МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

# Звіт

З лабораторної роботи № 7 з дисципліни
«Технології паралельних обчислень»

Тема: «Розробка паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності»

**Виконав(ла)** *ІП-14 Бабіч Денис* _____

(шифр, прізвище, ім'я, по батькові)

**Перевірив** *Дифучина О. Ю.* _____

(шифр, прізвище, ім'я, по батькові)

Київ 2024

# ОСНОВНА ЧАСТИНА

**Мета роботи**: Розробка паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності.

1.1. Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» (див. лекцію та документацію стандарту MPI).

1.2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями.

1.3. Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні методів обміну повідомленнями «один-до-одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох».

```
PS E:\KPI\Parallel Computations\Labworks\7\bin> mpiexec -n 3 labwork7.exe
matrix_int_multiply_mpi_collective. Execution time: 0.558 seconds.
matrix_int_multiply_sequential. Execution time: 0.880 seconds.
MPI Collective vs Sequential: Equal
```

Рисунок 1.1 – Приклад виконання алгоритму

Таблиця 1.1 – Отримані результати прискорення

| Matrix Size | Sequential algorithm | Collective communication | | | | | |
|---|---|---|---|---|---|---|---|
| | | 4 processors | | 9 processors | | 16 processors | |
| | | Time | Speed up | Time | Speed up | Time | Speed up |
| 500 | 0.559 | 0.336 | 1.66369 | 0.173 | 3.23121 | 0.197 | 2.83756 |
| 1000 | 6.637 | 3.287 | 2.01917 | 1.933 | 3.43352 | 2.011 | 3.30035 |
| 1500 | 27.499 | 11.054 | 2.4877 | 7.486 | 3.67339 | 7.453 | 3.68966 |
| 2000 | 71.515 | 27.315 | 2.61816 | 17.778 | 4.02267 | 17.023 | 4.20108 |
| 2500 | 138.792 | 54.508 | 2.54627 | 38.013 | 3.65117 | 34.644 | 4.00623 |
| 3000 | 250.456 | 100.427 | 2.49391 | 74.138 | 3.37824 | 66.413 | 3.77119 |

