Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Συστήματα Παράλληλης Επεξεργασίας

9ο εξάμηνο

Άσκηση 1:Παραλληλοποίηση Αλγορίθμων σε Πολυπύρηνες Αρχιτεκτονικές Κοινής Μνήμης
Ενδιάμεση Αναφορά

Δαζέα Ελένη 03114060
Καναβάκης Ελευθέριος 03114180

1. Conway's Game of Life

Σκοπός

Σκοπός της συγκεκριμένης άσκησης είναι η εξοικείωση με τις υποδομές του εργαστηρίου (πρόσβαση στα συστήματα, μεταγλώττιση προγραμμάτων, υποβολή εργασιών κλπ) μέσα από την παραλληλοποίηση ενός απλού προβλήματος σε αρχιτεκτονικές κοινής μνήμης.

Υλοποίηση

Παραλληλοποιήσαμε το Game_Of_Life.c. Το τροποποιήσαμε ώστε να τρέχει για 1000 γενιές με array size N= 64,1024,4096 και για 1,2 4,6,8 threads.Έπειτα τρέχοντας το qsub -q parlab scripts/run_on_clones_Omp.sh λαμβάνουμε τα αποτελέσματα στο run_omp.out:

#threads = 1

GameOfLife: Size 64 Steps 1000 Time 0.030790
GameOfLife: Size 1024 Steps 1000 Time 12.131021
GameOfLife: Size 4096 Steps 1000 Time 195.056502

#threads = 2

GameOfLife: Size 64 Steps 1000 Time 0.017332
GameOfLife: Size 1024 Steps 1000 Time 6.041834
GameOfLife: Size 4096 Steps 1000 Time 97.761518

#threads = 4

GameOfLife: Size 64 Steps 1000 Time 0.011827
GameOfLife: Size 1024 Steps 1000 Time 3.016668
GameOfLife: Size 4096 Steps 1000 Time 49.155209

#threads = 6

GameOfLife: Size 64 Steps 1000 Time 0.010139
GameOfLife: Size 1024 Steps 1000 Time 2.024827
GameOfLife: Size 4096 Steps 1000 Time 34.518015

#threads = 8

GameOfLife: Size 64 Steps 1000 Time 0.009814
GameOfLife: Size 1024 Steps 1000 Time 1.527707
GameOfLife: Size 4096 Steps 1000 Time 33.665297

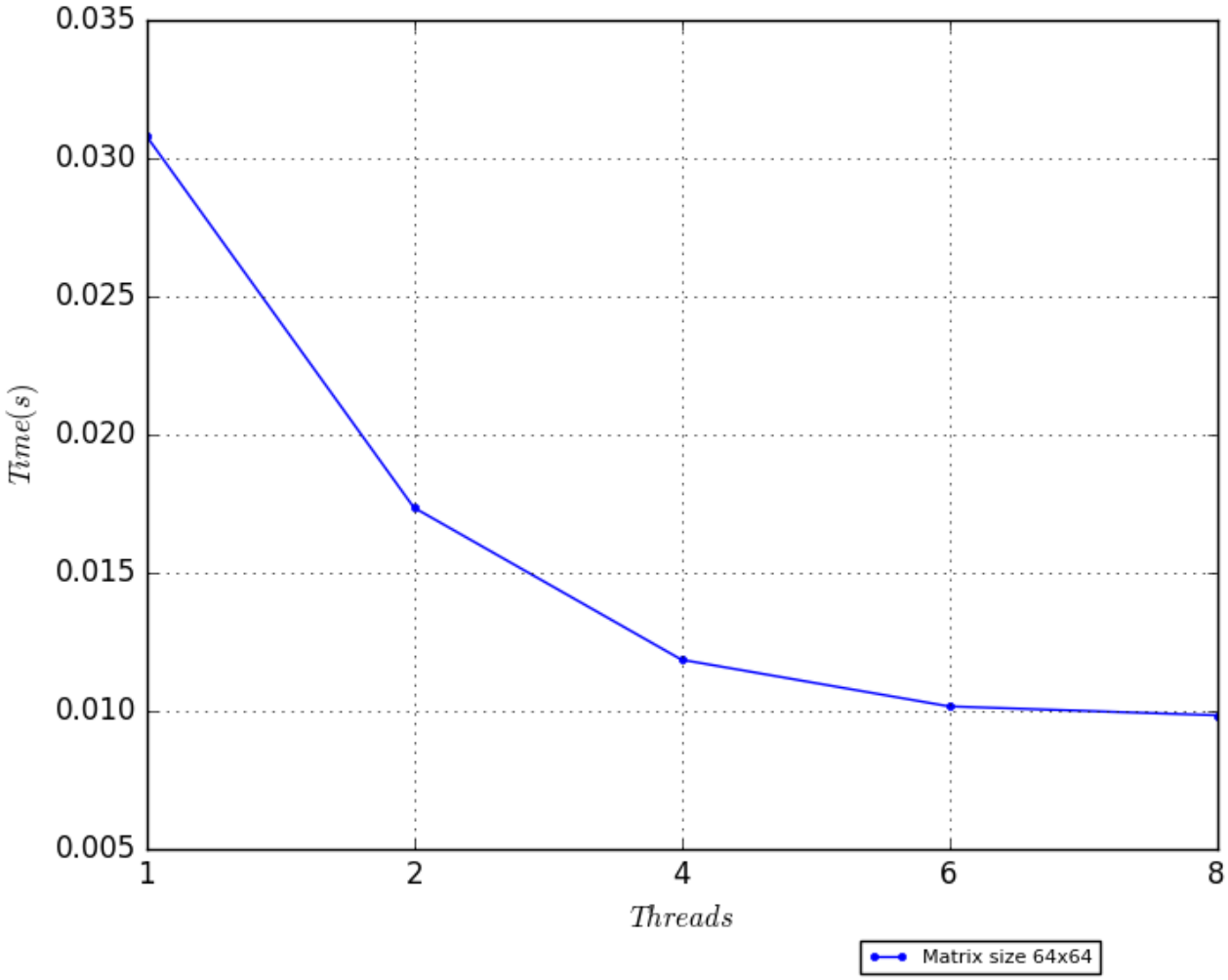
Πραγματοποιήσαμε μετρήσεις επίδοσης σε έναν από τους κόμβους της συστοιχίας των clones για 1,2,4,6,8 πυρήνες και μεγέθη ταμπλώ 64x64,1024x1024 και 4096x4096 (σε όλες τις περιπτώσεις τρέξαμε το παιχνίδι για 1000 γενιές).

