|  |
| --- |
| CS 495 Spring |
| Parallelizing DA |
| Using OpenMP |

|  |
| --- |
| Emily Bodenhamer  Dr. Donald Davendra |

Table of Contents

[Introduction 1](#_Toc40646365)

[DA Initialization 3](#_Toc40646366)

[ArrayMem.c 4](#_Toc40646367)

[SelectFunctions.c 5](#_Toc40646368)

[DA.c 5](#_Toc40646369)

[Hardware Specs 8](#_Toc40646370)

[GPU Specs 9](#_Toc40646371)

[Introduction 9](#_Toc40646372)

[ArrayMem.c Complexity 9](#_Toc40646373)

[SelectFunctions.c Complexity 11](#_Toc40646374)

[DA.c Complexity 14](#_Toc40646375)

[FillIn Chart 17](#_Toc40646376)

[Loop Unrolling 21](#_Toc40646377)

[ArrayMem.c 26](#_Toc40646378)

[SelectFunctions.c 27](#_Toc40646379)

[DA.c 28](#_Toc40646380)

[Conclusion 32](#_Toc40646381)

Table of Figures

[Figure 1. Dragonfly Optimization fork & join diagram 3](#_Toc40121106)

[Figure 2: Population Matrices execution time through different thread counts 17](#_Toc40121107)

[Figure 3: Larger Population Matrices execution time through different thread counts 18](#_Toc40121108)

Table of Tables

[Table 1: Time performance when using the IDE and command line for the fillIn function when using the population matrix. 10](#_Toc40705962)

[Table 2: Time performance when using the IDE and command line for the getFun function when using the fitness vector. 12](#_Toc40705963)

[Table 3: Time performance when using the IDE and command line for the random walk function when using the population matrix. 14](#_Toc40705964)

[Table 4: Time performance when using the IDE and command line for the findNeighbors function when using the neighbor population and step matrices. 16](#_Toc40705965)

[Table 5: Average execution time from fillIn method for 500x300 18](#_Toc40705966)

[Table 6: Average execution time from fillIn method for 500x30 19](#_Toc40705967)

[Table 7: Average execution time from fillIn method for 1000x500 19](#_Toc40705968)

[Table 8: Average execution time from fillIn method for 1000x1000 20](#_Toc40705969)

[Table 9: Average execution time from fillIn method for 10000x500 20](#_Toc40705970)

[Table 10: Average execution time from fillIn method for 10000x1000 20](#_Toc40705971)

[Table 11: Time execution on a 500x300 matrix when using PCG on multithreads 22](#_Toc40705972)

[Table 12: Time execution on a 500x300 matrix when using MT on multithreads 23](#_Toc40705973)

[Table 13: Average time taken from unrolling loops on a 500x300 matrix 27](#_Toc40705974)

[Table 14: Average time taken from unrolling loops for Schwefel on a 1000x500 matrix 27](#_Toc40705975)

[Table 15: Time in ms of the findNeighbors method for 500x300 matrix 29](#_Toc40705976)

# Introduction

This documentation will cover the possible pragmas that can be added to the Dragonfly algorithm that was first implemented in fall 2019. Visual Studio 2019 will be used to implement the OpenMP pragmas to the algorithm.

# DA Initialization

For parallelization, the randomly generated population, initialization, variable updates, vector updates and fitness calculations can be used to reduce the amount of time to get the optimal solution.

A screenshot of a cell phone

Description automatically generated

Figure 1. Dragonfly Optimization fork & join diagram

This diagram shows the different places parallelization will be placed throughout the algorithm. The initial population and fitness are parallelized, the weights are updated sequentially, all the different vectors, factors, and fitness are updated in parallel. Finally, the enemy and food position are updated sequentially, and the next iterations continue the cycle.

Starting with initialization, when a population and the fitness of that population is calculated it can be parallelized.

# ArrayMem.c

For ArrayMem.c file, one for loop that can be parallelized is within the fillIn function that will fill a matrix with random real numbers within a specified range.



Either loop can be executed in parallel, however by making the outer loop parallel it will reduce the number of forks/joins. Each thread will need its own private copy of j. The code would look like the following:

# SelectFunctions.c

Within the getFun function in SelectFunctions.c, every for loop could be parallelized to quicken the collection of the fitness results obtained. For example, this code could be changed by adding the following pragma:



Since there is no inner loop, i does not need to be explicitly declared private.

