# ECE445 Parallel and Network Computing
## Winter Semester 2022-2023
## Task 1

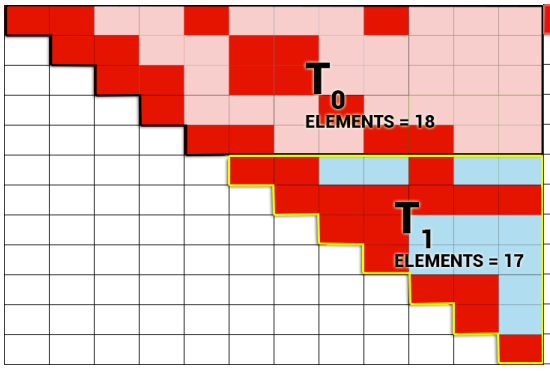
Student group:
Ioannis Roubos - 2980
Gerasimos Agoras - 2947

### Exercise 1

The data needed in both cases (2 or 4 processors) for partitioning in the obvious way are all 12 elements of vector b, since to calculate each c[i,j] element of the final vector we use the whole line i of A and the entire column vector b.

For non-obvious partitioning:

- 2 processors : We notice that the original matrix A is symmetrical with respect to its main diagonal, so we can calculate either the upper or the lower triangle and use the same data of A twice (figure 1).
- 4 processors : Using the same logic as that of the two processors, we take the upper triangular matrix from A and divide it into four blocks of the same number of non-zero cells (figure 2).




**Picture 1.**Separation of the upper triangular table in two threads

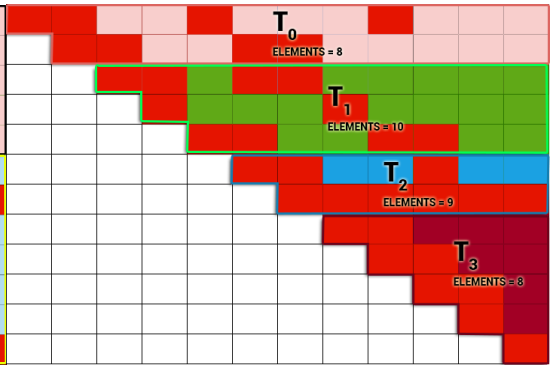



**Figure 2.**Separation of the upper triangular table in four threads

To divide the table into 2 blocks-columns:
To calculate each cell of c, each processor will temporarily store the sum of the multiplications between half the row i and half the column of b corresponding to it, and at the end of this process it will add the individual results to calculate the final value of each cell.

To divide the table into 4 blocks-columns:
Following the same logic, we will now have 4 results temporarily stored, which by adding them, we arrive at the final value of a cell of c.

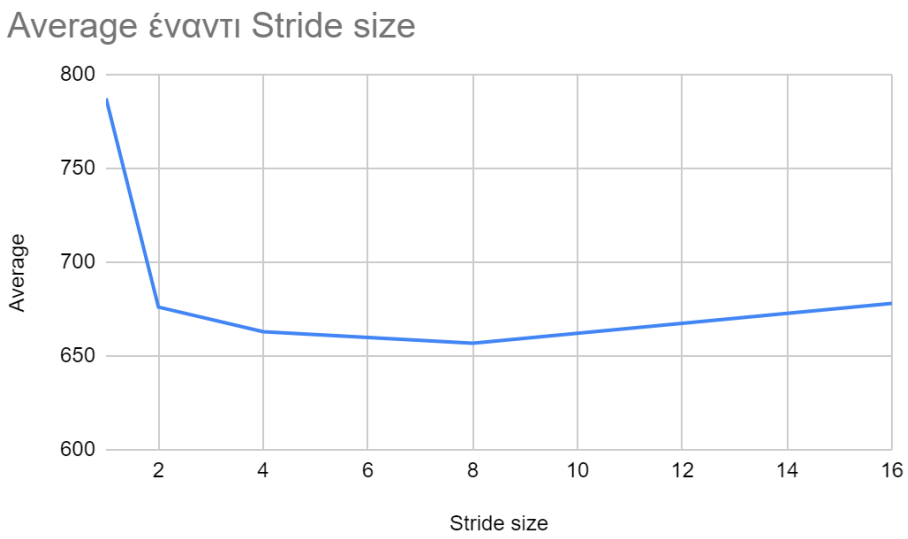
For communication requirements, when calculating individual results there is no dependency between processors, only sharing of the b array. Unique communication is found at the end, where the 4x12 results of the processors are added to get the vector c in its final form.

