PARALLEL SYSTEMS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY

TASK 1

OpenMP

STUDENT / WORK DETAILS

NAME: ATHANASIOU VASILEIOS EVANGELOS REGISTRATION
NUMBER: 19390005
STUDENT SEMESTER: 11
STUDY PROGRAM: PADA

LABORATORY LEADER: IORDANAKIS MICHALIS
THEORY LEADER: MAMALIS VASILIOS

PARALLEL SYSTEMS

CONTENTS

1. Introduction

1.1 Purpose of the exercise

The purpose of the exercise is to implement and evaluate a parallel program using OpenMP , which is, at a low level, the multithreaded parallel programming standard, with the logic of the fork - join parallel execution model .

Specifically, the programmer becomes familiar with the high level techniques provided by OpenMP , where it distinguishes the part of code that will be executed in parallel and understands the division of tasks into the corresponding threads and the synchronized communication between threads

Finally, the exercise aims to highlight the execution times of parallel tasks, so that the speed - ups can be calculated and compared both in the case where the program is executed sequentially (with 1 thread) and in parallel (> 1 threads).

1.2 Brief description of the problem being solved

The problem concerns the following tasks:

a) processing a square matrix A (N x N) with the aim of checking diagonal dominance (it is also checked whether they are strictly diagonals of dominance or not), for each line to check whether the property holds

$$|A_{ii}| > \sum |A_{ij}| \text{ όπου } j = 0 \dots N-1 \text{ για } i \neq j$$

b) the calculation of the largest element of the diagonal of A in absolute value

$$m = \max(|A_{ii}|), \qquad i = 0 \dots N-1$$

c) the creation of a new matrix B also square (N x N) with elements

$$B_{ij} = m - |A_{ij}| \text{ για } i \neq j \text{ και } B_{ij} = m \text{ για } i = j$$

d) the calculation of the minimum element of B by absolute value using various methods such as the binary tree algorithm

$$\text{min\_val} = \min(|B_{ij}|), \qquad \text{για } i,j = 0 \dots N-1$$

e) performance analysis for various parameter values

Regarding performance, the following parameters are taken into account:

- Table size N
- Number of threads T
- CZ elements to be shared across threads

2. Design

2.1 Description of the approach followed

The implementation is based on distributing the computations across multiple threads using OpenMP to achieve parallel processing. The approach is divided into distinct phases:

- **Task assignment to threads:** Using OpenMP , each thread performs specific pieces of the calculations.
- **Exploiting the potential of OpenMP :** In task distribution, synchronization and reduction instructions .
- **Problem structure analysis:** Data is organized to facilitate parallel processing, minimizing communication between threads.

For each task mentioned in chapter 1.2 Brief description of the problem being solved, a separate parallel processing area is defined, in order to distinguish the sequential tasks that must be performed before and after the parallel processing of each task.

2.2 Analysis of logic and methodologies

The logic is based on the following:

a) **Checking the matrix A for strict diagonal dominance:** The rows of the matrix are divided into threads according to the parameter CZ and it is checked in parallel whether A is strictly diagonal dominance through a common variable
b) **Calculation of the maximum element of the diagonal of A:** Each thread calculates the maximum element of the diagonal locally and the results are collected in a variable (reduction instruction)
c) **Create a new array B with values Bij = m – |Aij| for i ≠ j and Bij = m for i = j:** Each thread calculates the elements of array B locally, using the logic of uniform sharing (collapse instruction)
d) **Calculation of the minimum element of B:** Each thread calculates locally the minimum element that corresponds to it from the partitions and the local results are collected to calculate the minimum element of B. The work is carried out with 3 methods, with a reduction instruction , with a critical region protection mechanism and with a binary tree algorithm.

2.3 Description of data structures and algorithms

2.3.1 Data structures and variables

The following data structures and variables were used to solve the problem:

Variable Name	Variable Description
int	
** A	Dynamic square 2D matrix of size N, where integers (positive or negative) are stored via a generator program that creates either strictly diagonal dominating matrices or not
** B	Dynamic square 2D array of size N, where the integers mentioned in chapter 1.2 are stored. Brief description of the problem solved in task c)
*M	Dynamic 1D array of size T equal to the number of threads, where it is used to implement the binary tree algorithm to find the minimum value of B
i , j, k	Repetition indicators for shares
rowSum	Sum of the elements of a row of table A
chunk	Number of iterations per thread
flag	Variable "index" to check if A is strictly diagonally dominated
time	Thread ID
loc_sum	Local variable for each thread that calculates the sum of the elements that have been shared with it
loc_flag	Local variable "index" to check for each thread whether the diagonal dominance property is valid for the elements that have been shared with it
loc_index	Local variable for each thread that stores the element belonging to the diagonal of A
loc_min	Local variable for each thread used in the binary tree algorithm and stores the smallest element of M[tid], M [tid + incr]
incr	Recursion index for the binary tree algorithm
temp0	The element M[tid]
temp1	The element M[tid + incr]
m	The largest element of the diagonal of matrix A
min_val	The minimum element of table B
double	
loc_time_start	Start of parallel processing time measurement for each task separately
loc_time_end	End of parallel processing measurement time for each task separately
all_time_start	Start of time measurement of the entire parallel program
all_time_end	End of measurement time of the entire parallel program

FILE*	
fpA	Output file for saving table A
fpB	Output file for saving table B

2.3.2 Generator program for producing 2D arrays

This algorithm creates the square matrix A of dimensions N x N based on 2 choices: the choice to be strictly diagonally dominated or not. The choice is made randomly with the following command, which produces the value 1 to be strictly diagonally dominated or the value 0 to be not:

```
rand ( ) % 2
```

- **Strict diagonals of dominants (value 1):**
  - Code:

```
for ( i = 0 ; I < N ? i ++)
{
      rowSum = 0 ;
      for ( j = 0 ; j < N ? j ++ )
      {
            if ( i == j )
            {
                  Array [ i ][ i ] = rand () % 21 - 10 ;
                  Array [ i ][ i ] = Array [ i ][ i ] >= 0 ? Array [ i ][ i ]
+ 20 : Array [ i ][ i ] - 20 ;
            }
            else
            {
                  Array [ i ][ j ] = rand () % 21 - 10 ;
                  rowSum += abs ( Array [ i ][ j ]);
            }
      }
            if ( rowSum >= abs ( Array [ i ][ i ]))
            {
                Array [ i ][ i ] = rowSum + rand () % 5 + 1 ;
                Array [ i ][ i ] *= ( rand () % 2 == 0 ) ? 1 : - 1 ;
            }
        }
```

  - Operation:
    - With 2 loops we access the 2D array
    - We check if we are on the main diagonal
    - We choose values from -10 to 10

- We change the value by 20 depending on the sign, in order to give larger values to the main diagonal so that the diagonal dominance property applies.
- If we are not on the main diagonal, then we choose values again in the range -10 to 10, without changing the value, so as not to exceed the elements of the main diagonal.
- We add for each row the elements of the non-main diagonal to the variable *rowSum*
- We check if the property is true

$$|A_{ii}| > \sum |A_{ij}| \text{ όπου } j = 0 \dots N-1 \text{ για } i \neq j$$

- If not, then we adjust the elements of the main diagonal
- We add the sum of the remaining elements of the line with values from 1 to 5 and store the result in the element of the main diagonal
- We choose a random sign

- **Non-strict diagonals of dominants (value 0):**
  - Code

```
for ( i = 0 ; I < N ? i ++)
{
      rowSum = 0 ;
      for ( j = 0 ; j < N ? j ++ )
      {
            if ( i == j )
            {
                  Array [ i ][ i ] = rand () % 11 - 5 ;
            }
            else
            {
                   Array [ i ][ j ] = rand () % 21 - 10 ;
                   rowSum += abs ( Array [ i ][ j ]);
               }
            }
            if ( abs ( Array [ i ][ i ]) > rowSum )
            {
                Array [ i ][ i ] = rowSum - rand () % 5 - 1 ;
                Array [ i ][ i ] *= ( rand () % 2 == 0 ) ? 1 : - 1 ;
            }
        }
```

  - Function :
    - With 2 loops we access the 2D array
    - We check if we are on the main diagonal
    - We choose values from -5 to 5

- If we are not on the main diagonal, then we choose values in the range -10 to 10, so that they exceed the elements of the main diagonal.
- We add for each row the elements of the non-main diagonal to the variable *rowSum*
- We check if the property is true

$$|A_{ii}| > \sum |A_{ij}| \text{ όπου } j = 0 \dots N-1 \text{ για } i \neq j$$

- If true, then we adjust the elements of the main diagonal
- We subtract the sum of the remaining elements of the line with values from -1 to 3 and store the result in the element of the main diagonal
- We choose a random sign

2.3.3 Binary tree algorithm

The binary tree algorithm is a fast technique that replaces the reduce instruction. and synchronizes threads with phases to perform parallel calculations, such as sum, maximum element, etc. The idea of the algorithm is to successively reduce the number of threads acting, comparing their data pairwise in each phase until the desired result remains. Usually, the data is stored in a 1D array of size equal to the number of threads and the result is stored in the 1st position of the array. Its operation as well as its implementation in OpenMP is referred to in chapter 3.2 Explanation of parallel parts of the code .