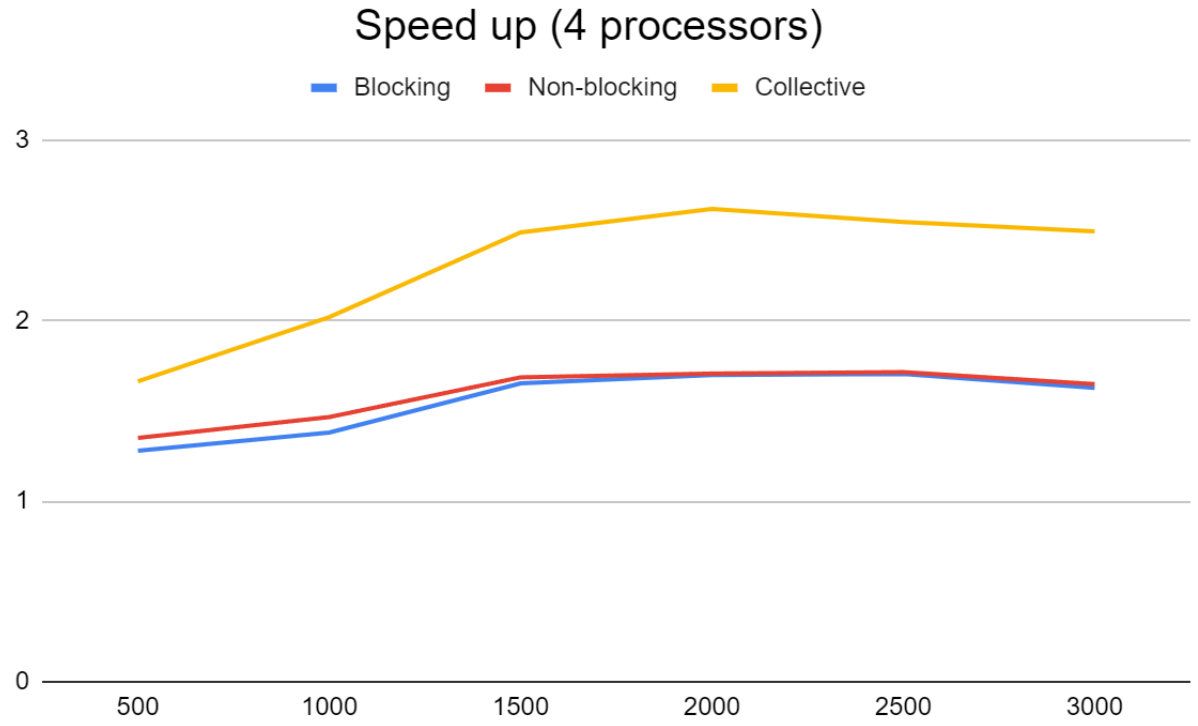


Рисунок 1.2 – Порівняння отриманих результатів прискорення

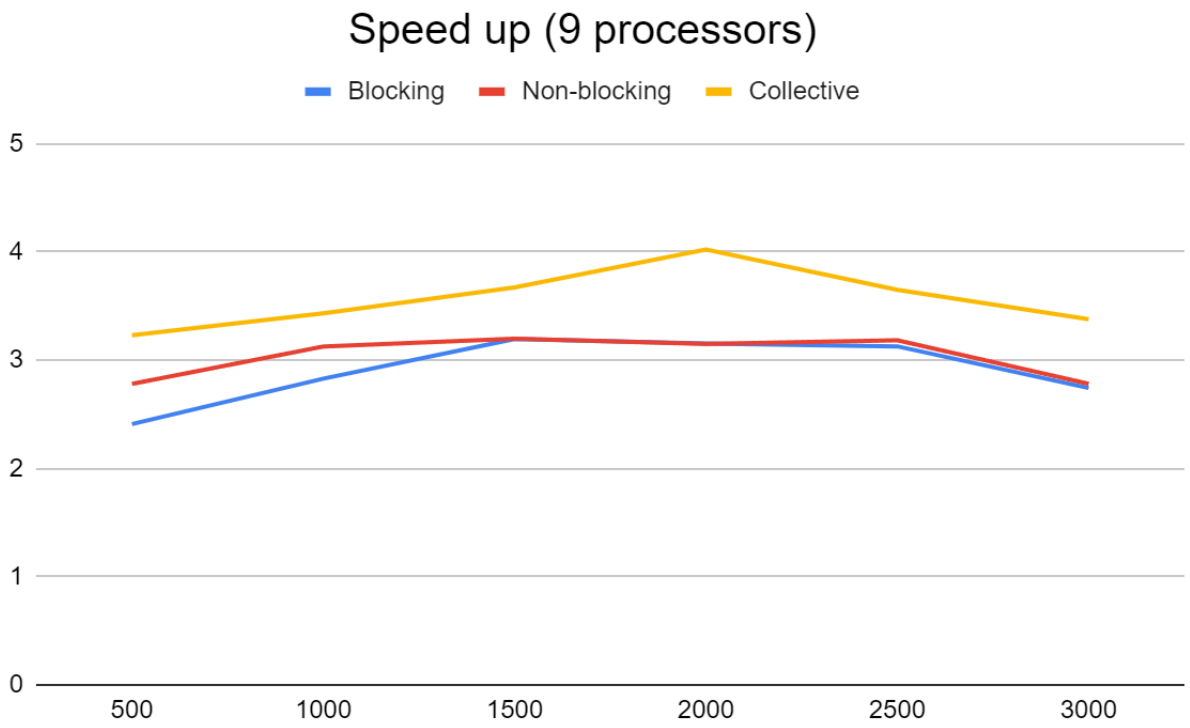


Рисунок 1.3 – Порівняння отриманих результатів прискорення

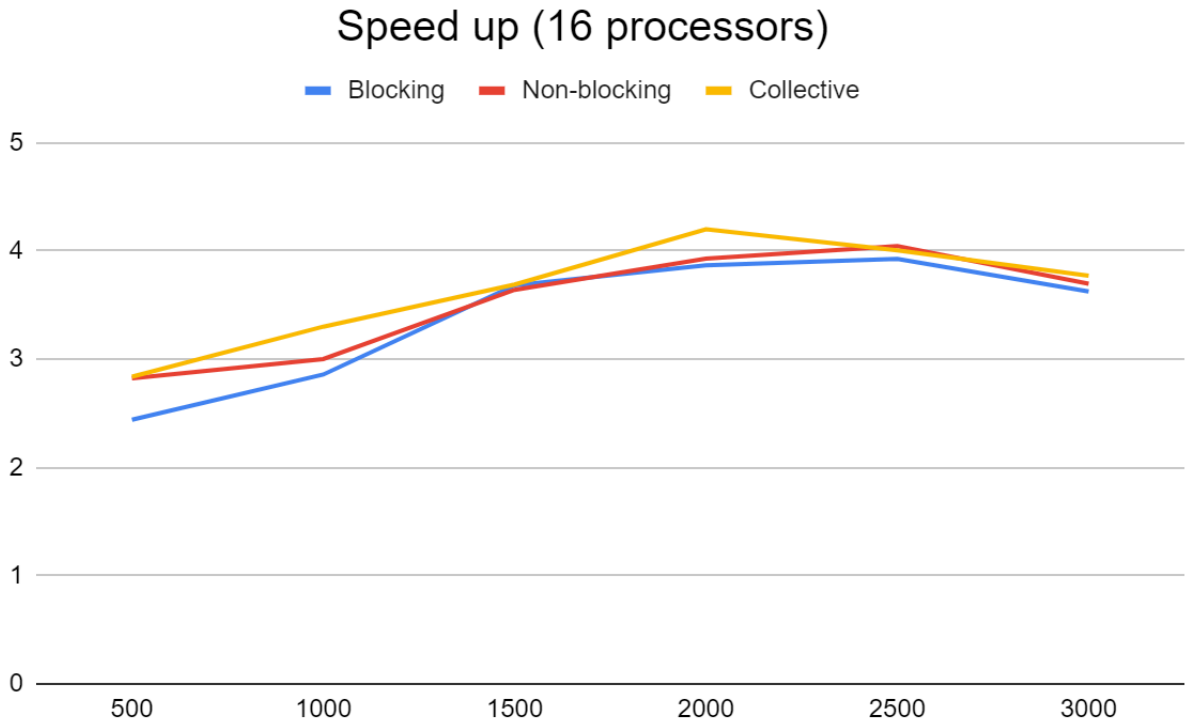


Рисунок 1.4 – Порівняння отриманих результатів прискорення

**Лістинг методу multipmatrix_int_multiply_mpi_collective**

```
MatrixInt* matrix_int_multiply_mpi_collective(const MatrixInt const* matrix1, const MatrixInt const* matrix2)
{
   int mpi_comm_rank;
   int mpi_comm_size;

   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_comm_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &mpi_comm_size);

   if (matrix1->columns != matrix2->rows)
   {
      perror("Matrices are not multipliable.");
      return NULL;
   }

   if (mpi_comm_size < 2)
   {
      perror("At least 2 MPI processors are required.");
      return NULL;
   }

   const clock_t TIMESTEP_START = clock();

      const int WORKER_PAYLOAD = matrix1->rows / mpi_comm_size * matrix1->columns;

   int *counts_buffer = (int*)malloc(mpi_comm_size * sizeof(int));
```

```
    if (counts_buffer == NULL)
    {
        perror("matrix_int_multiply_mpi_collective. Memory allocation failed");
        return NULL;
    }

    int *displacements_buffer = (int*)malloc(mpi_comm_size * sizeof(int));

    if (displacements_buffer == NULL)
    {
        perror("matrix_int_multiply_mpi_collective. Memory allocation failed");
        return NULL;
    }

    int index_start;
    int index_finish;
    int receiver_id = 0;

    for (int i = 0; i < mpi_comm_size; ++i)
    {
        index_start = i * WORKER_PAYLOAD;
                index_finish = (i == (mpi_comm_size - 1) ? (matrix1->rows *
matrix1->columns) : (index_start + WORKER_PAYLOAD));

        *(displacements_buffer + receiver_id) = index_start;
        *(counts_buffer + receiver_id) = (index_finish - index_start);

        ++receiver_id;
    }
```

```
    int receive_count = counts_buffer[mpi_comm_rank];