## Exercise 2

The number of operations in the program is n additions and n multiplications, where n is the size of the vectors.

The improvement we see in the efficiency with the increase of the step is due to the caching of the processor, as in each iteration of the loop the number of instructions related to the management of the loop and the corresponding burdens are significantly reduced.

From the results, we notice that 8 and 16 are considered critical k, as up to 8 there is an improvement in the times, but when we use 16 as a step the times get worse. This happens because of multiple memory accesses, which adds such overhead that it negates loop unrolling.

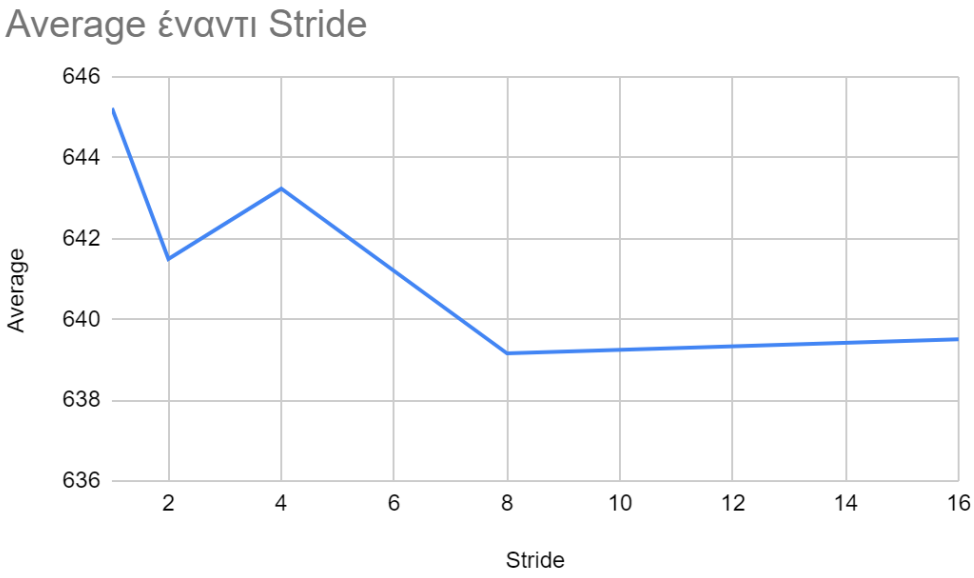



## Exercise 3

The FLOPS (Floating Point Operation Per Seconds) we have are $2n^3$.

The efficiency improvement is due to loop unrolling when calculating the inner loops.

We also notice that for stride = 4 the performance is worse compared to stride = 2, which is due to the Cache memories and cache misses that are created.



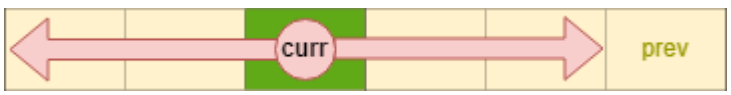

## Exercise 4

Machine Description:

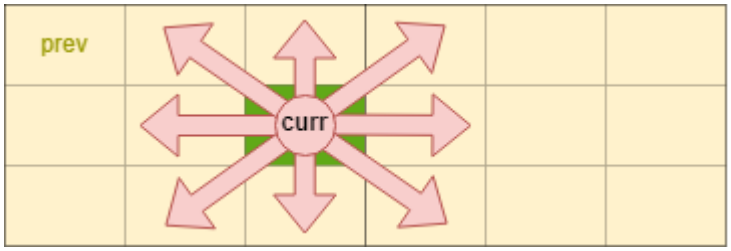
- Number of cores: 8
- Cache sizes:

| | |
|---|---|
| L1d cache: | 128KiB |
| L1i cache: | 128KiB |
| L2 cache: | 1Mb |
| L3 cache: | 6Mb |

1. Stencil 1D
   - The calculation of a cell of the curr array (10000x1) is based on the neighboring cells (2 right, 2 left) and the value of the same we got in the previous iteration of the algorithm.
   - Parallelism is applied to the calculation of each cell, without each process taking into account the location of the others, since the cells calculated in the same iteration of while are independent of each other.