3. Implementation

3.1 Reference to the basic functions of the code

The main functions of the code include:

- **Create an array (create2DArray):**
  The function creates an N x matrix N with random values. It depends on the result of the generator program mentioned in chapter 2.3.2 Generator program for producing 2D matrices whether the matrix will be:
  - Strictly dominant diagonals: The absolute value of each diagonal element is greater than the sum of the absolute values of the remaining elements of the corresponding row.
  - Non-strict diagonals of dominants: The above property does not apply.
- **Print array (print2DArray):**

The function prints an N x matrix N to an output file. Each row of the table is recorded on a line of the file with the elements separated by spaces.

- **Checking for strictly diagonal dominance (Task a):**
  A parallel check is performed to see if the matrix A is strictly diagonally dominated. For each row, the sum of the remaining elements is compared with the diagonal element. If any row violates the property, the program terminates.
- **Calculating maximum diagonal value (Task (b):**
  The maximum value is found $m = \max(|A_{ii}|)$, with parallel processing. The calculation is done using OpenMP and the reduction directive .
- **Create a new table B (Task c):**
  Table B is calculated in parallel as follows:
  - Αν $i = j$ τότε $B_{ij} = m$
  - Αν $i \neq j$ τότε $B_{ij} = m - |A_{ij}|$
- **Calculating the minimum value of array B (Task d):**
  The minimum element in table B is calculated using various techniques:
  - **(d1)** Using the OpenMP reduction directive
  - **(d2.1)** Without the use of the reduction directive, with a critical area protection mechanism
  - **(d2.2)** Without the use of the reduction directive, using a binary tree algorithm, where the comparison and minima calculation are done gradually in phases.
- **Time measurements:**
  In each task, the following is recorded:
  - The start and end time of each task
  - The total execution time of all parallel calculations

3.2 Explanation of parallel parts of the code

As mentioned in chapter 2.1 Description of the approach followed , a separate parallel processing area was defined for each task, in order to distinguish the sequential tasks that must be performed before and after parallel processing. The parallel sections are explained below:

- **Checking for strictly diagonal dominance (Task a):**
  - Code:

```
#pragma omp parallel shared ( flag ) private ( i , j , loc_sum , loc_flag ,
loc_index )
{
        loc_flag = 1 ;
        #pragma omp for schedule ( static , chunk )
        for ( i = 0 ; I < N ? i ++)
        {
            loc_sum = 0 ;
```

```
            for ( j = 0 ; j < N ? j ++ )
                if ( i != j )
                    loc_sum += abs ( A [ i ][ j ]);
                else
                    loc_index = abs ( A [ i ][ i ]);

            if ( loc_index <= loc_sum )
                loc_flag = 0 ;
        }

        #pragma omp atomic
        flag *= loc_flag ;
}
```

- Operation:
  - The **#pragma omp parallel** activates parallel threads
  - The **#pragma omp for** distributes the iterations of the outer **for loop** across the threads based on the CZ parameter we defined
  - The local variable *loc_flag* is used to avoid accessing the common variable *flag* , where it takes the value 0 if the diagonal dominance property does not hold for the elements of a *thread* . This is then multiplied by the common variable flag and specifies that it will always have the value 0 indicating that the array is not strictly diagonally dominated.
  - #pragma **omp atomic** is a critical area protection mechanism and ensures safe access to the shared *flag variable*

- **Calculating maximum diagonal value (Task (b):**
  - Code:

```
#pragma omp parallel default ( shared ) private ( i )
{
        #pragma omp for schedule ( static , chunk ) reduction ( max : m )
        for ( i = 0 ; I < N ? i ++)
            if ( A [ i ][ i ] > m )
                m = A [ i ][ i ];
}
```

  - Operation:
    - The **#pragma omp for** distributes CZ iterations across threads to calculate the maximum value.
    - **reduction (max : m)** directive ensures that each thread computes a local maximum and the results are finally collected in the variable *m*

- **Create a new table B (Task c):**

- Code:

```
#pragma omp parallel default ( shared ) private ( i , j )
{
        #pragma omp for schedule ( static , chunk ) collapse ( 2 )
        for ( i = 0 ; I < N ? i ++)
            for ( j = 0 ; j < N ? j ++ )
                if ( i == j )
                    B [ i ][ j ] = m ;
                else
                    B [ i ][ j ] = m - A [ i ][ j ];
}
```

- Operation:
  - **collapse (2)** directive merges the two **for loops** in one, allowing its parallel execution. Equivalently, the loop is executed:

```
for ( i = 0 ; I < N * N ; i ++) { ... }
```

  - The values of table B are calculated independently, so no synchronization mechanisms are required.

- **Calculate minimum value (Task d):**
  The minimum element of array B is calculated using three different methods.

  - With reduction instruction (Task d1):
    - Code

```
#pragma omp parallel default ( shared ) private ( i , j )
{
        #pragma omp for schedule ( static , chunk ) reduction ( min :
min_val )
        for ( i = 0 ; I < N ? i ++)
            for ( j = 0 ; j < N ? j ++ )
                if ( B [ i ][ j ] < min_val )
                    min_val = B [ i ][ j ];
}
```

    - Operation
      - The instruction **reduction (min : min _ val)** ensures that each thread finds the local minimum and the results are collected in the variable *min _ val*

  - With critical area protection (Task d2.1):
    - Code

```
#pragma omp parallel shared ( min_val ) private ( i , j )
{

        #pragma omp for schedule ( static , chunk )
        for ( i = 0 ; I < N ? i ++)
            for ( j = 0 ; j < N ? j ++ )
                if ( B [ i ][ j ] < min_val )
                {
                    #pragma omp critical ( inc_min_val )
                    {
                        min_val = B [ i ][ j ];
                    }
                }
}
```

- Operation
  - **The #pragma** directive **omp critical** protects the shared variable *min _ val* from simultaneous modification that would lead to incorrect results
  - Correct operation is ensured but at a performance cost due to serial access to the critical area

- With binary tree algorithm (Task d2.2):
  - Code

```
#pragma omp parallel default ( shared ) private ( tid , i , j , incr ,
temp0 , temp1 , loc_min )
{
        time = omp_get_thread_num ();
        loc_min = 1000000 ;

        #pragma omp for schedule ( static , chunk )
        for ( i = 0 ; I < N ? i ++)
            for ( j = 0 ; j < N ? j ++ )
                if ( B [ i ][ j ] < loc_min )
                    loc_min = B [ i ][ j ];

        M [ time ] = loc_min ;

#pragma omp barrier

        incr = 1 ;

        while ( incr < T )
        {
            if ( time % ( 2 * incr ) == 0 && time + incr < T )
            {
                temp0 = M [ time ];
```

```
                temp1 = M [ time + incr ];
                loc_min = ( temp0 <= temp1 ) ? temp0 : temp1 ;
                M [ time ] = loc_min ;
            }

#pragma omp barrier

            incr = 2 * incr ;
        }
}
```

- Operation
  - Each thread computes the local minimum element of array B, limiting its search to the elements allocated to it thanks to the use of the **# pragma directive. omp for** .
  - Stores the local minimum in the array *M* at location *M[tid]* , where *tid the* thread ID .
  - The algorithm is executed in phases, where threads compare pairs of elements of the array *M,* the smallest element is kept at position *M [tid]* , and the active threads are reduced in each phase.
  - In the 1st $^{\text{phase}}$ (*incr = 1*) the threads with *tid = 0, 2, 4, ...* compare the values *M [tid]* and *M [tid +1]*
  - In the 2nd $^{\text{phase}}$ (*incr = 2*) the threads with *tid = 0, 4, 8, ...* compare the values *M [tid]* and *M [time +2]* .
  - The process continues with the iteration index *incr* doubling in each phase.
  - The condition *tid % (2 * incr) == 0* ensures that the correct thread is responsible for this phase.
  - The condition *tid + incr < T* ensures that access outside the bounds of the array *M is avoided.*
  - **The #pragma** directive **omp barrier** ensures that all threads complete their phase before moving on to the next
  - The minimum element of array *B* is located at position *M[0].*

