Honglu Xu
EECS 438
Final Project

Analyze Dijkstra's shortest path algorithm parallelization with HPC clusters

1. Background.

Dijkstra's shortest path algorithm is a widely known algorithm that can efficiently compute the shortest path in a graph. The graph can both directed and undirected, and since this algorithm is correctness guaranteed and efficient, there are a lot of implementations for this algorithm now.

By Ian[1], the Dijkstra's shortest path algorithm is implemented like this, showing in Fig. 1.

```
procedure sequential_dijkstra
begin
  d_s = 0
  d_i = ∞, for i ≠ s
  T = V
  for i = 0 to N-1
    find v_m ∈ T with minimum d_m
    for each edge (v_m, v_t) with v_t ∈ T
      if (d_t > d_m + length((v_m, v_t))) then d_t = d_m + length((v_m, v_t))
    endfor
    T = T − v_m
  endfor
end
```

Fig. 1; Dijkstra procedures

The run time complexity for this algorithm should be O(n^2), which n is the vertices. There are also many other implementations, but for parallel implement and comparison, I will use this basic algorithm to start.

2. Objective.

The main objective for this project to trying to compare the Dijkstra's shortest path algorithm with different parallelization method, like multi-processing with fork() and multi-threading with OpenMP, to see which method fits the algorithm better. The main properties that I will compare for these methods are: run-time, which will represent the code efficiency; speed-up amount, which will represent whether it is the overhead times that slowing down the algorithm. For the run-time comparison, I will use different node sizes with five trails to avoid exceptions. For the speed-up comparison, I will use different process/thread quantities and also with five trails to avoid exceptions.

There actually some parallel implementations with very good code efficiencies on the internet. However, most of the implementations are not genetic. It will be very hard to change one’s code from one method to another. Thus, the first objective for this project is to implement a good serialized Dijkstra's shortest path code that could be easily changed to the parallel version without changing the structure very much.

3. General Methods.

In this project, I mainly used 3 methods for comparison, multi-processing with fork(), multi-threading with OpenMP parallel and multi-threading with OpenMP with parallel for. For all three paralleled methods, I will use same process/thread quantities for the variable control. The process/thread quantities will be 1, 2, 4, 8, 16, which will all be running at a single CPU node with 16 cores applied.

We also need to use the run-times to compare the general code efficiencies. For that I have used 5 different node sizes, 200, 250, 300, 350, 400, which are good enough to present the run-times. All the fastest run-time, which should be generated by 16 processes/threads, is over one seconds.

4. Code Preview.

Here in Table. 3, I listed out all the code files that is needed for this project, as long as the file descriptions.

File Name	Descroptions
graph_construct.c	C file to construct the graphs
dijkstra_serial.c	C file with serial version
run_dij_serial.slurm	SLURM files to run the sertial version
run_dij_serial_out.txt	Output file of serial version
dijkstra_openmp.c	C file of OpenMP parallel for version
run_dij_openmp.slurm	SLURM file to run the OpenMP parallel for version
run_dij_openmp_out.txt	Output file of OpenMP parallel for version
dijkstra_openmp_para.c	C file of OpenMP parallel version
run_dij_openmp_para.slurm	SLURM file to run the OpenMP parallel version
run_dij_openmp_para_out.txt	Output file of OpenMP parallel version
dijkstra_fork2.c	C file of fork() version
run_dij_fork.slurm	SLURM file to run the fork() version
run_dij_fork_out.txt	Output file of fork() version

Table. 3; File names and descriptions

We see all the files are listed out above. The project mainly can be separated to three parts: serial version, OpenMP parallel for version, and OpenMP parallel version. For each part, the ".c" files are the main code file, the "run***.slurm" files are SLURM files that will run a specific trails and with multiple thread/process numbers, and the "run***out.txt" files are out put files for the corresponding SLURM files. Lastly, the "graph_construct,c" is the file that can automatically generate the graph information that can directly be used by the code listed above.

To run the code, firstly, graphs is needed. the compile method is the standard "GCC" method, which can be "gcc graph_construct.c –o graph_construct". Then, to generate the graphs, specific numbers of nodes and edges will be needed. The detail will be discussed below, but as an example, you can use "graph_construct <filename> <node_number> <edge_number>", where <filename> is a string and both <node_number> and <edge_number> should be integers. For all the other SLURM files, you can use the standard "sbatch" command to submit the jobs, and all the output files will be generated automatically. Remember to use the "dos2unix" command to change the code style of the SLURM files to Unix first, since I used Windows as the OS.

For each output file, several trails will be included in the same file, so if you need to read the output file, manually separation will be needed. For each trail, the C file name will be showed in the first line, and followed up with information about this trails. Mostly, they will contain the "Size", "Thread", "Time", "Part1Time", "Part2Time". "Size" here will be the node numbers. "Thread" will be thread/process quantities, and "Time" will be the total run-time. "Part1Time", "Part2Time" will be contained in some files and they will show the run-time of some specific part, which I will explain more later.

5. Graph Constructor

To build a nice environment for the experiment, we need firstly be able to generate stable and testable data. There actually are a lot of data structures online, but since we need a C version and easy to parallelize. I decided to generate the data by myself. Basically, the program will take 2 parameters, the size and the connection number per node (n). The size is the total number of nodes we will use, and the n will be how many edges that one node can lead out. If n=5, then all the nodes except the first 5^{th} nodes and the last 5 nodes will have 5 edges to go in and 5 edges to go out, and the edges will be generated by sequence, which means node 0 will be connected to node 1, 2, 3, 4, 5.

Thus, by this algorithm, we can show an example for size=8 and n=2. Showing in Fig. 2.

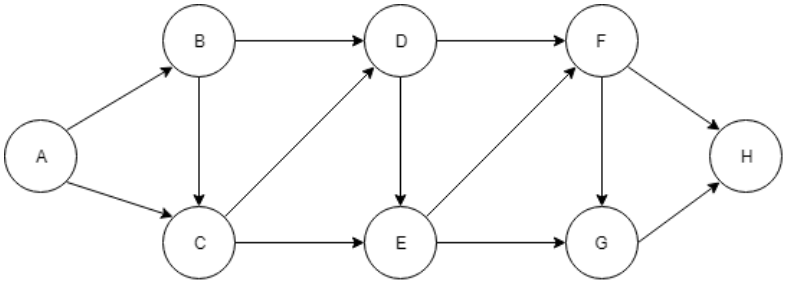

Fig. 2; Self-generated graph

By using this method, we can control the edge number that won't go too huge that the find path algorithm cannot even finish, and it is also very easy to verify the correctness for this graph even at a large size.

