Threaded Matrix Multiplication Analysis

Piero Marini

June 26, 2019

The algorithm used for threaded multiplication basically takes care of dividing the work between the assigned threads depending on the amount of operations that have to be done to perform the multiplication. This is calculated inside each **threadedMultiplication** function using the current **threadID** and the amount of rows and columns for the matrices in question.
Basically we assign to each thread a number of rows (from **m1**) to multiply with all the columns on the other Matrix (noted as **m2** on the function).

After this, the normal structure of a matrix multiplication is applied by multiplying each row element from matrix **m1** with each column element from matrix **m2**.

Listing 1: C++ Threaded Multiplication Function

```
1  template<typename X>
2  void threadedMultiplication(PMatrix<X> &ret,
3                              const PMatrix<X> &m1,
4                              const PMatrix<X> &m2,
5                              const long long threadID) {
6    const long long numElements =
7                          m1.m_Columns * m1.m_Rows;
8    const long long numOperations =
9                          numElements / m1.m_NumThreads;
10   const long long restOperations =
11                         numElements % m1.m_NumThreads;
12
13   long long start_op, end_op;
14
15   if (threadID == 0) {
16     start_op = numOperations * threadID;
17     end_op = (numOperations * (threadID + 1))
18              + restOperations;
19   } else {
20     start_op = numOperations * threadID
21              + restOperation
```

```
22      end_op = (numOperations * (threadID + 1))
23               + restOperations;
24    }
25
26    for (long long op = start_op; op < end_op; ++op) {
27      const long long row = op % m2.m_Rows;
28      const long long col = op / m1.m_Columns;
29      X temp = 0;
30      for (long long i = 0; i < m1.m_Columns; ++i) {
31        temp += m1.data.at(row).at(i)
32                * m2.data.at(i).at(col);
33      }
34      ret.data[row][col] = temp;
35    }
36  }
```

This function is then passed to **std::thread** to be executed by each desired thread on each part of the matrices and is timed using the **std::chrono** functions.

Listing 2: C++ Threaded Multiplication Function

```
1  template<typename T>
2  PMatrix<T> PMatrix<T>::operator*(const PMatrix &rhs) {
3    if(m_Rows != rhs.m_Columns) {
4      throw std::runtime_error("Row Count on A doesn't
5                                match Column Count on B");
6    }
7
8    PMatrix<T> ret {m_Columns, rhs.m_Rows, m_NumThreads};
9
10
11   auto t1 = std::chrono::high_resolution_clock::now();
12
13   std::thread threads[m_NumThreads];
14
15   for (long long i = 0; i < m_NumThreads; ++i) {
16     threads[i] = std::thread(threadedMultiplication<T>,
17                              std::ref(ret),
18                              std::ref(*this),
19                              std::ref(rhs), i);
20   }
21   for (long long i = 0; i < m_NumThreads; ++i) {
22     threads[i].join();
23   }
24
25   auto t2 = std::chrono::high_resolution_clock::now();
26
27   auto duration = std::chrono::duration_cast
28                       <std::chrono::milliseconds>(t2 - t1).count();
29   std::cout << "Time with " << m_NumThreads
30             << " threads: " << duration << " ms" << '\n';
31
32   return ret;
33 }
```

Timing obtained from running the **main.cpp** program after multiplying two matrices with sizes 1024×1024.

1. Time with 1 threads: 21373 ms
2. Time with 2 threads: 22120 ms
3. Time with 4 threads: 11958 ms
4. Time with 8 threads: 13243 ms
5. Time with 16 threads: 16780 ms

Graphing the results:

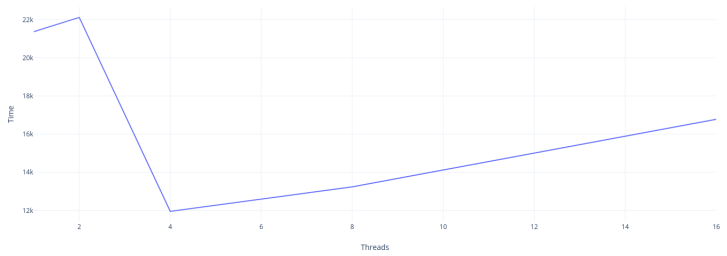

Figure 1: Execution Time Graph