CAB401 High Performance Parallel Computing Assignment – Parallelization

N9510761
Morgan Frearson

The Parallelization of Dijkstra Algorithm

Table of Contents

PLEASE READ:

For this instance of this assignment the program being parallelized will be referred to as 'Simple Dijkstra' this allows for clear separation when referencing to/from Dijkstra's algorithm itself.

Program Details

Name: Simple Dijkstra

Language: C

Development Framework: XCode Version 10.0

Description: Simple Dijkstra is an open source program written by Manuel Pasieka in C. It takes in a user desired graph size to produce a series of nodes and uses Dijkstra's Algorithm to solve the shortest path between each node. It is run inside the command line and contains two source files and one header file.

Development Machine Specifications:

Processer: Intel Core i5

Hyper-Threading: Yes

Number of Cores: 2 Physical Cores, 4 Virtual Cores

EFI Architecture: 64-Bit

Cache: L2 256KB, L3 3MB

Clock Speed: 2.8GHz

Memory: 8GB 1600 MHz DDR3

Environment: macOS V10.13.6

Running the program:

```
Morgans-MacBook-Pro:Dijkstra Sequential Version for GCC   MorganFrearson$ gcc dijkstra.c Dijkstra_tools.c -o dijkstra -w
Morgans-MacBook-Pro:Dijkstra Sequential Version for GCC   MorganFrearson$ ./dijkstra 10 2
Generating Graph of size 10x10 with init [2]!
Running dijkstra on graph
It[0] aN [0]
It[1] aN [7]
It[2] aN [2]
It[3] aN [8]
It[4] aN [3]
It[5] aN [6]
It[6] aN [9]
It[7] aN [1]
It[8] aN [5]
It[9] aN [4]
Finished Dijkstra

Lowest distances!
D=[0,107,69,82,181,113,85,49,75,101,]
Was working for [0.000040] sec.
```

As shown above to compile the program the standard GNU compiler of *gcc* is used. The source code files *dijkstra.c* and *Dijkstra_tools.c* are taken as the inputs and the outputted executable *dijkstra* is created in the same directory. It is recommended to turn off all warnings with the *-w* flag as the program contains formatting cautions which do not affect the results or efficiency.

The program takes in two arguments. The first specifies the number of nodes that the graph will generate at runtime to test Dijkstra Algorithm, the second is an optional argument that passes the value to the *srand()* function. This value is then used to produce the path distance between each node in the graph.

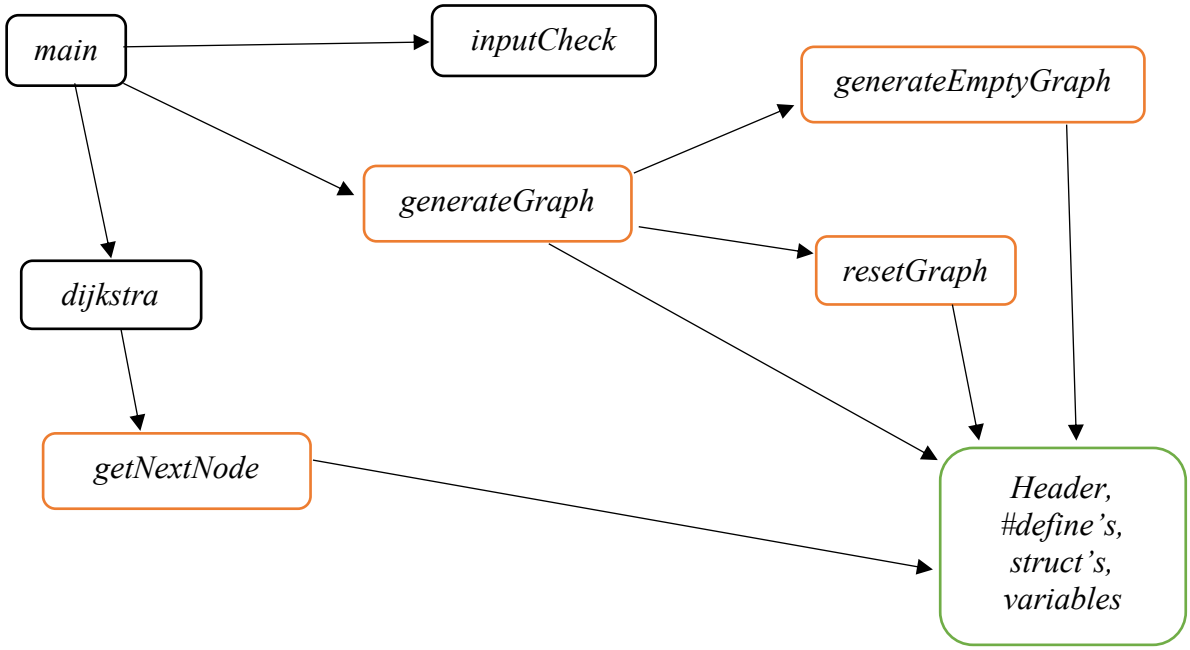
Program Overview

Functional Overview:

dijkstra.c ----------

dijkstra_tools.c -----

dijkstra_tools.h -----



Overview of Dijkstra's Algorithm:

Dijkstra's algorithm finds the shortest path between nodes in a graph. It focuses around the single source shortest path problem (SSPP) where finding the shortest path from the source vertex to all other vertices is required. For example, if nodes represented cities and edges represent driving distances then Dijkstra's algorithm can be used to find the shortest route between cities. The benefit of Dijkstra's algorithm is that any sub path B -> D of a shortest path A -> D is also the shortest path for the vertices B and D.

The algorithm does not attempt to make a singular direct evaluation towards the destination point but rather determines its distance from the starting point upon each node. This is then refined to find the shortest path distance by updating a stored variable with the new shortest distance when compared against each node option. For more information please see Appendix A.

