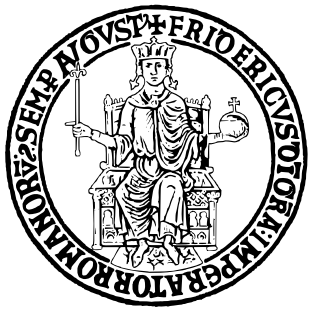
Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione



Parallel and Distributed Computing submissions

Giuliano Aiello

2025

Contents

Chapter 1

Prolusion

1.1 Goal

A comprehensive overview of a Parallel and Distributed Computing project developed in `C` is presented. The project consists of multiple modules delivered in incremental phases.

Every algorithm is implemented within the **Single Program Multiple Data** parallel model.

This report is not intended as a user guide, but rather aims to describe the project's exploration of Parallel and Distributed Computing techniques, leveraging High Performance Computing in certain instances.

1.2 Environment

The project was entirely developed on macOS with the help of Xcode IDE.
Naturally, this will mainly impact the build process.

1.3 Project directory layout

The structure of the project's root directory is outlined below.

```
parallel-distributed-computing/
├── common/
├── hpc/
│   ├── gemm/
│   ├── matmatblock/
│   ├── matmatdist/
│   └── matmatthread/
├── laplace/
├── maxsum/
├── ringsum/
└── parallel-distributed-computing.entitlements
```

`common` package serves as a library of utility functions designed to support and be reused by various modules across the project.

The remaining directories represent the individual project modules, which constitute the deliverables of the project. Within each module, the directory structure follows a standard format:

```
<module>/
├── build/
│   ├── deploy-cluster.pbs
│   └── Makefile
├── src/
│   ├── <module>/
│   │   ├── <module>.c
│   │   └── <module>.h
│   └── main.c
├── config.sh
└── run.sh
```

Most parts of the `main.c` files are provided by the project supervisor.

1.4 Build

The project was primarily compiled using the Clang compiler.
The build process was carried out either through the `Makefile` (some of which support compilers other than Clang) or via Xcode.

Regardless of the build process, every module of the project was compiled with:

- `-O3` optimization flag to maximize performance.

1.4.1 Libraries

The following are the dynamically linked libraries integrated into the project.

- `math.h`
- `mpi.h`
- `omp.h`
- `stdio.h`
- `stdbool.h`
- `stdlib.h`
- `sys/time.h`
- `unistd.h`

1.4.2 Xcode

When it came to build with the Xcode, the development process adhered to the workflow and conventions defined by the chosen IDE, leveraging its built-in tools and features to organize and manage the project. Particularly, this includes:

- Xcode targets
- Xcode schemes
- Xcode `.entitlements` file

1.5 Run

1.5.1 Network Requirement

Running an MPI module with no internet connection, makes the following error occur:

```
[Giulianos-MacBook-Pro.local:05355] ptl_tool: problems getting address for
index 0 (kernel index -1)
--------------------------------------------------------------------------
The PMIx server's listener thread failed to start. We cannot continue.
--------------------------------------------------------------------------
Program ended with exit code: 213
```

Chapter 2

Maxsum

2.1 Problem at a glance

Given a matrix, the objective is to identify the row with the maximum squared root sum.

2.2 Algorithm

2.2.1 Parallelization

The algorithm employs a multithreaded approach, each thread can compute independently its local maximum squared root sum. Once a thread has completed its task, a synchronization is needed with the other threads in order to determine the overall maximum squared root sum.

2.2.2 Pseudocode

Input:

- global matrix
- number of thread(s) for parallel execution

Steps:

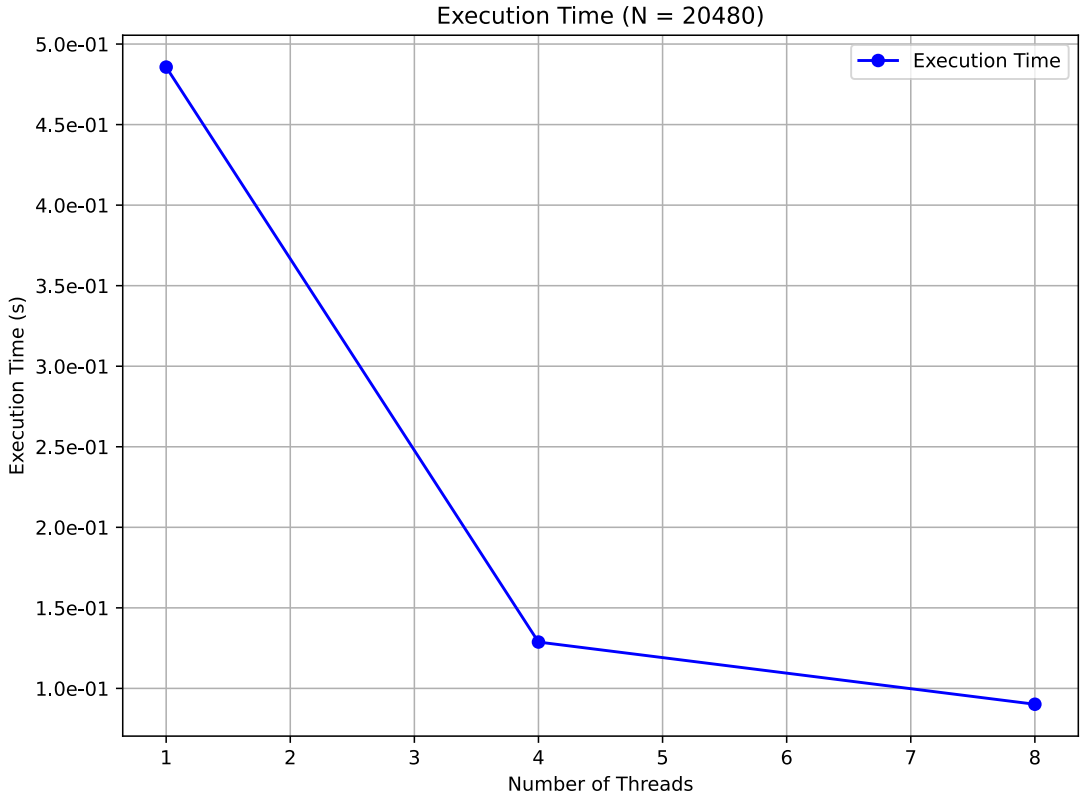
1. Compute the indices of the global matrix corresponding to the local matrix of the thread.
2. Compute the local maximum squared root sum.
3. Perform a thread-safe update of the overall maximum squared root sum variable.

