ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Συστήματα Παράλληλης Επεξεργασίας

9ο εξάμηνο

2η Εργαστηριακή Άσκηση

Χειμερινό εξάμηνο 2016-2017

Ομάδα 01

Μπουγουλιάς Παναγιώτης	03112025
Χορτομάνης Ιωάννης	03110825

ΑΣΚΗΣΗ 2

“Παράλληλη επίλυση του αλγορίθμου Floyd-Warshall σε πολυπύρηνες αρχιτεκτονικές ”

1. Ανακαλύψτε τον παραλληλισμό του αλγορίθμου σε κάθε έκδοση και σχεδιάστε την παραλληλοποίησή του.

Στην κλασική έκδοση του αλγόριθμου, παρατηρούμε πως ο υπολογισμός κάθε στοιχείου του πίνακα εξαρτάται μόνο από την τιμή κάποιων στοιχείων του πίνακα μιας προηγούμενης χρονικής στιγμής. Έτσι ο κώδικας δεν μπορεί να παραλληλοποιηθεί θεωρώντας ξεχωριστές εκδόσεις του πίνακα διαφορετικών στιγμών ως ξεχωριστά νήματα και στέλνοντάς τα για (παράλληλη) εκτέλεση σε διαφορετικούς επεξεργαστές, αφού κάθε νήμα χρειάζεται τις τιμές από το νήμα της προηγούμενης χρονικής στιγμής που υπολογίζεται σε άλλον επεξεργαστή.

Λόγω του γεγονότος όμως ότι για τον υπολογισμό κάθε στοιχείου απαιτούνται τιμές από τον πίνακα μόνο της προηγούμενης χρονικής στιγμής (και όχι της τρέχουσας) μπορούμε να παραλληλοποιήσουμε το πρόβλημα θεωρώντας τις γραμμές του πίνακα ως ξεχωριστά νήματα και υπολογίζοντας έτσι όλες τις γραμμές ενός πίνακα (κάποιας χρονικής στιγμής) πριν αρχίσουμε τον υπολογισμό των γραμμών του πίνακα της επόμενης χρονικής στιγμής.

Ο γράφος των εξαρτήσεων για την περίπτωση ενός πίνακα 3x3 δίνεται στο τέλος της αναφοράς.

2. Υλοποιήστε παράλληλες εκδόσεις του αλγορίθμου Floyd-Warshall (τουλάχιστον μία για κάθε σειριακή έκδοση) στο προγραμματιστικό εργαλείο που επιλέξατε και πραγματοποιήστε μετρήσεις για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096 για 1, 2, 4, 8, 16, 32 και 64 threads στο μηχάνημα sandman.

Αμέσως μετά ακολουθούν οι παράλληλες υλοποιήσεις των σειριακών προγραμμάτων που μας δόθηκαν (με πράσινο οι γραμμές κώδικα που προστέθηκαν). Για οικονομία χώρου, παρατίθενται μόνο τα τμήματα κώδικα που παραλληλοποιήθηκαν. Χρησιμοποιήθηκε το εργαλείο παράλληλου προγραμματισμού OpenMP.

1. fw.c

```
    for(k=0;k<N;k++)
      #pragma omp parallel shared(N, A, k) private(i, j) default(none) // Δημιουργία νημάτων
      {
            #pragma omp for schedule(static,1) // Κυκλική ανάθεση βρόχων στα νήματα με το
            for(i=0; i<N; i++)                 // κάθε νήμα να υπολογίζει μια γραμμή του πίνακα
                  for(j=0; j<N; j++)
                        A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
       }
```

2. fw_sr.c

```
            #pragma omp parallel shared(A,B,C,myN,bsize) private(k,i,j)
firstprivate(arow,acol,brow,bcol,crow,ccol) default(none)
            {

                        #pragma omp single
                        {
                              #pragma omp task
                              {
                                    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);
                              }
                        }
            }
      }
```

3. i) fw_tv2.c

```
#pragma omp parallel shared(N,B,A) private(i,j,k) default(none) // Δημιουργία νημάτων
for(k=0;k<N;k+=B){

      FW(A,k,k,k,B);            // Υπολογισμός του εκάστοτε διαγώνιου πλακιδίου

      #pragma omp for schedule(static,1) // Κυκλική ανάθεση βρόχων στα νήματα με το
      for(i=0; i<N; i+=B)               // κάθε νήμα να υπολογίζει μια γραμμή του πίνακα
            if(i != k)            // Υπολογισμός ολόκληρου του "σταυρού" εκτός από το
            {                     // κεντρικό (διαγώνιο) πλακίδιο που υπολογίστηκε πριν
                  FW(A,k,i,k,B);  // Απαιτείται αναμονή ώστε να υπολογιστεί ο
                  FW(A,k,k,i,B);  // "σταυρός" πριν αρχίσουν οι υπολογισμοί των
            }                     // παράπλευρων στοιχείων (δεν τέθηκε "nowait")

      #pragma omp for schedule(static,1)          // Κυκλική ανάθεση με αναμονή για
      for(i=0; i<N; i+=B) {               // τον υπολογισμό όλων των παράπλευρων πλακιδίων
            if(i != k) {                  // πριν από τον υπολογισμό του διαγώνιου πλακιδίου
            for(j=0;j<N;j+=B)                     // και του "σταυρού" της επόμενης φάσης
                  if(j != k) {
                        FW(A,k,i,j,B);
                  }
            }
      }
}
```

ii) fw_tv2_cl3.c

```
#pragma omp parallel shared(N,B,A) private(i,j,k) default(none)     // Δημιουργία νημάτων
      for(k=0;k<N;k+=B){

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

      #pragma omp for schedule(static,1) // Κυκλική ανάθεση βρόχων στα νήματα με το
      for(i=0; i<N; i+=B)               // κάθε νήμα να υπολογίζει μια γραμμή του πίνακα
            if(i != k)
            {
                  FW(A,k,i,k,B);
                  FW(A,k,k,i,B);
            }

      #pragma omp for schedule(static,1) // Κυκλική ανάθεση βρόχων στα νήματα με το
      for(i=0; i<N; i+=B) {             // κάθε νήμα να υπολογίζει μια γραμμή του πίνακα
            if(i != k) {
               #pragma omp parallel shared(N,B,A) private(j) firstprivate(i,k) default(none)
                  {       // Εδώ δημιουργούμε ένθετη παράλληλη περιοχή
                        #pragma omp for schedule(static,1) nowait // Με κυκλική
```

```
                        for(j=0;j<N;j+=B)     // ανάθεση βρόχων με το κάθε νήμα πλέον να
                            if(j != k) {      // υπολογίζει μια στήλη του πίνακα
                                FW(A,k,i,j,B);
                            }
                        }
                    }
                }
            }
```

iii) fw_tv3.c

```
 #pragma omp parallel shared(N,B,A) private(i,j,k) default(none)    // Δημιουργία νημάτων
      for(k=0;k<N;k+=B){
      {
            FW(A,k,k,k,B); // Ένα νήμα εκτελεί πάντα τον υπολογισμό του κεντρικού πλακιδίου

            #pragma omp for schedule(static,1) nowait         // Κάθε νήμα υπολογίζει ένα tile
            for(i=0; i<k; i+=B)                               // στην πάνω πλευρά του “σταυρού”
                  FW(A,k,i,k,B);

            #pragma omp for schedule(static,1) nowait         // Κάθε νήμα υπολογίζει ένα tile
            for(i=k+B; i<N; i+=B)                             // στην κάτω πλευρά του “σταυρού”
                  FW(A,k,i,k,B);

            #pragma omp for schedule(static,1) nowait         // Κάθε νήμα υπολογίζει ένα tile
            for(j=0; j<k; j+=B)                               // στην αριστερή πλευρά του “σταυρού”
                  FW(A,k,k,j,B);

            #pragma omp for schedule(static,1) nowait         // Κάθε νήμα υπολογίζει ένα tile
            for(j=k+B; j<N; j+=B)                             // στην δεξιά πλευρά του “σταυρού”
                  FW(A,k,k,j,B);

            #pragma omp barrier           // Συγχρονισμός των νημάτων ώστε να υπολογιστεί
                                          // όλος ο “σταυρός” πριν από τις παράπλευρες περιοχές
            #pragma omp for collapse(2) schedule(static) nowait  // Κάθε νήμα υπολογίζει ένα
            for(i=0; i<k; i+=B)           // tile στην άνω αριστερή παράπλευρη περιοχή
                  for(j=0; j<k; j+=B)     // Τα 2 loops “ενώνονται” με την λειτουργία collapse(2)
                        FW(A,k,i,j,B);    //  η  οποία  προϋποθέτει  τέλεια  φωλιασμένους
βρόχους

            #pragma omp for collapse(2) schedule(static) nowait  // Κάθε νήμα υπολογίζει ένα
            for(i=0; i<k; i+=B)           // tile στην άνω δεξιά παράπλευρη περιοχή
                  for(j=k+B; j<N; j+=B)
                        FW(A,k,i,j,B);

            #pragma omp for collapse(2) schedule(static) nowait // Κάθε νήμα υπολογίζει ένα
            for(i=k+B; i<N; i+=B)         // tile στην κάτω αριστερή παράπλευρη περιοχή
                  for(j=0; j<k; j+=B)
                        FW(A,k,i,j,B);
```

```
        #pragma omp for collapse(2) schedule(static) nowait // Κάθε νήμα υπολογίζει ένα
        for(i=k+B; i<N; i+=B)           // tile στην κάτω δεξιά παράπλευρη περιοχή
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);

        #pragma omp barrier             // Συγχρονισμός των νημάτων ώστε να υπολογιστούν
                                        // οι παράπλευρες περιοχές πριν από τον επόμενο πίνακα
    }
  }
```

iv) fw_tiled.c // Λειτουργεί όπως και η “tv3” αλλά χωρίς την λειτουργία “collapse”

```
#pragma omp parallel shared(N,B,A) private(i,j,k) default(none)    // Δημιουργία νημάτων
    for(k=0;k<N;k+=B){

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

        #pragma omp for schedule(static,1) nowait
        for(i=0; i<k; i+=B)
            FW(A,k,i,k,B);

        #pragma omp for schedule(static,1) nowait
        for(i=k+B; i<N; i+=B)
            FW(A,k,i,k,B);

        #pragma omp for schedule(static,1) nowait
        for(j=0; j<k; j+=B)
            FW(A,k,k,j,B);

        #pragma omp for schedule(static,1) nowait
        for(j=k+B; j<N; j+=B)
            FW(A,k,k,j,B);

        #pragma omp barrier

        #pragma omp for schedule(static,1) nowait
        for(i=0; i<k; i+=B)
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for schedule(static,1) nowait
        for(i=0; i<k; i+=B)
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for schedule(static,1) nowait
        for(i=k+B; i<N; i+=B)
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for schedule(static,1) nowait
```

