ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ ΠΡΟΠΑΡΑΣΚΕΥΑΣΤΙΚΗΣ ΑΣΚΗΣΗΣ

Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 21.10.2025

- **Ενότητα 1.4.3 – Το Πρόγραμμα**

Δοθέντος του αρχικού προγράμματος του Game of Life με τη σειριακή υλοποίηση και έχοντας μελετήσει τα παραδείγματα των εισαγωγικών διαφανειών του εργαστηρίου, συντάξαμε το ακόλουθο παράλληλο πρόγραμμα, με χρήση του OpenMP:

a1/life_par.c

```c
/*****************************************************
 ************* Conway's game of life ******************
 *****************************************************

    Usage: ./exec ArraySize TimeSteps

    Compile with -DOUTPUT to print output in output.gif
    (You will need ImageMagick for that - Install with
    sudo apt-get install imagemagick)
    WARNING: Do not print output for large array sizes!
    or multiple time steps!
 *****************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.h>

#define FINALIZE "\
convert -delay 20 `ls -1 out*.pgm | sort -V` output.gif\n\
rm *pgm\n\
"

int **allocate_array(int N);
void free_array(int **array, int N);
void init_random(int **array1, int **array2, int N);
void print_to_pgm(int **array, int N, int t);

int main(int argc, char *argv[])
{
    int N;                    // array dimensions
    int T;                    // time steps
    int **current, **previous; // arrays - one for current timestep, one for previous timestep
    int **swap;               // array pointer
    int i, j, t, nbrs;        // helper variables

    double time; // variables for timing
    struct timeval ts, tf;

    /*Read input arguments*/
    if (argc != 3)
    {
        fprintf(stderr, "Usage: ./exec ArraySize TimeSteps\n");
        exit(-1);
    }
    else
    {
        N = atoi(argv[1]);
        T = atoi(argv[2]);
    }

```

```
52      /*Allocate and initialize matrices*/
53      current = allocate_array(N);  // allocate array for current time step
54      previous = allocate_array(N); // allocate array for previous time step
55
56      init_random(previous, current, N); // initialize previous array with pattern
57
58  #ifdef OUTPUT
59      print_to_pgm(previous, N, 0);
60  #endif
61
62      /*Game of Life*/
63
64      gettimeofday(&ts, NULL);
65
66      gettimeofday(&ts, NULL);
67
68      for (t = 0; t < T; ++t)
69      {
70
71  /* Parallelize rows; implicit barrier at loop end */
72  /*  scedule is static
73      (i, j) as loop indices are private
74      (N, previous, current) are shared, as they are enclosed by the loop
75      nbrs must be private
76      by default */
77  /* Kept here for clarity */
78  #pragma omp parallel for schedule(static) private(i, j, nbrs) shared(N, previous, current)
79          for (i = 1; i < N - 1; ++i)
80          {
81              for (j = 1; j < N - 1; ++j)
82              {
83                  nbrs =
84                      previous[i + 1][j + 1] + previous[i + 1][j] + previous[i + 1][j - 1] +
85                      previous[i][j - 1] + previous[i][j + 1] +
86                      previous[i - 1][j - 1] + previous[i - 1][j] + previous[i - 1][j + 1];
87
88                  current[i][j] = (nbrs == 3 || (previous[i][j] + nbrs == 3)) ? 1 : 0;
89              }
90          } /* implicit barrier here: all threads finished step t */
91
92  #ifdef OUTPUT
93          print_to_pgm(current, N, t + 1); /* single thread here: we're back in serial */
94  #endif
95
96          /* Safe to swap: we're outside the parallel region created by 'parallel for' */
97          swap = current;
98          current = previous;
99          previous = swap;
100     }
101
102     gettimeofday(&tf, NULL);
103     time = (tf.tv_sec - ts.tv_sec) + (tf.tv_usec - ts.tv_usec) * 0.000001;
104
105     free_array(current, N);
```

```c
    free_array(previous, N);
    printf("GameOfLife: Size %d Steps %d Time %lf\n", N, T, time);
#ifdef OUTPUT
    system(FINALIZE);
#endif
}

int **allocate_array(int N)
{
    int **array;
    int i, j;
    array = malloc(N * sizeof(int *));
    for (i = 0; i < N; i++)
        array[i] = malloc(N * sizeof(int));
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            array[i][j] = 0;
    return array;
}

void free_array(int **array, int N)
{
    int i;
    for (i = 0; i < N; i++)
        free(array[i]);
    free(array);
}

void init_random(int **array1, int **array2, int N)
{
    int i, pos, x, y;

    for (i = 0; i < (N * N) / 10; i++)
    {
        pos = rand() % ((N - 2) * (N - 2));
        array1[pos % (N - 2) + 1][pos / (N - 2) + 1] = 1;
        array2[pos % (N - 2) + 1][pos / (N - 2) + 1] = 1;
    }
}

void print_to_pgm(int **array, int N, int t)
{
    int i, j;
    char *s = malloc(30 * sizeof(char));
    sprintf(s, "out%d.pgm", t);
    FILE *f = fopen(s, "wb");
    fprintf(f, "P5\n%d %d 1\n", N, N);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            if (array[i][j] == 1)
                fputc(1, f);
            else
                fputc(0, f);
    fclose(f);
```

```
160     free(s);
161 }
162
163
```

