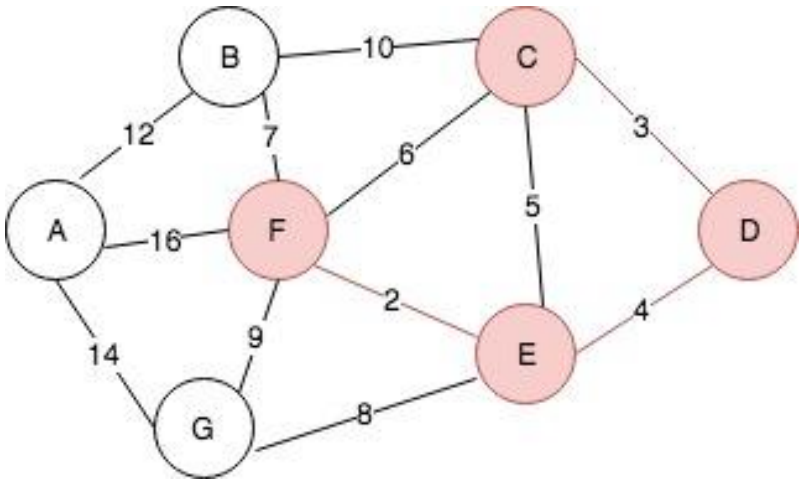

Figure 2(d). Choose vertex *F*.

5) Choose vertex *G*

S = {*D* (0), *C* (3), *E* (4), *F* (6), *G* (12)}

U = {*A* (22), *B* (13)}

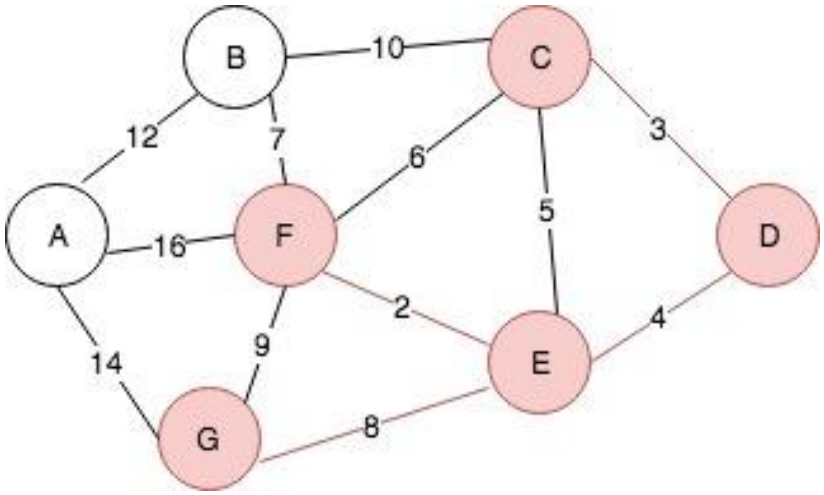

Figure 2(e). Choose vertex *G*.

6) Choose vertex *B*

S = {*D* (0), *C* (3), *E* (4), *F* (6), *G* (12), *B* (13)}

U = {*A* (22)}

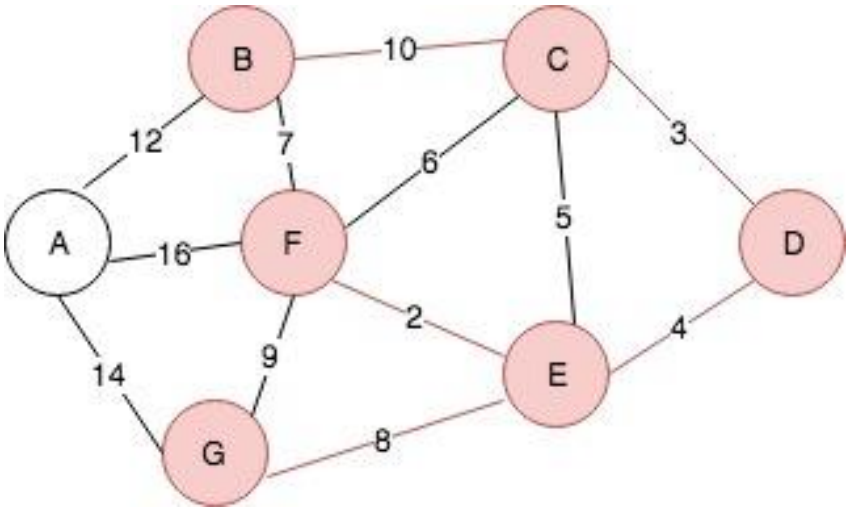

Figure 2(f). Choose vertex *B*.

7) Choose vertex *A*

S = {*D* (0), *C* (3), *E* (4), *F* (6), *G* (12), *B* (13), *A* (22)}

U = {}

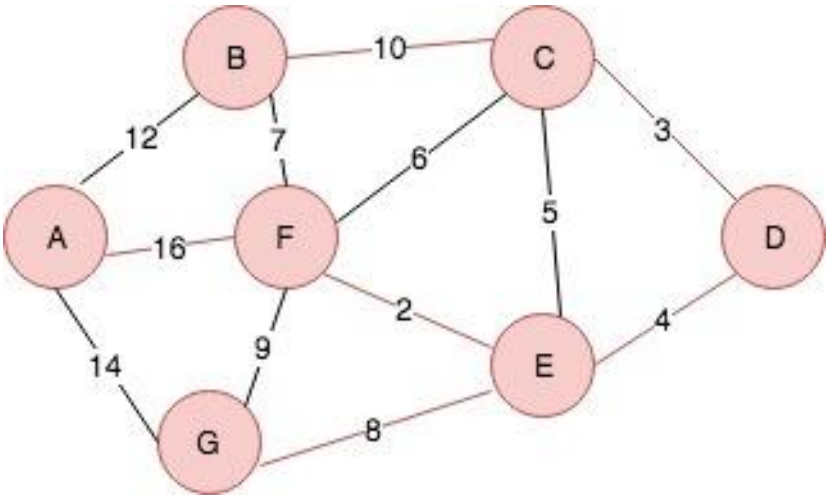

Figure 2(g). Choose vertex *A.*

At this point, the shortest distance from the source vertex *D* to each vertex is calculated:

A (22) *B* (13) *C* (3) *D* (0) *E* (4) *F* (6) *G* (12)

Chapter 2: Dijkstra's Algorithm in Parallel Computing

2.1 Parallel Computing System

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.

Here are some reasons why we need parallel computing:

- Save time and/or money.
- Solve larger/more complex problems.
- Provide concurrency.
- Take advantage of non-local resources.
- Make better use underlying parallel hardware.

From computational simulation in scientific and engineering applications to business applications in data mining and transaction processing, parallel computing has made a huge impact in various fields. The cost advantages of parallelism and the performance requirements of applications make compelling arguments for supporting parallel computing.

2.2 Communication Model of Parallel Platforms

There are two main forms of data exchange between parallel tasks-accessing shared data space and exchanging messages.

2.2.1 Shared-Address-Space Platforms [2]

The Shared-Address-Space view of the parallel platform supports a common data space accessible by all processors. The processor interacts by modifying the data object stored in this shared-address-space. A shared-address-space platform that supports program multiple data (SPMD) programming is also known as a multiprocessor. Memory in a shared-address-space

platform can be local (processor-specific) or global (common to all processors). If it takes the same amount of time for the processor to access any memory word (global or local) in the system, the platform will be classified as a unified memory access (UMA) multicomputer. On the other hand, if it takes longer to access some memory words than others, the platform is called non-uniform memory access (NUMA) multicomputer. Figure 3(a) and (b) illustrated the UMA platform, and Figure3(c) illustrates the NUMA platform. In Figure 3(b), accessing stored words in the cache is faster than accessing locations in memory. However, we still classify it as a UMA architecture. The reason is that all current microprocessors have a cache hierarchy. Therefor. If you consider cache access time, even a single processor would not be called UMA. Therefore, we define NUMA and UMA architectures based on memory access time, not cache access time. The existence of global memory space makes programming such platforms easier. Programmers do not see all read-only interactions because they are encoded in the same way as in serial programs. This greatly reduces the burden of writing parallel programs.

