Parallel Programming HW7 Write Up
Due Thursday March 25, 2021, 11:59 pm

Connor Osborne
A01880782

Monday March 22, 2021

1 Description

For this assignment I used MPI to solve the traveling salesman problem for 100 prescribed "cities", given their coordinates, in parallel using a genetic algorithm styled approach.

2 Implementation

The implementation for this code follows a master and slave process dynamic where process 0 receives the best results from each process and reports on them while every other process starts out with two randomly shuffled versions of the original list of "cities". Process rank 0 first checks to see if a termination signal has been sent from the other processes when they have finished a set number of iterations. If no signal is received it will begin waiting to receive results for the other processes. It will then use these results to update the best result found and what time it was found. Process 0 also stores all found results to a results.txt file to know what results were found and how long it took to find them.

```
if (rank == 0){
    std::chrono::high_resolution_clock::time_point t1 =
        std::chrono::high_resolution_clock::now();
    ofstream fout;
    fout.open("results.txt");
    int original = calculatePathLength(cities);
    double time;
    int bestResult = 2147483647;
    double bestTime;
    while(1){
        int terminateFlag;
        MPI_Iprobe(MPI_ANY_SOURCE, 0, MCW, &terminateFlag, &myStatus);
        if(terminateFlag){
            MPI_Recv(&terminate, 1, MPI_C_BOOL, MPI_ANY_SOURCE, 0, MCW, &myStatus);
        }
        if(terminate){
            fout.close();
            cout << "The original path length was " << original << endl;
            cout << "The shortest distance found was " << bestResult << " found in " << bestTime
                << " seconds." << endl;
            break;
        }

        int resultFlag;
        MPI_Iprobe(MPI_ANY_SOURCE, 1, MCW, &resultFlag, &myStatus);
        while(resultFlag){
```

```
            MPI_Recv(&result, 1, MPI_INT, MPI_ANY_SOURCE, 1, MCW, MPI_STATUS_IGNORE);
            // cout << "printing to file" << endl;
            std::chrono::high_resolution_clock::time_point t2 =
                std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> time_span =
                std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
            time = time_span.count();
            if(result < bestResult){
                bestResult = result;
                bestTime = time;
            }
            fout << result << " found in " << time << " seconds." << endl;
            MPI_Iprobe(MPI_ANY_SOURCE, 1, MCW, &resultFlag, &myStatus);
        }
    }
}
```

All other processes will make a 3 dimensional array using the 2d array of cities and a randomly shuffled second "parent" to use in the genetic algorithm. Once both "parents" are accounted for the processes enter a while loop that lasts until the predetermined set of iterations is completed. This loop starts with the creation of 2 "child" lists that start as copies of each "parent" respectively before being mutated by randomly picking a 50 element wide section to compare the parents to eachother. At each element in the overlapping section the "city's" number is checked and then used to swap the values in each child at the indexes gleaned from the "city" numbers.

```
else{
    int iterations = 1000;
    int parents[2][100][3];
    int parent[100][3];
    for (int j = 0; j < 100; ++j){
        for (int k = 0; k < 3; ++k){
            parent[j][k] = cities[j][k];
            parents[0][j][k] = cities[j][k];
        }
    }
    random_shuffle(begin(parent), end(parent));
    for (int j = 0; j < 100; ++j){
        for (int k = 0; k < 3; ++k){
            parents[1][j][k] = parent[j][k];
        }
    }
    while(iterations > 0){
        int childA[100][3];
        int childB[100][3];
        for (int i = 0; i < 100; ++i){
            for(int j = 0; j < 3; ++j){
                childA[i][j] = parents[0][i][j];
                childB[i][j] = parents[1][i][j];
            }
        }
        int index = rand() % 50;
        for (int i = 0; i < index + 50; ++i){
            int indexA = parents[0][i][0];
            int indexB = parents[1][i][0];
            swap(childA[indexA - 1], childA[indexB -1]);
            swap(childB[indexA - 1], childB[indexB -1]);
        }
```