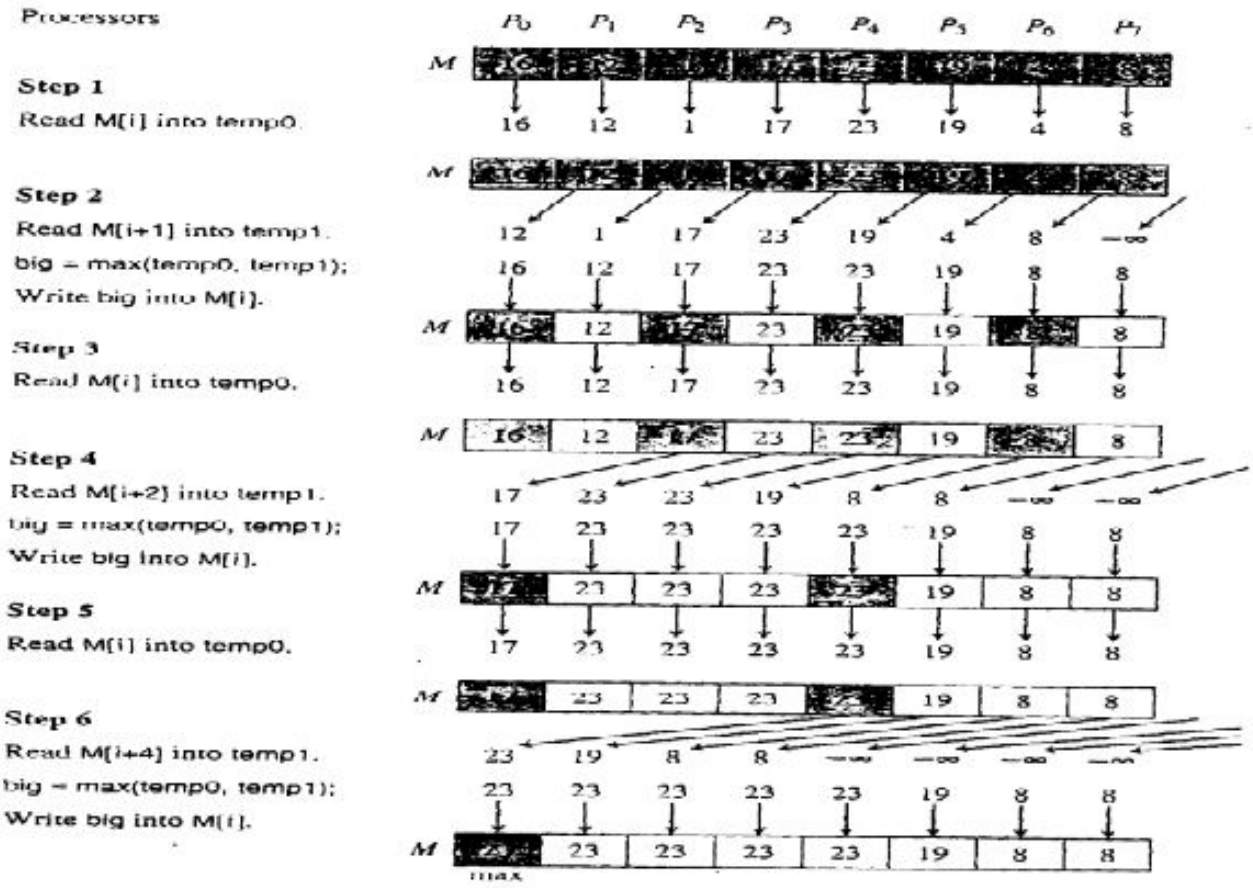

Figure 1. Max calculation in the PRAM – EREW model

Source: Lesson 4 ¯ Parallel Computing Standards PRAM – Introduction to Parallel Computing – pp. 22-23

3.3 Description of communication and synchronization between threads

Communication and synchronization between threads is ensured by various mechanisms provided by OpenMP . The methods that are described below were used in the program:

- **Shared and Private Memory:**
  - Variables :
    They are used by all threads simultaneously, *e.g. flag , min _ val , B* etc.
  - Private Variables :
    Each thread has its own copy of the *variable , e.g. loc_sum , loc_flag , tid ,* etc.
  - Use in the program:

```
#pragma omp parallel shared ( flag ) private ( i , j , loc_sum , loc_flag , loc_index )
```

  - Shared variables are used to collect results from threads

- Private variables ensure that each thread performs independent computational operations

- **Synchronization Mechanisms**
  The correct operation of the program relies on synchronization to avoid conflicts:
  - Atomic Instruction :
    - Used for mutual exclusion when it is to ensure the individuality of simple update commands of a common variable
    - In the program it is used to update the common variable *flag* when checking whether the matrix *A* is strictly diagonally dominated (Task a):

```
#pragma omp atomic
flag *= loc_flag ;
```

  - Critical Area (Critical Directive):
    - Used when multiple threads need to have exclusive access to a variable
    - In the program it is applied to update *min _ val* when calculating the minimum value in table *B* (Task d2.1):

```
#pragma omp critical ( inc_min_val )
{
      min _ value = B [ i ][ j ];
}
```

  - Barrier Mechanism :
    - Defines a synchronization point where all threads must reach before proceeding.
    - In the program, the binary tree algorithm is used to calculate the minimum element of *B* (Task d2.2):

```
#pragma omp barrier
```

  - Instruction :
    - Used to calculate a summary result (e.g. sum, maximum, minimum, etc.) by combining the values calculated by the individual threads
    - In the program it is used to calculate the maximum element of the diagonal of *A* (Task (b):

```
#pragma omp for schedule ( static , chunk ) reduction ( max : m )
```

    - and the minimum element of *B* (Task d1):

```
#pragma omp for schedule ( static , chunk ) reduction ( min : min_val )
```

- **Task Sharing**

Sharing loop iterations across threads is ensured through the **# pragma directive omp for** . In the program, the **schedule(static, chunk) directive is used** , where:

- Static : Repetitions are statically distributed across threads in order of priority according to their ID.
- Chunk : The number of iterations each thread will execute
- The program uses in each task:

```
#pragma omp for schedule ( static , chunk )
```

4. Tests and Results

4.1 Reporting of execution conditions

The execution is done for different array sizes N, thread numbers T and allocation CZ of iterations per thread.

The program is compiled via command line in a Linux environment , with the compiler GNU **gcc and** the **–fopenmp switch** for linking it with the **omp library . h .**

```
gcc –o omp omp.c - fopenmp
```

The program is executed via command line in a Linux environment and the user must pass 2 txt files as parameters , so that tables A and B are saved respectively. Indicative execution command:

```
./omp A . txt B.txt
```

4.2 Presentation of results in text format

*The results are stored in the Output folder and tables A and B in their respective folders. To save space, not all results are presented in text format in this documentation. To be redirected to the output files, click on the link in the sub- headings below and respectively for the tables whose name is found both in the corresponding folders and in the text results. Due to its large size, the **10000 x 10000** dimension tables were **not chosen to be included.***

The program requires the user to pass 2 . txt output files with a name of his choice, in which the tables A and B will be stored. In case the user does not enter the required number of parameters, the program terminates and a characteristic message is displayed.

4.2.1 Output_no_args.txt

```
Usage : . / omp A.txt B.txt
```

If the board is not strictly diagonally dominated, then the program terminates prematurely.

4.2.2 Output_no_strict_diagonal.txt

```
Threads : 1
Matrix size: 10 x 10
Chunk size: 2
================================== [Task a.] ==============================
Is A strictly diagonal dominant?
NO
The array has been stored in file A/ A_no_strict_diagonal.txt

----------------------------------------
Task a. finished in 0.000005 sec.
----------------------------------------
================================================================================
========

----------------------------------------
Parallel program finished in 0.000005 sec.
----------------------------------------
```

If the board is strictly diagonally dominated, then the remaining parallel operations are performed.