# DA.c

After the initialization has been parallelized, parallelizing every dragonfly can be done by adding the following pragmas to the code:



Only the inner for loop needs the to be parallelized so i needs to be private. Finding neighboring dragonflies can also be parallelized.



The distance function that is called in the findNeighbors function can be parallelized by using the parallel pragma:

The next step is to update the separation, alignment, cohesion, distraction, and attraction factors. For separation, the first double for loop can be parallel while the next for loop will need to wait to be executed until myDA->sVector is done being calculated. Alignment, cohesion, distraction, and attraction have similar for loops, so they will also have the same pragmas implemented. Finally, to update the velocity vector and population, the following pragmas can be added:

The nowait pragma was added to ensure that population[i][t] was not updated before myDA->step[i][t] was done updating.

Code Complexity Break Down

# Hardware Specs

|  |  |
| --- | --- |
| Processor | Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2496 Mhz, 4 Core(s), 4 Logical Processor(s) |
| OS Name | Microsoft Windows 10 Home |
| Installed Physical Memory (RAM) | 16.0 GB |
| Total Physical Memory | 15.9 GB |
| Available Physical Memory | 9.28 GB |
| Total Virtual Memory | 18.3 GB |
| Available Virtual Memory | 8.12 GB |
| Disk Drive | CT1000MX500SSD4  Size - 931.51 GB (1,000,202,273,280 bytes)  ST1000LM035-1RK172  Size - 931.51 GB (1,000,202,273,280 bytes) |
| Memory | Realtek PCIe GBE Family Controller  Intel(R) 100 Series/C230 Series Chipset Family PCI Express Root Port #4 - A113  PCI Express Root Complex  Qualcomm Atheros QCA61x4A Wireless Network Adapter  Intel(R) Xeon(R) E3 - 1200/1500 v5/6th Gen Intel(R) Core(TM) PCIe Controller (x16) - 1901  NVIDIA GeForce GTX 1050 Ti  Trusted Platform Module 2.0  Intel(R) Serial IO I2C Host Controller - A160  Intel (R) Smart Sound Technology (Intel(R) SST) Audio Controller  Intel(R) USB 3.0 eXtensible Host Controller - 1.0 (Microsoft)  Intel(R) Management Engine Interface  Intel(R) HD Graphics 630  Intel(R) Serial IO GPIO Host Controller - INT345D  Intel(R) Serial IO I2C Host Controller - A161  PCI Express Root Complex |

# GPU Specs

|  |  |
| --- | --- |
| Graphics Card Name | NVIDIA GeForce GTX 1050 Ti |
| CUDA Cores | 768 |
| Graphics Clock (MHz) | 1290 |
| Processor Clock (MHz) | 1392 |
| Memory Clock | 7 Gbps |
| RAM amount | 4 GB |
| Memory Interface | GDDR5 |
| Memory Bandwidth (GB/sec) | 112 |

# Introduction

Many of the functions initially chosen to parallelize in the section prior were not parallelized. This was chosen because when testing the time performance, there was no benefit found. Often time there was an increase of the execution time because the amount of time and processing power needed for parallelizing was more than what the original function needed. The following functions showed performance increase and often times the execution time had decreased by more than half of the sequential code.

# ArrayMem.c Complexity

Originally, the fillIn function for the ArrayMem.c file would use 4 threads. As depicted in the following code:



However, after testing the time increase/decrease by parallelizing the following function, it was discovered that adding more than 1 thread caused the function to produce worse time. The following table illustrates the time performance for the function below:

Table 1: Time performance when using the IDE and command line for the fillIn function when using the population matrix.



When following the table above, only the sequential version of the fillIn function and using 1 thread produces the best results. So, the updated version of the code should use 1 thread or remain sequential.



The time complexity for this function is O(n2). The space complexity for this function is determined by finding the total number of bytes each variable uses.

* 8\*row\*col bytes of space is needed for double matrix arr and another 8\*row\*col bytes are needed for the return.
  + So, 64\*2row\*2col.
* 4 bytes for each row, col, j, and i.
  + So, 4\*4 = 16 bytes.
* 8 bytes are needed for min, max, and the value from genrand\_real1().
  + So, 8\*3 = 24 bytes.

So, the total space complexity is row\*col when the constants are removed. There are no data dependencies in this function since it is only a simple assignment operator.

