CS 495 Spring

Parallelizing DA

Using OpenMP

Emily Bodenhamer

Dr. Donald Davendra

Table of Contents

Table of Figures

Table of Tables

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.

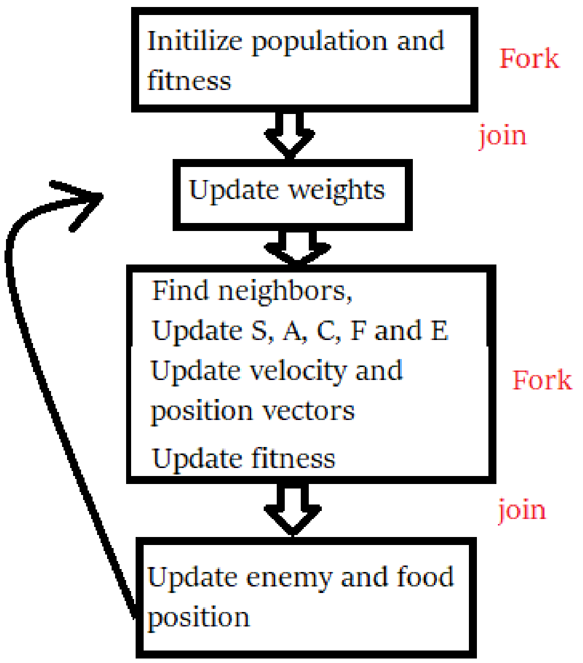
Initilize population and fitness
Fork
join
Update weights
Find neighbors,
Update S, A, C, F and E
Update velocity and position vectors
Update fitness
Fork
join
Update enemy and food position

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.

```
double **fillIn(double **arr, int row, int col, double min, double max) {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
        }
    }
    return arr;
}
```

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:

```
#pragma omp parallel for private(j)
double **fillIn(double **arr, int row, int col, double min, double max) {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
        }
    }
    return arr;
}
```

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:

```
#pragma omp parallel for
for (int i = 0; i < row; i++) {
    results[i] = schwefel(arr[i], col);
}
```

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:

```
#pragma omp parallel for private(i)
for (int t = 0; t < iterations; t++) {
    // update weights and radius
    updateWeights(myDA, myData, t, iterations);

    for (int i = 0; i < NS; i++) {
```

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

```
for (int k = 0; k < NS; k++) {
        distance(myDA, myData, i, k, DIM);
        if (lessR(myDA, DIM)) {
            index++;
            myDA->numNeighbors++;
            #pragma omp parallel for
            for (int j = 0; j < DIM; ++j) {
                myDA->neighborsPop[index][j] = myData->population[k][j];
                myDA->neighborsStep[index][j] = myDA->step[k][j];
            }
        }
    }
```

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

```
#pragma omp parallel for
    for (int k = 0; k < DIM; k++) {
        myDA->o[k] = sqrt(pow((myData->population[i][k] - myData->population[j][k]), 2));
    }
```

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.

```
#pragma omp parallel private(j,k)
 for (int j = 0; j < myDA->numNeighbors; ++j) {
     for (int k = 0; k < DIM; ++k) {
         myDA->sVector[k] += myDA->neighborsPop[j][k] - myData->population[i][k];
     }
 }
 #pragma omp for nowait
 for (int k = 0; k < DIM; ++k) {
     myDA->sVector[k] = -myDA->sVector[k];
 }
```

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:

```
        #pragma omp parallel for
        for (int t = 0; t < DIM; ++t) {
            // velocity matrix
            myDA->step[i][t] = (myDA->s * myDA->sVector[t] + myDA->a * myDA->aVector[t] +
                                    myDA->c * myDA->cVector[t] + myDA->f * myDA->fVector[t] +
                                    myDA->e * myDA->eVector[t]) + myDA->w * myDA->step[i][t];

            // if the new position is outside the range of
            // the bounds, then make it equal to the bounds
            checkBounds(myData, myDA->step[i][t]);

            #pragma omp nowait
            // position matrix
            myData->population[i][t] = myData->population[i][t] + myDA->step[i][t];

            // if the new population is outside the range of
            // the bounds, then make it equal to the bounds
            checkBounds(myData,myData->population[i][t]);

        }
```

