Τελική Αναφορά 1ης Άσκησης

Φίλιππος Μαλανδράκης - 03116200
Βιτάλιος Σαλής - 03115751

Ζήτημα 1 - Conway's Game of Life

Εισαγωγή

Σε αυτό το ζήτημα μας δίνεται σειριακός C κώδικας, ο οποιός υλοποιεί το Conway's Game of Life, και καλούμαστε να:

1) Μετατρέψουμε την σειριακή υλοποίηση σε μια παράλληλη με την χρήση του OpenMP.
2) Πραγματοποιήσουμε μετρήσεις επίδοσης σε διάφορους αριθμούς από threads και μεγέθη ταμπλό.
3) Επιδείξουμε τις μετρήσεις αυτές σε γραφήματα όπου θα συγκρίνεται το speedup ή ο χρόνος εκτέλεσης με τον αριθμό απο threads.

Υλοποίηση

Η μετατροπή της σειριακής υλοποίησης σε μια παράλληλη με την χρήση του OpenMP, χρειάστηκε μια μοναδική αλλαγή. Η πρόσθεση της γραμμής:

#pragma omp parallel for private(nbrs)

πριν την εκτέλεση του τρίτου εμφωλευμενου loop στην υλοποίηση του αλγορίθμου.

Αυτή η εντολή, δείχνει στο OpenMP πως θέλουμε να παραλληλοποιήθει το for loop που ακολουθεί και να έχει ως ιδιωτική μεταβλητή την *nbrs*, δηλαδή να δημιουργηθεί ένα τοπικό αντίγραφο σε κάθε thread. Η ανάθεση των tasks γίνεται με στατικό τρόπο μιας και η δουλειά μπορεί να κατανεμηθεί σε ισόποσα κομμάτια.

Επιλέξαμε την χρήση του *parallel for* αντί του παραδοσιακού *parallel* μιας και το OpenMP στην πρώτη περίπτωση αναλαμβάνει την διάσπαση

του loop σε ισόποσα κομμάτια ώστε να μοιραστούν σε threads κάνοντας πολύ πιο εύκολη την παραλληλοποίηση. Αντίθετα, αν χρησιμοποιούσαμε το *parallel* θα χρειαζόταν να υλοποιήσουμε μόνοι μας την διαδικασία της παραλληλοποίησης, και πιθανώς με χειρότερο τρόπο.

Τέλος, επιλέξαμε πως οι μεταβλητές *nbrs* και *j* θα είναι ιδιωτικές για κάθε thread, δηλαδή πως κάθε thread θα έχει το προσωπικό του αντίγραφο. Η επιλογή αυτή έγινε με τη βάση πως οι μεταβλητές είναι βοηθητικές μεταβλητές που χρησιμοποιούνται μόνο στο εσωτερικό του loop που παραλληλοποιούμε, χωρίς να μας ενδιαφέρουν οι προηγούμενες καταστάσεις τους. Αμα μοιράζονταν ανάμεσα σε processes, πιθανόν να είχαμε race conditions και θα σπαταλήσουμε χρόνο διότι πολλά threads θα επιχειρούσαν να κάνουν access μια συγκεκριμένη θέση μνήμης. Οι υπόλοιπες μεταβλητές είναι shared (δεν χρειάζεται να το σημάνουμε στο compiler directive, μιας και όσες δεν είναι private γίνονται shared by default) δεν υπάρχει λόγος να είναι ιδιωτικές σε κάθε thread, μιας και γίνονται μονο reads σε αυτές και αφορούν την προηγούμενη κατάσταση.

Ακολουθεί ο κώδικας του *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 <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define FINALIZE "\
convert -delay 20 out*.pgm output.gif\n\
rm *pgm\n\
"
```

```c
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
    int tid;

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

    /*Read input arguments*/
    if ( argc != 3 ) {
        fprintf(stderr, "Usage: ./exec ArraySize TimeSteps\n");
        exit(-1);
    }

    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 schedule(static) private(nbrs, j)
        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 Threads %d\n",
        N, T, time, atoi(getenv("OMP_NUM_THREADS")));
    #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);
}
```

Εκτέλεση & Μετρήσεις Επίδοσης

Για να κάνουμε μετρήσεις χρειαζόμαστε να υποβάλουμε το πρόγραμμα μας προς εκτέλεση στο parlab queue του scirouter.

Χρησιμοποιούμε το ακόλουθο Makefile

```
all: Game_Of_Life

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

clean:
	rm Game_Of_Life
```

Για να τρέξουμε το make στο queue, χρησιμοποιούμε το αρχείο *make_on_queue.sh* με τα ακόλουθα περιεχόμενα:

```
#!/bin/bash

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

## Output and error files
#PBS -o make_conway_gameoflife.out
#PBS -e make_conway_gameoflife.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/parlab13/conway_gameoflife/
make clean
make
```

Τέλος, το ενδιαφέρον κομμάτι βρίσκεται στην εκτέλεση του προγράμματος. Υλοποιούμε ένα bash script το οποίο εκτελεί το Game of Life για όλους τους συνδυασμούς από αριθμούς threads και μεγέθη ταμπλό που μας έχουν ζητηθεί. Ακολουθεί η υλοποίηση του *run_on_queue.sh*

```
#!/bin/bash

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

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

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

##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/parlab13/conway_gameoflife

nthreads=( 1 2 4 6 8 )
sizes=( 64 1024 4096 )