# SelectFunctions.c Complexity

Originally, the for loop in the getFun method in the SelectFunctions.c file was parallelized. After testing the time taken for different threads and different benchmark functions, such as Schwefel and Sine envelope, there is a lot of improvement when using 4 threads in the for loop. The table below shows the different times from the two benchmark functions when using 2 or 4 threads or using sequential code. The best times were highlighted in yellow were also all from using 4 threads.

Table 2: Time performance when using the IDE and command line for the getFun function when using the fitness vector.



So, the updated version of the code will use 4 threads. The pragma will also be added to all 18 functions, because there were also similar improvements for the other functions like Schwefel and Sine Envelope.



The time complexity for this function is O(n2), because in order to get the result for each row, the function will need to be called and within the function, include another for loop that goes until the number of columns. This is shown in the code below, each of the 18 different benchmark functions have a for loop included so the entire getFun method will be O(n2).



The space complexity for this function is determined by finding the total number of bytes each variable uses.

* 8\*row\*col bytes of space is needed for double matrix arr and another 8\*row bytes are needed for the results array return.
  + So, 64\*2row\*col.
* 4 bytes for each row, col, counter, and i.
  + So, 4\*4 = 16 bytes.

So, the total space complexity is row\*col when the constants are removed. There are no data dependencies in this function since it is only a simple assignment operator. If the benchmark functions are also considered then, sum would be a data dependency because all of the threads would have to combine all their individual sum values together.

# DA.c Complexity

The following function was not originally considered for parallelization, however after adding pragmas to this function, improvements were obtained. The table below shows that for different NS and DIM for the population matrix, the time decreases by the increase of the number of threads used for parallelization. By using 4 threads, the time decreases by more than half of the time required when using sequential code with the largest number of solutions for the population matrix.

Table 3: Time performance when using the IDE and command line for the random walk function when using the population matrix.

 The time complexity for this function is O(n), because i does not change after each iteration of t, which means the current row will be iterated through linearly.



The space complexity for this function is determined by finding the total number of bytes each variable uses.

* Calculating the costliest bytes, for the DA and myData structs, both population and step matrices are 8\*row\*col.
  + So, 64\*2row\*2col.
* 4 bytes for each DIM and i.
  + So, 4\*2 = 8 bytes.

So, the total space complexity is row\*col when the constants are removed. There are no data dependencies in this function since only an assignment operation is needed.

The following function originally had a pragma only on the second for loop. In the implemented function the pragma was moved to the first for loop. Having the pragma on the first for loop helped with the time execution.



The table below shows that for different NS and DIM for the neighbor population and step matrices, the time decreases by the increase of the number of threads used for parallelization. By using 4 threads, the time decreases by more than half of the time required when using sequential code with the largest number of solutions for the matrices.

Table 4: Time performance when using the IDE and command line for the findNeighbors function when using the neighbor population and step matrices.



The time complexity for this function is O(n2), since there are two for loops, the first from 0 to NS and the other from 0 to DIM.



The space complexity for this function is determined by finding the total number of bytes each variable uses.

* Calculating the costliest bytes, for the DA and myData structs, both neighbor population and neighbor step matrices are 8\*row\*col.
  + So, 64\*2row\*2col.
* 4 bytes for each index, i, j, k, DIM, NS, myDA->numNeighbors,
  + So, 4\*7 = 28 bytes.

So, the total space complexity is row\*col when the constants are removed. There are no data dependencies in this function since only an assignment operation is needed.

# FillIn Chart



Figure 2: Population Matrices execution time through different thread counts

The graph above shows the time taken for different sized population matrices. This graph shows that for matrices with number of solutions less than 1000, using 2 or three threads will provide the longest execution time. The sequential code is often better to use for smaller dimensions and number of solutions.



Figure 3: Larger Population Matrices execution time through different thread counts

The graph above shows the time taken for different sized population matrices. The sizes for these matrices have number of solutions greater than 10000. Using more than 1 thread causes a large overhead that it is a hindrance to the program to use any more than 1 thread. The execution time doubles when using 2 or 3 threads.

Table 5: Average execution time from fillIn method for 500x300



The graphs above were created by taking the average of six iterations of the fillIn method. All the different sized matrices have their own table calculated. In order to get the time executed, the gettimeofday() method from the <sys/time.h> library was used instead of the clock() method from the <time.h> library. The gettimeofday() gave a closer time estimation.

The following tables are the rest of the iterations from the different sized matrices.

