Parallel and Distributed Algorithms

2. Project - Mesh Multiplication (*MM*)

Jozef Méry (xmeryj00@vutbr.cz)

25.4.2021

1 Algorithm analysis

Mesh multiplication is a parallel algorithm for multiplying two matrices X and Y with X having dimensions $n, k \in \mathbb{N}$ and Y having dimensions $k, m \in \mathbb{N}$ where, $k \leq m \wedge k \leq n$. It uses $n * m$ parallel processors with each processor requiring k calculations to finish the algorithm. The mesh multiplication architecture is presented on figure 1. The mesh represents the shape of the resulting matrix. Processors receive values from the left and upper neighboring processor, multiply each pair, and add the results to a local accumulator. Finally, the received values are sent to right and bottom neighbors. In the end, the accumulator contains the corresponding matrix cell value.

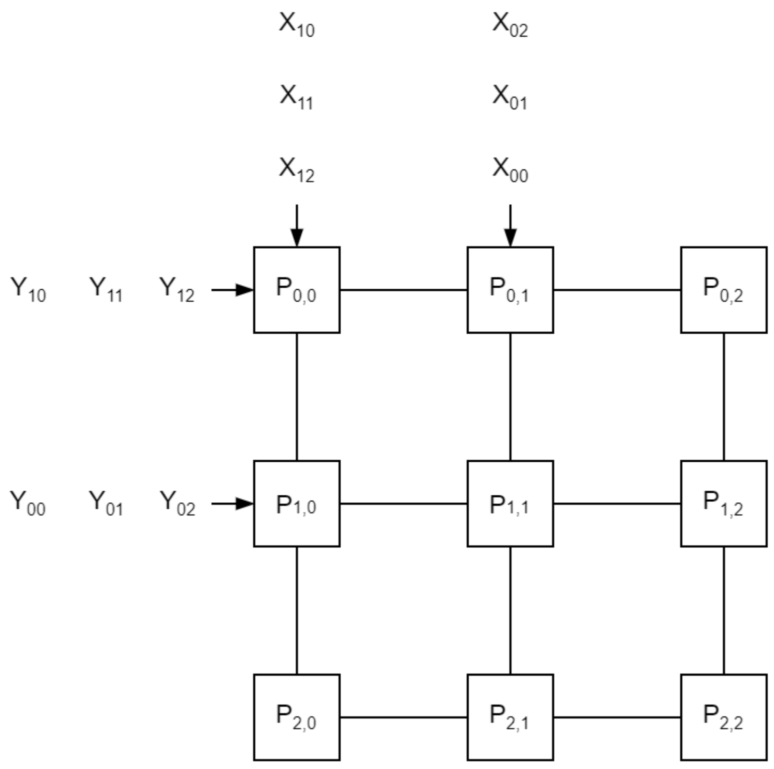

Figure 1: Nine processors in a 3x3 mesh illustrating the algorithm.

1.1 Algorithm complexity

Elements $X_{n-1,k-1}$ and $Y_{k-1,m-1}$ (indexed from **0**) require $m+n+k-2$ steps to reach $P_{n-1,m-1}$, after which the final product can be calculated. Assuming the worst case where $m=n=k$, the time complexity is linear:

$$t(n) = O(3n-2) = O(n).$$

With a quadratic spatial complexity $p(n) = O(n^2)$, the total cost becomes cubic:

$$c(n) = t(n) * p(n) = O(n) * O(n^2) = O(n^3).$$

2 Implementation

The application is implemented in the *C++0x* language using *OpenMPI*[1] library.