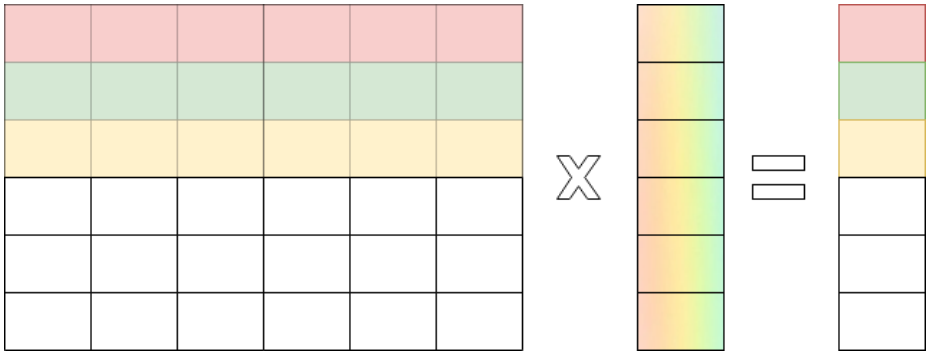



2. Stencil 2D
   - The cell calculation of the curr array (3000x3000) is based on the 3x3 square centered on the cell itself, based on the values of the array in the previous iteration of the algorithm.
   - As in the previous implementation, parallelism is applied between non-dependent cells of the same while iteration.

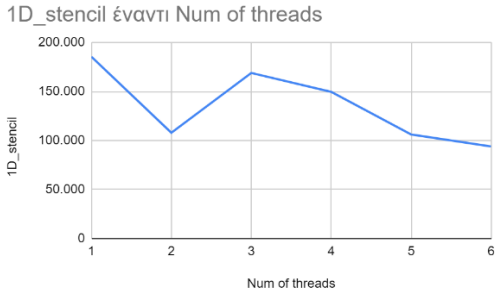


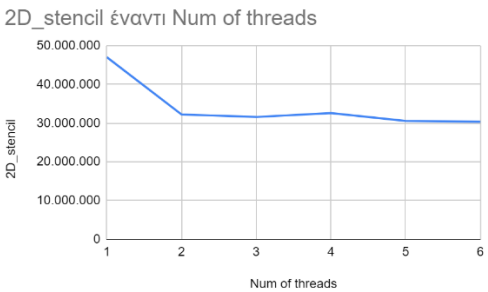

3. Matrix-vector product

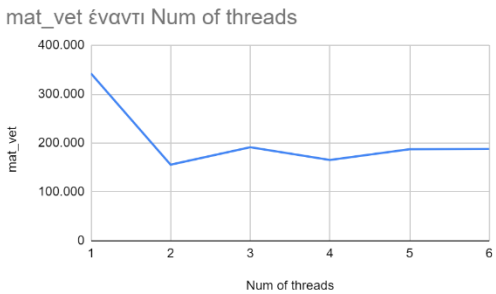
- The algorithm is the basic algorithm between array and vector, which is based on the sum of the inner products of each element of the column-vector with the cell of the corresponding row of the array.
- The parallelism in this particular case exists in the calculation of the final result-element of the above process.
  Each thread also calculates a different element of the final vector.




Below we see the measurements for all three algorithms of exercise 4.