Creating a New Graph:

The following is a high-level breakdown of the steps and functions called when generating a new graph:

1. After the users input variables are checked and approved the *main* function calls *generateGraph*. It is passed the users node size, the value for *srand()*, an empty struct, and a debug option. The *graph G* struct (Figure 1) is created in the *main* method and passed to/from where it is needed.
2. *generateGraph* immediately calls *generateEmptyGraph* passing it the node size and the empty *graph G*.
3. The point of *generateEmptyGraph* is to initialise the members of the *G* struct. It assigns a portion of memory to each member and updates the number of nodes.
4. After *generateEmptyGraph* is completed *resetGraph* is called passing it struct *G*. This function iterates through all the nodes inside *G* and sets them to 'Not Visited' and their shortest path length to 'Infinity'. This is similarly represented on image 2 of Appendix A.
5. Once *resetGraph* is completed we continue to step through *generateGraph*. An algorithm to create path distances (can also be called weights or lengths) between each node is used. Using the user defined value passed to *srand()*, the distances are randomly generated between each node. The algorithm ensures that fully connected graphs have at least one edge to someone else.

```
typedef struct {
    long N; //Number of nodes
    char** node; //Matrix of Nodes and connections

    int* D; //Result of Dijkstra Algorithm. Shortest path length for each node.
    char* visited; //Used to flag which nodes have been visted yet or not.
} graph;
```

***Figure 1*:** The members represented inside the struct *graph*. Inside Simple Dijkstra, *graph* is created and defined as variable *G*.

Evaluating/Solving using Dijkstra:

To solve the newly created graph, Dijkstra's algorithm must be applied. The algorithm is called from the *main* where it is passed its graph, starting point and a debug option. Appendix B displays the sequential version of the algorithm. The algorithm begins by setting the initial and next node of the graph to 0. A 'for loop' iterates through all the nodes in the graph setting the state of the current vertex to 'visited'. With the state set to 'visited' the algorithm now knows which vertices it has checked. From here a nested 'for loop' iterates through all nodes to find which ones are connected to the current vertex being checked. The lowest distance is then updated after two *if* statements evaluate if there is a connection to that node at all and if so is the new connection a smaller distance than any previous. Once the nested 'for loop' is completed the algorithm grabs the next node to be evaluated and continues.

Profiling:

After visually analysing Simple Dijkstra's code it was time to use a profiler to examine the applications computationally hot spots. Apple's IDE Xcode 10 comes with an integrated performance analyser called Instruments. Using the Instruments tool and operating the profiling method CPU Usage, I was able to examine the CPU consumption run against the application.

To gain accurate and reliable results all profiling tests in the following report contain the same input variables of 150,000 nodes and an *srand()* value of 19. The development machine was plugged into power and the only application open during the tests was Xcode.

Weight		Self Weight	Symbol Name
8.54 min	100.0%	0 s	▼Dijkstra Profiling (78936)
8.54 min	99.9%	0 s	▼Main Thread 0xc857fb
8.54 min	99.9%	3.00 ms	▼main Dijkstra Profiling
5.07 min	59.3%	5.07 min	▼generateGraph Dijkstra Profiling
1.00 ms	0.0%	1.00 ms	generateEmptyGraph Dijkstra Profiling
3.36 min	39.3%	2.82 min	▼dijkstra Dijkstra Profiling
32.27 s	6.2%	32.27 s	getNextNode Dijkstra Profiling
6.41 s	1.2%	6.41 s	DYLD-STUB$$rand Dijkstra Profiling
641.00 ms	0.1%	639.00 ms	▶<Unknown Address>
1.00 ms	0.0%	1.00 ms	DYLD-STUB$$puts Dijkstra Profiling

Figure 2: The time and percentage breakdown of Simple Dijkstra's CPU consumption. Total application running time is 8 minutes and 54 seconds.

Figure 2 above illustrates that Simple Dijkstra spends 59.3% of its time generating a fresh graph. This is understandable as this function performs the majority of memory allocations as well as read and writes required by the program. The line by line breakdown in Figure 3 demonstrates that the creation of a random number equates for 54.8% of the overall consumption inside *generateGraph*.

```
//Initialize Matrix                                                          
for(i = 0; i< G->N; i++)                                                     
{                                                                            
    linkCnt=0; //Keep track of # of outgoing edges                           
    for(j = 0; j < G->N; j++)                                        8.9%    
    {                                                                        
        if(i == j){                                                  2.4%    
            t = NO_CONN;                                                     
        } else {                                                             
            t = (rand() % ((MAX_EDGE_WEIGTH-1) * 2)+1); //50% of having no connection  54.8%
            if(t > MAX_EDGE_WEIGTH){                                 2.3%    
                //t = INF; //Like no connection                              
                t = NO_CONN; //Like no connection                            
            } else {                                                         
                linkCnt++;                                           3.6%    
                G->visited[j] = VISITED; //Do this to find isolated nods that have no incomming edge  1.9%
            }                                                                
        }                                                                    
                                                                             
        G->node[i][j] = t;                                           22.0%   
    }                                                                        
```

Figure 3: Line by line breakdown of the computationally heavy section of the *generateGraph* function.

It was also discovered that 39.3% (Figure 2) of the applications CPU time was spent solving Dijkstra's Algorithm. A line by line breakdown in Figure 4 shows that the arithmetic, logical and conditional operations performed by the two *if* statements induced a large percentage of runtime inside *dijkstra* function.

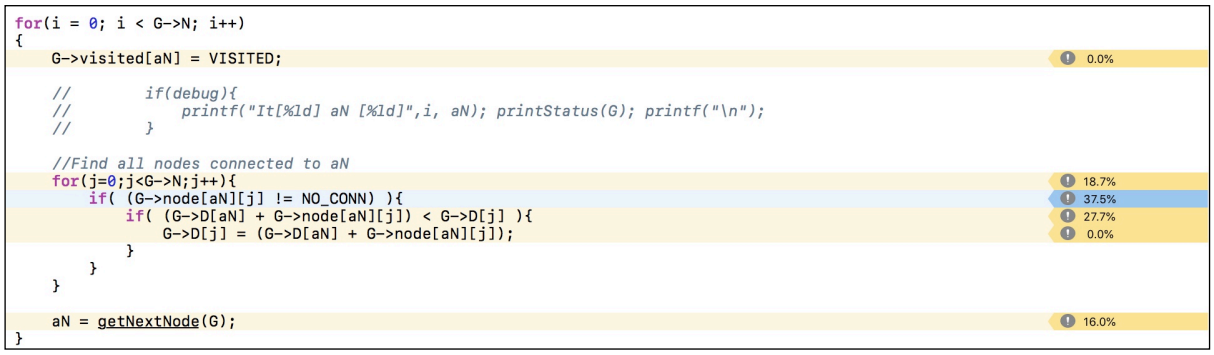


***Figure 4*:** Line by line breakdown of the computationally heavy section of the *dijkstra* function.

On the final line of Figure 4 the *getNextNode* function is called, overall consumption of this function was 6.2% (Figure 2). After analysing the line by line profiling of *getNextNode* (Figure 5) it was clear to see that logical and conditional operations performed by the *if* statement accounted for a heavy chunk of the compute time.