For simplicity, I decided to use the directed graph only, and the first node (A in the example) will be the source and the last node (H in the example) will be the destination. All the weights will be generated randomly, so one same graph can be used for several times if we just change the weights.

5. Serial Version

I also decided to implement the serial version of Dijkstra's algorithm by myself. The reason is similar as above, we need a version that is in C code, and we need it be able to parallelize so without changing the internal methods so we can compare the speed-ups with small other factors that will change the run time. Also, as noticed in the previous homework, the recursive function calls will cause extra trouble for the paralleled version, so I decided to use the non-recursive version for the code.

By manually checking the result for several small sized tests, I can confirm that the code is giving the correct results at least. Now, the other property that we need to care is the run-time.

Table. 5 will show the serial version average run-time, which is concluded by 5 trails.

dijkstra_serial.c	(Time in Seconds)	
	Size	AVE Time
	200	31.0865064
	250	69.776461
	300	147.6553118
	350	228.287893
	400	443.814327

Table. 5; Serial version AVE run-time

As showed in the Table. 5, I used 5 node sizes, from 200 to 400, and the fastest run-time, which is at 200 nodes, is 31 seconds. Thus, this should be a good testing and comparing data at start. Here we only showed the average data here, and the whole run-time data will be showed in the Appendix [1].

From the data above, we can generate a plot to show the curve, shows in Fig. 4. and the information in Fig. 6.

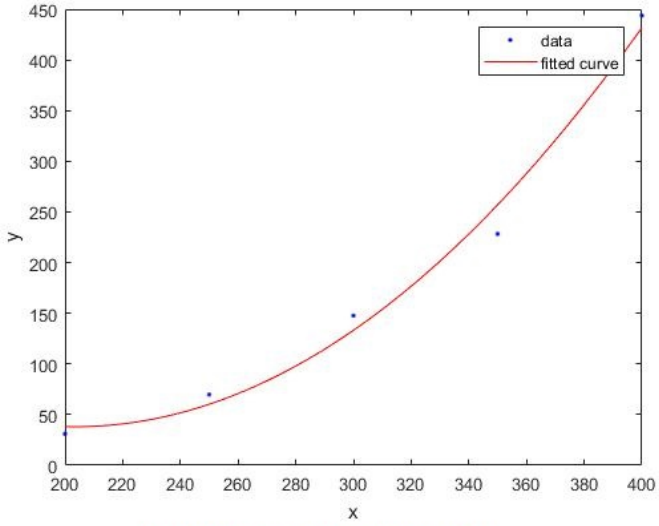

Fig. 4; data curve for serial version

```
gof1 =

  包含以下字段的 struct:

          sse:  1.3238e+03
      rsquare:  0.9877
          dfe:  2
    adjrsquare:  0.9753
         rmse:  25.7279
```

Fig. 6; Fit information for serial

This is the curve that generated by MATLAB “fit()”, it will try to fit the points to a curve as close as possible, and the Fig. 6 shows the information about the error values while fitting that figure. “rsquare” represents “R-squared (coefficient of determination)” and “rmse” represents “Root mean squared error (standard error)”. We see that the points are not strictly fitting the O(n^2) curve, but they still have some tendency to get close to the curve. It can be possible that some overhead time from the memory reading or function call made the run-time not stably increases with the size. However, it is true that I did not obey the Dijkstra's algorithm strictly, due to some reasons such as the C data structure and code efficiency, but it should be as close as possible.

In the serial version, I have three main parts that will cost the run-time. Fig. 7 will show the pseudocode for my algorithm.

```
read data from file and put into edges;

while last node not confirmed:
    search the whole graph to get all the reachable nodes from the cluster;

    for all the reachable nodes traceback to the root and calculate the total distance;
    get the min distance, record the node and put it into the cluster.
```

Fig. 7; Pseudocode for serial algorithm

The three main parts are: reading the data from the file, searching the next nodes, and tracing back and computing the min for all distance. Here, we don't care about the time cost by reading data, so I just recorded the time for last two parts, and these parts should be a main comparison for the paralleled version, too.

6. OpenMP Parallel.

The whole first trail data will be shown in Appendix [2], and the speed-ups will be shown in Appendix [3].

OpenMP parallel is a very good way to use for the parallelization. It is good with automated parallelization with multi-threading methods. To use the parallel method on my code, I need to change two main part of the code, as talked above, the searching nods part and the tracing back part. For each part, since I am using the parallel method, I need to separate the work for all the threads as even as possible. Also, for the convenient of reuse and testing, we need to make the threads quantities dynamic, which means we can pass the threads quantities as a parameter. To achieve this, I used the different start-number and end-number for each thread, so all of them will take their own part for parallelization.

Table. 8, shows the serial version of OpenMP, although we included the OpenMP tags there, we can still set the thread number to 1, so it is the serial version.

dijkstra_openmp_para.c		Thread--1		
Trail 1	Size	Time Total	Time Part1	Time Part2
	200	26.028726	19.34	6.682
	250	58.378586	48.17	10.2
	300	123.842498	93.607	30.2347
	350	191.545258	161.776	29.76
	400	372.216702	303.06	69.147

Table. 8; OpenMP parallel with 1 thread

We see that the basic time for 200 nodes is 26 seconds. Which is faster than the real serial version. This is actually strange. To achieve the parallel version, I separated the part that needed to be paralleled into two parts, the first part will do the parallel code, then, after the parallel part finished, we need to take the general data out and filter it serially, which definitely increased the overhead time. However, it shows that the OpenMP serial version is faster than the real serial version, which means the OpenMP is using some of the Optimization in the compiler. which will make some particular codes run faster.

We can plot all the sizes and compare to the real serial version, which shows in Fig. 9.