for nthread in "${nthreads[@]}";
do
	for size in "${sizes[@]}";
	do
		export OMP_NUM_THREADS=${nthread};
		./Game_Of_Life ${size} 1000;
	done
done
```

Αφού το script τρέξει επιτυχώς το *run_conway_gameoflife.out* περιέχει τις μετρήσεις μας, σε μορφή:

GameOfLife: Size <array_size> Steps 1000 Time <elapsed_time> Threads <thread_num>

Ο αριθμός των threads που χρησιμοποιήθηκαν στο output είναι δική μας προσθήκη, και έγινε με την χρήση *getenv(“OMP_NUM_THREADS”)* στον κώδικα.

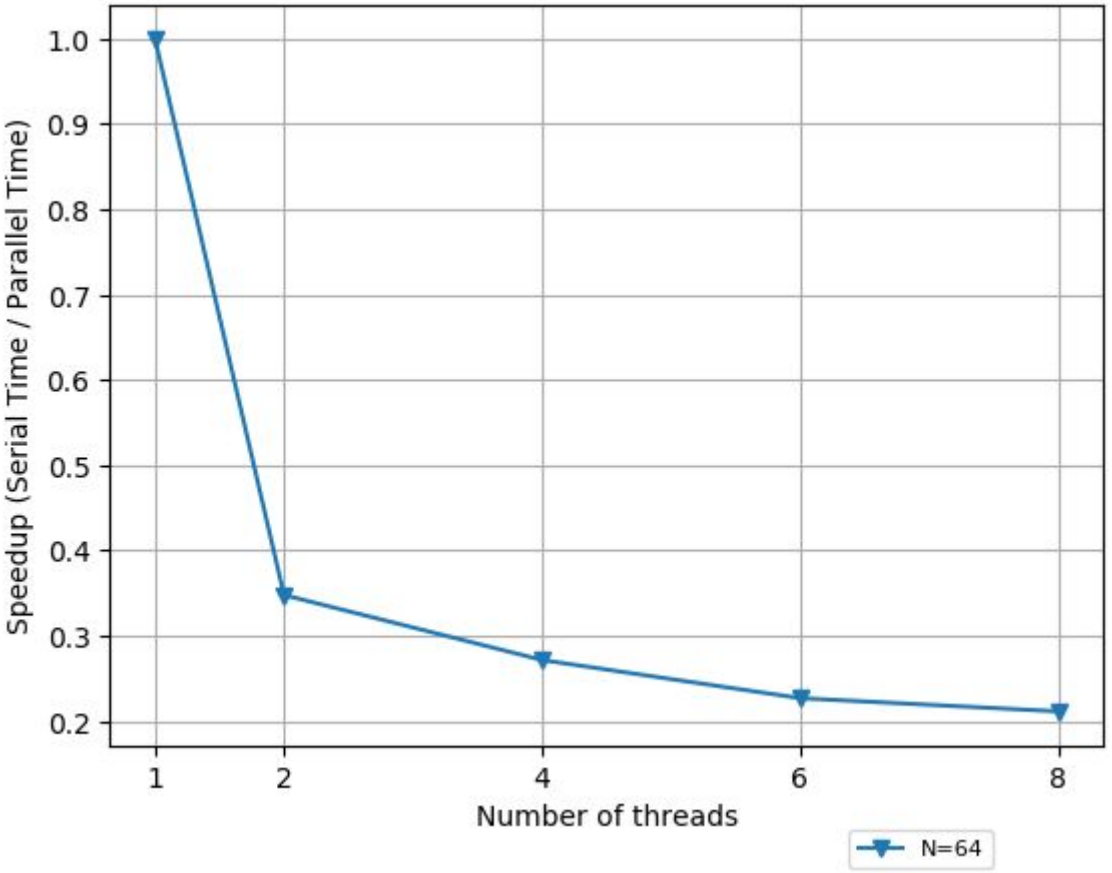
Επίδειξη Μετρήσεων

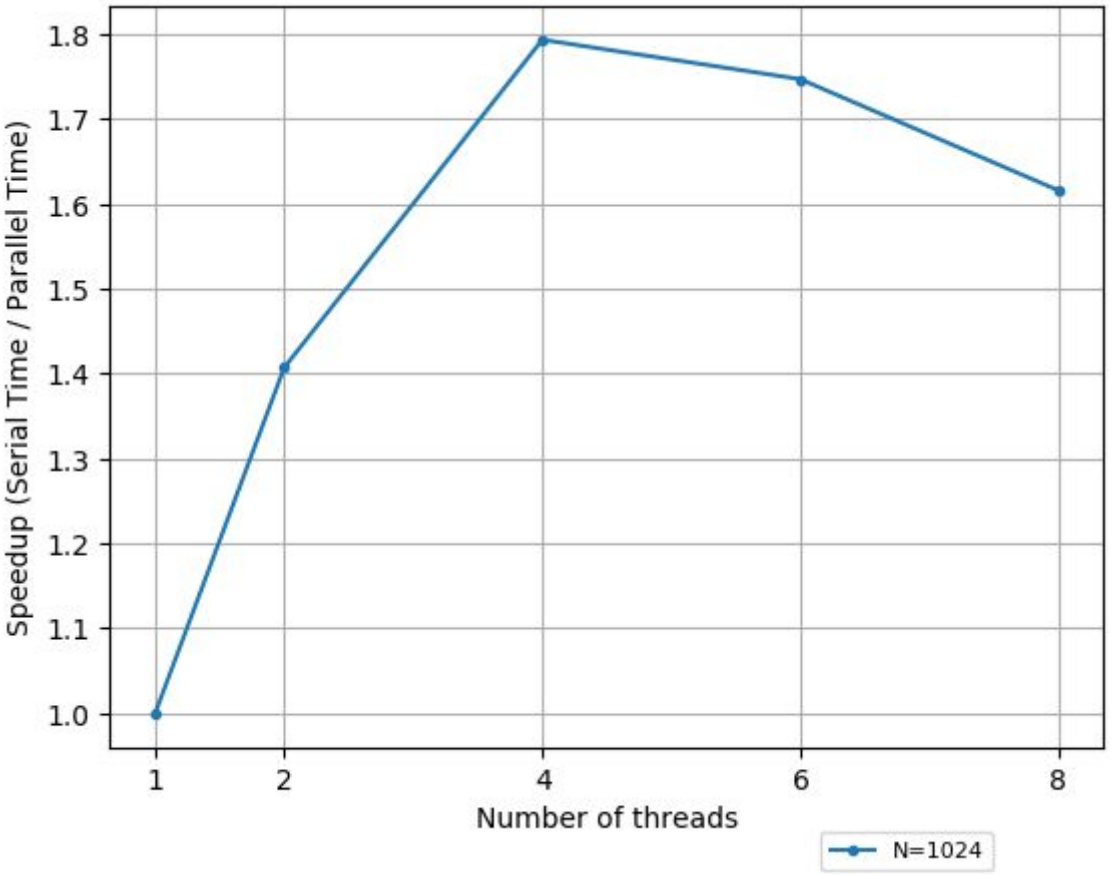
Για να δημιουργησουμε γραφήματα, υλοποιήσαμε το ακόλουθο Python πρόγραμμα:

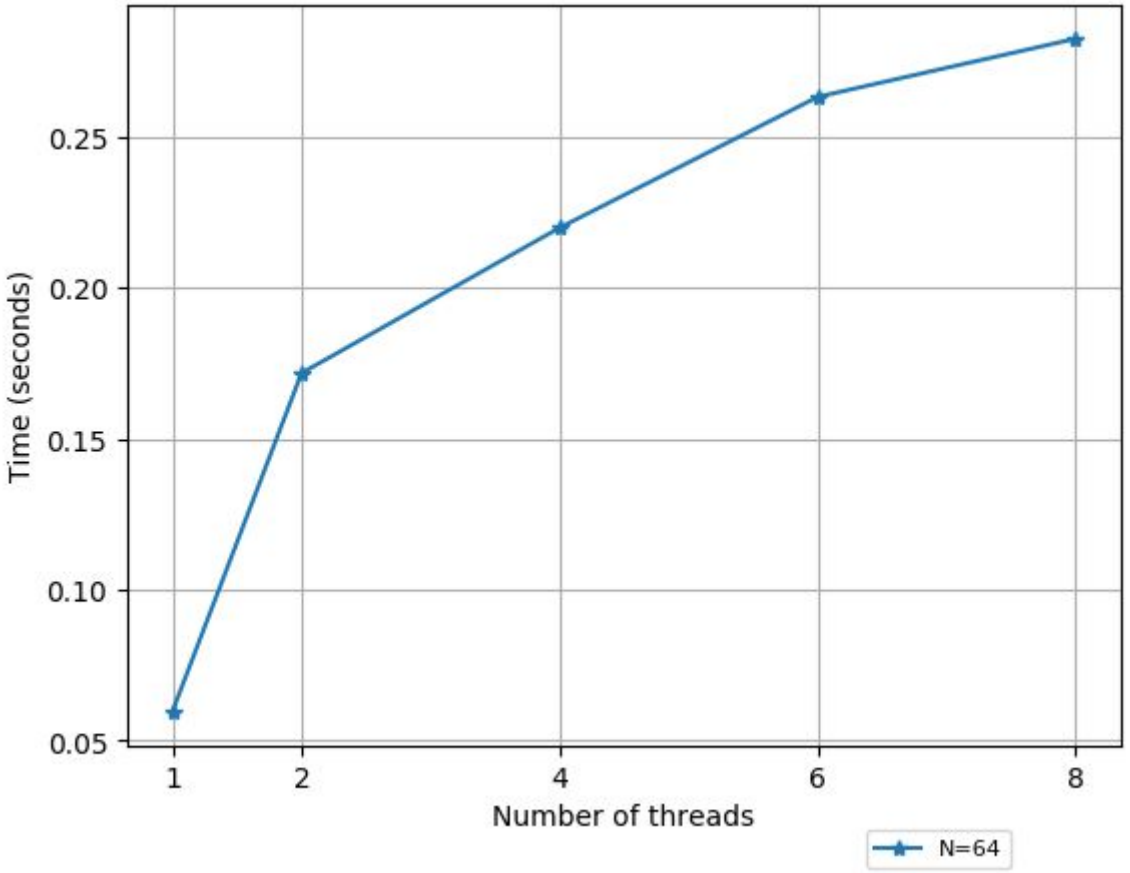
```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import sys

def parse_file(fname):
	with open(fname) as f:
		lines = f.readlines()

	size_stats = {}
	for line in lines:
		# line of the form:
		# GameOfLife: Size <array_size> Steps 1000 Time <elapsed_time>
Threads <thread_num>
		splitted = line.split(" ")
		# size of array
		array_size = splitted[2]
		# elapsed time
		elapsed_time = splitted[6]
		# thread number
		thread_num = splitted[8]
		if not size_stats.get(array_size, None):
			size_stats[array_size] = []
		size_stats[array_size].append({"elapsed": elapsed_time, "nthread":
thread_num})
	return size_stats

if len(sys.argv) < 2:
	print ("Usage parse_stats.py <input_file>")
	exit(-1)

stats_by_size = parse_file(sys.argv[1])

markers = set(['.', 'o', 'v', '*', 'D', 'X'])

x_ticks = [1, 2, 4, 6, 8]
serial_time = {}
i = 0
for size, stats in stats_by_size.items():
	i += 1
	fig = plt.figure(i)
```

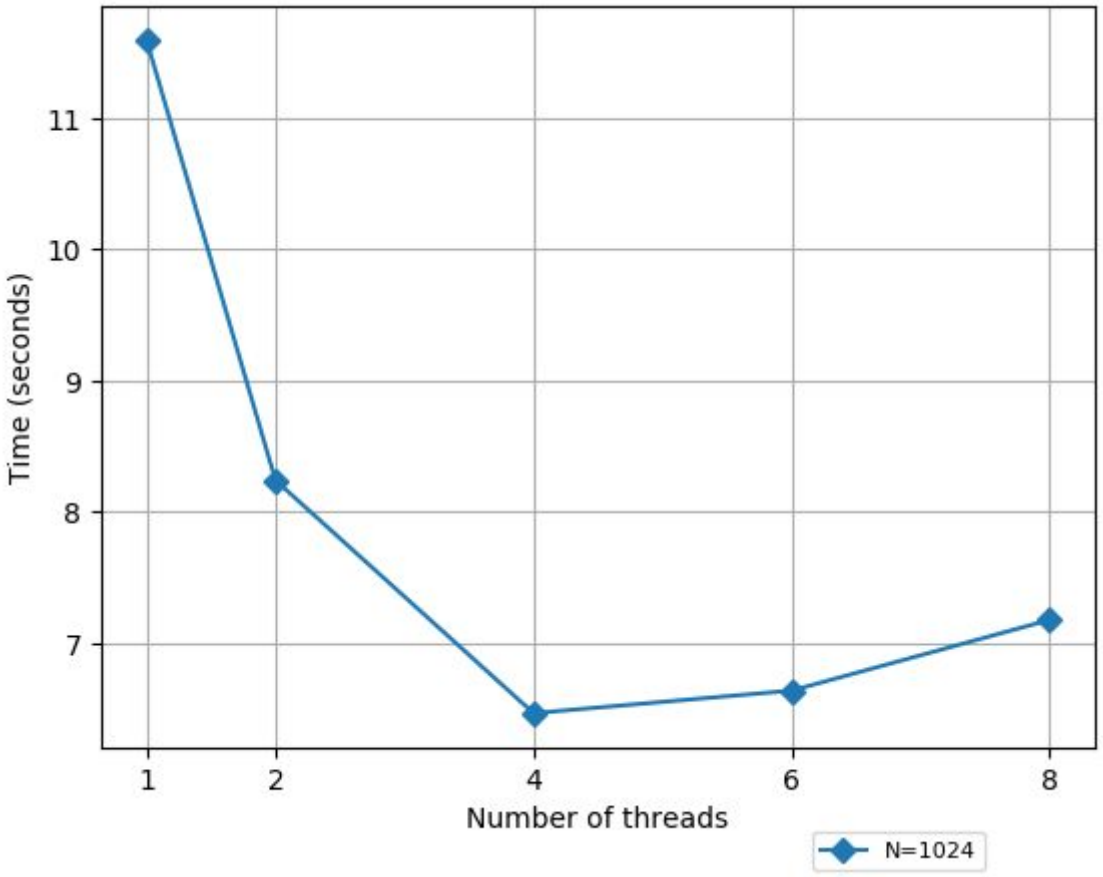
```
	plt.grid(True)
	ax = plt.subplot(111)
	ax.set_xlabel("Number of threads")
	ax.set_ylabel("Time (seconds)")
	ax.xaxis.set_ticks(x_ticks)
	ax.xaxis.set_ticklabels(map(str, x_ticks))
	y_axis = [0 for _ in range(len(x_ticks))]
	for stat in stats:
	pos = x_ticks.index(int(stat["nthread"]))
	if int(stat["nthread"]) == 1:
		serial_time[size] = float(stat["elapsed"])
	y_axis[pos] = float(stat["elapsed"])

	ax.plot(x_ticks, tuple(y_axis), label="N="+size, marker=markers.pop())
	lgd = ax.legend(ncol=len(stats_by_size.keys()), bbox_to_anchor=(0.9,
-0.1), prop={'size':8})
	plt.savefig("stats-time-" + size + ".png", bbox_extra_artists=(lgd,),
bbox_inches='tight')