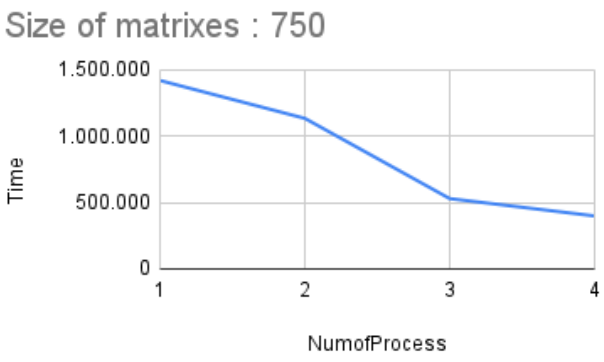


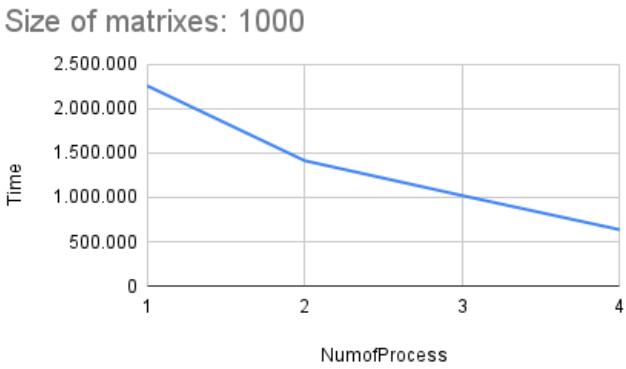



## Exercise 5

In this exercise we performed 2D matrix multiplication with a vector. According to the machine requirements, which are known from the previous exercise, we were able to run the program on up to 4 processors. We did an experiment with a table of size 1000x1000 and another with a size of 750x750. The following diagrams show the relationship between the number of processors and the calculation time. As expected, increasing the number of processors decreases the execution time.




Note that the time has been calculated in micro-seconds (μs).

All exercise measurements are also on file "Measurements_Assignment1_PDC" with the analytical measurements.

# ANNEX

## Code of Practice 4 (eg)

Here is the code for query 4. File Name “mat_vec.c”

```
/*********************************
 * Ergasia 1 – Askhsh 4
 * Ioannis Roumpos - 2980
 * Gerasimos Agoras - 2947
 *********************************/
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include "timer.h"

# define A(i,j)     A[(i)*M+j]
# define b(i)       b[i]
# define c(i)       c[i]

int main(int argc, char **argv) {

  int N = 50;
  int M = 40;

  double *A, *b, *c;

  int  size?
  int  i, j;

  /* Time */

  double time?

  if ( argc > 1 ) N = atoi(argv[1]); if
  ( argc > 2 ) M = atoi(argv[2]);

  printf("N=%d, M=%d\n", N, M);

  size = N * M * sizeof(double);
  A = (double *)malloc(size);
  size = N * sizeof(double); c =
  (double *)malloc(size); size =
  M * sizeof(double); b =
  (double *)malloc(size);
```

```
  /* Initialize */

  for ( i=0 ; i < N ; i++ ) {
    for ( j=0 ; j < M ; j++ ) {
      A(i,j) = i + j;
      b(j) = 1;
    }
  }

 /* Start Timer */
  initialize_timer ();
  start_timer();

  /* Compute */
  # pragma omp parallel for schedule(guided) private(j)
  for ( i=0 ; i < N ; i++ ) {
    c(i) = 0;
    for ( j=0; j < M ; j++ ) {
      c(i) += A(i,j) * b(j);
    }
  }

  /* stop timer */
  stop_timer();
  time=elapsed_time();

  /* print results */

  for ( i=0 ; i < N ; i+= N/8 ) {
    printf("c[%d] = %lf\n", i, c(i));
  }

  printf("elapsed time = %lf\n", time);
  return 0;
}
```