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

        #pragma omp barrier
    }
```

Μετρήσεις που ελήφθησαν από τον sandman:

Το σημείο αναφοράς των μετρήσεων είναι ο χρόνος εκτέλεσης του σειριακού προγράμματος και όχι ο χρόνος εκτέλεσης της παράλληλης έκδοσης με 1 νήμα.

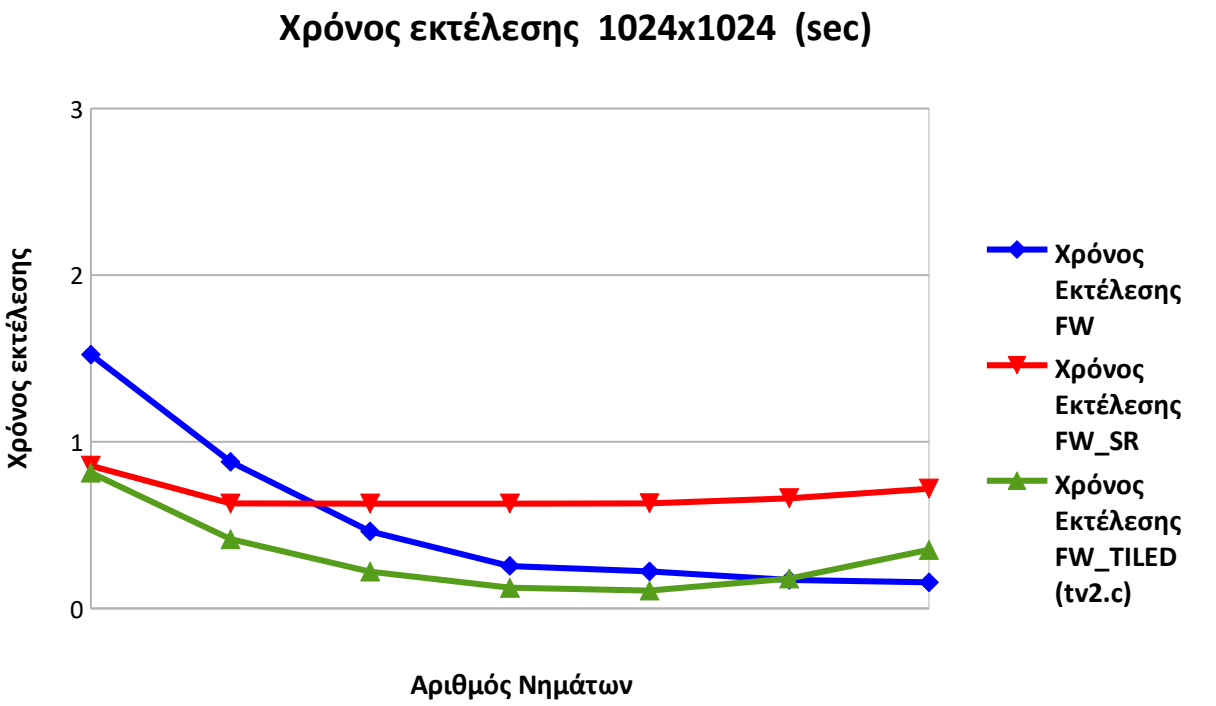
Ακολουθούν τα αποτελέσματα για πίνακες 1024, 2048, 4096 συγκρίνοντας τις εκδόσεις fw.c, fw_sr.c, fw_tiled.c μεταξύ τους για block size=32 και 64(18 διαγράμματα).

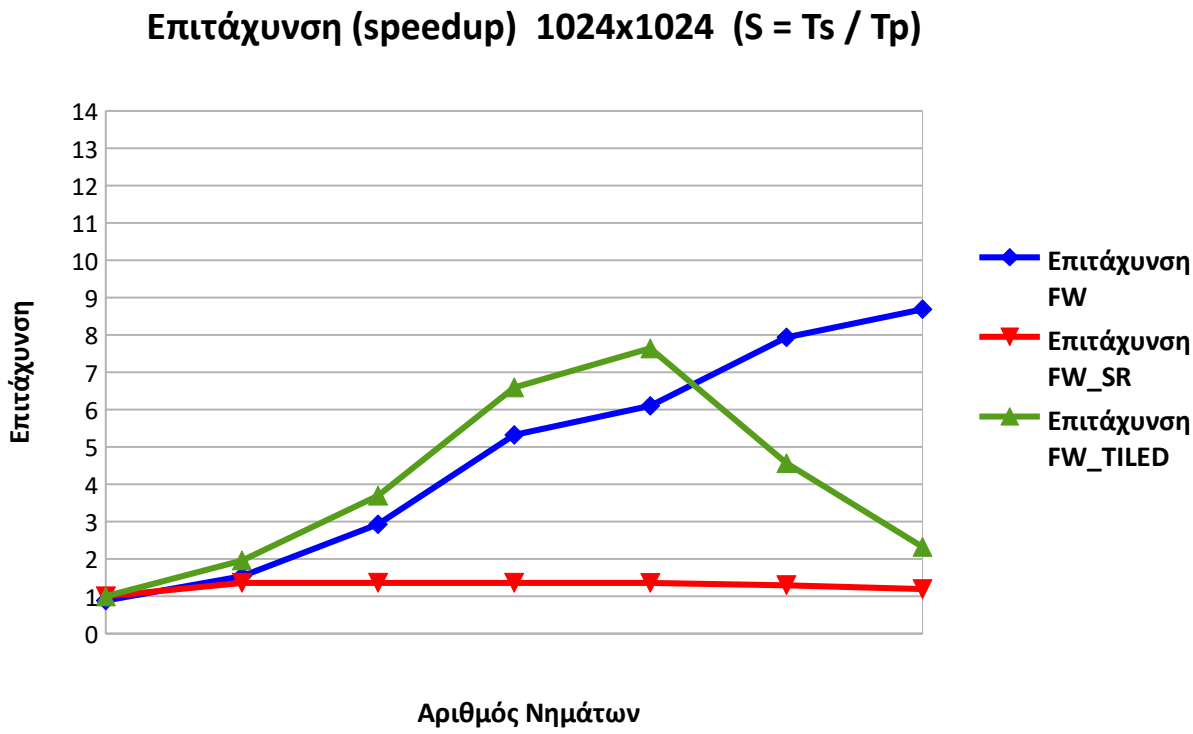
Στην περίπτωση των 4-16 threads κάναμε αρκετές δοκιμές για το πώς θα πρέπει να μοιράζονται στους επεξεργαστές διαφορετικών κόμβων και από αυτά τα νούμερα κρατήσαμε το καλύτερο. Για συντομία δεν παρουσιάζονται άλλα αποτελέσματα, αλλά υπάρχει κάποιος σύντομος σχολιασμός στην αντίστοιχη ενότητα των παρατηρήσεων.

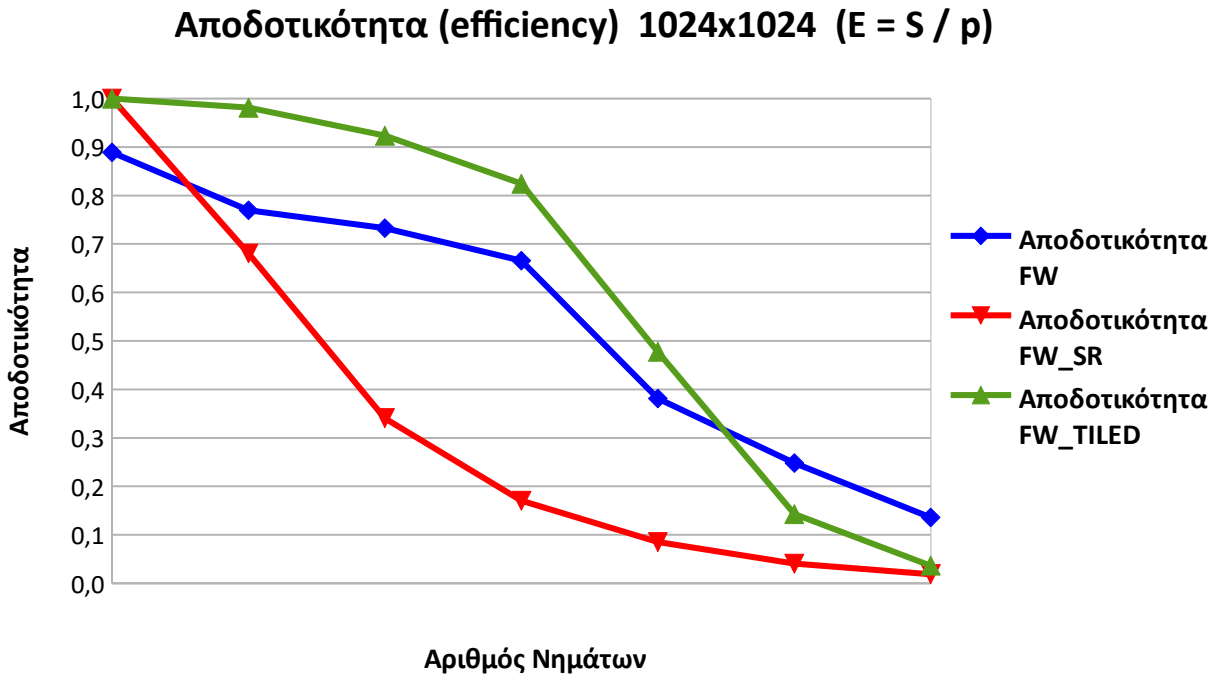
Άλλα block size δοκιμάστηκαν αλλά πιστεύουμε ότι ενδιαφέρον έχουν αυτά τα δυο μεγέθη που παρουσιάζονται.

1) Block size = 32, N=1024,2048,4096

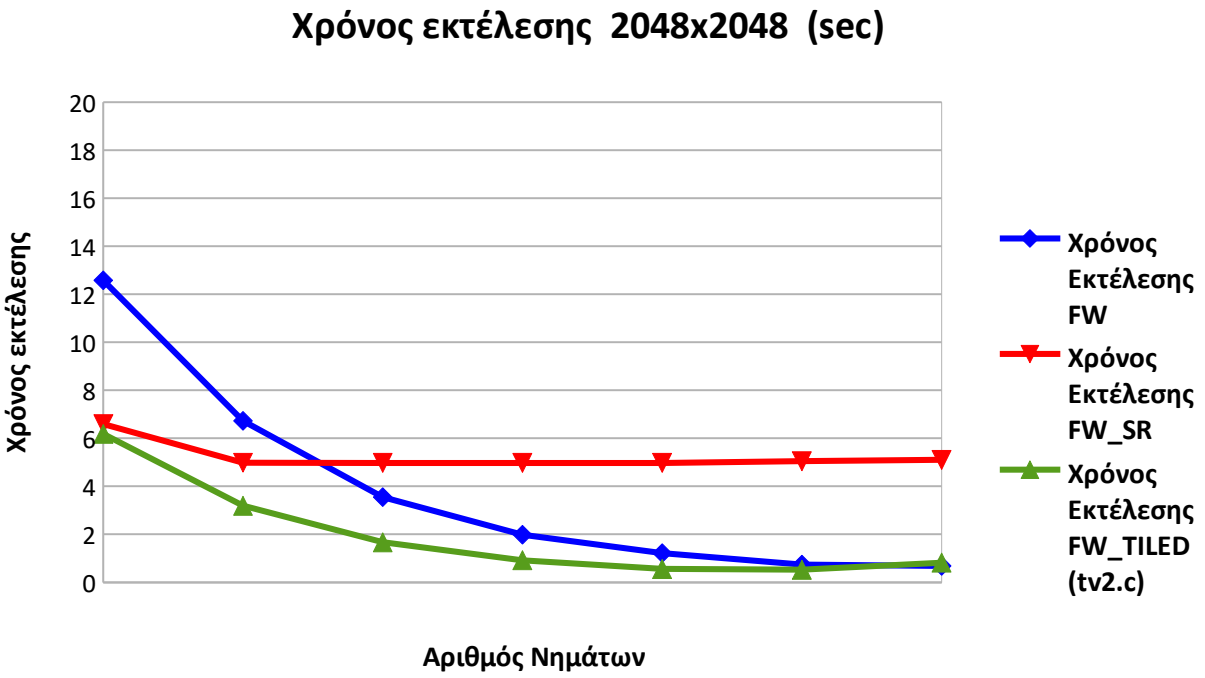
Αριθμος Νημάτων (1024x 1024)	Χαρακτηριστικά Προσο Μοίωσης	Χρόνος Εκτέλεσης FW	Επιτάχυνση FW	Αποδοτικότητα FW	Χρόνος Εκτέλεσης FW_SR	Επιτάχυνση FW_SR	Αποδοτικότητα FW_SR	Χρονος Εκτέλεσης FW_TILED (tv2.c)	Επιτάχυνση FW_TILED	Αποδοτικότητα FW_TILED
1	Serial	1,354	1,000	1,000	0.8559	1,000	1,000	0,814	1,000	1,000
1	1 thread	1,523	0,889	0,889	0,856	1,000	1,000	0,814	1,000	1,000
2	2 threads	0,880	1,539	0,769	0,629	1,361	0,680	0,415	1,962	0,981
4	4 threads	0,462	2,931	0,733	0,629	1,362	0,340	0,220	3,694	0,924
8	8 threads	0,254	5,323	0,665	0,629	1,362	0,170	0,123	6,595	0,824
16	16 threads	0,222	6,098	0,381	0,630	1,359	0,085	0,107	7,641	0,478
32	32 threads	0,171	7,935	0,248	0,660	1,297	0,041	0,178	4,564	0,143
