Parallelization of the Strassen’s Algorithm

Problem Description

Problem: 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 general algorithms run in $O(n^3)$ time (assume that $m = p = n$), Strassen’s Algorithm run in $O(n^\log_2 7)$ time. Strassen’s Algorithm takes the following seventh-recursive divide-and-conquer approach: Divide the matrices $A$ and $B$ into four quarters, compute seven factors, and then they combine in four parts for the matrix $C$.

Algorithm described as follow:

Let $A$, $B$ be two square matrices over a ring $F$. We want to calculate the matrix product $C$ as

$$C = AB$$

$A$, $B$, $C \in F^{2^n \times 2^n}$

If the matrices $A$, $B$ are not of type $2^n \times 2^n$ we fill the missing rows and columns with zeros.

We partition $A$, $B$ and $C$ into equally sized block matrices

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

With

$$A_{i,j}, B_{i,j}, C_{i,j} \in F^{2^{n-1} \times 2^{n-1}}$$

Then

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$
$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$
$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$
With this construction we have not reduced the number of multiplications. We still need 8 multiplications to calculate the $C_{i,j}$ matrices, the same number of multiplications we need when using standard matrix multiplication.

Now comes the important part. We define new matrices

\[
\begin{align*}
M_1 & := (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 & := (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 & := A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 & := A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 & := (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 & := (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 & := (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{align*}
\]

which are then used to express the $C_{i,j}$ in terms of $M_k$. Because of our definition of the $M_k$ we can eliminate one matrix multiplication and reduce the number of multiplications to 7 (one multiplication for each $M_k$) and express the $C_{i,j}$ as

\[
\begin{align*}
C_{1,1} & = M_1 + M_4 - M_5 + M_7 \\
C_{1,2} & = M_3 + M_5 \\
C_{2,1} & = M_2 + M_4 \\
C_{2,2} & = M_1 - M_2 + M_3 + M_6
\end{align*}
\]

We iterate this division process $n$ times until the submatrices degenerate into numbers (elements of the ring $F$).

Practical implementations of Strassen’s algorithm switch to standard methods of matrix multiplication for small enough sub-matrices, for which they are more efficient. The particular crossover point for which Strassen’s algorithm is more efficient depends on the specific implementation and hardware. It has been estimated that Strassen's algorithm is faster for matrices with widths from 32 to 128 for optimized implementations, and 60,000 or more for basic implementations.[1][2]

**Numerical analysis**
The standard matrix multiplication takes approximately $2N^3$ (where $N = 2^n$) arithmetic operations (additions and multiplications); the asymptotic complexity is $O(N^3)$.

The number of additions and multiplications required in the Strassen algorithm can be calculated as follows: let $f(n)$ be the number of operations for a matrix. Then by recursive application of the Strassen algorithm, we see that $f(n) = 7f(n-1) + 4n$, for some constant $l$ that depends on the number of additions performed at each application of the algorithm. Hence $f(n) = (7 + o(1))n$, i.e., the asymptotic complexity for multiplying matrices of size $N = 2^n$ using the Strassen algorithm is

$$O((7 + o(1))n) = O(N^\log_2 7 + o(1)) \approx O(N^{2.807})$$.

The reduction in the number of arithmetic operations however comes at the price of a somewhat reduced numerical stability.

**Serial Code**

- **Record seven factors for add(sub)**
  ```c
  double **M1, **M2, **M3, **M4, **M5, **M6, **M7;
  ```

- **Divide two matrices A and B**
  ```c
  double **A11, **A12, **A21, **A22;
  double **B11, **B12, **B21, **B22;
  ```

- **Record the middle variables**
  ```c
  double **tAM1, **tBM1, **tAM2; **tBM3, **tBM4, **tAM5, **tAM6, **tBM6, **tAM7, **tBM7;
  ```