Συγκεντρώσαμε τα αποτελέσματά μας σε γραφήματα για το χρόνο και το speedup T_s/T_p

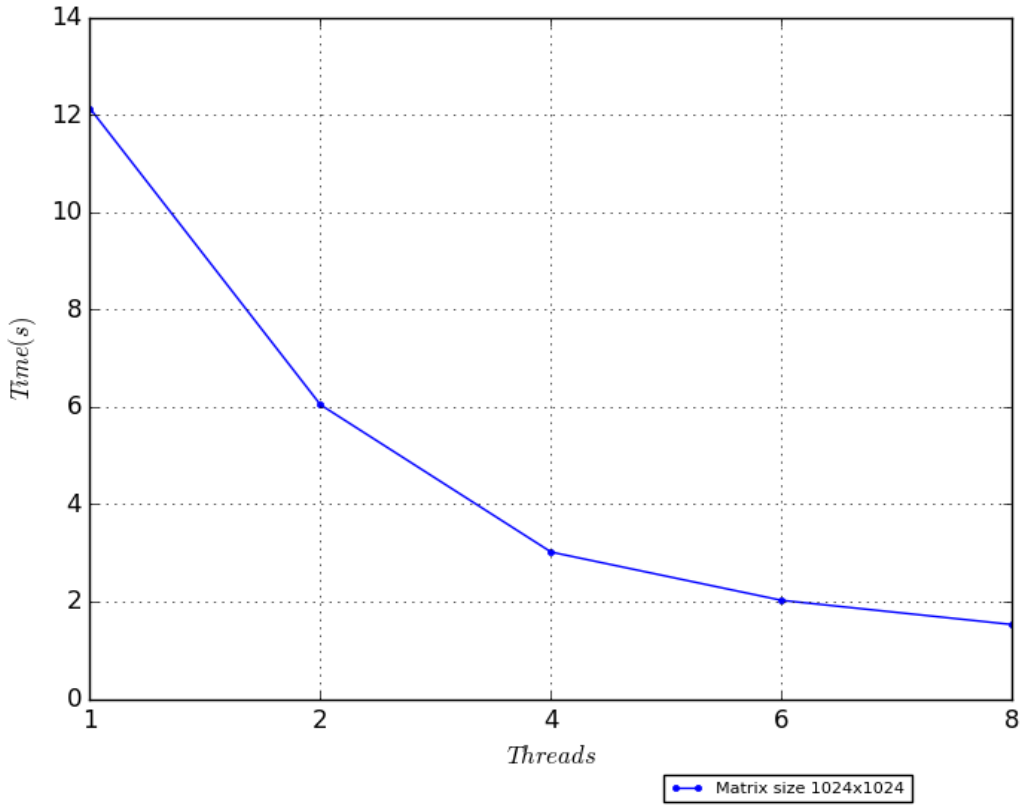
Τα γραφήματα δίνονται από κάτω:

Time:

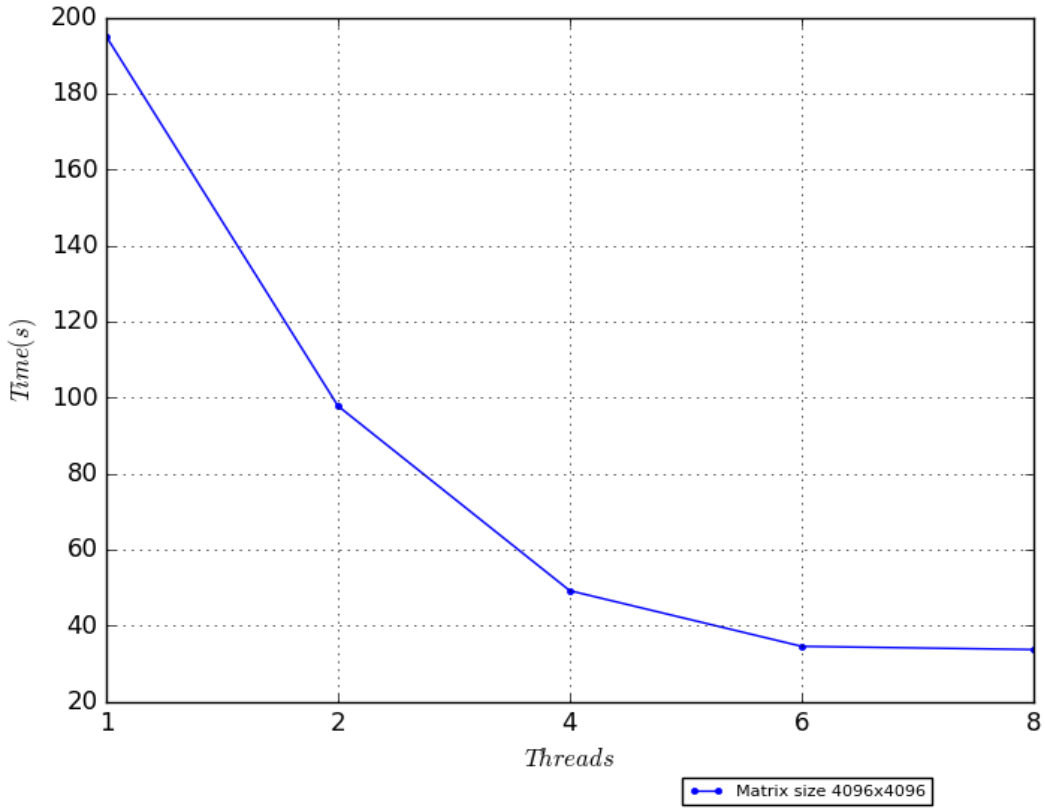
64x64 :



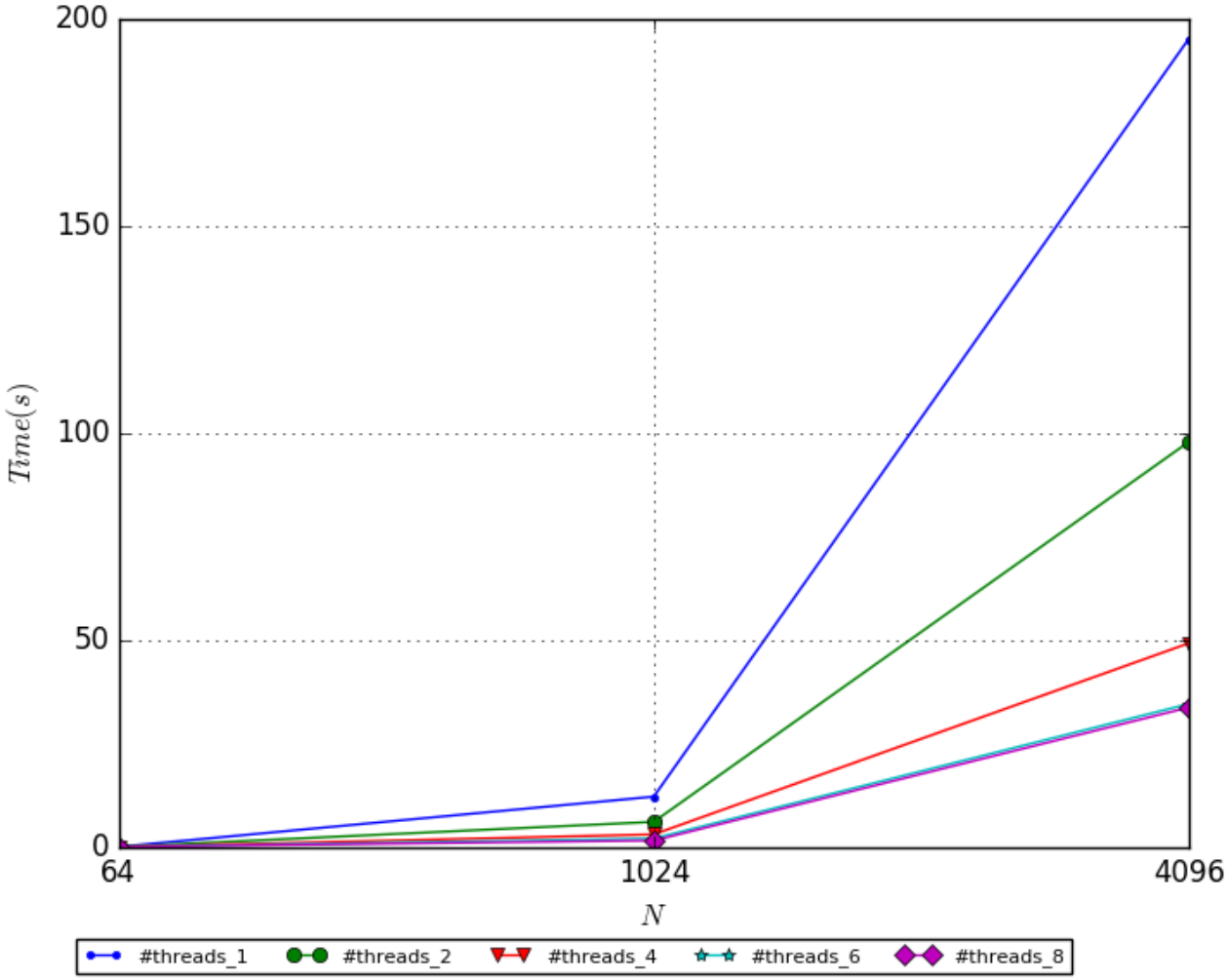
1024x1024:



4096x4096 :

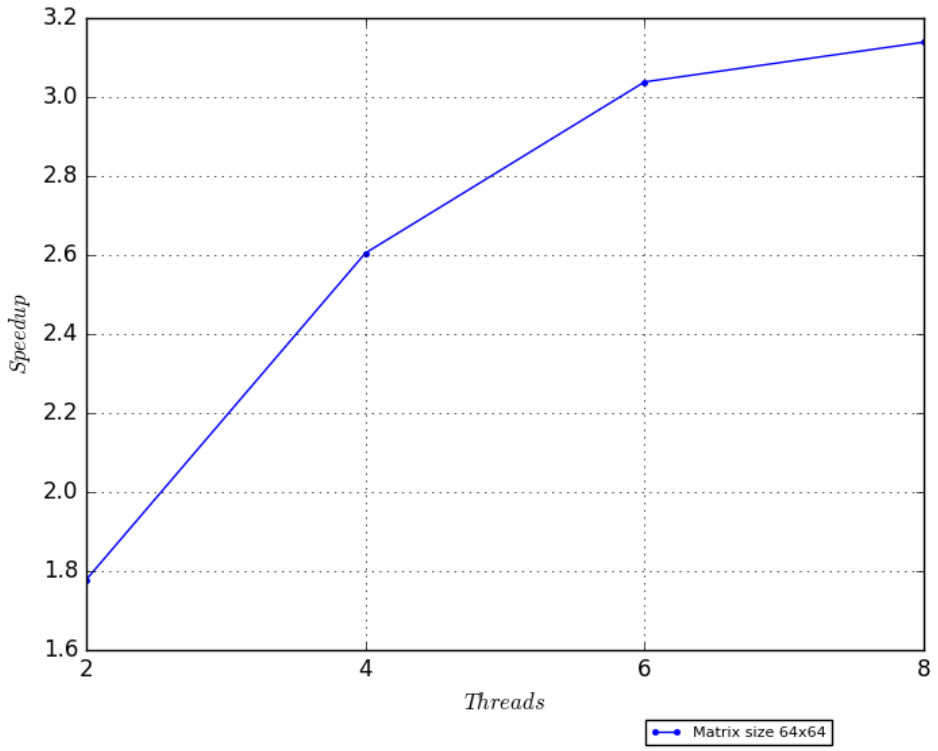


Συνολικά έχουμε :



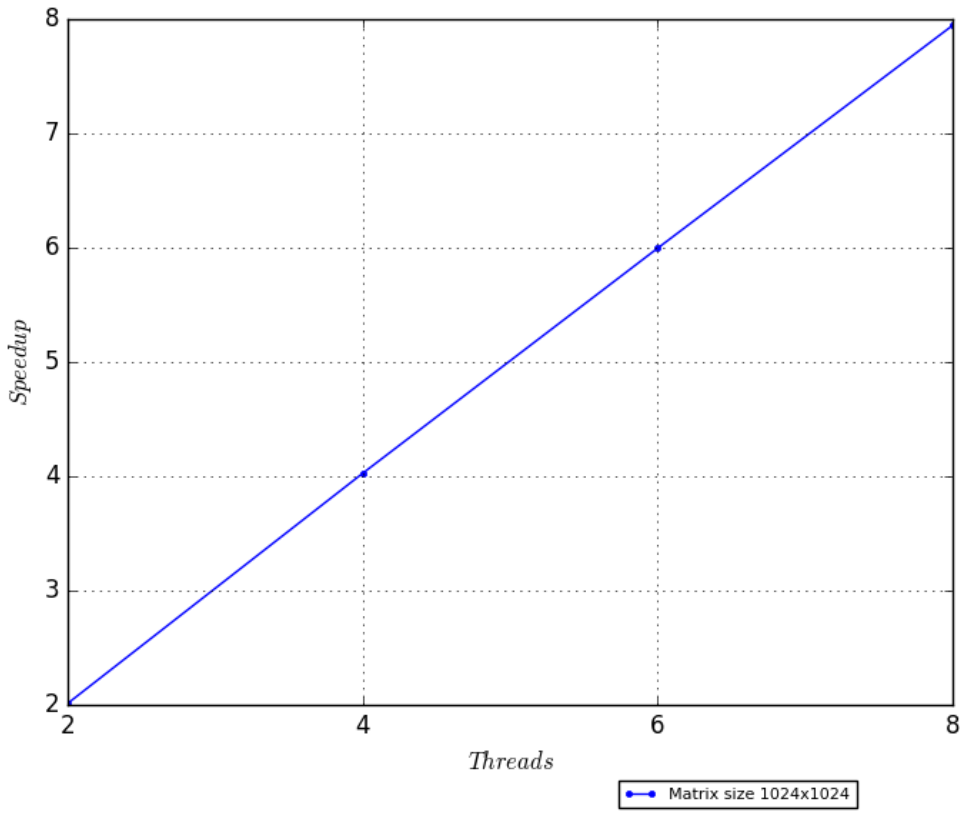
Speedup:

64x64 :

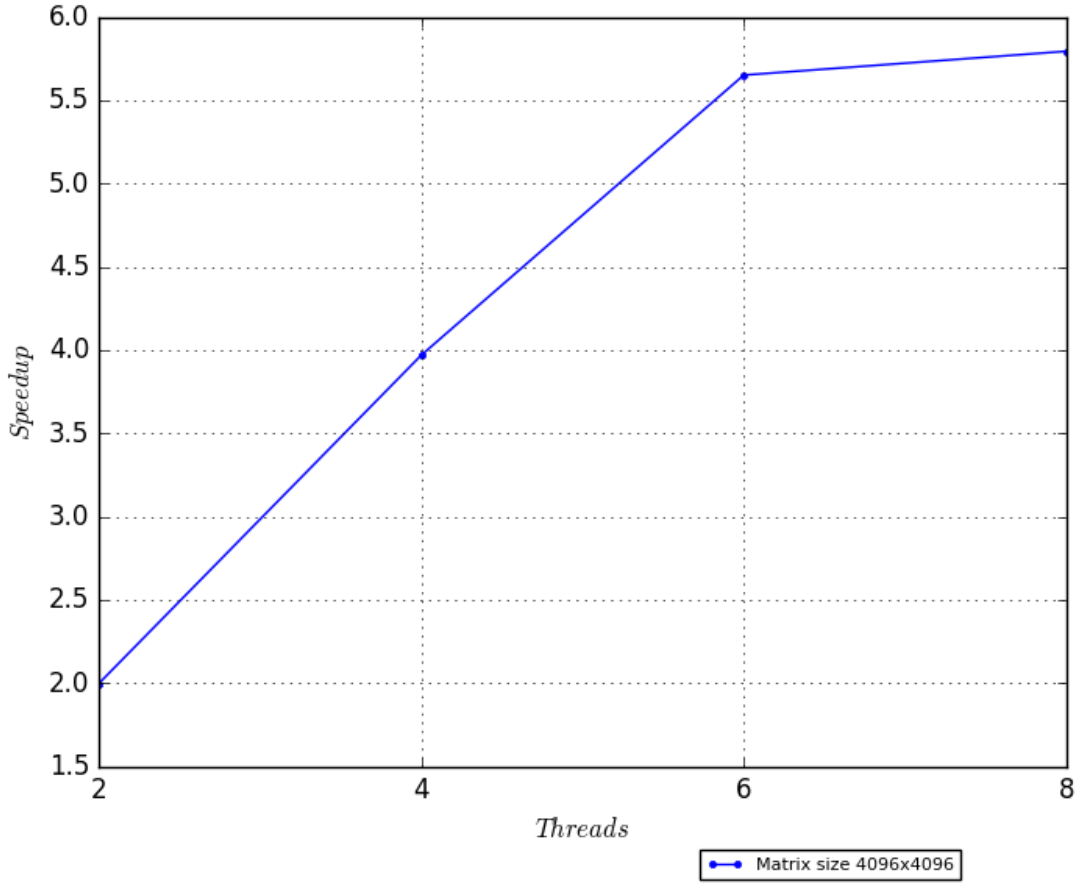


1024x1024

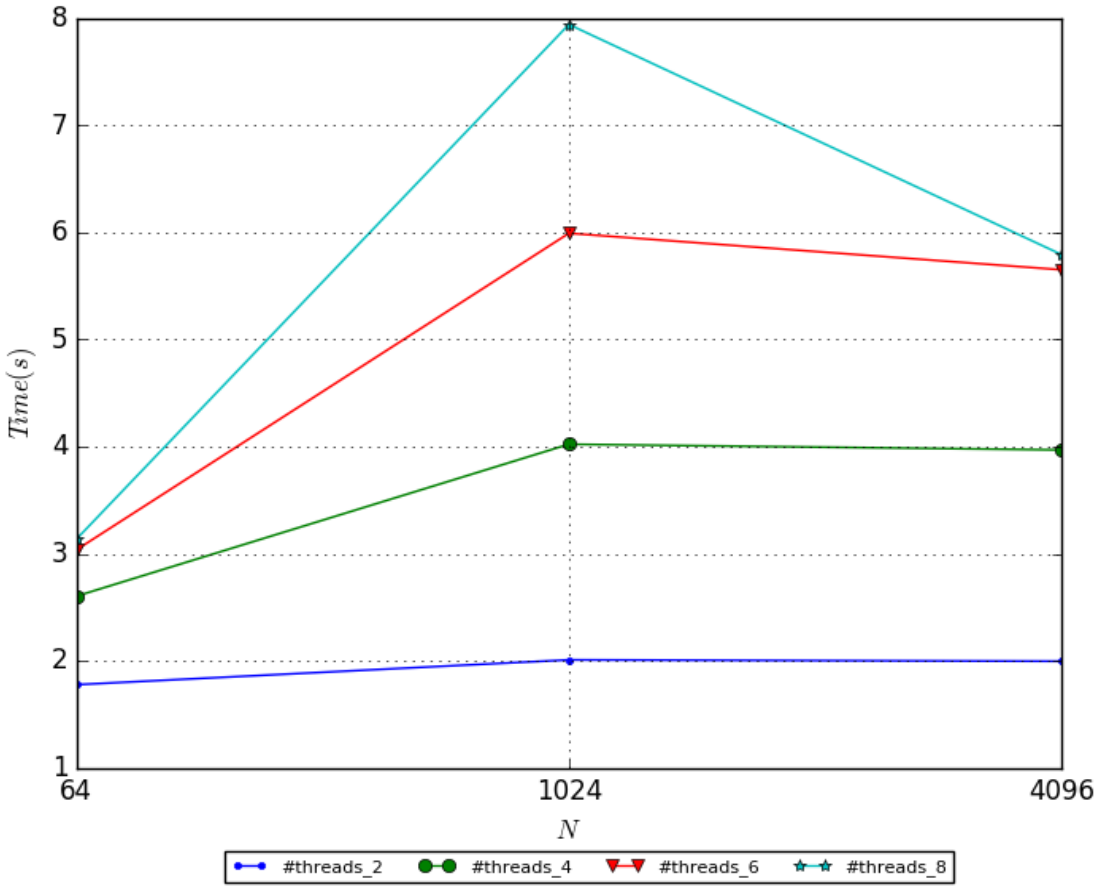
1024x1024 :