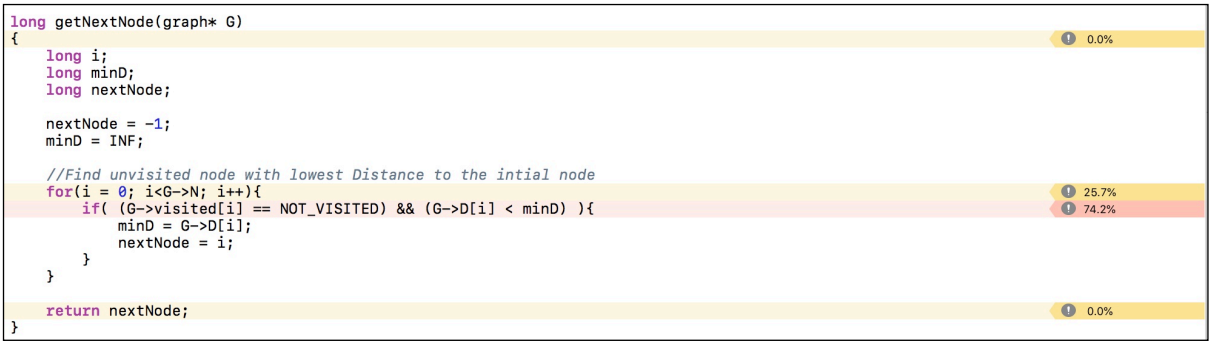


***Figure 5*:** Line by line breakdown of the *getNextNode* function.

Potential Parallelism

Parallelizing Dijkstra's algorithm is useful in the real world for, applications that require fast results when path changes occur such as GPS's. The average case time complexity for Dijkstra algorithm is $O(n^2)$. Initially my thought was that it would be possible to increase this average case with parallelization. However, with some research I discovered that even when distribution across multiple threads occur the speed at which a Dijkstra's is calculated will only ever be proportional to the number of nodes. Meaning the complexity will stay the same but the overall time taken will be shorter then when run sequentially.

From Figure 2 we can see that the focal point of parallelization must occur inside *generateGraph*, *dijkstra*, *getNextNode* and *generateEmptyGraph*. The remaining percentage of run time for Simple Dijkstra is library calls, these are irrelevant to the parallelization of Simple Dijkstra's itself.

Algorithm 1 (inside *generateGraph*)**:**

> Refer to – Figure 3 for a line by line breakdown of the computational hot spots and see Appendix C for the entire algorithm.
>
> Description – As described previously in the section 'Creating a New Graph', the aim of *generateGraph* is to create a fresh graph for testing on Dijkstra's algorithm. It can be seen that the computational hot spot of this function begins with the nested *for* loop this was an ideal spot to attempt parallelization of this algorithm.

Control Flow Constraints – By exploiting the nested *for* loop of *j*, I could eliminate any control flow issues that would occur if parallelization began on the outer *for* loop of *i*. This permits the *j* loop to be broken into chunks that I could then distribute onto threads evenly.

Dependencies – The first dependencies acting on the algorithm occurs on line 121 where *G -> node[i][j] = t*. By parallelizing only, the inner *for* loop, this dependency becomes negligible as *i* will remain the same across all threads while each chunk being computed for *j* updates the *node* variable correctly.
The second dependency for the algorithm involves the variable *linkCnt*. If I parallelized the outer *for* loop, this variable would become private to each thread. This causes an dependency error when *if(linkCnt == 0)* is called. The *if* statement needs to know the final value for a singular variable of *linkCnt* to avoid the possibility of the *if* statement executing on each thread. This dependency is avoided by parallelizing the nested *for* loop only.
The third dependence requires the variable *t* to be private to each thread if the nested *for* loop is to be parallelized. Without *t* being private to each thread a race condition occurs where the variable can possibly be updated by one thread and used by another.
The final problem facing the parallelization of this algorithm is the *rand* function. *rand* is not thread safe it only uses a single static variable to hold state. Even with the consistent input seed, the results from using *rand* on this algorithm cause a different set of numbers to be generated to that of the sequential version with the same seed. Further attempt to fix this issue is discussed in section 'Parallelizing The Code'.

Algorithm 2 (inside *dijkstra*)**:**

Refer to – Figure 4 for a line by line breakdown of the algorithm.

Description – *dijkstra* use's Dijkstra's algorithm illustrated in Appendix A to solve an input graph of nodes. It can be seen that the computational hot spot of this function begins with the nested *for* loop this was an ideal spot to parallelize.

Control Flow Constraints – By exploiting the nested *for* loop of *j*, I could eliminate any control flow issues that would occur if parallelization began on the outer *for* loop of *i*. This permits the *j* loop to be broken into chunks that I could then distribute onto threads evenly.

Dependencies – The only dependency acting on the algorithm revolves around analysing the correct node. The variable *aN* holds the value for the current node being explored. This dependency effects the arithmetic logic and variable write of the nested *for* loop for any value that is not the initial node of 0. By parallelizing the nested *for* loop I can break the problem into chunks. This works as the *j* values are only ever using their current iteration never requiring a previous value. With this the shortest path length (*D*) for each node can be obtained as the *aN* variable is safely independent outside the parallel loop. Thus the outer *for* loop will give each thread a section of *G -> D* (path lengths) values to obtain for the current *aN* (starting node).