64	64 threads	0,156	8,689	0,136	0,718	1,191	0,019	0,350	2,324	0,036

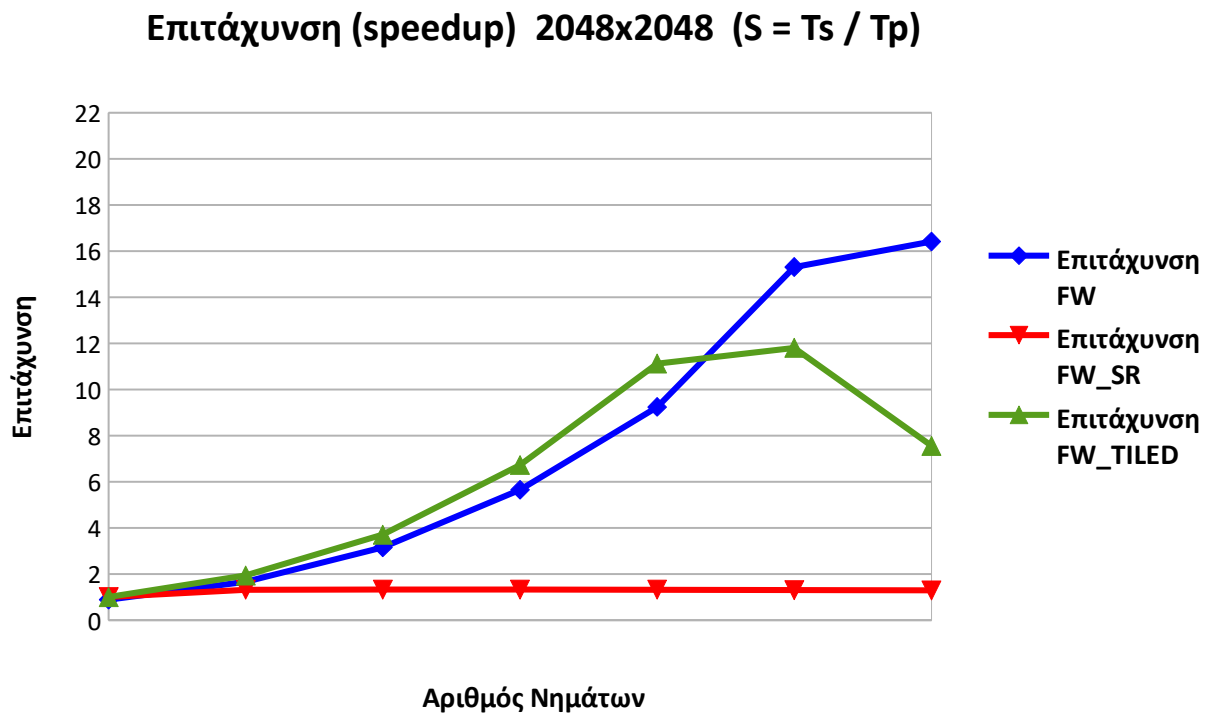

Χρόνος εκτέλεσης 1024x1024 (sec)
Χρόνος εκτέλεσης
3
2
1
0
Αριθμός Νημάτων
Χρόνος Εκτέλεσης FW
Χρόνος Εκτέλεσης FW_SR
Χρόνος Εκτέλεσης FW_TILED (tv2.c)

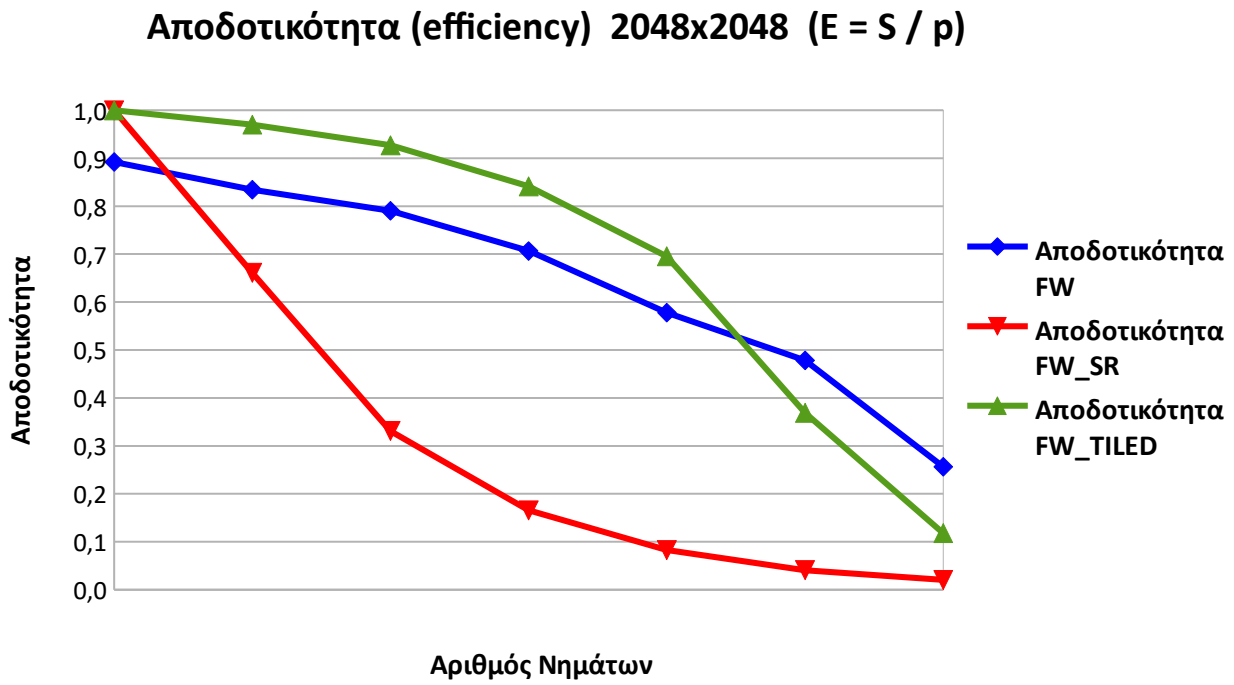

Επιτάχυνση (speedup) 1024x1024 (S = Ts / Tp)
Επιτάχυνση
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Αριθμός Νημάτων
Επιτάχυνση FW
Επιτάχυνση FW_SR
Επιτάχυνση FW_TILED



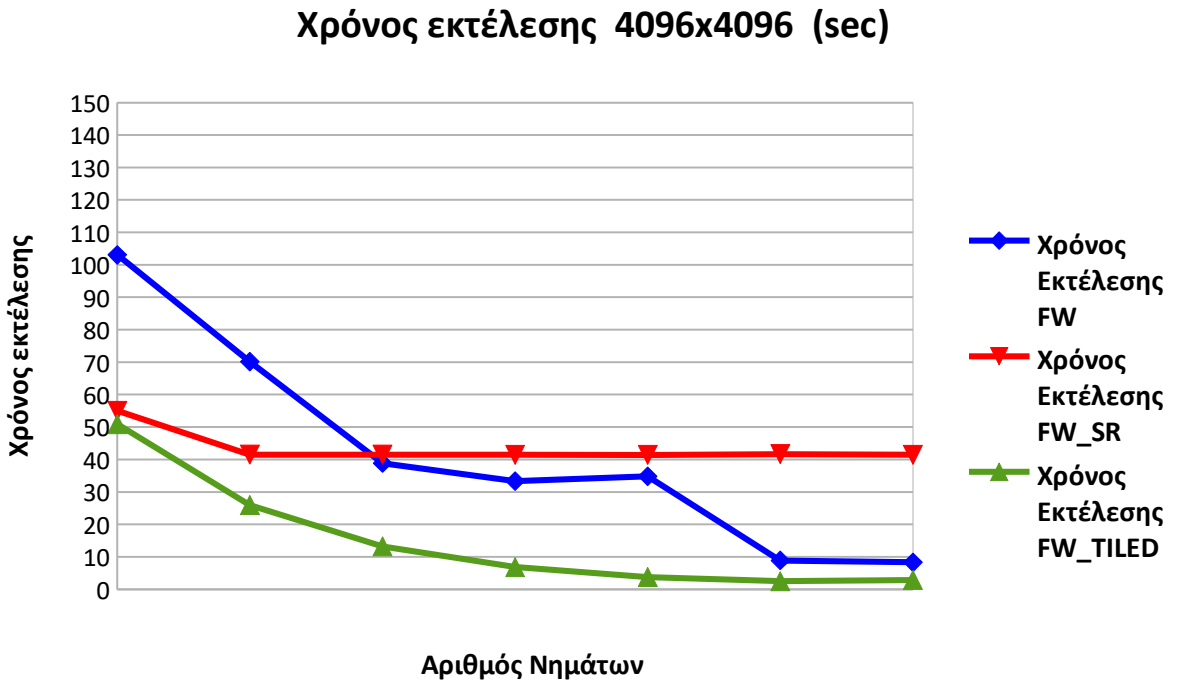
Αριθμος Νημάτων (2048x 2048)	Χαρακτηριστικά Προσο Μοίωσης	Χρόνος Εκτέλεσης FW	Επιτάχυνση FW	Αποδοτικότητα FW	Χρόνος Εκτέλεσης FW_SR	Επιτάχυνση FW_SR	Αποδοτικότητα FW_SR	Χρονος Εκτέλεσης FW_TILED (tv2.c)	Επιτάχυνση FW_TILED	Αποδοτικότητα FW_TILED
1	Serial	11,223	1,000	1,000	6,587	1,000	1,000	6,182	1,000	1,000
1	1 thread	12,582	0,892	0,892	6,587	1,000	1,000	6,182	1,000	1,000
2	2 threads	6,728	1,668	0,834	4,983	1,322	0,661	3,187	1,940	0,970
4	4 threads	3,549	3,162	0,790	4,974	1,324	0,331	1,668	3,708	0,927
8	8 threads	1,985	5,653	0,707	4,971	1,325	0,166	0,919	6,730	0,841
16	16 threads	1,214	9,242	0,578	4,976	1,324	0,083	0,556	11,125	0,695
32	32 threads	0,733	15,311	0,478	5,045	1,306	0,041	0,524	11,805	0,369
64	64 threads	0,684	16,417	0,257	5,100	1,292	0,020	0,818	7,557	0,118

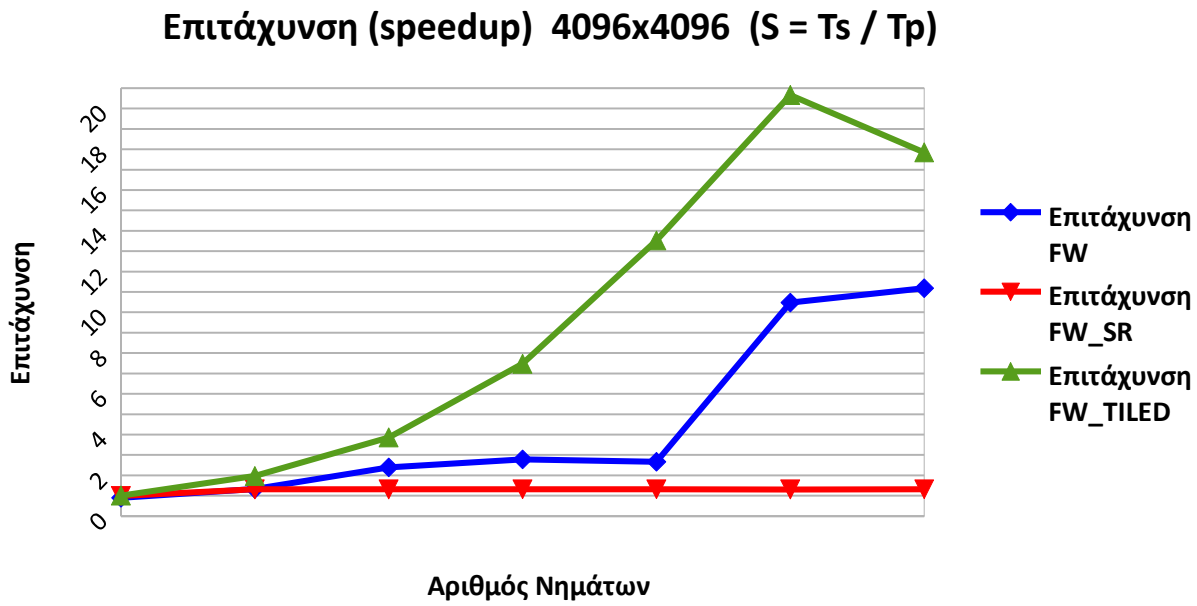

Χρόνος εκτέλεσης 2048x2048 (sec)
Χρόνος εκτέλεσης
20
18
16
14
12
10
8
6
4
2
0
Αριθμός Νημάτων
Χρόνος Εκτέλεσης FW
Χρόνος Εκτέλεσης FW_SR
Χρόνος Εκτέλεσης FW_TILED (tv2.c)

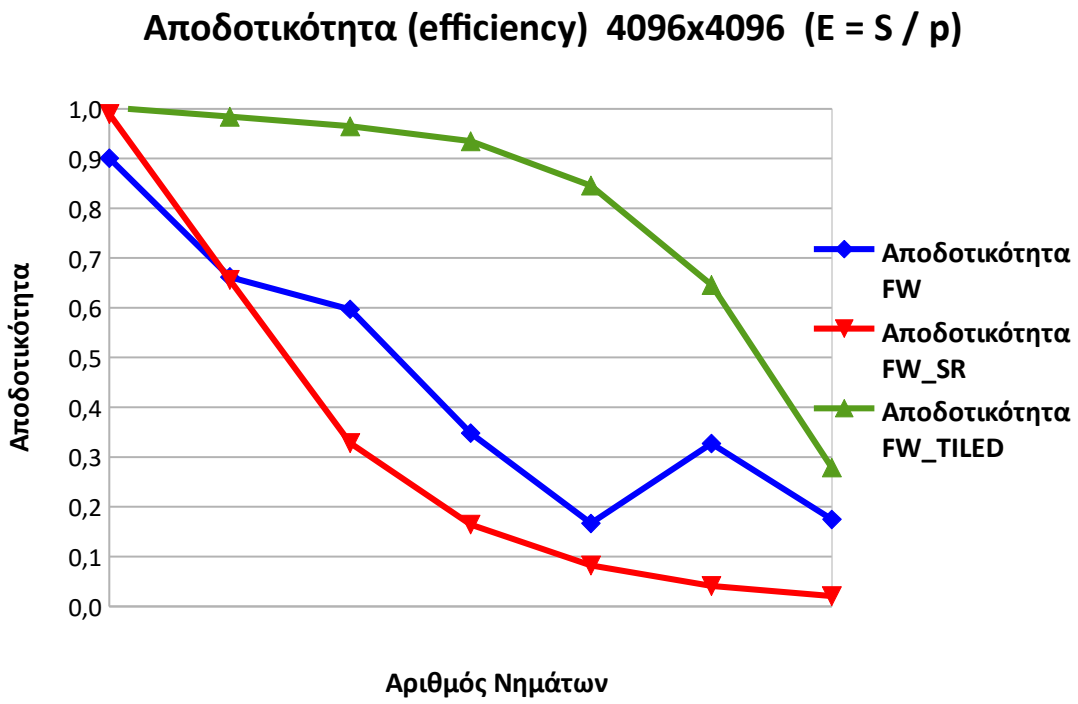

Επιτάχυνση (speedup) 2048x2048 (S = Ts / Tp)
Επιτάχυνση
22
20
18
16
14
12
10
8
6
4
2
0
Αριθμός Νημάτων
Επιτάχυνση FW
Επιτάχυνση FW_SR
Επιτάχυνση FW_TILED