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:

```
#pragma omp parallel for private(j)
double **fillIn(double **arr, int row, int col, double min, double max) {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
        }
    }
    return arr;
}
```

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.

process	object	NS	DIM	IDE time ms	cmd line time ms
sequential	population matrix	500	30	0	0
1 thread	population matrix	500	30	0	0
2 thread	population matrix	500	30	2	1
3 thread	population matrix	500	30	1	1
4 thread	population matrix	500	30	7	2
sequential	population matrix	1000	300	7	0
1 thread	population matrix	1000	300	7	7
2 thread	population matrix	1000	300	11	13
3 thread	population matrix	1000	300	11	12
4 thread	population matrix	1000	300	13	12
sequential	population matrix	10000	1000	261	202
1 thread	population matrix	10000	1000	261	202
2 thread	population matrix	10000	1000	448	442
3 thread	population matrix	10000	1000	390	387
4 thread	population matrix	10000	1000	389	394

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.

```
double **fillIn(double **arr, int row, int col, double min, double max) {
    int j, i;

    #pragma omp parallel for private(j) num_threads(1)
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
        }
    }

    return arr;
}
```

The time complexity for this function is O(n^2). 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.

process	object	function	NS	DIM	time ms IDE	cmd line time ms
sequential	fitness vector	schwefel	500	300	16	15
2 thread	fitness vector	schwefel	500	300	12	8
4 thread	fitness vector	schwefel	500	300	5	4
sequential	fitness vector	sineEv	500	300	133	84
2 thread	fitness vector	sineEv	500	300	45	43
4 thread	fitness vector	sineEv	500	300	36	20
sequential	fitness vector	schwefel	10000	1000	1014	985
2 thread	fitness vector	schwefel	10000	1000	498	489
4 thread	fitness vector	schwefel	10000	1000	324	265
sequential	fitness vector	sineEv	10000	1000	5662	5626
2 thread	fitness vector	sineEv	10000	1000	2869	2827
4 thread	fitness vector	sineEv	10000	1000	1420	1413

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.

```
double *getFun(double *results, double **arr, int row, int col, int counter) {
    switch (counter) {
        case 0:
            #pragma omp parallel for num_threads(4)
            for (int i = 0; i < row; i++) {
                results[i] = schwefel(arr[i], col);
            }
            break;
        case 1:
            #pragma omp parallel for num_threads(4)
            for (int i = 0; i < row; i++) {
                results[i] = deJong(arr[i], col);
            }
            break;
        case 2:
             .
             .
             .
        case 17:
            #pragma omp parallel for num_threads(4)
            for (int i = 0; i < row; i++) {
                results[i] = levy(arr[i], col);
            }
    }
        return results;
}
```

The time complexity for this function is O(n^2), 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(n^2).

```
double schwefel(double *array, int n) {
    double sum = 0.0;

    for(int i = 0; i < n; i++) {
        sum += (array[i] * -1) * sin(sqrt(fabs(array[i])));
    }

    return sum = (418.9829 * n) - sum;
}
```

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.

process	object	NS	DIM	time ms IDE	cmd line time ms
sequential	population matrix	500	300	5	5
2 thread	population matrix	500	300	2	2
4 thread	population matrix	500	300	1	1
sequential	population matrix	10000	1000	104	103
2 thread	population matrix	10000	1000	52	51
4 thread	population matrix	10000	1000	26	26

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.

```
void randomWalk(DA *myDA, initData *myData, int i, int DIM) {
    #pragma omp parallel for num_threads(4)
    for (int t = 0; t < DIM; ++t) {
        myData->population[i][t] = myData->population[i][t] + levyFlight(DIM) *
myData->population[i][t];
        myDA->step[i][t] = 0;
        // if the new position is outside the range of
        // the bounds, then make it equal to the bounds
        checkBounds(myData,myData->population[i][t]);
    }
}
```

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.

```
for (int k = 0; k < NS; k++) {
        distance(myDA, myData, i, k, DIM);
        if (lessR(myDA, DIM)) {
            index++;
            myDA->numNeighbors++;
            #pragma omp parallel for
            for (int j = 0; j < DIM; ++j) {
                myDA->neighborsPop[index][j] = myData->population[k][j];
                myDA->neighborsStep[index][j] = myDA->step[k][j];
            }
        }
    }
```