4.2.3 Output_T1_N10_CZ2.txt

```
Threads : 1
Matrix size: 10 x 10
Chunk size: 2
================================== [Task a.] ==============================
```

```
Is A strictly diagonal dominant?
YES
The array has been stored in file A/ A_T1_N10_CZ2.txt
----------------------------------------
Task a. finished in 0.000007 sec.
----------------------------------------
================================================================================
========
================================== [Task b.]
======================================
m = max( | Aii |) =>
m = 61
----------------------------------------
Task b. finished in 0.000001 sec.
----------------------------------------
================================================================================
========
================================== [Task c.] ==============================
Bij = m - | Ai | for i <> j and Bij = m for i = j
The array has been stored in file B/ B_T1_N10_CZ2.txt
----------------------------------------
Task c. finished in 0.000001 sec.
----------------------------------------
================================================================================
========
================================== [Task d1.] ==============================
With reduction
m = min( | By |) =>
m = 51
----------------------------------------
Task d1. finished in 0.000003 sec.
----------------------------------------
================================================================================
========
================================== [Task d2.1] ==============================
With critical section
m = min( | By |) =>
m = 51
----------------------------------------
Task d2.1 finished in 0.000001 sec.
----------------------------------------
================================================================================
========
================================== [Task d2.2] ==============================
Binary Tree Algorithm
m = min( | By |) =>
m = 51
----------------------------------------
```

```
Task d2.2 finished in 0.000001 sec.
----------------------------------------
================================================================================
========

----------------------------------------
Parallel program finished in 0.000015 sec.
----------------------------------------
```

4.2.4 Output_T2_N16_CZ3.txt

```
Threads : 2
Matrix size: 16 x 16
Chunk size: 3
================================== [Task a.] ==============================
Is A strictly diagonal dominant?
YES
The array has been stored in file A/ A_T2_N16_CZ3.txt
----------------------------------------
Task a. finished in 0.000081 sec.
----------------------------------------
================================================================================
========
================================== [Task b.]
========================================
m = max( | Aii |) =>
m = 102
----------------------------------------
Task b. finished in 0.000002 sec.
----------------------------------------
================================================================================
========
================================== [Task c.] ==============================
Bij = m - | Ai | for i <> j and Bij = m for i = j
The array has been stored in file B/ B_T2_N16_CZ3.txt
----------------------------------------
Task c. finished in 0.000002 sec.
----------------------------------------
================================================================================
========
================================== [Task d1.] ==============================
With reduction
m = min( | By |) =>
m = 92
----------------------------------------
Task d1. finished in 0.000002 sec.
----------------------------------------
================================================================================
========
```

```
================================== [Task d2.1] ==============================
With critical section
m = min( | By |) =>
m = 92
-----------------------------------------
Task d2.1 finished in 0.000002 sec.
-----------------------------------------
================================================================================
========
================================== [Task d2.2] ==============================
Binary Tree Algorithm
m = min( | By |) =>
m = 92
-----------------------------------------
Task d2.2 finished in 0.000002 sec.
-----------------------------------------
================================================================================
========


-----------------------------------------
Parallel program finished in 0.000091 sec.
-----------------------------------------
```

4.2.5 Output_T4_N25_CZ5.txt

```
Threads : 4
Matrix size: 25 x 25
Chunk size: 5
================================== [Task a.] ================================
Is A strictly diagonal dominant?
YES
The array has been stored in file A/ A_T4_N25_CZ5.txt
-----------------------------------------
Task a. finished in 0.000112 sec.
-----------------------------------------
================================================================================
========
================================== [Task b.]
=======================================
m = max( | Aii |) =>
m = 175
-----------------------------------------
Task b. finished in 0.000002 sec.
-----------------------------------------
================================================================================
========
================================== [Task c.] ================================
Bij = m - | Ai | for i <> j and Bij = m for i = j
The array has been stored in file B/ B_T4_N25_CZ5.txt
```

```
----------------------------------------
Task c. finished in 0.000003 sec.
----------------------------------------
================================================================================
========
================================== [Task d1.] ==============================
With reduction
m = min( | By |) =>
m = 165
----------------------------------------
Task d1. finished in 0.000004 sec.
----------------------------------------
================================================================================
========
================================== [Task d2.1] =============================
With critical section
m = min( | By |) =>
m = 165
----------------------------------------
Task d2.1 finished in 0.000004 sec.
----------------------------------------
================================================================================
========
================================== [Task d2.2] =============================
Binary Tree Algorithm
m = min( | By |) =>
m = 165
----------------------------------------
Task d2.2 finished in 0.000004 sec.
----------------------------------------
================================================================================
========


----------------------------------------
Parallel program finished in 0.000128 sec.
----------------------------------------
```

4.2.6 Output_T8_N400_CZ20.txt

```
Threads : 8
Matrix size: 400 x 400
Chunk size: 20
================================== [Task a.] ===============================
Is A strictly diagonal dominant?
YES
The array has been stored in file A/ A_T8_N400_CZ20.txt
----------------------------------------
Task a. finished in 0.001359 sec.
----------------------------------------
```

```
================================================================================
========
================================== [Task b.]
=======================================
m = max( | Aii |) =>
m = 2300
----------------------------------------
Task b. finished in 0.000556 sec.
----------------------------------------
================================================================================
========
================================= [Task c.] ==============================
Bij = m - | Ai | for i <> j and Bij = m for i = j
The array has been stored in file B/ B_T8_N400_CZ20.txt
----------------------------------------
Task c. finished in 0.000718 sec.
----------------------------------------
================================================================================
========
================================= [Task d1.] ==============================
With reduction
m = min( | By |) =>
m = 2290
----------------------------------------
Task d1. finished in 0.000331 sec.
----------------------------------------
================================================================================
========
================================= [Task d2.1] ==============================
With critical section
m = min( | By |) =>
m = 2290
----------------------------------------
Task d2.1 finished in 0.000315 sec.
----------------------------------------
================================================================================
========
================================= [Task d2.2] ==============================
Binary Tree Algorithm
m = min( | By |) =>
m = 2290
----------------------------------------
Task d2.2 finished in 0.000374 sec.
----------------------------------------
================================================================================
========


----------------------------------------
```

```
Parallel program finished in 0.003653 sec.
----------------------------------------
```

4.2.7 Output_T12_N1000_CZ100.txt

```
Threads : 12
Matrix size: 1000 x 1000
Chunk size: 100
================================== [Task a.] ==============================
Is A strictly diagonal dominant?
YES
The array has been stored in file A/ A_T12_N1000_CZ100.txt
----------------------------------------
Task a. finished in 0.001395 sec.
----------------------------------------
================================================================================
========
================================== [Task b.]
========================================
m = max( | Aii |) =>
m = 5513
----------------------------------------
Task b. finished in 0.000109 sec.
----------------------------------------
================================================================================
========
================================== [Task c.] ==============================
Bij = m - | Ai | for i <> j and Bij = m for i = j
The array has been stored in file B/ B_T12_N1000_CZ100.txt
----------------------------------------
Task c. finished in 0.001915 sec.
----------------------------------------
================================================================================
========
================================== [Task d1.] =============================
With reduction
m = min( | By |) =>
m = 5503
----------------------------------------
Task d1. finished in 0.001270 sec.
----------------------------------------
================================================================================
========
================================== [Task d2.1] ============================
With critical section
m = min( | By |) =>
m = 5503
```

```
----------------------------------------
Task d2.1 finished in 0.001271 sec.
----------------------------------------
================================================================================
========
================================== [Task d2.2] ==============================
Binary Tree Algorithm
m = min( | By |) =>
m = 5503
----------------------------------------
Task d2.2 finished in 0.001415 sec.
----------------------------------------
================================================================================
========


----------------------------------------
Parallel program finished in 0.007376 sec.
----------------------------------------
```