Όσον αφορά τις λεπτομέρειες της υλοποίησής μας (όσον αφορά το παράλληλο τμήμα του προγράμματος, δηλ. τις γραμμές 69-100), επισημαίνουμε τα εξής:

- Η μεταβλητή t, δηλαδή ο αριθμός των γενεών, δεν είναι μια διαδικασία που επιδέχεται παραλληλοποίησης, καθώς αφενός μεν η μία διαδέχεται την άλλη, χωρίς να μπορούμε να πάμε στην επόμενη χωρίς να έχει τελειώσει η προηγούμενη, αφετέρου δε το μεγαλύτερο «κέρδος» προκύπτει από την παραλληλοποίηση των υπολογισμών εντός μιας γενιάς. Αυτό διότι, στο τέλος κάθε γενιάς πρέπει όλα τα threads να «συναντηθούν», ώστε να ενημερώσουν τα κοινά μας δεδομένα (δηλαδή τους πίνακες). Επομένως, στο τέλος κάθε γενιάς, υπάρχει ένα νοητό «φράγμα», στο οποίο συναντιούνται και συγχρονίζονται όλα τα threads, μέχρι όλα να τελειώσουν. Δηλαδή, οι γενιές εκτελούνται σειριακά, ενώ οι υπολογισμοί καθεμίας παράλληλα.
- Η παραλληλοποίηση του προγράμματος γίνεται στην γραμμή:

  #pragma omp parallel for schedule(static) private(i, j, nbrs) shared(N, previous, current)

  όπου παραλληλοποιούμε το for loop για το i. Σημειώνουμε (υπάρχουν και σχόλα στον κώδικα) ότι το scheduling είναι by default static, οι μεταβλητές του loop {i, j} είναι by default private και τα {N, previous, shared} by default shared, καθώς βρίσκονται εντός του παράλληλου τμήματος. Επομένως, η δήλωση των ανωτέρω δεν χρειάζεται, αλλά γίνεται για λόγους διαφάνειας. Η μεταβλητή nbrs πρέπει να δηλωθεί ως private, για να αποφευχθούν race conditions, εφόσον δεν δηλώνεται μέσα στο παράλληλο τμήμα.
- Στο τέλος του nested for loop, τα threads περιμένουν μέχρι να τελειώσουν όλα (βλ. σχόλιο) και να γίνει η ασφαλής-ατομική πρόσβαση στους πίνακες που ενημερώνουμε {previous, current}, καθώς αυτοί βρίσκονται εκτός του παράλληλου τμήματος.