for size, stats in stats_by_size.items():
	i += 1
	fig = plt.figure(i)
	plt.grid(True)
	ax = plt.subplot(111)
	ax.set_xlabel("Number of threads")
	ax.set_ylabel("Speedup (Serial Time / Parallel Time)")
	ax.xaxis.set_ticks(x_ticks)
	ax.xaxis.set_ticklabels(map(str, x_ticks))
	y_axis = [0 for _ in range(len(x_ticks))]
	for stat in stats:
	pos = x_ticks.index(int(stat["nthread"]))
	y_axis[pos] = serial_time[size] / float(stat["elapsed"])

	ax.plot(x_ticks, tuple(y_axis), label="N="+size, marker=markers.pop())
	lgd = ax.legend(ncol=len(stats_by_size.keys()), bbox_to_anchor=(0.9,
-0.1), prop={'size':8})
	plt.savefig("stats-speedup-" + size + ".png", bbox_extra_artists=(lgd,),
bbox_inches='tight')
```

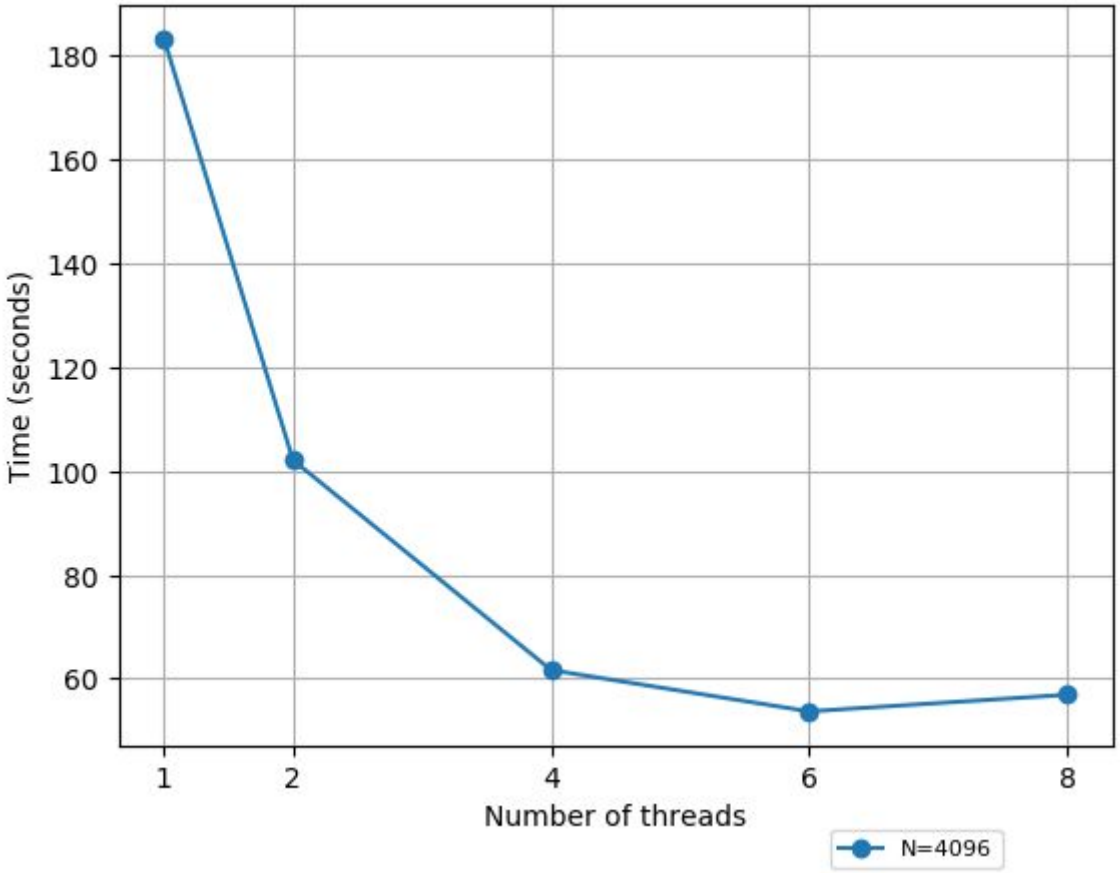
Ακολουθούν γραφήματα που δείχνουν το speedup και τον χρόνο εκτέλεσης για τα διάφορα μεγέθη ταμπλό.

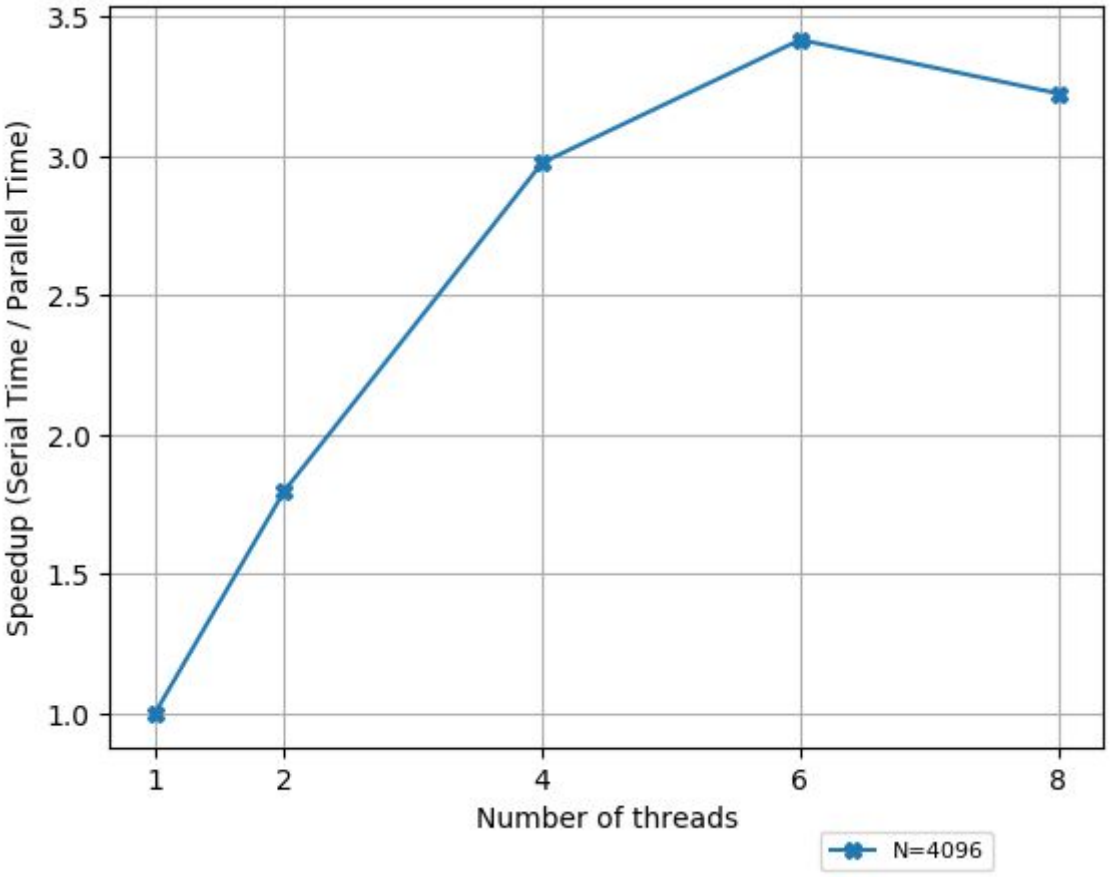

Speedup (Serial Time / Parallel Time)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
1
2
4
6
8
Number of threads
N=64


Speedup (Serial Time / Parallel Time)
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
1
2
4
6
8
Number of threads
N=1024


0.25
0.20
0.15
0.10
0.05
Time (seconds)
1
2
4
6
8
Number of threads
N=64


11
10
9
8
7
Time (seconds)
1
2
4
6
8
Number of threads
N=1024


Time (seconds)
180
160
140
120
100
80
60
1
2
4
6
8
Number of threads
N=4096


Speedup (Serial Time / Parallel Time)
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
6
8
Number of threads
N=4096

Συμπεράσματα

Για μέγεθος ταμπλό N=64 βλέπουμε πως η παραλληλοποίηση έχει μικρή επίδραση. Αυτό οφείλεται στο ότι για μικρά μεγέθη, το overhead της παραλληλοποίησης (δημιουργία, συντονισμός threads) είναι αντικρούει το κέρδος που δίνει. Συνεπώς, για μικρά μεγέθη ταμπλό ο σειριακός αλγόριθμος υπερτερεί μιας και είναι πιο απλός και χρησιμοποιεί λιγότερους πόρους.

Για μέγεθος ταμπλό N=1024 βλέπουμε πως δεν έχουμε πολύ καλό scaling για μεγάλο αριθμό threads. Το βέλτιστο speedup είναι αρκετά μακριά από το ideal (4 vs 1.8). Η καθυστέρηση για μεγάλο αριθμό threads μπορεί να οφείλεται στην συμφόρηση του διαδρόμου κοινής μνήμης.

Για μέγεθος ταμπλό N=4096 πάλι δεν έχουμε πολύ καλό scaling για μεγάλο αριθμό threads. Το βέλτιστο speedup είναι μακριά από το ideal (6 vs 3.8). Οι λόγοι καθυστέρησης είναι οι ίδιοι για το N=1024 σε μεγάλο αριθμό threads.

Θα μπορούσαμε να βελτιώσουμε το speedup μέσω υλοποίησης. Συγκεκριμένα, θα μπορούσαμε να μοιράσουμε την δομή του πίνακα σε δύο διαστάσεις αντι για μια. Επιλέξαμε τον διαμοιρασμό σε μια διάσταση, δηλαδή να παραλληλοποιήσουμε μόνο τις προσβάσεις στις γραμμές του πίνακα, διοτί οδηγεί σε πολύ πιο απλή υλοποίηση.

Θα ήταν πολύ πιο εύκολο να μοιράσουμε την δομή του πίνακα σε δύο διαστάσεις (blocks) αν είχαμε στην διάθεση μας την εντολή *collapse(n)* του OpenMP η οποία επιτρέπει την παραλληλοποίηση εμφωλευμένων loops, αλλά παρατηρήσαμε πως το scirouter έχει έκδοση του gcc όπου δεν υποστηρίζεται αυτή η εντολή.

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

Εισαγωγή

Σε αυτό το ζήτημα καλούμαστε να παραλληλοποιήσουμε τρεις διαφορετικές εκδόσεις του αλγορίθμου Floyd-Warshall σε αρχιτεκτονικές κοινής μνήμης.

Μας δίνεται σειριακός κώδικας για τις ακόλουθες εκδόσεις:

1. Standard έκδοση
2. Recursive έκδοση
3. Tiled έκδοση

Για τις ανάγκες της ενδιάμεσης αναφοράς θα μελετήσουμε τον παραλληλισμό κάθε έκδοσης και θα σχεδιάσουμε την παράλληλη υλοποίηση.

Standard Floyd-Warshall

Θα θεωρήσουμε το core κομμάτι του κώδικα που μας δόθηκε:

```
for(k=0;k<N;k++)
     for(i=0; i<N; i++)
          for(j=0; j<N; j++)
               A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
```

Η παραλληλοποίηση είναι εμφανής. Η εκτελέσεις των εμφωλευμένων loops i και j είναι ανεξάρτητες μεταξύ τους και συνεπώς μπορούν να παραλληλοποιηθούν. Το εξωτερικό loop k δεν μπορεί να

παραλληλοποιηθεί, μιας και στον αλγόριθμο Floyd-Warshall πρέπει πρώτα να γίνουν update οι αποστάσεις για τον κόμβο που έχουμε επιλέξει προτού επιλέξουμε τον επόμενο κόμβο.

Συνεπώς, ο παραλληλοποιημένος κώδικας είναι:

```
for(k=0;k<N;k++)
     #pragma omp parallel for private(j)
     for(i=0; i<N; i++)
           for(j=0; j<N; j++)
                 A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
```

Recursive Floyd-Warshall

Το core κομμάτι της αναδρομικής έκδοσης του προγράμματος είναι:

```
FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2,
bsize);
FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
ccol+myN/2, myN/2, bsize);
FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2,
bsize);
FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2,
bsize);
FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
```

Παρατηρούμε ότι οι υπολογισμοί στους αντιδιαμετρικούς υποπίνακες A_{12} και A_{21} μπορούν να παραλληλοποιηθούν, καθώς είναι ανεξάρτητοι μεταξύ τους. Οι υπολογισμοί για τον υποπίνακα A_{11} πρέπει να προηγηθούν αυτών, ενώ οι υπολογισμοί για τον υποπίνακα A_{22} έπονται αυτών.

Συνεπώς, ο παραλληλοποιημένος κώδικας είναι:

```
#pragma omp parallel
{
        #pragma omp single
        {
                FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
                #pragma omp task
                        FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow,
ccol+myN/2, myN/2, bsize);
                #pragma omp task
                        FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow,
ccol, myN/2, bsize);
                #pragma omp taskwait
                FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow,
ccol+myN/2, myN/2, bsize);

                FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2,
bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
                #pragma omp task
                        FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
                #pragma omp task
                        FW_SR(A,arow, acol+myN/2,B,brow,
bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
                #pragma omp taskwait
                FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2,
bsize);
        }
}
```

Tiled Floyd-Warshall

Εδώ το κομμάτι που επικεντρωνόμαστε είναι:

```
for(k=0;k<N;k+=B){
        FW(A,k,k,k,B);

        for(i=0; i<k; i+=B)
           FW(A,k,i,k,B);

        for(i=k+B; i<N; i+=B)
           FW(A,k,i,k,B);

        for(j=0; j<k; j+=B)
```

```
            FW(A,k,k,j,B);

        for(j=k+B; j<N; j+=B)
            FW(A,k,k,j,B);

        for(i=0; i<k; i+=B)
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        for(i=0; i<k; i+=B)
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);

        for(i=k+B; i<N; i+=B)
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        for(i=k+B; i<N; i+=B)
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);
    }
```

Η προφανής σκέψη είναι η χρήση parallel for, μόνο που εδώ διαπιστώνουμε ότι ο φόρτος των for δεν είναι ισοκατανεμημένος, όπως στην κλασική εκδοχή του Floyd-Warshall. Άρα κάποια threads θα έχουν λιγότερη δουλειά από κάποια άλλα, έχοντας ως αποτέλεσμα stalls και μη αποτελεσματική χρήση των threads.
Έτσι καταλήγουμε και πάλι στην χρήση tasks, πρώτα για τα N, S, E, W tiles και έπειτα για όλα τα εναπομείναντα tiles(ενδιάμεσα μεσολαβεί synchronization).

Εν τέλει, ο παραλληλοποιημένος κώδικας είναι:

```
for(k=0;k<N;k+=B){
    #pragma omp parallel
    {
        #pragma omp single
        {
            FW(A,k,k,k,B);

            for(i=0; i<k; i+=B) {
```

```
        #pragma omp task
            FW(A,k,i,k,B);
    }

    for(i=k+B; i<N; i+=B) {
        #pragma omp task
            FW(A,k,i,k,B);
    }

    for(j=0; j<k; j+=B) {
        #pragma omp task
            FW(A,k,k,j,B);
    }

    for(j=k+B; j<N; j+=B) {
        #pragma omp task
            FW(A,k,k,j,B);
    }

    #pragma omp taskwait

    for(i=0; i<k; i+=B)
        for(j=0; j<k; j+=B) {
            #pragma omp task firstprivate(j)
                FW(A,k,i,j,B);
        }

    for(i=0; i<k; i+=B)
        for(j=k+B; j<N; j+=B) {
            #pragma omp task firstprivate(j)
                FW(A,k,i,j,B);
        }

    for(i=k+B; i<N; i+=B)
        for(j=0; j<k; j+=B) {
            #pragma omp task firstprivate(j)
                FW(A,k,i,j,B);
        }

    for(i=k+B; i<N; i+=B)
        for(j=k+B; j<N; j+=B) {
            #pragma omp task firstprivate(j)
                FW(A,k,i,j,B);
        }
```

```
            #pragma omp taskwait
        }
    }
}
```

Εκτέλεση & Αρχικές Μετρήσεις Επίδοσης

Χρησιμοποιούμε το ακόλουθο Makefile:

```
all: fw fw_sr fw_tiled

CC=gcc
CFLAGS= -Wall -O3 -Wno-unused-variable -fopenmp

HDEPS+=%.h

OBJS=util.o

fw: $(OBJS) fw.c
    $(CC) $(OBJS) fw.c -o fw $(CFLAGS)
fw_sr: fw_sr.c
    $(CC) $(OBJS) fw_sr.c -o fw_sr $(CFLAGS)
fw_tiled: fw_tiled.c
    $(CC) $(OBJS) fw_tiled.c -o fw_tiled $(CFLAGS)

%.o: %.c $(HDEPS)
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f *.o fw fw_sr fw_tiled
```

Για να τρέξουμε το make στο queue, χρησιμοποιούμε το αρχείο *make_on_queue.sh* με τα ακόλουθα περιεχόμενα:

```
#!/bin/bash

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

## Output and error files
#PBS -o make_FW.out
```

```
#PBS -e make_FW.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/parlab13/fillm/FW-serial
make
```

Τέλος, ακολουθούν οι υλοποιήσεις των 3 script που υποβλήθηκαν στην ουρά για τους Standard FW(run_std.sh), Recursive FW(run_rec.sh) και Tiled FW(run_tiled.sh), με αυτή τη σειρά.

```
#!/bin/bash

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

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

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

##How long should the job run for?
#PBS -l walltime=00:25:00

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

module load openmp
cd /home/parallel/parlab13/FW-serial

nthreads = ( 1 2 4 8 16 32 64 )
sizes = ( 1024 2048 4096 )
```

```
for nthread in “${nthreads[@]}”;
do
    for size in “${sizes[@]}”;
    do
        export OMP_NUM_THREADS=${nthread};
        ./fw ${size};
    done
done
```

Στο run_rec.sh, για κάθε συνδυασμό #threads και size τρέχουμε με blocksize 128, 64 και 32.

```
#!/bin/bash

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

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

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

##How long should the job run for?
#PBS -l walltime=00:25:00

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

module load openmp
cd /home/parallel/parlab13/FW-serial

nthreads = ( 1 2 4 8 16 32 64 )
sizes = ( 1024 2048 4096 )

for nthread in “${nthreads[@]}”;
do
    for size in “${sizes[@]}”;
    do
        export OMP_NUM_THREADS=${nthread};
        ./fw_sr ${size} 128;
        ./fw_sr ${size} 64;
```

```
        ./fw_sr ${size} 32;
    done
done
```

Στο run_tiled.sh, για κάθε συνδυασμό #threads και size τρέχουμε με blocksize 256, 128 και 64.

```
#!/bin/bash

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

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

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

##How long should the job run for?
#PBS -l walltime=00:25:00

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

module load openmp
cd /home/parallel/parlab13/FW-serial

nthreads = ( 1 2 4 8 16 32 64 )
sizes = ( 1024 2048 4096 )

for nthread in “${nthreads[@]}”;
do
    for size in “${sizes[@]}”;
    do
        export OMP_NUM_THREADS=${nthread};
        ./fw_tiled ${size} 128;
        ./fw_tiled ${size} 64;
        ./fw_tiled ${size} 32;
    done
```