4.2.8 Output T 8 N 10000 CZ 1000 .txt

```
Threads : 8
Matrix size: 10000 x 10000
Chunk size: 1000
================================== [Task a.] ==============================
Is A strictly diagonal dominant?
YES
The array has been stored in file A/ A_T8_N10000_CZ1000.txt
----------------------------------------
Task a. finished in 0.101806 sec.
----------------------------------------
================================================================================
========
================================== [Task b.]
========================================
m = max( | Aii |) =>
m = 53490
----------------------------------------
Task b. finished in 0.000772 sec.
----------------------------------------
================================================================================
========
================================== [Task c.] ==============================
Bij = m - | Ai | for i <> j and Bij = m for i = j
The array has been stored in file B/ B_T8_N10000_CZ1000.txt
----------------------------------------
Task c. finished in 0.162018 sec.
----------------------------------------
```

```
================================================================================
========
================================== [Task d1.] ================================
With reduction
m = min( | By |) =>
m = 53480
----------------------------------------
Task d1. finished in 0.081824 sec.
----------------------------------------
================================================================================
========
================================== [Task d2.1] ===============================
With critical section
m = min( | By |) =>
m = 53480
----------------------------------------
Task d2.1 finished in 0.075407 sec.
----------------------------------------
============================================================
=========================
================================== [Task d2.2] ===============================
Binary Tree Algorithm
m = min( | By |) =>
m = 53480
----------------------------------------
Task d2.2 finished in 0.080179 sec.
----------------------------------------
================================================================================
========


----------------------------------------
Parallel program finished in 0.502006 sec.
----------------------------------------
```

4.3 Efficiency analysis

4.3.1 Execution times of parallel tasks

The times recorded for different number of threads T and array size N are presented in the diagrams below. Specifically, the results data are:

- **Number of threads:** 1, 2, 4, 8 and 12
- **Board Size:** 10 x 10, 16 x 16, 25 x 25, 400 x 400, 1000 x 1000, 10000 x 10000

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	0.000007	0.000001	0.000001	0.000003	0.000001	0.000001	0.000015
2	0.00013	0.000002	0.000002	0.000002	0.000002	0.000002	0.000139

4	0.000123	0.000003	0.000002	0.000002	0.000003	0.000002	0.000134
8	0.000269	0.000083	0.000071	0.000082	0.000104	0.000175	0.000784
12	0.000333	0.000103	0.000079	0.000105	0.000159	0.000578	0.001358

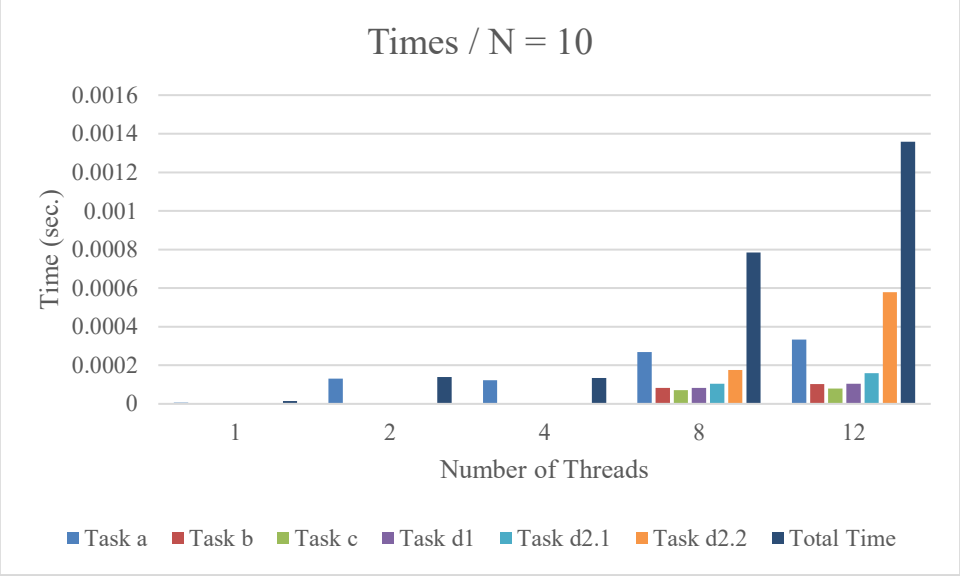

Figure 2. Parallel task execution times for a 10 x 10 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	0.000006	0.000001	0.000002	0.000002	0.000001	0.000002	0.000013
2	0.000081	0.000002	0.000002	0.000002	0.000002	0.000002	0.000091
4	0.000109	0.000002	0.000003	0.000002	0.000004	0.000002	0.000122
8	0.000268	0.000073	0.000118	0.000165	0.000094	0.000186	0.000905
12	0.0004	0.000109	0.000176	0.000187	0.000117	0.000235	0.001224

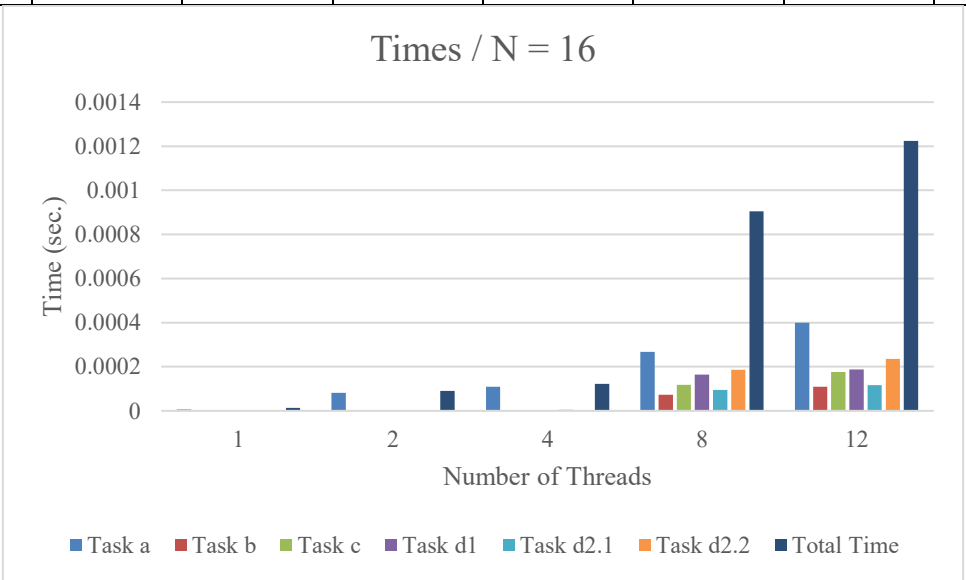

Figure 3. Parallel task execution times for a 16 x 16 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	0.000008	0.000001	0.000004	0.000005	0.000003	0.000003	0.000024

2	0.000078	0.000002	0.000003	0.000003	0.000003	0.000003	0.000091
4	0.000112	0.000002	0.000003	0.000004	0.000004	0.000004	0.000128
8	0.000252	0.000072	0.000076	0.000161	0.000112	0.00027	0.000942
12	0.000363	0.000106	0.000083	0.00008	0.000115	0.000246	0.000993

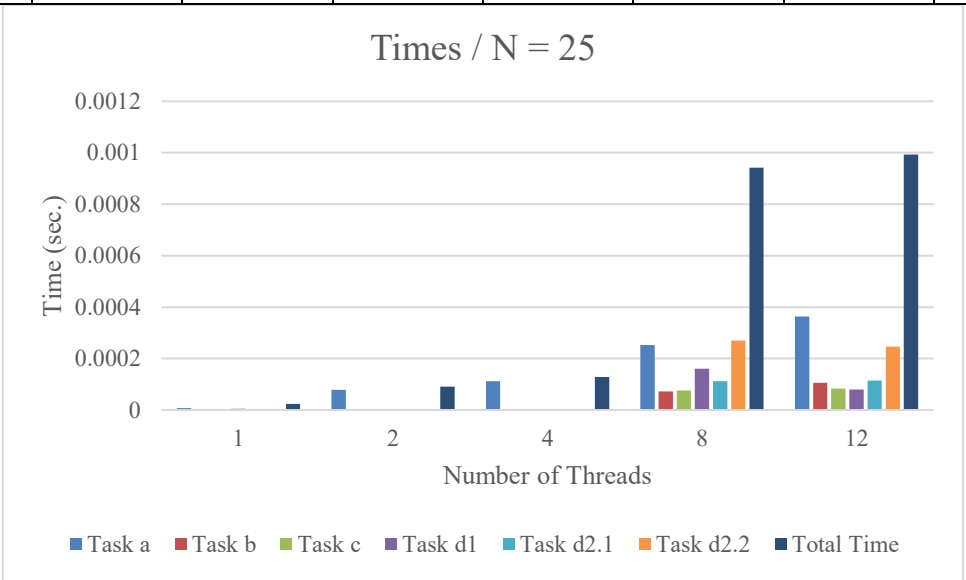

Figure 4. Parallel task execution times for a 25 x 25 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	0.00054	0.000003	0.000637	0.000602	0.000542	0.00069	0.003013
2	0.000378	0.000002	0.000335	0.000294	0.000218	0.000239	0.001466
4	0.001065	0.000004	0.000242	0.000274	0.000171	0.001126	0.002882
8	0.001359	0.000556	0.000718	0.000331	0.000315	0.000374	0.003653
12	0.000661	0.000152	0.000425	0.000372	0.000307	0.000442	0.00236