Τέλος, τα αρχεία Makefile, make_on_queue.sh και run_on_queue.sh, έχουν συνταχθεί σε πλήρη αντιστοιχία με τα παραδείγματα των διαφανειών, φέρουν μικρές διαφορές που διευκολύνουν την υλοποίηση και την οργάνωση και παρουσιάζονται, για λόγους πληρότητας (χωρίς να χρειάζεται κάποια επεξήγηση), ακολούθως:

a1/Makefile

```
all: life_par

life_par: life_par.c
    gcc -O3 -fopenmp -o life_par life_par.c

clean:
    rm life_par

```

a1/make_on_queue.sh

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N makejob

## Output and error files
#PBS -o makejob.out
#PBS -e makejob.err

## How many machines should we get?
#PBS -l nodes=1

## Start
## Load appropriate module
module load openmp

## Run make in the src folder (modify properly)
cd /home/parallel/parlab05/a1/
make

```

a1/run_on_queue.sh

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N life_par

## Output and error files
#PBS -o life_par.out
#PBS -e life_par.err

## How many machines should we get?
#PBS -l nodes=1:ppn=8

## How long should the job run for?
#PBS -l walltime=01:00:00

## Module Load
module load openmp

## Defaults if not passed via -v
: "${THREADS:=8}"
: "${N:=1024}"
: "${STEPS:=1000}"

## Start
cd /home/parallel/parlab05/a1/ || exit 1

# --- OpenMP runtime settings ---
export OMP_NUM_THREADS="${THREADS}"   # 1,2,4,6,8 per the assignment

# Run and capture outputs by config
RESULT_DIR="benchmarks/N${N}_T${THREADS}"
mkdir -p "${RESULT_DIR}"

./life_par "${N}" "${STEPS}" \
  > "${RESULT_DIR}/life_${THREADS}_${N}.out" \
  2> "${RESULT_DIR}/life_${THREADS}_${N}.err"


```

▪ Ενότητα 1.4.4 – Οι Μετρήσεις

Έχοντας συντάξει το παράλληλο πρόγραμμά μας, το υποβάλλαμε στα μηχανήματα του εργαστηρίου για κάθε συνδυασμό των παραμέτρων {μέγεθος ταμπλό, πυρήνες}. Οι παράμετροι αυτές, πήραν τις ακόλουθες τιμές, όπως ζητούνταν από την εκφώνηση της άσκησης:

- Μέγεθος ταμπλό: {64, 1024, 4096}
- Πυρήνες: {1, 2, 4, 6, 8}

Σύνολο 3*5=15 υποβολές.

Έτσι, ανοίγοντας τα αντίστοιχα αρχεία “filename.out”, λάβαμε τα ακόλουθα αποτελέσματα, τα οποία και παρουσιάζουμε στον κάτωθι πίνακα.

SIZE	THREADS	TIME	SPEEDUP
64	1	0.022613	1.000000
64	2	0.013416	1.685525
64	4	0.009780	2.312168
64	6	0.008942	2.528853
64	8	0.009163	2.467860
1024	1	11.793298	1.000000
1024	2	5.868753	2.009507
1024	4	2.929962	4.025069
1024	6	1.965196	6.001080
1024	8	1.476472	7.987485
4096	1	189.347966	1.000000
4096	2	94.832356	1.996660
4096	4	47.729763	3.967084
4096	6	32.393775	5.845196
4096	8	28.594441	6.621845