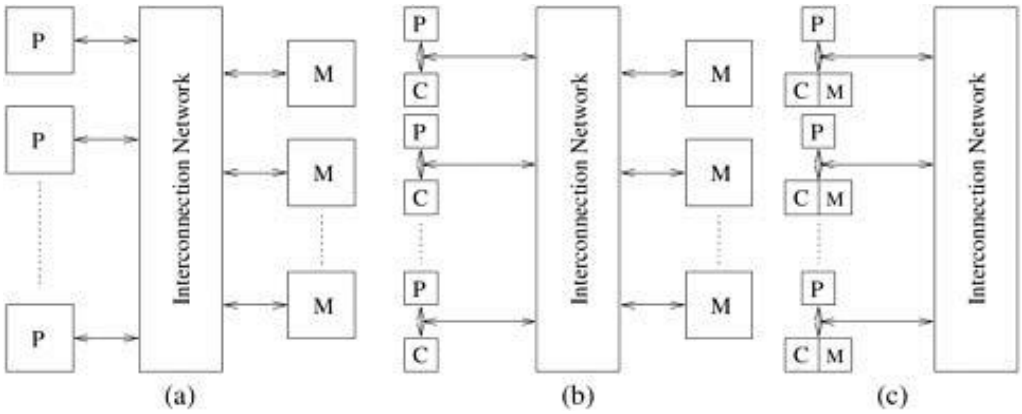

Figure 3. Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space ccomputer with caches and memories; (c) non-uniform-memory-access shared-address-space computer with local memory only [2].

Shared-address-space programming paradigms such as threads (POSIX, NT) and directives (OpenMP) support synchronization using locks and related mechanisms.

OpenMP stands for Open Multi-Processing. OpenMP is an API that can be used with FORTRAN, c and C++ for programming shared address space machines. All OpenMP programs begin as a single process called the master thread. When the master thread reaches the parallel region, it creates muiltiple threads to execute the parallel codes enclosed in the parallel region. When the threads complete the parallel region, they synchronize and terminate, leaving only the master thread. We initiate the OpenMP programming model with the aid of a simple program. OpenMP directives on C and C++ are based on the `#pragma` compiler directives. The directive itself consists of a directive name followed by clauses.

```
#pragma omp parallel [clause list]
```

OpenMP programs execute serially until they encounter the `parallel` directive. This directive is responsible for creating a group of threads. The exact number of threads can be specified in the directive, set using an environmnet variable, or at runtime using OpenMP functions. The main thread that encounters the `parallel` directive becomes the master of this group of threads and is assigned the thread id 0 within the group. Each thread created by this directive executes the structured block specified by the `parallel` directive. The clause list is used to specify conditional parallelization (`if`), number of threads (`num_threads`), and data handling (`private`, `firstprivate`).

2.2.2 Message-Passing Platforms [2]

The logical view of a machine supporting the message-passing paradigm consists of *p* processes, each with its own exclusive address space. Each data element must belong to one of

the partitions of the space; hence, data must be explicitly partitioned and placed. On such platforms, interactions between processes running on different nodes must be accomplished using messages, hence the name message passing. This exchange of messages is used to transfer data, work, and to synchronize actions among the processes. In its most general form, message-passing paradigms support execution of a different program on each of the *p* nodes.

All interactions (read-only or read/write) require cooperation of two processes-the process that has the data and the process that wants to access the data. Most message-passing programs are written using the single program multiple data (SPMD) model.

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C/C++ and Fortran. MPI's goals are high performance, scalability, and portability. The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to computer instances) in a language-independent way. MPI library functions include, but not limited to, Point-to-Point (Send and Receive Routines), Collective Communication and Computation Operations (Barrier, Broadcast, Reduction, Gather, Scatter, All-to-All), Groups and Communicators (Split).

2.3 Parallel Formulation of Dijkstra's Algorithm [2]

According Algorithm 1, Dijkstra's algorithm is iterative. Each iteration adds a new vertex to the computed set. Since the value of $d[v]$ for a vertex v may change every time a new vertex u is added in S, it is hard to select more than one vertex. This is not easy to perform different

iterations of the while loop in parallel. However, each iteration can be performed in parallel as follows.

Let p be the number of processes, and let n be the number of vertices in the graph. The set V is partitioned into p subsets using the 1-D block mapping. Each subset has n/p consecutive vertices, and the work associated with each subset is assigned to a different process. Let V_i be the subset of vertices assigned to process P_i for $i = 0, 1, \ldots, p - 1$. Each process P_i stores the part of the array d that corresponds to V_i (Figure 4.a). Each process P_i computes $d_i[u] = \min\{d_i[v] | v \in (V - S) \cap V_i\}$ during each iteration of the while loop. The global minimum is then obtained over all $d_i[u]$ by using the all-to-one reduction operation and is stored in process P_0. Process P_0 now holds the new vertex u, which will be inserted into S. Process P_0 broadcasts u to all processes by using one-to-all broadcast. The process P_j responsible for vertex u marks u as belonging to set S. Finally, each process updates the values of $d[v]$ for its local vertices.

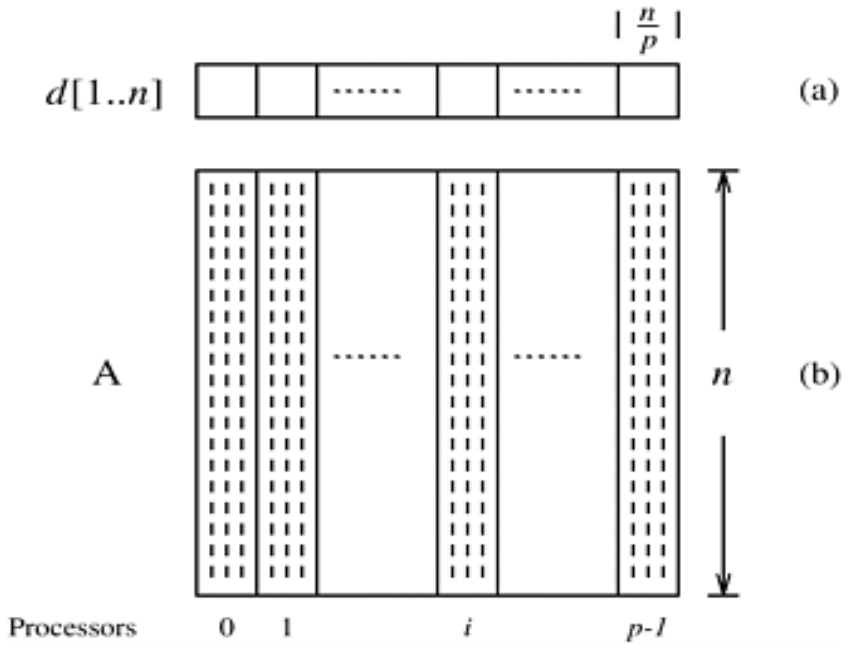

Figure 4. The partitioning of the distance array d and the adjacency matrix A among p processes [2].

When a new vertex *u* is inserted into *S*, the values of *d*[*v*] for *v* $\in$ (*V* - *S*) must be updated. The process responsible for *v* must know the weight of the edge (*u, v*). Hence, each process P_i needs to store the columns of the weighted adjacency matrix corresponding to set *S* of the vertices assigned to it. This corresponds to 1-D block mapping of the matrix. The space to store the required part of the adjacency matrix at each process is Θ (n^2/p). Figure 4.b illustrates the partitioning of the weighted adjacency matrix.

