Cannon's algorithm MPI implementation and analysis

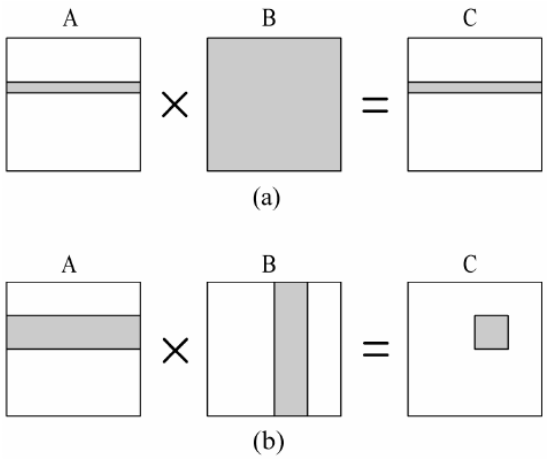
JesusADM2k18 Team

ABSTRACT

Algorithm parallelization plays and important role in modern multi-core architectures. The present paper presents the implementation of the Cannon's algorithm for matrix multiplication, using MPI. The performance, efficiency and scalability analysis was made compared to the most basic non-parallel matrix multiplication algorithm.

Keywords

matrix multiplication, Cannon's algorithm, parallel computing, MPI



1. INTRODUCTION

There exist plenty of algorithms for matrix multiplication. The simplest way to do it is to compute the value of each coefficient of the resulting matrix. We will use this basic algorithm as a comparison for our Cannon's implementation. There exist other sequential algorithms that provide much better results. For example, we could imagine a recursive block-oriented algorithm or an iterative row-oriented algorithm.

However the better results come with parallelization. Cannon's algorithm (b) is particularly interesting because each process is in charge of computing one block of the resulting matrix. To do so, each process requires to know one row of block of matrix A and one column of blocks of matrix B. If we compare this to another parallel row-oriented algorithm (a), where each process is in charge of computing one row of the resulting matrix, we can see how efficient the Cannon's algorithm is in terms of communication between the processes. In the second algorithm, each process requires to know the entire B matrix in order to compute their targeted row of C.

This paper presents the analysis of Cannon's algorithm for matrix multiplication.

It is an iterative algorithm easy to implement. It has sqrt(p) iterations, where p is the number of processes. Thus p needs to be a perfect square, since we are dividing the matrix into same-sized blocks and each process is responsible for one block. We also need that the size of the matrix n is a multiple of sqr(p).

As we will see more in detail in the sext segment, during each iteration each process multiplies two (n / sqr(p)) * (n / sqr(p)) blocks, resulting in a computational complexity of :

$O(n^3 / p)$

During each iteration each process sends and receives two blocks, resulting in a communication complexity of:

$O(n^2 / sqrt(p))$

As we will see Cannon's algorithm is also highly scalable compared for example to algorithm (a) above.

2. CANNON'S ALGORITHM

The basic idea is the following:

In the beginning each process, p(i,j), responsible for computing the block C(i,j) of the resulting matrix, is assigned with the blocks A(i,j) and B(i,j) of the two matrices to multiply. The process p(i,j) must compute C(i,j) = ∑A(i,k)*B(k,j). The idea is for the processes to share their blocks in a way that each process receives only the blocks that he needs.

In the beginning some processes have incompatible blocks. For instance, the process p(0,2) can't multiply the blocks A(0,2) and B(0,2) in order to compute C(0,2). There needs to be a first rearrangement of the blocks. To do so, rather than starting with the blocks A(i,j) and B(i,j), each process starts with the blocks A(i,k) and B(k,j), where k is (i+j)%N and N=sqrt(p).

In the following iterations, each process sends its current block of the matrix A cyclically to the left, and its current block of the matrix B cyclically upwards. After N iterations, each process has received all the blocks needed to compute C(i,j).

Here is the pseudo-code of the algorithm, where a(i,j) represents the current block of A held by p(i,j)

Pseudo-code for process p(i,j) :

k = (I + j) mod N

a(i,j) = A(i,k)

b(i,j) = B(k,j)

c(i,j) = 0

for k=0 to N-1
    c(i,j) = c(i,j) + a(i,j)*b(i,j)
    Left-circular-shift each row of a by 1,
        so that A(i,j) is overwritten by a(i, (j+1) mod N)
    Up-circular-shift each column of b by 1,
        so that B(i,j) is overwritten by b((i+1) mod N, j)
end for

3. MPI IMPLEMENTATION

Each matrix is seen as a 1D vector.

