Συστήματα Παράλληλης Επεξεργασίας

Άσκηση 1 - Παραλληλοποίηση αλγορίθμων σε πολυπύρηνες αρχιτεκτονικές κοινής μνήμης

Χειμερινό Εξάμηνο 2020-2021

Τελική Αναφορά

Ηλιάδης Θρασύβουλος - 03115761
Πέππας Αθανάσιος - 03115749

####

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

2.1

Για το πρώτο ζητούμενο της άσκησης σας παραπέμπουμε στην ενδιάμεση αναφορά μας στην οποία ανακαλύψαμε τρόπους να παραλληλοποιήσουμε τις 3 εκδόσεις του αλγορίθμου.

2.2

- Για τη standard έκδοση του FW η υλοποίηση μας αρκέστηκε σε μία εντολή παραλληλοποίησης στο σημείο που φαίνεται:

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

Με αρχή τις παρατηρήσεις μας από την ενδιάμεση αναφορά είδαμε πως οι 2 nested for λούπες μπορούν να εκτελούνται ταυτόχρονα.

Το παραπάνω είναι και το μοναδικό κομμάτι του κώδικα που αλλάξαμε στο αρχείο `fw.c`

- Για την αναδρομική έκδοση τροποποιήσαμε το παρακάτω κομμάτι του αλγορίθμου

```
void FW_SR (int **A, int arow, int acol,
            int **B, int brow, int bcol,
            int **C, int crow, int ccol,
            int myN, int bsize)
{
     int k,i,j;

     if(myN<=bsize)
        for(k=0; k<myN; k++)
           #pragma omp parallel for shared(myN,k,A) private(i,j)
           for(i=0; i<myN; i++)
              for(j=0; j<myN; j++)
                 A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i]
[bcol+k]+C[crow+k][ccol+j]);
```

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

Αρχικά έχουμε παραλληλοποιήσει την επεξεργασία κάποιων tiles του πίνακα που είδαμε πως δεν έχουν κάποιο dependency και άρα μπορούν να τρέχουν ταυτόχρονα.

Επιπλέον έχουμε κάνει και το base case του αλγορίθμου (το if branch), που εκτελείται όταν το tile φτάσει σε αρκετά μικρό μέγεθος, επίσης παράλληλο κατά την standard έκδοση του αλγορίθμου.

- Τέλος για την tiled έκδοση αλλάξαμε τον κώδικα ως εξής:

```
/*
     * k is the step number
     * i, j are the row and column numbers
     * N is the size of graph
     * B is the size of tile
     */

    /*
     * Because of the nature of the algortithm, we need each block of code
     * that can run in parallel to finish before we move on to the next.
     * One idea is to use the directive #pragma omp barrier after each block
     * of parallel code to ensure the meeting of the dependency conditions.
     */


    /* For every B sized tile (of N total): */
    for(k=0;k<N;k+=B){
       /*
```

```
         * Firstly compute FW for the K-th diagonal tile
         */
        FW(A,k,k,k,B);

        /*
         * Then compute the tiles on the k-th row and k-th column.
         * We can try to parallelize this!
         */

        /* N, B, k should be private, i and j shared */

        #pragma omp parallel
        {
                #pragma omp single
                {
                        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 barrier

        /*
         * Then compute the rest of the tiles for this step.
         * We can also parallelize these computations!
         */
        #pragma omp parallel
        {
                #pragma omp single
                {

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

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

                for(i=k+B; i<N; i+=B)
                   for(j=0; j<k; j+=B)
                        #pragma omp task
```

```
                        {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
                        {FW(A,k,i,j,B);}
                }
        }
        #pragma omp barrier
     /* End of big k for loop */
     }
```

Χρησιμοποιήσαμε openMP tasks για να τρέξουμε τα 4 instances που μπορούν να τρέξουν ταυτόχρονα, και συγχρονίσαμε στα σημεία όπου επιβάλλεται από τον αλγόριθμο. Αν δείτε και την ενδιάμεση αναφορά περιγράφουμε τα κομμάτια κώδικα που πρέπει να εκτελεστούν αυστηρά *πριν* συνεχίσουμε στην εκτέλεση των υπολοίπων.