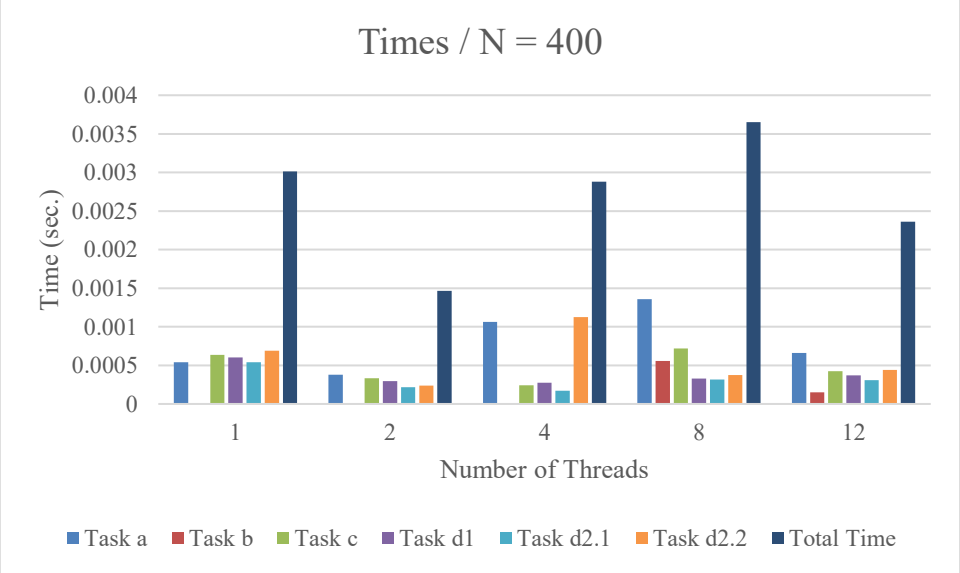

Figure 5. Parallel task execution times for a 400 x 400 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time

1	0.00311	0.000028	0.003345	0.002461	0.002426	0.00289	0.01426
2	0.00217	0.000017	0.002321	0.001914	0.001833	0.001825	0.01008
4	0.001383	0.00001	0.001525	0.001148	0.001104	0.001066	0.006236
8	0.001469	0.000089	0.001609	0.001971	0.001283	0.00144	0.007861
12	0.001395	0.000109	0.001915	0.00127	0.001271	0.001415	0.007376

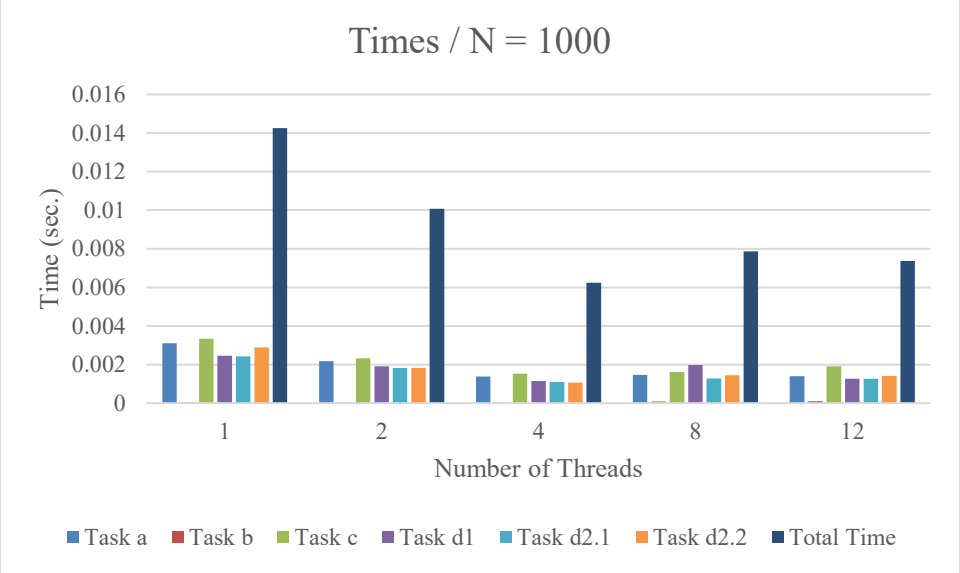

Figure 6. Parallel task execution times for a table size of 1000 x 1000

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	0.274573	0.000663	0.379998	0.305948	0.314634	0.255239	1.531054
2	0.13722	0.000315	0.200705	0.127488	0.125518	0.123352	0.714598
4	0.095803	0.000218	0.137405	0.079142	0.078211	0.08829	0.479069
8	0.101806	0.000772	0.162018	0.081824	0.075407	0.080179	0.502006
12	0.0761	0.000303	0.122951	0.068925	0.066186	0.068788	0.403253

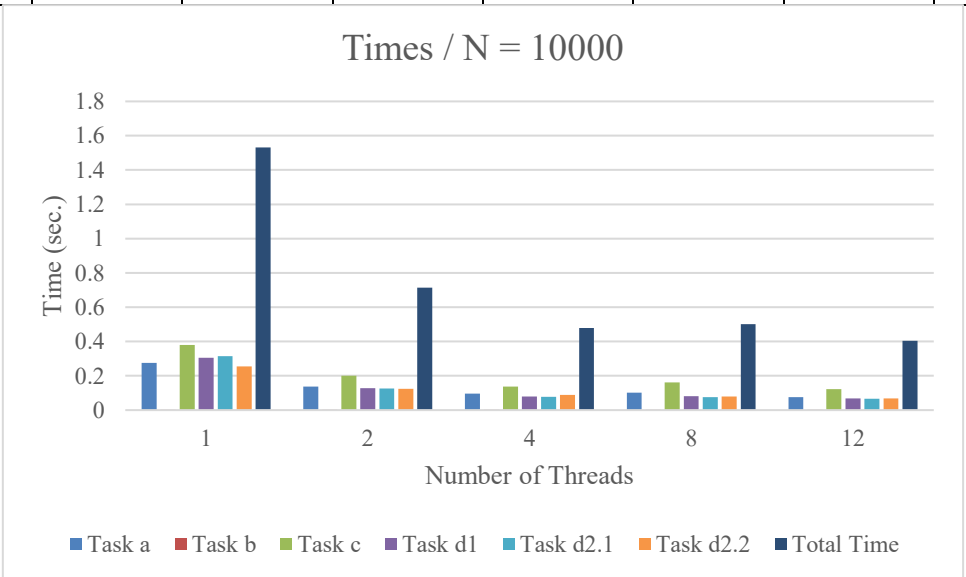

Figure 7. Parallel job execution times for a table size of 10000 x 10000

4.3.2 Accelerations

Accelerations are calculated by the following mathematical formula:

$$\text{Speedup} = {}^{t_s}/_{t_n}$$

t_s: The execution time of the sequential program, i.e., execution with a 1 thread
t_n: The execution time of the parallel program, i.e., execution with n threads

Source: Lesson 2 – Parallel Computing – Introduction to Parallel Computing – p. 7

The accelerations are presented in the diagrams below. Specifically, the results data are:

- **Number of threads:** 1, 2, 4, 8 and 12
- **Board Size:** 10 x 10, 16 x 16, 25 x 25, 400 x 400, 1000 x 1000, 10000 x 10000

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	1	1	1	1	1	1	1
2	0.0538	0.5	0.5	1.5	0.5	0.5	0.1079
4	0.0569	0.3333	0.5	1.5	0.3333	0.5	0.1119
8	0.026	0.012	0.0141	0.0366	0.0096	0.0057	0.0191
12	0.021	0.0097	0.0127	0.0286	0.0063	0.0017	0.011