Here is the code for query 4. File Name “stencil_1D.c”

```
/*********************************
 * Ergasia 1 – Askhsh 4
 * Ioannis Roumpos - 2980
 * Gerasimos Agoras - 2947
 *********************************/
# include <stdio.h>
# include <stdlib.h>
# include "timer.h"
# include <omp.h>

# define INIT_VALUE 5000

void printResult(double *data, int size);

int main(int argc, char **argv) {

  int   N?
  int   t?
  int   MAX_ITERATION = 2000;
  double *prev, *cur;
  double error = INIT_VALUE;

  // Timer
  double time?

  // temporary variables
  int i,j;
  double *temp;

  // Check commandline args. if
  ( argc > 1 ) {
    N = atoi(argv[1]);
  } else {
    printf("Usage : %s [N]\n", argv[0]);
    exit(1);
  }
  if ( argc > 2 ) {
    MAX_ITERATION = atoi(argv[2]);
  }

  // Memory allocation for data array. prev =
  (double *) malloc( sizeof(double) * N); cur =
  (double *) malloc( sizeof(double) * N); if ( prev ==
  NULL || cur == NULL ) {
    printf("[ERROR] : Fail to allocate memory.\n");
```

```
    exit(1);
  }

  // Initialization

  for ( i=1 ; i < N-1 ; i++ ) {
      prev[i] = 0.0;
  }

    prev[0] = prev[1] = INIT_VALUE;
    prev[N-1] = prev[N-2] = INIT_VALUE;
    cur[0] = cur[1] = INIT_VALUE; cur[N-1]
    = cur[N-2] = INIT_VALUE;

  initialize_timer();
  start_timer();

  // Computation
  t = 0;


  while ( t < MAX_ITERATION) {

    // Computation
    # pragma omp parallel for
    for ( i=2 ; i < N-2 ; i++ ) {
        cur[i] = (prev[i-2]+prev[i-1]+prev[i]+prev[i+1]+prev[i+2])/5;
    }

    {
    temp = prev;
    prev = cur;
    cur = temp;
    t++;
    }
  }

  stop_timer();
  time = elapsed_time();

  printResult(prev, N);

  printf("Data size : %d , #iterations : %d , time : %lf sec\n", N, t, time);
}

void printResult(double *data, int size) {
  int i;
```

```
/* print a portion of the vector */
    printf("data[%d]: %lf \n",0, data[0]);
    printf("data[%d]: %lf \n",1, data[1]);
    printf("data[%d]: %lf \n",size/10, data[size/10]);
    printf("data[%d]: %lf \n",size/5, data[size/5]);
    printf("data[%d]: %lf \n",size/2, data[size/2]);

  return?
}
```

Here is the code for query 4. File Name “stencil_2D.c”

```
/*********************************
 * Ergasia 1 – Askhsh 4
 * Ioannis Roumpos - 2980
 * Gerasimos Agoras - 2947
 *********************************/
# include <stdio.h>
# include <stdlib.h>
# include "timer.h"
# include <math.h>
# define   INIT_VALUE        10000.0
# define   prev(i,j)        prev[(i)*N+(j)]
# define   cur(i,j)         cur[(i)*N+(j)]


void printMatrix(double *data, int size);

int main(int argc, char **argv) {

  int   N?
  int   t?
  int   MAX_ITERATION = 2000;
  double *prev, *cur;

  //double error = INIT_VALUE;

  // Timer
  double time?

  // temporary variables
  int i,j;
  double *temp;

  // Check commandline args. if
  ( argc > 1 ) {
```

```
  N = atoi(argv[1]);
} else {
  printf("Usage : %s [N]\n", argv[0]);
  exit(1);
}
if ( argc > 2 ) {
  MAX_ITERATION = atoi(argv[2]);
}

// Memory allocation for data array.
prev = (double *) malloc( sizeof(double) * N * N ); cur
= (double *) malloc( sizeof(double) * N * N ); if ( prev
== NULL || cur == NULL ) {
  printf("[ERROR] : Fail to allocate memory.\n");
  exit(1);
}

// Initialization

for ( i=2 ; i < N-2 ; i++ ) {
  for ( j=2 ; j < N-2 ; j++ ) {
    prev(i,j) = 0.0;
  }
}

for ( i=0 ; i < N ; i++ ) {
  prev(i , 0 ) = INIT_VALUE; prev(i , 1 ) = INIT_VALUE;
  prev(i , N-1) = INIT_VALUE; prev(i , N-2) = INIT_VALUE;
  prev(0 , i ) = INIT_VALUE; prev(1 , i ) = INIT_VALUE;
  prev(N-1, i ) = INIT_VALUE; prev(N-2, i ) = INIT_VALUE;

  cur( i , 0 ) = INIT_VALUE; cur( i , 1 ) = INIT_VALUE;
  cur( i , N-1) = INIT_VALUE; cur( i , N-2) = INIT_VALUE;
  cur( 0 , i ) = INIT_VALUE; cur( 1 , i ) = INIT_VALUE;
  cur( N-1, i ) = INIT_VALUE; cur( N-2, i ) = INIT_VALUE;
}

initialize_timer();
start_timer();

// Computation
t = 0;


while ( t < MAX_ITERATION) {

  // Computation
  # pragma omp parallel for private(j) shared(cur)
```

```
    for ( i=2 ; i < N-2 ; i++ ) {
      for ( j=2 ; j < N-2 ; j++ ) {
        cur(i,j) = (prev(i-2,j)+prev(i-1,j)+prev(i+1,j)+prev(i+2,j)+
                    prev(i,j)+
                          prev(i,j-2)+prev(i,j-1)+prev(i,j+1)+prev(i,j+2)
                   ) /9;
      }
    }

    temp = prev;
    prev = cur;
    cur = temp;
    t++;

  }

  stop_timer();
  time = elapsed_time();

  printf("Data : %d by %d , Iterations : %d , Time : %lf sec\n", N, N, t, time);
  printf("Final data\n");
  printMatrix(prev, N);



}

void printMatrix(double *data, int size) {
  int i,j;

/* print a portion of the matrix */

  //#pragma omp parallel for private(j)
  for ( i= 0 ; i < 5 ; i++ ) {
    for ( j=0 ; j < 5 ; j++ ) {
      printf("%lf ", data[i*size+j]);
    }
    printf("\n");
  }

  return?
}
```