- **The general codes**
  ```c
  strassenMMult:
  if ((ml-mf)*(nl-nf)*(pl-pf) < GRAIN)
    matmultleaf(mf, ml, nf, nl, pf, pl, A, B, C);
  else {
    //M1 = (A11 + A22)*(B11 + B22)
    AddMatBlocks(tAM1, m2, p2, A11, A22);
    AddMatBlocks(tBM1, p2, n2, B11, B22);
    strassenMMult(0, m2, 0, n2, 0, p2, tAM1, tBM1, M1);
    //M2 = (A21 + A22)*B11
    AddMatBlocks(tAM2, m2, p2, A21, A22);
    strassenMMult(0, m2, 0, n2, 0, p2, tAM2, B11, M2);
    //M3 = A11 *(B12 - B22)
    SubMatBlocks(tBM3, p2, n2, B12, B22);
    strassenMMult(0, m2, 0, n2, 0, p2, A11, tBM3, M3);
    //M4 = A22 *(B21 - B11)
    SubMatBlocks(tBM4, p2, n2, B21, B11);
    strassenMMult(0, m2, 0, n2, 0, p2, A22, tBM4, M4);
    //M5 = (A11 + A12)*B22
  ```
AddMatBlocks(tAM5, m2, p2, A11, A12);
strassenMMult(0, m2, 0, n2, 0, p2, tAM5, B22, M5);

//M6 = (A21 - A11)*(B11 + B12)
SubMatBlocks(tAM6, m2, p2, A21, A11);
AddMatBlocks(tBM6, p2, n2, B11, B12);
strassenMMult(0, m2, 0, n2, 0, p2, tAM6, tBM6, M6);

//M7 = (A12 - A22)*(B21 + B22)
SubMatBlocks(tAM7, m2, p2, A12, A22);
AddMatBlocks(tBM7, p2, n2, B21, B22);
strassenMMult(0, m2, 0, n2, 0, p2, tAM7, tBM7, M7);

for (int i = 0; i < m2; i++)
    for (int j = 0; j < n2; j++) {
        C12[i][j] = M3[i][j] + M5[i][j];
        C21[i][j] = M2[i][j] + M4[i][j];
    }

FAQ

1. Use the abs() defined in the stdlib.h may not be right, I think it should use fabs() defined in the math.h
2. In function “CheckResults()”, THRESHOLD may not be available as use abs()?
3. The dynamic array are not be evaluated with 0. It may lead to some problems.
   So we should change as:
   T *curPtr = new T [nRows * nCols];
   for (int j = 0; j < nRows * nCols; j++) curPtr[j] = 0;
4. I use Microsoft Visual Studio 2008 –Chinese, Does it adapt to the sever?

Parallel Optimization

1. My program use the OpenMP library included in the visual studio. User should open the switch of "OpenMP Support",
   you can do as follow:
   click your solution and click Right button, then
   1. Expand the Configuration Properties node.
   2. Expand the C/C++ node.
   3. Select the Language property page.
   4. Modify the OpenMP Support property.
2. In the factor matrix multiply function: matmultleaf(),
   I swap the order of the three layer: to make full use of caches. As follow:
   for (int k = pf; k < pl; k++)
   for (int i = mf; i < ml; i++)
   for (int j = nf; j < nl; j++)
        C[i][j] += A[i][k]*B[k][j];
3. Also I turn the sequence of the four formulas to avoid data race:

\[
\begin{align*}
C_{11}[i][j] &= M_{1}[i][j] + M_{7}[i][j] + M_{4}[i][j] - M_{5}[i][j]; \\
C_{22}[i][j] &= M_{1}[i][j] + M_{6}[i][j] - M_{2}[i][j] + M_{3}[i][j]; \\
C_{12}[i][j] &= M_{3}[i][j] + M_{5}[i][j]; \\
C_{21}[i][j] &= M_{2}[i][j] + M_{4}[i][j].
\end{align*}
\]

4. It must be official sure that the three input datas m, n, p each should be a power of 2 and can not be divide into a factor which are equal to 1 until the scale of m * n * p are less than the GRAIN. Otherwise you should put a clause insert into the if clause as :

```c
StrassenMMult():
if ((ml-mf)*(nl-nf)*(pl-pf) < GRAIN || ml-mf==2 || ml-nf==2 || pl-pf==2 )
```

5. At present I can not confirm the running core num of the server. So I assume that it has four or eight or sixteen cores. In my program, I make 14 threads. In the function “strassenMMult”, make 7threads in the first layer of the recursive function, to deal with 7 sections which to compute the M1 – M7.Then in the second layer, make 2 threads nested. Maybe making more threads nested will be better.