Algorithm 3 (inside *getNextNode*)**:**

Refer to – Figure 5 for a line by line breakdown of the entire algorithm

Description – *getNextNode* uses an algorithm that finds the next unvisited node with the lowest distance to the initial node. It does this by finding the next unvisited node in the graph and saving its distance to the variable *minD*. If it finds an unvisited node with a lower distance it will write over the currently stored node and distance, then return the final node to where it was called in *dijkstra's* function. It can be seen that the computational hot spot of this function centres around the *for* loop.
Control Flow Constraints and Dependencies – Parallelization of this *for* loop results in a race condition. If each thread was iterating a chunk of the loop and needed to update the variables *minD* and *nextNode*, there is a possibility this can occur at the same time as another thread. An attempt was made to parallelize the algorithm using private thread variables and then comparing each sequentially at the end, but this did not work successfully.

Algorithm 4 (inside *generateEmptyGraph*)**:**

```
void generateEmptyGraph(long N, graph *G)
{
    int i;

    assert((N > 0) && "N has to be bigger than zero!" );

    G->N = N;
    G->D = (int*)malloc(sizeof(int) * G->N );
    if(G->D == NULL) { perror("malloc"); exit(EXIT_FAILURE); }
    G->visited = (char*)malloc(sizeof(char) * G->N);
    if(G->visited == NULL) { perror("malloc"); exit(EXIT_FAILURE); }

    //Ensure continues data array (for OpenMPI send)
    char* temp;
    temp = (char*) malloc(G->N*G->N);                          100.0%
    if(temp == NULL) { perror("malloc"); exit(EXIT_FAILURE); }
    G->node = (char**)malloc(sizeof(char*) * G->N);
    if(G->node == NULL) { perror("malloc"); exit(EXIT_FAILURE); }
    for(i = 0; i < G->N; i++){
        G->node[i] = &temp[i*G->N];
    }
}
```

Figure 6: Line by line breakdown of the *generateEmptyGraph* function.

Description – As described previously in the section 'Creating a New Graph', the aim of *generateEmptyGraph* is to initialize the members of the graph struct. As show above the heaviest CPU consumption occurs on a single memory allocation. Using the profiling tool Allocations I can see that this function is 99.9% (Figure 7) of Simple Dijkstra's entire memory bound usage. The majority of runtime inside this function is spent allocating memory to create space for the matrix for this reason parallelization will not occur.

Bytes Used		Count	Symbol Name
95.50 MB	99.9%	7	start libdyld.dylib
95.50 MB	99.9%	7	main Dijkstra Profiling
95.50 MB	99.9%	4	generateGraph Dijkstra Profiling
95.50 MB	99.9%	4	generateEmptyGraph Dijkstra Profiling
95.50 MB	99.9%	4	malloc libsystem_malloc.dylib
160 Bytes	0.0%	3	printf libsystem_c.dylib
91.70 KB	0.0%	749	<Allocated Prior To Attach> Dijkstra Profiling

Figure 7: The bytes and percentage breakdown of Simple Dijkstra's memory usage. This test was performed with 10,000 nodes.

Tools Used for Parallelizing

Language Choice: OpenMP
OMP was an ideal language to parallelize Simple Dijkstra. As each algorithm only requires *for* loop parallelization, OMP's higher level ability to divide work across multiple threads with relative ease was favoured over PThreads.

Tools:
Using Xcode and Instrument's I was able to work on OMP parallelization with profiling. I attempted to use Intel's Parallel Studio however, found the detailing in Instruments to be of the same quality. The complier GCC was a possibility but adjustments to environment variables on macOS made this a hassle to set up. For this reason, LLVM's compiler Clang was utilised.

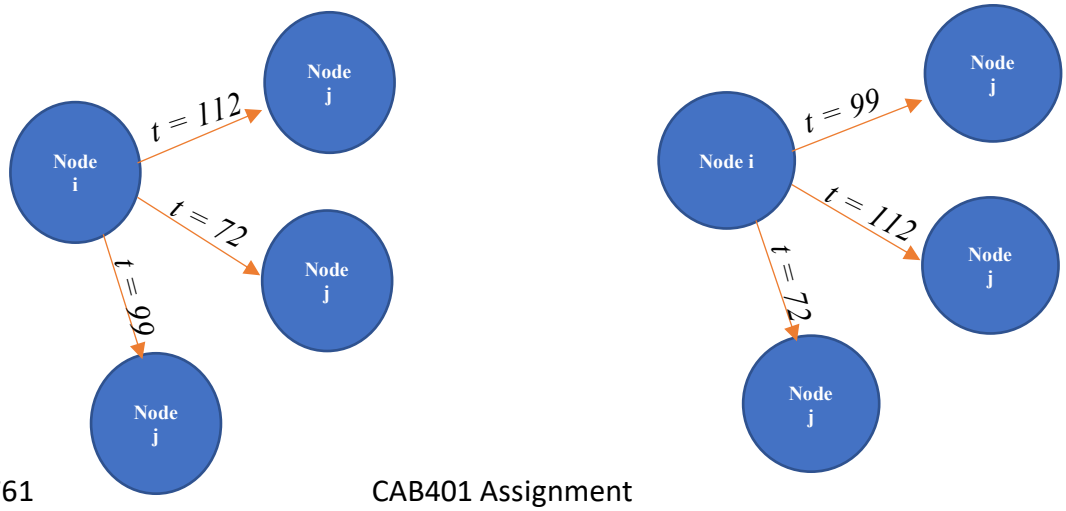
Parallelizing The Code

OMP Parallelization:

New file names: *dijkstra_omp.c Dijkstra_tools_omp.c Dijkstra_tools.h*

As established from the analysis it was possible for me to parallelize the nested *for* loop of - algorithm 1 and 2. OMP's automatic distribution across threads made it easy to use the single function call *#pragma omp parallel for schedule(runtime)* before each loop to create parallelism. By adding the new function *testScheduler* (Appendix D) called from the *main* I can initialize my threads and use the directive *omp_set_schedule* to my advantage. With the *static* call I am able to evenly divide the loops into chunks for each thread to execute in a round-robin fashion. Denoting the chunk size as, the input graph size divided by the number of threads allows for each loop to compute a portion of the matrix at a time (*G->N / nThreads)*.