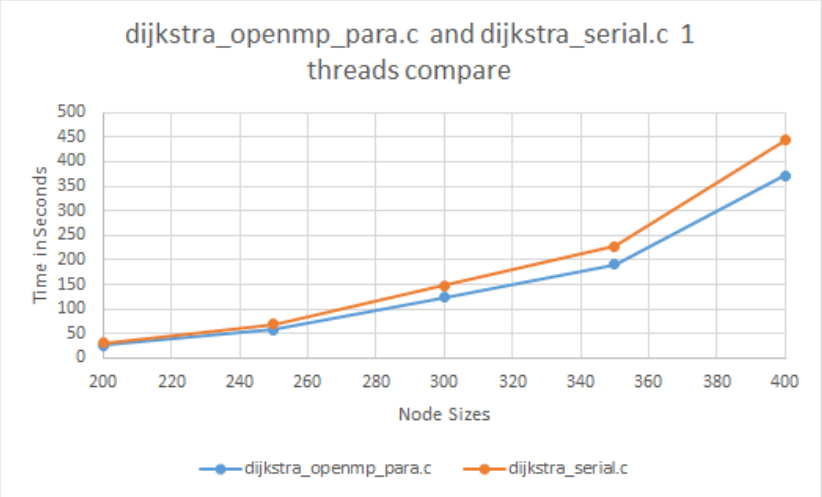

Fig. 9; Comparison of OpenMP 1 thread and serial

We see that the OpenMP is not only faster at the beginning, with the increasing of the node sizes, the difference is getting bigger, which conforms that this is not an exception situation. Also, it seems that both of the lines are fitting with the same curve, which means it should not be changing the algorithms as the reasons of this cause.

The structure of this OpenMP version is a little different compared to the serial version. There actually are many places that I added for the parallel version that will cause the overhead situation. For the first part, which is the part to search for new nodes, there's one place to check whether these node has already been searched before, if it is, then we don't need to do the rest parts. However, in a parallel situation, we cannot precisely tell whether this node has been searched, since each route is occupied by the separated threads and there is seldom communication between them. Thus, that is reason to add a new serial part after the parallel part to filter out the same nodes we reached out from different nodes, which will add some overhead time. This happens for both of the paralleled part. Also, I need to use a shared list for all the threads, but I need to keep the index changing and the list value passing in atom state, which I used the tag of "#pragma omp critical", which also blocks the threads for some time. This happened in both first and second parallel part.

We can also see whether the paralleled version is Obey the rules of O(N^2), which shows in Fig. 10, and information in Fig. 11.

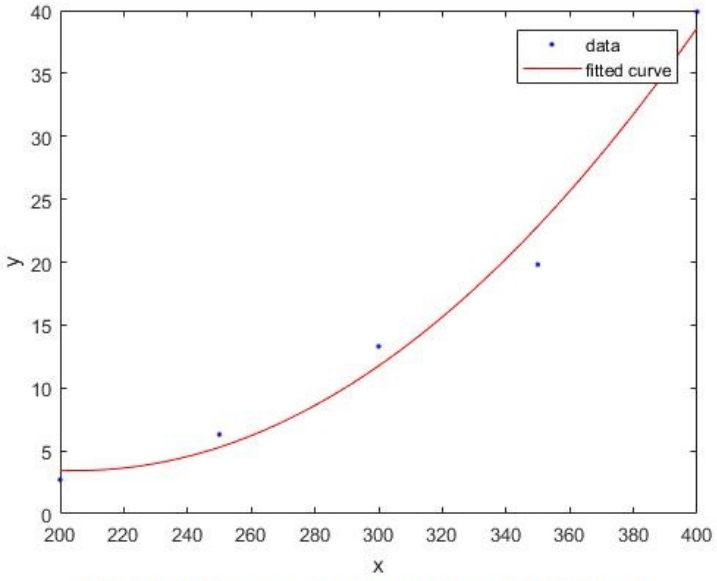

Fig. 10; Data curves for OpenMP parallel 16 threads

```
gof2 =

  包含以下字段的 struct:

           sse: 15.0326
       rsquare: 0.9826
           dfe: 2
     adjrsquare: 0.9652
          rmse: 2.7416
```

Fig. 11; Curve fit for OpenMP parallel 16 threads

From the figures above, we see that the OpenMP version with 16 threads' curve is actually very similar to the real serial version, even the data is kind of similar. However, the "rsquare" and "adjrsquare" indeed dropped some points, which means it is less coefficient than the real serial version.

The speed-ups will be shown later with OpenMP parallel for version.

7. OpenMP parallel for.

The OpenMP time data will be shown in Appendix [4] and Appendix [5]. The speed-up will be shown in Appendix [6].

Since all the paralleled parts I used is in a double for loop, it will be a nice try to use the parallel for tag from the OpenMP to let it parallel automatically.

For this section, there aren't many structures to change, so I will focus on the analyzation part.

Since the time for OpenMP parallel for and OpenMP parallel are similar for the small quantities of threads, I will mostly use the large number of threads such as 16 threads to analyze the parallel situation.

Table. 12 will show the OpenMP parallel for version times with 16 threads, and we can also compare the parallel version with this parallel for version. (Time in seconds)

Thread--16	Size	AVE Time Total	AVE Time Part1	AVE Time Part2
dijkstra_openmp.c	200	2.681983333	1.803	0.872333333
	250	6.330472	4.813333333	1.507
	300	13.27760567	9.388666667	3.869166667
	350	19.781957	15.69233333	4.066
	400	39.88456967	31.11666667	8.762
Thread-- 16	Size	Time Total	Time Part1	Time Part2
dijkstra_openm_para.c	200	5.62	3.3	2.32
	250	10.36	6.44	3.922
	300	18.612	11.84	6.76
	350	28.574	19.64	8.93
	400	50.657	34.09	16.56

Table. 12; OpenMP parallel version and parallel for version times

From the figure, we see that in 16 threads, the automated parallel for version is indeed faster than the parallel version. in 400 node size, the time difference is approximately 11 seconds, which is really a large time difference based on the run-time as 39.88 seconds.

Let's plot out the times so we can have a general idea about how it goes. Shows in Fig. 13.