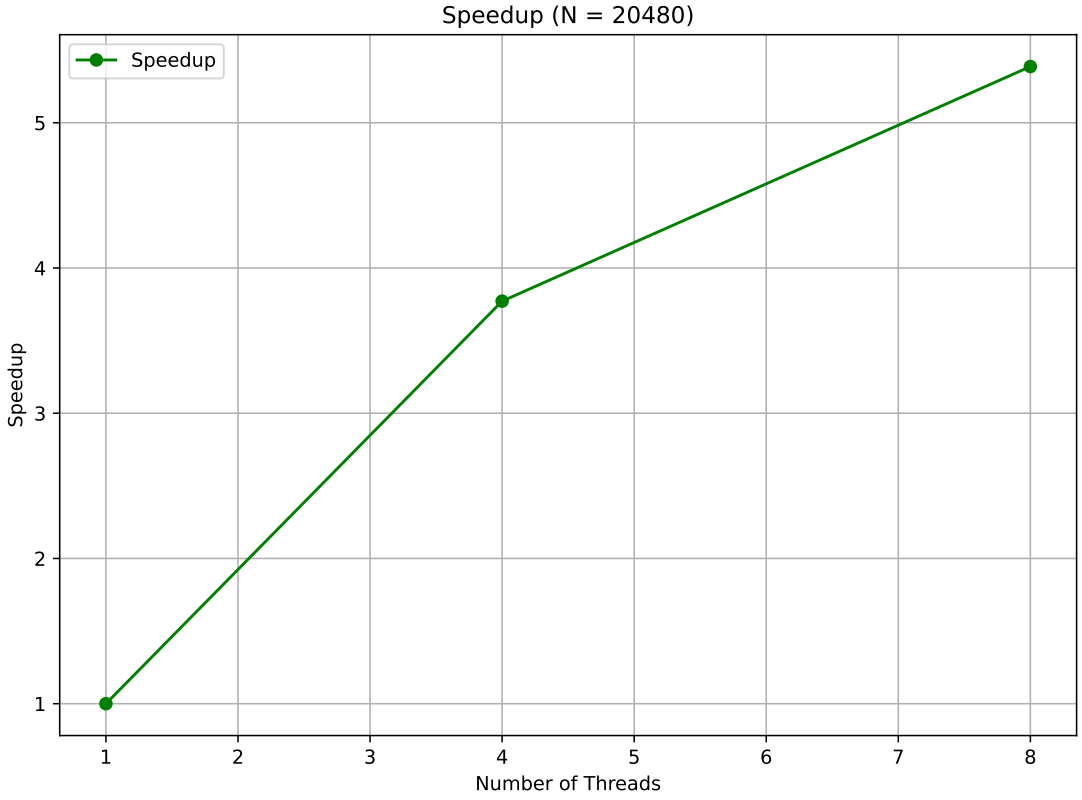
Algorithm 1: Maxsum

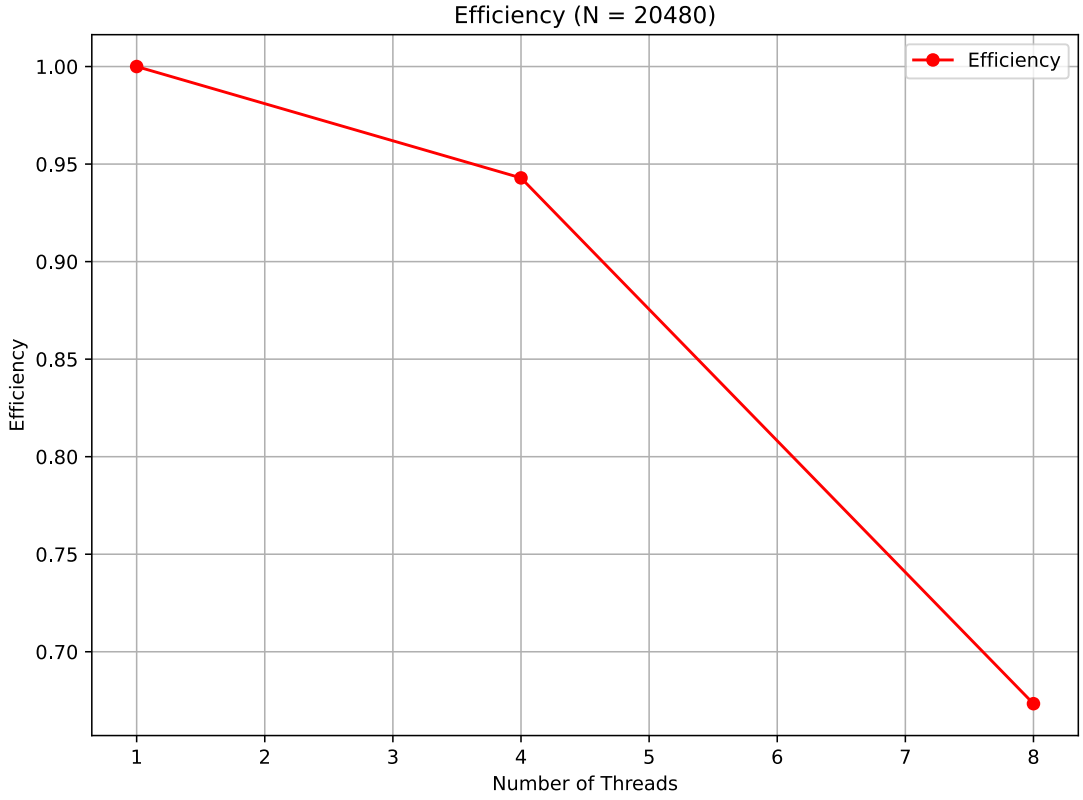
```
Input  : matrix, numThreads
Output: maxSumOverall

STARTTHREADS(numThreads)
startRow ← getStartingRowIndex(threadRank)
endRow ← getEndingRowIndex(threadRank)
for currRow ← startRow to endRow do
    currSum ← squareRootSum(currRow)
    if currSum > maxSum then
        maxSum ← currSum
    end
    CRITICAL
    if maxSumOverall < maxSum then
        maxSumOverall ← maxSum
    end
end
```

2.2.3 Performance Analysis




Speedup (N = 20480)
Speedup
Speedup
Number of Threads
1
2
3
4
5
6
7
8


Efficiency (N = 20480)
Efficiency
1.00
0.95
0.90
0.85
0.80
0.75
0.70
Efficiency
1
2
3
4
5
6
7
8
Number of Threads

Chapter 3

Laplace

3.1 Problem at a glance

Given a matrix and a number of iterations, the objective is to compute the discrete Laplacian of the matrix over the specified iterations.

3.2 Algorithm

3.2.1 Parallelization

The algorithm employs a multiprocess approach. A synchronization between processes is needed from the outset.

The global matrix is distributed among the processes, harnessing the advantages of parallelism, representing a spatial advantage other than a temporal one.