4096x4096 :



Συνολικά έχουμε :



Παρατηρήσεις-Σχόλια:

Παρατηρώντας κανείς τις παραπάνω γραφικές παραστάσεις εύκολα μπορεί να καταλάβει ότι αύξηση του αριθμού των πυρήνων συνεπάγεται μείωση του χρόνου εκτέλεσης και κατ' επέκταση αύξηση του speedup. Το γεγονός αυτό είναι αρκετά λογικό καθώς στον κώδικα μας υπάρχει εξάρτηση μόνο στον άξονα του χρόνου . Έτσι μπορούμε εύκολα να παραλληλοποιήσουμε τον κώδικα μας αναθέτοντας σε κάθε thread την υλοποίηση μίας γραμμής του πίνακα current .

Στην συνέχεια , θα σχολιάσουμε αναλυτικότερα τις παραπάνω γραφικές παραστάσεις καθώς κάποια από τα αποτελέσματά μας παρουσιάζουν ιδιαίτερο ενδιαφέρον. Παρατηρούμε λοιπόν , ότι ναι μεν η αύξηση των πυρήνων συντελεί σε μείωση του συνολικού χρόνου εκτέλεσης αλλά ο ρυθμός μείωσης του χρόνου αυτού γίνεται όλο και μικρότερος όσο αυξάνεται ο αριθμός των πυρήνων. Το γεγονός αυτό μπορεί να έχει παραπάνω από μία εξηγήσεις . Για την περίπτωση του πίνακα με μέγεθος 64x64 θα μπορούσαμε να πούμε ότι το μέγεθος του είναι αρκετά μικρό για να επωφεληθεί ιδιαίτερα από την αύξηση των πυρήνων . Για την περίπτωση του πίνακα 4096x4096 τώρα ο κορεσμός που παρατηρείται πιθανώς να οφείλεται σε capacity misses (λόγω του μεγάλου μεγέθους του πίνακα) καθώς και σε conflict misses (καθώς περισσότερα από ένα cache blocks γράφονται στο ίδιο cache line). Παράλληλα , το παραπάνω φαινόμενο σίγουρα σχετίζεται και με τα locks που χρησιμοποιεί το openmp για συγχρονισμό των private μεταβλητών . Είναι λογικό πως με τα locks αυτά όσο αυξάνεται ο αριθμός των thread τόσο αυξάνεται και ο χρόνος αναμονής για να γίνει acquire κάποιο lock. Βέβαια , παρατηρούμε ότι για μέγεθος πίνακα 1024x1024 ο παραπάνω κορεσμός δεν παρατηρείται . Το γεγονός αυτό πιθανώς να οφείλεται στο ότι το μέγεθος αυτό είναι ιδανικό ώστε να αποφεύγεται η παραπάνω καθυστέρηση.

Τέλος , δοκιμάσαμε να παραλληλοποιήσουμε και τα δύο loops με τον κώδικα που φαίνεται παρακάτω. Η υλοποίηση αυτή λοιπόν , ενώ ήταν αισθητά καλύτερη από την αρχική (σειριακή) παρουσίασε αποτελέσματα χειρότερα από την υλοποίηση που χρησιμοποιήσαμε παραπάνω καθώς αυξάνεται αισθητά ο αριθμός των waits (barrier) ειδικά στο δεύτερο loop. Μάλιστα, το barrier στο loop αυτό δεν φαίνεται να έχει και ιδιαίτερο νόημα. Επιπρόσθετα , μεγάλο είναι και το κόστος του διαδρόμου (coherence misses) καθώς με κάθε write στον πίνακα current αυτός πρέπει να γίνει invalid σε όλες τις υπόλοιπες caches και να μεταφερθεί στην μνήμη .