    int *receive_buffer = (int*)malloc(receive_count * sizeof(int));

    if (receive_buffer == NULL)
    {
        perror("matrix_int_multiply_mpi_collective. Memory allocation failed");
        return NULL;
    }

    int *matrix_row = (int*)malloc(matrix1->columns * sizeof(int));

    if (matrix_row == NULL)
    {
        perror("matrix_int_multiply_mpi_collective. Memory allocation failed");
        return NULL;
    }

    int *local_results = (int*)malloc((receive_count) * sizeof(int));

    if (local_results == NULL) {
        perror("matrix_int_multiply_mpi_collective. Memory allocation failed");
        return NULL;
    }

        int *global_results = (int*)malloc((matrix1->rows * matrix1->columns) *
sizeof(int));
    if (global_results == NULL) {
```

```
    perror("matrix_int_multiply_mpi_collective. Memory allocation failed");
    return NULL;
  }

  int const *flatten_data = matrix_int_get_flatten_data(matrix1);

      MPI_Scatterv(flatten_data, counts_buffer, displacements_buffer, MPI_INT,
receive_buffer, receive_count, MPI_INT, MPI_ROOT_RANK,
MPI_COMM_WORLD);

  int local_index = 0;
  int local_index_row;
  int local_index_column;
  int *local_matrix_column;
  index_start = 0;
  index_finish = counts_buffer[mpi_comm_rank];

  for (int i = index_start; i < index_finish; i += matrix1->columns) {
    for (int  j = 0; j < matrix2->columns; ++j) {
      local_results[local_index] = 0;

      for (int k = 0; k < matrix1->columns; ++k) {
        matrix_row[k] = receive_buffer[i + k];
      }

      local_matrix_column = matrix_int_get_column(matrix2, j);

      for (int k = 0; k < matrix1->columns; ++k) {
        local_results[local_index] += matrix_row[k] * local_matrix_column[k];
```

```
            }

            ++local_index;

            free(local_matrix_column);
        }
    }
        MPI_Gatherv(local_results, receive_count, MPI_INT, global_results, counts_buffer, displacements_buffer, MPI_INT, MPI_ROOT_RANK, MPI_COMM_WORLD);
