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 № 6 з дисципліни
«Технології паралельних обчислень»

Тема: «Розробка паралельного алгоритму множення матриць з використанням MPI-методів обміну повідомленнями «один-до-одного» 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 обміну повідомленнями «один-до-одного» та дослідження його ефективності.

1.1. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів блокуючого обміну повідомленнями.

```
PS E:\KPI\Parallel Computations\Labworks\6\bin> mpiexec -n 4 labwork6.exe
matrix_int_multiply_mpi_blocking. Execution time: 0.437 seconds.
matrix_int_multiply_sequential. Execution time: 0.559 seconds.
MPI Blocking vs Sequential: Equal
```

Рисунок 1.1 – Результати виконання функції з блокуючим обміном повідомлень

Таблиця 1.1 – Оцінка результатів отриманого прискорення

| Matrix Size | Sequential algorithm | Blocking communication | | | | | |
|---|---|---|---|---|---|---|---|
| | | 4 processors | | 9 processors | | 16 processors | |
| | | Time | Speed up | Time | Speed up | Time | Speed up |
| 500 | 0.559 | 0.437 | 1.27918 | 0.232 | 2.40948 | 0.229 | 2.44105 |
| 1000 | 6.637 | 4.809 | 1.38012 | 2.346 | 2.82907 | 2.322 | 2.85831 |
| 1500 | 27.499 | 16.645 | 1.65209 | 8.602 | 3.19681 | 7.467 | 3.68274 |
| 2000 | 71.515 | 42.12 | 1.69789 | 22.668 | 3.15489 | 18.489 | 3.86798 |
| 2500 | 138.792 | 81.451 | 1.70399 | 44.37 | 3.12806 | 35.344 | 3.92689 |
| 3000 | 250.456 | 153.874 | 1.62767 | 91.303 | 2.74313 | 69.071 | 3.62607 |

**Лістинг методу multipmatrix_int_multiply_mpi_blocking**

```
MatrixInt* matrix_int_multiply_mpi_blocking(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.");
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
      return NULL;
   }

   MPI_Status mpi_status;

   int *mpi_stripe;
   int mpi_index;
   int mpi_index_row;
   int mpi_index_start;
```

```
   int mpi_index_finish;
   int mpi_index_column;

            const MatrixInt const *RESULT = matrix_int_init(matrix1->rows,
matrix2->columns);

   if (RESULT == NULL)
   {
      return NULL;
   }

   if (mpi_comm_rank == MPI_ROOT_RANK)
   {
      const clock_t TIMESTEP_START = clock();

      const int TOTAL_TASKS = (matrix1->rows * matrix2->columns);
      const int WORKER_PAYLOAD = TOTAL_TASKS / mpi_comm_size;

      for (int i = 1; i < mpi_comm_size; ++i)
      {
         mpi_index_start = (i - 1) * WORKER_PAYLOAD;
               mpi_index_finish = (i == (mpi_comm_size - 1)) ? TOTAL_TASKS :
mpi_index_start + WORKER_PAYLOAD;

                                    MPI_Send(&mpi_index_start,  1,  MPI_INT,  i,
MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD);
                                   MPI_Send(&mpi_index_finish,  1,  MPI_INT,  i,
MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD);
      }
```

```
for (int i = 1; i < mpi_comm_size; ++i)
{
                                MPI_Recv(&mpi_index_start, 1, MPI_INT, i,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD, &mpi_status);
                                MPI_Recv(&mpi_index_finish, 1, MPI_INT, i,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD, &mpi_status);

    mpi_index = 0;
            mpi_stripe = (int*)malloc((mpi_index_finish - mpi_index_start) *
sizeof(int));

    if (mpi_stripe == NULL)
    {
      perror("matrix_int_multiply_mpi_blocking. Memory allocation failed.");
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
      return NULL;
    }

     MPI_Recv(mpi_stripe, (mpi_index_finish - mpi_index_start), MPI_INT, i,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD, &mpi_status);

    for (int j = mpi_index_start; j < mpi_index_finish; ++j)
    {
      mpi_index_row = j / matrix1->columns;
      mpi_index_column = j % matrix2->columns;

                matrix_int_set(RESULT, mpi_index_row, mpi_index_column,
*(mpi_stripe + mpi_index++));
```

```
        }

        free(mpi_stripe);
    }

    const clock_t TIMESTEP_FINISH = clock();

      printf("matrix_int_multiply_mpi_blocking. Execution time: %.3f seconds.\n",
(double)(TIMESTEP_FINISH - TIMESTEP_START) / CLOCKS_PER_SEC);
  }
  else
  {
    int sum;
    int *matrix_row;
    int *matrix_column;

              MPI_Recv(&mpi_index_start, 1, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD, &mpi_status);
             MPI_Recv(&mpi_index_finish, 1, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD, &mpi_status);

    mpi_index = 0;
    mpi_stripe = (int*)malloc((mpi_index_finish - mpi_index_start) * sizeof(int));

    if (mpi_stripe == NULL) {
      perror("matrix_int_multiply_mpi_blocking. Memory allocation failed.");
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
      return NULL;
    }
```

```
        for (int i = mpi_index_start; i < mpi_index_finish; ++i) {
            sum = 0;

            mpi_index_row = i / matrix1->columns;
            mpi_index_column = i % matrix2->columns;

            matrix_row = matrix_int_get_row(matrix1, mpi_index_row);
            matrix_column = matrix_int_get_column(matrix2, mpi_index_column);

            for (int j = 0; j < matrix1->columns; ++j) {
                sum += (*(matrix_row + j)) * (*(matrix_column + j));
            }
            *(mpi_stripe + mpi_index++) = sum;
            free(matrix_column);
        }

                    MPI_Send(&mpi_index_start, 1, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD);
                    MPI_Send(&mpi_index_finish, 1, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD);
                    MPI_Send(mpi_stripe, mpi_index, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD);

        free(mpi_stripe);
    }

    return RESULT;
}
```