The algorithm comes in two versions: in the first one, communication between processes takes place in a classic way where sending a message "blocks" the execution and will resume it when a reception happens; in the second one, sending a message won't "block" the execution, instead, execution is blocked when the expected receive message needs to be accessed.
Clearly, the second version is expected to be slightly faster than the other one.

3.2.2 Pseudocode

Input:

- local process matrix
- number of iteration(s)

Steps:
For each iteration:

1. If $P_i \neq P_1$ Then
   (a) Send to P_{i-1} the local first row
   (b) Receive from P_{i-1} its last row
2. If $P_i \neq P_n$ Then
   (a) Receive from P_{i+1} its first row
   (b) Send to P_{i+1} the local last row
3. Compute the Laplacian considering only the local inner matrix, the matrix boundaries are left out.
4. If $P_i \neq P_1$
   (a) Compute the Laplacian considering only the local top row, with the auxiliary last row from the previous process.
5. If $P_i \neq P_n$
   (a) Compute the Laplacian considering only the local bottom row, with the auxiliary first row from the next process.

An auxiliary matrix is involved to facilitate the computation.

Algorithm 2: Laplace

```
Input  : matrix, iterations
Output: outputMatrix

for iter ← 1 to iterations do
    if P_i ≠ P_1 then
        SEND(P_{i−1}, localFirstRow)
        RECV(P_{i−1}, receivedLastRow)
    end
    if P_i ≠ P_n then
        RECV(P_{i+1}, receivedFirstRow)
        SEND(P_{i+1}, localLastRow)
    end
    laplacian(inner(matrix))
    if P_i ≠ P_1 then
        laplacian(matrix, receivedLastRow)
    end
    if P_i ≠ P_n then
        laplacian(matrix, receivedFirstRow)
    end
    copy(outputMatrix, inner(matrix))
    if P_i ≠ P_1 then
        copy(outputMatrix, lastRow(matrix))
    end
    if P_i ≠ P_n then
        copy(outputMatrix, firstRow(matrix))
    end
end
```

Algorithm 3: Laplace non blocking

```
Input  : matrix, iterations
Output: outputMatrix

for iter ← 1 to iterations do
    if P_i ≠ P_1 then
        IMMEDIATE_SEND(P_{i-1}, localFirstRow, SEND_NEXTTOPREV)
        IMMEDIATE_RECV(P_{i-1}, receivedLastRow, RECV_PREVTONEXT)
    end
    if P_i ≠ P_n then
        IMMEDIATE_RECV(P_{i+1}, receivedFirstRow, RECV_NEXTTOPREV)
        IMMEDIATE_SEND(P_{i+1}, localLastRow, SEND_PREVTONEXT)
    end
    laplacian(inner(matrix))
    if P_i ≠ P_1 then
        WAIT RECV_PREVTONEXT
        laplacian(matrix, receivedLastRow)
    end
    if P_i ≠ P_n then
        WAIT RECV_NEXTTOPREV
        laplacian(matrix, receivedFirstRow)
    end
    copy(outputMatrix, inner(matrix))
    if P_i ≠ P_1 then
        WAIT SEND_NEXTTOPREV
        copy(outputMatrix, firstRow(matrix))
    end
    if P_i ≠ P_n then
        WAIT SEND_PREVTONEXT
        copy(outputMatrix, lastRow(matrix))
    end
end
```