After initializing the values of a and b (containing the current block of A and B held by each process), here is the code that implements the algorithm:

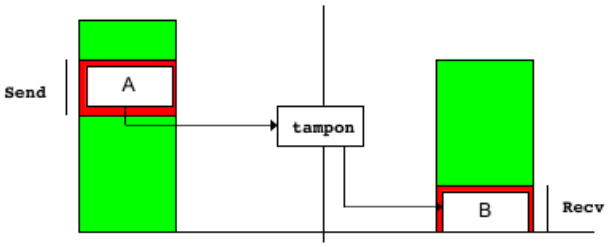
```
update_block(c,a,b, size_block);

for(int l = 0; l < N-1 ; l++){
      MPI_Send(a, size_block*size_block, MPI_INT,
         i*N + ((j+N-1)%N), 1, MPI_COMM_WORLD);
      MPI_Send(b, size_block*size_block, MPI_INT,
         ((i+N-1)%N)*N + j, 1, MPI_COMM_WORLD);
      MPI_Recv(a, size_block*size_block, MPI_INT,
         i*N + ((j+1)%N) , 1, MPI_COMM_WORLD, &status);
      MPI_Recv(b, size_block*size_block, MPI_INT,
         ((i+1)%N)*N + j, 1, MPI_COMM_WORLD, &status);

      MPI_Barrier(MPI_COMM_WORLD);
      update_block(c,a,b, size_block);
 }

 MPI_Gather(c, size_block*size_block, MPI_INT, final_matrix,
       size_block*size_block,  MPI_INT, 0, MPI_COMM_WORLD);
```

The id of the process p(i,j) is equal to N*i + j. This mapping allows us to send and receive the blocks to the right processes.

After each loop, the function MPI_Barrier assures that no process enters a new loop before everybody has completed the same amount of loops.

At the bottom, the function MPI_Gather collects all the blocks computed by the processes and stores them in the value final_matrix of the process 0.

We have chosen to share the blocks with the function MPI_Send. The process calling this function waits until the message has completely been sent to a buffer, from which the receiving process can get the data when it calls the MPI_Recv function.



If he had used the function MPI_Ssend instead, our code wouldn't have worked since all the processes would wait forever for the receiver to actually receive the data. However, as the MPI documentation states, MPI_Send "may block until the message is received by the destination process" as well. We have encountered this issuewhen the blocks where too big. While running our code, bare in mind that this problem might occur when the block size is too big, since we haven't had the time to address this issue.

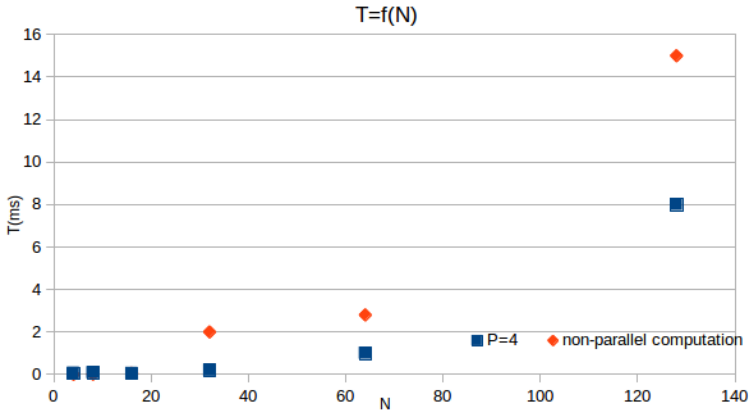
The function used to update the block c multiplies two (n / sqr(p)) * (n / sqr(p)) blocks, which requires a complexity of $O(n^3/sqrt(p)^3)$. Since each process calls this function sqrt(p) times, the computation complexity rises to $O(n^3/p)$. On top of that, each process sends and receives two blocks of the size per iteration, thus sharing $O(n^2/ sqrt(p))$ elements in total.