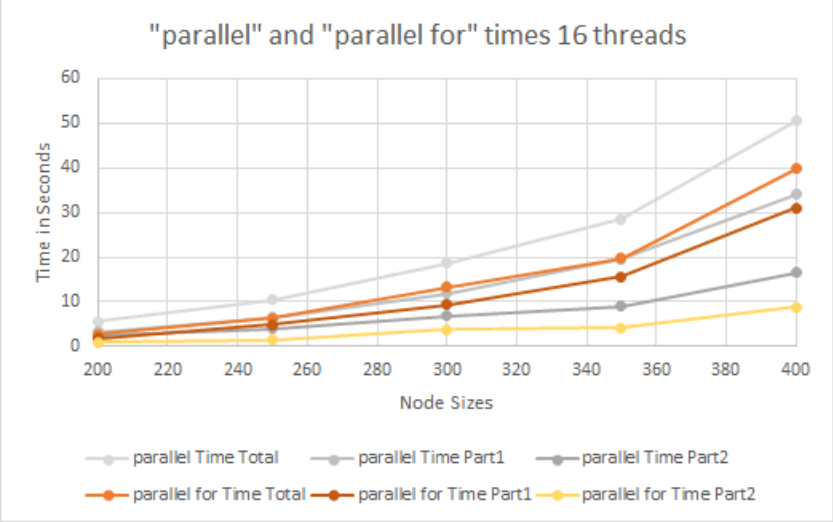

Fig. 13; Times for parallel and parallel for with 16 threads

In this figure, I used different gray color for all parallel version and other colors for parallel for version, so we can distinguish them clearly. We see that for all the general data, parallel for is faster, no matter it is in the first parallel part or the second. It is also interesting that for some point the parallel for version's total time is smaller than the parallel version's part1 time, which means the parallel for is really fast and efficient.

Now let's see the speed-ups. The speed-ups for 16 threads of parallel for version will be shown in Table. 14. (Compared with the 1 thread version OpenMP parallel for)

Thread--16	Size	Total Speedup	Part1 Speedup	Part2 Speedup
	200	9.733124093	10.76068819	7.683262285
	250	9.2480664	10.03550256	6.795031984
	300	9.346599087	9.989292147	7.834742882
	350	9.693275554	10.31949197	7.332222922
	400	9.345309114	9.752437774	7.905682995

Fig. 14; Speed-ups for parallel for 16 threads

We see that generally speed-ups for 16 threads is approximately 9.5, though the speed-ups are not stable for different node sizes, but it still seems that the speed-ups are not affected by the sizes. 9.5 is not quite a very good speedup as 16 threads, but it is also not a very bad one. We see some paralleled version from previous works have nearly 12 for the speed-ups, and some for version, the speed-ups just did not pass 7. Thus, 9.5 is sort of in the average.

We can also plot out all the speed-ups we got for now, and see the general concept. Shows in Fig. 15. All showed for 400 node sizes.

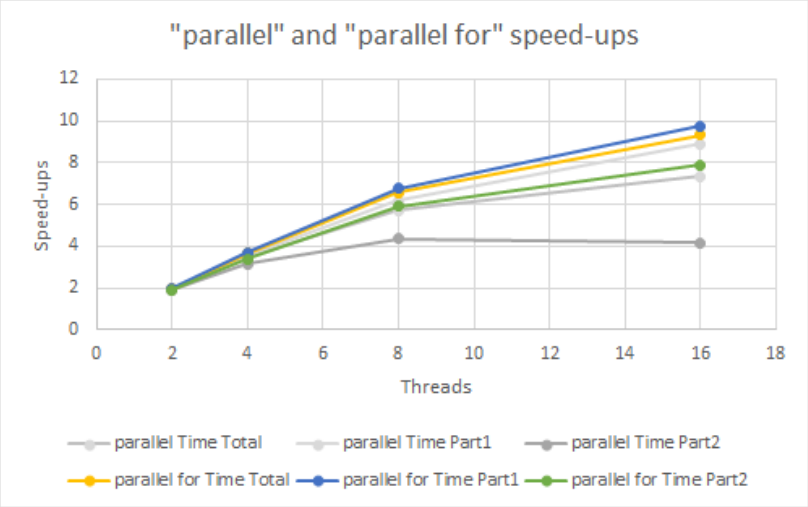

Fig. 15; Speed-ups for parallel and parallel for

In the Fig.15, we now have a good concept of all the speed-ups for both parallel version and parallel for version. We see that, generally the speed-ups of parallel for is better than the parallel version, and that might be the reason that parallel for is faster in run-time. We also noticed that, the first part of parallel 's line is following the total closely, but the second part is actually not acting very good. For the parallel version, it even dropped under the last speed-up at 16 threads. Thus, it should be the second part, which is actually slowing down the program.

There three parts might cause this behavior, after inspecting the code. The first is the main for loop for this part is not divided evenly, so the work is not separated evenly. For all the graphs I tested, the edges per node is being set to 10, which means no matter how large the graph is, there only 10 edges connected in and 10 out for each nodes maximum. Thus, for some particular situation, we may not find more than 16 nodes is reachable, which gives that the second part will only use less than 16 threads for calculation. The second is, we have a critical area that will visit the C structure list for every loop, which will block the threads for a long time totally. Lastly if we do find many possible route at this time, we will also waste a lot time on the min calculation which will be calculate in serial.

8. Multi-processing with fork().
The whole time data of fork will be shown in Appendix [7] and Appendix [8], and the speed-ups will be shown in Appendix [9].

For the fork version, since it is not fully automated, I need to manually call the fork functions to get things paralleled, also there will be three part of shared memory that will be used in the program, which will also cause some overhead run-times. Another structure change is, since we used "#pragma omp critical" for both of the parallel parts, we do need to calculate some parts in an atom way. However, it is a little complicated to implement the atom part here with the fork version. Thus, I changed the size of the shared memory part to general size instead of the dynamic size, so it will replace the atom section. This is a way that sacrifices the space efficiency and try to get the time efficiency.

We can plot out the times, showing in Fig. 16.