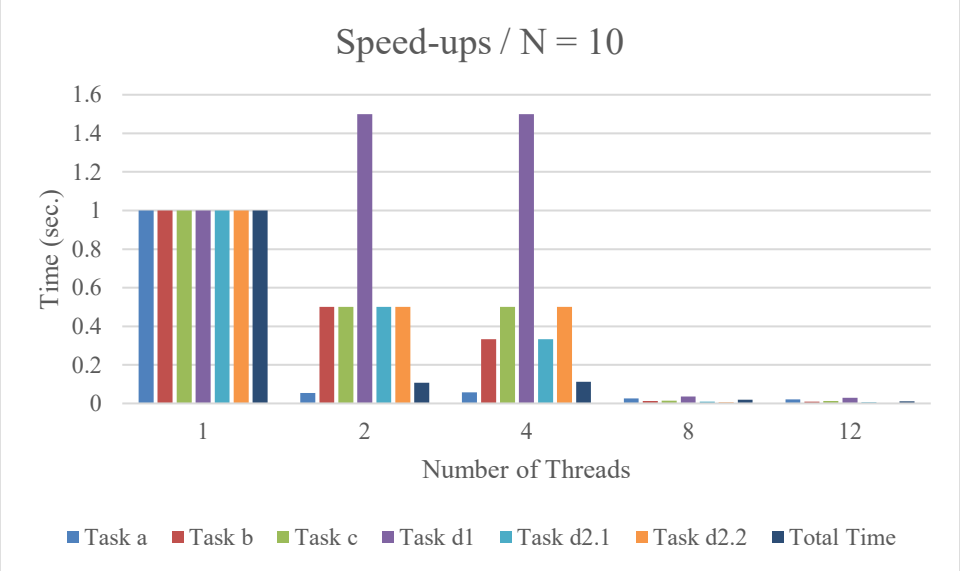

Figure 8. Speedups for 10 x 10 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	1	1	1	1	1	1	1
2	0.0741	0.5	1	1	0.5	1	0.1429

4	0.055	0.5	0.6667	1	0.25	1	0.1066
8	0.0224	0.0137	0.0169	0.0121	0.0106	0.0108	0.0144
12	0.015	0.0092	0.0114	0.0107	0.0085	0.0085	0.0106

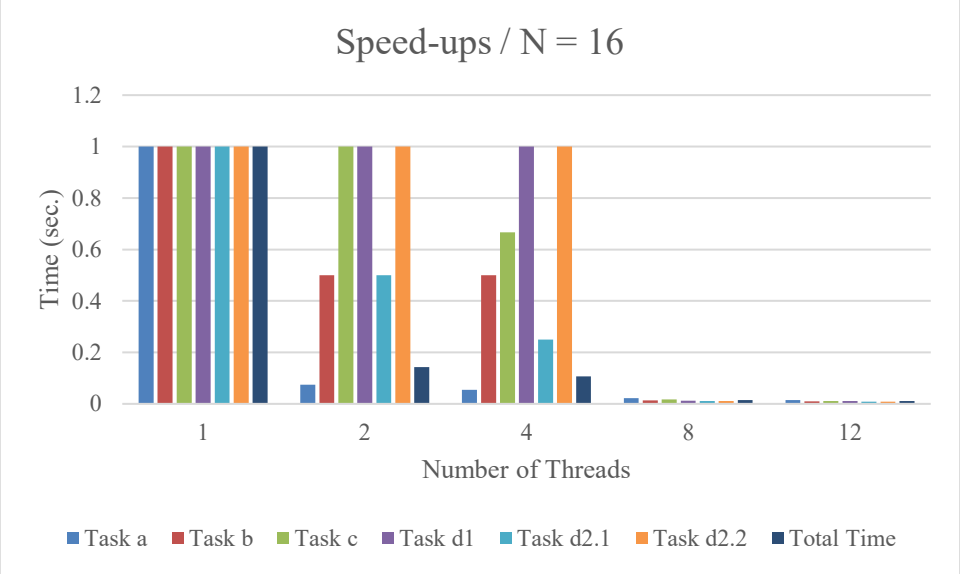

Figure 9. Speedups for 16 x 16 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	1	1	1	1	1	1	1
2	0.1026	0.5	1.3333	1.6667	1	1	0.2637
4	0.0714	0.5	1.3333	1.25	0.75	0.75	0.1875
8	0.0317	0.0139	0.0526	0.0311	0.0268	0.0111	0.0255
12	0.022	0.0094	0.0482	0.0625	0.0261	0.0122	0.0242

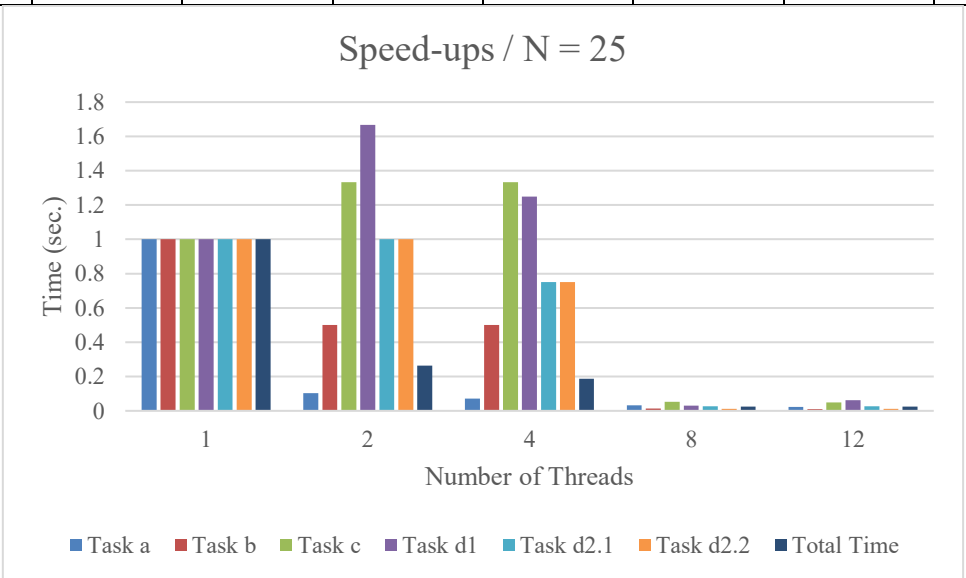

Figure 10. Accelerations for a 25 x 25 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	1	1	1	1	1	1	1

2	1.42857	1.5	1.9003	2,048	2.4876	2.8895	2.0551
4	0.507	0.75	2.6347	2,1985	3.1696	0.6134	1,046
8	0.3978	0.0054	0.887	1.8184	1.7206	1,843	0.8244
12	0.8168	0.0197	1.5	1.6161	1.7661	1.5581	1,277

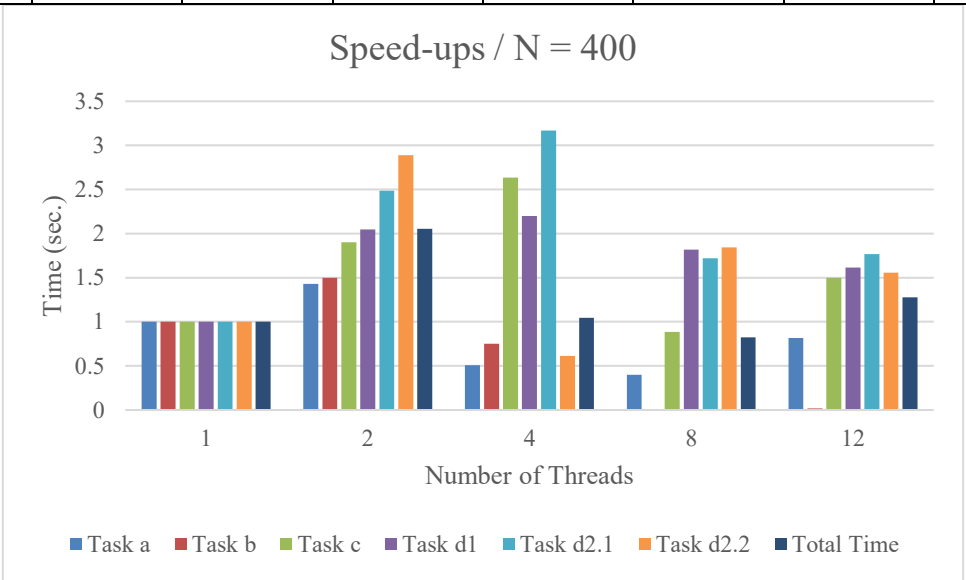

Figure 11. Speedups for a 400 x 400 board size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time
1	1	1	1	1	1	1	1
2	1.4332	1.6471	1.4415	1.2855	1.3252	1,586	1.4143
4	2.2505	2.8	2.1922	2.1455	2,1991	2.7162	2.2855
8	2.1186	0.3146	2.0789	1.2492	1.8927	2.0139	1.8141
12	2.2298	0.2578	1.7466	1.9378	1.9083	2.0432	1.9355