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

```
PS E:\KPI\Parallel Computations\Labworks\6\bin> mpiexec -n 4 labwork6.exe
matrix_int_multiply_mpi_non_blocking. Execution time: 0.425 seconds.
matrix_int_multiply_sequential. Execution time: 0.558 seconds.
MPI Non-Blocking vs Sequential: Equal
```

Рисунок 1.2 – Результати виконання функції з не блокуючим обміном повідомлень

Таблиця 1.2 – Оцінка результатів отриманого прискорення

| Matrix Size | Sequential algorithm | Non-blocking communication | | | | | |
|---|---|---|---|---|---|---|---|
| | | 4 processors | | 9 processors | | 16 processors | |
| | | Time | Speed up | Time | Speed up | Time | Speed up |
| 500 | 0.559 | 0.414 | 1.35024 | 0.201 | 2.78109 | 0.198 | 2.82323 |
| 1000 | 6.637 | 4.528 | 1.46577 | 2.123 | 3.12624 | 2.211 | 3.00181 |
| 1500 | 27.499 | 16.314 | 1.68561 | 8.597 | 3.19867 | 7.553 | 3.6408 |
| 2000 | 71.515 | 41.91 | 1.70639 | 22.701 | 3.1503 | 18.201 | 3.92918 |
| 2500 | 138.792 | 80.951 | 1.71452 | 43.59 | 3.18403 | 34.313 | 4.04488 |
| 3000 | 250.456 | 151.974 | 1.64802 | 90.033 | 2.78182 | 67.737 | 3.69748 |