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.

process	object	NS	DIM	time ms IDE	cmd line time ms
sequential	neighborpop & step matrix	500	300	20	19
2 thread	neighborpop & step matrix	500	300	11	10
4 thread	neighborpop & step matrix	500	300	7	5
sequential	neighborpop & step matrix	10000	1000	1328	1286
2 thread	neighborpop & step matrix	10000	1000	659	649
4 thread	neighborpop & step matrix	10000	1000	367	350

The time complexity for this function is O(n^2), since there are two for loops, the first from 0 to NS and the other from 0 to DIM.

```
void findNeighbors(DA *myDA, initData *myData, int i, int DIM, int NS) {
    int index = 0;
    myDA->numNeighbors = 0;
    int j, k;
    #pragma omp parallel for private(k, j) num_threads(4)
    for (k = 0; k < NS; k++) {
        distance(myDA, myData, i, k, DIM);
        if (lessR(myDA, DIM)) {
            index++;
            myDA->numNeighbors++;
            for (j = 0; j < DIM; ++j) {
                myDA->neighborsPop[index][j] = myData->population[k][j];
                myDA->neighborsStep[index][j] = myDA->step[k][j];
            }
        }
    }
}
```

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

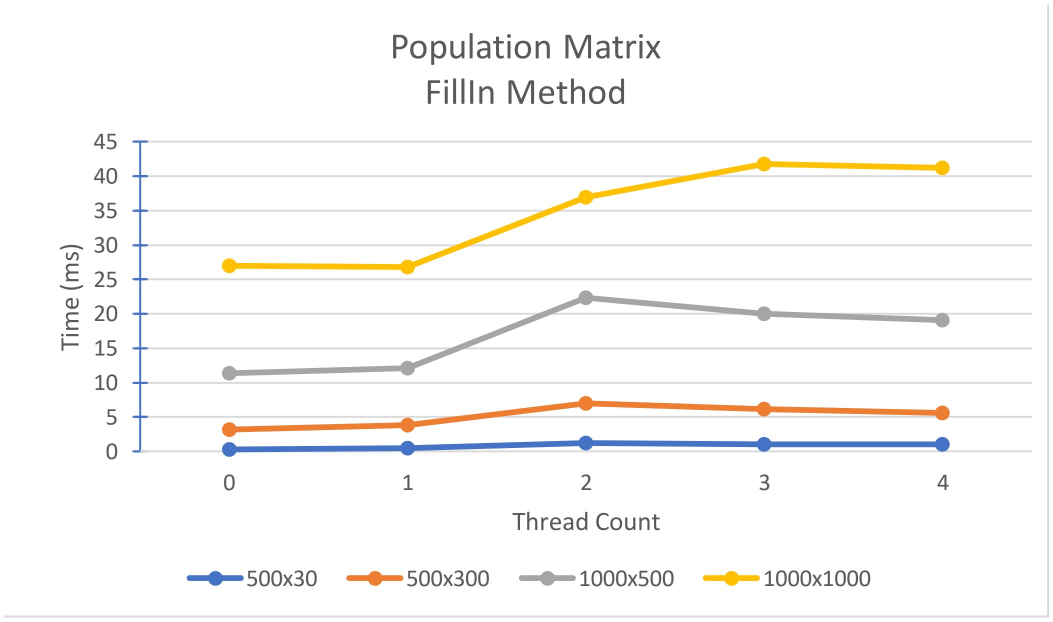

Figure 2: Population Matrices execution time through different thread counts

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.