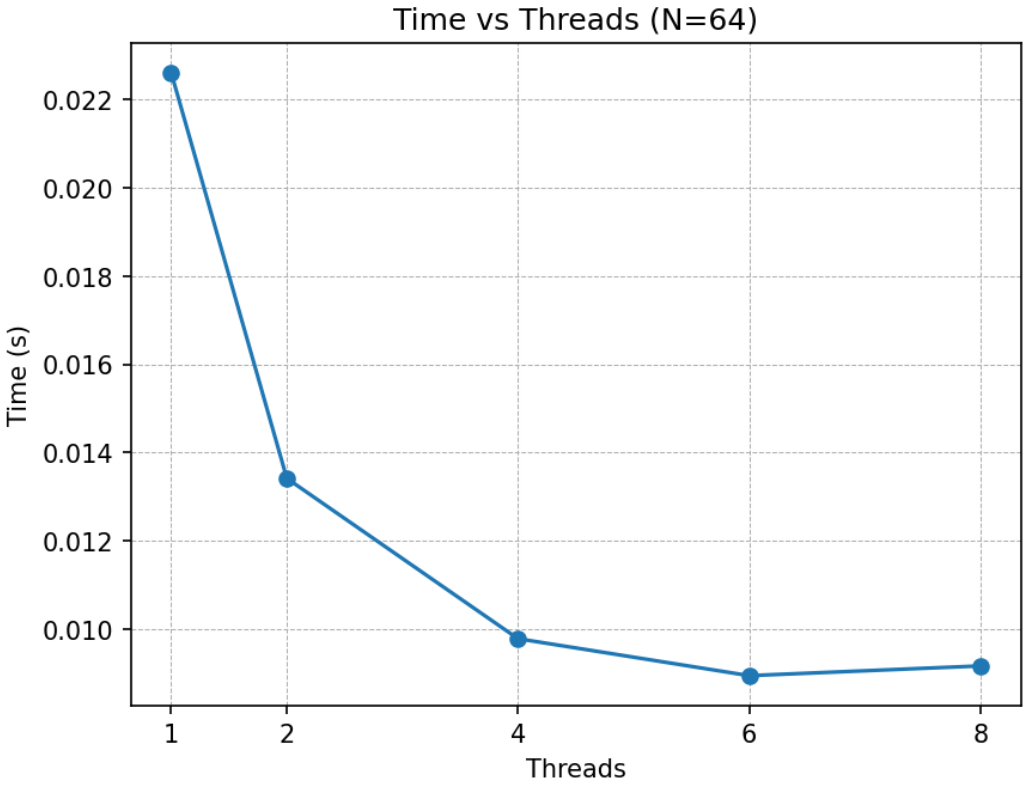
Τα αποτελέσματα αυτά αποτελούν τα δείγματα αναφοράς, με βάση τα οποία θα καταστρώσουμε τα ζητούμενα διαγράμματα (χρόνου και speedup) του επόμενου ερωτήματος. Επισημαίνουμε ότι:

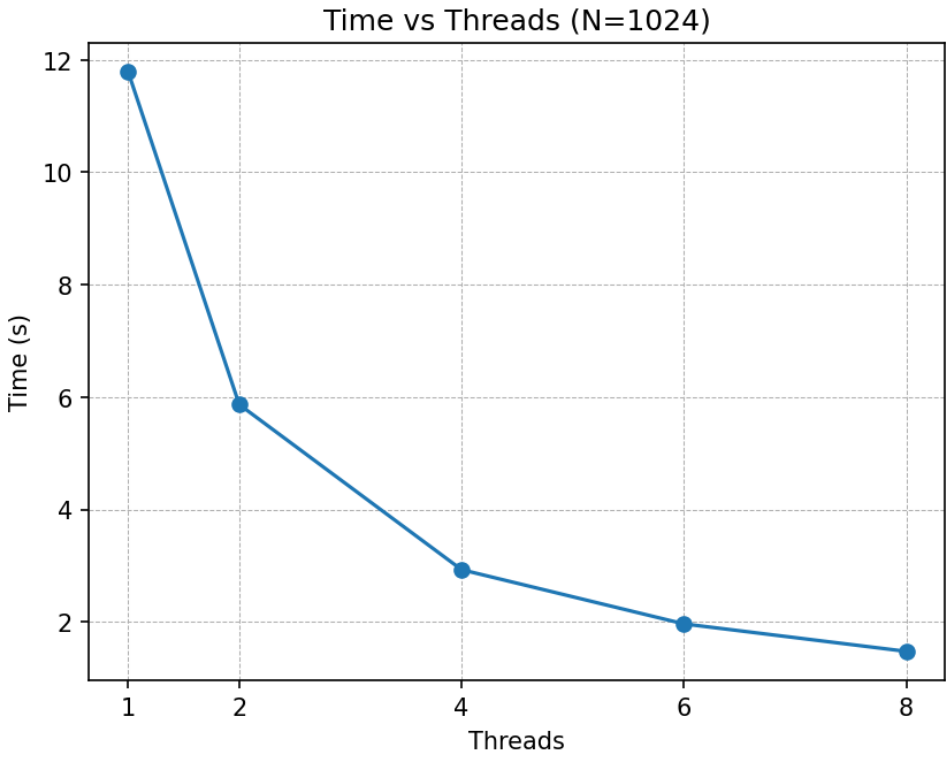
$$\boldsymbol{Speedup} = \frac{Time\_of\_N\_cores}{Time\_of\_1\_core}$$