Table 6: Average execution time from fillIn method for 500x30



Table 7: Average execution time from fillIn method for 1000x500



Table 8: Average execution time from fillIn method for 1000x1000



Table 9: Average execution time from fillIn method for 10000x500



Table 10: Average execution time from fillIn method for 10000x1000



# PCG Random Number Generator

In an attempt for further optimization, the pcg generator was implemented in the fillIn() method to test the time execution. The pcg generator time execution was compared with the mersenne twister generator.



The code above shows the implementation of the pcg generator. Ldexp() was used in order to generate doubles.

Table 11: Time execution on a 500x300 matrix when using PCG on multithreads



The table above shows the time in ms of using pcg on a 500x300 matrix for thirty iterations. The averages are displayed at the bottom. The averages indicate that using multithreads improves the time execution.

Table 12: Time execution on a 500x300 matrix when using MT on multithreads



The table above shows the time in ms of using mersenne twister on a 500x300 matrix for thirty iterations. When using multithreads for Mersenne twister, the time execution worsens. When comparing the time execution from pcg, Mersenne twister still produces better time.

Table 13: Time execution on a 1000x1000 matrix when using PCG on multithreads



The table above shows the time in ms of using pcg on a 1000x1000 matrix for thirty iterations. The averages are displayed at the bottom. The averages indicate that using multithreads improves the time execution.

Table 14: Time execution on a 1000x1000 matrix when using MT on multithreads

The table above shows the time in ms of using mersenne twister on a 1000x1000 matrix for thirty iterations. When using multithreads for Mersenne twister, the time execution worsens. When comparing the time execution from pcg, Mersenne twister still produces better time when using a larger matrix.

# Loop Unrolling

## ArrayMem.c

In order to further optimize the performance of DA, loop unrolling was implemented. Loop unrolling was first implemented on the fillIn method.



The code above shows the implementation of one unroll for the fillIn method.



The code above shows the implementation of three unroll for the fillIn method. Time in ms was taken from thirty iterations of zero loop unrolling, one unroll, and three unroll. The average was calculated for each unroll.

Table 15: Average time taken from unrolling loops on a 500x300 matrix



As seen from the table above, unrolling the loops does not produce any optimization. If the amount of unrolling was increased it is likely to see that the time will continue to increase.

## SelectFunctions.c

In SelectFunction.c, the next method that will be tested is getFun(). This function was tested using zero, one, three, four, and nine unrolls.



The code above shows the implementation used for three unrolls on the schwefel function. For the other schwefel amounts and the sine envelope function, the implementation will look like the code presented above.

Table 16: Average time taken from unrolling loops for Schwefel on a 1000x500 matrix



The table above shows the time in ms of the different unrolling amounts used for a 1000x500 matrix for the schwefel and sine envelope functions. Unrolling the loops does not produce any optimization. If the amount of unrolling was increased it is likely to see that the time will continue to increase.

## DA.c

In the DA.c file, the findNeighbors() function was tested using zero, one, and three loop unrolling.



The code above shows the implementation used for nine unrolls on the findNeighbors() function.

Table 17: Time in ms of the findNeighbors method for 500x300 matrix



The table above shows the time in ms of the different unrolling amounts used for a 500x300 matrix for the findNeighbors function. The last row on this table shows the averages of all the times obtained. This is the first function that shows optimization from implementing loop unrolling. Although the time has improved, the amount of lines added to the function causes the code to look unpolished and cluttered.

The next function to be modified is the randomWalk() function, which was tested using zero, one, and three loop unrolling amounts.



The code above shows the implementation used for four unrolls on the randomWalk() function.

Table 16: Time in ms of the randomWalk method for 1000x500 matrix



The table above shows the time in ms of the different unrolling amounts used for a 1000x500 matrix for the randomWalk() function. The last row on this table shows the averages of all the times obtained.

# Further Optimization

After testing, it was observed that 10000 number of solutions with 6600 dimensions produced a failure in thread creation. The output from the IDE shows the following:

libgomp: Thread creation failed: Resource temporarily unavailable

However, 6500 dimensions does not produce any errors. When testing with 10000 number of solutions and 6700 dimensions, the program runs out of available memory.

# Conclusion

There are many different pragmas that the OpenMP API has. The implemented pragmas have increased the optimization of the original code. There may be additional clauses that could be added to continue to optimize the performance or adding different pragmas to other functions. OpenMP encourages incremental parallelization so changing pragmas around will not be difficult if better pragmas are found.