Να σημειώσουμε εδώ πως αρχικά είχαμε αποφασίσει να παραλληλοποιήσουμε και εδώ το base case. Μετά από δοκιμές όμως το πρόγραμμα μας πετούσε segmentation fault που πιστεύουμε πως οφείλεται στην κοινή χρήση του πίνακα A. Θυμίζουμε πως για την παραλληλία χρειαζόμαστε τον πίνακα A σαν κοινή μεταβλητή ανάμεσα στα threads.

```
inline void FW(int **A, int K, int I, int J, int N)
{
     int i,j,k;
     /* We make the base case of FW parallel the same way we did on the fw.c file
*/
     for(k=K; k<K+N; k++)
        /* We get a segmentation fault here! */
        /* #pragma omp parallel for shared(N, k, A) private(i, j) */
        for(i=I; i<I+N; i++)
           for(j=J; j<J+N; j++)
              A[i][j]=min(A[i][j], A[i][k]+A[k][j]);

}
```

2.3

- Περί δυνατότητες compiler

  Στην αναζήτησή μας για τις παράλληλες δυνατότητες του gcc πέσαμε πάνω σε συζητήσεις όπως αυτή:

https://stackoverflow.com/questions/40088101/can-gcc-make-my-code-parallel, οπότε και αρχίσαμε να ερευνούμε τις έμφυτες ικανότητες του compiler στο να μας βοηθήσει με την παραλληλοποίηση. Μετά από λίγη περιπλάνηση ασχοληθήκαμε με τα υπάρχοντα flags περί optimization του gcc.

Όσον αφορά τα optimization flags του gcc compiler

, τρέχοντας τη μεταγλώττιση με παράμετρο `-O3` ενεργοποιούμε (εκτός των υπολοίπων flags περί optimization) 2 flags πολύ εύφορα για την παραλληλοποίηση, το `-ftree-loop-distribution` και το `-ftree-loop-vectorize`.

Συγκεκριμένα για το flag `-ftree-loop-distribution`:

> Perform loop distribution. This flag can improve cache performance on big loop bodies and allow further loop optimizations, like parallelization or vectorization, to take place.

Ένα άλλο ενδιαφέρον flag που *δεν ενεργοποιείται αυτόματα με το* `-O3` είναι το εξής:

> -floop-parallelize-all
>
> Use the Graphite data dependence analysis to identify loops that can be parallelized. Parallelize all the loops that can be analyzed to not contain loop carried dependences without checking that it is profitable to parallelize the loops.

, όπως και αυτό:

> -ftree-parallelize-loops=n
>
> Parallelize loops, i.e., split their iteration space to run in n threads. This is only possible for loops whose iterations are independent and can be arbitrarily reordered. The optimization is only profitable on multiprocessor machines, for loops that are CPU-intensive, rather than constrained e.g. by memory bandwidth. This option implies -pthread, and thus is only supported on targets that have support for -pthread.

Τώρα όσων αφορά το δεύτερο flag δεν μας βολεύει ιδιαίτερα μια και προτιμάμε να θέτουμε τα threads με τον τρόπο του PBS.

Το πρώτο από την άλλη θα μπορούσε να χρησιμοποιηθεί, και θα κάνουμε μία δοκιμή σε αυτό το σημείο.

Πρώτο μας ένστικτο πριν την εκτέλεση είναι βέβαια πως το flag αυτό είναι λίγο αχρείαστο, εφόσον στο openMP εμείς έχουμε ήδη σε προγραμματιστικό επίπεδο ορίσει ακριβώς τις λούπες και τις περιοχές οι οποίες δύναται να παραλληλοποιηθούν.

Για τη δοκιμή μας, αλλάζουμε την τιμή του CFLAGS στο Makefile σε:

```
CFLAGS= -Wall -O3 -fopenmp -Wno-unused-variable -floop-parallelize-all
```

Κάνουμε make και τρέχουμε την standard έκδοση του αλγορίθμου. Παραθέτουμε εδώ τα αποτελέσματα για μέγεθος πίνακα 4096x4096 για 1, 2, 4, 6, 8, 16, 32 και 64 threads

Threads	**Χωρίς flag**	**Με flag** `-floop-parallelize-all`
1	18.4265	18.3301
2	13.1114	16.1790
4	19.2292	18.9219
6	18.8111	16.6751
8	21.0311	20.2083
16	18.4186	17.6962
32	16.5774	**11.6994**
64	**10.7545**	20.2693

Κοιτώντας κανείς τα αποτελέσματα μπορεί να μην είναι κανείς σίγουρος για την επίδραση του flag στην απόδοση, παίρνοντας όμως την καλύτερη επίδοση από όλες τις δοκιμές καταλήγουμε στο ότι η χρήση του παραπάνω flag είναι κάπως άσκοπη, αφού μπορούμε να επιτύχουμε καλύτερη απόδοση με 64 πυρήνες χωρίς αυτό και βέβαια *εφόσον έχουμε πρόσβαση σε 64 πυρήνες*. Σε άλλη περίπτωση ίσως να αξίζει να ερευνήσουμε παραπάνω τη χρήση αυτού του flag.