3.2.3 Performance Analysis

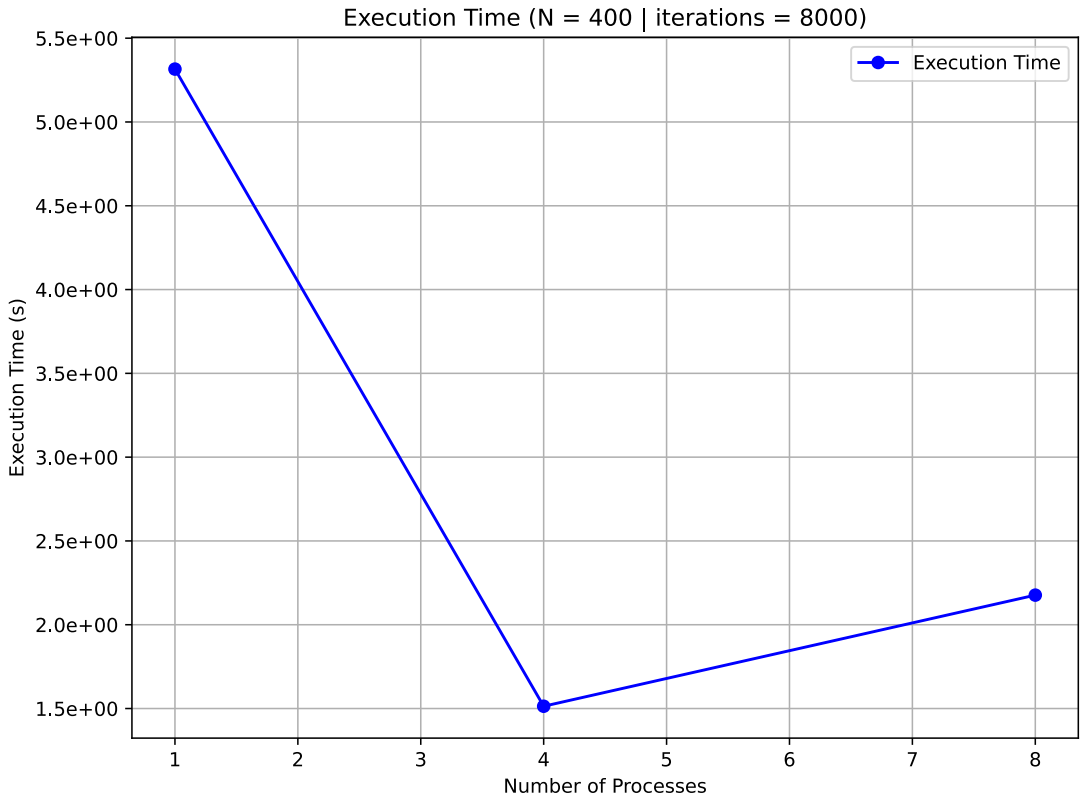


Figure 3.1: *Laplace algorithm execution time*

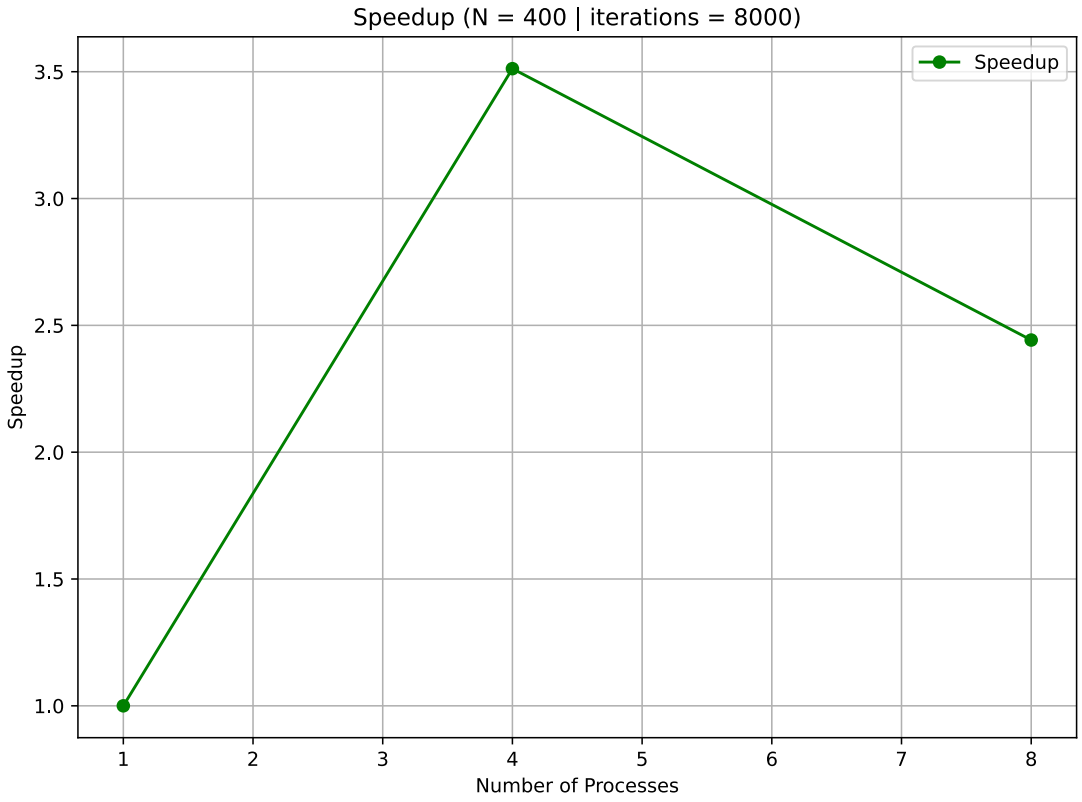


Figure 3.2: *Laplace algorithm speedup*

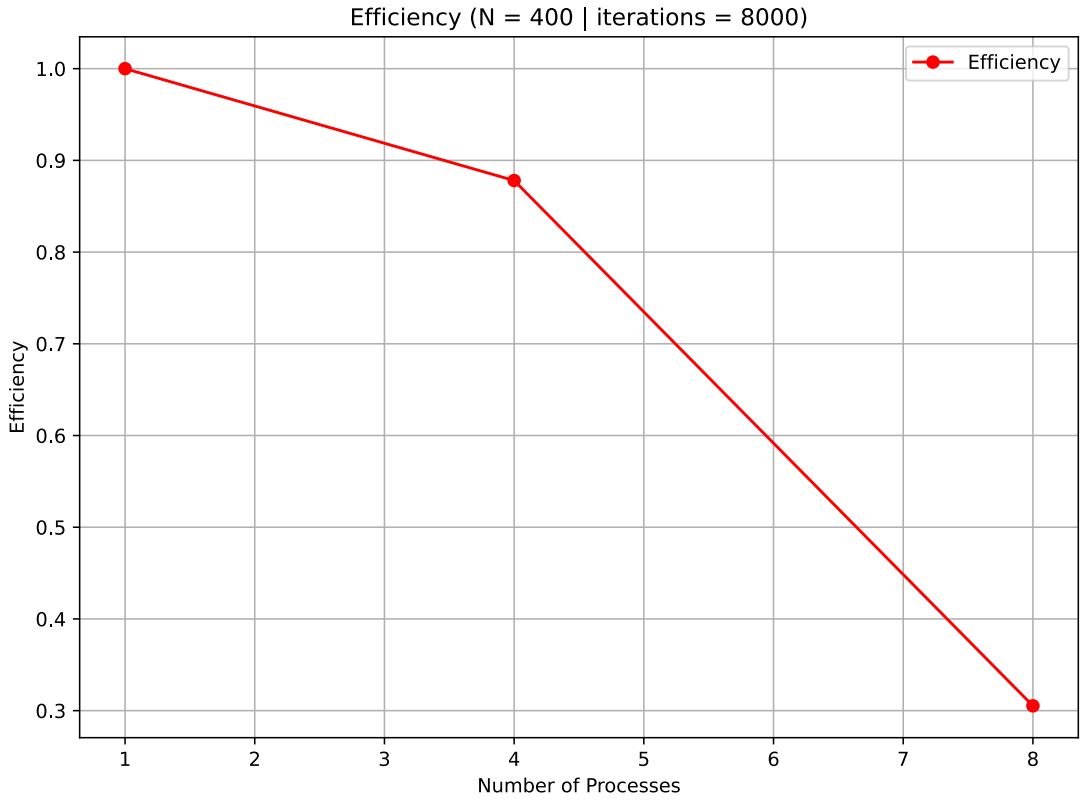


Figure 3.3: *Laplace algorithm efficiency*

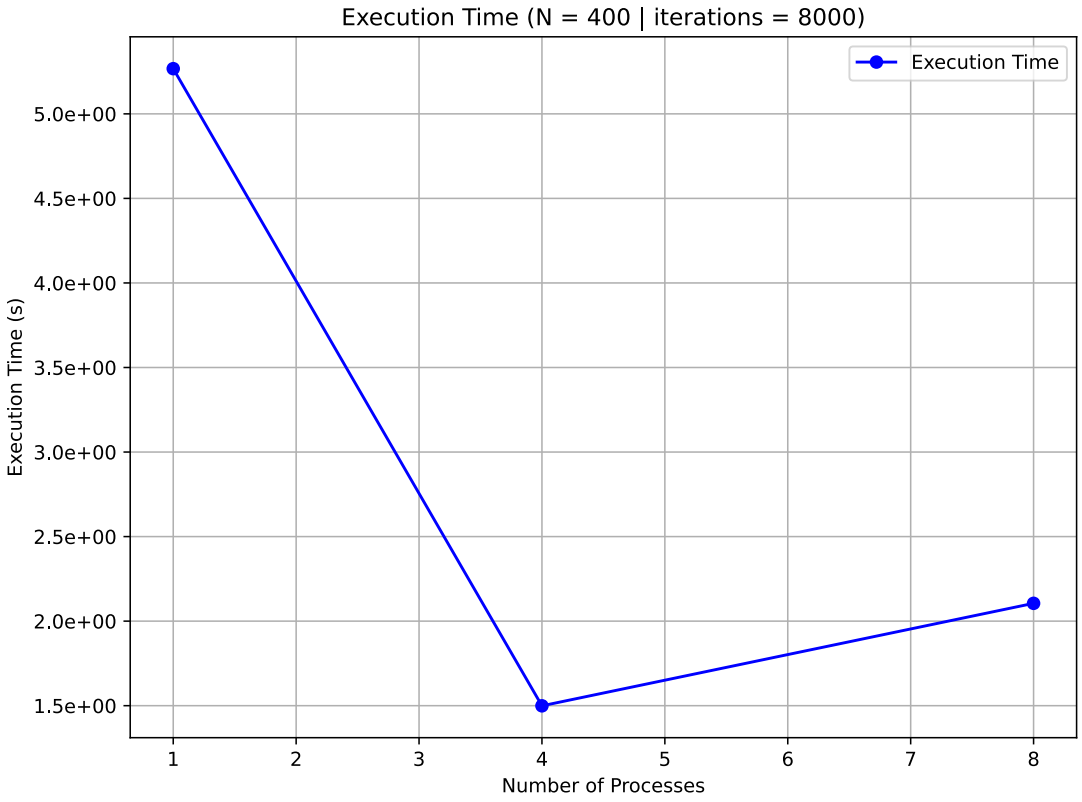


Figure 3.4: *Laplace non-blocking algorithm execution time*

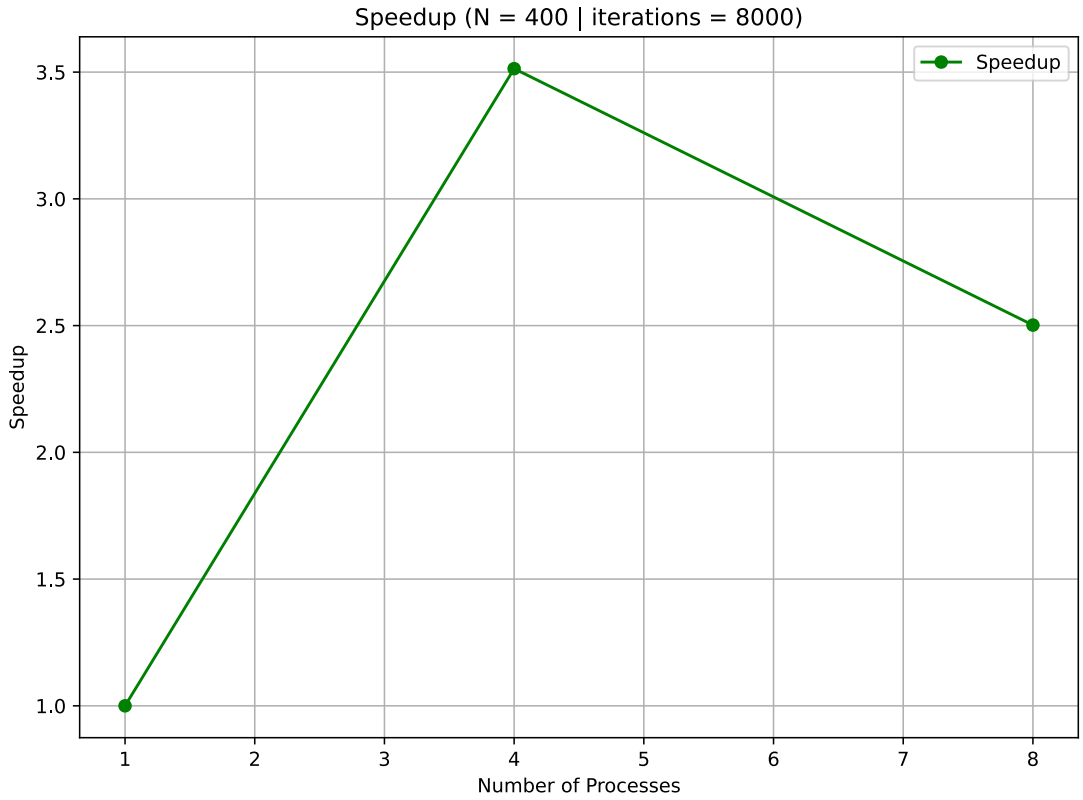