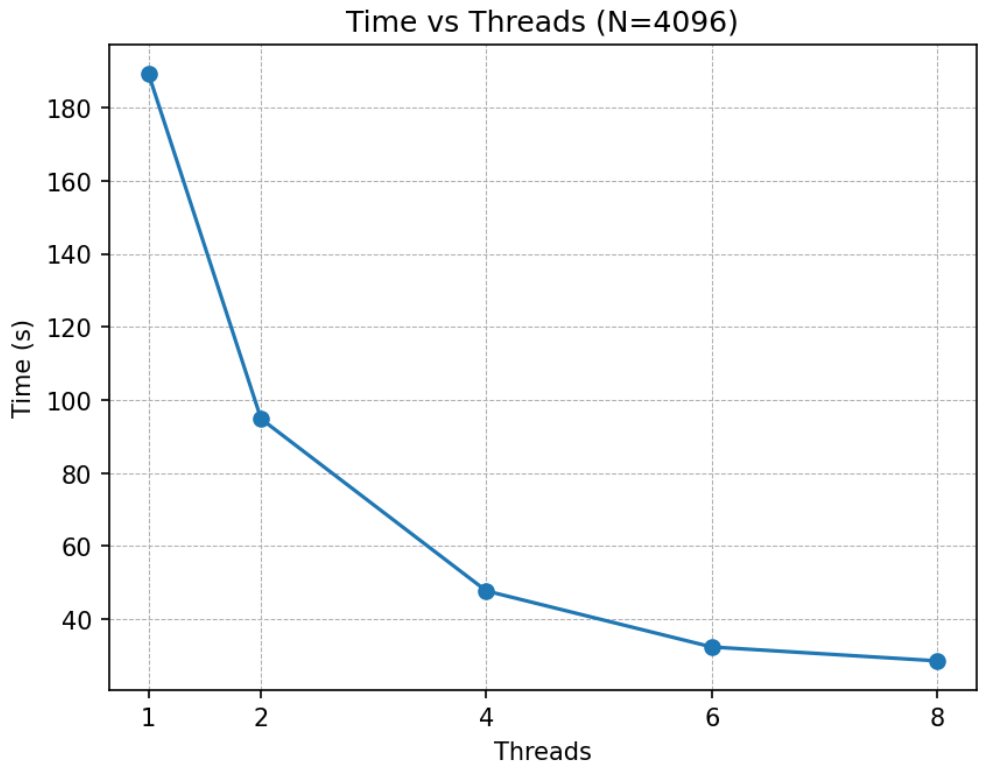
▪ Ενότητα 1.4.5 – Τα Διαγράμματα

Σύμφωνα με τις μετρήσεις του παραπάνω πίνακα, καταστρώνουμε τα ακόλουθα διαγράμματα για κάθε μέγεθος ταμπλό (με χρήση ενός προγράμματος Python):