The computation performed by a process to minimize and update the values of *d*[*v*] during each iteration is Θ (*n/p*). The communication performed in each iteration is due to the all-to-one reduction and the one-to-all broadcast. For a *p*-process message-passing parallel computer, a one-to-all broadcast to one word takes time log *p*. Finding the global minimum of one word at each iteration is Θ (log *p*). The parallel run time of this formulation is given by

$$T_P = \Theta\left(\frac{n^2}{p}\right) + \Theta(n \log p).$$

Equation 1 [2]

Since the sequential run time is $W = \Theta(n^2)$, the speedup and efficiency are as follows:

$$S = \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n \log p)}$$

$$E = \frac{1}{1 + \Theta((p \log p)/n)}$$

Equation 2 [2]

For a cost-optimal parallel formulation ($p \log p$) /n = O (1). Thus, this formulation of Dijkstra's algorithm can use only p = O (n/log n) processes. Furthermore, the isoefficiency function due to communication is Θ ($p^2 \log^2 p$). Since n must grow at least as fast as p in this

formulation, the isoefficiency function due to concurrency is $\Theta(p^2)$. Thus, the overall isoefficiency of this formulation is $\Theta(p^2 \log^2 p)$ [2].

Chapter 3: Parallel Design and Implementation of Dijkstra's Algorithm

3.1 Technologies

In this part, we talk about which technology to use in the implementation and made various decisions.

3.1.1 Message Passing API/system

The message passing API must be available on all systems on which it is implemented, and it should be as simple and straightforward as possible, preferably supporting collective operations. The MPI matches this, so I did not seriously consider alternatives when choosing it. The MPI implementation is free, easily available, with C bindings, and I already know about it.

3.1.2 Shared Address API/system

The chosen shared address API must allow a lot of control over the tasks that are performed simultaneously. It must also be provided for free. OpenMP provides a layer on top of native threads to facilitate various thread-related tasks. Using the instructions provided by OpenMP, the programmer does not need to perform the task of initializing the attribute object, setting parameters for the thread, and dividing the iteration space. This facility is especially useful when the underlying problem has a static or regular task diagram. In the context of various applications, the overhead associated with automatically generating thread code from instructions has been shown to be minimal.

3.1.3 Language

I choose C due to the availability of a relatively stable MPI implementation for message passing and the library for OpenMP.

3.1.4 System Environment

The serial Dijkstra's algorithm implementation, Dijkstra's algorithm implementation in MPI, Dijkstra's algorithm implementation in OpenMP, the three implementations are run in Minnesota Supercomputing Institute (MSI) system [5].

3.2 Test Data

After the technologies to use were determined, I write a JAVA program to generate test data. This program generates a 2D array with random numbers in the range of 1 to 15, which represents the input graph for Dijkstra's algorithm. The weights are created from the Random function in JAVA. Assume *G* is a 2D array, *G*[i][j] represents the weight from vertex *i* to vertex *j*. If they have no direct connect, the weight is set as 9999999, otherwise it is a random number between 1 ~15. If *i* = *j*, that means it's the vertex *i* (or *j*) itself. We set *G*[i][j] = 0 if (*i*==*j*). The weights are randomly generated. The calculated distance will not be too large. This does not affect our experimental goals because I only need to get results from different programs that use same data. I will run the same set of data on serial Dijkstra's algorithm implementation, Dijkstra's algorithm implementation in MPI, Dijkstra's algorithm implementation in OpenMP to compare the time consumed. We totally have six sets of data are used for input adjacency matrix. That means, for the graph, 8 vertices, 64 vertices, 256 vertices, 512 vertices, 1024 vertices and 2048 vertices are used. Here is an example for the 8 vertices matrix.

0	2	9999999	3	4	3	9999999	3
2	0	8	8	9	9999999	7	7
9999999	8	0	6	7	9999999	9999999	2
3	8	6	0	7	3	9	7
4	9	7	7	0	9999999	9999999	4
3	9999999	9999999	3	9999999	0	8	3
9999999	7	9999999	9	9999999	8	0	9999999
3	7	2	7	4	3	9999999	0

3.3 Algorithm

Details on how the algorithm was implemented are given in the section below. The complete source code for the implementations described can be found in Appendix A. Pseudo-code describing the implementations in simplified form has been provided here.

3.3.1 Implementation Dijkstra's Algorithm with MPI

The algorithm implementation, simplified in the pseudocode, is shown below.

```
/* wgt: points to locally stored portion of the weight adjacency matrix of the graph;
 * lengths: points to a vector that will store the distance of the shortest paths from
 * the source to the locally stored vertices;
 * /
```

```
1.  procedure DIJKSTRA_SINGLE_SOURCE_SP_MPI (V, E, wgt, lengths, s)
2.  begin
3.      for  all v ∈ V do
4.        set lminpair[0]:local min distance;
5.        set lminpair[1]:corresponding vertex;
6.        for vertices in each processor
7.              find a vertex u at the smallest distance from the source s;
8.      MPI_Allreduce();
9.      Get the global minmum vertex u and mark it;
10.     for all v ∈ nlocals do  // The number of vertices stored locally.
11.       lengths[v] := min{lengths[v], udist + wgt[u*nlocal + v]};
12.     endwhile
13. end DIJKSTRA_SINGLE_SOURCE_SP_MPI
```

Algorithm 2. Dijkstra's MPI single-source shortest paths algorithm.

The main computational loop of Dijkstra's parallel single-source path algorithm executes three steps. First, each process will find the locally stored vertex in *Vo* with the shortest distance from the source. Then, the process determines the vertex with the shortest distance and includes it in *Vc*. Third, each process updates the distance array to reflect the fact that *Vc* contains new vertices.

The first step is to scan the vertices stored locally in *Vo* to determine the short vertex *v* [*v*]. The calculation result is stored in the array *lminpair*. Specifically, *lminpair*[*0*] stores the distance between vertices, and *lminpair*[*1*] stores the vertices themselves. Consider the following steps to clarify why this storage solution should be used. The next step is to calculate the vertex with the smallest total distance to the source. We can find the sum of the shortest distance by minimizing the distance value stored in *lminpair*[*0*].

However, in addition to the shortest distance, we also need to know the specific vertex of the shortest distance. Therefore, the appropriate reduction operation is `MPI_MINLOC`, which returns the minimum value and the index value associated with the minimum value. Because of `MPI_MINLOC`, we use a two-element array lminpair to store the distance and the vertex that reaches that distance. In addition, all processes need the result of the restore operation to perform the third step, so we use the `MPI_Allreduce` operation to perform the reduction. The result of the reduction operation is returned to the *gminpair* array. We can perform the third and last step of each iteration by scanning the local vertices belonging to *Vo* and updating the shortest distance between them and the source vertex.

In our MPI program, we assign *n/p* consecutive *W* columns to each processor, and uses the `MPI_MINLOC` reduction operation to select the vertex v to be included in *Vc* at each iteration. Recall that the index returned by the `MPI_MINLOC` operation on (*a, i*) and (*a, j*) has a smaller index (because the value is the same). Therefore, among the vertices that are close to the source vertices, they are biased toward the least vertices. This can lead to load imbalance, because vertices stored in lower-level processes tend to be included in *Vc* faster than vertices in higher-level processes (especially many vertices in *Vo* have the smallest same distance to source). Therefore, in higher-level processes, the configured *Vo* size will be larger, and the entire runtime will dominate.

One way to solve this problem is to use circular distribution to distribute the columns of *W*. This allocation process will get all *p* vertices starting from vertex *i.* In this scheme, each process also allocates *n/p* vertices, but the indexes of these vertices almost cover the entire graph. Therefore, `MPI_MINLOC` preferentially selects the vertex with the smallest number, and will not cause load imbalance [2].