- Περί locality

Για την αρχιτεκτονική του μηχανήματος που τρέχουν τα προγράμματα γνωρίζουμε τα εξής

Αριθμός πυρήνων	**4 * 8 = 32**
Αριθμός threads	4 * 8 * 2 = 64
L1 cache size	32 KiB
L2 cache size	256 KiB
L3 cache size	16 MiB
RAM	64 GB * 4

Για τις υλοποιήσεις που χωρίζουν τον αρχικό πίνακα σε μικρότερα κομμάτια έως ένα συγκεκριμένο μέγεθος (εκεί όπου το tile size B είναι μεταβλητή), θα πρέπει να προσπαθήσουμε να εκμεταλλευτούμε την τοπικότητα των δεδομένων. Για αυτό και στις δοκιμές που κάνουμε μεταβάλλουμε το μέγεθος του tile μεταξύ 32, 64, 128 και 256 (B = 32x32 ή 64x64 κοκ) και τελικά *ερμηνεύουμε* τα αποτελέσματα με βάση τα παραπάνω, αντί του αντίθετου που θα ήταν να επιλέξουμε ρητά το μέγεθος του tile με βάση τη θεωρητική αντίληψη του προβλήματος.

- **Tiled: Tasks vs Parallel for**

Κάνοντας δοκιμές με τον κώδικα από το 2.2 αρχίσαμε πάνω στην πορεία της βελτιστοποίησης να πέφτουμε πάνω σε Segmentation Fault το οποίο (πιθανότατα) οφειλόταν στο ότι δεν είχαμε ρητά θέσει τα counters i, j ως private για κάθε task.

Μετά από αυτό αλλάξαμε την υλοποίηση και τελικά χρησιμοποιήσαμε parallel for loops για την εκτέλεση. Ο κώδικας λοιπόν για την tiled έκδοση έγινε:

```
...
     /*
      * k is the step number
      * i, j are the row and column numbers
      * N is the size of graph
      * B is the size of tile
      */

     /*
      * Because of the nature of the algortithm, we need each block of code
      * that can run in parallel to finish before we move on to the next.
      * One idea is to use the directive #pragma omp barrier after each block
      * of parallel code to ensure the meeting of the dependency conditions.
      */


     /* For every B sized tile (of N total): */
     for(k=0;k<N;k+=B){
        /*
```

```
         * Firstly compute FW for the K-th diagonal tile
         */
        FW(A,k,k,k,B);

        /*
         * Then compute the tiles on the k-th row and k-th column.
         * We can try to parallelize this!
         */

        /* N, B, k should be private, i and j shared */

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

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

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

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

        }

        /*
         * Then compute the rest of the tiles for this step.
         * We can also parallelize these computations!
         */
        #pragma omp parallel shared(A,B,k)
        {
                #pragma omp for private(i, j)
                for(i=0; i<k; i+=B)
                   for(j=0; j<k; j+=B)
                        {FW(A,k,i,j,B);}

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

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

                #pragma omp for private(i, j)
                for(i=k+B; i<N; i+=B)
                   for(j=k+B; j<N; j+=B)
                        {FW(A,k,i,j,B);}
        }
    /* End of big k for loop */
```

```
    }
```

Στον παραπάνω κώδικα μπορούμε να βελτιώσουμε ακόμα περισσότερο την απόδοση, εάν πούμε στις 4 for loops του πρώτου block και στις 4 του δεύτερου ότι μπορούν να τρέχουν χωρίς η μία να περιμένει την ολοκλήρωση της προηγούμενης. Αυτό επιτυγχάνεται με το keyword `nowait` του OpenMP. Άρα τελικά ο κώδικάς αλλάζει κάθε γραμμή από

```
            #pragma omp for private(myPrivateVars)
```