6. To identify the layer of the recursive function, we use a global variable P to compare with the current layer's scale of p. Code as follow:

```c
int threads;  //use P the confirm the parallel threads
if (pl == P) { threads = 7; omp_set_nested(1); }
else if (pl == (P>>1)) threads = 2;
else threads = 1;
```

7. Main parallel code as follow:

```c
int threads = 1;
if (pl == P) { threads = 7; omp_set_nested(1); }
else if (pl == (P>>1)) threads = 2;
#pragma omp parallel sections num_threads(threads) default (shared)
{
#pragma omp section
{
    //M1 = (A11 + A22)*(B11 + B22)
    AddMatBlocks(tAM1, m2, p2, A11, A22);
    AddMatBlocks(tBM1, p2, n2, B11, B22);
    strassenMMult(0, m2, 0, n2, 0, p2, tAM1, tBM1, M1);
}
#pragma omp section
{
    //M6 = (A21 - A11)*(B11 + B12)
    SubMatBlock(tAM6, m2, p2, A21, A11);
    AddMatBlocks(tBM6, p2, n2, B11, B12);
    strassenMMult(0, m2, 0, n2, 0, p2, tAM6, tBM6, M6);
}
#pragma omp section
```
8. Because there are very many variables constructed in every layer whose capacity are quite much more than the function information, so I don't think of that iterative functions may be faster than the recursive function. Moreover it will construct some temporary variables in the iterative function just like Stack or Queue. So use BFS or DFS to Strassen's Algorithm may not be faster, even slower.
Performance Test

<table>
<thead>
<tr>
<th>Unit: ms</th>
<th>Mode</th>
<th>Threads:</th>
<th>1</th>
<th>1</th>
<th>16</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>GRAIN</td>
<td>8192</td>
<td>8192</td>
<td>8192</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A行</td>
<td>B列</td>
<td>A列B行</td>
<td>normal</td>
<td>serial</td>
<td>recursive</td>
<td>iterative</td>
</tr>
<tr>
<td>M</td>
<td>N</td>
<td>P</td>
<td>Standard</td>
<td>Strassen</td>
<td>PStr</td>
<td>NPStr</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>128</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>256</td>
<td>203</td>
<td>109</td>
<td>63</td>
<td>187</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>512</td>
<td>2312</td>
<td>437</td>
<td>266</td>
<td>1219</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>50766</td>
<td>2703</td>
<td>1750</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>512</td>
<td>125</td>
<td>125</td>
<td>78</td>
<td>140</td>
</tr>
<tr>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>6344</td>
<td>453</td>
<td>266</td>
<td>891</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>128</td>
<td>140</td>
<td>140</td>
<td>78</td>
<td>187</td>
</tr>
<tr>
<td>1024</td>
<td>512</td>
<td>256</td>
<td>2234</td>
<td>578</td>
<td>328</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>256</td>
<td>1024</td>
<td>5735</td>
<td>2515</td>
<td>1600</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>1024</td>
<td>250</td>
<td>484</td>
<td>344</td>
<td>828</td>
</tr>
<tr>
<td>512</td>
<td>128</td>
<td>2048</td>
<td>1266</td>
<td>594</td>
<td>438</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>4096</td>
<td>21781</td>
<td>2578</td>
<td>1640</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>2048</td>
<td>256</td>
<td>9875</td>
<td>2594</td>
<td>1437</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1024</td>
<td>64</td>
<td>593</td>
<td>406</td>
<td>219</td>
<td>1250</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>1024</td>
<td>12687</td>
<td>1078</td>
<td>703</td>
<td>3344</td>
</tr>
</tbody>
</table>

Configuration:
- System Type: XP Professional SP3
- Processor: intel core T2350 1.86 GHz, 2 cores
- Install Physical Memory(Ram): 2 GB
- Total Virtual Memory: 2 GB
- SMP: Enabled

Conclusion
It is really a nice problem! Classic, Graceful, Colorful. The Algorithm has a lot of improve methods in the multi-core computer, giving us a wonderful space to imagine. Looking from the horizontal direction, it is a fractal structure, we can use the grammar of fractal trees to describe every factor movement. Looking from the vertical direction, it is a progressive structure, we can use Pipeline Design.

References