3.3.2 Implementation Dijkstra's Algorithm with OpenMP

Most OpenMP constructs apply to a structured block, that is a block of one or more statements with one point of entry at the top and one point of exit at the bottom. We can find computational intensive loops in Dijkstra's sequential algorithm and make the loop iterations independent, then place the appropriate OpenMP directives and test.

```
1.  procedure DIJKSTRA_SINGLE_SOURCE_SP_OPENMP (V, E, w, distances, s)
2.  begin
3.      #pragma omp parallel private
4.        shared ()
5.          omp_get_thread_num ( );
6.          omp_get_num_threads ( );
7.          Each thread finds the min distance u unconnected vertex inner
8.          # pragma omp critical   // update overall min
9.          # pragma omp barrier
10.         # pragma omp single   // mark new vertex as done
11.         for all v in each thread
12.             distances[v] := min{distances[v], distances[u]+ w[u][v]};
12.     endwhile
13. end DIJKSTRA_SINGLE_SOURCE_SP_OPENMP
```

Algorithm 3. Dijkstra's OpenMP single-source shortest paths algorithm.

As the pseudocode shows, OpenMP Dijkstra's algorithm implementation is very similar to the sequential one. Compared with MPI implementation, OpenMP has less lines of code. The function `omp_set_num_threads` sets the default number of the threads that will be created on encountering the next `parallel` directive. We use this function in the *main* function.

The `omp_get_num_threads` function returns the number of threads participating in a team. The `omp_get_thread_num` returns a unique thread id for each thread in a team. This integer lies between 0 and `omp_get_num_threads()` − 1.

The `critical` directive ensures that at any point in the execution of the program, only one thread is within a critical section specified by a certain name.

OpenMP provides a `critical` directive for implementation critical regions. There is a critical region that allows different threads to execute different code while being protected from each other.

A barrier is one of the most frequently used synchronization primitives. OpenMP provides a `barrier` directive. On encountering this directive, all threads in a team wait until others have caught up, and then release.

A `single` directive specifies a structured block that is executed by a single thread. On encountering the `single` block, the first thread enters the block. All the other threads proceed to the end of the block [2].

Chapter 4: Implementation Results and Analysis

Table 1 contains all primary results of running the Dijkstra's algorithm in sequential, MPI and OpenMP programs. The code is run in the same system environment, and the input data source is generated by Random function. In this paper, the total six sets of data are used for input adjacency matrix. That means, for the graph, 8 vertices, 64 vertices, 256 vertices, 512 vertices, 1024 vertices and 2048 vertices are used. After running the code, we can get the results, which is the duration in seconds. For OpenMP and MPI parallel computation, 2, 4, 8, 16, 32, 64, 128, 256, 512 processors (the numbers of vertices should larger than processors) are used to run the code.

Table 1. Execution time in seconds for all three implementations.

	8 vertices	64 vertices	256 vertices	512 vertices	1024 vertices	2048 vertices
Seq	0.0035	0.0029	0.0099	0.1162	0.3568	0.9965
OpenMP2	0.0003	0.0015	0.0093	0.0651	0.2493	0.8583
OpenMP4	0.0003	0.0016	0.0109	0.0635	0.2488	0.7933
OpenMP8	0.0007	0.8831	0.0140	0.0854	0.2807	0.9432
OpenMP16		0.0574	0.0233	0.2736	0.6794	1.7490
OpenMP32		0.0640	0.1186	0.5117	1.3014	2.8708
OpenMP64		0.0864	0.2030	0.7614	1.7558	3.5983
OpenMP128			0.3302	1.3960	2.8929	5.9080
OpenMP256			0.9902	2.7855	5.3655	10.5335
OpenMP512				6.1405	10.6755	20.9920
MPI2	0.0246	0.0178	0.0287	0.1137	0.1316	0.5040
MPI4	0.0353	0.0210	0.0218	0.0413	0.1281	0.4723
MPI8	0.0144	0.0164	0.0264	0.0440	0.1324	1.3663
MPI16		0.0264	0.0453	0.1010	0.2055	3.3525
MPI32		10.8733	1.2415	56.9774	3.9622	21.4067
MPI64		27.2983	3.1004	67.9168	7.1446	41.7286
MPI128			7.3566	19.8329	17.9661	119.9792
MPI256			18.8646	32.8870	44.8353	343.0892
MPI512				488.7678	1733.9085	2399.9986

Table 2. The best execution time in seconds the three implementations (the numbers in brackets indicate how many threads/processors).

	8 vertices	64 vertices	256 vertices	512 vertices	1024 vertices	2048 vertices
Seq	0.0035	0.0029	0.0099	0.1162	0.3568	0.9965
OpenMP	0.0003(2)	0.0015(2)	0.0093(2)	0.0635(4)	0.2488(4)	0.7933(4)
MPI	0.0144(8)	0.0164(8)	0.0218(4)	0.0413(4)	0.1281(4)	0.4723(4)

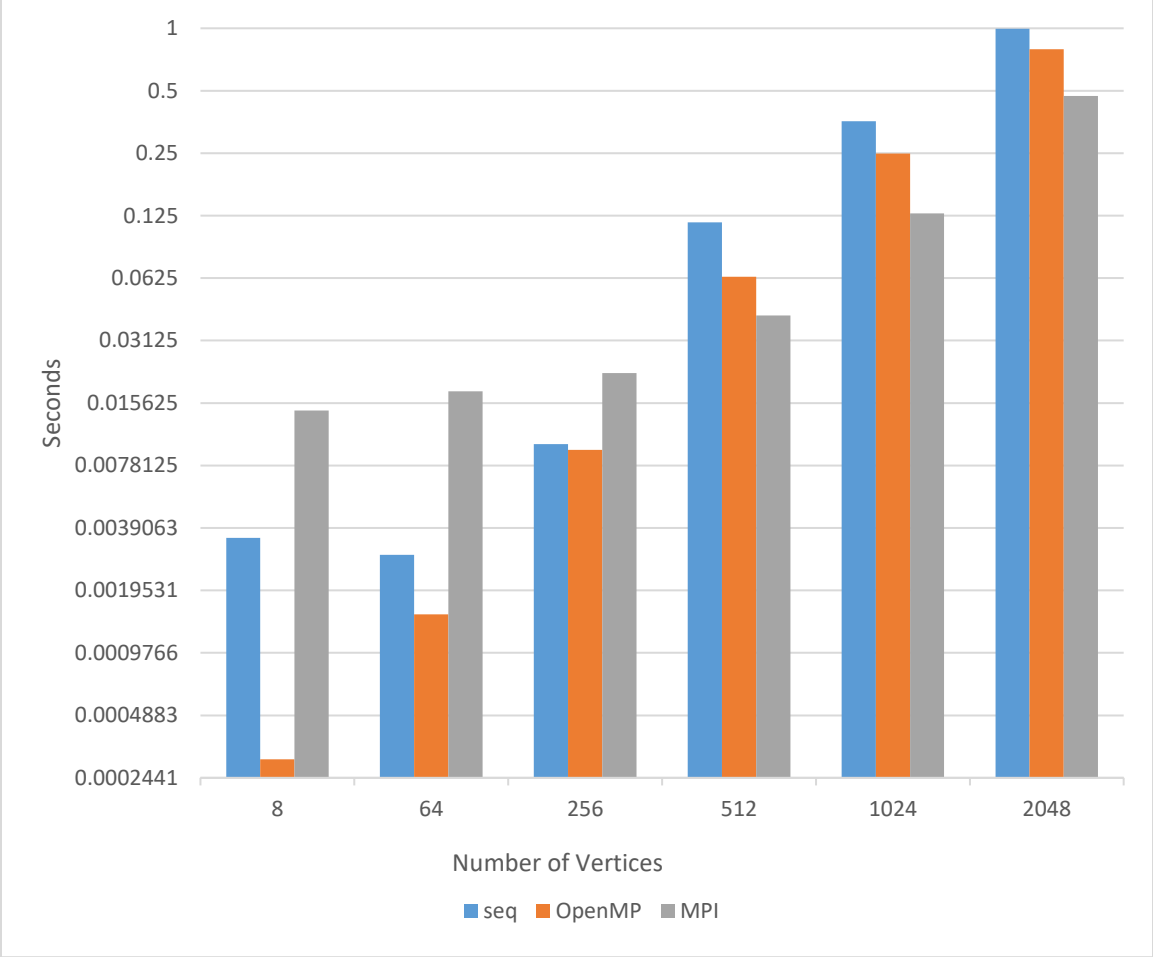

Figure 5. Best execution time for each implementation at different data sets.