Figure 3.5: *Laplace non-blocking algorithm speedup*

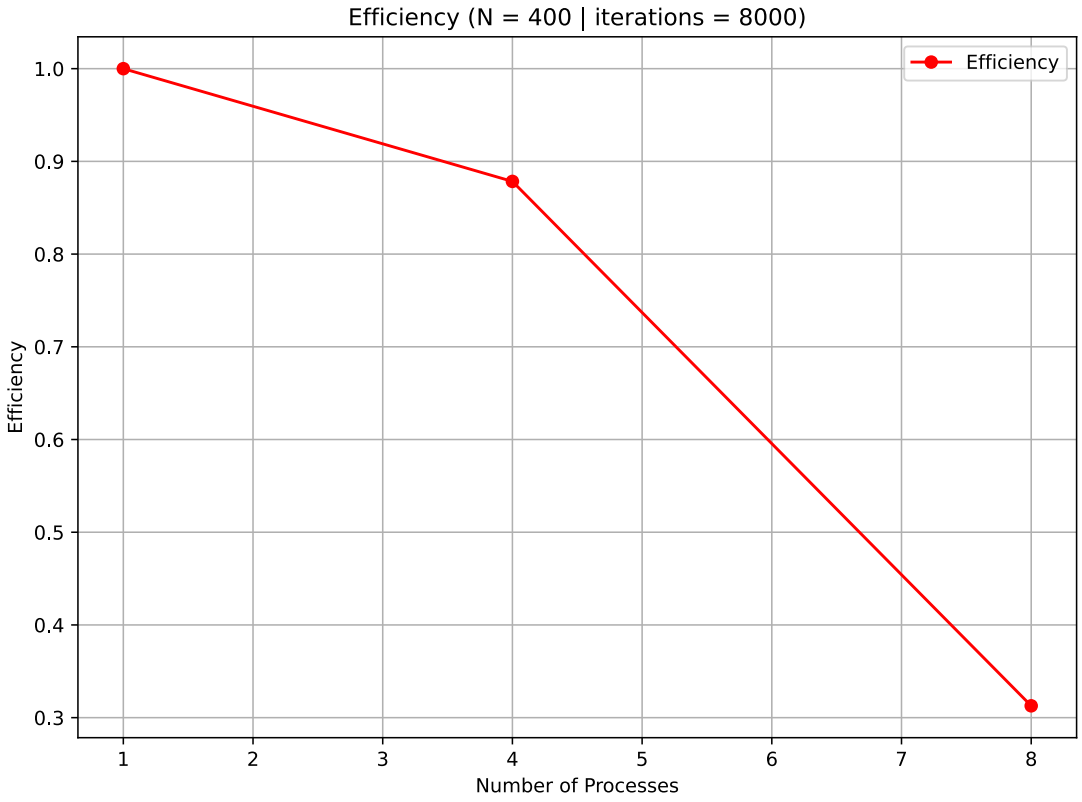


Figure 3.6: *Laplace non-blocking algorithm efficiency*

Chapter 4

MatMatikj

4.1 Problem at a glance

Given matrices A, B, C, the goal is to compute the GeMM operation $C = C + AB$ by iterating over the matrices while leveraging the ikj ordering of indices. This increase cache memory utilization.