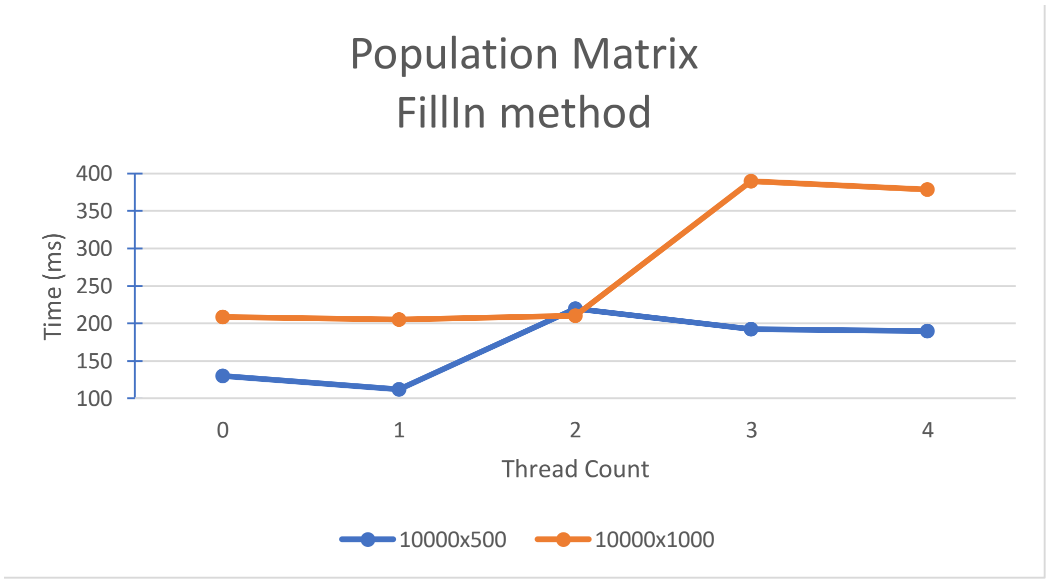

Figure 3: Larger Population Matrices execution time through different thread counts

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

0	1	2	3	4
2.992	3.989	7.027	6.957	5.5
3.989	3.037	6.935	5.973	4.982
3.019	3.952	6.981	5.984	5.985
2.987	4.023	6.939	5.982	4.996
2.994	4.001	6.981	5.954	5.981
3.035	3.983	6.981	5.954	5.985
3.169333	**3.830833**	**6.974**	**6.134**	**5.5715**

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

0	1	2	3	4
0	**0**	**1.024**	**1.995**	**0.996**
0	0.969	0.97	0	0.998
0.834	0.996	0.998	0.997	0.998
0.995	0.997	1.992	0.997	1.007
0	0	0.998	0.97	1.015
0	0	0.997	0.998	0.979
0.304833	**0.493667**	**1.163167**	**0.992833**	**0.998833**

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

0	1	2	3	4
11.016	11.935	22.942	21.007	18.92
11.968	14.539	22.947	19.928	17.952
10.97	12.015	23.08	19.911	19.945
10.971	11.964	20.909	19.949	19.948
10.966	9.932	21.901	19.176	18.91
11.968	11.968	21.938	19.939	18.968
11.30983	**12.05883**	**22.28617**	**19.985**	**19.10717**

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

0	1	2	3	4
25.963	21.942	35.93	40.864	38.911
22.987	24.932	42.859	43.39	37.883
23.939	25.891	45.878	39.876	38.874
28.907	24.982	30.453	43.883	44.855
21.939	24.933	34.905	39.869	39.882
37.921	37.899	31.902	42.874	46.848
26.94267	**26.76317**	**36.98783**	**41.79267**	**41.20883**

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

0	1	2	3	4
139.197	119.708	224.395	188.498	183.468
143.159	109.706	223.992	189.998	184.507
145.116	109.666	221.408	194.478	194.569
131.157	111.75	218.417	196.475	194.042
110.712	112.646	216.374	195.493	194.48
110.653	111.702	213.46	191.488	187.467
129.999	**112.5297**	**219.6743**	**192.7383**	**189.7555**

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

0	1	2	3	4
204.465	224.374	237.328	391.953	388.961
205.446	202.486	207.445	388.96	368.015
202.47	202.469	204.44	393.95	371.979
204.487	197.466	205.464	386.929	369.012
203.463	204.442	203.451	390.248	384.958
232.344	198.458	203.485	385.414	386.939
208.7792	**204.9492**	**210.2688**	**389.5757**	**378.3107**

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.

```
#pragma omp parallel for private(j) num_threads(4)
for (i = 0; i < row; i++) {
    for (j = 0; j < col; j++) {
        arr[i][j] = (max - (min)) * (ldexp(pcg32_random(), -32)) + min;
    }
}
```