Figure 5 shows the best execution time for sequential, OpenMP and MPI with different number of vertices. We can see the performance is better when using OpenMP and MPI to run the algorithm. For a small number of vertices, more time could be spent on parallelization and synchronization than it is spent on execution of code as sequential. So, when the number of vertices less then 512, it is not obvious that parallelization is superior to sequential. We can predict the cost time of MPI and OpenMP to be significantly better than sequential for a higher number of vertices.

Another result is that the best execution time for MPI is slightly better than OpenMP as number of vertices increases. In a shared-address-space system, whenever one processor needs to read data that another processor has written, its cache must be updated. When multiple processors read and write data on the same cache line, the cache needs to be updated continuously, this means that the cache is never effective as it must be constantly updated. This can have a big impact on the performance of algorithms on systems with a shared-address-space. In contrast, distributed storage systems that use message passing have a separate cache for each processor which is not invalidated or updated directly by other processors. Therefore, cache coherence is not such an issue on message passing systems.

The following table shows the computation time of OpenMP and MPI when the number of vertices is 1024 and 2048.

Table 3. The results for OpenMP and MPI implementation with different threads/processors.

number of threads / processors	OpenMP 1024 vertices	MPI 1024 vertices	OpenMP 2048 vertices	MPI 2048 vertices
2	0.2493	0.1316	0.8583	0.5040
4	0.2488	0.1281	0.7933	0.4723
8	0.2807	0.1324	0.9432	1.3663
16	0.6794	0.2055	1.7490	3.3525
32	1.3014	3.9622	2.8708	21.4067
64	1.7558	7.1446	3.5983	41.7286
128	2.8929	17.9661	5.9080	119.9792

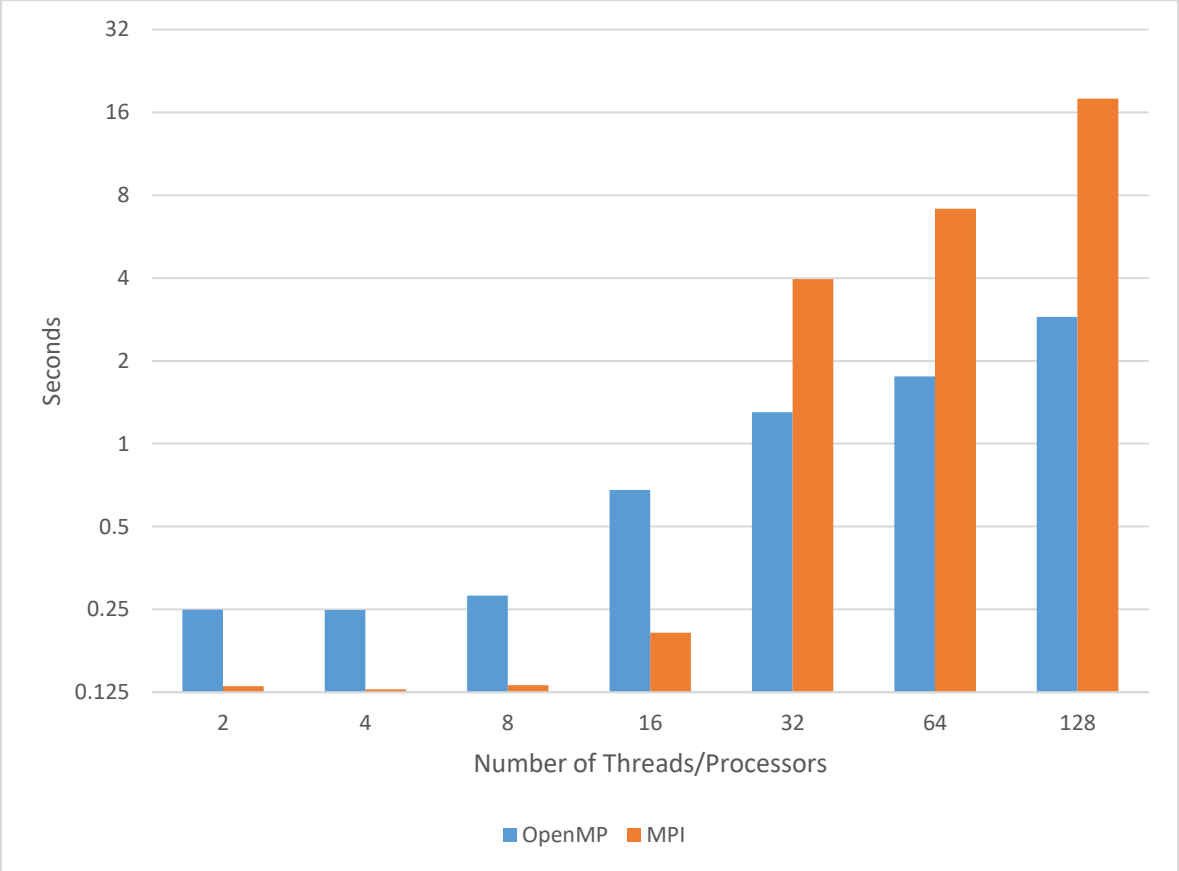

Figure 6. Different threads/processors for OpenMP and MPI implementation in 1024 graph vertices.