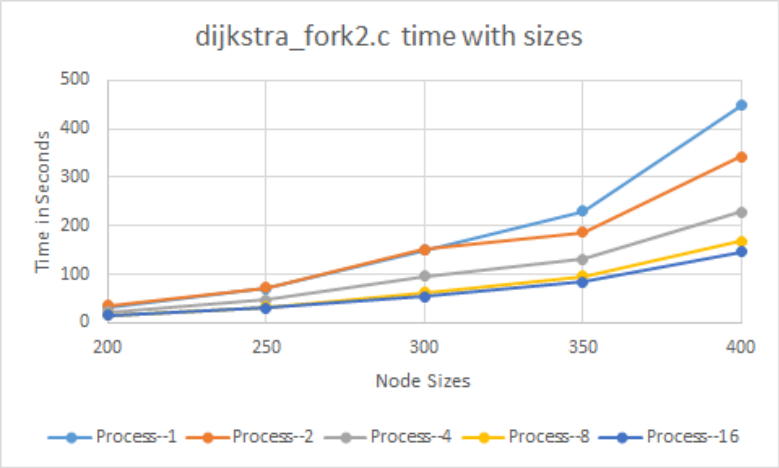

Fig. 16; Times with sizes for fork version

The plot shows all the total times with different sizes and different quantities of processes. We can see that there are very strange points for the 2 processes The first three nodes stay the same as the 1 process version, which should be slower.

This is not the correct situation, further investigation leads to the plot below, which is the Fig. 17.

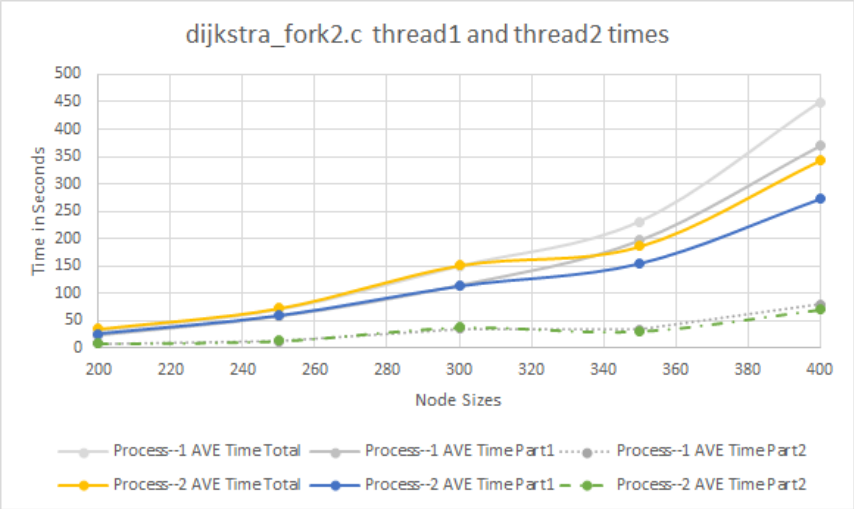

Fig. 17; Fork version times with 1 and 2 processes

Fig. 17 shows the separated parts for parallel version. We see that the 2 processes' part 2 is always stays the same as 1 process, which is not correct. However, it is not the reason that the total time is acting wrongly, we see that the total time is changing with the part1's time, and that's the reason slowing down the program, but I can't find out the specific reason for this.

And for the speed-ups, we can also plot out it, showing in Fig. 18.

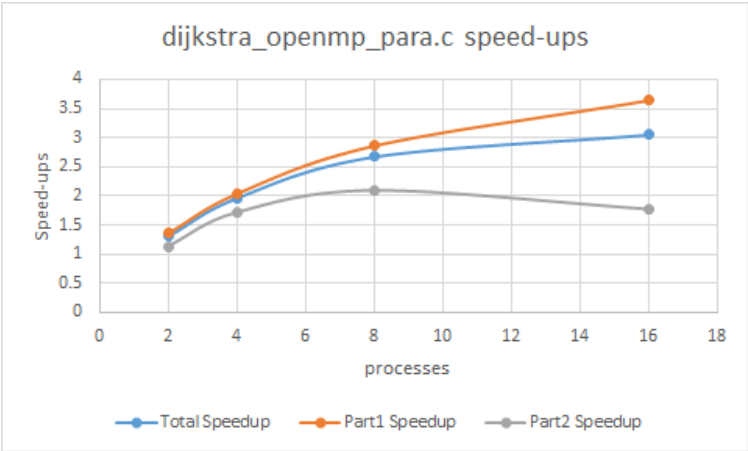

Fig. 18; Speed-ups for fork version.

The speed-ups are lower now, we see the plots max range is 4, which is much lower than 9.5. Also, the speed-ups for the part2's curve is actually similar to the OpenMP parallel version, the reason should be I am using the same way to separate the threads/processes works for both of them. However, it seems the fork version drops more points.

9. Conclusion.

To give conclusions, we could put all the related data in a single graph to see the results.

Firstly, for the run-time as serial version, shows in Fig.19.

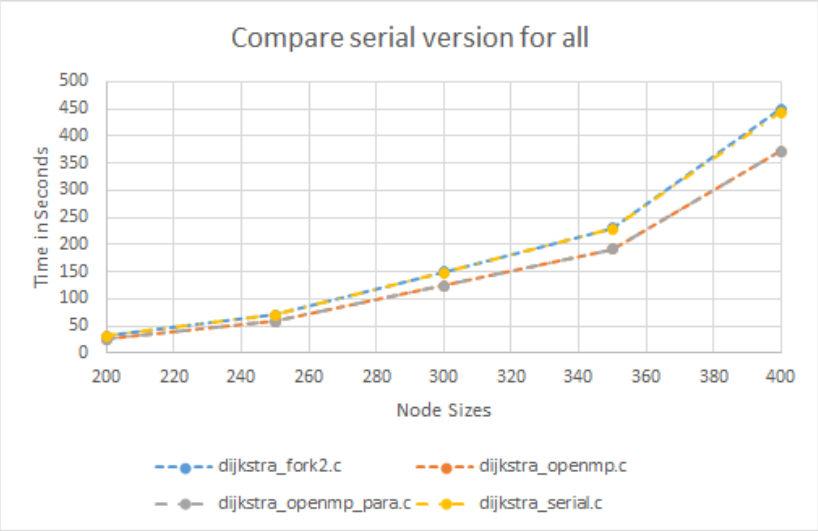

Fig. 19; Comparison all the serial versions

We see that two OpenMP versions are similar to each other, and the fork is close to the real serial version, even with the shared memory visiting. Thus, it confirms that OpenMP has some sort of tech to optimize the code efficiency, and since the fork version is slightly slower than the real serial version, the overhead time does exist.

Then, the real objective is to see the time in multi-threading /processing methods. Thus, we can compare the run-times for 16 threads/processes. Showing in Fig. 20.