```
done
```

Αφού τα script τρέξουν, οι έξοδοί τους βρίσκονται στα αρχεία run_std.out, run_rec.out και run_tiled.out αντίστοιχα, στις παρακάτω μορφές:

FW, <array_size>, <elapsed_time>, <thread_num>

FW_SR, <array_size>, <block_size>, <elapsed_time>, <thread_num>

FW_TILED, <array_size>, <block_size>, <elapsed_time>, <thread_num>

Ο αριθμός των threads που χρησιμοποιήθηκαν στο output έγινε και πάλι με την χρήση *getenv("OMP_NUM_THREADS")* στον κώδικα.

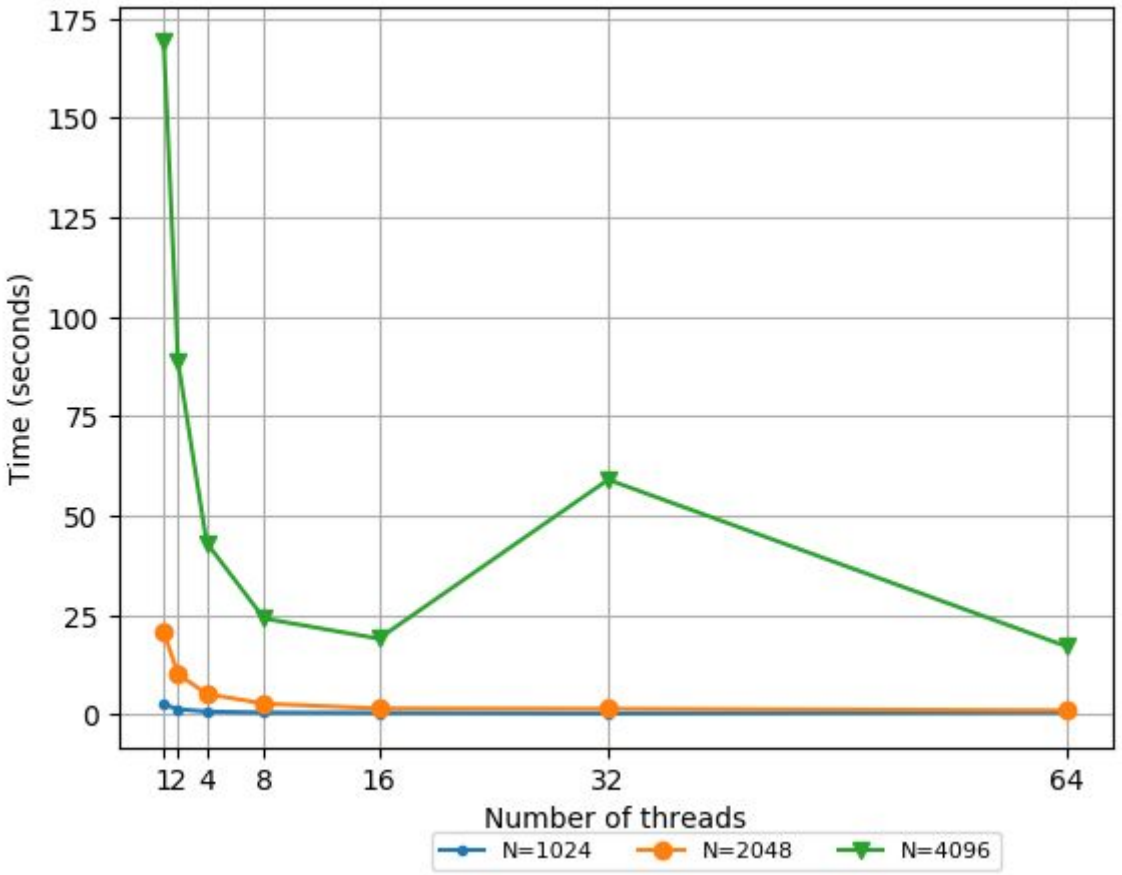
Επίδειξη Αρχικών Μετρήσεων

Για να δημιουργησουμε τα standard γραφήματα, υλοποιήσαμε το ακόλουθο Python πρόγραμμα:

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import sys

def parse_file(fname):
        with open(fname) as f:
                lines = f.readlines()

        size_stats = {}
        for line in lines:
                # line of the form:
                # FW_SR,<size>,time,processes
                splitted = line.split(",")
                # size of array
                array_size = splitted[1].strip()
                # elapsed time
                elapsed_time = splitted[2].strip()
                # thread number
```

```
                thread_num = splitted[3].strip()
                if not size_stats.get(array_size, None):
                    size_stats[array_size] = []

                size_stats[array_size].append({"elapsed": elapsed_time,
"nthread": thread_num})
        return size_stats

if len(sys.argv) < 2:
    print ("Usage parse_stats.py <input_file>")
    exit(-1)

stats_by_size = parse_file(sys.argv[1])
markers = ['.', 'o', 'v', '*', 'D', 'X']

x_ticks = [1, 2, 4, 8, 16, 32, 64]
fig = plt.figure(1)
plt.grid(True)
ax = plt.subplot(111)
ax.set_xlabel("Number of threads")
ax.set_ylabel("Time (seconds)")
ax.xaxis.set_ticks(x_ticks)
ax.xaxis.set_ticklabels(map(str, x_ticks))

for j, size in enumerate(sorted(stats_by_size.keys(), key=lambda x: int(x))):
    stats = stats_by_size[size]
    y_axis = [0 for _ in range(len(x_ticks))]
    for stat in stats:
        pos = x_ticks.index(int(stat["nthread"]))
        y_axis[pos] = float(stat["elapsed"])

    ax.plot(x_ticks, tuple(y_axis), label="N="+str(size), marker=markers[j])

lgd = ax.legend(ncol=len(stats_by_size.keys()), bbox_to_anchor=(0.9, -0.1),
prop={'size':8})
plt.savefig("classic-fw-init.png", bbox_extra_artists=(lgd,),
bbox_inches='tight')
```

Ακολουθεί το γράφημα που προέκυψε για τους χρόνους εκτέλεσης.



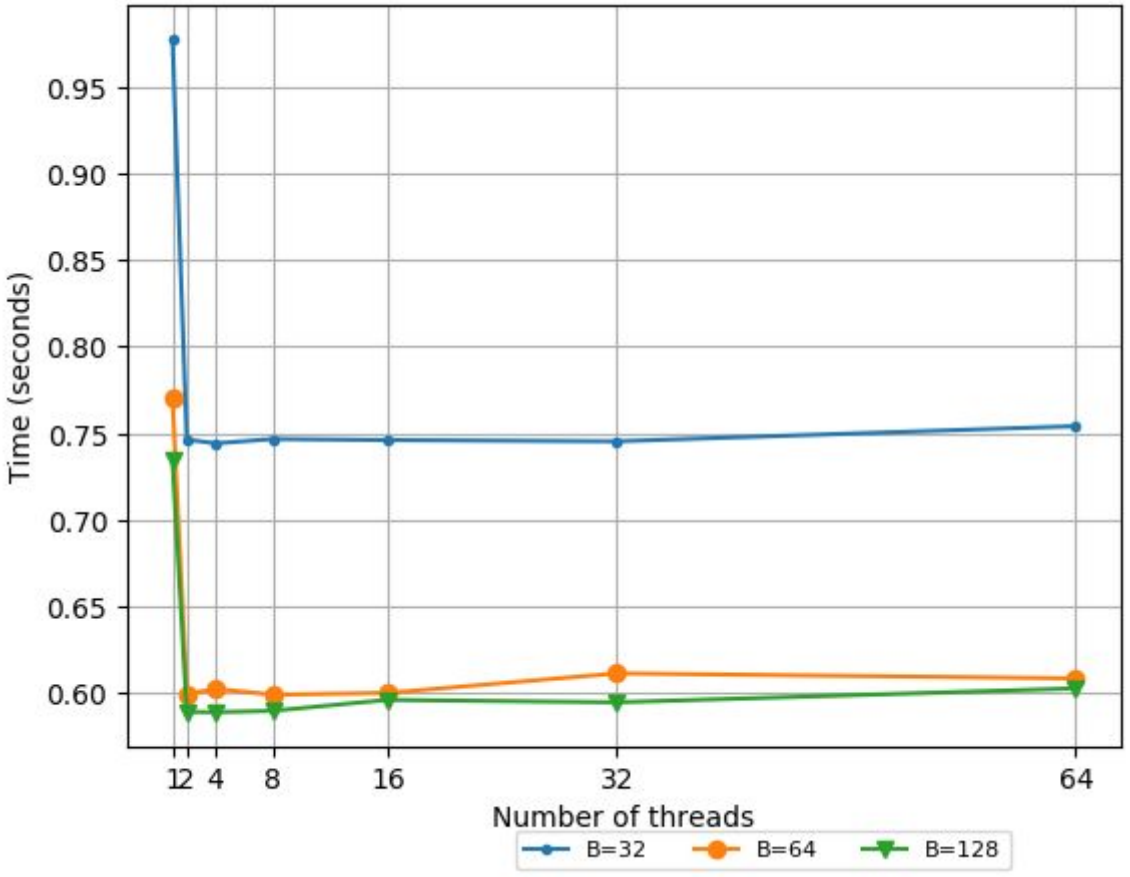
Για να δημιουργήσουμε τα recursive και tiled γραφήματα, υλοποιήσαμε το ακόλουθο Python πρόγραμμα:

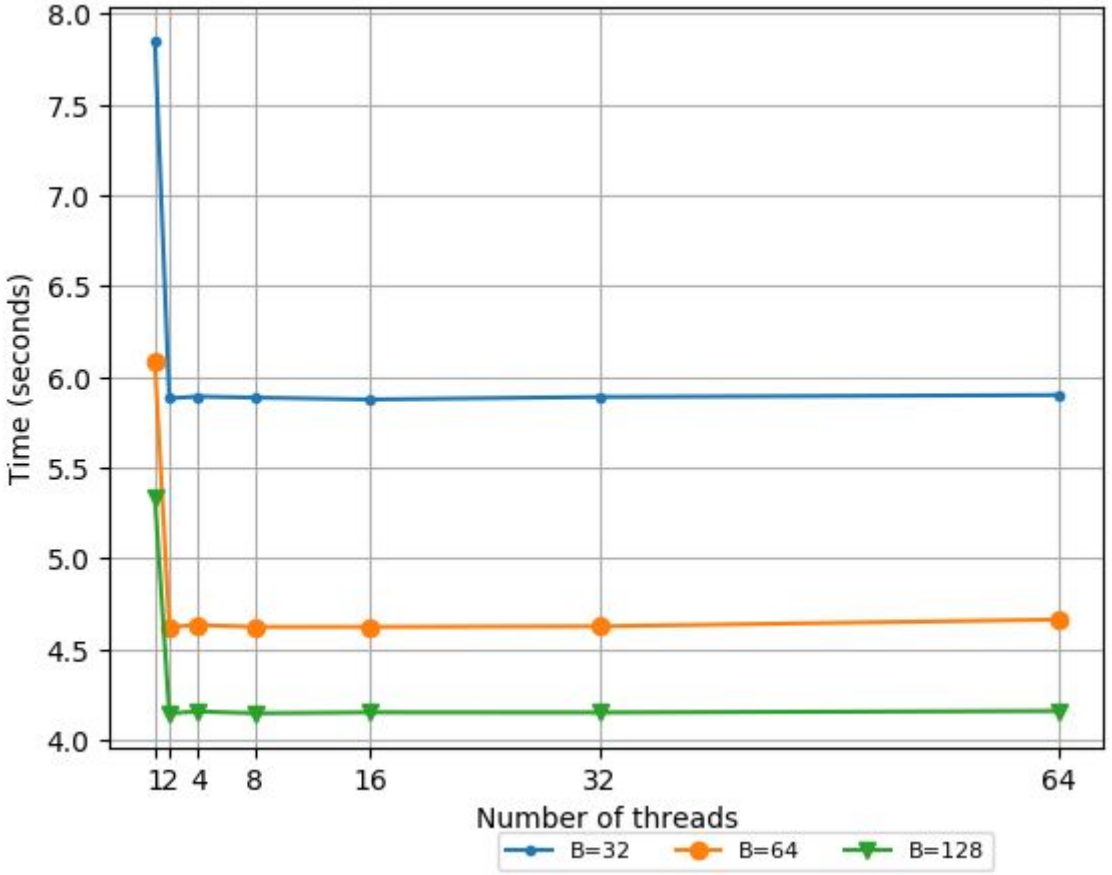
```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import sys

def parse_file(fname):
        with open(fname) as f:
                lines = f.readlines()

        size_stats = {}
        for line in lines:
                # line of the form:
                # FW_SR,<size>,<block_size>,time,processes
                splitted = line.split(",")
                # size of array
```