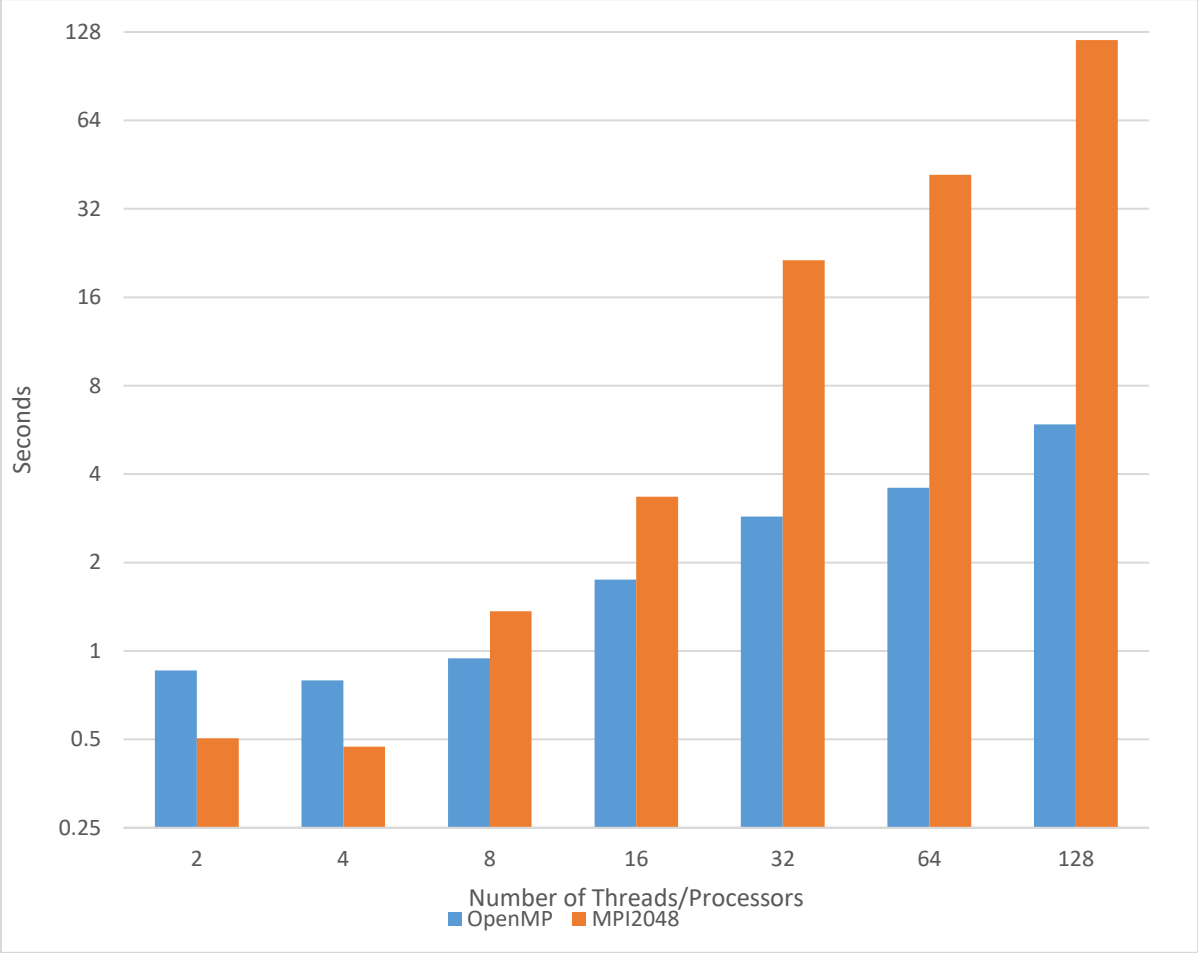

Figure 7. Different threads/processors for OpenMP and MPI implementation in 2048 graph vertices.

Figure 6 and Figure 7 show the performance of Dijkstra's algorithm on different processor/thread configurations. According to Table 1, we know the performance for OpenMP and MPI implementation are better than the sequential ones for Dijkstra's algorithm. However, through the Figure 6 and Figure 7, we observe the smaller number of processors/threads give the fastest result. For instance, if we use 1024 vertices, the best number of processors for MPI implementation is 2, 4, 8; for OpenMP, the number is the same. When the number of vertices is 2048, we have reached a similar conclusion. This is likely because each added process/thread in code causes extra communication costs in updating them. As the number of processors/threads

increase obviously, these communication costs become significantly impact. Especially for MPI Dijkstra's algorithm, it's very poor compared to the OpenMP one, and increasing the number of processors causes the slowdown to worsen. The parallel performance is likely very poor because it is dominated by the communication time, the time taken to do the `MPI_Allreduce` each iteration.

According to parallel formulation in Dijkstra's algorithm described in Section 2.3, we can compute the speedup in each condition. The following table is a comparison for theoretical speedup and experiment speedup in MPI implementation with 1024 vertices graph input.

Table 4. The comparison for theoretical speedup and experiment speedup for MPI implementation with 1024 graph vertices and different number of processors.

number of processors	Theoretical speedup 1024 vertices	Experiment speedup 1024 vertices
2	1.9961	2.7112
4	3.9690	2.7853
8	7.8168	2.6947
16	15.0588	1.7363
32	27.6757	0.0901
64	46.5455	0.0499
128	68.2667	0.0199

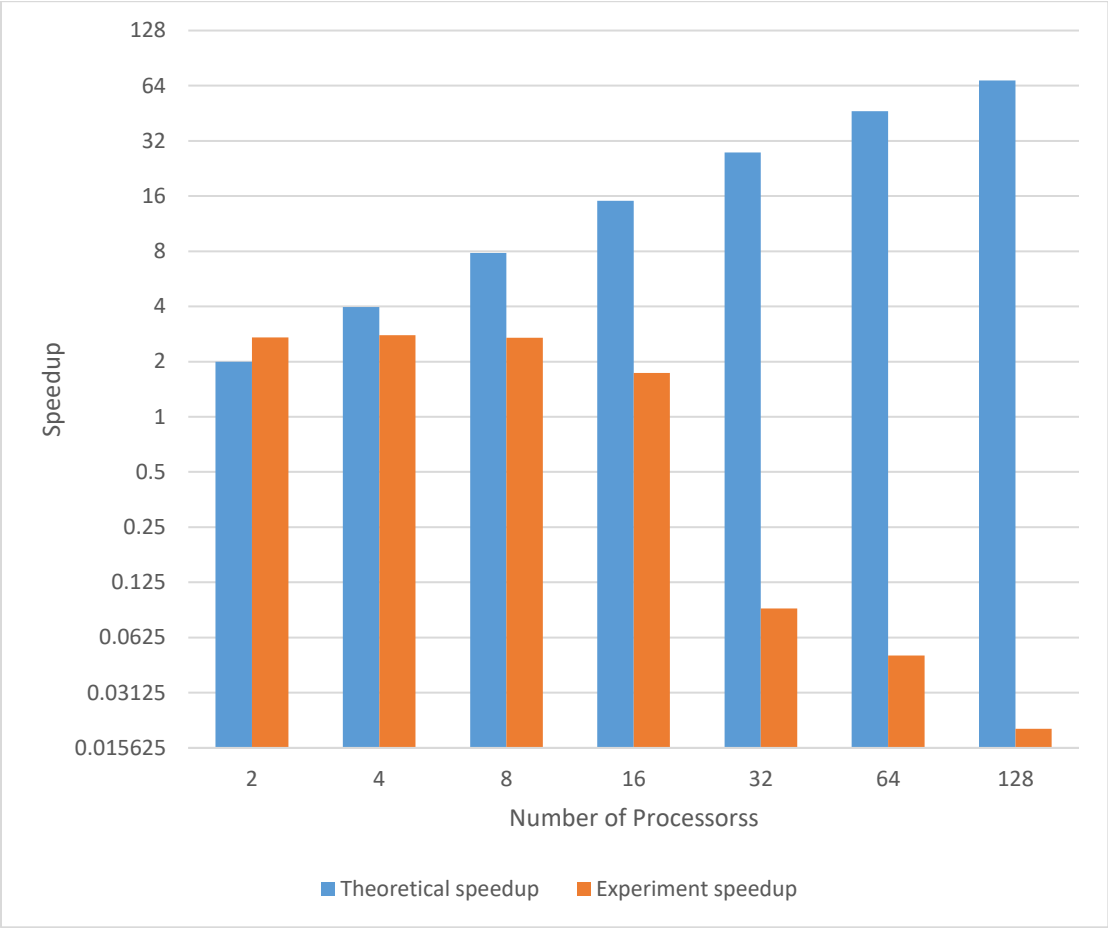

Figure 8. The comparison for theoretical speedup and experiment speedup for MPI implementation with 1024 graph vertices in different processors.

Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem using the best sequential algorithm to the time required to solve the same problem on a parallel computer with *p* identical processing elements. If speedup can maintain a linear growth with processors, multiple machines can well shorten the required time. However, this speedup is very difficult to achieve, because when the machine increases, there is a problem of communication loss, as well as the problem of each computer node itself (the skew of the slaves). For example, the total time spent by the algorithm

is usually determined by the slowest machine. If the time required by each computer is different, there is the problem of the skew of the slaves.

There may be many reasons why the high level of parallel execution of Dijkstra's has not reached the expected speedup. One reason may be that the code used is inefficient.

In the experiment, the speedup decreases also maybe because the communication latency outperforms the benefit from using more processors. We should consider all the information needed to evaluate the performance of parallel algorithm on a specific architecture with specific technology dependent constants, like CPU speed, communication speed.