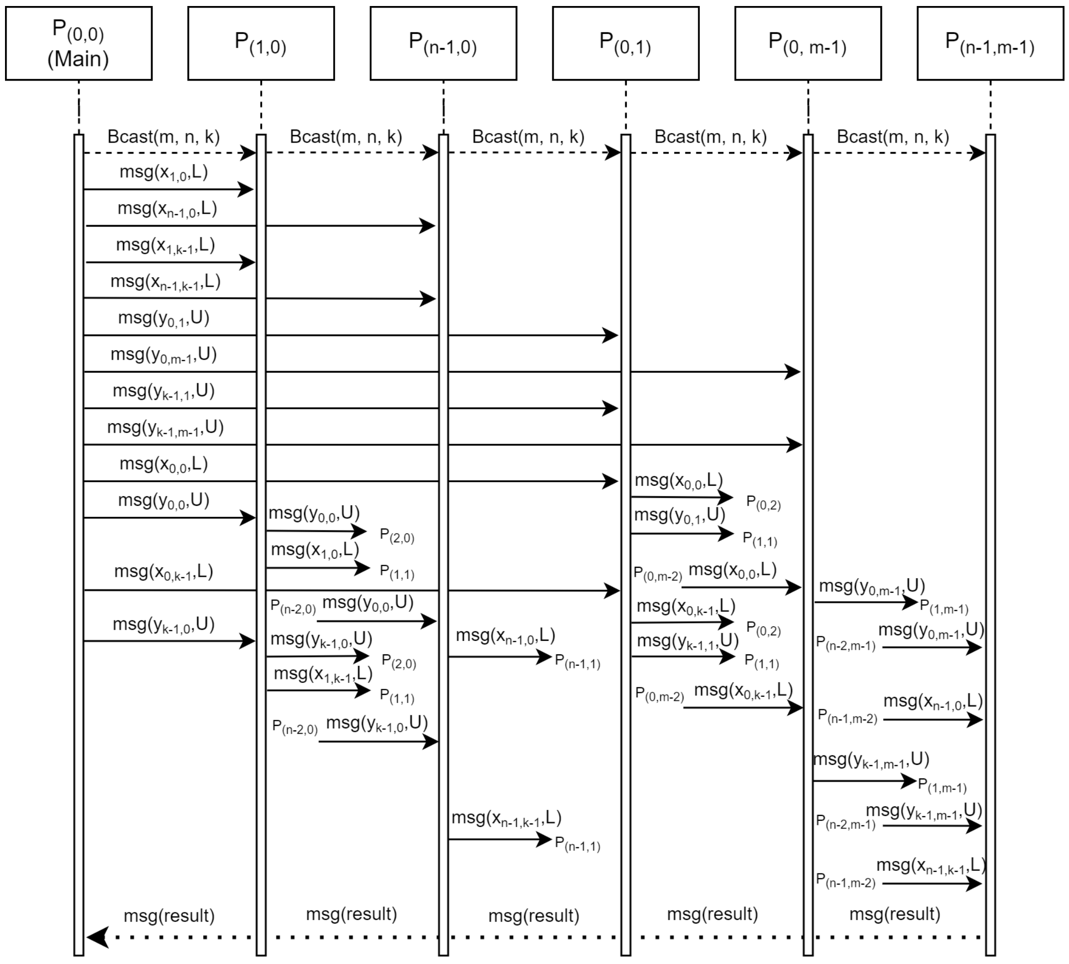

Figure 2: Sequence diagram of the communication in a Mesh Multiplication OpenMPI Application.

[1]OpenMPI home page.

There are *two* classes the define most of the application logic. An instance of the *Enumerator* class is executed within every process except $P_{0,0}$, which executes an extension (trough inheritance) of *Enumerator*, an instance of the *Main* class instead. *Main* begins the whole algorithm by loading, transforming, and verifying the input matrices from files *mat1* and *mat2*. Next, *Main* broadcasts the dimensions m, n, and k, then the input matrices are propagated to the first column and row over k iterations respectively, thus $k*(m+n)$ messages are sent by *Main*, making the overall time complexity of *Main* $t(n) = O(n^2)$. Finally, *Main* and every other enumerator executes the *enumerate* method, which does k iterations, receives $2k$ messages (*UP* and *LEFT* tags), does k multiplications followed by accumulations, sends $2k$ messages, and finally sends the accumulator value to *Main*, which is the value in the corresponding result matrix cell. The communication between processes is visualised on figure 2. For clarity, the *send* and *receive* parts of messaging are merged into one *msg*. Processes and input values are indexed from **0**. The initial broadcast sends *3* values m, n, k, while the rest of the messages contain a single number only. Except results, messages have tags U and L, which correspond to *UP* and *LEFT* tags respectively. The tag is actually only relevant for processes $P_{0,1}$, $P_{1,0}$, and Main, as all of them receive both U and L values from *Main* (as *Main* is also an *Enumerator* it propagates messages to *itself* to maintain code readability and decomposition, but this is omitted in figure 2). All the other processes have unique U and L value senders. The process indexing in the actual implementation uses 1D indexing, $pid \in \langle 0, n*m)$, exactly as *OpenMPI*.