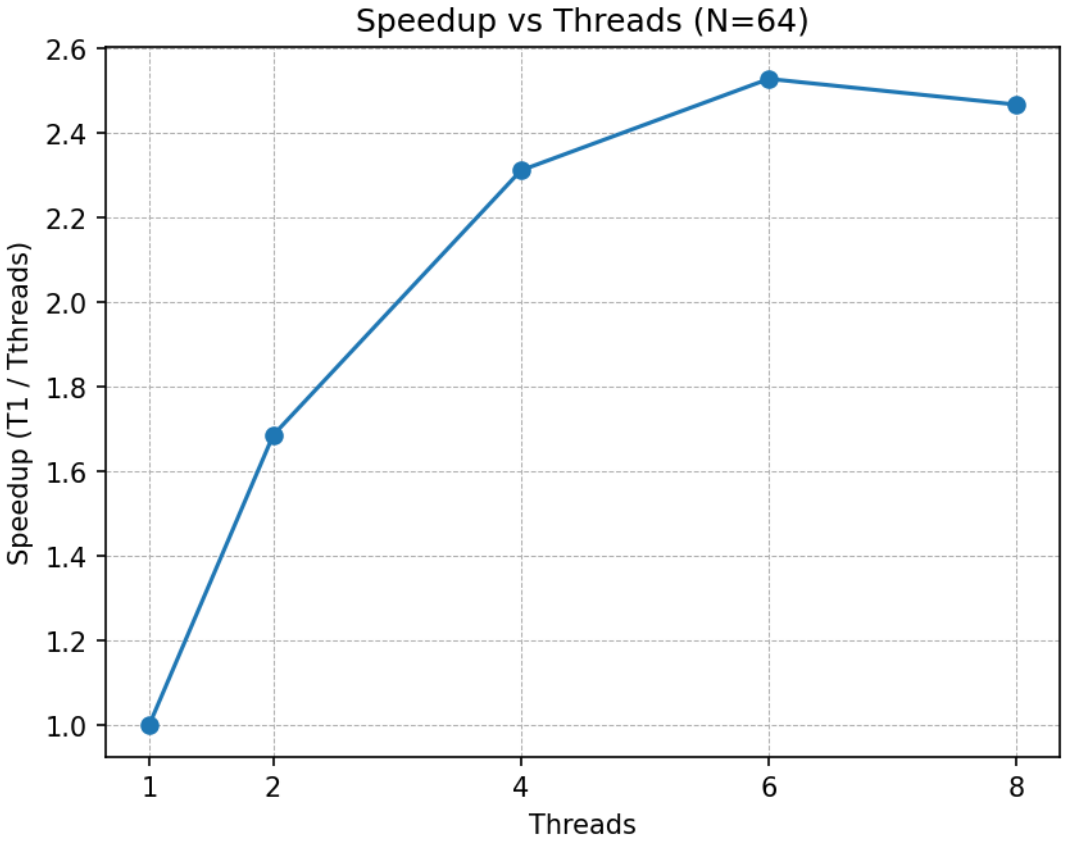
Διαγράμματα Χρόνου

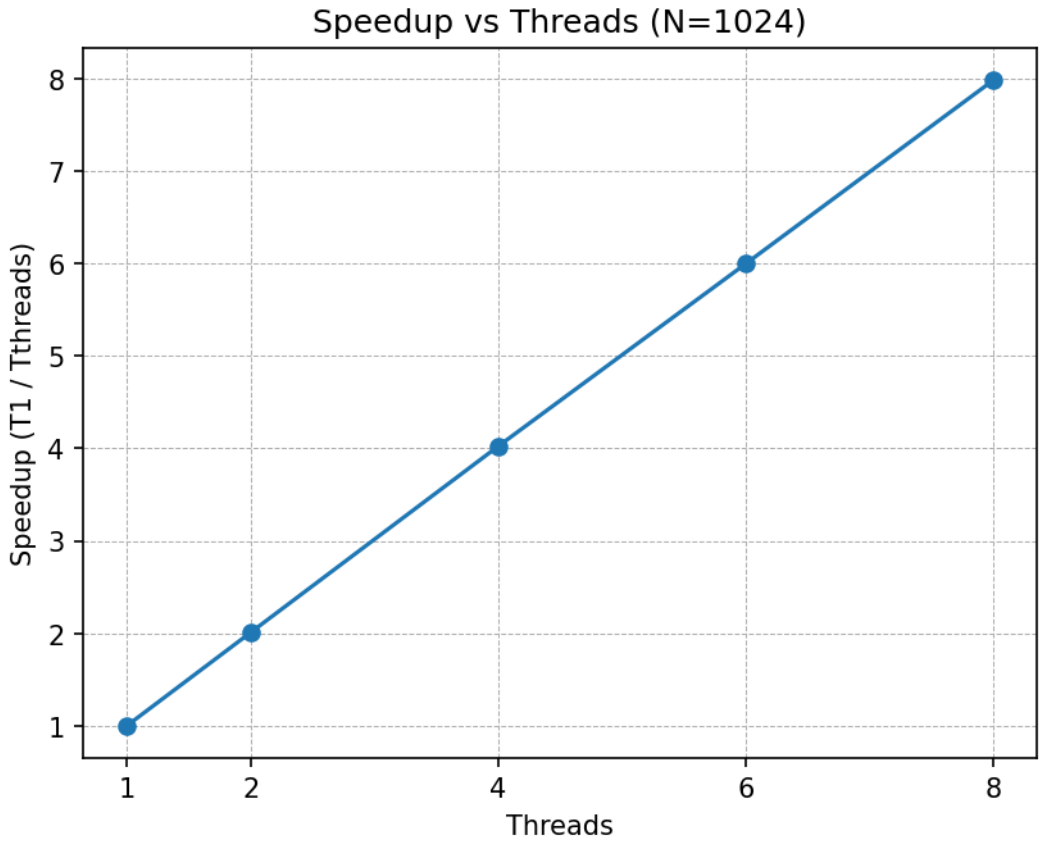


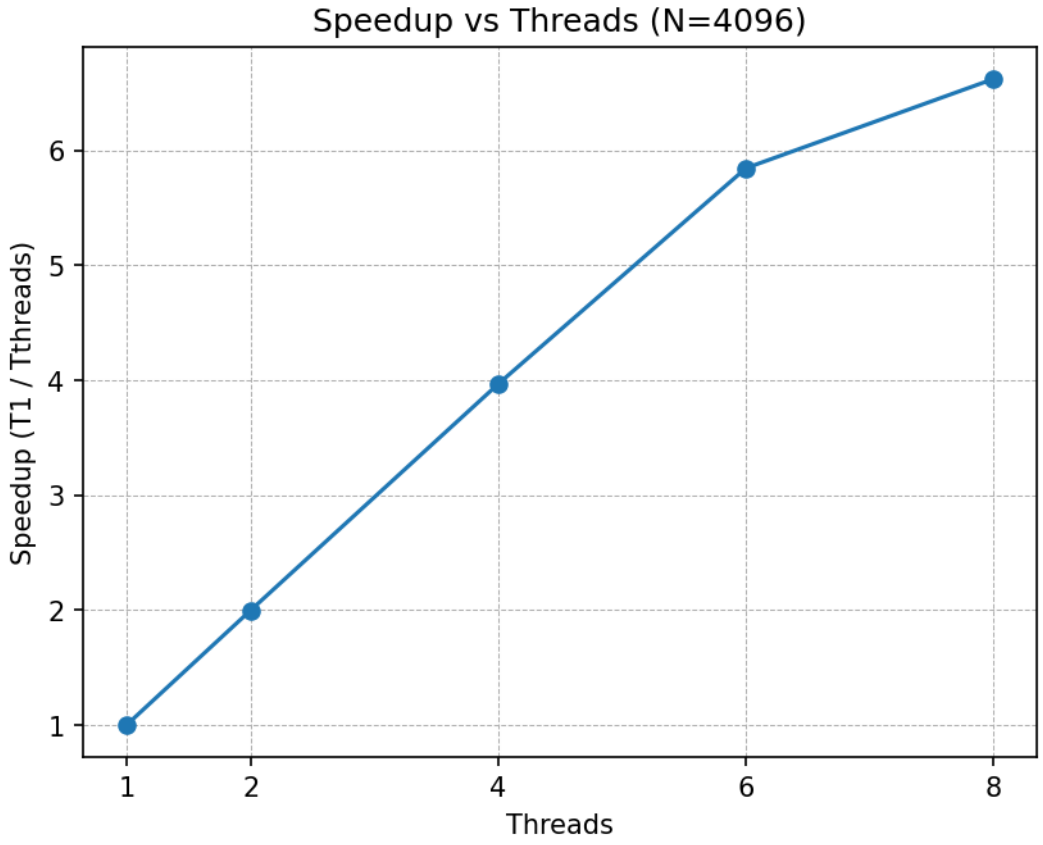




Διαγράμματα Speedup




Speedup vs Threads (N=1024)
Speedup (T1 / Tthreads)
Threads
1
2
3
4
5
6
7
8
1
2
4
6
8


Speedup vs Threads (N=4096)
Speedup (T1 / Tthreads)
Threads
1
2
3
4
5
6
1
2
4
6
8

Με βάση τα παραπάνω αποτελέσματα και διαγράμματα παρατηρούμε ότι:

- **Για N=64**, ο χρόνος εκτέλεσης δεν κλιμακώνει πέραν των 2 threads (δηλ. δεν είναι ανάλογος του 1/threads) και από τα 4 threads και έπειτα μειώνεται ελάχιστα με την αύξηση των νημάτων. Το speedup είναι