Chapter 5: Conclusions and Further Work

We introduced, designed and implemented parallel Dijkstra's algorithm in this paper. The results found allow the following conclusions to be drawn:

- The performance of Dijkstra's algorithm is better when using OpenMP and MPI implementation than using sequential implementation. Especially for handling large input data sets.
- For parallelization Dijkstra's algorithm, the best execution time for MPI is slightly better than OpenMP as number of vertices increases.
- For both OpenMP and MPI implementations, the smaller number of processors/ threads give the fastest result.
- Compared theoretical speedup and experiment speedup in MPI implementation with 1024 vertices input. The experiment speedup is not a linear growth with processors increasing. Compared with seral execution, the parallel execution of Dijkstra's algorithm does not have a good performance in terms of speedup.

The following would be useful topics for further research:

- Optimization of the algorithm implementations. Fully optimized implementations, particularly the use of a priority queue for replacing the array is an area that allows for much further work. A priority queue is that each element additionally has a "priority" associated with it. For a min-priority queue, the minimum element has highest priority and it will be served before an element with low priority. A min-priority queue provides 3 basic operations: `add_with_priority()`, `decrease_priority()` and `extract_min()`. Suppose $|V|$ is the number of vertices and $|E|$ is the number of edges in a graph. If Dijkstra's algorithm uses an array

to scan all the vertices directly, it costs time $O(|V|^2)$. For sparse graphs, if the number of edges is smaller than number of vertices, we can implement the input graph by adjacency list instead of adjacency matrix and use the binary heap or Fibonacci heap as a priority queue for optimization.

The process is below:

(1) Add the source vertex to the heap and adjust the heap;

(2) Select the top element *u* and delete it from heap;

(3) Deal with the vertices that are adjacent to *u*: if the vertex is in the queue, update the distance and adjust the position of the element in the heap; if the vertex is not in the heap, add it to the heap and update the heap;

(4) If the obtained *u* is the end point, end this algorithm; otherwise, repeat steps 2 and 3.

The complexity of using a binary heap requires $O((|E| + |V|) \log |V|)$. The Fibonacci heap improves this to $O(|E| + |V| \log |V|)$. [5]

- Doing experimental runs with different system environment.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[2] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Addison Wesley, 2003.

[3] C. Wong. "Parallel-Dijkstra's-Algorithm," November 2015, https://github.com/courtniwong/Parallel-Dijkstras-Algorithm [Accessed March 2019].

[4] "Dijkstra's algorithm," https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm [Accessed March 2019].

[5] Minnesota Supercomputing Institute, https://www.msi.umn.edu/ [Accessed October 2020].

Appendix

Selected Code

A.1 Serial Dijkstra's Implementation

```
/* seqDijk.c

* Test program that does sequential Dijkstra's Algorithm.

*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/time.h>


/* Number of vertices in the graph. */

#define N 2048

/* Define the source vertex. */

#define SOURCE 1

#define MAXINT 9999999


/* Function that implements Dijkstra's single source shortest path algorithm

   for a graph represented by adjacency matrix, and use source vertex as input. */

void dijkstra (int graph[N][N], int source);


int main (int argc, char *argv[]) {
```

```
int weight[N][N];
int i, j;
char fn[255];
FILE *fp;
double time_start, time_end;
struct timeval tv;
struct timezone tz;
gettimeofday(&tv, &tz);
time_start = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;

/* Open input file, read adjacency matrix */
strcpy(fn, "input2048.txt");
fp = fopen(fn, "r");
if ((fp = fopen(fn, "r")) == NULL) {
        printf("Can't open the input file: %s\n\n", fn);
        exit(1);
}
//printf("\nThe adjacency matrix: \n");
for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
                fscanf(fp, "%d", &weight[i][j]);
                //if (weight[i][j] == 9999999) printf("%4s", "INT");
                //else printf("%4d", weight[i][j]);
```

```
            }
            //printf("\n");
        }
        dijkstra(weight, SOURCE);
        printf("\n");
        printf("Nodes: %d ", N);
        gettimeofday(&tv, &tz);
        time_end = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;
        printf("time cost is %1f\n", time_end - time_start);
        printf("\n");
        return 0;
}

void dijkstra(int graph[N][N], int source) {
        /* This array holds the shortest distance from source to other vertices. */
        int distance[N];

        /* This value sets to 1 if vertices are finished to compute. */
        int visited[N];
        int i, j, count, nextNode, minDistance;

        /* Initialize all vertices' distance and status. */
        for (i = 0; i < N; i++) {
```

```
            distance[i] = graph[source][i];
            visited[i] = 0;
}
visited[source] = 1;
count = 1;

/* Find shortest path for all vertices. */
while (count < N) {
            minDistance = MAXINT;

            /* Pick the minimum distance vertex from the set of vertices that
              is not processed. */
            for (i = 0; i < N; i++) {
                        if (distance[i] < minDistance && !visited[i]) {
                                    minDistance = distance[i];
                                    nextNode = i;
                        }
            }
            /* Mark this vertex is true. That means the vertex is processed. */
            visited[nextNode] = 1;
            count++;
            /* Update the dist value of the picked vertex. */
            for (i = 0; i < N; i++) {
```

```
                        if (!visited[i] && minDistance + graph[nextNode][i] < distance[i]) {
                                distance[i] = minDistance + graph[nextNode][i];
                        }
                }
        }

        /* Print the distance values. */
        //printf("\nThe distance vector is\n");
        //for (i = 0; i < N; i++) {
        //        printf("%d ", distance[i]);
        //}
        //printf("\n");
}
```

