HPC Tutorial 7 Report

CS22B2012
K Aditya Sai

The code for Parallelized State Space Search is implemented in C with OpenMP.

1. **Thread Allocation**: The code is executed with thread counts ranging from 1 to 64

2. **Game Board** : The program initializes a grid of size 5x5 with the target value as 16384.

3. **Performance Analysis**: Execution times for the multiplication are recorded and speedup/parallelization fractions are calculated.

4. **Visualization**: Python scripts generate plots for execution time and speedup.

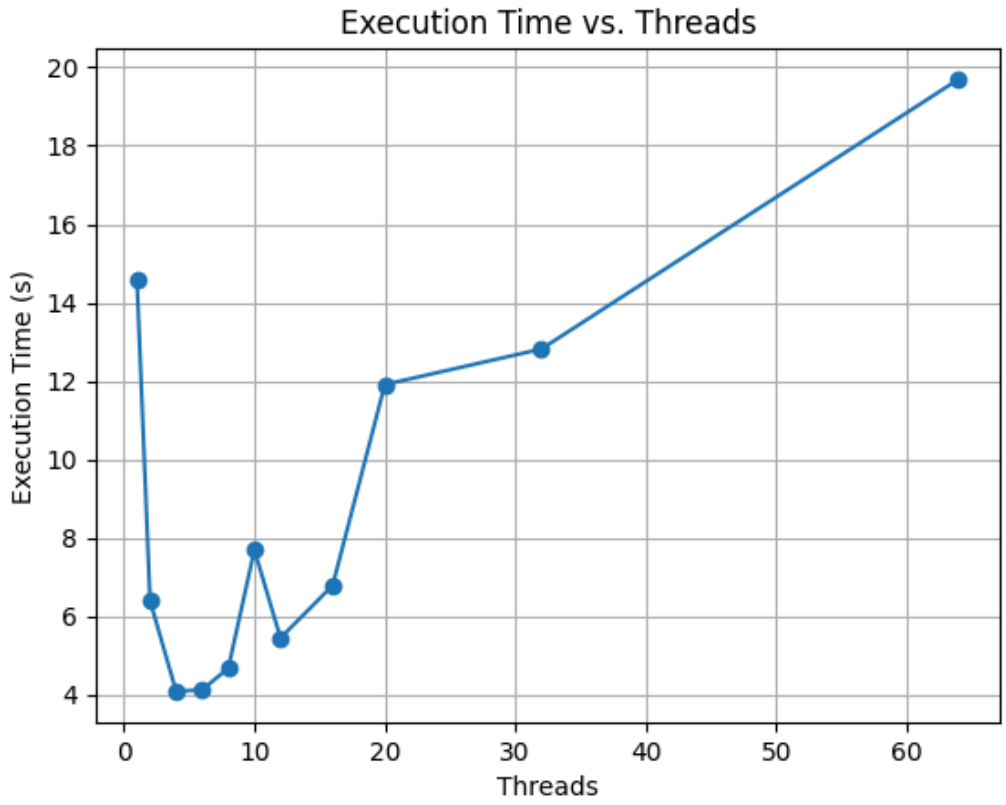
5. **Scalability :** To increase the complexity and execution time of the problem, one may simply increase the grid size and target value. This results in multiple possible next states and also multiple heuristic comparisons.

6. The number of iterations taken to complete the problem is not always constant, thanks to the random generation of the number **2**, there is a good chance that the number of iterations is never constant.

7. **Heuristic :** There is the use of 3 different heuristics in the program, to determine the best possible next state.

- Sum : The first heuristic is the sum of all the elements in the grid. The greater the sum, the higher the chance of reaching the goal soon.
- LogMax : The logarithm of the max value of the grid.
- LogSecondMax : The logarithm of the second max value of the grid.