Αριθμος Νημάτων (4096x 4096)	Χαρακτηριστικά Προσο Μοίωσης	Χρόνος Εκτέλεσης FW	Επιτάχυνση FW	Αποδοτικότητα FW	Χρόνος Εκτέλεσης FW_SR	Επιτάχυνση FW_SR	Αποδοτικότητα FW_SR	Χρόνος Εκτέλεσης FW_TILED	Επιτάχυνση FW_TILED	Αποδοτικότητα FW_TILED
1	Serial	92,815	1,000	1,000	54.4284	1,000	1,000	51.0463	1,000	1,000
1	1 thread	103,064	0,901	0,901	55,008	0,989	0,989	50,897	1,003	1,003
2	2 threads	70,126	1,324	0,662	41,493	1,312	0,656	25,937	1,968	0,984
4	4 threads	38,874	2,388	0,597	41,481	1,312	0,328	13,228	3,859	0,965
8	8 threads	33,332	2,785	0,348	41,428	1,314	0,164	6,827	7,477	0,935
16	16 threads	34,806	2,667	0,167	41,411	1,314	0,082	3,773	13,528	0,845
32	32 threads	8,864	10,471	0,327	41,656	1,307	0,041	2,470	20,665	0,646
64	64 threads	8,303	11,179	0,175	41,446	1,313	0,021	2,860	17,848	0,279

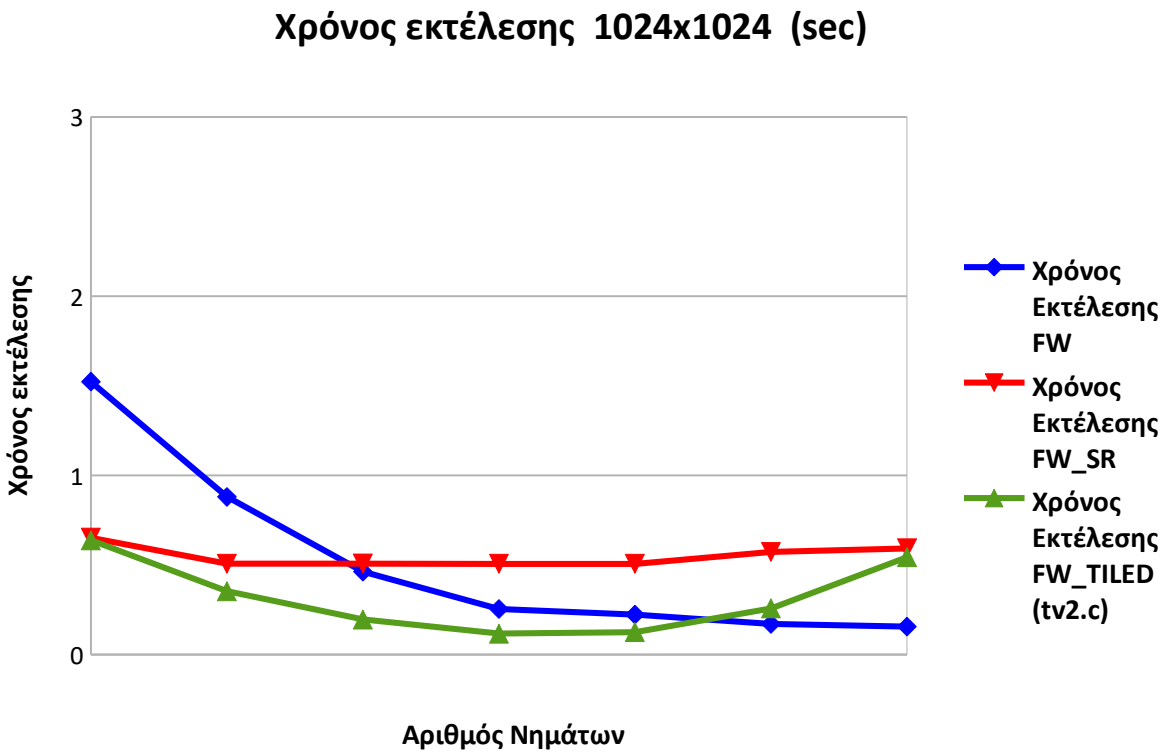

Χρόνος εκτέλεσης 4096x4096 (sec)
Χρόνος εκτέλεσης
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
Αριθμός Νημάτων
Χρόνος Εκτέλεσης FW
Χρόνος Εκτέλεσης FW_SR
Χρόνος Εκτέλεσης FW_TILED

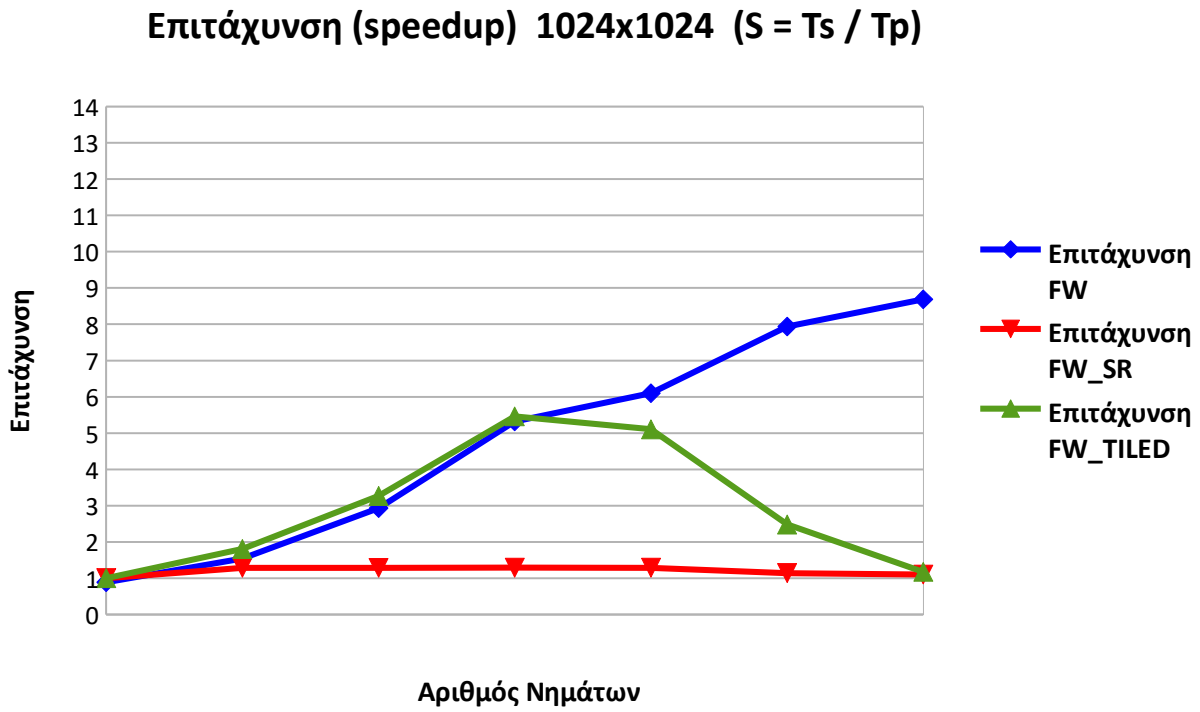

Επιτάχυνση (speedup) 4096x4096 (S = Ts / Tp)
Επιτάχυνση
0
2
4
6
8
10
12
14
16
18
20
Αριθμός Νημάτων
Επιτάχυνση FW
Επιτάχυνση FW_SR
Επιτάχυνση FW_TILED

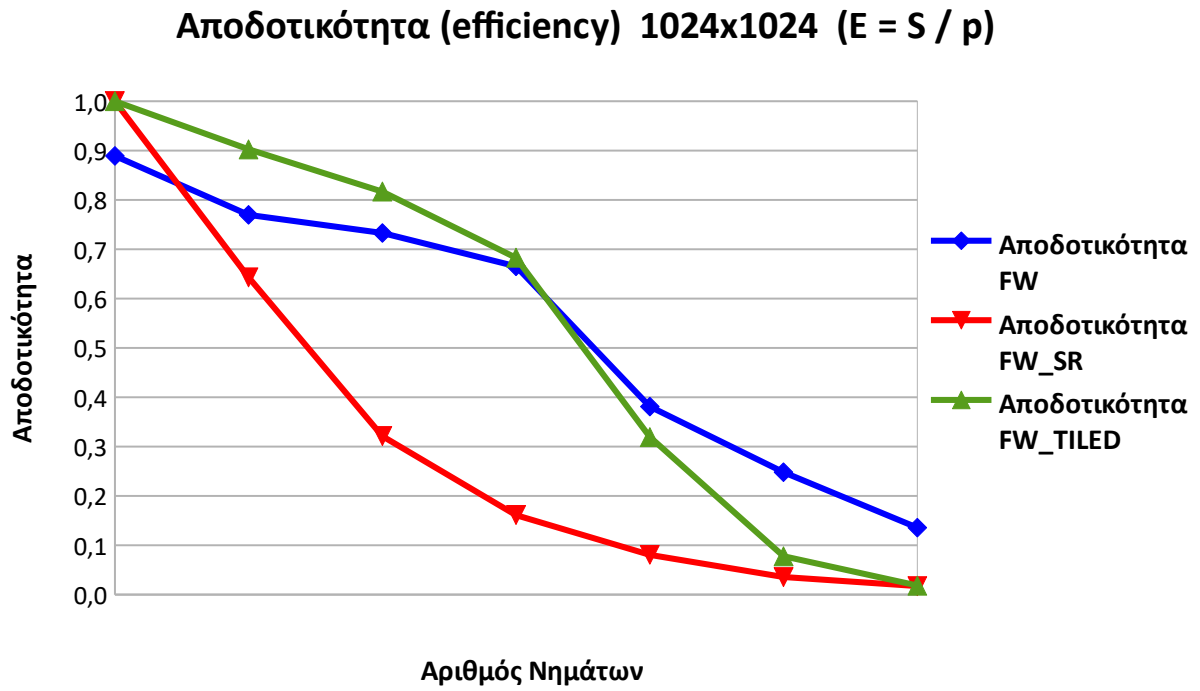

Αποδοτικότητα (efficiency) 4096x4096 (E = S / p)
Αποδοτικότητα
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
Αριθμός Νημάτων
Αποδοτικότητα FW
Αποδοτικότητα FW_SR
Αποδοτικότητα FW_TILED

2) Block size = 64, N = 1024,2048,4096

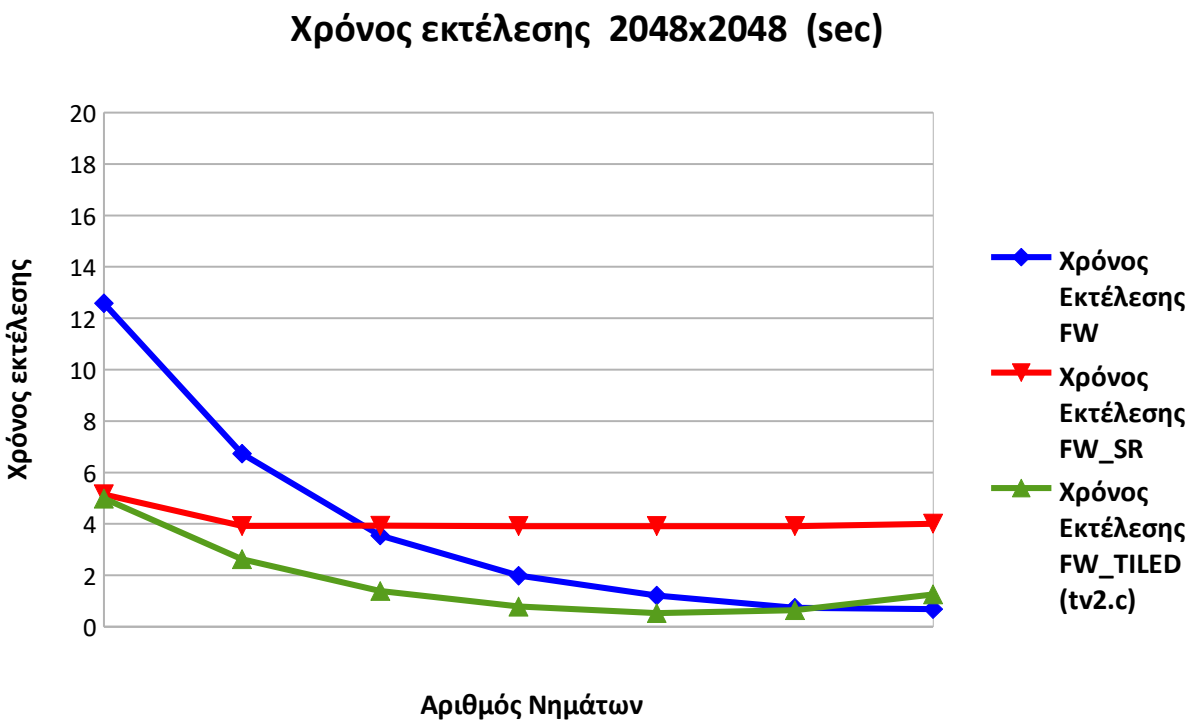
Χαρακτηριστικά Προσομοίωσης	Χρόνος Εκτέλεσης FW	Επιτάχυνση FW	Αποδοτικότητα FW	Χρόνος Εκτέλεσης FW_SR	Επιτάχυνση FW_SR	Αποδοτικότητα FW_SR	Χρόνος Εκτέλεσης FW_TILED (tv2.c)	Επιτάχυνση FW_TILED	Αποδοτικότητα FW_TILED
