Parallel and Distributed Computing

Project Report

Group Members:

Syed Ahmad Mustafa 21I-0886

Saman Ali Ahmed 21I-2499

Syed Ata-ul Muhaimen 21I-0888

Project Summary

- **Setup:**

  The following experiment was conducted to test the effectiveness of parallelism specifically using OPENMP and MPI methods. A .csv file named “doctorwho.csv” containing a dataset related to cities and all available paths between those cities was used in this experiment.

  The experiment was based on finding the **“k”** shortest possible paths between a source and destination pair. The code pathway involved preprocessing steps like reading data from file followed by creating and initializing an adjacency matrix based on that data. The adjacency matrix was then used to calculate **“k”** shortest paths between any two random nodes of the adjacency matrix by using a modified version of **Dijkstra** algorithm.

- **Observations:**

  Serialized version:

  For the given dataset, the average computational time for the given experiment (excluding the time consumed in preprocessing) was observed to be between **15-20 seconds.**

  Parallel version:

  When parallelization strategies were applied using OPENMP and MPI libraries the computational time of the parallelized code was significantly reduced and was obtained to be in the range of **0.1-2 seconds.**

- **Deductions:**

  Based on the observations, we can deduce that parallelization indeed helps in decreasing computational time significantly and is an efficient and effective way of coding procedures that have the tendency to be executed independently in parallel.

Code Analysis:

- **dataSetSize():**
  - A counter is initiated and is incremented simultaneously as the data file is read line by line. This function returns the total number of lines the data file contains.
- **readFile():**
  - A 3D dimensional array with 4 columns is created that retrieves data from the file.
    - First column stores the source city.
    - Second column stores the destination city.
    - The third column stores the weight of the path.
    - The fourth column stores the type of path i.e. directed/undirected.

    This array is then returned to the main function.
- **distanceMatrix():**
  - This function uses the dataNode array containing the data from the file and initializes the adjacency matrix.
- **shortestPath():**

- Implements a modified version of the Dijkistra path and returns the shortest possible path between two nodes.

- **kMinPaths():**
  - This function returns the **k** minimum paths between two nodes. This is done by implementing the following steps.
    - First the shortest path is calculated directly from the source node to the destination node.
    - It is then followed by changing our source node to the adjacent neighbors (destinations) of our original source. The code then performs 3 important steps.
      - First it computes the shortest distance between our original source and the new source.
      - It then calculates the shortest distance between our new source and destination. If there is no direct path between our new source and destination, we will select the path with the shortest distance, and use the destination of this path as our source for step 3.
      - Lastly, the new destination of step 2 (if the original one is not yet reached) is used as source, and the original destination as the destination.

    The three costs are combined to form the kth shortest path between a source and destination.

Parallelization Strategy:

As the project requirements demanded the code to be executed for 5 random source, destination pairs from a dataset, it demonstrated a case of "embarrassingly parallel programs".

MPI Library:

MPI library was used in parallelizing the code. As calculating **k** shortest paths for a source and destination pair is independent of the same implementation for other source/destination pairs, therefore we planned to initialize **five** processes, each finding **k** shortest paths for one of the 5 pair of nodes.

To increase efficiency, we decided that only the **root** process must read the data file and initialize an adjacency matrix for it, followed by broadcasting the adjacency matrix to all of the remaining (4) processes. This measure helped in reducing the time consumed at the

preprocessing stage. The root process is also responsible for broadcasting the number of distinct cities present in the data (through the variable “uniqueNodes”).

OPENMPI:

We decided to implement OPENMPI based parallelism in kMinPaths() and shortestPath functions respectively. OPENMPI was specifically used to parallelize all the distinct paths between a source and destination. We decided to use fine-grained data, by parallelizing each path/iteration as a separate process. This was done using the “#pragma omp parallel for” command before each iteration of all the loops present in the above-mentioned functions.

We used “#pragma omp task” command to specify the three steps of kMinPath() function (explained under the kMInPath() heading of the code analysis section), so that they may be executed simultaneously, to cut-short our computational time, as we would otherwise had to wait for the first call of the shortestPath() function to finish execution, so that we could move to step 2 and 3 respectively.

After the simultaneous execution of our tasks we used “#pragma omp barrier” to collect all the costs before it is returned as total cost.

Lastly, we used “#pragma omp critical” to lock the code snippet, used for the insertion of shortest path into our global array containing **k** shortest paths, to 1 process at a time.

Key Insights:

- Parallelization significantly reduces computational time for processes that contain independent parallelizable code sections.
- OPENMP races the execution of iterations of a loop and executes them simultaneously, thus reducing overall execution time.
- Using mpi to divide and then map code as different processes is an effective strategy and significantly reduces execution time, as demonstrated in the above experiment, where we created a **SIMD** model, where although the adjacency matrix was same, but the source and destination nodes for each process was different, whereas the instructions for each concurrent process were the same. In conclusion, using mpi parallelization for such models demonstrates the true essence of parallelization and its benefits.