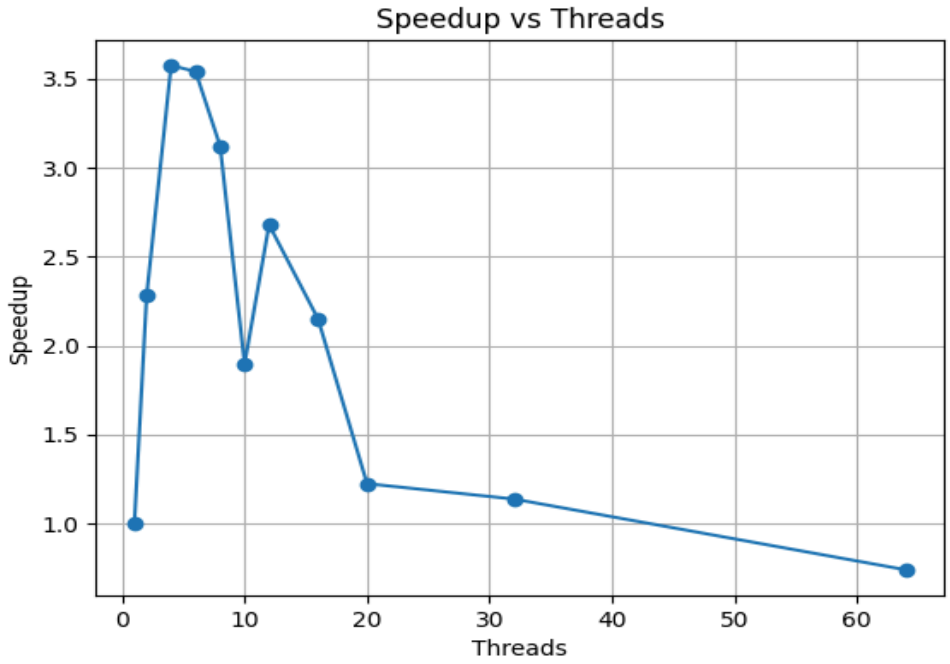
Plot Threads v/s Time :



Observations :

1. When increasing thread count from 2 to 12, execution time decreases till 6 threads and then has slight fluctuations and then, after 12 threads, we see a sudden increase in the execution time.
2. The sharp reduction in the execution time shows that the parallelization is effective up-to 32 threads, and beyond that the thread context switch overhead is dominant and starts to increase the execution time.
3. The execution time is also dependent on how the grid is initialized and how the random generation of **2** occurs.
4. The generation of **2** in different grid cells may affect the program positively or negatively depending on how beneficial it is to generate a **2** in that grid cell.

Plot Threads v/s Speedup:



Observations :

1. The sudden spike in the overall speedup from 2 thread to 16 threads shows that for the system on which this was performed, 6 threads is the most optimal.
2. The speedup has a smooth descent from 32 to 64 threads indicating that beyond 32 threads, parallelization is disadvantageous.
3. The parallel construct seems to efficiently parallelize and reduce the execution time up-to 32 threads after which the CPU and thread management overheads are expected to negatively affect the speedup.

Estimated Parallelization Fraction :

```
=== Parallelization Fraction Table ===
   Threads       Parallelization Fraction
   -------      -------------------------
         1                      0.000000
         2                      1.122753
         4                      0.960280
         6                      0.861003
         8                      0.776737
        10                      0.525226
        12                      0.683702
        16                      0.570402
        20                      0.192359
        32                      0.124466
        64                     -0.357924
```

Observations :

1. The parallelization fraction reaches its peak at 4 threads, maintaining similar PF till 8 threads.
2. The parallelization fraction does drop slightly until 16 threads and then suffers a massive drop.
3. Up to 16 threads, the program maintains a parallelization fraction (≥ 0.5).

Inferences :

1. Beyond 32 threads, the parallelization fraction starts to go below zero indicating an increase in execution time compared to when number of threads is 1.
2. Up-to 16 threads a major part of the program is parallelized effectively.
3. After 16 threads, the parallelization fraction drops sharply indicating the effect of thread context switch overhead.