4.2 Algorithm

4.2.1 Parallelization

No parallelization is planned for this algorithm.

4.2.2 Pseudocode

Input:

- $A^{N1 \times N2}$, $B^{N2 \times N3}$, $C^{N1 \times N3}$ matrices

Steps:

- Iterate over the three input matrices using the 'ikj' order
- Compute $C = C + AB$ incrementally for each step

Algorithm 4: MatMatikj

Input : $A^{N1 \times N2}$, $B^{N2 \times N3}$, $C^{N1 \times N3}$
Output: $C^{N1 \times N3}$

for $i \leftarrow 1$ **to** $N1$ **do**
  for $k \leftarrow 1$ **to** $N2$ **do**
    for $j \leftarrow 1$ **to** $N3$ **do**
      $\mathbf{C}(i,j) = \mathbf{C}(i,j) + \mathbf{A}(i,k) \cdot \mathbf{B}(k,j)$
    end
  end
end

If a matrix row fits entirely in the cache line as a cache block, number of main memory accesses is $\mathcal{O}(N^2)$, otherwise it $\mathcal{O}(N^3)$.

Chapter 5

MatMatBlock

5.1 Problem at a glance

Given matrices A, B, C, $blockN1$, $blockN2$, $blockN3$, the goal is to compute the GeMM operation $C = C + AB$ in 'blocks', by leveraging the best performance of MatMatikj algorithm ($\mathcal{O}(N^2)$ main memory accesses).

5.2 Algorithm

5.2.1 Parallelization

No parallelization is planned for this algorithm.

5.2.2 Pseudocode

Input:

- $A^{N1 \times N2}$, $B^{N2 \times N3}$, $C^{N1 \times N3}$ matrices
- $blockN1$, $blockN2$, $blockN3$ block dimensions of $N1$, $N2$, $N3$

Steps:

- Iterate over the three input matrices in steps of $blockN1$, $blockN2$, $blockN3$
- Execute MatMatikj algorithm for each step

Algorithm 5: MatMatBlock

Input : $A^{N1\times N2}$, $B^{N2\times N3}$, $C^{N1\times N3}$, $blockN1$, $blockN2$, $blockN3$
Output: $C^{N1\times N3}$

for $ii \leftarrow 1$ **to** $N1$ **by** $blockN1$ **do**
  for $jj \leftarrow 1$ **to** $N2$ **by** $blockN2$ **do**
    for $kk \leftarrow 1$ **to** $N3$ **by** $blockN3$ **do**
      $MatMatikj(\mathbf{A}(ii, jj), \mathbf{B}(jj, kk), \mathbf{C}(ii, kk))$
    end
  end
end

Chapter 6

MatMatThread

6.1 Problem at a glance

Given matrices A, B, C, $blockN1$, $blockN2$, $blockN3$, a number of $NTROW \times NTCOL$ threads, the goal is to compute the GeMM operation $C = C + AB$ in a multithreaded MatMatBlock approach.

6.2 Algorithm

6.2.1 Parallelization

The algorithm employs a multithreaded approach where each thread is responsible for executing a single GeMM operation of a C block.

Each thread can compute independently its local GeMM operation, no synchronization is needed among the threads.

6.2.2 Pseudocode

Input:

- $A^{N1 \times N2}$, $B^{N2 \times N3}$, $C^{N1 \times N3}$ matrices
- $blockN1$, $blockN2$, $blockN3$ block dimensions of $N1$, $N2$, $N3$ (used by MatMatBlock algorithm)
- $NTROW, NTCOL$ number of thread(s) on rows and columns of C that will be assigned to compute part of the whole GeMM operation on C

Steps:

- Compute the indices of the global matrix corresponding to the local matrix of the thread.
- Execute MatMatBlock algorithm

Algorithm 6: MatMatThread

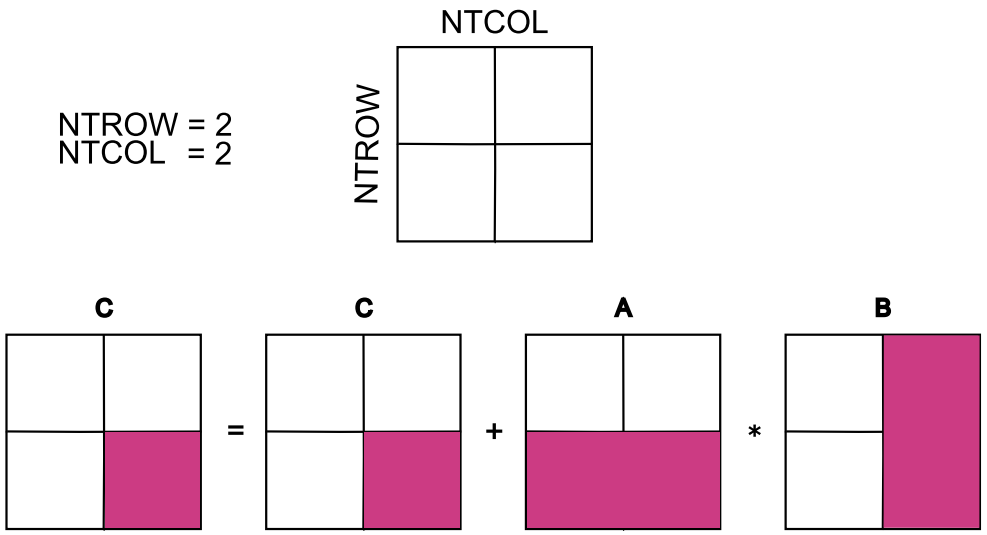
Input : $A^{N1\times N2}$, $B^{N2\times N3}$, $C^{N1\times N3}$, $blockN1$, $blockN2$, $blockN3$, $NTROW$, $NTCOL$
Output: $C^{N1\times N3}$