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

Iterations	single thread	4 threads	10 threads
1	9.758	10.735	6.831
2	7.808	6.832	5.856
3	8.785	6.834	5.856
4	13.664	5.854	5.856
5	8.784	5.856	5.857
6	7.807	5.857	5.856
7	8.784	6.832	5.856
8	21.471	5.855	5.855
9	8.785	5.855	5.856
10	10.735	5.86	5.856
11	12.689	5.852	5.856
12	8.783	5.856	5.856
13	8.826	5.856	5.856
14	8.742	5.856	5.856
15	8.825	9.761	11.713
16	8.744	6.832	6.832
17	12.687	5.855	5.855
18	7.808	5.856	4.88
19	7.809	5.856	5.856
20	13.664	5.857	5.855
21	7.808	4.879	5.857
22	7.807	5.856	5.856
23	7.808	5.857	5.856
24	8.784	5.856	5.856
25	7.808	5.857	5.856
26	7.808	4.878	5.856
27	7.809	5.856	5.856
28	7.807	5.856	5.856
29	7.808	5.856	5.856
30	12.716	4.88	4.88
Averages	9.6307	6.181267	6.0511667

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

Iterations	single thread	4 threads	10 threads
1	3.903	4.879	5.854
2	6.832	3.903	5.857
3	2.928	2.928	5.856
4	3.904	5.857	5.858
5	2.928	5.859	4.878
6	3.904	4.878	5.856
7	2.927	5.854	5.856
8	5.857	5.856	5.855
9	3.905	4.879	5.856
10	2.928	5.857	5.856
11	2.927	5.39	5.857
12	3.904	5.855	5.855
13	2.928	5.855	5.856
14	11.713	5.856	5.856
15	3.903	4.881	5.857
16	2.928	5.856	5.855
17	3.904	5.855	4.88
18	2.928	8.785	5.858
19	3.907	8.784	4.878
20	2.925	4.88	5.858
21	2.927	4.881	5.854
22	3.904	5.857	5.856
23	2.928	5.854	5.856
24	2.929	4.881	5.857
25	3.903	5.856	4.879
26	2.929	5.864	5.856
27	3.905	5.848	5.856
28	2.926	5.387	5.856
29	2.928	5.856	5.856
30	3.905	4.88	4.88
Averages	3.8389	5.597033	5.693266

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

Iterations	single thread	4 threads	10 threads
1	88.234	82.744	69.812
2	65.822	138.629	59.841
3	66.33	56.925	51.86
4	65.823	46.837	51.861
5	64.827	46.876	48.87
6	65.825	47.38	49.044
7	65.824	47.872	51.862
8	65.824	46.384	49.867
9	64.869	47.872	50.863
10	72.804	47.38	49.375
11	78.789	46.875	50.863
12	82.29	46.874	51.864
13	69.811	47.872	50.862
14	65.824	46.874	49.275
15	65.824	48.87	50.864
16	65.858	47.871	51.377
17	65.87	65.336	54.849
18	65.818	59.839	52.857
19	65.83	59.841	49.868
20	65.902	63.828	59.349
21	64.791	56.848	51.861
22	65.863	55.851	49.866
23	65.82	48.869	50.865
24	65.864	48.869	49.868
25	65.073	46.874	49.865
26	66.075	48.87	52.859
27	65.827	46.874	49.866
28	65.976	47.872	50.864
29	66.428	47.873	47.872
30	64.86	47.871	50.864
Averages	67.8191667	54.655	51.99777

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

Iterations	single thread	4 threads	10 threads
1	57.887	46.867	43.885
2	28.314	49.867	46.873
3	27.053	48.869	46.875
4	27.926	49.868	46.875
5	26.927	47.871	46.881
6	26.928	48.87	45.878
7	26.929	47.871	46.495
8	26.439	47.872	47.873
9	27.926	47.873	48.87
10	25.931	48.476	46.873
11	26.964	49.375	43.885
12	25.927	48.87	46.875
13	26.938	53.857	46.874
14	33.507	42.885	46.943
15	26.928	47.872	45.968
16	25.935	48.868	47.002
17	26.89	48.871	46.991
18	26.928	48.869	46.494
19	25.965	48.869	45.482
20	26.928	49.867	46.002
21	25.934	49.866	46.966
22	26.924	47.872	45.991
23	25.934	47.872	46
24	26.891	48.379	46.966
25	26.963	47.871	45.99
26	26.894	49.867	46.999
27	26.946	48.869	45.971
28	26.943	48.869	46.002
29	26.935	48.87	47.004
30	26.918	46.875	45.993
Averages	28.0817333	48.59057	**46.4592**