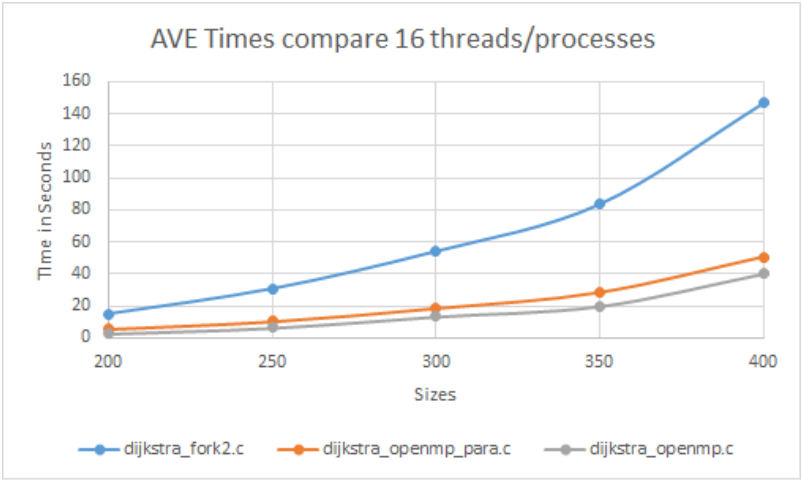

Fig. 20; Comparison for 16 threads/processes

We see that the most efficient is using the OpenMP parallel for version, and the fork version is the slowest, which indeed suit the analyzes we did above.

Then, for the speed-ups, will show in Fig. 21.

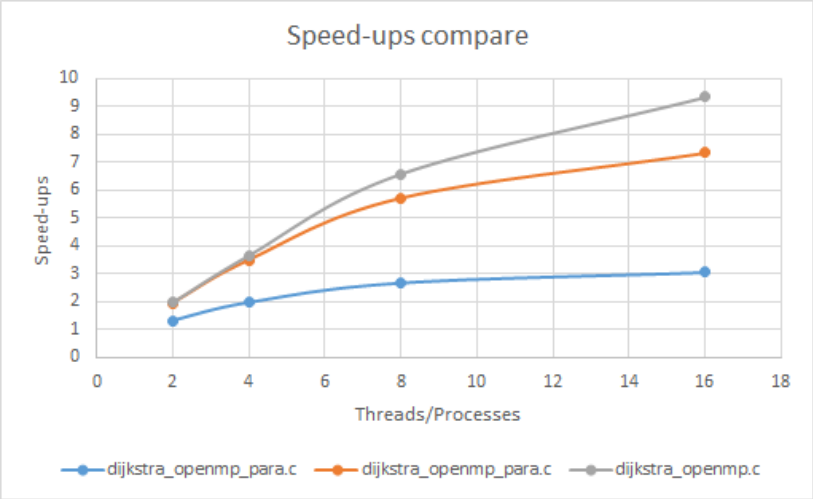

Fig. 21; Speed-ups comparison for all

As expected, the OpenMP parallel for version got the best speed-up as 9.5, and the fork version is still the lowest.

Thus, the OpenMP parallel for is the best way to fit this algorithm.

10. About OpenACC.

My initial thought is trying to implement his algorithm with OpenACC, but I struggled a very long time on this, and finally gave up. The main problem I faced is cannot implementing the atom section on the OpenACC, which is really a critical problem. Also, since I need to do the serial filter part after the parallel part, I need to transfer the data from CPU to GPU and then transfer back twice for each while loop, which did cause a lot of problems. Thus, I did not finish this part.

Reference:

[1] Ian Foster, *3.9 Case Study: Shortest-Path Algorithms*, http://www.mcs.anl.gov/~itf/dbpp/text/node35.html

Appendix:

[1] Whole run-time data for real serial version.

dijkstra_serial.c	(Time in Seconds)	
Trail 1	Size	Time Total
	200	31.063696
	250	69.745327
	300	147.576663
	350	228.276819
	400	443.726542
Trail 2	Size	Time Total
	200	31.089934
	250	69.776902
	300	147.662335
	350	228.230643
	400	443.678388
Trail 3	Size	Time Total
	200	31.095058
	250	69.803394
	300	147.658811
	350	228.303873
	400	443.793201
Trail 4	Size	Time Total
	200	31.096852
	250	69.755274
	300	147.704179
	350	228.29569
	400	443.873688
Trail 5	Size	Time Total
	200	31.086992
	250	69.801408
	300	147.674571
	350	228.33244
	400	443.999816

Appendix [1]; Whole run-time data for serial version

[2] OpenMP parallel version time with sizes.

dijkstra_openmp_para.c		Thread--1			Thread--2			Thread--4		
Trail 1	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2
	200	26.028726	19.34	6.682	13.618033	9.968	3.649592	7.755541	5.552555	2.2
	250	58.378586	48.17	10.2	30.279465	24.71	5.56	17.15	13.6	3.547
	300	123.842498	93.607	30.2347	63.672315	47.85	15.816	36.025	26.5	9.5247
	350	191.545258	161.776	29.76	98.663033	82.606	16.05	55.582	45.51	10.06
	400	372.216702	303.06	69.147	191.382	154.9	36.41	106.6	84.55	22.04
		Thread-- 8			Thread-- 16					
	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2			
	200	5.23	3.49	1.73	5.62	3.3	2.32			
	250	11.1751	8.26	2.9	10.36	6.44	3.922			
	300	23.361	16.2	7.14	18.612	11.84	6.76			
	350	34.529015	26.27	8.251	28.574	19.64	8.93			
	400	65.01	49.1231	15.89	50.657	34.09	16.56			

Appendix [2]; OpenMP parallel version time with sizes

[3] Speed-ups for OpenMP parallel version.

dijkstra_openmp_para.c				
Thread--2	Size	Total Speedup	Part1 Speedup	Part2 Speedup
	200	1.911342556	1.940208668	1.83088959
	250	1.927992651	1.949413193	1.834532374
	300	1.944997571	1.956259143	1.911652757
	350	1.941408572	1.958404958	1.854205607
	400	1.944888767	1.956488057	1.899121121
Thread--4	Size			
	200	3.356145754	3.483081212	3.037272727
	250	3.403999184	3.541911765	2.87566958
	300	3.43768211	3.532339623	3.174346699
	350	3.446174265	3.554735223	2.958250497
	400	3.491713902	3.584387936	3.137341198
Thread--8	Size			
	200	4.976811855	5.541547278	3.862427746
	250	5.223987794	5.831719128	3.517241379
	300	5.301249861	5.778209877	4.234551821
	350	5.547371044	6.158203274	3.606835535
	400	5.725529949	6.169398918	4.351604783
Thread--16	Size			
	200	4.631445907	5.860606061	2.880172414
	250	5.634998649	7.479813665	2.600713921
	300	6.653905975	7.905996622	4.472588757
	350	6.703480717	8.23706721	3.332586786
	400	7.347784156	8.889997067	4.175543478