Serial	1,354	1,000	1,000	0.6514	1,000	1,000	0,637	1,000	1,000
1 thread	1,523	0,889	0,889	0,651	1,000	1,000	0,637	1,000	1,000
2 threads	0,880	1,539	0,769	0,507	1,286	0,643	0,353	1,804	0,902
4 threads	0,462	2,931	0,733	0,507	1,285	0,321	0,195	3,267	0,817
8 threads	0,254	5,323	0,665	0,506	1,289	0,161	0,117	5,461	0,683
16 threads	0,222	6,098	0,381	0,506	1,287	0,080	0,125	5,107	0,319
32 threads	0,171	7,935	0,248	0,572	1,139	0,036	0,257	2,480	0,077
64 threads	0,156	8,689	0,136	0,592	1,101	0,017	0,543	1,173	0,018

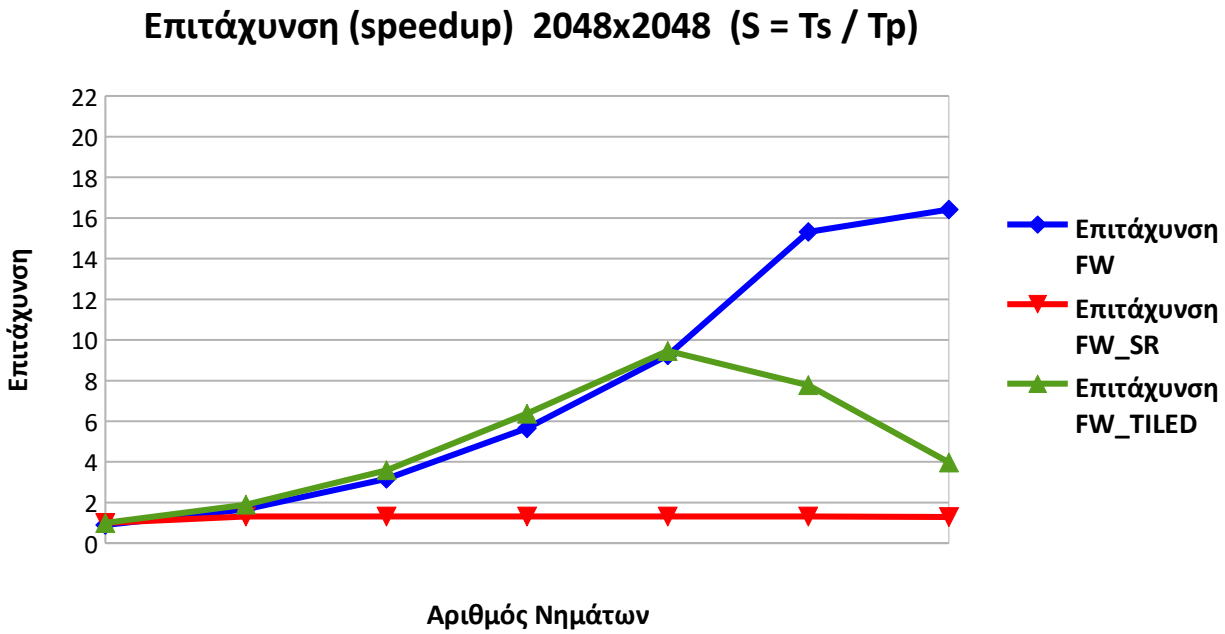


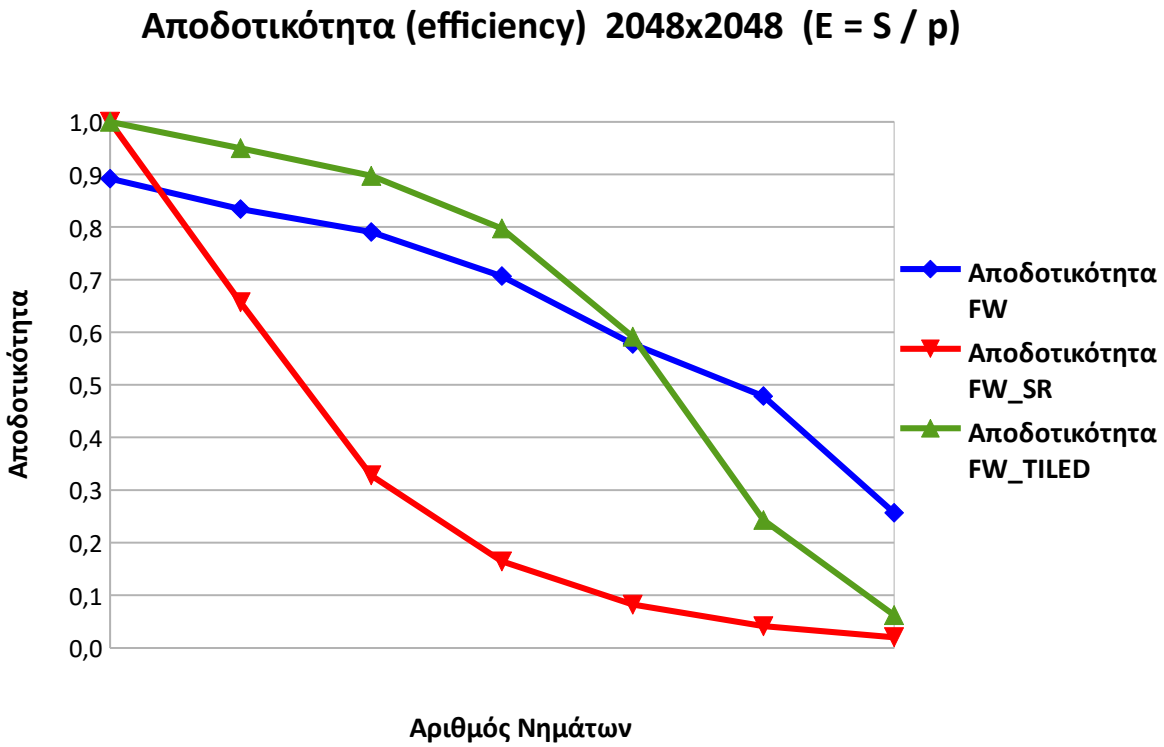

Επιτάχυνση (speedup) 1024x1024 (S = Ts / Tp)
Επιτάχυνση
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Αριθμός Νημάτων
Επιτάχυνση FW
Επιτάχυνση FW_SR
Επιτάχυνση FW_TILED


Αποδοτικότητα (efficiency) 1024x1024 (E = S / p)
Αποδοτικότητα
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
Αριθμός Νημάτων
Αποδοτικότητα FW
Αποδοτικότητα FW_SR
Αποδοτικότητα FW_TILED

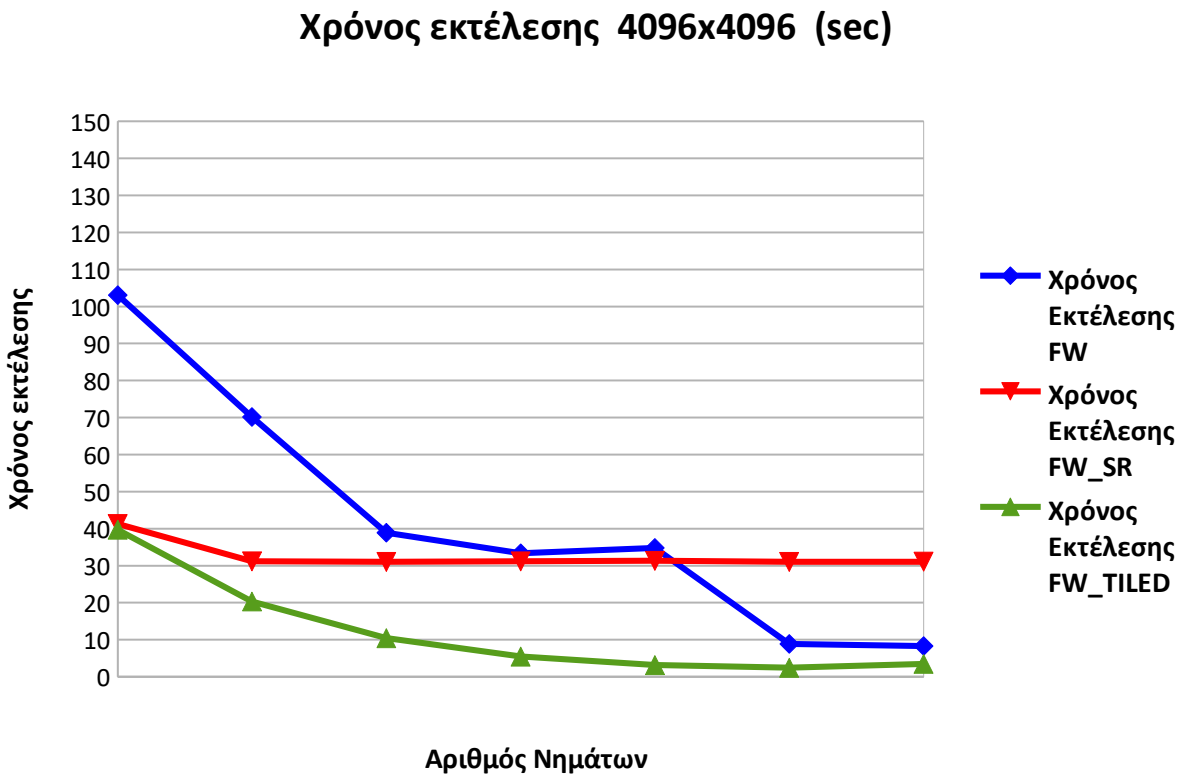
Αριθμος Νημάτων (2048x 2048)	Χαρακτηριστικά Προσο Μοίωσης	Χρόνος Εκτέλεσης FW	Επιτάχυνση FW	Αποδοτικότητα FW	Χρόνος Εκτέλεσης FW_SR	Επιτάχυνση FW_SR	Αποδοτικότητα FW_SR	Χρονος Εκτέλεσης FW_TILED (tv2.c)	Επιτάχυνση FW_TILED	Αποδοτικότητα FW_TILED
1	Serial	11,223	1,000	1,000	5,148	1,000	1,000	4,989	1,000	1,000
1	1 thread	12,582	0,892	0,892	5,148	1,000	1,000	4,989	1,000	1,000
2	2 threads	6,728	1,668	0,834	3,918	1,314	0,657	2,626	1,900	0,950
4	4 threads	3,549	3,162	0,790	3,927	1,311	0,328	1,390	3,589	0,897
8	8 threads	1,985	5,653	0,707	3,914	1,315	0,164	0,782	6,378	0,797
16	16 threads	1,214	9,242	0,578	3,913	1,316	0,082	0,527	9,463	0,591
32	32 threads	0,733	15,311	0,478	3,911	1,316	0,041	0,642	7,775	0,243