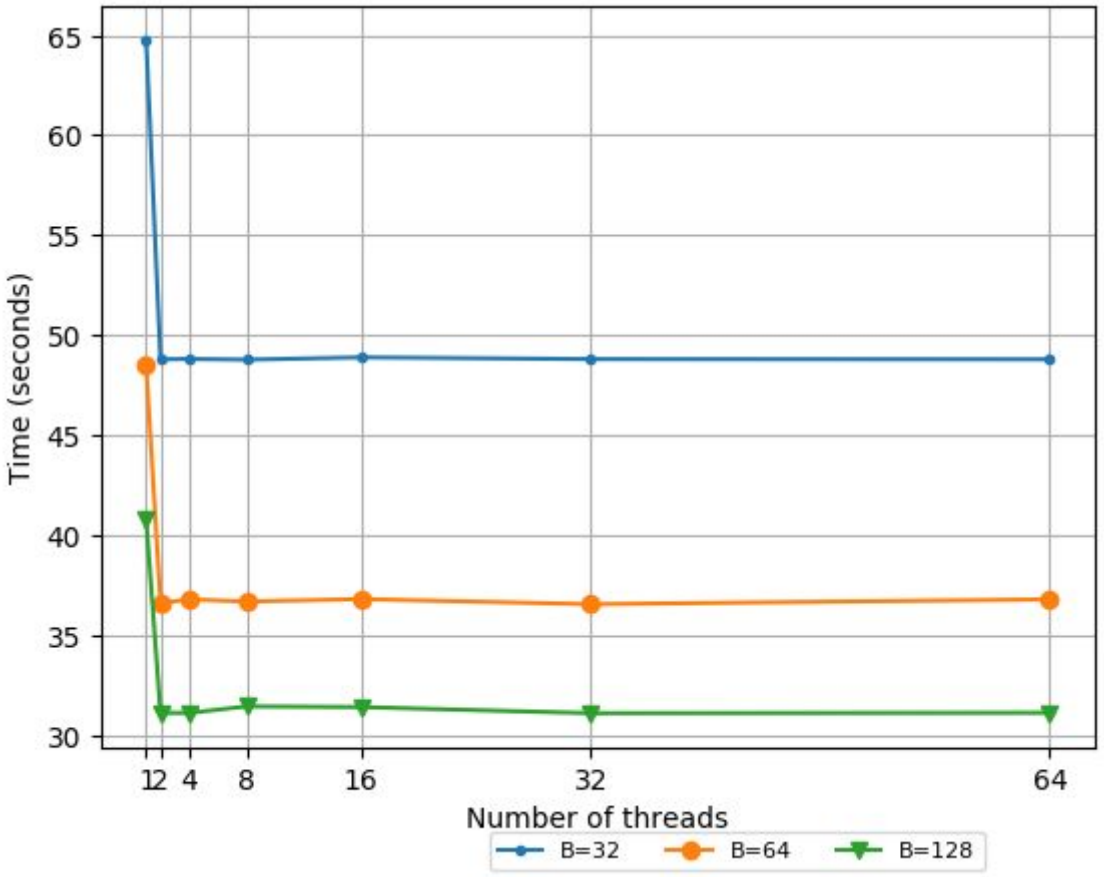
```
                array_size = splitted[1].strip()
                # block size
                block_size = splitted[2].strip()
                # elapsed time
                elapsed_time = splitted[3].strip()
                # thread number
                thread_num = splitted[4].strip()
                if not size_stats.get(array_size, None):
                    size_stats[array_size] = {}

                if not size_stats[array_size].get(block_size, None):
                    size_stats[array_size][block_size] = []

                size_stats[array_size][block_size].append({"elapsed":
elapsed_time, "nthread": thread_num})
        return size_stats

if len(sys.argv) < 2:
    print ("Usage parse_stats.py <input_file>")
    exit(-1)

stats_by_size = parse_file(sys.argv[1])
for size, size_stats in stats_by_size.items():
    print ("For size {}".format(size))
    for block_size, block_stats in size_stats.items():
        print ("Block {} with stats {}".format(block_size, block_stats))

markers = ['.', 'o', 'v', '*', 'D', 'X']

x_ticks = [1, 2, 4, 8, 16, 32, 64]
serial_time = {}
i = 0
for size, size_stats in stats_by_size.items():
    i += 1
    fig = plt.figure(i)
    plt.grid(True)
    ax = plt.subplot(111)
    ax.set_xlabel("Number of threads")
    ax.set_ylabel("Time (seconds)")
    ax.xaxis.set_ticks(x_ticks)
    ax.xaxis.set_ticklabels(map(str, x_ticks))

    for j, block_size in enumerate(sorted(size_stats.keys(), key=lambda x:
int(x))):
        block_stats = size_stats[block_size]
        y_axis = [0 for _ in range(len(x_ticks))]
        for stat in block_stats:
            pos = x_ticks.index(int(stat["nthread"]))
            #if int(stat["nthread"]) == 1:
            #    serial_time[size] = float(stat["elapsed"])
```

```
            y_axis[pos] = float(stat["elapsed"])

        ax.plot(x_ticks, tuple(y_axis), label="B="+str(block_size),
marker=markers[j])

    lgd = ax.legend(ncol=len(stats_by_size.keys()), bbox_to_anchor=(0.9, -0.1),
prop={'size':8})
    plt.savefig("tiled-fw-size-" + str(size) + "-init.png",
bbox_extra_artists=(lgd,), bbox_inches='tight')
```

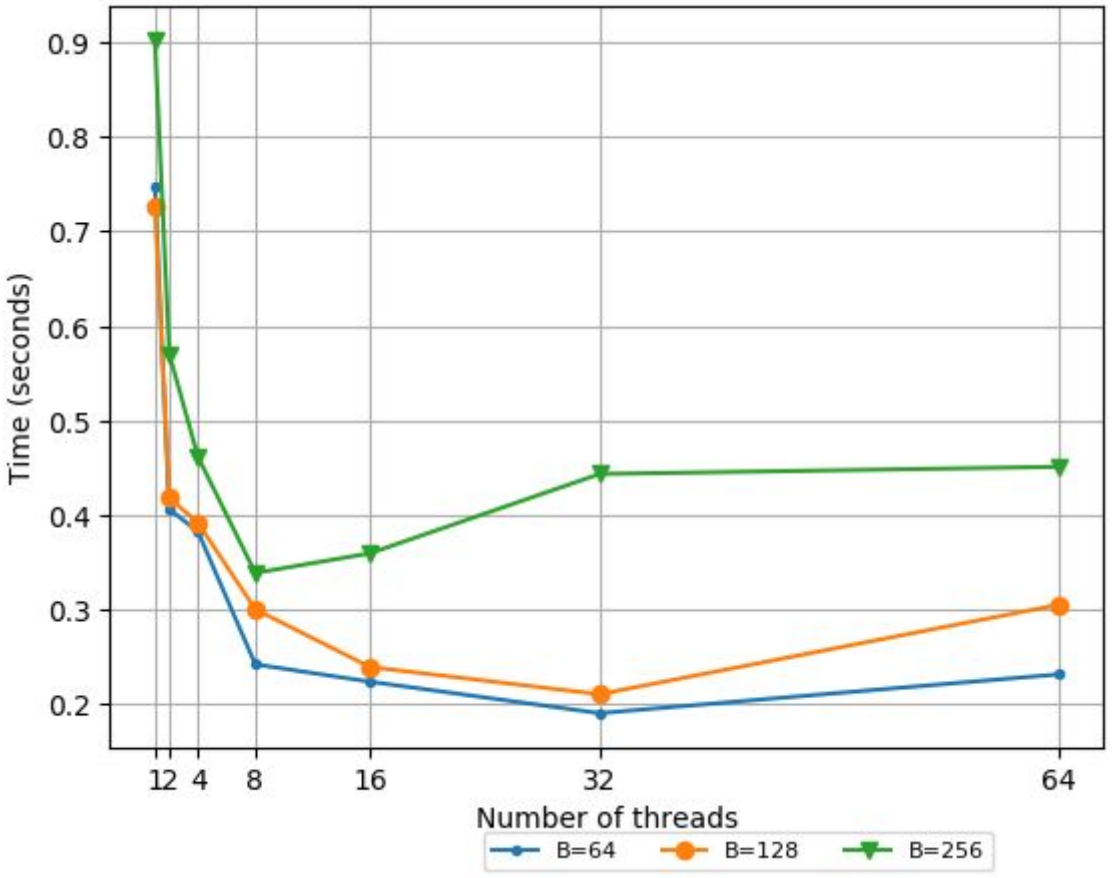
Δίνουμε πρώτα τα γραφήματα που προέκυψαν για τους διάφορους χρόνους εκτέλεσης του recursive FW, για μεγέθη ταμπλο 1024x1024, 2048x2048 και 4096x4096 αντίστοιχα.