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.

```
#pragma omp parallel for private(j) num_threads(1)
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j+=2) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
            arr[i][j+1] = (max - (min)) * (genrand_real1()) + min;
        }
        for (; j < col; j++) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
        }
    }
```

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

```
#pragma omp parallel for private(j) num_threads(1)
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j+=4) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
            arr[i][j+1] = (max - (min)) * (genrand_real1()) + min;
            arr[i][j+2] = (max - (min)) * (genrand_real1()) + min;
            arr[i][j+3] = (max - (min)) * (genrand_real1()) + min;
        }
        for (; j < col; j++) {
            arr[i][j] = (max - (min)) * (genrand_real1()) + min;
        }
    }
```

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

Unroll Amount	0	1	3
Averages	3.5787	4.034133	4.489267

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.

```
#pragma omp parallel for num_threads(4)
for ( i = 0; i < row; i+=4) {
    results[i] = schwefel(arr[i], col);
    results[i+1] = schwefel(arr[i+1], col);
    results[i+2] = schwefel(arr[i+2], col);
    results[i+3] = schwefel(arr[i+3], col);
}
for (; i < row; i++) {
    results[i] = schwefel(arr[i], col);
}
```

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

Function	0	1	3	4	9
Schwefel	13.28073333	15.9573	14.42803	14.2286	14.44727
Sine envelope	71.9916	82.691	78.953	93.01197	79.7257

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.

```
#pragma omp parallel for private(k, j) num_threads(4)
        for (k = 0; k < NS; k++) {
            distance(myDA, myData, i, k, DIM);
            if (lessR(myDA, DIM)) {
                index++;
                myDA->numNeighbors++;
                for (j = 0; j < DIM; j+=4) {
                    myDA->neighborsPop[index][j] = myData->population[k][j];
                    myDA->neighborsStep[index][j] = myDA->step[k][j];
                    myDA->neighborsPop[index][j+1] = myData->population[k][j+1];
                    myDA->neighborsStep[index][j+1] = myDA->step[k][j+1];
                    myDA->neighborsPop[index][j+2] = myData->population[k][j+2];
                    myDA->neighborsStep[index][j+2] = myDA->step[k][j+2];
                    myDA->neighborsPop[index][j+3] = myData->population[k][j+3];
                    myDA->neighborsStep[index][j+3] = myDA->step[k][j+3];
                }
                for (; j < DIM; ++j) {
                    myDA->neighborsPop[index][j] = myData->population[k][j];
                    myDA->neighborsStep[index][j] = myDA->step[k][j];
                }
            }
        }
```

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

Iteration	0	1	3
1	10.969	5.981	6.982
2	6.983	4.987	6.499
3	5.983	6.982	6.983
4	6.981	7.978	7.978
5	9.973	8.976	5.984
6	5.984	7.979	6.981
7	11.97	8.976	6.98
8	11.967	7.979	9.974
9	5.983	6.982	5.984
10	6.983	5.982	4.986
11	5.982	5.985	8.978
12	4.987	10.975	4.986
13	4.987	12.961	5.985
14	9.973	8.98	6.98
15	13.963	10.966	12.967
16	5.492	6.981	4.985
17	9.973	8.977	4.986
18	7.977	5.983	5.988
19	12.984	6.982	4.984
20	4.979	7.981	4.986
21	6.983	6.98	5.985
22	6.98	10.97	4.986
23	4.987	8.975	4.987
24	7.979	4.981	5.986
25	6.981	7.98	5.983
26	9.973	9.972	4.985
27	5.985	73.803	13.963
28	11.967	7.979	17.951
29	5.984	8.976	5.985
30	8.977	7.978	4.987
Averages	8.0623	10.27223	7.0318