4. BENCHMARKING AND ANALYSIS

In this section we will showcase the performance of the Cannon's algorithm compared to a basic non-parallel algorithm with a $O(n^3)$ complexity.

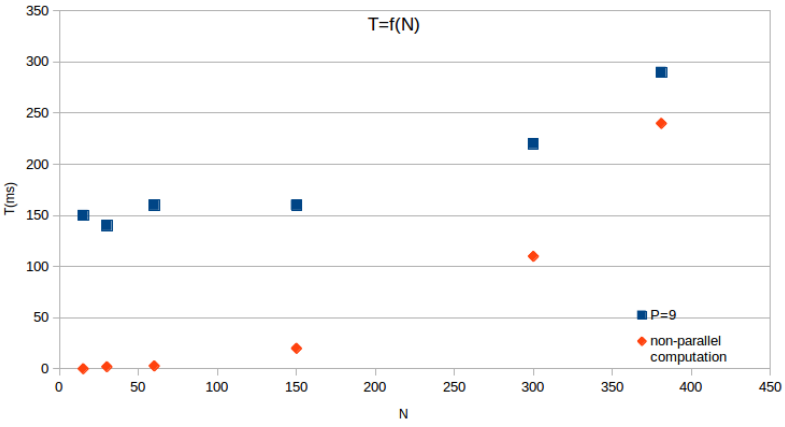
We have tested the algorithm with 4, 9 and 16 processes.

Note: The results of the non-parallel algorithm have been obtained while running with only one process.



This graph shows the computation time (in ms) with 4 processes for different matrix size.

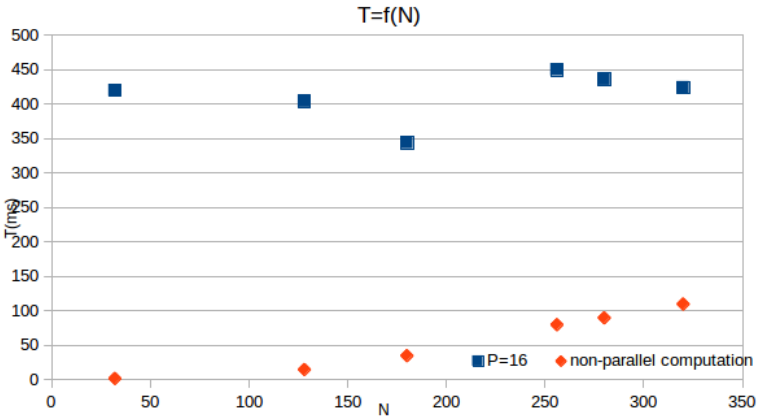
Up to N=32, the non-parallel algorithm is much faster than the Cannon's algorithm (we can't really see it in the graph because both are very fast for such small matrices). As we can see, after this point the Cannon's algorithm outperforms the non-parallel one significantly. The non-parallel algorithm is also not very scalable, since it grows at a faster pace than Cannon's algorithm.



This graph shows the computation time (in ms) with 9 processes for different matrix size.

This time, we can see from the start that the Cannon's algorithm is much slower for the small matrices. However, we can see again that the non-parallel algorithm is not very scalable because its computation time grows very fast with the size of the matrix. The Cannon's algorithm is much more robust and scalable. If

we could test it with bigger matrices (we can't because of the problems encountered with MPI_Send), we would see the Cannon's algorithm outperform the non-parallel algorithm as we can see it in the other graph.



This graph shows the computation time (in ms) with 12 processes for different matrix size.

We can see that having a lot of processes to multiply two small matrices is really not a good idea. This shows that calling the MPI functions for communicating isn't "free", but demands some time that can't be overlooked with small matrices.

5. CONCLUSION

In this paper we have introduced the Cannon's algorithm for matrix multiplication. In the first sections we explained how the algorithm works and our implementation using MPI.

Like the other parallel algorithms that we have seen in class, they are not a good solution when we don't need to do a heavy computation. However, the Cannon's algorithm is very reliable for large matrices and outperforms the non-parallel algorithms considerably. Its main strength is its high scalability, as shown in section 4

The main strength of this algorithm is its high scalability, as shown in section 4.

6. REFERENCES

Quinn. J. M. 2003. *Parallel Programming in C and OpenMP*, International Edition .