Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

Report
about laboratory work №2
«Matrix multiplication»

Student

Sultan Zhumabaev J4133c

St. Petersburg
2020

1. GOAL OF LABORATORY WORK

The purpose of this work is to implement two parallel algorithms for multiplying two matrices A and B using *MPI* and determining the speedup for multiplying.

2. TASK DEFINITION

The problem is to obtain the result of multiplying two randomly given matrices, and determine which algorithm is working faster. The result of multiplying matrices $A(N, N)$ and $B(N, N)$ is a square matrix $C(N, N)$, each element of which is the scalar product of the corresponding rows of matrix A and columns of matrix B. It is assumed that this problem will be solved by following these steps:

1 Implement the serial version of the Matrix Multiplication.

2 Chose the problem decomposition.

3 Define sub-problems and size of each sub-problem.

4 Define information dependencies between sub-problems.

5 Implement the parallel algorithm#1 of matrix multiplication.

6 Implement the parallel algorithm#2 of matrix multiplication using MPI. Derived Data Types (define and use columns).

7 Determine the speedup.

3. BRIEF THEORY

The sequential algorithm is represented by three nested loops and is focused on the sequential calculation of the rows of the resulting matrix C.

This algorithm is iterative and focuses on the sequential calculation of the rows of matrix C. It is assumed that $n \times n \times n$ multiplication operations and the same number of addition operations of the elements of the original matrices are performed. The number of operations performed is of order $O(n^3)$.

Since each element of the resulting matrix is a scalar product of the row and column of the original matrices, then to calculate all the elements of the matrix C size $n \times n$, it is necessary to perform $n^2(2n-1)$ scalar operations and spend time $T_1 = n^2(2n-1)t$, where t is the time to perform one elementary scalar operation.

4. ALGORITHM (METHOD) OF IMPLEMENTATION

We describe a parallel implementation of the product of matrices.

1 Matrices distributed by block rows:

- matrix order n, p processes;

- assume n evenly divisible by p, $\overline{n} = \frac{n}{p}$;
- process q assigned rows $q\overline{n}, q\overline{n}+1, \ldots, (q+1)\overline{n}-1$.

2 Gather block of $\overline{n}$ columns onto each process.

3 Each process forms dot product of its rows with the gathered columns.

4 Repeat preceding two steps for each successive block of $\overline{n}$ columns.

5 Local submatrices stored as linear arrays in row-major order.

6 We don't want to overwrite B. So we allocate a block of order $n \times \overline{n}$ to store the column block.

7 Observe that on any process, the array entries that it contributes to the column block are not contiguous. The entries are grouped into subblocks of size $\overline{n}$ and there are $\overline{n}$ of them. Between the starts of successive rows in any column block, there are n elements.

8 Thus, we use the following arguments to *MPI_type_vector*:

- first argument is the number of rows or sublocks contributed by the process: $\overline{n}$;
- second argument is the number of contiguous elements to take from a row or sublock: $\overline{n}$;
- third argument is the number of elements between the starts of successive blocks: n;
- fourth argument is the type of the elements;
- fifth argument is storage for the new type.

9 After creating the type, with the call to *MPI_Type_vector*, before the type can be used in communication, it has to be *committed*. This allows the system to make optimizations that wouldn't be necessary if the type were only being used to make a more complex type.

10 The *Allgather* uses the address of the start of the block as its first argument. The count is only 1, since *gather_mpi_t* speci

es the entire block.

11 Note that in the column block, the entries contributed by a process are contiguous. Hence we just use a count of $\overline{n}^2$ and a type of *MPI_Float*. This says the received elements will be copied into a contiguous sequence of locations in the destination array.

5. RESULT AND EXPERIMENTS

The results of measurements for a computer with 4 physical and 8 logical cores are shown below (Table 1).

Table 1. Average results of 200 measurements

		size				average time, s
		100x100	500x500	1000x1000	5000x5000	
serial		0.002	0.274	2.488	312.853	78.904
algorithm#1	4	0.002	0.128	0.644	105.111	26.471
	8	0.002	0.070	0.544	90.668	22.821
	16	0.005	0.099	0.627	93.711	23.611
algorithm#2	4	0.002	0.104	0.702	106.237	26.761
	8	0.002	0.067	0.525	89.415	22.502
	16	0.004	0.089	0.624	95.641	24.090

As can be seen from the results, on 4, 8, 16 threads the developed programs for algorithm#1 and algorithm#2 works much faster and with an increase in the size of the input matrices N, this becomes more and more noticeable. However, an algorithm that uses MPI derived data type works a bit faster.

It is also worth noting that with 16 threads, pseudo-parallelism begins, when one thread takes resources from another.

6. CONCLUSION

In this paper, we implemented two parallel algorithms for multiplying two matrices A and B using MPI with different methods of transferring data to parallel streams.

As a result of this work, the skill of parallelizing calculations using mpi was obtained, which can be used in the future when working with large-scale projects that require big data reduction.

7. APPENDIX

The source code is located here: `https://github.com/vanSultan/parallel_algorithm/tree/main/lab_02`.