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.

```
#pragma omp parallel for num_threads(4)
    for (t = 0; t < DIM; t += 2) {
        myData->population[i][t] = myData->population[i][t] + levyFlight(DIM) *
myData->population[i][t];
        myDA->step[i][t] = 0;
        // if the new position is outside the range of
        // the bounds, then make it equal to the bounds
        checkBounds(myData, myData->population[i][t]);
        myData->population[i][t + 1] = myData->population[i][t + 1] + levyFlight(DIM)
* myData->population[i][t + 1];
        myDA->step[i][t + 1] = 0;
        // if the new position is outside the range of
        // the bounds, then make it equal to the bounds
        checkBounds(myData, myData->population[i][t + 1]);
        myData->population[i][t + 2] = myData->population[i][t + 2] + levyFlight(DIM)
* myData->population[i][t + 2];
        myDA->step[i][t + 2] = 0;
        // if the new position is outside the range of
        // the bounds, then make it equal to the bounds
        checkBounds(myData, myData->population[i][t + 2]);
        myData->population[i][t + 3] = myData->population[i][t + 3] + levyFlight(DIM)
* myData->population[i][t + 3];
        myDA->step[i][t + 3] = 0;
        // if the new position is outside the range of
        // the bounds, then make it equal to the bounds
        checkBounds(myData, myData->population[i][t + 3]);
    }
    for (; t < DIM; t++) {
        myData->population[i][t] = myData->population[i][t] + levyFlight(DIM) *
myData->population[i][t];
        myDA->step[i][t] = 0;
        // if the new position is outside the range of
        // the bounds, then make it equal to the bounds
        checkBounds(myData, myData->population[i][t]);
    }
```

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

Iteration	0	1	3
1	5.985	9.974	4.986
2	4.986	9.973	5.986
3	5.984	8.976	4.985
4	6.981	9.973	4.987
5	6.982	5.984	5.984
6	6.982	9.482	5.492
7	4.986	9.973	4.987
8	9.974	9.973	5.985
9	5.983	9.974	7.978
10	6.981	7.978	6.981
11	5.984	9.974	4.986
12	4.987	9.973	4.987
13	5.984	5.984	5.985
14	5.984	4.986	5.984
15	4.986	5.984	4.986
16	6.983	7.979	5.985
17	5.983	4.986	4.985
18	4.986	4.988	4.987
19	5.985	7.978	6.982
20	5.984	8.976	4.986
21	5.984	9.974	4.987
22	5.495	8.976	5.984
23	5.983	7.978	5.499
24	6.982	4.986	4.98
25	4.986	4.988	4.987
26	7.979	7.979	4.987
27	6.981	16.954	5.984
28	4.987	9.973	4.986
29	5.984	16.956	5.984
30	4.986	10.97	5.985
Averages	6.1339	8.7934	5.585567

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.

Percent Differences

The percent difference was calculated for the different parallelized functions. The first function is the fillIn method. The following table shows percent difference for a 10000x1000 matrix.

Table 18: Execution time percentage difference for a 10000x1000 matrix for the fillIn method

sequential	1 threads	percent difference
208.7792	204.9492	1.83%

The following table shows the percent difference in the time execution for the findNeighbors function. The time execution is taken from a 10000x1000 matrix.

Table 19: Execution time percentage difference for a 10000x1000 matrix for the findNeighbors method

sequential	4 threads	percent difference
1337.433	341.6	74.46%

The following table shows the percent difference in the time execution for the selectFunctions method. The unlooping version was used for this calculation. The time execution is taken from a 10000x1000 matrix for both schwefel and sine envelope.

Table 20: Execution time percentage difference for a 10000x1000 matrix for the selectFunctions method

schwefel	sequential	4 threads	percent difference
schwefel	985	265	73.10%
sineEv	sequential	4 threads	percent difference
sineEv	5626	1413	74.88%

The last table shows the percent difference in the time execution for the randomWalk function. The unlooping version for randomWalk was not used because there were data dependencies within that method that created memory allocation issues. The time execution is taken from a 10000x1000 matrix.

Table 21: Execution time percentage difference for a 10000x1000 matrix for the randomWalk method

sequential	4 threads	percent difference
103	26	74.76%

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.