**Лістинг методу matrix_int_multiply_mpi_non_blocking**

```
MatrixInt* matrix_int_multiply_mpi_non_blocking(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.");
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
      return NULL;
   }

   int *mpi_stripe;
   int mpi_index;
   int mpi_index_row;
   int mpi_index_start;
   int mpi_index_finish;
   int mpi_index_column;
```

```
        const MatrixInt const *RESULT = matrix_int_init(matrix1->rows, matrix2->columns);

    if (RESULT == NULL)
    {
        return NULL;
    }

    if (mpi_comm_rank == MPI_ROOT_RANK)
    {
        const clock_t TIMESTEP_START = clock();

                    MPI_Status status_scattering, status_gathering_index_start, status_gathering_index_finish, status_gathering_stripe;
                    MPI_Request request_scattering, request_gathering_index_start, request_gathering_index_finish, request_gathering_stripe;

        const int TOTAL_TASKS = (matrix1->rows * matrix2->columns);
        const int WORKER_PAYLOAD = TOTAL_TASKS / mpi_comm_size;

        for (int i = 1; i < mpi_comm_size; ++i)
        {
            mpi_index_start = (i - 1) * WORKER_PAYLOAD;
                mpi_index_finish = (i == (mpi_comm_size - 1)) ? TOTAL_TASKS : mpi_index_start + WORKER_PAYLOAD;

                                MPI_Isend(&mpi_index_start, 1, MPI_INT, i, MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD, &request_scattering);
```

```
                MPI_Isend(&mpi_index_finish, 1, MPI_INT, i, MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD, &request_scattering);
        }

        for (int i = 1; i < mpi_comm_size; ++i)
        {
                MPI_Irecv(&mpi_index_start, 1, MPI_INT, i, MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD, &request_gathering_index_start);
                MPI_Irecv(&mpi_index_finish, 1, MPI_INT, i, MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD, &request_gathering_index_finish);

            MPI_Wait(&request_gathering_index_start, &status_gathering_index_start);
                MPI_Wait(&request_gathering_index_finish, &status_gathering_index_finish);

            mpi_index = 0;
                mpi_stripe = (int*)malloc((mpi_index_finish - mpi_index_start) * sizeof(int));

            if (mpi_stripe == NULL)
            {
                perror("matrix_int_multiply_mpi_non_blocking. Memory allocation failed.");
               MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
               return NULL;
            }
```

```
            MPI_Irecv(mpi_stripe, (mpi_index_finish - mpi_index_start), MPI_INT, i,
MPI_TAG_WORKER_MESSAGE,                    MPI_COMM_WORLD,
&request_gathering_stripe);
        MPI_Wait(&request_gathering_stripe, &status_gathering_stripe);

        for (int j = mpi_index_start; j < mpi_index_finish; ++j)
        {
           mpi_index_row = j / matrix1->columns;
           mpi_index_column = j % matrix2->columns;

                        matrix_int_set(RESULT,  mpi_index_row,  mpi_index_column,
*(mpi_stripe + mpi_index++));
        }

        free(mpi_stripe);
     }

     const clock_t TIMESTEP_FINISH = clock();

              printf("matrix_int_multiply_mpi_non_blocking.  Execution  time:  %.3f
seconds.\n",     (double)(TIMESTEP_FINISH     -     TIMESTEP_START)     /
CLOCKS_PER_SEC);
  }
  else
  {
     int sum;
     int *matrix_row;
     int *matrix_column;
```

```
MPI_Status status_index_start, status_index_finish, status_sending_stripe;
MPI_Request request_index_start, request_index_finish, request_sending_stripe, request_callback;

MPI_Irecv(&mpi_index_start, 1, MPI_INT, MPI_ROOT_RANK, MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD, &request_index_start);
MPI_Irecv(&mpi_index_finish, 1, MPI_INT, MPI_ROOT_RANK, MPI_TAG_ROOT_MESSAGE, MPI_COMM_WORLD, &request_index_finish);

MPI_Wait(&request_index_start, &status_index_start);
MPI_Wait(&request_index_finish, &status_index_finish);

mpi_index = 0;
mpi_stripe = (int*)malloc((mpi_index_finish - mpi_index_start) * sizeof(int));

if (mpi_stripe == NULL)
{
   perror("matrix_int_multiply_mpi_non_blocking. Memory allocation failed.");
   MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   return NULL;
}

for (int i = mpi_index_start; i < mpi_index_finish; ++i)
{
   sum = 0;

   mpi_index_row = i / matrix1->columns;
   mpi_index_column = i % matrix2->columns;
```

```
        matrix_row = matrix_int_get_row(matrix1, mpi_index_row);
        matrix_column = matrix_int_get_column(matrix2, mpi_index_column);

        for (int j = 0; j < matrix1->columns; ++j)
        {
           sum += (*(matrix_row + j)) * (*(matrix_column + j));
        }

        *(mpi_stripe + mpi_index++) = sum;

        free(matrix_column);
      }

                MPI_Isend(&mpi_index_start, 1, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD, &request_callback);
                MPI_Isend(&mpi_index_finish, 1, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_WORKER_MESSAGE, MPI_COMM_WORLD, &request_callback);
                MPI_Isend(mpi_stripe, mpi_index, MPI_INT, MPI_ROOT_RANK,
MPI_TAG_WORKER_MESSAGE,                    MPI_COMM_WORLD,
&request_sending_stripe);

      MPI_Wait(&request_sending_stripe, &status_sending_stripe);

      free(mpi_stripe);
   }

   return RESULT;
}
```

