CS433A : Report | Design Exercise 1

> *Submitted on : March 10, 2022 by Group No. 24*
>
> *Dipanshu Garg | 190306*
>
> *Gurbaaz Singh Nandra | 190349*

Question 1

Program Specifications

- Language : `C++`
- Filename: `Q1.cpp`
- Library : `OpenMP directives`

Algorithm

For the TSP problem, it is well known that the brute force solution of iterating through each permutation takes time of order `O(n!)`, where `n` is the number of vertices. We, on the other hand, implemented Dynamic Programming solution (with bit-masking) to solve this problem, which runs in `O(n^2 2^n)` asymptotically, which has clearly better time complexity over the brute force approach. The pseudo code of the unparalleled DP solution can be summarized as follows:

Let `dp(i, j)` be the minimum cost to travel from vertex `1` to vertex `j`, covering all the vertices represented by bit-mask `i` (`i` will always contain `1` and `j`) . Hence we need to minimise value of `dp(2^n - 1, j)` for some `2 <= j <= n`.

```
for i = 1 to 2^n- 1
    for j = 1 to n
        dp(i, j) = inf

dp(1, 1) = 0

for subset_size = 2 to n do
// L1
   for all bitmasks with number of set bits = subset_size and least significant
bit is set      // L2
       for all set bits j in mask and j ≠ least significant bit
// L3
          dp(S, j) = min {dp(S - 2^j, i) + d(i, j) for some ith bit which is set
and i ≠ j}     // L4

Return min dp(2^n - 1, j) + d(j, i) among all j such that 2 <= j <= n
// L5
```

Parallelization of Algorithm

First and foremost, we observed that outermost loop `L1` cannot be parallelized. This is because each computation on a bit-mask with `k` bits set depends on computation of bit-mask with `k-1` bits set.

On careful analysis, we observe that the inner loop `L2`) and all the nested loops (`L3` and `L4`) can be parallelised as each iteration of `L2` is indeed an independent computation on a particular subset bit-mask. This provides a great oppurtunity for parallelisation. We also observed that each iteration of `L2` does the same amount of work, hence static assignment was adopted. We also chose block decomposition as that would (along with some optimisations stated below) would help in efficient cache usage.

Parallel Algorithm 1

```
for i = 1 to 2^n- 1
    for j = 1 to n
        dp(i, j) = inf

dp(1, 1) = 0

for subset_size = 2 to n do
// L1
#pragma omp parallel for num_threads (nThreads)
   for all bitmasks with number of set bits = subset_size and least significant
bit is set      // L2
       for all set bits j in mask and j ≠ least significant bit
// L3
          dp(S, j) = min {dp(S - 2^j, i) + d(i, j) for some ith bit which is set
and i ≠ j}     // L4

Return min dp(2^n - 1, j) + d(j, i) among all j such that 2 <= j <= n
// L5
```

This program works well, but can be optimised further. We can save the time of threads creation and destruction and instead create them once.

Parallel Algorithm 2

```
for i = 1 to 2^n- 1
    for j = 1 to n
        dp(i, j) = inf

dp(1, 1) = 0

#pragma omp parallel num_threads (nThreads)
for subset_size = 2 to n do
// L1
#pragma omp for
   for all bitmasks with number of set bits = subset_size and least significant
bit is set      // L2
       for all set bits j in mask and j ≠ least significant bit
// L3
          dp(S, j) = min {dp(S - 2^j, i) + d(i, j) for some ith bit which is set
and i ≠ j}     // L4

Return min dp(2^n - 1, j) + d(j, i) among all j such that 2 <= j <= n
// L5
```

This algorithm takes care of unnecessary threads creation and destruction.

Optimization (if any)

We observed that we can further improve the algorithm by pre-grouping the bit-masks by the number of set bits. The pre-grouping computation in our code looks like as follows:

```
for(int i = 1; i < m; i += 2) {
    int size = __builtin_popcount(i);
    mask[size].push_back(i);
}
```

This optimisation also had an additional benefit of cache optimisation with block decomposition, each thread will access contiguous chunk of masks. That will have great cache hit-rate, further helping in speedup.

Additional approaches and thoughts

1. We saw an oppurtunity to further parallelise the pregrouping and came up with the following code:

   ```
   #pragma omp parallel for num_threads (nThreads)
   for(int i = 1; i < m; i += 2) {
       int size = __builtin_popcount(i);
   #pragma omp critical
       mask[size].push_back(i);
   }
   ```

   But this does not gave us any significant speedup due to the overhead of critical section.

2. We also tried to parallelize the final result computation of minimum distance as follows:

   ```
   int ans = INT_MAX, st = -1;
   #pragma omp parallel num_threads (nThreads)
   for(int i = 1; i < n; i++) {
       if(dp[m - 1][i] == INT_MAX) continue;
       if(dp[m - 1][i] + d[i][0] < ans) {
   #pragma omp critical
           if(dp[m - 1][i] + d[i][0] < ans) {
               ans = dp[m - 1][i] + d[i][0];
               st = i;
           }
       }
   }
   ```

   This also did not give us any signficant speedup, for similar reason of critical section overhead as mentioned above and also the fact that computation here was very less (of `O(n)`), which is significantly less than exponential computations done above.

Performance Results

We tested on graph of different sizes, ranging from number of vertices `19` to `23`. Following are the results of variation of time with thread count. Note that we tested these programs on an `8`-core machine.