## Code of Practice5 (eg)

Here is the code for question 6. File Name “ask5.c”

```
/*********************************
 * Ergasia 1 – Askhsh 5
 * Ioannis Roumpos - 2980
 * Gerasimos Agoras - 2947
 *********************************/
# include <stdio.h>
# include <stdlib.h>
# include <mpi.h>
# include <time.h>

# define N 100000

int main(int argc,char*argv[]){
   /*int N;
   N = atoi(argv[1]);
   if(argc > 2 || argc == 0){
      printf("./ask5 [N]\n");
      return -1;
   }*/

   int rank,size;
   int A[N],B[N][N],C[N],B_temp[N][N]; int
   sum[N];
   struct timespec t_start = {0,0}, t_stop = {0,0};

   //Start mpi program
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   int root = 0;
   int thread_workload = N/size; /
   *Initialise matrix-vector*/ if
   (rank == root) {
      for(int i=0; i<N; i++){
         for(int j=0; j<N; j++){
            B[i][j] = rand()%100;
         }
      }

      for(int i=0; i<N; i++){
```

```
            C[i] = rand()%100;
        }
    }

    /*Scatter matrix B row-wise*/
    MPI_Scatter(B, thread_workload, MPI_INT, B_temp, thread_workload, MPI_INT, 0,
    MPI_COMM_WORLD);

    /*Broadcast vector C */
    MPI_Bcast(C, N, MPI_INT, root, MPI_COMM_WORLD);

    /*Calculate inner product and save in sum*/
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++)
            sum[i] += C[j]*B_temp[i][j];
    }
    clock_gettime(CLOCK_MONOTONIC, &t_stop);

    /*Gather the results*/
    MPI_Gather(&sum, thread_workload, MPI_INT, A, thread_workload, MPI_INT, 0,
    MPI_COMM_WORLD);


/*printf("\n");
    if(rank == 0) {
        for(int i=0; i<N; i++){
            printf("A[%d]=%d\n", i,A[i]);
        }
    }*/

printf("Computation time is: %.6f\n",
          (((double)t_stop.tv_sec + 1.0e-9*t_stop.tv_nsec) -
          ((double)t_start.tv_sec + 1.0e-9*t_start.tv_nsec))*1000);


    MPI_Finalize();
}
```

.

.