64	64 threads	0,684	16,417	0,257	4,005	1,285	0,020	1,256	3,973	0,062

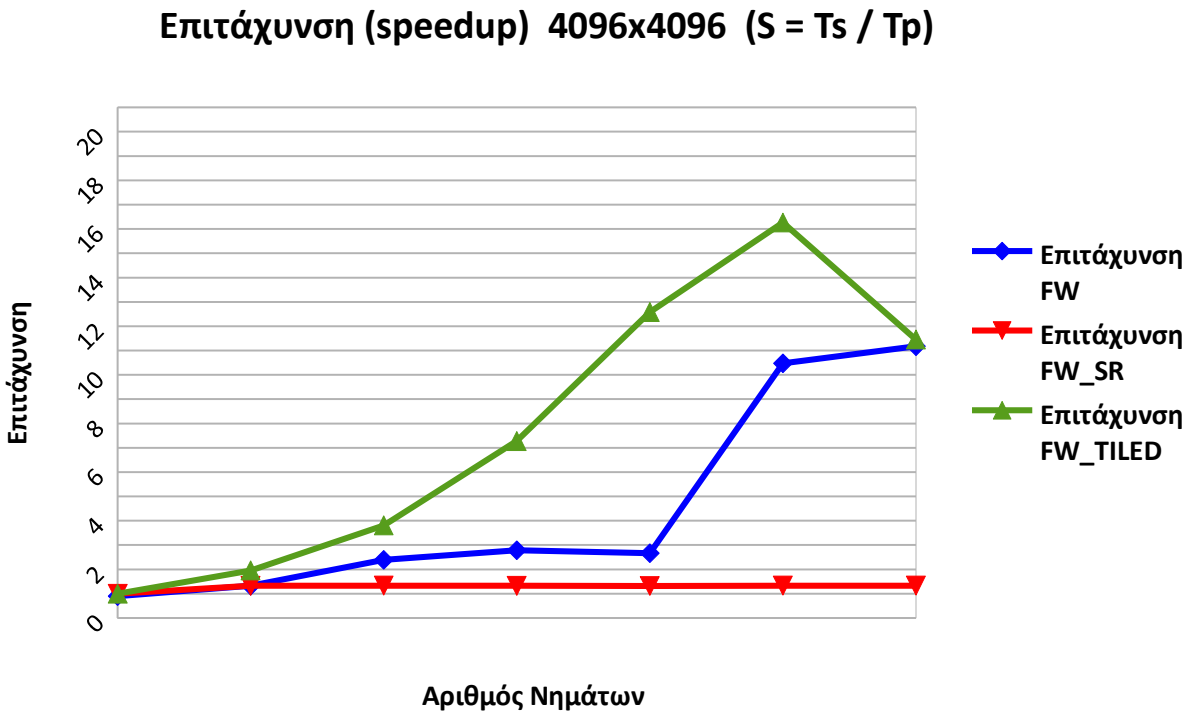


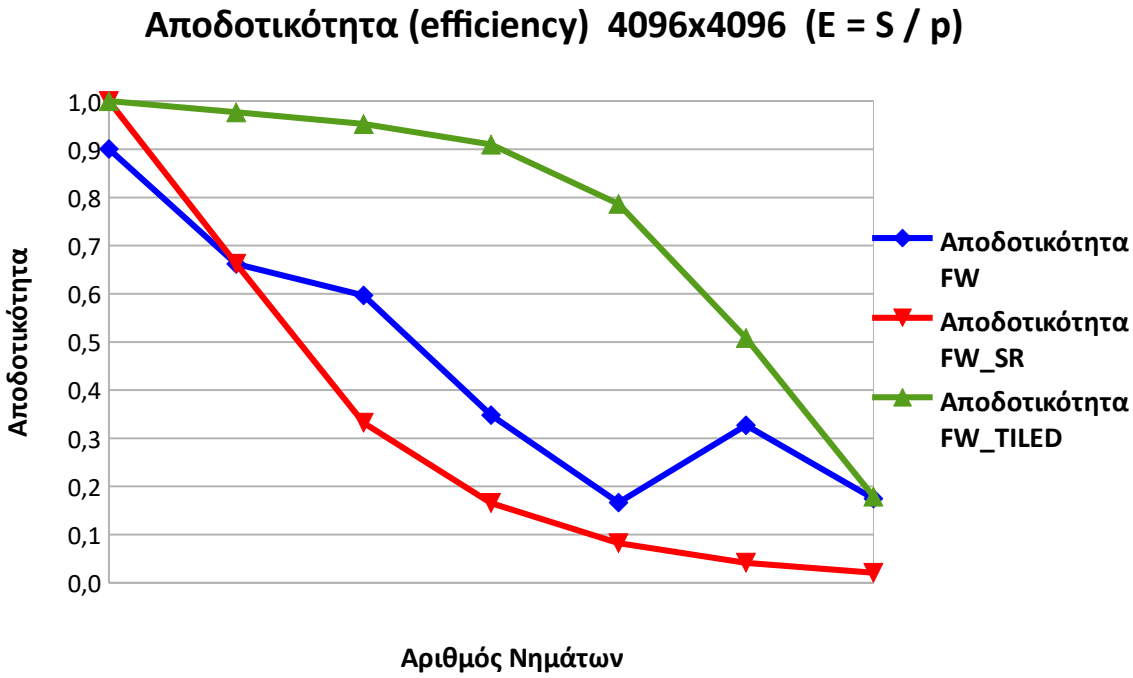

Επιτάχυνση (speedup) 2048x2048 (S = Ts / Tp)
Επιτάχυνση
22
20
18
16
14
12
10
8
6
4
2
0
Αριθμός Νημάτων
Επιτάχυνση FW
Επιτάχυνση FW_SR
Επιτάχυνση FW_TILED


Αποδοτικότητα (efficiency) 2048x2048 (E = S / p)
Αποδοτικότητα
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
Αριθμός Νημάτων
Αποδοτικότητα FW
Αποδοτικότητα FW_SR
Αποδοτικότητα FW_TILED

Αριθμος Νημάτων (4096x 4096)	Χαρακτηριστικά Προσο Μοίωσης	Χρόνος Εκτέλεσης FW	Επιτάχυνση FW	Αποδοτικότητα FW	Χρόνος Εκτέλεσης FW_SR	Επιτάχυνση FW_SR	Αποδοτικότητα FW_SR	Χρόνος Εκτέλεσης FW_TILED	Επιτάχυνση FW_TILED	Αποδοτικότητα FW_TILED
1	Serial	92,815	1,000	1,000	41.2591	1,000	1,000	39,714	1,000	1,000
1	1 thread	103,064	0,901	0,901	41,259	1,000	1,000	39,714	1,000	1,000
2	2 threads	70,126	1,324	0,662	31,152	1,324	0,662	20,329	1,954	0,977
4	4 threads	38,874	2,388	0,597	31,039	1,329	0,332	10,423	3,810	0,953
8	8 threads	33,332	2,785	0,348	31,178	1,323	0,165	5,457	7,278	0,910
16	16 threads	34,806	2,667	0,167	31,314	1,318	0,082	3,156	12,584	0,787
32	32 threads	8,864	10,471	0,327	31,032	1,330	0,042	2,442	16,264	0,508
64	64 threads	8,303	11,179	0,175	31,061	1,328	0,021	3,468	11,453	0,179




Επιτάχυνση (speedup) 4096x4096 (S = Ts / Tp)
Επιτάχυνση
0
2
4
6
8
10
12
14
16
18
20
Αριθμός Νημάτων
Επιτάχυνση FW
Επιτάχυνση FW_SR
Επιτάχυνση FW_TILED


Αποδοτικότητα (efficiency) 4096x4096 (E = S / p)
Αποδοτικότητα
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
Αριθμός Νημάτων
Αποδοτικότητα FW
Αποδοτικότητα FW_SR
Αποδοτικότητα FW_TILED

3. Στρατηγική παραλληλοποίησης.

Γράφοι εξαρτήσεων, σχεδιασμός παραλληλισμού, τοπικότητα