An attempt was made to modify the *rand* function of algorithm 1 to a thread safe version of *rand_r* using a consistent seed. This did not produce the desired results, after each run the program still produced a non-consistent set of random values. This is because the *t* value was dependent on which thread section of finishing nodes (*j* nested loop*)* were assigned too, with respect to their consistent starting node *i (*outer loop). This is demonstrated in the diagram below, where each thread would have a different section of the loop each time the program was run. It was possible to get the same results however, multiple tests had to be run for the chance of this occurring. For this reason, *generateGraph* was eliminated from being parallelized because I was unable to gain consistent results.



Granularity control was a big factor in deciding the point where exploiting parallelism on the algorithm was worth doing. The table below shows the point where using parallel architecture is more beneficial than sequential as I increased the number of nodes.

To keep testing/timing uniform the debug options for printing the graph and final distances were kept off. These graphs were scrupulously checked against the original sequentially application graphs to make sure the results after parallelisation were the same.

– Shortest time

***dijkstra (Algorithm 2)* Appendix E**							
Number of Threads (OMP)							
Sequential		**2**		**3**		**4**	
Number of Nodes	**Time Taken (s)**	**Number of Nodes**	**Time Taken (s)**	**Number of Nodes**	**Time Taken (s)**	**Number of Nodes**	**Time Taken (s)**
50	0.000052	50	0.0005	50	0.00058	50	0.00059
150	0.00039	150	0.0008	150	0.0009	150	0.0013
400	0.0021	400	0.0019	400	0.0021	400	0.0022
500	0.003	500	0.0027	500	0.0028	500	0.0028

The results show that parallelism of this algorithm is only worth exploiting after 400 nodes. After 400 nodes using 2 threads becomes more beneficial to the program, with 3 threads producing the same time as the sequential version. With the current development machine, it is most beneficial to exploit parallelism after 500 nodes so that positive results are seen running across 2, 3 or 4 threads. See Appendix G for the implementation of granularity control.
Scalability for this algorithm is easily achievable with OMP. As the number of threads increase OMP's automatic distribution allows for scaling. The parallelized *for* loop for *dijkstra's* function can be simply broken into chunk sizes across parallel hardware.

In an attempt to gain the best speed possible, I chose to test and analyse the quality POSIX threads had on algorithm 2.

Language Choice: Pthreads
Pthreads' complex lower level architecture made it a possibility that speed up could be improved if I exploited it correctly.

Tools:
Using Xcode and Instrument's I was able to work on Pthread parallelization with profiling. The complier GCC from the GNU project was used. In addition, the -lpthread flag was included to incorporate the correct library.

Pthreads complex nature meant that I had to create and manually distribute work across threads. Depending on the number of threads being tested I created a series of functions that distributed the *for* loop manually across the threads. For example, testing with 2 threads

meant I created two functions where one function would solve half the nodes on the first thread and the second function would solve the other half the nodes on the second thread. To see this for 1, 2, 3 and 4 thread testing refer to Appendix F.

Granularity control of the POSIX Thread version of algorithm 2 is shown below.

To keep testing/timing uniform the debug options for printing the graph and final distances were kept off. These graphs were scrupulously checked against the original sequentially application graphs to ensure the results after parallelisation were the same.

– Shortest time

***dijkstra (Algorithm 2)* Appendix F**							
Number of Threads (POSIX THREADS)							
Sequential		**2**		**3**		**4**	
Number of Nodes	**Time Taken (s)**	**Number of Nodes**	**Time Taken (s)**	**Number of Nodes**	**Time Taken (s)**	**Number of Nodes**	**Time Taken (s)**
50	0.000052	50	0.019	50	0.013	50	0.012
150	0.00039	150	0.09	150	0.076	150	0.073
400	0.0021	400	0.56	400	0.54	400	0.58
500	0.003	500	0.99	500	0.94	500	0.99

The results show that the parallelism of this algorithm in Pthreads isn't worth exploiting to the extent that OMP is. OMP improves speed of the algorithm compared to the sequential version after 400 nodes, Pthreads did not have this effect. Scalability for this algorithm in Pthreads is also a lot harder than OMP. Manual scaling must occur. Depending on the number of threads being used, loops to handle each section of nodes for that thread must be created.

Results

Profiling:

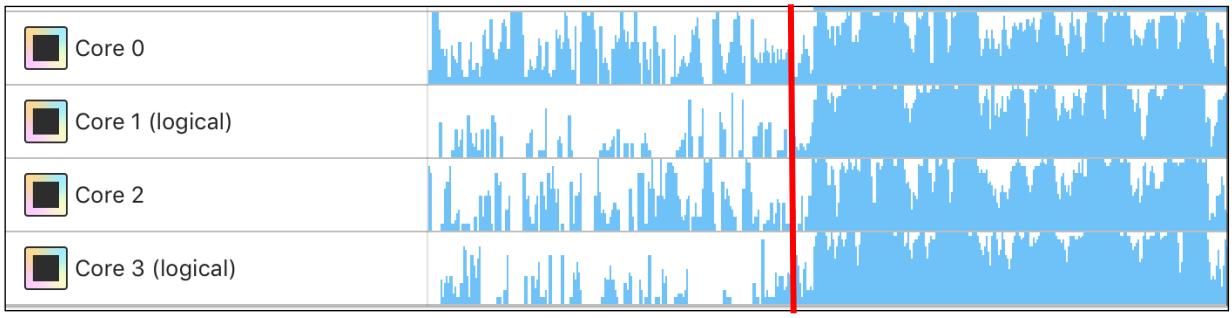


Figure 8: CPU consumption acting on the 4 threads of the parallel version of Simple Dijkstra. The red line denotes where the parallelism of *dijkstra* begins.

