Strassen's Matrix Multiplication Algorithm

Problem Description

Write a threaded code to multiply two random matrices using Strassen's Algorithm. The application will generate two matrices $A(M,P)$ and $B(P,N)$, multiply them together using (1) a sequential method and then (2) via Strassen's Algorithm resulting in $C(M,N)$. The application should then compare the results of the two multiplications to ensure that the Strassen's results match the sequential computations.

The input to the application comes from the command line. The input will be 3 integers describing the sizes of the matrices to be used: $M$, $N$, and $P$.

Code restrictions: A very simple, serial version of the application, written in C, will be available here. This source file should be used as a starting point. Your entry should keep the body of the main function, the matrix generation function, the sequential multiplication code, and the function to compare the two matrix product results. Changes needed for implementation in a different language are permitted. You are also allowed to change the memory allocation and other code to thread the Strassen's computations. (It would be a good idea to document the changes and the reason for such changes in your solution write-up.) After the needed changes, your submitted solution must use some form of Strassen's Algorithm to compute the second matrix multiplication result.

Timing: The time for execution of the Strassen's Algorithm will be used for scoring. Each submission should include timing code to measure and print this time to stdout. If not, the total execution time will be used.
Matrix multiplication

Given two matrices $A_{M \times P}$ and $B_{P \times N}$, the product of the two is a matrix $C_{M \times N}$ which is computed as follows:

```c
void seqMatMult(int m, int n, int p, double** A, double** B, double** C) {
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < p; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}
```

Strassen's algorithm

To calculate the matrix product $C = AB$, Strassen's algorithm partitions the data to reduce the number of multiplications performed. This algorithm requires $M$, $N$ and $P$ to be powers of 2. The algorithm is described below.

1. Partition $A$, $B$ and $C$ into 4 equal parts:

   \[
   A = \begin{pmatrix}
   A_{11} & A_{12} \\
   A_{21} & A_{22}
   \end{pmatrix}
   \]

   \[
   B = \begin{pmatrix}
   B_{11} & B_{12} \\
   B_{21} & B_{22}
   \end{pmatrix}
   \]

   \[
   C = \begin{pmatrix}
   C_{11} & C_{12} \\
   C_{21} & C_{22}
   \end{pmatrix}
   \]

2. Evaluate the intermediate matrices:

   \[
   M_1 = (A_{11} + A_{22}) \ (B_{11} + B_{22})
   \]

   \[
   M_2 = (A_{21} + A_{22}) \ B_{11}
   \]

   \[
   M_3 = A_{11} \ (B_{12} - B_{22})
   \]

   \[
   M_4 = A_{22} \ (B_{21} - B_{11})
   \]

   \[
   M_5 = (A_{11} + A_{12}) \ B_{22}
   \]

   \[
   M_6 = (A_{21} - A_{11}) \ (B_{11} + B_{12})
   \]

   \[
   M_7 = (A_{12} - A_{22}) \ (B_{21} + B_{22})
   \]

3. Construct $C$ using the intermediate matrices:

   \[
   C_{11} = M_1 + M_4 - M_5 + M_7
   \]

   \[
   C_{12} = M_3 + M_5
   \]

   \[
   C_{21} = M_2 + M_4
   \]

   \[
   C_{22} = M_1 - M_2 + M_3 + M_6
   \]
**Serial Algorithm**

1. Partition A and B into quarter matrices as described above.
2. Compute the intermediate matrices:
   1. If the sizes of the matrices are greater than a threshold value, multiply them recursively using Strassen's algorithm.
   2. Else use the traditional matrix multiplication algorithm.
3. Construct C using the intermediate matrices.

**Parallelization**

The evaluations of intermediate matrices $M_1, M_2 \ldots M_7$ are independent and hence, can be computed in parallel. On a machine with $Q$ processors, $Q$ jobs can be run at a time.

**Initial Approach**

The initial approach to a parallel solution used a *task pool* model to compute $M_1, M_2 \ldots M_7$. As shown in the diagram below, the second level of the algorithm creates 49 ($7 \times 7$) independent multiplication tasks which can all be executed in parallel, $Q$ jobs at a time.

![Diagram]

**A Realization**

Since the number of jobs is 49, ideally, on a machine with $Q$ cores (where $Q = 2^q$, $Q <= 16$), 48 of these would run concurrently while 1 would end up being executed later, thus, giving poor processor utilization. It would be a better idea to split the last task further.
Final Parallel Algorithm

The final solution uses thread pooling with a pool of Q threads (where Q is the number of processors), in conjunction with the Strategy pattern to implement Strassen's algorithm. The algorithm is described below:

1. If the sizes of A and B are less than the threshold
   1.1 Compute C = AB using the traditional matrix multiplication algorithm.
2. Else use Strassen's algorithm
   2.1 Split matrices A and B
   2.2 For each of Mi, i = 1 to 7
      2.2.1 Create a new thread to compute Mi = A'i B'i
      2.2.2 If the sizes of the matrices are less than the threshold
         2.2.2.1 Compute C using the traditional matrix multiplication algorithm.
      2.2.3 Else use Strassen's algorithm
         2.2.3.1 Split matrices A'i and B'i
         2.2.3.2 For each of Mij, j = 1 to 7
            2.2.3.2.1 If i=7 and j=7 go to step 1 with A = A'77 and B = B'77
            2.2.3.2.2 Get a thread from the thread pool to compute Mij = A'i B'j
            2.2.3.2.3 Execute the recursive version of Strassen's algorithm in this thread
         2.2.3.3 Wait for the Mij threads to complete execution
         2.2.3.4 Compute Mi
   2.3 Wait for the Mi threads to complete execution
   2.4 Compute C

The Strategy pattern

The above algorithm defines 3 distinct strategies to be used with Strassen's algorithm:
1. Execute each child multiplication operation in a new thread (M1, M2, ..., M7)
2. Execute each child multiplication operation in a thread (Mi) from the thread pool
3. Execute each child multiplication operation using recursion

Hence, the Strategy pattern was used in the implementation.
Problems

Initially, an attempt was made to use the originally allocated storage in all computations. However, due to cache thrashing, extremely poor performance was observed in this case. Top showed less than 10% processor utilization on a quad core processor. This problem was resolved by creating copies of the required matrices for each thread. This speeds up the solution at the expense of memory requirement.

Performance

The machine used for these tests had the following configuration:
Core 2 Quad Q8300 2.33GHz, 4GB DDR3 1333MHz RAM

Following are the results for M = N = P = 2048. The traditional algorithm takes approximately 154.54 seconds to perform the multiplication.

<table>
<thead>
<tr>
<th>#Threads</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.24</td>
</tr>
<tr>
<td>2</td>
<td>10.81</td>
</tr>
<tr>
<td>4</td>
<td>6.52</td>
</tr>
</tbody>
</table>

For 4 threads the traditional algorithm uses about 1.7% of the available memory whereas the parallel version uses about 14%.

Conclusion

Strassen's algorithm definitely performs better than the traditional matrix multiplication algorithm due to the reduced number of multiplications and better memory separation. However, it requires a large amount of memory to run. The performance gain is sub-linear which could be due to the fact that there are threads waiting for other threads to complete execution.