    const MatrixInt const *RESULT = matrix_int_unflatten_data(matrix1->rows, matrix2->columns, global_results);

    free(matrix_row);
    free(flatten_data);
    free(counts_buffer);
    free(local_results);
    free(global_results);
    free(receive_buffer);
    free(displacements_buffer);

    const clock_t TIMESTEP_FINISH = clock();

    if (mpi_comm_rank == MPI_ROOT_RANK) {
        printf("matrix_int_multiply_mpi_collective. Execution time: %.3f seconds.\n", (double)(TIMESTEP_FINISH - TIMESTEP_START) / CLOCKS_PER_SEC);
    }
    return RESULT;
}
```

# ВИСНОВКИ

В ході виконання цієї лабораторної роботи було проведено дослідження методів колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» в контексті паралельного множення матриць за допомогою розподілених обчислень в MPI.

Алгоритм був реалізований з використанням різних методів обміну повідомленнями, що дозволило оцінити їх вплив на ефективність обчислень. Під час експериментів, варіюючи розмір матриць та кількість процесорів, на яких запускалася програма, було зареєстровано час виконання алгоритму для кожного методу обміну повідомленнями. Результати дослідження показали, що ефективність розподіленого обчислення алгоритму множення матриць суттєво залежить від вибору методу обміну повідомленнями. Було виявлено, що методи обміну повідомленнями «один-до-багатьох» та «багато-до-багатьох» забезпечують вищу ефективність в порівнянні з методами «один-до-одного» та «багато-до-одного».

Таким чином, можна зробити висновок про важливість вибору відповідного методу обміну повідомленнями для конкретної задачі та ресурсів, доступних для виконання обчислень. Завдяки цьому можна оптимізувати процес паралельного множення матриць та підвищити ефективність розподілених обчислень. Зокрема, методи обміну повідомленнями «один-до-багатьох» та «багато-до-багатьох» можуть бути особливо корисними при виконанні обчислень на великій кількості процесорів під час роботи з великими матрицями.