**PARALLEL COMPUTING MAJOR PROJECT**

**Parallel Implementation of Strassen’s Matrix Multiplication in OpenMP**

Student Name: Swarna Prabhu

UIN: 932003046

* Implementation of Strassen’s multiplication in parallelized approach:

In Strassen’s algorithm for matrix multiplication, the 2 matrices to be multiplied say A and B are initially broken into sub matrices of equal sized named A11, A12, A21, A22 and B11, B12, B21, B22 and then uses M1, M2, M3, M4, M5, M6, M7 for the computation of result matrix C where

M1 = (A11 + A22) (B11 + B22) , M2 = (A21 + A22 ) B11

M3 = A11 (B12 – B22)

M4 = A22 (B21 – B11)

M5 = (A11 + A12 )B22 M6 = (A21 – A11)(B11 + B12) M7 = (A12 – A22)(B21 + B22)

And then result matrix C becomes that involves 7 multiplication computations and 18 additions and becomes ideal approach for matrix multiplication involving larger matrix sizes particularly.

In this project the above algorithm is parallelized using OpenMp with shared memory approach as it provides the benefit of using explicit directives for synchronization and parallel region. It comes with implied barrier at the end of the parallel region.

In this project user has to input three parameters namely k , k’ and t- number of threads

k – is used to denote the size of the matrix by power of 2. For matrix of size n, n = 2^k.

k’ – is used to denote the size of the matrix by power of 2 till where recursive approach is used, that is up to matrix size s where s = 2^k’ recursive process is used and from s = 2^k’ to s= 1 standard multiplication is used.

t- 3rd parameter given as user input is the number of threads from where we can control to what extent we want the algorithm to be parallelized.

To analyze the parallel performance of plots of execution time vs no of threads, speed up vs no of threads and efficiency vs no of threads is used.

Formula used for speed up calculation – Execution time with single thread/ Execution time with t threads.

Formula for efficiency – Speed up/ Number of threads used.

**Parallel Performance of the Code:**

Below are the 2 major approaches used to analyze the parallel performance using k, k‘ and number of threads:

1. Fixed k’ and vary the matrix size by changing k from small value like k -4 to k – 8, 9, 10 and 11.
2. Fixed k and changing the value of terminal matrix k’ from k’-2 ,4,5 to k’-6
3. Since Strassen’s Algorithm involves the usage of recursive function calls for calculating M1, M2…M7 it is observed that greater parallelism or speed up was not obtained for smaller matrix size. Hence increasing the number of threads for k – 4 did not parallelize the algorithm well. For smaller matrix size it is seen that memory usage becomes too intensive and hence it becomes unideal to use for such size. In fact, there was a decrease in the speed up as there is no parallel performance with very less efficiency and the same is shown below:

Table

Description automatically generatedTable

Description automatically generated

Table

Description automatically generated

As seen, there is gradual decrease in speed up but the efficiency also decreases as it is calculated based on number of threads at each stage and is very less and not desirable.

In the next step, the parallel code was implemented on larger sizes of k such as k -8,9, 10, 11 by keeping fixed or similar range for k’(a smaller value of k’2 and then for k’6 ) and then a desired parallel performance was obtained when the number of threads was increased by a factor of 2 at every iteration from 2^0 to 2^ 6.

* **Speed up and efficiency plot for k-8, k’-2**

Table

Description automatically generated with medium confidence Table

Description automatically generated

* **Speed up and efficiency for k -9, k’-2**

Table

Description automatically generated with medium confidenceTable

Description automatically generated

* **Speed up and efficiency for k -10, k’-2**

Table

Description automatically generated with medium confidenceTable

Description automatically generated with medium confidence

* **In this section again experiment was conducted by varying the k from k-8, k-9,k-10,k-11 with a different value for k’-6 for better validation of speed up improvement with k size.**
* **Speed up and efficiency for k-8, k’-6**

Table

Description automatically generated

* **Speed up and efficiency for k-9, k’-6**

Table

Description automatically generatedTable

Description automatically generated

* **Speed up and efficiency for k-10 and k’-6**

Table

Description automatically generatedText, table

Description automatically generated

* **Speed up and efficiency for k-11 and k’-6**

Table

Description automatically generated

**Observation for parallel performance by varying k values:**

Initially when lower value of k was taken as 4 that is matrix size being 16 x 16, we didn’t observe any speed up and the efficiency was very less. Therefore, we concluded that parallelized Strassen’s multiplication is not so effective for smaller matrixes. From the above values for execution time for values of k – 8, 9, 10, 11 we see that execution time increases as the value of matrix size increases as a greater number of recursive calls is involved that increases the overall computation time.

Strassen’s parallel code is very effective as value of matrix size increases as we got better speed up. For k-8,9,10 and 11 as we increase the number of threads by factor of 2 code gets parallelized well as execution time is decreasing slowly thereby increasing the speed up. However, beyond a point we increase threads or use large number of threads like 32 ,64 due to the overhead in the shared memory processor we observe that execution time suddenly increases thereby decreasing the speed up. Efficiency as seen in the graphs is gradually decreasing. Also, from the speed up graphs from above where k’-6 and k is increased from k – 8,9 ,10 to k -11 we see how effective parallel performance is as speed up is slowly increasing with value of k. The trend for efficiency plot is its high in the beginning and gradually decreases as we increase threads.