$numThreads \leftarrow NTROW * NTCOL$

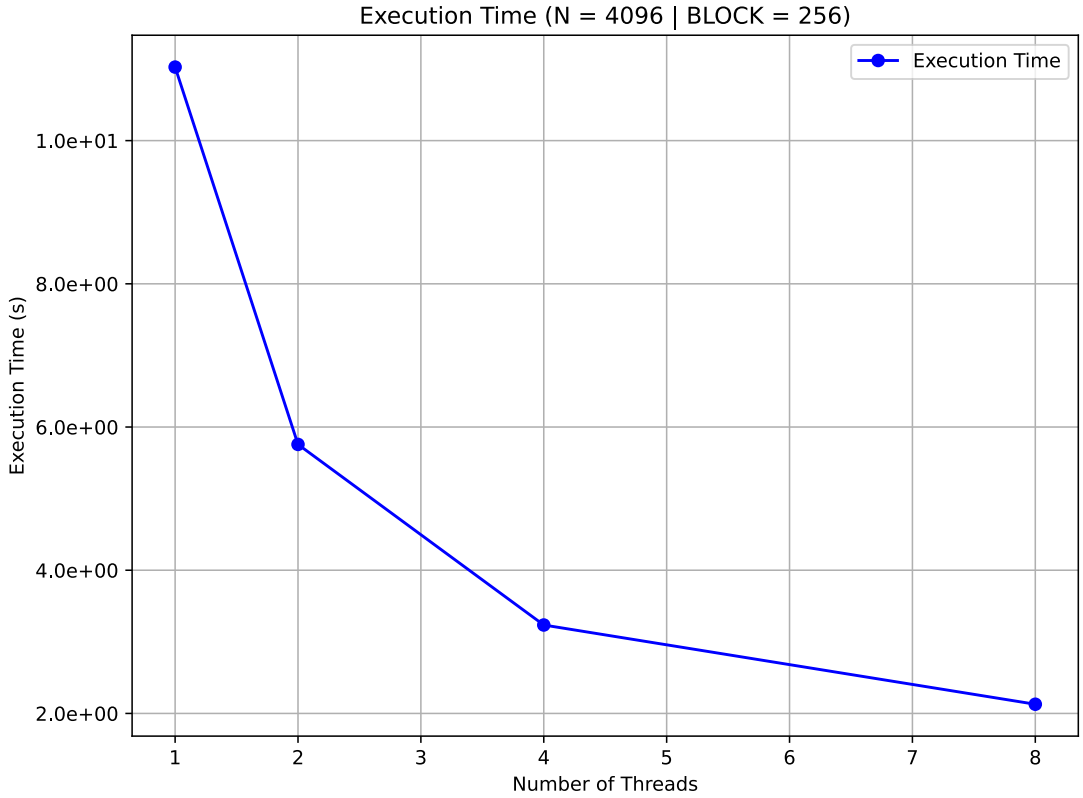
`STARTTHREADS(numThreads)`

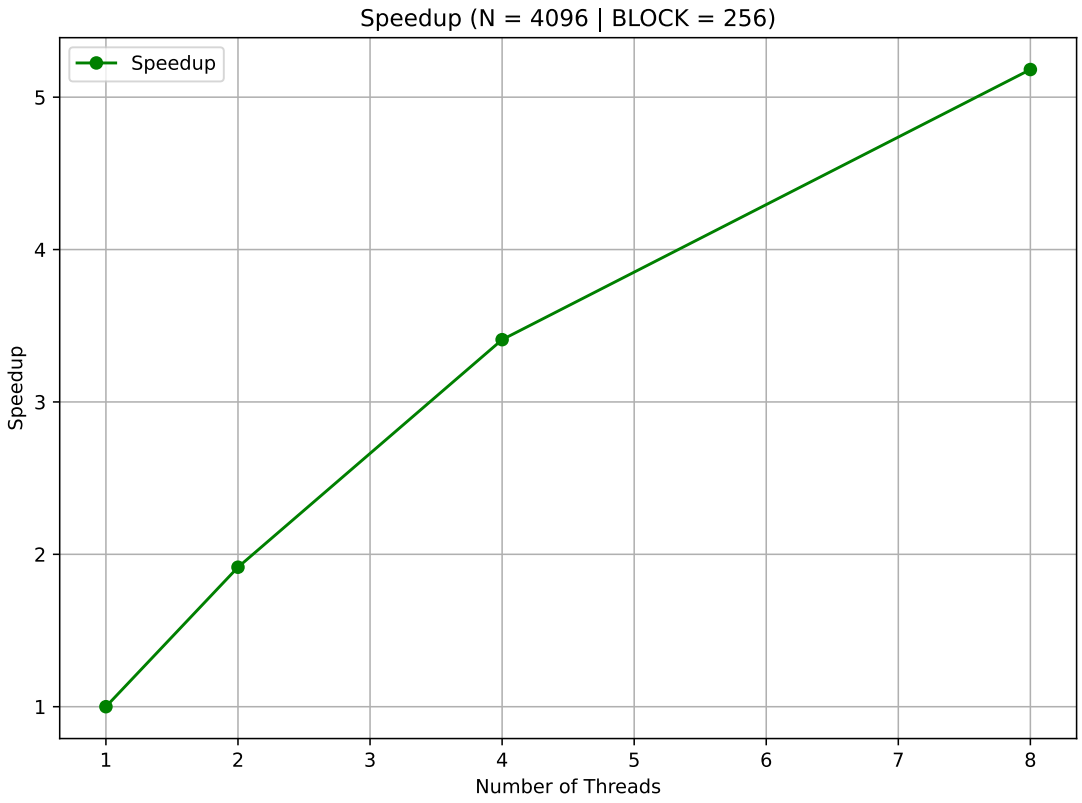
$startRow \leftarrow getStartingRowIndex(threadRank)$
$startCol \leftarrow getStartingColIndex(threadRank)$