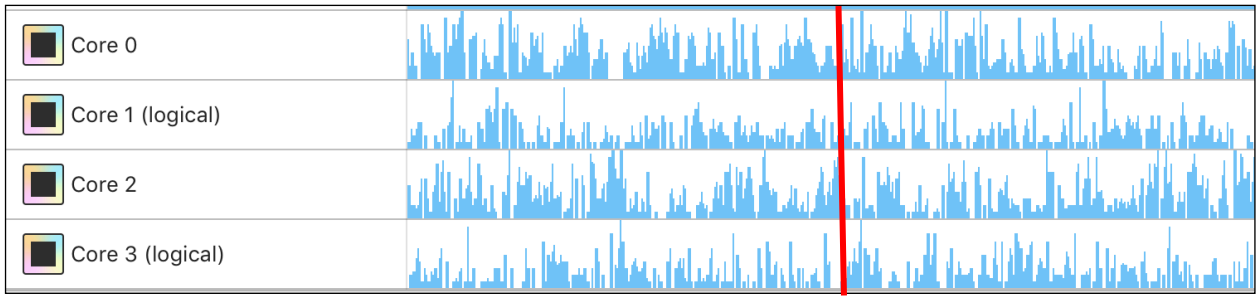


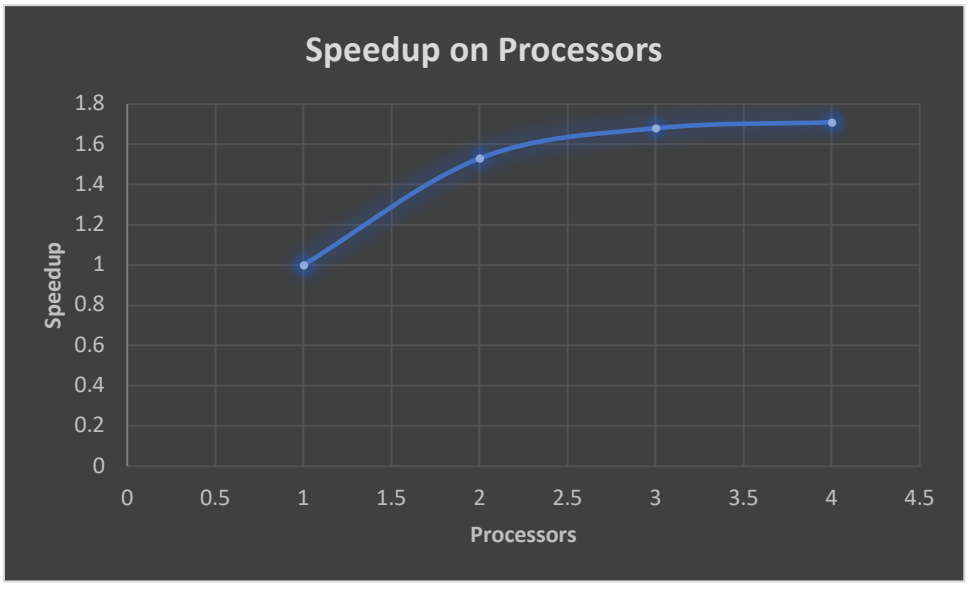
Figure 9: CPU consumption acting on the sequential version of Simple Dijkstra. The red line denotes where *dijkstra* begins.

The profiling results in Figure 8 shows how distribution occurs upon algorithm 2 being parallelized with 4 threads. This is compared to Figure 9 showing how the sequential version acts on CPU consumption. It is visibly possible to see how the distribution on each thread for the parallelized version is fairly even compared the sequential. OMP divides the *for* loop of algorithm 2 up into equal chunks producing this rather evenly distributed CPU consumption graph.

For timing results the *gettimeofday* function was used over *clock*. The *clock* function measures CPU time used by processes not the wall-clock-time. When you have multiple threads running at the same time you can burn through CPU time much faster.

Using 150,000 nodes the following data was collected.

Processors	Time (s)	Speedup
Sequential	518.1295	1x
2 Threads	338.375	1.53x
3 Threads	308.0	1.68x
4 Threads	302.7	1.71x



From the above graph it is clear to see a speedup of up to 1.71x with 4 threads. The speedup is most significant from the sequentially version to 2 threads. This proves that the use of OMP parallelization on algorithm 2 was beneficial from 400 nodes onwards. The development machine only allowed use of 4 threads however it is assumed the predicted logarithmic trend would continue with added processors and ease of scalability.

Problems Overcome

The biggest problem I faced parallelising Simple Dijkstra was analysing and understanding how dependencies effect the results. After considerable of trial, research and debugging I found new dependencies in certain algorithms that I did not see or understand previously. This had an effect because I was constantly updating parts of Simple Dijkstra that could or couldn't be parallelized. To overcome these problems, I found running break points on possible dependant variables would allow me to visualise when these variables were updated, making it clear which loops were unparallelizable.

Reflection

From the challenges faced throughout this assignment I have a new understanding of parallelization. Being able to profile results and visually see how each core acts under sequential and parallel formats gave me deeper understanding of computer hardware. With my attempt at parallelizing Simple Dijkstra I learnt how dependencies can affect the results and how private VS shared memory can play a large part in producing various outcomes.

I believe my attempt was successful but there was room for improvement. Being able to parallelize algorithm 3 (*getNextNode*) is a possibility but time restrictions did not allow me to achieve this correctly. Using private thread variables and then comparing each sequentially at the end to find the smallest distance was an option. I also believe using a global struct indexed by thread number to hold the results of each thread, then traversing the list to find the smallest distance node could also solve this problem.
Further investigation into a consistent random number generator across threads would also allow me to test parallelism on algorithm 1 (*generateGraph*).

Even though the profiling did not show any need to improve the *for* loops inside *resetGraph* and *generateEmptyGraph* I managed to parallelize these successfully. This was unnecessary to the overall speedup as the CPU consumption was so small on these loops it was not even recognised upon profiling.

Although the speed up was only achieved by 1 loop in the end, the analysis of the entire program taught me valuable lessons. In future research it would be of interest to test this OMP version of Simple Dijkstra when it is converted and run on a C++ or Fortran environment.

Bibliography

Cohen, D. (n.d.). *Implementation of Parallel Path Finding in a Shared Memory Architecture.* New York: Department of Computer Science Rensselaer Polytechnic Institute.