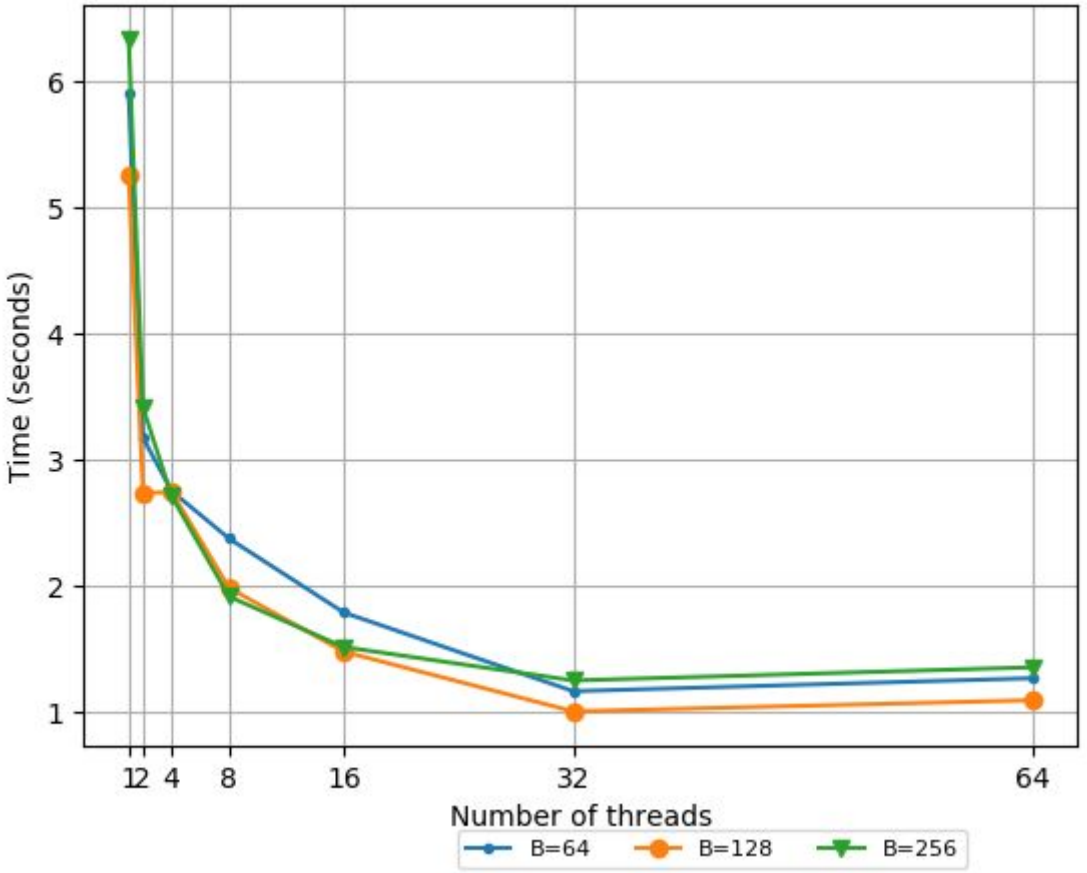


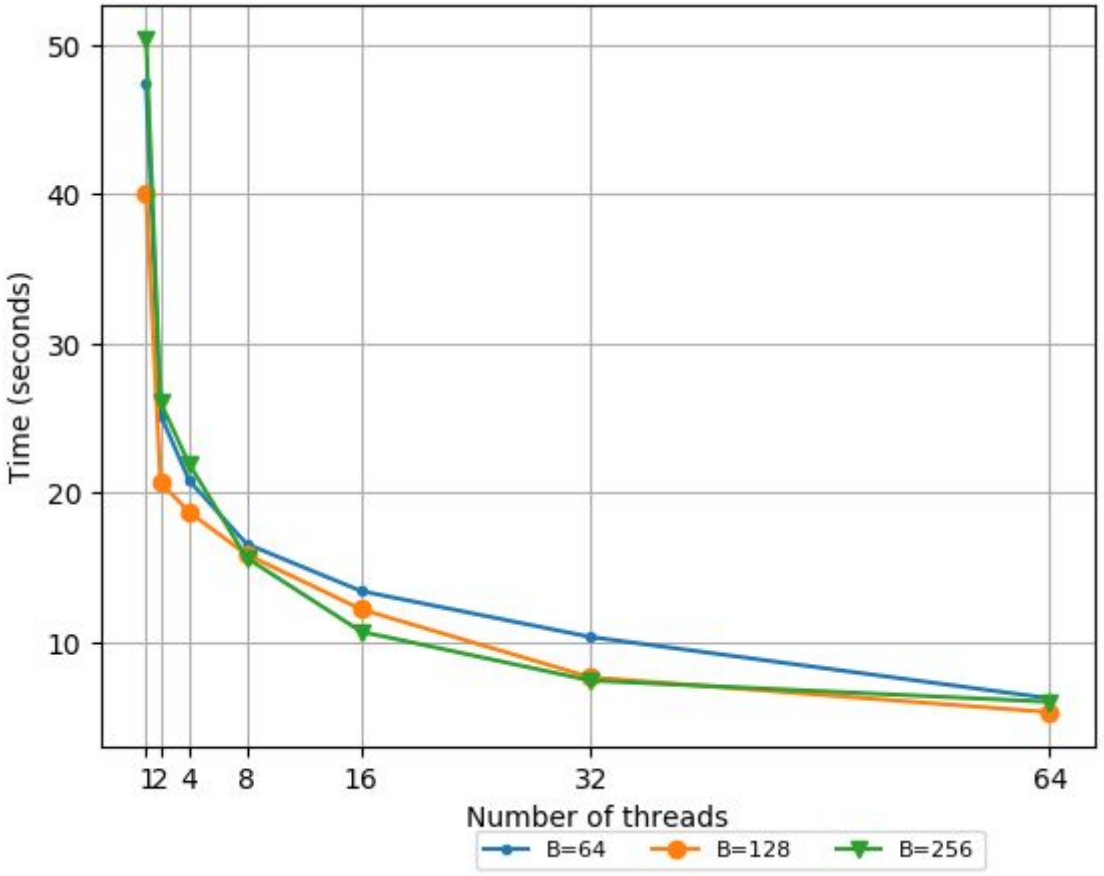

8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
Time (seconds)
1
2
4
8
16
32
64
Number of threads
B=32
B=64
B=128


65
60
55
50
45
40
35
30
Time (seconds)
1
2
4
8
16
32
64
Number of threads
B=32
B=64
B=128

Έπειτα, δίνουμε τα γραφήματα που προέκυψαν για τους διάφορους χρόνους εκτέλεσης του tiled FW, για μεγέθη ταμπλο 1024x1024, 2048x2048 και 4096x4096 αντίστοιχα.




Time (seconds)
1
2
3
4
5
6
1
2
4
8
16
32
64
Number of threads
B=64
B=128
B=256


Time (seconds)
10
20
30
40
50
1
2
4
8
16
32
64
Number of threads
B=64
B=128
B=256

Σκέψεις - Παρατηρήσεις

Θα επικεντρωθούμε στα αποτελέσματα των μετρήσεων για το μεγάλο ταμπλό.
Όσον αφορά τον κλασσικό FW, η υλοποίησή μας κάνει scale μέχρι και τα 8 threads. Η μειωμένη αποδοτικότητα στα 16 threads ίσως οφείλεται στη συμφόρηση που δημιουργεί στο διάδρομο του Sandy Bridge η χρησιμοποίηση και των 8 cores του. Όσον αφορά την τραγική απόδοση των 32 και των 64 threads, πιθανότατα αποδίδεται(πέραν του overhead της παραλληλοποίησης) στο γεγονός ότι οι 4 RAM είναι NUMA, οπότε οι πιο απομακρυσμένοι επεξεργαστές έχουν περαιτέρω χρονική επιβάρυνση. Ιδιαίτερα για τα 64 threads, αξίζει να σημειωθεί ότι ¾ threads επιβαρύνονται από την πολιτική NUMA των RAM, ενώ μόνο το ¼ αυτών επωφελείται.
Για τον recursive FW, τα αποτελέσματά μας είναι απογοητευτικά, με το scale να σταματά μόλις μετα τα 2 threads. Το πιο αποδοτικό block size είναι τα 128B, αποτέλεσμα αναμενόμενο, αφού έτσι επιτυγχάνεται αυξημένο locality. Οι άσχημες επιδόσεις οφείλονται στην ανισοκατανομή του φόρτου εργασίας των tasks(κάποια θα είναι πιο χρονοβόρα από άλλα, εξουδετερώνοντας την θεωρητική υπεροχή των παραπάνω threads). Η αφαίρεση των εντολών *#pragma omp taskwait* θα έλυνε το πρόβλημα αλλά θα κατέλυε την ορθότητα της υλοποίησής μας. Ακόμη, πιθανότατα η ευθύνη να εντοπίζεται στο overhead των προσβάσεων στην “ουρά” των tasks.
Το καλύτερο scale μέχρι στιγμής καταγράφηκε στον tiled FW, ναι μεν αρκετά μακριά από το θεωρητικά αναμενόμενο speedup αλλά με συνεχή βελτίωση με την αύξηση των threads. Πιθανοί ανασταλτικοί παράγοντες αναλύθηκαν και στις προηγούμενες 2 περιπτώσεις. Υπεροχή στα 64 threads είχε η εκτέλεση με tile size 128B, μέγεθος που μάλλον ταίριαξε καλύτερα με τις ανάγκες των tasks(ούτε αχρείαστα μεγάλο, ούτε περιοριστικά μικρό).

Βελτιωμένος Tiled FW

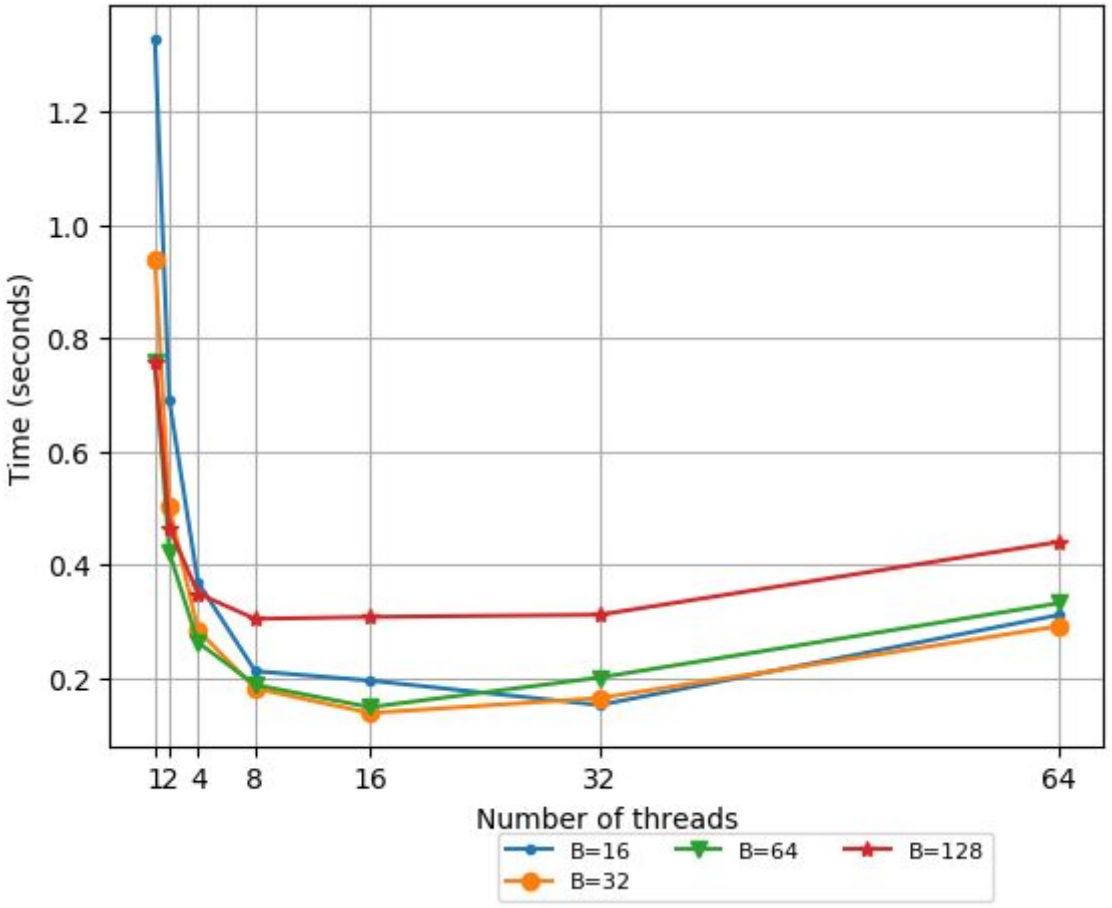
Παρά τις αρχικές μας αμφιβολίες, δοκιμάσαμε επίσης μια δεύτερη υλοποίηση με χρήση του parallel for. Παρακάτω δίνεται το ανανεωμένο core κομμάτι του κώδικα.