Στην τελευταία σελίδα παρατίθενται οι γράφοι εξαρτήσεων και για τις τρεις εκδοχές του αλγόριθμου (κλασική, αναδρομική (recursive) και tiled).

Παρατηρήθηκε ότι για την αρχική σειριακή έκδοση “fw.c” ο υπολογισμός κάθε στοιχείου a_{ij} εξαρτάται από την ολοκλήρωση του υπολογισμού των στοιχείων a_{ik} a_{ij} a_{kj} του πίνακα της προηγούμενης χρονικής στιγμής (δηλαδή για k=k-1).

Για την tiled έκδοση “fw_tiled” οι εξαρτήσεις σε γενικές γραμμές συνοψίζονται στο ότι απαιτείται κατ' αρχήν ο υπολογισμός του άνω αριστερά διαγώνιου πλακιδίου και εν συνεχεία πρέπει να γίνει ο υπολογισμός της γραμμής και της στήλης (“σταυρός”) στην οποία ανήκει το πλακίδιο και ως τρίτο βήμα ο υπολογισμός των υπόλοιπων πλακιδίων, ο οποίος πρέπει να έχει ολοκληρωθεί πριν αρχίσει να υπολογίζεται το επόμενο instance του πίνακα (καθώς ο υπολογισμός των πλακιδίων του επόμενου στιγμιότυπου απαιτεί να έχουν υπολογιστεί τα πλακίδια του προηγούμενου στιγμιότυπου του πίνακα).

Για κάθε στιγμιότυπο του πίνακα (δηλαδή για κάθε k), ο υπολογισμός της γραμμής και της στήλης στην οποία ανήκει το εκάστοτε διαγώνιο πλακίδιο έχει δομηθεί ώστε να είναι “embarassingly parallel” υπολογισμός, όπως επίσης και ο υπολογισμός των πλακιδίων που απομένουν (εκτός του “σταυρού”). Ο υπολογισμός λοιπόν των τεσσάρων τμημάτων του “σταυρού” (αριστερό, δεξιό, πάνω, κάτω) μπορεί να γίνει όλος μαζί και αφού αυτός ολοκληρωθεί τότε το ίδιο μπορεί να γίνει και με τις 4 περιοχές του πίνακα τις οποίες χωρίζει ο “σταυρός”.

Recursive:

Για την αναδρομική έκδοση “fw_sr.c” ο υπολογισμός κάθε τεταρτημορίου του πίνακα εξαρτάται αναδρομικά από τον υπολογισμό των εσωτερικών του τεταρτημορίων κλπ.

Πιστεύουμε ότι δεν μπορούμε να πετύχουμε πολύ μεγάλο speedup αφού όπως φαίνεται από το task graph στους 8 υπολογισμούς το μέγιστο μονοπάτι είναι 6, δηλαδή speedup = 8/6.