Appendix [3]; Speed-ups for OpenMP parallel

[4] OpenMP parallel for version times.

dijkstra_openmp.c		Thread--1			Thread--2			Thread--4		
Trail 1	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2
	200	26.089398	19.392564	6.696648	13.442307	9.82942	3.612564	7.433156	5.328	2.10413
	250	58.536455	48.296647	10.239526	30.110461	24.4	5.623161	16.21518	13.04	3.1695
	300	124.090204	93.772784	30.317017	63.8942	47.59	16.30259	35.38484	25.8023	9.58
	350	191.758867	161.941957	29.81641	98.485235	82.01	16.47028	53.762826	44.146	9.6152
	400	372.682682	303.41041	69.271593	190.526017	153.912385	36.6126	101.663135	81.362	20.2991
Trail 2	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2
	200	26.091493	19.393394	6.697908	13.453893	9.8344	3.6191	7.380845	5.33	2.05
	250	58.534885	48.297688	10.236902	30.085379	24.458	5.6269	16.464	13.194	3.269
	300	124.102933	93.799259	30.303284	63.62119	47.5277	16.0928	35.256851	25.7157	9.54
	350	191.744732	161.931168	29.813064	98.483532	82.05	16.42	53.693841	44.14	9.552187
	400	372.664989	303.406316	69.25799	190.121868	153.514	36.606	102.13936	81.47	20.667
Trail 3	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2
	200	26.094798	19.395382	6.69923	13.478601	9.841	3.637	7.390357	5.31	2.07966
	250	58.551349	48.30896	10.242093	30.063989	24.4	5.60715	16.2584	13.053	3.2
	300	124.144546	93.820217	30.323937	63.689052	47.56136	16.30259	35.004079	25.66	9.343
	350	191.757572	161.939796	29.817262	98.3507	81.9422	16.4	53.73	44.13	9.59
	400	372.67907	303.415238	69.263142	189.849623	153.43	36.415	102.769452	82.25	20.51
		Thread-- 8			Thread-- 16					
Trail 1	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2			
	200	4.189716	2.960758	1.228594	2.66395	1.77	0.88			
	250	9.071157	7.195	1.87542	6.539416	4.95	1.581			
	300	19.294939	14.073351	5.22	12.983817	9.23	3.75			
	350	29.768962	24.23	5.532512	18.122181	14.538	3.582			
	400	56.472189	44.865	11.605	36.755709	28.89	7.856			
Trail 2	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2			
	200	4.295612	3.03	1.2595	2.52	1.73	0.787			
	250	9.048625	7.23	1.81	6.1	4.63	1.46			
	300	19.176033	13.983	5.192	12.89	9.137	3.7575			
	350	29.822776	24.28	5.53	20.75	16.3	4.386			
	400	56.820288	44.88063	11.938213	41.59	32.26	9.33			
Trail 3	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2			
	200	4.170909	2.959	1.2106	2.862	1.909	0.95			
	250	9.144465	7.2339	1.9	6.352	4.86	1.48			
	300	19.343743	13.98	5.358	13.959	9.799	4.1			
	350	29.639509	24.244	5.39	20.47369	16.239	4.23			
	400	56.545066	44.91	11.624	41.308	32.2	9.1			

Appendix [4]; OpenMP parallel for times

[5] OpenMP parallel AVE times.

dijkstra_openmp.c				
Thread--1	Size	AVE Time Total	AVE Time Part1	AVE Time Part2
	200	26.1040766	19.4015208	6.7023658
	250	58.5446254	48.304219	10.2401132
	300	124.100457	93.7861342	30.313926
	350	191.7519602	161.9369078	29.8128184
	400	372.7336324	303.4633554	69.2695944
Thread--2	Size			
	200	13.477498	9.851964	3.6165128
	250	30.0813658	24.4279188	5.6252002
	300	63.7501314	47.572012	16.210196
	350	98.4574454	82.00584	16.446396
	400	189.9940086	153.515277	36.47332
Thread--4	Size			
	200	7.401452667	5.322666667	2.07793
	250	16.31252667	13.09566667	3.212833333
	300	35.21525667	25.726	9.487666667
	350	53.728889	44.13866667	9.585795667
	400	102.190649	81.694	20.49203333
Thread--8	Size			
	200	4.218745667	2.983252667	1.232898
	250	9.088082333	7.219633333	1.861806667
	300	19.27157167	14.012117	5.256666667
	350	29.743749	24.25133333	5.484170667
	400	56.61251433	44.88521	11.72240433
Thread--16	Size			
	200	2.681983333	1.803	0.872333333
	250	6.330472	4.813333333	1.507
	300	13.27760567	9.388666667	3.869166667
	350	19.781957	15.69233333	4.066
	400	39.88456967	31.11666667	8.762

Appendix [5]; OpenMP parallel AVE times

[6] OpenMP parallel for version speed-ups.

dijkstra_openmp.c				
Thread--2	Size	Total Speedup	Part1 Speedup	Part2 Speedup
	200	1.936863697	1.969304882	1.853267545
	250	1.946209018	1.977418518	1.820399779
	300	1.946669823	1.97145612	1.870053021
	350	1.947561806	1.974699702	1.812726533
	400	1.961817823	1.976763234	1.899185333
Thread--4	Size			
	200	3.526885569	3.645075301	3.225501244
	250	3.588936686	3.688565098	3.187253162
	300	3.524053741	3.645577789	3.195087587
	350	3.568880053	3.66882192	3.110103682
	400	3.647433851	3.714634556	3.380318257
Thread--8	Size			
	200	6.187639328	6.50347891	5.436269505
	250	6.441911864	6.690674827	5.500094818
	300	6.439560776	6.693216607	5.766758275
	350	6.446798626	6.677443486	5.436158029
	400	6.583944147	6.760876364	5.909162697
Thread--16	Size			
	200	9.733124093	10.76068819	7.683262285
	250	9.2480664	10.03550256	6.795031984
	300	9.346599087	9.989292147	7.834742882
	350	9.693275554	10.31949197	7.332222922
	400	9.345309114	9.752437774	7.905682995