μικρότερο από γραμμικό (κοίλη συνάρτηση) και στα 8 νήματα εμφανίζει μικρή υποχώρηση. Αυτό οφείλεται στο μικρό φορτίο ανά νήμα: το κόστος δημιουργίας και συγχρονισμού των νημάτων αποτελεί σημαντικό ποσοστό του συνολικού χρόνου και άρα δεν έχουμε μεγάλο CPU intensity.
- **Για N=1024**, ο χρόνος μειώνεται σχεδόν γραμμικά με τον αριθμό νημάτων, επιτυγχάνοντας ιδανική κλιμάκωση (~ 1/threads). Το πρόβλημα είναι αρκετά μεγάλο ώστε να επικρατεί ο υπολογιστικός φόρτος έναντι του overhead δημιουργίας νημάτων, ενώ τα δεδομένα χωρούν πλήρως στην cache, αποφεύγοντας καθυστερήσεις από τη μνήμη. Έτσι, και η επιτάχυνση είναι γραμμική.
- **Για N=4096**, η επιτάχυνση αρχικά είναι σχεδόν γραμμική έως τα 4 νήματα, αλλά στη συνέχεια μειώνεται. Από τα 6 και ειδικά στα 8 νήματα, η βελτίωση της απόδοσης περιορίζεται, καθώς το πρόβλημα γίνεται memory-bound (έχουμε αρχιτεκτονική κοινής μνήμης, αφού τρέχουμε το πρόγραμμα σε 1 node, με πολλά threads): η αυξημένη κίνηση στη μνήμη και το περιορισμένο εύρος ζώνης (bandwidth) επιβραδύνουν την περαιτέρω κλιμάκωση, παρότι ο συνολικός χρόνος συνεχίζει να μειώνεται. Έτσι, η επιτάχυνση αρχικά είναι γραμμική, αλλά στο τέλος γίνεται κοίλη.

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.

Οκτώβριος 2025