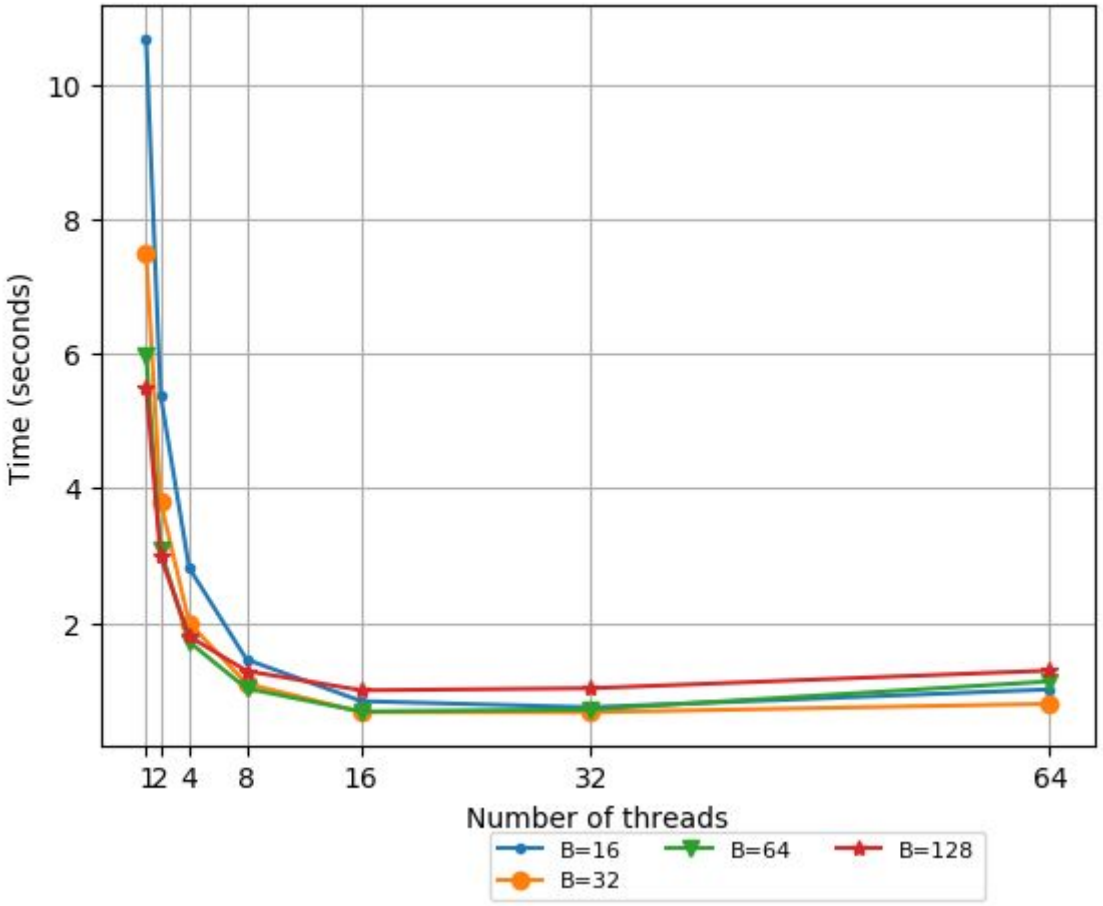
```
for(k=0;k<N;k+=B){
    FW(A,k,k,k,B);

    #pragma omp parallel for
    for(i=0; i<k; i+=B)
       FW(A,k,i,k,B);
    #pragma omp parallel for
    for(i=k+B; i<N; i+=B)
       FW(A,k,i,k,B);
    #pragma omp parallel for
    for(j=0; j<k; j+=B)
       FW(A,k,k,j,B);
    #pragma omp parallel for
    for(j=k+B; j<N; j+=B)
       FW(A,k,k,j,B);
    #pragma omp parallel for private(j)
    for(i=0; i<k; i+=B)
       for(j=0; j<k; j+=B)
          FW(A,k,i,j,B);
    #pragma omp parallel for private(j)
    for(i=0; i<k; i+=B)
       for(j=k+B; j<N; j+=B)
          FW(A,k,i,j,B);
    #pragma omp parallel for private(j)
    for(i=k+B; i<N; i+=B)
       for(j=0; j<k; j+=B)
          FW(A,k,i,j,B);
    #pragma omp parallel for private(j)
    for(i=k+B; i<N; i+=B)
       for(j=k+B; j<N; j+=B)
          FW(A,k,i,j,B);
 }
```

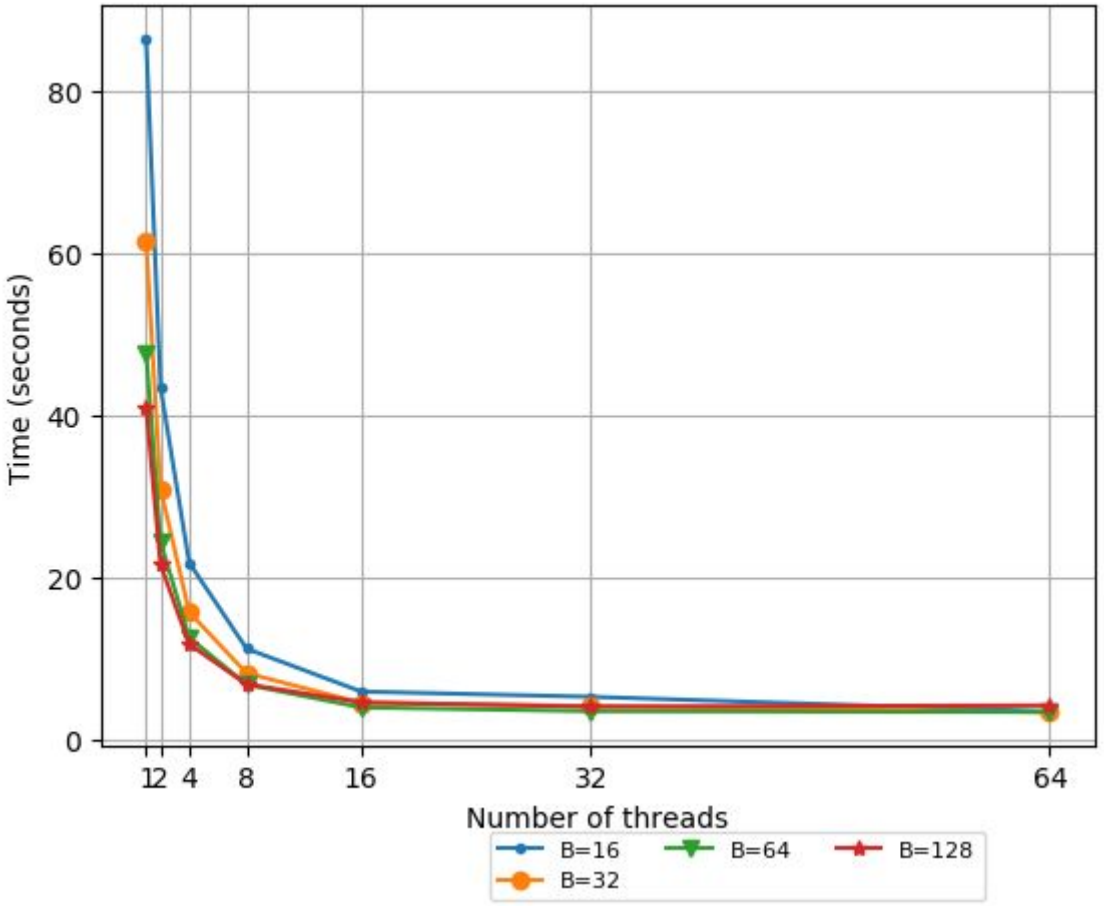
Με αμελητέες διαφορές κάναμε και πάλι make και τρέξαμε στην ουρά του sandman την υλοποίησή μας με tile sizes 16, 32, 64 και 128B.

Επίδειξη Τελικών Μετρήσεων

Με το ίδιο Python πρόγραμμα που χρησιμοποιήθηκε και προηγουμένως, εξάγαμε τα εξής γραφήματα για μεγέθη ταμπλό 1024x1024, 2048x2048 και 4096x4096 αντίστοιχα.




Time (seconds)
10
8
6
4
2
1
2
4
8
16
32
64
Number of threads
B=16
B=32
B=64
B=128



Τελικά Συμπεράσματα

Η χρήση των parallel for βελτίωσε σημαντικά τις επιδόσεις μας. Συγκεκριμένα για tile size 32B, που αποδείχτηκε οριακά το βέλτιστο για μεγάλο αριθμό threads, είχαμε αρκετά καλό scale μέχρι τα 8 threads, και την κορυφαία μας επίδοση χρονικά, στα 3.36 second για 64 threads. Εν τέλει, φαίνεται πως το overhead που προσέθεταν οι προσβάσεις στην ουρά των tasks επισκίαζε τη δυναμική της παραλληλοποίησης που προσφέρει η tiled έκδοση του FW.