Code Snippet : Parallelized part :

The below code is the parallelized version of the code which is considered the hotspot in **gprof.**

```
int generateNextStates(GameState currentState, GameState *nextStates)
{
    int validMoves = 0;
#pragma omp parallel
    {
        GameState localNextStates[4];
        int localValidMoves = 0;

#pragma omp for
        for (int move = 0; move < 4; move++)
        {
            int movedGrid[GRID_SIZE][GRID_SIZE];
            switch (move)
            {
            case 0:
                moveLeft(currentState.grid, movedGrid);
                break;
            case 1:
                moveRight(currentState.grid, movedGrid);
                break;
            case 2:
                moveUp(currentState.grid, movedGrid);
                break;
            case 3:
                moveDown(currentState.grid, movedGrid);
                break;
            }

            int changed = 0;
            int m = 0, k = 0;
            int h = 0;

#pragma omp parallel for collapse(2) reduction(+ : h) reduction(max : m, k)
            for (int i = 0; i < GRID_SIZE; i++)
            {
                for (int j = 0; j < GRID_SIZE; j++)
                {
                    if (movedGrid[i][j] != currentState.grid[i][j])
                    {
```

```
                        changed = 1;
                    }
                    h += movedGrid[i][j];
                    if (movedGrid[i][j] > m)
                    {
                        k = m;
                        m = movedGrid[i][j];
                    }
                    else if (movedGrid[i][j] > k)
                    {
                        k = movedGrid[i][j];
                    }
                }
            }

            if (changed)
            {
                randomGenerate(movedGrid);

                double logM = (m > 0) ? log2((double)m) : 0.0;
                double logK = (k > 0) ? log2((double)k) : 0.0;

                GameState newState;
                initializeGameState(&newState, movedGrid);
                newState.gCurr = currentState.gCurr + 1;
                newState.heuristicScore = h;
                newState.logMax = logM;
                newState.logSecondMax = logK;

                localNextStates[localValidMoves++] = newState;
            }
        }

        #pragma omp critical
        {
            memcpy(&nextStates[validMoves], localNextStates, localValidMoves *
sizeof(GameState));
            validMoves += localValidMoves;
        }
    }
    return validMoves;
}
```

```
void randomGenerate(int grid[GRID_SIZE][GRID_SIZE])
{
    int emptyTiles[GRID_SIZE * GRID_SIZE][2];
    int emptyCount = 0;

#pragma omp parallel
    {
        int localEmptyTiles[GRID_SIZE * GRID_SIZE][2];
        int localEmptyCount = 0;

#pragma omp for collapse(2)
        for (int i = 0; i < GRID_SIZE; i++)
        {
            for (int j = 0; j < GRID_SIZE; j++)
            {
                if (grid[i][j] == 0)
                {
                    localEmptyTiles[localEmptyCount][0] = i;
                    localEmptyTiles[localEmptyCount][1] = j;
                    localEmptyCount++;
                }
            }
        }

        // #pragma omp critical
        {
            memcpy(&emptyTiles[emptyCount], localEmptyTiles, localEmptyCount * sizeof(emptyTiles[0]));
            emptyCount += localEmptyCount;
        }
    }

    if (emptyCount > 0)
    {
        int randomIndex = rand() % emptyCount;
        int x = emptyTiles[randomIndex][0];
        int y = emptyTiles[randomIndex][1];
        grid[x][y] = (rand() % 10 == 0) ? 4 : 2;
    }
}
```

Conclusion:

- The optimal number of threads for this workload appears to be around **6-8**.
- Beyond this point, there is an improvement compared to serial code.
- After 32 threads there will be a steady reduction in the performance of the code.
- We could imply that the randomness involved in the code may also be a deciding factor in the performance of the code as we may not always find an optimal solution due the inconsistency of the grid in different repetitions of the program.

The full code has been attached along with the report.