Ο κώδικας της δεύτερης υλοποίησης φαίνεται παρακάτω :

```
    for ( t = 0 ; t < T ; t++ ) {
        #pragma omp parallel for private (i)
        for ( i = 1 ; i < N-1 ; i++ ) {
            #pragma omp parallel for private(j,nbrs)
            for ( j = 1 ; j < N-1 ; j++ ) {
                nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-
1] \
                    + previous[i][j-1] + previous[i][j+1] \
                    + previous[i-1][j-1] + previous[i-1][j] + previous[i-
1][j+1];
                if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
                        current[i][j]=1;
                else
                        current[i][j]=0;
            }
        }
```

Κώδικας:

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

#define FINALIZE "\
convert -delay 20 out*.pgm 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 t, i, j, 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]);
    }

    /*Allocate and initialize matrices*/
    current = allocate_array(N);            //allocate array for current time
step
    previous = allocate_array(N);           //allocate array for previous time
step

    init_random(previous, current, N);  //initialize previous array with pattern

    //#ifdef OUTPUT
```

```
    //print_to_pgm(previous, N, 0);
    //#endif

    /*Game of Life*/

    gettimeofday(&ts,NULL);
    for ( t = 0 ; t < T ; t++ ) {
        #pragma omp parallel for shared(previous, current) private (i,j,nbrs)
        for ( i = 1 ; i < N-1 ; i++ ) {
            for ( j = 1 ; j < N-1 ; j++ ) {
                nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-
1] \
                    + previous[i][j-1] + previous[i][j+1] \
                    + previous[i-1][j-1] + previous[i-1][j] + previous[i-
1][j+1];
                if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
                        current[i][j]=1;
                else
                        current[i][j]=0;
            }
        }

        //#ifdef OUTPUT
        //print_to_pgm(current, N, t+1);
        //#endif
        //Swap current array with previous array
        swap=current;
        current=previous;
        previous=swap;

    }
    gettimeofday(&tf,NULL);
    time=(tf.tv_sec-ts.tv_sec)+(tf.tv_usec-ts.tv_usec)*0.000001;

    free_array(current, N);
    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);
    free(s);
}
```

make_on_queue.sh :

```
#!/bin/bash

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

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

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

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab26/ex1a
make
```

run_on_queue.sh :

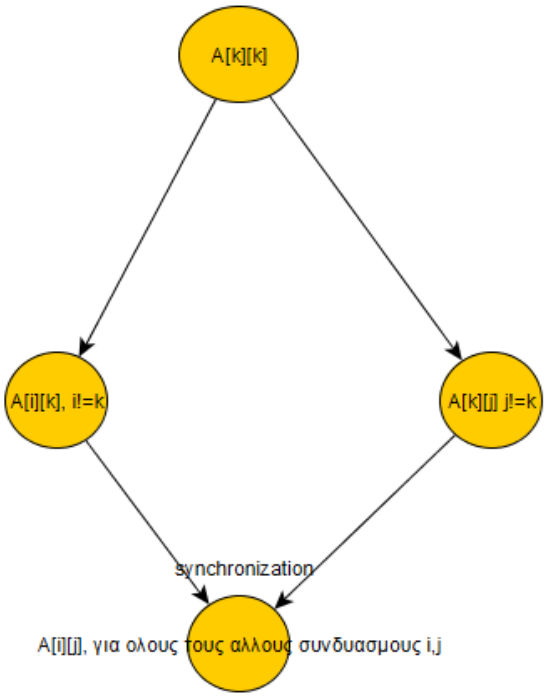
```
#!/bin/bash

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

## Output and error files
#PBS -o run_gol2.out
#PBS -e run_gol2.err
```

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

##How long should the job run for?
#PBS -l walltime=00:20:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab26/ex1a
export OMP_NUM_THREADS=1
echo "#threads = 1"
echo " "
./game_of_life 64 1000
./game_of_life 1024 1000
./game_of_life 4096 1000
echo " "
echo "-----------------------------------------------"
echo " "
export OMP_NUM_THREADS=2
echo "#threads = 2"
echo " "
./game_of_life 64 1000
./game_of_life 1024 1000
./game_of_life 4096 1000
echo " "
echo "-----------------------------------------------"
echo " "
export OMP_NUM_THREADS=4
echo "#threads = 4"
echo " "
./game_of_life 64 1000
./game_of_life 1024 1000
./game_of_life 4096 1000
echo " "
echo "-----------------------------------------------"
echo " "
export OMP_NUM_THREADS=6
echo "#threads = 6"
echo " "
./game_of_life 64 1000
./game_of_life 1024 1000
./game_of_life 4096 1000
echo " "
echo "-----------------------------------------------"
echo " "
export OMP_NUM_THREADS=8
echo "#threads = 8"
echo " "
./game_of_life 64 1000
./game_of_life 1024 1000
./game_of_life 4096 1000
echo " "
echo "-----------------------------------------------"
echo " "
```