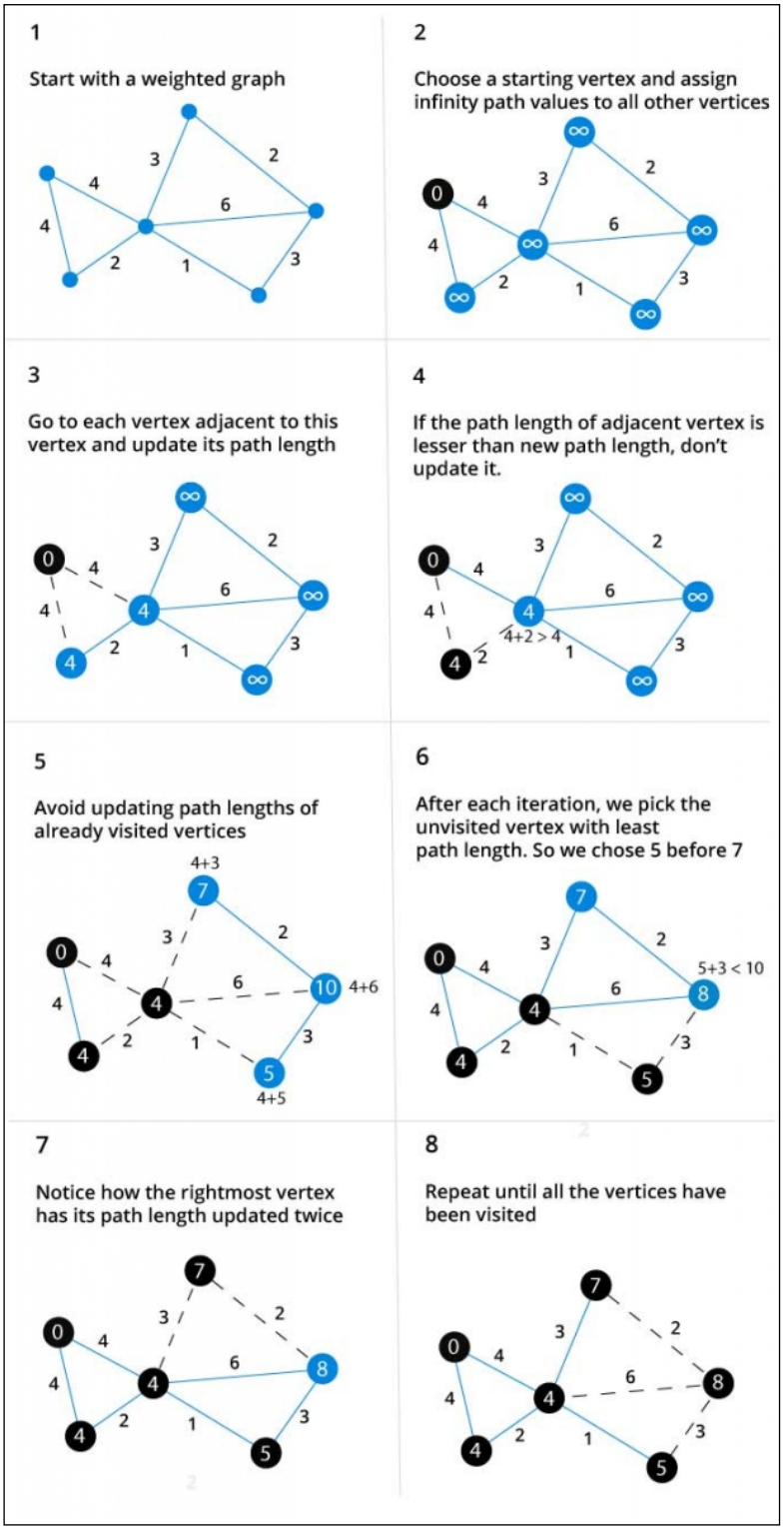
Dijkstra's Algorithm . (2017). Retrieved from Programiz: https://www.programiz.com/dsa/dijkstra-algorithm

MCDOUGAL, R. A. (2009, March 14). *BASIC THREADING IN C*. Retrieved from ROBERT A MCDOUGAL: http://ramcdougal.com/threads.html

Appendix

Appendix A

The following is a visual breakdown of how Dijkstra Algorithm works.



Appendix B

The sequential version of Dijkstra's algorithm.

```
64  void dijkstra(graph* G, long initial_node, char debug)
65  {
66      long i,j,k;
67      long aN; //actualNode
68      G->D[initial_node] = 0;
69      aN = initial_node;
70
71      for(i = 0; i < G->N; i++)
72      {
73          G->visited[aN] = VISITED;
74
75          //Find all nodes connected to aN
76          for(j=0;j<G->N;j++){
77              if( (G->node[aN][j] != NO_CONN) ){
78                  if( (G->D[aN] + G->node[aN][j]) < G->D[j] ){
79                      G->D[j] = (G->D[aN] + G->node[aN][j]);
80                  }
81              }
82          }
83
84          aN = getNextNode(G);
85      }
86  }
```

Appendix C

The sequential algorithm for parallelization inside *generateGraph*.

```
102     //Initialize Matrix
103     for(i = 0; i< G->N; i++)
104     {
105         linkCnt=0; //Keep track of # of outgoing edges
106         for(j = 0; j < G->N; j++)
107         {
108             if(i == j){
109                 t = NO_CONN;
110             } else {
111                 t = (rand() % ((MAX_EDGE_WEIGTH-1) * 2)+1); //50% of having no connection
112                 if(t > MAX_EDGE_WEIGTH){
113                     //t = INF; //Like no connection
114                     t = NO_CONN; //Like no connection
115                 } else {
116                     linkCnt++;
117                     G->visited[j] = VISITED; //Do this to find isolated nods that have no incomming
                            edge
118                 }
119             }
120
121             G->node[i][j] = t;
122         }
123
124         //Be sure to only generate fully connected graphs by each node having at least one edge to
                someone else!
125         if(linkCnt == 0)
126         {
127             printf("Adding outgoing link for [%ld]\n", i);
128             t = rand() % (G->N);
129
130             if(t == i) //NO self loops
131                 t = (t*t)%G->N;
132
133             G->node[i][t] = rand() % (MAX_EDGE_WEIGTH);
134         }
135     }
```

Appendix D

The new function added to the parallel version of Simple Dijkstra. This function allows the initialization of threads denoted to the size of the user's choice.

```
void testScheduler(int nThreads, graph* G)
{
    double runtime;

    //Set max nThreads
    omp_set_num_threads(nThreads);

    resetGraph(G);
    omp_set_schedule(omp_sched_static, G->N/nThreads);
    tick();
        dijkstra(G, 0, 1);
    runtime = tack();
    printf("Dijkstra algorithm was working for [%f] sec.\n",runtime);
}
```