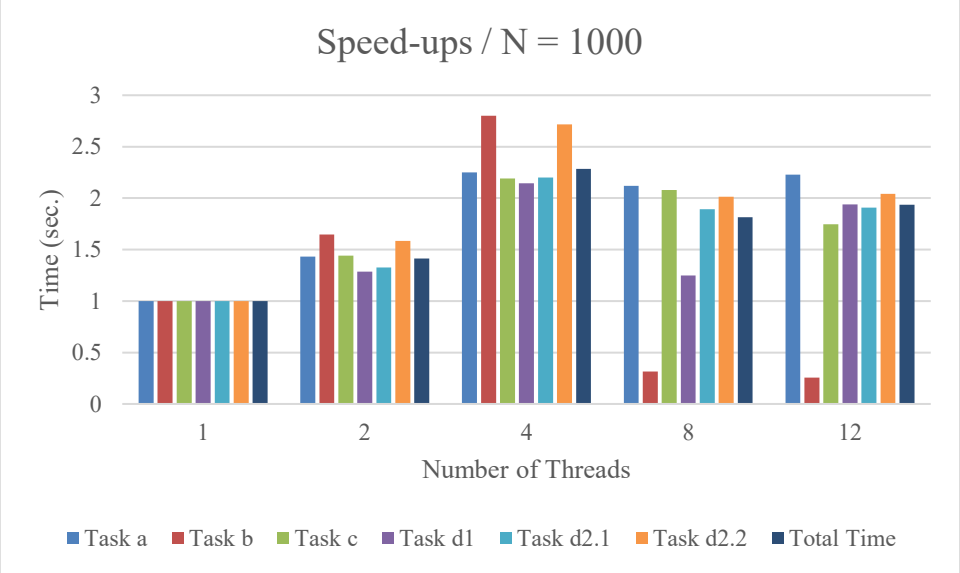

Figure 12. Speedups for a 1000 x 1000 array size

Threads	Task a	Task b	Task c	Task d1	Task d2.1	Task d2.2	Total Time

1	1	1	1	1	1	1	1
2	2.0003	2.1048	1.8942	2,399	2.5106	2.0681	2.1415
4	2.8689	3.0431	2.7642	3.8672	4.0203	2.8895	3,1943
8	2.6972	0.8581	2.3475	3.7375	4.1731	3.1817	3.0475
12	3,607	2.1898	3.0899	4.4404	4.7562	3,711	3.7967

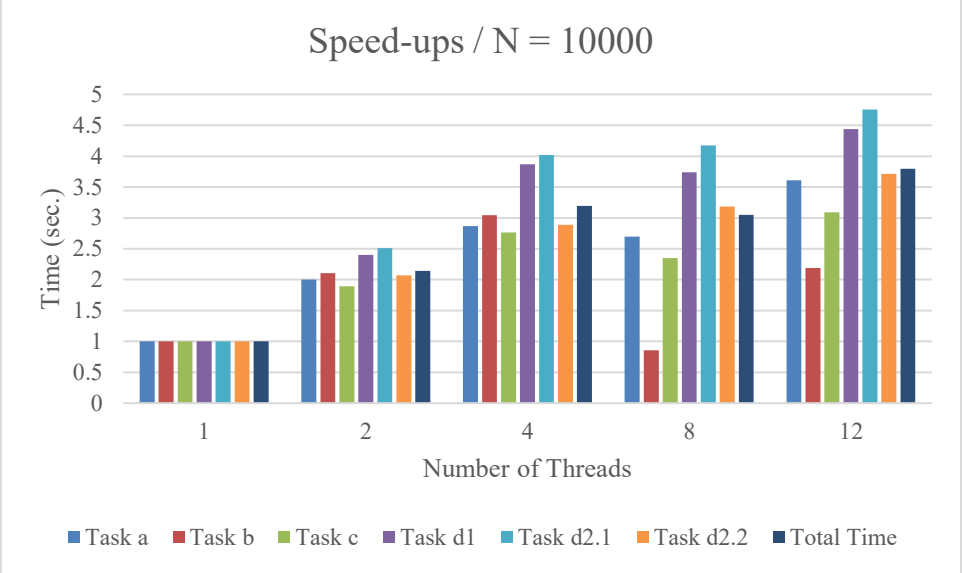

Figure 13. Speedups for a table size of 10000 x 10000

4.3.3 Observations

We observe that for small N, parallelism does not yield better times than sequential. This is also due to the fact that creating and managing more threads can cause additional overhead rather than less. This time contradicts the benefits of parallel computing, as threads compete for memory due to the small amount of data, causing delays. There are also delays in synchronization, especially when there is critical region protection, as competition for access to a common variable also causes significant delays.

For example, in the data we collected for **N = 10** , we observe that execution times increase as we increase the number of active threads and therefore we do not achieve speedups > 1 sec , as the execution time of tasks with 1 thread is shorter than the corresponding one with 2, 4, 8 or even 12 threads.

However, the benefits of parallel computing are noticeable for large N , as we observe in the data we collected for **N = 10000** , as there we achieve better times with more threads and therefore better speedups. This is due to the efficient task distribution, as the workload is distributed exclusively to the threads, minimizing idle time and ensuring that the threads contribute to the calculation.

Therefore, the benefits of parallel computing are more efficient for large volumes of data, as we do not have threads that remain inactive.

5. Problems and Solutions

5.1 Reporting problems

A problem occurred mainly during the development of the binary tree algorithm (Task d 2.2), as we did not observe the same results with the other Tasks d 1 and Task d 2.1 to calculate the minimum value of array B. Error messages for Segmentation Faults and irrational values in the M table were among the most frequent results.

also identified a problem with the execution times of parallel tasks when the working environment was WSL (Windows Subsystem for Linux), as it was taking longer to complete tasks than expected.

5.2 Solutions tested and implemented

For the binary tree algorithm, the addition of the condition was applied:

```
time % ( 2 * incr ) == 0 && time + incr < threads
```

where we ensure that the responsible threads will perform the work in this phase of the algorithm and that they will not have access outside the boundaries of the table M which was the reason for Segmentation Fault . Thus, we managed to synchronize the threads and not escape the boundaries of the table, as the main problem was that the correct threads were not participating in each phase of the algorithm and that is why we were seeing incorrect results.

For the time being, we chose to change environment and go to a pure Linux distribution (Ubuntu) and the results were clearly better.

WSL

```
Threads : 1
Matrix size: 10 x 10
Chunk size: 2
================================== [Task a.] ==============================
Is A strictly diagonal dominant?
NO
The array has been stored in file A.txt

----------------------------------------
Task a. finished in 0.000316 sec.
----------------------------------------
================================================================================
========

----------------------------------------
Parallel program finished in 0.000316 sec.
```

Linux

```
Threads : 1
Matrix size: 10 x 10
Chunk size: 2
================================== [Task a.] ==============================
```

```
Is A strictly diagonal dominant?
NO
The array has been stored in file A.txt

----------------------------------------
Task a. finished in 0.000005 sec.
----------------------------------------
================================================================================
========

----------------------------------------
Parallel program finished in 0.000005 sec.
```

After some research on the internet, we discovered the reason why parallel task execution times are longer in WSL :

OMP is incredibly slow in WSL2 due to filesystem boundary

https://github.com/JanDeDobbeleer/oh-my-posh/issues/1268

The performance issue with OpenMP in WSL 2 can often be attributed to the way WSL 2 handles the file system. WSL 2 uses a virtualized environment that interacts with the Windows file system , and this interaction can introduce significant latency when there are frequent I/O operations. This is especially noticeable with tools like OpenMP , where parallel threads may interact heavily with the file system for data storage or synchronization.

6. Conclusions

6.1 Recap

To summarize, the results showed that the parallel implementation achieves significant speedup compared to the serial execution, especially for larger array dimensions. However, limitations in scaling were observed, especially in cases where the work per thread was small or when the number of threads exceeded the available computing power of the system.

Overall, the work highlighted the importance of the correct choice of parallelism and the impact of the number of threads, load balancing and synchronization on the efficiency of the program.

PARALLEL SYSTEMS

Thank you for your attention.

PARALLEL SYSTEMS