2.1 Benchmarks

Time measuring was done using *std::chrono* library. During benchmarking, the output is suppressed and only the measured time is printed in milliseconds. The time is measured from broadcasting to receiving the result, meaning that the time of input loading, parsing, printing, and various checks is ignored. The input generation and calling of the application was automated by a Python script, which also parsed the output of the specially compiled binary, and using *Matplotlib*[2] generated figure 3. The benchmarks were executed for square input matrices with size from 1 to 20. The actual measurings do not follow perfect linear scaling due to various negative factors, but the *Ordinary-Least-Square* regression predicts a near-perfect linear growth.

3 Conclusions

The algorithm was implemented on the theoretical base described in section 1. Benchmarks prove the theory predicted linear time complexity scaling of the algorithm despite the quadratic messaging time complexity of *Main*.

[2]https://matplotlib.org/

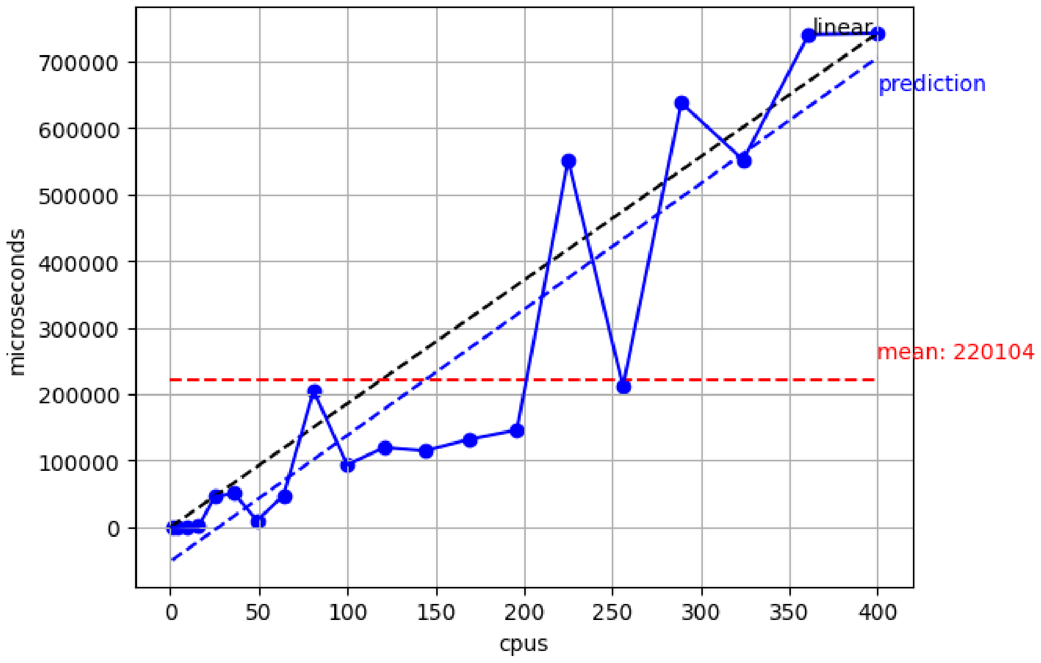

Figure 3: Benchmark results with up to 400 processors showing a linear time complexity.