Appendix E

The OMP parallelized version of Algorithm 2 inside *dijkstra*'s function.

```
for(i = 0; i < G->N; i++)
{
    G->visited[aN] = VISITED;

    //Find all nodes connected to aN
    #pragma omp parallel for schedule(runtime)
    for(j=0;j<G->N;j++){
        if( (G->node[aN][j] != NO_CONN) ){
            if( (G->D[aN] + G->node[aN][j]) < G->D[j] ){
                G->D[j] = (G->D[aN] + G->node[aN][j]);
            }
        }
    }

    aN = getNextNode(G);
}
```

Appendix F

The functions created to manually distribute Pthreads across threads evenly with algorithm 2. By creating a series of functions, I was able to distribute correctly the number of nodes depending on the number of threads being tested. The image on the left is the new algorithm 2, the image on the right is an example of distribution across 4 threads.

```
pthread_t threads [nThreads]; // Create threads

for(i = 0; i < G->N; i++)
{
    G->visited[aN] = VISITED;

    //Find all nodes connected to aN
    /* Launch Threads */
    if (nThreads = 1){
        pthread_create(&threads[0], NULL, pthreadLoop_1_thread, (void*) G);
    }
    if (nThreads = 2){
        pthread_create(&threads[0], NULL, pthreadLoop_2_thread_1, (void*) G);
        pthread_create(&threads[1], NULL, pthreadLoop_2_thread_1, (void*) G);
    }
    if (nThreads = 3){
        pthread_create(&threads[0], NULL, pthreadLoop_3_thread_1, (void*) G);
        pthread_create(&threads[1], NULL, pthreadLoop_3_thread_2, (void*) G);
        pthread_create(&threads[2], NULL, pthreadLoop_3_thread_3, (void*) G);
    }
    if (nThreads = 4){
        pthread_create(&threads[0], NULL, pthreadLoop1, (void*) G);
        pthread_create(&threads[1], NULL, pthreadLoop2, (void*) G);
        pthread_create(&threads[2], NULL, pthreadLoop3, (void*) G);
        pthread_create(&threads[3], NULL, pthreadLoop4, (void*) G);
    }
        /* Wait for Threads to Finish */
        if (nThreads = 1){
            pthread_join(threads[0], NULL);
        }
        if (nThreads = 2){
            pthread_join(threads[0], NULL);
            pthread_join(threads[1], NULL);
        }
        if (nThreads = 3){
            pthread_join(threads[0], NULL);
            pthread_join(threads[1], NULL);
            pthread_join(threads[2], NULL);
        }
        if (nThreads = 4){
            pthread_join(threads[0], NULL);
            pthread_join(threads[1], NULL);
            pthread_join(threads[2], NULL);
            pthread_join(threads[3], NULL);
        }
        aN = getNextNode(G);
   }
```

```
/* For four threads part 1 */
void * pthreadLoop1(void * G){
   graph *Gg = (graph *)G;
        for(long j=0;j<Gg->N/4;j++){
            if( (Gg->node[aN][j] != NO_CONN) ){
                if( (Gg->D[aN] + Gg->node[aN][j]) < Gg->D[j] ){
                    Gg->D[j] = (Gg->D[aN] + Gg->node[aN][j]);
                }
            }
        }
}

/* For four threads part 2 */
void * pthreadLoop2(void * G){
    graph *Gg = (graph *)G;
    for(long j=Gg->N/4;j<Gg->N/2;j++){
        if( (Gg->node[aN][j] != NO_CONN) ){
            if( (Gg->D[aN] + Gg->node[aN][j]) < Gg->D[j] ){
                Gg->D[j] = (Gg->D[aN] + Gg->node[aN][j]);
            }
        }
    }
}

/* For four threads part 3 */
void * pthreadLoop3(void * G){
    graph *Gg = (graph *)G;
    for(long j=Gg->N/2;j<(Gg->N/2+Gg->N/4);j++){
        if( (Gg->node[aN][j] != NO_CONN) ){
            if( (Gg->D[aN] + Gg->node[aN][j]) < Gg->D[j] ){
                Gg->D[j] = (Gg->D[aN] + Gg->node[aN][j]);
            }
        }
    }
}

/* For four threads part 4 */
void * pthreadLoop4(void * G){
    graph *Gg = (graph *)G;
    for(long j=Gg->N/4;j<Gg->N;j++){
        if( (Gg->node[aN][j] != NO_CONN) ){
            if( (Gg->D[aN] + Gg->node[aN][j]) < Gg->D[j] ){
                Gg->D[j] = (Gg->D[aN] + Gg->node[aN][j]);
            }
        }
    }
}
```

Appendix G

The exploitation of granularity control on the parallelised algorithm 2. Using an *if else* statement outside the *for* loop allows the program to make one comparison on the number of nodes. Whereas if the comparison was made inside the *for* loop each iteration would mean checking the *if else* statement, unnecessarily slowing the program down.

```
void dijkstra(graph* G, long initial_node, char debug)
{
    long i,j,k;
    long aN; //actualNode
    G->D[initial_node] = 0;
    aN = initial_node;

    /* Grainuality Control */

    if (G->N > 500){  // Exploit Parallelism after 500 nodes
        for(i = 0; i < G->N; i++)
        {
            G->visited[aN] = VISITED;

            //Find all nodes connected to aN
            #pragma omp parallel for schedule(runtime)
            for(j=0;j<G->N;j++){
                if( (G->node[aN][j] != NO_CONN) ){
                    if( (G->D[aN] + G->node[aN][j]) < G->D[j] ){
                        G->D[j] = (G->D[aN] + G->node[aN][j]);
                    }
                }
            }
            aN = getNextNode(G);
        }
    }
    else{
        for(i = 0; i < G->N; i++)
        {
            G->visited[aN] = VISITED;
            //Find all nodes connected to aN
            for(j=0;j<G->N;j++){
                if( (G->node[aN][j] != NO_CONN) ){
                    if( (G->D[aN] + G->node[aN][j]) < G->D[j] ){
                        G->D[j] = (G->D[aN] + G->node[aN][j]);
                    }
                }
            }
            aN = getNextNode(G);
        }
    }
}
```