CS 495 Spring

Parallelizing DA

Using OpenMP

Emily Bodenhamer

Dr. Donald Davendra

Table of Contents

Table of Figures

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.

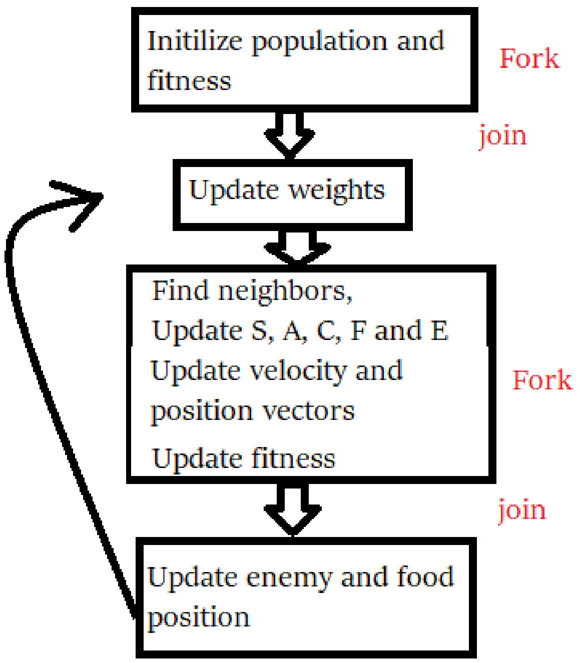
Initilize population and fitness
Fork
join
Update weights
Find neighbors,
Update S, A, C, F and E
Update velocity and position vectors
Update fitness
Fork
join
Update enemy and food position

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.

Conclusion

There are many different pragmas that the OpenMP API has. There may be different pragmas that are better to use than the ones used in this documentation. OpenMP encourages incremental parallelization so changing pragmas around will not be difficult if the pragmas chosen to use do not produce optimal results.