Appendix [6]; OpenMP parallel for speed-ups

[7] Fork version times.

dijkstra_fork2.c		Process--1			Proces--2			Proces--4		
Trail 1	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2
	200	31.493659	23.6002	7.86	35.13133	25.69	9.37834	21.578061	16.105	5.41
	250	69.82	58	11.68	73.904962	59.83	13.96034	45.676658	36.66	8.89
	300	149.416508	114.6	34.73	147.935004	110.47	37.28	97.48	72.741	24.538
	350	230.928785	196.59	34.216385	186.049631	154.468	31.35	126.790809	104.1	22.416
	400	449.952786	370.135	79.63	342.523601	271.88	70.279	235.778377	188.1	47.23
Trail 2	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2
	200	31.178869	23.52	7.63163	36.980238	27.52	9.389	21.776097	16.07	5.63
	250	70.01308	58.288	11.678	77.703	63.301	14.2	46.861662	37.72	9.01
	300	149.363791	114.55	34.73	158.917	119.95	38.776471	94.911385	70.74	23.96
	350	231.025734	196.65	34.25	191.502894	159.222	32.047551	132.975125	109.39	23.303782
	400	448.713141	368.959	79.573	342.74297	272.307	70.08	226.402952	179.616	46.3816
Trail 3	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2
	200	31.213539	23.56	7.629	32.8689	24.13	8.67	21.555209	16.0568	5.436502
	250	70.372095	58.21	12.106565	66.441341	53.64	12.6918	46.64395	37.58	8.941
	300	149.373648	114.572	34.72589	146.42467	108.795	37.449818	96.476033	71.83	24.44
	350	230.889607	196.542	34.231509	180.130073	149.48	30.42	131.915456	108.7	22.89
	400	448.705712	368.959	79.5655	343.732634	273.572	69.805	223.163699	177.03	45.72
		Proces-- 8			Proces-- 16					
Trail 1	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2			
	200	14.654369	10.069	4.52	15.031826	9.0303	5.938			
	250	31.556386	23.589	7.841	30.7196	19.88	10.709			
	300	61.693908	43.987	17.502	54.207896	35.4	18.59			
	350	95.497907	74.55	20.638	83.614151	58.66	24.6416			
	400	165.994148	127.888	37.65	146.8442	101.4229	44.93			
Trail 2	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2			
	200	14.306139	9.95	4.28	15.014533	9.04	5.907			
	250	31.772123	23.838	7.8086	30.818036	19.963	10.729			
	300	62.324893	44.345	17.77	54.142254	35.35	18.577			
	350	95.056063	74.2575	20.4	83.687179	58.74	24.633			
	400	168.175033	129.63	38.086	146.876271	101.6	44.77			
Trail 3	Size	Time Total	Time Part1	Time Part2	Time Total	Time Part1	Time Part2			
	200	14.743274	10.11	4.56	15.024979	9	5.951021			
	250	31.677018	23.7	7.845	30.834	20	10.707794			
	300	62.845937	44.73	17.905	54.183291	35.33	18.645			
	350	95.826028	74.9	20.61	83.697282	58.711	24.67			
	400	169.175849	130.21	38.49	147.259073	101.7	45.07			

Appendix [7]; Times for fork version with sizes

[8] AVE times for fork version.

dijkstra_fork2.c				
Process--1	Size	AVE Time Total	AVE Time Part1	AVE Time Part2
	200	31.29535567	23.56006667	7.706876667
	250	70.06839167	58.166	11.82152167
	300	149.384649	114.574	34.72863
	350	230.948042	196.594	34.23263133
	400	449.1238797	369.351	79.5895
Process--2	Size			
	200	34.99348933	25.78	9.14578
	250	72.683101	58.92366667	13.61738
	300	151.0922247	113.0716667	37.83542967
	350	185.8941993	154.39	31.272517
	400	342.999735	272.5863333	70.05466667
Process--4	Size			
	200	21.63645567	16.07726667	5.492167333
	250	46.39409	37.32	8.947
	300	96.28913933	71.77033333	24.31266667
	350	130.5604633	107.3966667	22.86992733
	400	228.4483427	181.582	46.44386667
Process--8	Size			
	200	14.56792733	10.043	4.453333333
	250	31.668509	23.709	7.831533333
	300	62.288246	44.354	17.72566667
	350	95.45999933	74.56916667	20.54933333
	400	167.7816767	129.2426667	38.07533333
Process--16	Size			
	200	15.02377933	9.023433333	5.932007
	250	30.79054533	19.94766667	10.71526467
	300	54.17781367	35.36	18.604
	350	83.666204	58.70366667	24.6482
	400	146.9931813	101.5743	44.92333333

Appendix [8]; AVE times for fork version

[9] Speed-ups for fork version.

dijkstra_openmp_para.c				
Process--2	Size	Total Speedup	Part1 Speedup	Part2 Speedup
	200	0.894319379	0.91388932	0.842670244
	250	0.964025897	0.987141556	0.868120128
	300	0.988698454	1.013286559	0.917886497
	350	1.242362822	1.273359674	1.094655455
	400	1.309400078	1.354987227	1.136105613
Process--4	Size			
	200	1.446417849	1.465427374	1.403248699
	250	1.510287014	1.558574491	1.321283298
	300	1.551417429	1.596397769	1.428417149
	350	1.76889723	1.830540985	1.496840407
	400	1.965975653	2.034072761	1.71367084
Process--8	Size			
	200	2.148236667	2.345919214	1.730586078
	250	2.212557328	2.453329959	1.509477284
	300	2.39827991	2.583171755	1.959228426
	350	2.419317448	2.636397975	1.665875519
	400	2.676835091	2.857810114	2.090316565
Process--16	Size			
	200	2.083054801	2.610986949	1.299202221
	250	2.275646336	2.915930017	1.10324122
	300	2.757303016	3.240214932	1.866729198
	350	2.760350428	3.348921987	1.388849138
	400	3.055406214	3.636264291	1.771673963

Appendix [9]; Speed-ups for fork version