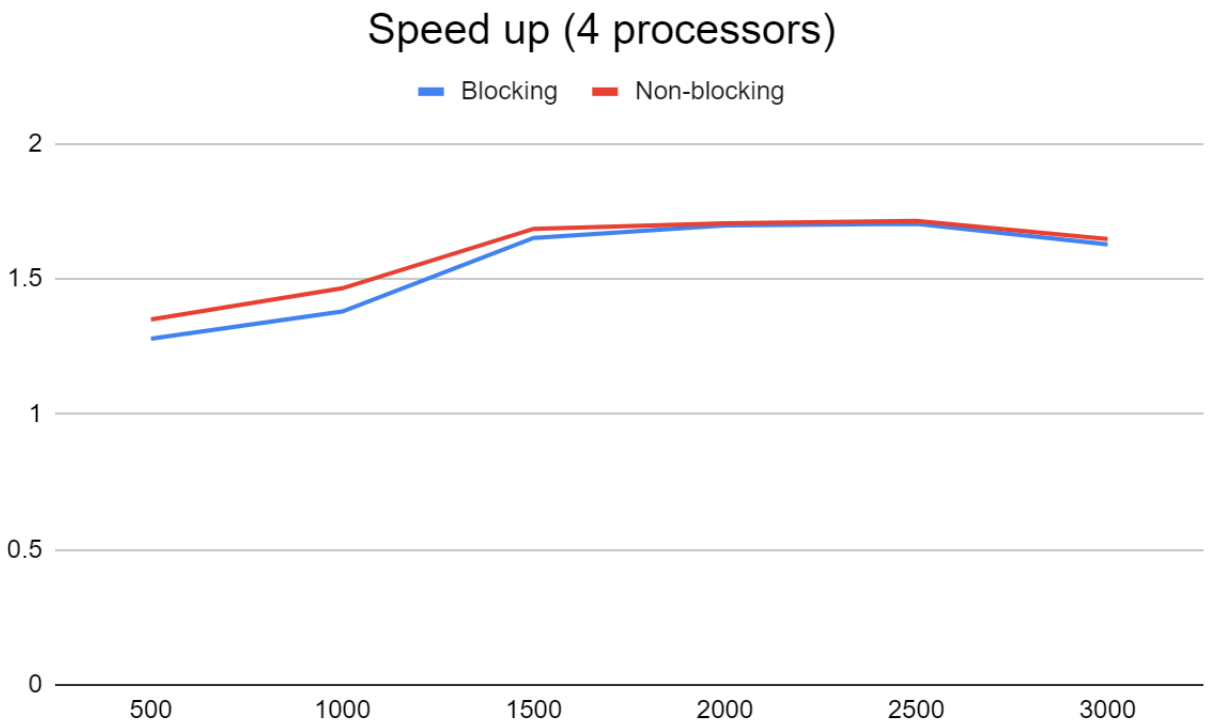


Рисунок 1.4 – Порівняння отриманого прискорення на 4 процесорах

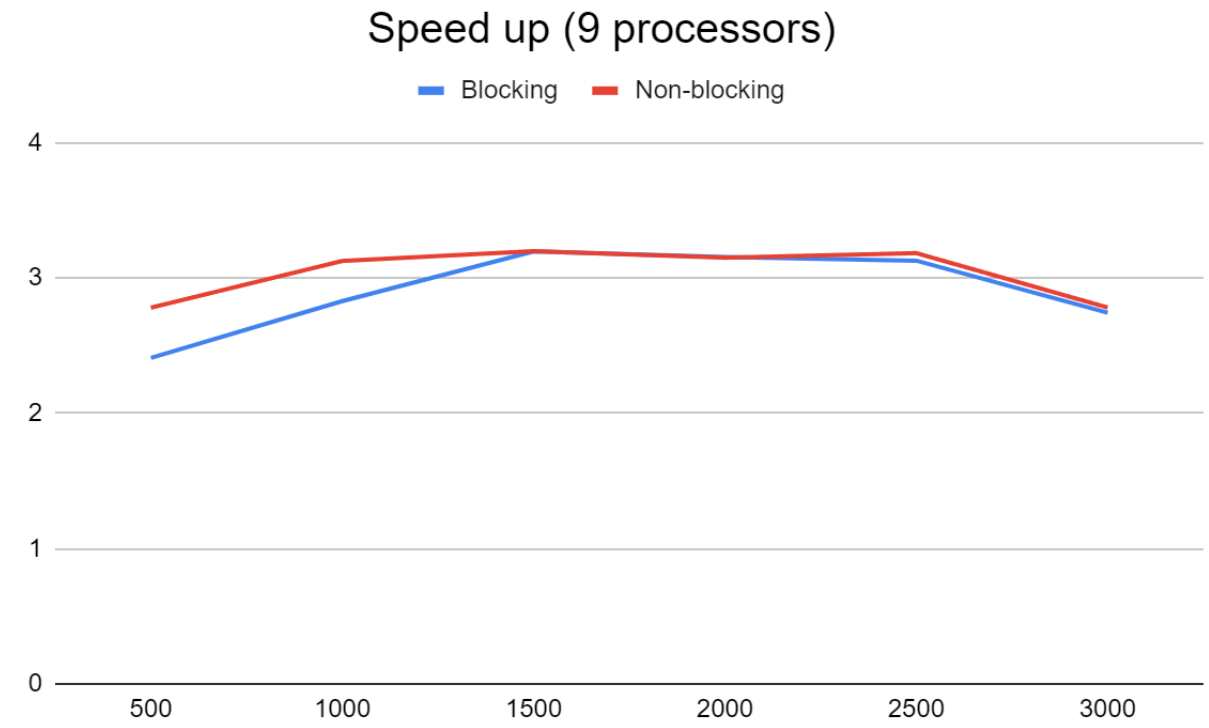


Рисунок 1.5 – Порівняння отриманого прискорення на 9 процесорах

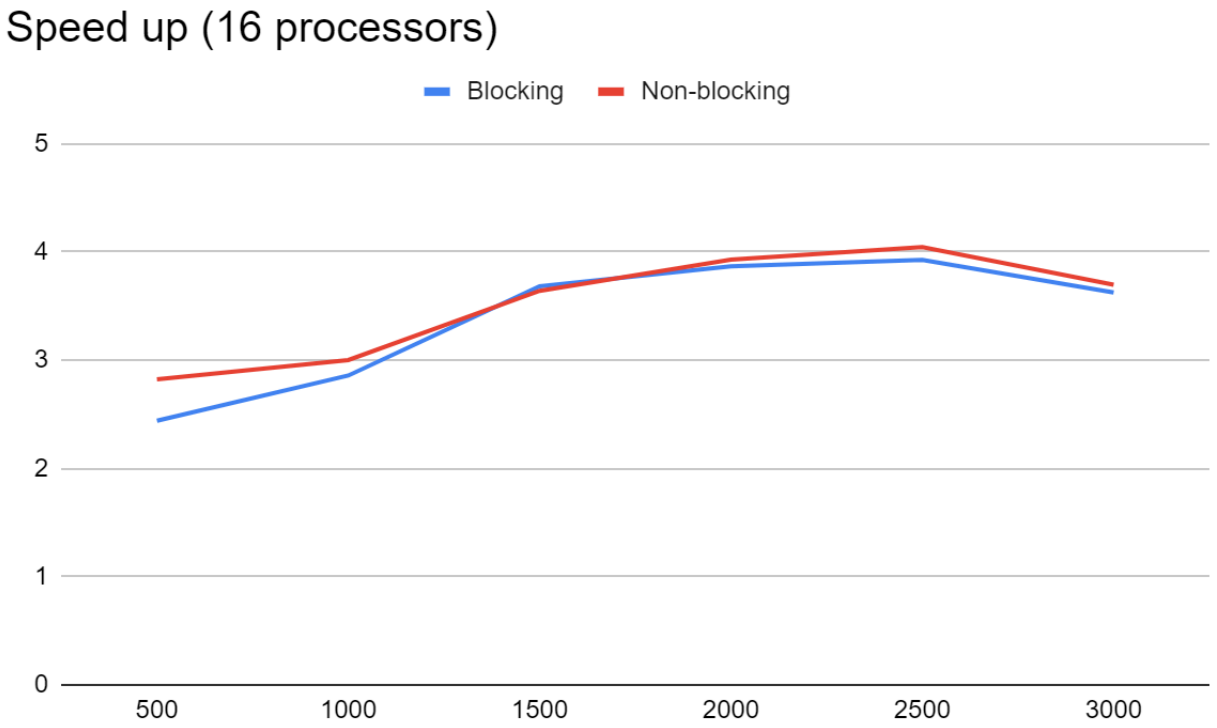


Рисунок 1.6 – Порівняння отриманого прискорення на 16 процесорах

# ВИСНОВКИ

Під час виконання цієї лабораторної роботи було реалізовано алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI. Алгоритм був реалізований двічі: один раз з використанням методів блокуючого обміну повідомленнями і другий раз з використанням методів неблокуючого обміну повідомленнями.

Під час проведення експериментів, варіюючи розмір матриць та кількість вузлів, на яких запускалася програма, було зареєстровано час виконання обох варіантів алгоритму. Результати дослідження показали, що ефективність розподіленого обчислення алгоритму множення матриць залежить від вибору методу обміну повідомленнями. Було виявлено, що алгоритм з використанням неблокуючих методів обміну повідомленнями виявився ефективнішим в порівнянні з алгоритмом, що використовує блокуючі методи. Це може бути пояснено тим, що неблокуючі методи дозволяють більш ефективно розподіляти обчислення між вузлами, що дозволяє досягти більшої продуктивності при оптимальній кількості вузлів та розмірах матриць.

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.