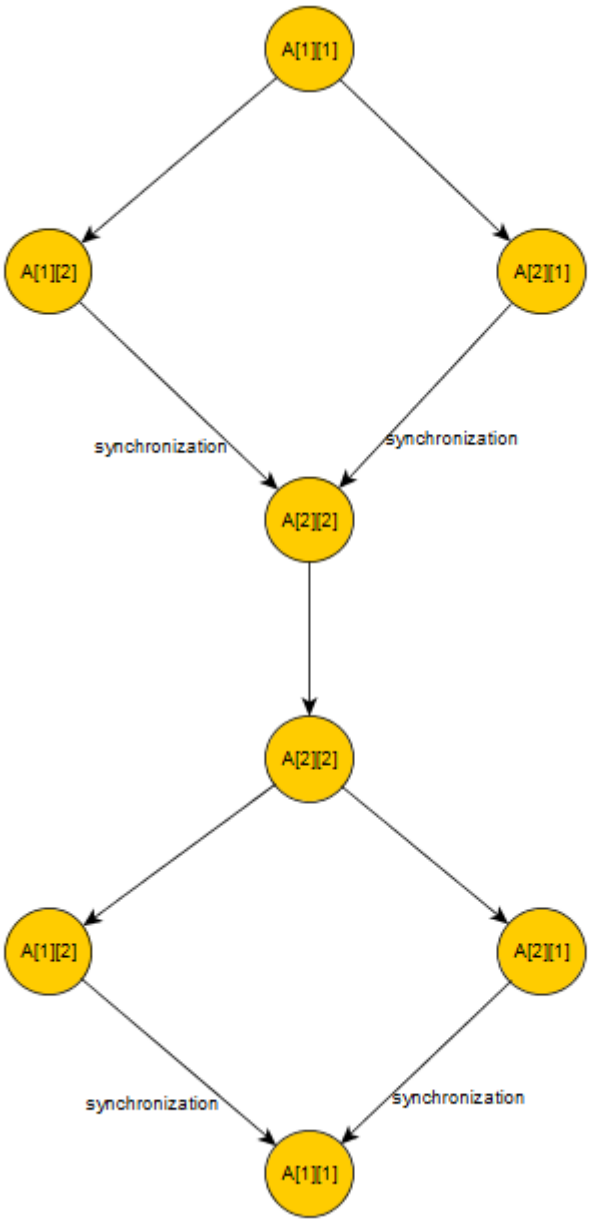
2. Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου Floyd-Warshall σε αρχιτεκτονικές κοινής μνήμης

Παρακάτω παραθέτουμε τα task graphs για τις διάφορες υλοποιήσεις του Floyd-Warshall:

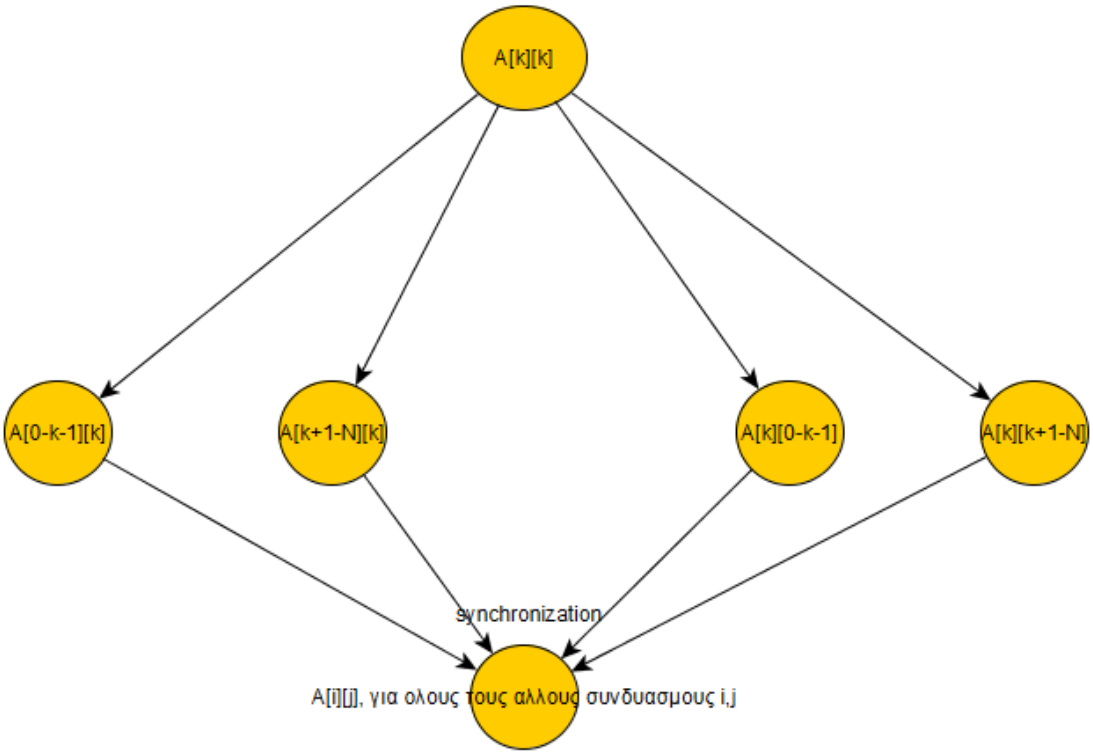
Σειριακή :



Αναδρομική :


A[1][1]
A[1][2]
A[2][1]
synchronization
synchronization
A[2][2]
A[2][2]
A[1][2]
A[2][1]
synchronization
synchronization
A[1][1]

Tiled :


A[k][k]
A[0-k-1][k]
A[k+1-N][k]
A[k][0-k-1]
A[k][k+1-N]
synchronization
A[i][j], για ολους τους αλλους συνδυασμους i,j