Once the "family" of solutions is set the overall distance of each solution's path is calculated. Then a series of if statements is used to check which "family member" solution has the shortest path. That solution is reported to process 0 for record keeping and checking to see if it is the new best solution. Once the number of iterations ends process 0 sends the termination code to process 0.

```
while(iterations > 0){
        int childA[100][3];
        int childB[100][3];
        for (int i = 0; i < 100; ++i){
            for(int j = 0; j < 3; ++j){
                childA[i][j] = parents[0][i][j];
                childB[i][j] = parents[1][i][j];
            }
        }
        int index = rand() % 50;
        for (int i = 0; i < index + 50; ++i){
            int indexA = parents[0][i][0];
            int indexB = parents[1][i][0];
            swap(childA[indexA - 1], childA[indexB -1]);
            swap(childB[indexA - 1], childB[indexB -1]);
        }

        int parentADistance = calculatePathLength(parents[0]);
        int parentBDistance = calculatePathLength(parents[1]);
        int childADistance = calculatePathLength(childA);
        int childBDistance = calculatePathLength(childB);

        if (childADistance <= childBDistance){
            if(childADistance <= parentADistance || childADistance <= parentBDistance){
                if(parentADistance <= parentBDistance && childADistance <= parentBDistance){
                    for (int i = 0; i < 100; ++i){
                        for(int j = 0; j < 3; ++j){
                            parents[1][i][j] = childA[i][j];
                        }
                    }
                    if(childADistance < parentADistance){
                        MPI_Send(&childADistance, 1, MPI_INT, 0, 1, MCW);
                    }
                    else{
                        MPI_Send(&parentADistance, 1, MPI_INT, 0, 1, MCW);
                    }
                }
                else if(parentBDistance <= parentADistance && childADistance <= parentADistance){
                    for (int i = 0; i < 100; ++i){
                        for(int j = 0; j < 3; ++j){
                            parents[0][i][j] = childA[i][j];
                        }
                    }
                    if(childADistance < parentBDistance){
                        MPI_Send(&childADistance, 1, MPI_INT, 0, 1, MCW);
                    }
                    else{
                        MPI_Send(&parentBDistance, 1, MPI_INT, 0, 1, MCW);
                    }
                }
            }
            else{
                if(parentADistance < parentBDistance){
                    MPI_Send(&parentADistance, 1, MPI_INT, 0, 1, MCW);
```

```
                }
                else{
                    MPI_Send(&parentBDistance, 1, MPI_INT, 0, 1, MCW);
                }
            }
        }
        else{
            if(childBDistance <= parentADistance || childBDistance <= parentBDistance){
                if(parentADistance <= parentBDistance && childBDistance <= parentBDistance){
                    for (int i = 0; i < 100; ++i){
                        for(int j = 0; j < 3; ++j){
                            parents[1][i][j] = childB[i][j];
                        }
                    }
                    if(childBDistance < parentADistance){
                        MPI_Send(&childBDistance, 1, MPI_INT, 0, 1, MCW);
                    }
                    else{
                        MPI_Send(&parentADistance, 1, MPI_INT, 0, 1, MCW);
                    }
                }
                else if(parentBDistance <= parentADistance && childBDistance <= parentADistance){
                    for (int i = 0; i < 100; ++i){
                        for(int j = 0; j < 3; ++j){
                            parents[0][i][j] = childB[i][j];
                        }
                    }
                    if(childBDistance < parentBDistance){
                        MPI_Send(&childBDistance, 1, MPI_INT, 0, 1, MCW);
                    }
                    else{
                        MPI_Send(&parentBDistance, 1, MPI_INT, 0, 1, MCW);
                    }
                }
                else{
                    if(parentADistance < parentBDistance){
                        MPI_Send(&parentADistance, 1, MPI_INT, 0, 1, MCW);
                    }
                    else{
                        MPI_Send(&parentBDistance, 1, MPI_INT, 0, 1, MCW);
                    }
                }
            }
            iterations -= 1;
                if(iterations <= 0){
                    terminate = true;
                    sleep(3);
                    if(rank == 1){
                        MPI_Send(&terminate, 1, MPI_C_BOOL, 0, 0, MCW);
                    }
                }
        }
    }

    MPI_Finalize();
    return 0;
}
```

No graph was made to report findings as the results were too sporadic and relied too heavily on randomly getting a good path to be reliable results. However it is noted that the more processes and iterations that were used the more likely a better path was to be found though this does not occur 100% of the time.

---

3 Example Input and Output with 4, 8, 16, and 32 Processes at 1000 iterations

Compile command: mpic++ traveler.cpp

- Input:
  mpirun -np 4 -oversubscribe a.out

  Output:
  The original path length was 10978991
  The shortest distance found was 10649691 found in 0.0007255 seconds.

- Input:
  mpirun -np 8 -oversubscribe a.out

  Output:
  The original path length was 10978991
  The shortest distance found was 10678591 found in 0.000997 seconds.

- Input:
  mpirun -np 16 -oversubscribe a.out

  Output:
  The original path length was 10978991
  The shortest distance found was 10701027 found in 0.0332894 seconds.

- Input:
  mpirun -np 32 -oversubscribe a.out

  Output:
  The original path length was 10978991
  The shortest distance found was 10271541 found in 0.0166039 seconds.