σε

```
            #pragma omp for nowait private(myPrivateVars)
```

Παραθέτουμε εδώ και μία σύγκριση των χρόνων πριν και μετά τη χρήση του keyword nowait (Η βέλτιστη τιμή του B (tile size) για την tiled έκδοση βρέθηκε 64, θα φανεί και παρακάτω στα γραφήματα, οπότε οι τιμές εδώ είναι για B=64):

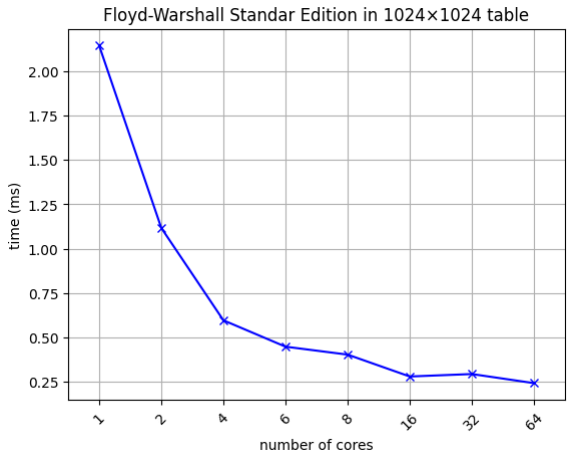
Χρόνοι (4096x4096) B=64 Tiled	**Χωρίς nowait**	**Με nowait**
1	41.9891	41.9739
2	21.4384	21.1951
4	11.0679	11.2203
6	7.7468	7.7431
8	6.3297	6.2827
16	4.1642	3.8116
32	3.7418	**2.9948**
64	**3.5390**	3.0426

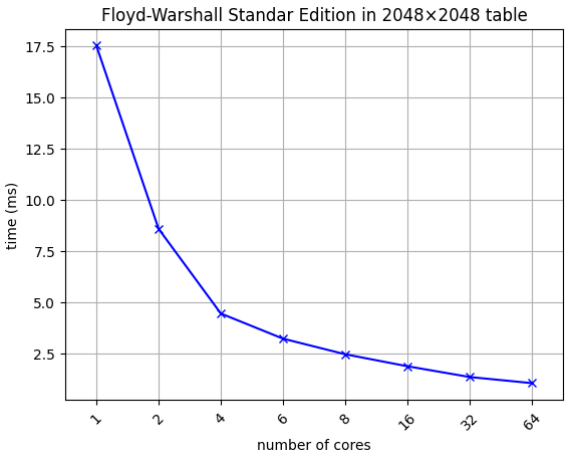
Έχουμε μία μικρή βελτίωση, κυρίως όσο ο βαθμός των threads αυξάνεται όπου έχει και μεγαλύτερη επίδραση το nowait.

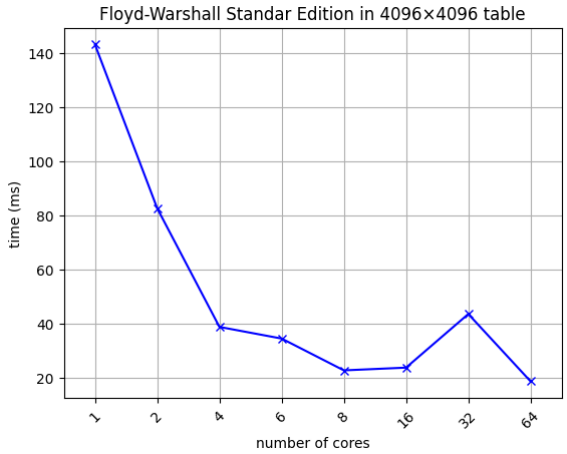
2.4

Standard

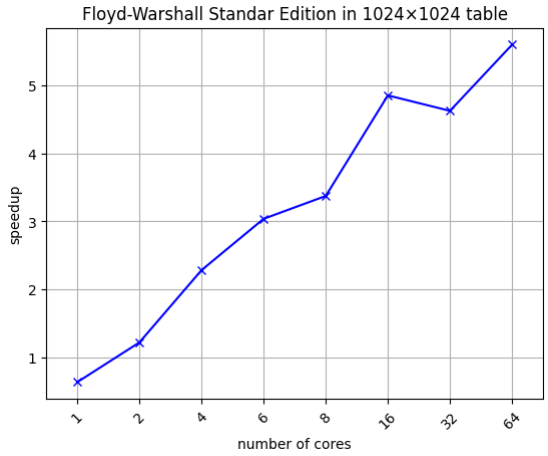
Παρακάτω βλέπουμε για 1024x1024, 2048x2048 και 4096x4096 μέγεθος πίνακα τους χρόνους εκτέλεσης σε σχέση με τον αριθμό των ανατιθέμενων πυρήνων:

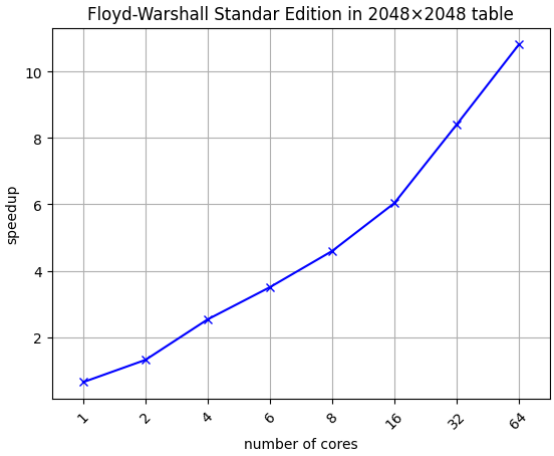


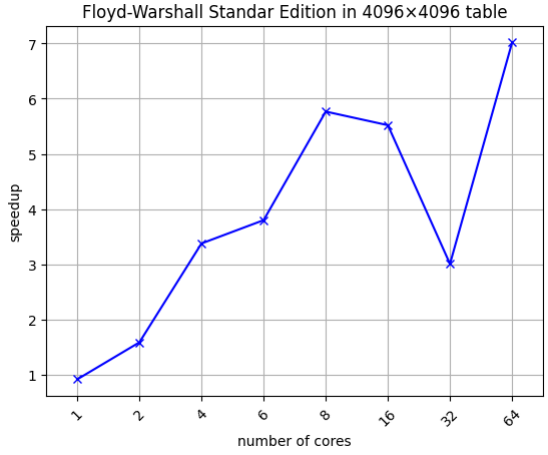




Με βάση την τελική μορφή του αλγορίθμου στην βασική του έκδοση ακολουθούν τα διαγράμματα για το speedup από την παραλληλοποίηση:

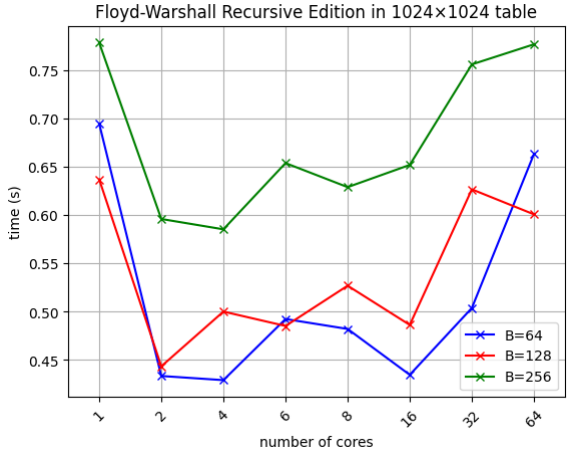


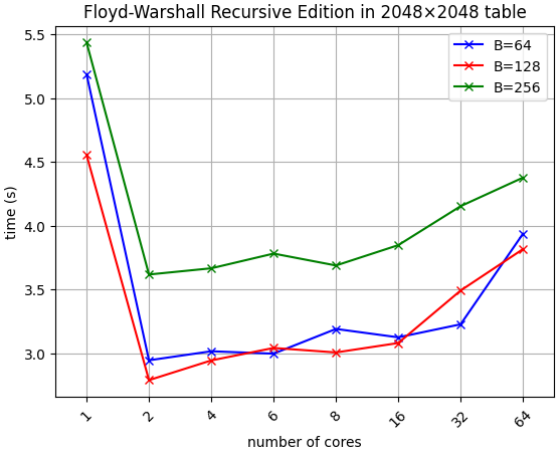


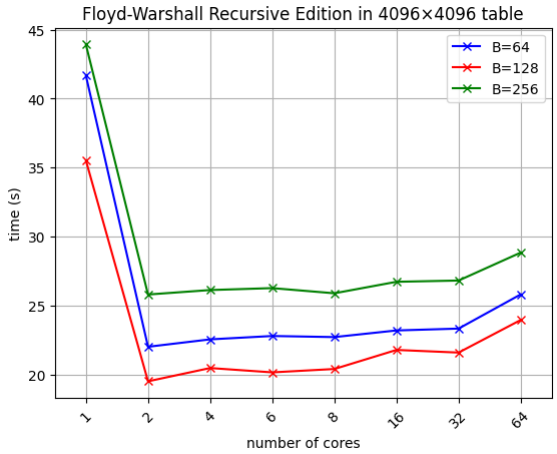


Recursive

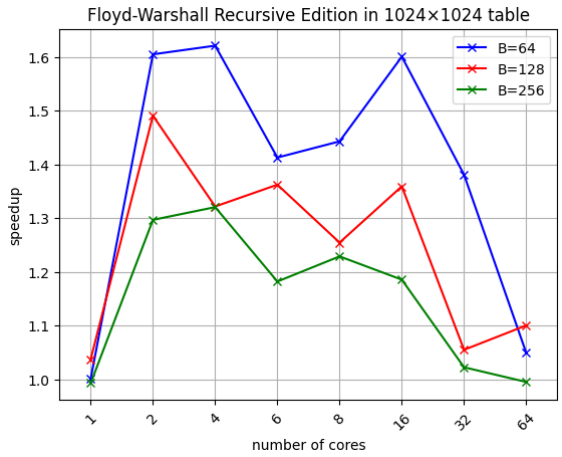
Οι χρόνοι για την recursive έκδοση:

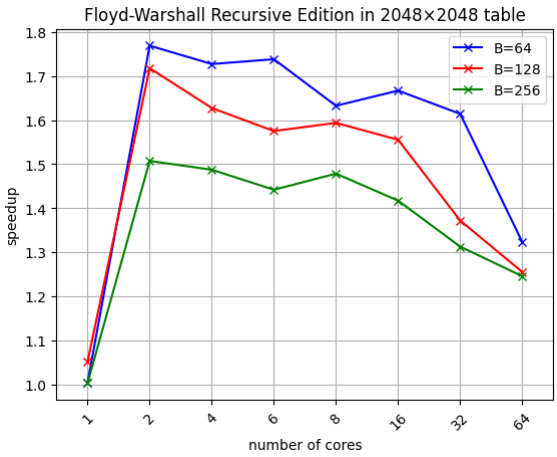


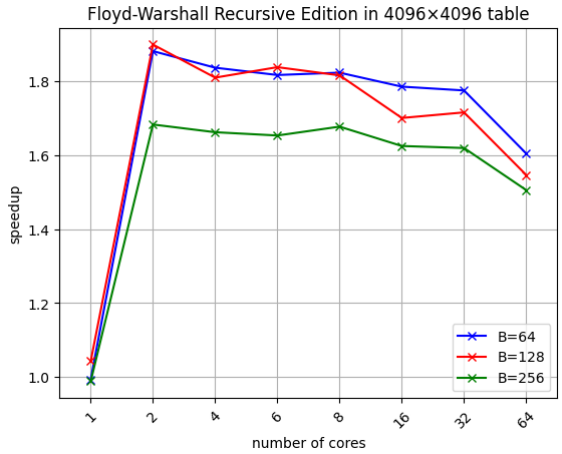




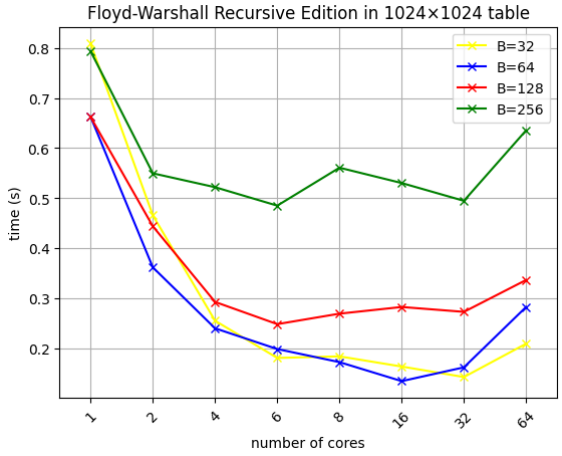
Και το speedup:

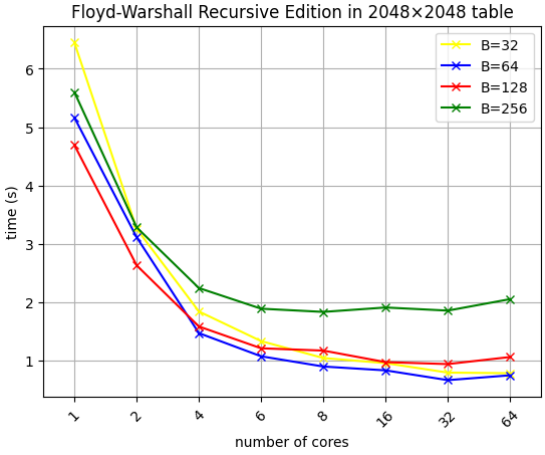


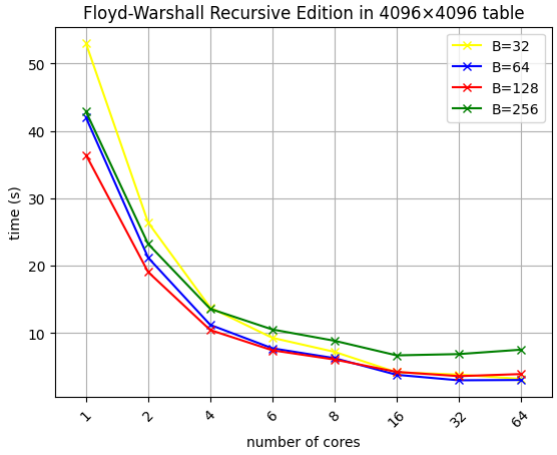




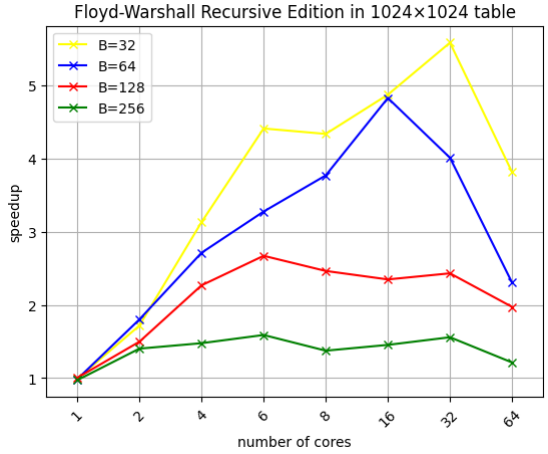
Tiled

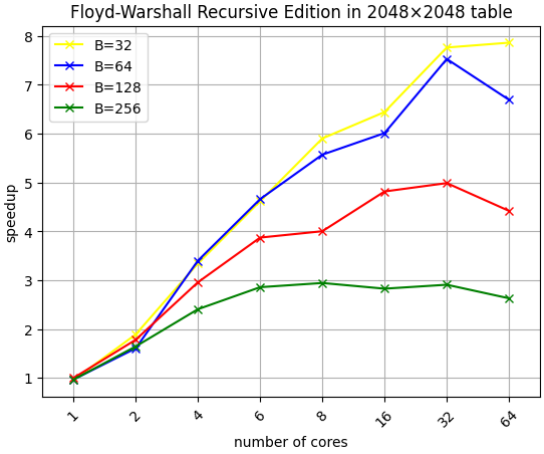


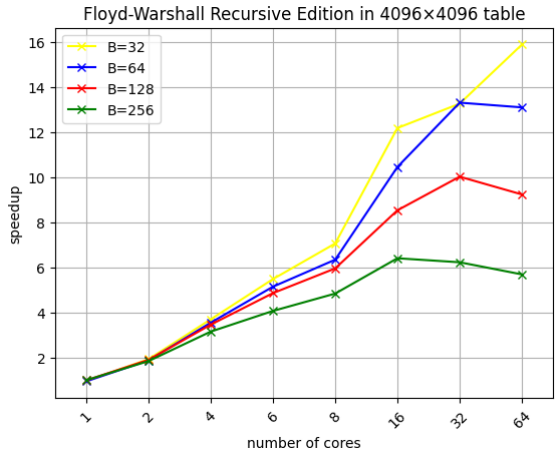




Και το speedup:







2.5 - Καλύτερες Επιδόσεις

1. **Standard / N = 4096x4096 : 64 threads με χρόνο 18.6267**
2. **Recursive / N = 4096x4096 / B = 2 threads με χρόνο 19.4938 :(**
3. **Tiled / N = 4096x4096 / B = 64 : 32 threads με χρόνο 2.9948**