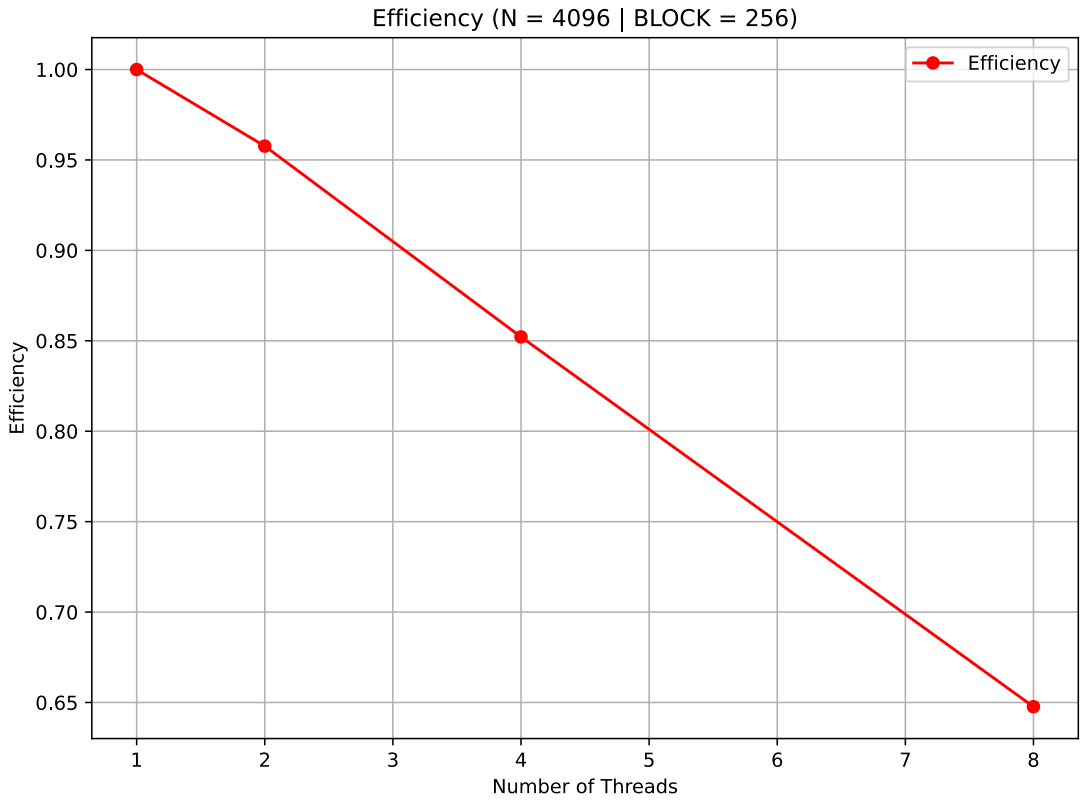
$MatMatBlock(\mathbf{A}^{(N1/NTROW)\times N2}(startRow, 0),$
$\mathbf{B}^{N2\times(N3/NTCOL)}(0, startCol),$
$\mathbf{C}^{(N1/NTROW)\times(N3/NTCOL)}(startRow, startCol),$
$blockN1, blockN2, blockN3)$



6.2.3 Performance Analysis




Speedup (N = 4096 | BLOCK = 256)
Speedup
Speedup
Number of Threads
1
2
3
4
5
6
7
8


Efficiency (N = 4096 | BLOCK = 256)
Efficiency
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
Efficiency
1
2
3
4
5
6
7
8
Number of Threads

Chapter 7

MatMatDist

7.1 Problem at a glance

Given matrices A, B, C, $blockN1$, $blockN2$, $blockN3$, a number of $NTROW \times NTCOL$ threads, a number of $NPROW \times NPCOL$ processes, the goal is to compute the GeMM operation $C = C + AB$ using SUMMA in a MatMatThread approach.

7.2 Algorithm

7.2.1 Parallelization

The algorithm employs a SUMMA. It is a multiprocess approach where each process is responsible for executing a single GeMM operation of a C block.

Each process cannot independently compute the local GeMM operation; instead, the missing blocks are received from a process performing a broadcast.

7.2.2 Pseudocode

Input:

- $A^{M1 \times M2}$, $B^{M2 \times M3}$, $C^{M1 \times M3}$ matrices
  $M1 = N1/NPROW, M2 = N2/mcm(NPROW, NPCOL), M3 = N3/NPCOL$
- $blockN1$, $blockN2$, $blockN3$ block dimensions of $N1$, $N2$, $N3$ (used by MatMatBlock algorithm)
- $NPROW, NPCOL$ number of processes on rows and columns of C that will be assigned to compute part of the whole GeMM operation on C

Steps:

- Construct a row communicator and a column communicator for each process
- Calculate the process that will broadcast on row communicator and on column communicator
- Perform a broadcast of **A** on row communicator

- Perform a broadcast of **B** on column communicator
- Execute MatMatThread algorithm

Algorithm 7: MatMatDist

Input : $A^{M1\times M2}$, $B^{M2\times M3}$, $C^{M1\times M3}$, $blockN1$, $blockN2$, $blockN3$, $NTROW$, $NTCOL$, $NPROW$, $NPCOL$
Output: $C^{N1\times N3}$

$getRowColComm$(`COMM_WORLD`, `COMM_ROW`, `COMM_COL`)
for $k \leftarrow 1$ **to** $mcm(NPROW, NPCOL)$ **do**
  $senderRow \leftarrow k \bmod NPCOL$
  $senderCol \leftarrow k \bmod NPROW$
  `BCAST(COMM_ROW,` A`,` $-i$`)`
  `BCAST(COMM_COL,` B`,` $-j$`)`
  $MatMatThread(\mathbf{A}^{(M1\times M2)}, \mathbf{B}^{(M2\times M3)}, \mathbf{C}^{(M1\times M3)}, blockN1, blockN2, blockN3, NTROW, NTCOL)$
end

7.2.3 Performance Analysis

Chapter 8

Further Algorithms

8.1 Parallel Sum

The project includes additional modules not part of the presented submissions, such as two parallel summation algorithms: **Ringsum** and **Cascadesum**.

The latter is more efficient with respect to time complexity, requiring only $\log_2(P)$ steps for message passing, as opposed to $P-1$, where P denotes the number of processes involved.

8.2 MatMatThread

MatMatThread is implemented with $NTROW \times NTCOL$ threads. Each thread performs a single matrix-matrix product, i.e. computes a **single block** of **C**. It gives the best $\frac{N_{mem}}{Nflop}$ ratio, $N_{mem} = \mathcal{O}(N^2)$.

Another similar idea of approaching MatMatThread has been analyzed: $NTHREADS$ threads are given, each of which performs a single matrix-matrix product, i.e. computes a **single block of rows** of **C**, $N_{mem} = \mathcal{O}(N^2)$.

Other two approaches have been analyzed, too. They exhibit $N_{mem} = \mathcal{O}(N^3)$ and are:

- $NTROW \times NTCOL$ threads are given. A thread performs more vector-vector products, i.e. computes **more sparse** elements $\mathbf{C}(i,j)$.
- $NTHREADS$ threads are given. A thread performs more vector-matrix products, i.e. computes **more sparse** rows $\mathbf{C}(i,:)$.

This difference in the complexity of N_{mem} that shows up is due to more main memory accesses to the same elements.

8.3 MatMatDist

Another analyzed way of approaching MatMatDist is the Cannon algorithm. Despite it reaches a better efficiency $\left(E = \frac{1}{1+\frac{\sqrt{P}}{N}\cdot\frac{N_{mem}}{N_{flop}}}\right)$ compared to SUMMA $\left(E = \frac{1}{1+\frac{\sqrt{P}\log_2\sqrt{P}}{N}\cdot\frac{N_{mem}}{N_{flop}}}\right)$, it can't be used for a number of processes which is not a perfect square.

Acronyms

GeMM General Matrix Multiply 21, 23, 25, 31

HPC High Performance Computing 1

IDE Integrated Development Environment 1, 3

MPI Message Passing Interface 4

PDC Parallel and Distributed Computing 1

SPMD Single Program Multiple Data 1

SUMMA Scalable Universal Matrix Multiplication Algorithm 31, 35