A.2 Message Passing Dijkstra's Implementation

```
/* MPIdijk.c
 * The program that does MPI Dijkstra's Algorithm.
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
```

```
#include "mpi.h"


#define N 2048
#define SOURCE 1
#define MAXINT 9999999
/*single source Dijkstra's Algorithm*/
/*@param n: number of vertices;
  @param source: rank of the root
  @param wgt: points to locally stored portion of the weight adjacency matrix of the graph;
  @param lengths: points to a vector that will store the distance of the shortest paths from the
source to the locally stored vertices;
 */
void SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm comm) {
        int temp[N];
        int i, j;
        int nlocal; /* The number of vertices stored locally */
        int *marker; /* Used to mark the vertices belonging to Vo */
        int firstvtx; /* The index number of the first vertex that is stored locally */
        int lastvtx; /* The index number of the last vertex that is stored locally */
        int u, udist;
        int lminpair[2], gminpair[2];
        int npes, myrank;
        MPI_Status status;
```

```
MPI_Comm_size(comm, &npes);
MPI_Comm_rank(comm, &myrank);
nlocal = n / npes;
firstvtx = myrank*nlocal;
lastvtx = firstvtx + nlocal - 1;

/* Set the initial distances from source to all the other vertices */
for (j = 0; j<nlocal; j++) {
        lengths[j] = wgt[source*nlocal + j];
}
/* This array is used to indicate if the shortest part to a vertex has been found or not. */
/* if marker [v] is one, then the shortest path to v has been found, otherwise it has not. */
marker = (int *)malloc(nlocal*sizeof(int));
for (j = 0; j<nlocal; j++) {
        marker[j] = 1;
}
/* The process that stores the source vertex, marks it as being seen */
if (source >= firstvtx && source <= lastvtx) {
        marker[source - firstvtx] = 0;
}
/* The main loop of Dijkstra's algorithm */
for (i = 1; i<n; i++) {
        /* Step 1: Find the local vertex that is at the smallest distance from source */
```

```
lminpair[0] = MAXINT; /* set it to an architecture dependent large number */
lminpair[1] = -1;
for (j = 0; j<nlocal; j++) {
        if (marker[j] && lengths[j] < lminpair[0]) {
                lminpair[0] = lengths[j];
                lminpair[1] = firstvtx + j;
        }
}
/* Step 2: Compute the global minimum vertex, and insert it into Vc */
MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC, comm);
udist = gminpair[0];
u = gminpair[1];

/* The process that stores the minimum vertex, marks it as being seen */
if (u == lminpair[1]) {
        marker[u - firstvtx] = 0;
}
/* Step 3: Update the distances given that u got inserted */
for (j = 0; j<nlocal; j++) {
        if (marker[j] && ((udist + wgt[u*nlocal + j]) < lengths[j])) {
                lengths[j] = udist + wgt[u*nlocal + j];
        }
}
```

```
        }
        free(marker);
}

int main(int argc, char *argv[]) {
    int npes, myrank, nlocal;
    int weight[N][N]; /*adjacency matrix*/
    int distance[N]; /*distance vector*/
    int *localWeight; /*local weight array*/
    int *localDistance; /*local distance vector*/
    int sendbuf[N*N]; /*local weight to distribute*/
    int i, j, k;
    char fn[255];
    FILE *fp;
    double time_start, time_end;
    struct timeval tv;
    struct timezone tz;

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;

    /* Initialize MPI and get system information */
    MPI_Init(&argc, &argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

nlocal = N/npes; /* Compute the number of elements to be stored locally. */

/*allocate local weight and local disatance arrays for each prosess*/
localWeight = (int *)malloc(nlocal*N*sizeof(int));
localDistance = (int *)malloc(nlocal*sizeof(int));

/* Open input file, read adjacency matrix and prepare for sendbuf */
if (myrank == SOURCE) {
        strcpy(fn,"input2048.txt");
        fp = fopen(fn,"r");
        if ((fp = fopen(fn,"r")) == NULL) {
                printf("Can't open the input file: %s\n\n", fn);
                exit(1);
        }
       //printf("\nThe adjacency matrix: \n");
                for(i = 0; i < N; i++) {
                        for(j = 0; j < N; j++) {
                                fscanf(fp,"%d", &weight[i][j]);
                                // if (weight[i][j] == 9999999) printf("%4s", "INT");
                                // else printf("%4d", weight[i][j]);
```

```
                }
                // printf("\n");
            }
            /*prepare send data  */
            for(k=0; k<npes; ++k) {
                for(i=0; i<N;++i) {
                    for(j=0; j<nlocal;++j) {
                        sendbuf[k*N*nlocal+i*nlocal+j]=weight[i][k*nlocal+j];
                    }
                }
            }
    }
    /*distribute data*/
    MPI_Scatter(sendbuf, nlocal*N, MPI_INT, localWeight, nlocal*N, MPI_INT, SOURCE,
MPI_COMM_WORLD);

    /*Implement the single source dijkstra's algorithm*/
    SingleSource(N, SOURCE, localWeight, localDistance, MPI_COMM_WORLD);

    /*collect local distance vector at the source process*/
    MPI_Gather(localDistance, nlocal, MPI_INT, distance, nlocal, MPI_INT, SOURCE,
MPI_COMM_WORLD);
```

```
    if (myrank == SOURCE) {
            printf("Nodes: %d\n", N);
                    //printf("The distance vector is \n");
               //for (i = 0; i < N; ++i) {
            //                  printf("%d ", distance[i]);
            //          }
              // printf("\n");
                    gettimeofday(&tv, &tz);
                    time_end = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;
                    printf("time cost is %1f\n", time_end - time_start);
    }
    free(localWeight);
    free(localDistance);
    MPI_Finalize();
    return 0;
}
```

A.3 Shared-Address Dijkstra's Implementation

```
/* openMPdijk.c
 * The program that does OpenMP parallel Dijkstra's Algorithm.
 */
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <sys/time.h>
#include <omp.h>

#define N 2048
#define SOURCE 1
#define MAXINT 9999999

void dijkstra ( int graph[N][N], int source );

/* This program runs single source Dijkstra's algorithm. Given the distance
   matrix that defines a graph, we seek a minimum distance array between
   source vertex and all other vertices.  */
int main(int argc, char **argv) {
        int i, j;
        char fn[255];
        FILE *fp;
        int graph[N][N];
        int threads;
        printf("Please enter number of threads: ");
        scanf("%d", &threads);
        omp_set_num_threads(threads);
        double time_start, time_end;
```

```
struct timeval tv;
struct timezone tz;
gettimeofday(&tv, &tz);
time_start = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;

strcpy(fn, "input2048.txt");
fp = fopen(fn, "r");
if ((fp = fopen(fn, "r")) == NULL) {
        printf("Can't open the input file: %s\n\n", fn);
        exit(1);
}
//printf("\nThe adjacency matrix: \n");
for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
                fscanf(fp, "%d", &graph[i][j]);
                //if (graph[i][j] == 9999999) printf("%4s", "INT");
                //else printf("%4d", graph[i][j]);
        }
        //printf("\n");
}
dijkstra(graph, SOURCE);

gettimeofday(&tv, &tz);
```

```
	time_end = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;
	printf("Nodes: %d\n", N);
	printf("time cost is %1f\n", time_end - time_start);
	return 0;
}

void dijkstra(int graph[N][N], int source){
	int visited[N];
	int i;
	int md;
	int distance[N]; /* This array holds the shortest distance from source to other vertices. */
	int mv;
	int my_first; /* The first vertex that stores in one thread locally. */
	int my_id; /* ID for threads */
	int my_last; /* The last vertex that stores in one thread locally. */
	int my_md; /* local minimum distance */
	int my_mv; /* local minimum vertex */
	int my_step; /* local vertex that is at the minimum distance from the source */
	int nth; /* number of threads */

	/* Initialize all vertices' distance and status. */
	for (i = 0; i < N; i++) {
		visited[i] = 0;
```

```
        distance[i] = graph[source][i];
}
visited[source] = 1;

/* OpenMP parallelization starts here */
# pragma omp parallel private ( my_first, my_id, my_last, my_md, my_mv, my_step ) \
shared ( visited, md, distance, mv, nth, graph )
{
        my_id = omp_get_thread_num ( );
        nth = omp_get_num_threads ( );
        my_first =   (my_id * N ) / nth;
        my_last  =   ((my_id + 1) * N) / nth - 1;
        //fprintf(stdout, "P%d: First=%d   Last=%d\n", my_id, my_first, my_last);
        for (my_step = 1; my_step < N; my_step++) {
                # pragma omp single
                {
                        md = MAXINT;
                        mv = -1;
                }
                int k;
                my_md = MAXINT;
                my_mv = -1;
```

```
        /* Each thread finds the minimum distance unconnected vertex inner of
        the graph */
        for (k = my_first; k <= my_last; k++) {
                if (!visited[k] && distance[k] < my_md) {
                        my_md = distance[k];
                        my_mv = k;
                }
        }
        /* 'critical' specifies that code is only be executed on one thread at a time,
          * because we need to determine the minimum of all the my_md here. */
        # pragma omp critical
        {
                if (my_md < md) {
                        md = my_md;
                        mv = my_mv;
                }
        }
/* 'barrier' identifies a synchronization point at which threads in a parallel
 * region will wait until all other threads in this section reach the same point. So
* that md and mv have the correct value. */
        # pragma omp barrier

        # pragma omp single
```

```
                {
                        /* It means we find the vertex and set its status to true. */
                        if (mv != - 1){
                                visited[mv] = 1;
                        }
                }
                # pragma omp barrier

                if ( mv != -1 ){
                        int j;
                        for (j = my_first; j <= my_last; j++) {
                                if (!visited[j] && graph[mv][j] < MAXINT &&
distance[mv] + graph[mv][j] < distance[j]) {
                                        distance[j] = distance[mv] + graph[mv][j];
                                }
                        }
                }
                #pragma omp barrier
        }
    }
    /*
    printf("\nThe distance vector is\n");
    for (i = 0; i < N; i++) {
```

```
            printf("%d ", distance[i]);
        }
        printf("\n");
        */
}
```