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.

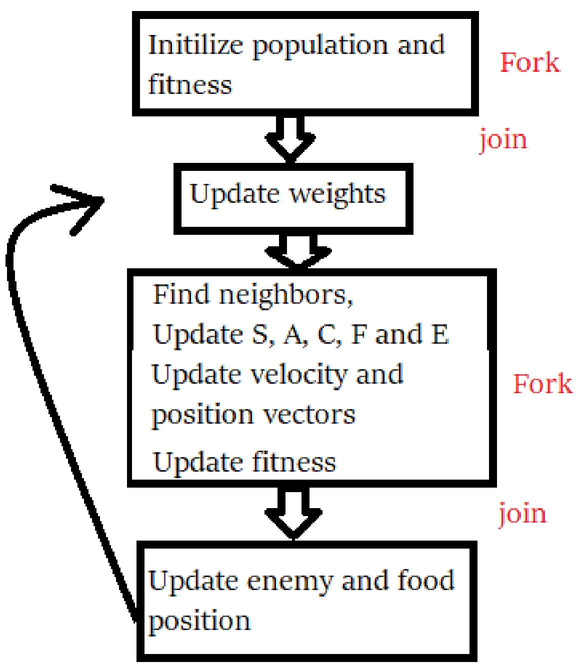

Figure 1. Dragonfly Optimization fork & join diagram

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
Graphics	NVIDIA GeForce GTX 1050 Ti
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

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.

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.