Number of Vertices: `19`

Time (in microseconds)	Thread Count
267185	1
140463	2
75453	4
59860	8

Number of Vertices: `20`

Time (in microseconds)	Thread Count
585370	1
302298	2
165152	4
137464	8

Number of Vertices: `21`

Time (in microseconds)	Thread Count
1326633	1
682560	2
386822	4
311086	8

Number of Vertices: `22`

Time (in microseconds)	Thread Count
2893789	1
1546334	2
888111	4
729689	8

Number of Vertices: `23`

Time (in microseconds)	Thread Count
6393244	1
3562141	2
2065366	4
1711146	8

Observed Trends

We observe nearly ideal speedup with `2` threads and `4` threads. There is further speedup with `8` threads, but then saturation is observed. This is because we have initially good parallelisation of computation, and as the thread count increases beyond `8`, the overhead of thread communication due to false sharing and also due to overhead of context switches start to increase and the speedup gets compromised and saturates.

Question 2

Program Specifications

- Language : `C++`
- Filename: `Q2.cpp`
- Library : `OpenMP directives`

Algorithm

To solve a lower triangular system of equations `Lx = y`, we used the forward substitution algorithm, stated as follows:

$$x_i = \frac{y_i - \sum_{j=1}^{i-1} L_{ij} x_j}{L_{ii}} \quad \text{, where } 1 \leq i \leq n$$

Simply put, we can easily see that `x[0] = y[0]/L[0][0]`, and then can use the value of `x[0]` to calculate `x[1]` using multiplication of second row of `L` with `x`, and so on. We can write unparalleled code for the same as:

```
for(int i = 0; i < n; i++) {                // L1
    for(int j = 0; j < i; j++) {            // L2
        y[i] -= (L[i][j] * x[j]);
    }
    x[i] = y[i] / L[i][i];                  // a3
}
```

Time complexity for the algorithm is `O(n^2)`.

Parallelization of Algorithm

We observe that the outermost loop `L1` cannot be parallelised since all the values of `x[j], (j<i)` need to be known to compute the result for `x[i]`.

We then tried to parallelise the inner loop `L2` as all the computations of `L[i][j]*x[j]` inside loop `L2` are independent of each other. However it is important to notice that all these computations modify the same variable `y[i]`. In order to ensure correctness, we could make that computation atomic. However since `-=` could be computed using reduction, we chose to go with this since it is a better optimisation.

Since all the computations require the same computational resources, we chose to go with static assignment, and also, to maximise the cache benefit, we use block decomposition.

Parallel Algorithm 1

```
int j;

for(int i = 0; i < n; i++) {
#pragma omp parallel for num_threads (nThreads) reduction (-:y[i])
    for(j = 0; j < i; j++) {
        y[i] -= (L[i][j] * x[j]);
    }
    x[i] = y[i] / L[i][i];              // a3
}
```

This program works well, but can be optimised further. We can save the time of threads creation and destruction and instead create them once. But in order to that, we need to make sure to add a barrier after line `a3` since it will be executed by all the threads.

Parallel Algorithm 2

```
int j;

#pragma omp parallel num_threads (nThreads)
for(int i = 0; i < n; i++) {
#pragma omp for reduction (-:y[i])
    for(j = 0; j < i; j++) {
        y[i] -= (L[i][j] * x[j]);
    }
    x[i] = y[i] / L[i][i];                  // a3
#pragma omp barrier
}
```

We observe that the second algorithm indeed works better than the former as the cost of thread creation and destruction is minimised, which helps to more than compensate the overhead of introducing the barrier.

Optimization (if any)

We had 3 choices to handle the shared variable `y[i]` i.e. using

- critical section,
- atomic instruction, and
- reduction

We chose reduction since it offers the best speedup.

Additional approaches and thoughts

We also tried another way to parallelise the forward substitution algorithm, as follows:

```
#pragma omp parallel num_threads (nThreads)
    for(int j = 0; j < n; j++){
        x[j] = y[j] / L[j][j];
#pragma omp for
        for(int i = j+1; i < n; i++){
            y[i] -= L[j][j]*x[j];
        }
    }
```

This algorithm works by achieving parallelism in a column-wise fashion. Most significant advantage of using this is that now `y[i]` is no longer a shared variable and did not requre any reduction. However this algorithm had a significant downside, that the cache utilisation was very poor since we are iterating in a column-wise fasion.

On testing its performance, we found this algorithm takes roughly `10x` more time than our `Parallel Algorithm 2`. We can conclude that the cache overhead far exceeds the time saved by avoiding reduction.

Performance Results

Size of matrix: `5000`

Time (in microseconds)	Thread Count
36430	1
21929	2
16450	4
21992	8

Size of matrix: `8000`

Time (in microseconds)	Thread Count
88350	1
50573	2
35489	4
47495	8

Size of matrix: `10000`

Time (in microseconds)	Thread Count
139159	1
79990	2
51336	4
68757	8

Size of matrix: `15000`

Time (in microseconds)	Thread Count
310715	1
189892	2
109913	4
131849	8

Size of matrix: `20000`

Time (in microseconds)	Thread Count
552765	1
296699	2
188093	4
266955	8

Observed Trends

We observe a nearly ideal speedup with `2` threads. There is further speedup with `4` threads, but roughly only about `1.5x`. Beyond `4` threads on moving to `8` threads, the time starts to increase. This is because of the synchronisation overhead of the two barriers, once implicit after the loop `L2` and before line `a3`, and one explicit just after the line `a3`.