1. **Fixed k and changing the value of terminal matrix k’ from k’-2 ,4, 5 ,6, 7**

In this section different experiments were conducted to observe the trend of execution time and parallel performance by varying k’ values for matrix size k-8 and matrix size k-9.

* **Speed up and efficiency plot for k-8, k’-2**

Table

Description automatically generated

* **Speed up and efficiency plot for k-8, k’-** 4

Table

Description automatically generated

* **Speed up and efficiency plot for k-8, k’- 5**

Table

Description automatically generated

* **Speed up and efficiency plot for k-8, k’- 6**

Table

Description automatically generated

The above experiment is now conducted with k value as 9 and changing k ‘to 2,6 and 7.

* **Speed up and efficiency plot for k-9 and k’-2**

Table

Description automatically generated

* **Speed up and efficiency plot for k-9 and k’-6**

Table

Description automatically generated

* **Speed up and efficiency plot for k-9 and k’-7**

Table

Description automatically generated

**Observation and analysis for changing values of terminal matrix size k’**

For k-8 and when k’ was increased there was a decrease in execution time and increase in speedup, this was because with higher values of k’ the number of levels for normal multiplication is more which gets efficiently parallelized as compared to the parallelization of recursive part using pragma task. Using task in recursive calls can result in overhead , hence compared to Strassen’s approach normal multiplication is getting effectively parallelized and hence execution time is decreasing by slowly leading to better speed up.

**Design choices to improve the parallel performance of the code:**

* In order to get better parallel performance, I have parallelized both the normal matrix multiplication that is used during the termination of recursion process and the recursive process of Strassen’s multiplication. Since in normal multiplication we have 3 for loops with 2 outermost for loops being independent, I have used pragma omp parallel for that parallelizes the 2 outermost for loops. So, from levels 1 to k’ code is getting parallelized in this process.
* For levels beyond k’ to k where k> k’ recursive calls are being made to calculate the value of M1, M2, M3 , M4 ,M5 ,M6 and M7. Here task construct has been used since it defines an explicit task for every recursive call. Since its first private by default for each thread will take its own private variable as it was initially. However, it was found that task construct did parallelize only the call of the Strassen’s multiplication. On successive recursive calls when omp\_get\_num\_threads were used to print only 1 thread was printed showing task doesn’t support nested parallel region by default. When nested parallel region was enabled, so that thread team remains same in each recursive call a high jump in the execution time was observed due to overhead due to the presence of large number of concurrent threads. This was decreasing the speed up. So to get best parallel performance omp nested call can be kept as 0 or ignored as it is 0 by default.
* Thirdly, since the function calls to add matrix, subtract matrix , split the matrix into sub matrices involve for loops I have used #pragma omp parallel for to simply parallelize the for loops in each of such function calls which also adds to better parallel performance.
* The main insight I obtained by working in this project is while using threads can parallelize the program for better execution time, using large no of threads can also increase the overhead and hence show less speed up. So the range of selection of threads from 1,2,4,8,16,32 was crucial to analyze the parallel performance.
* Also, another insight is the hybrid approach to solve the problem added as an advantage where some levels were executed by normal multiplication and some levels by Strassen’s recursive approach. Using this hybrid approach, we could tune value of k k’ to get optimized parallel performance. If terminal matrix size would be 1 all the time, then there is a high chance of getting speed up decrease very earlier when we increase threads. This is because at every recursive calls a task is created and if the depth of calls is very large then large number of task will result in significant overhead and decrease the performance. Tuning of threshold on when to stop recursive was an important insight.
* Third insight that can be taken from the project is that using Open MP for parallelizing the algorithm is relatively better as it provides a great support to synchronization between threads. Since pragma parallel region when enclosed implies a barrier already we need to add special barrier commands to synchronize the threads.
* Fourth insight I took from the project is that parallelizing Strassen’s multiplication is preferred only for sufficiently larger matrix size. The execution time for parallel part for small matrix size was high and hence no speed up as such.

**Brief Description on how to compile and execute the code:**

* To compile and execute below steps are used.
* Initially load the Intel software stack by running module load Intel.
* To compile the code in non-dedicated mode the command icc -qopenmp -o strass.exe strass.c if where the source code file name is strass.c
* ./strass.exe 11 6 1 run the code by passing 3 arguments where the first input 11 in this case refers to k (size of matrix is 2^11), second argument refers to size of terminal matrix k’ (size is 2^k’) and third argument is the number of threads(1 in this case).
* Also, to check the parallel performance please try with high value of k >=7
* To compile and run in dedicated mode batch file needs to be submitted using the command sbatch strassen.grace\_job. Below screenshot shows the values given in the batch file for obtaining results in dedicated mode where no of nodes used is 1 with 16GB memory per node.

Text

Description automatically generated