4. Αξιοποιήστε τις παρατηρήσεις σας από τις (α)πρώτες υλοποιήσεις, την (β)κατανόηση της αρχιτεκτονικής του μηχανήματος sandman, (γ)τις δομές παράλληλου προγραμματισμού που προσφέρει το εργαλείο που επιλέξατε και τις (δ)δυνατότητες του μεταγλωττιστή για να βελτιώσετε την παράλληλη επίδοση του αλγορίθμου Floyd-Warshall.

α. Γενικά συμπεράσματα(συνέχεια από την ενδιάμεση αναφορά)

Το FW δεν πετυχαίνει τόσο καλούς χρόνους γιατί για πολύ μεγάλες εισόδους (μεγάλα μεγέθη πίνακα) ο πίνακας δεν χωράει στην κρυφή μνήμη. Θα δαπανηθούν λοιπόν πολλοί κύκλοι επεξεργαστή στην αναμονή κάθε φορά τμήματος του πίνακα προκειμένου να μεταφερθεί από την κύρια μνήμη στην κρυφή. Οι χρόνοι του συγκεκριμένου αλγόριθμου βέβαια βελτιώνονται αρκετά με την κατανομή του κώδικα σε περισσότερα νήματα. Βασικά βέβαια μας ενδιαφέρει ο απόλυτος χρόνος εκτέλεσης.

Η αναδρομική έκδοση (SR) του αλγόριθμου εμφανίζει αναλογικά τις χειρότερες επιδόσεις στην παραλληλοποίηση, λόγω του γεγονότος ότι ακολουθήθηκε task centric τρόπο σχεδίασης της υλοποίησης. Ο A01 και ο A10 μπορούν να υπολογιστούν παράλληλα. Σε αυτή την περίπτωση κατάλληλο μέγεθος μπλοκ είναι το 64, αφού στην cache θα πρέπει να είναι 2 μπλοκ. Ενδιαφέρον παρουσιάζει και η περίπτωση των 32 όπου μπορούμε να εκμεταλλευτούμε περισσότερους εργάτες (νήματα).

Η έκδοση Tiled εμφανίζει συνολικά τις καλύτερες επιδόσεις. Στην χειρότερη περίπτωση κάθε στιγμή στην L1 cache πρέπει να είναι 3 tiles μεγέθους BxB το καθένα, οπότε εκμεταλλευόμενη η υλοποίηση μικρότερα κομμάτια του πίνακα (τα Tiles) και με δεδομένο ότι όλα τα εσωτερικά for είναι παραλληλοποιήσιμα, πετυχαίνει τους καλύτερους χρόνους σε σχέση με όλες τις προηγούμενες περιπτώσεις. Λάβαμε υπόψιν μας ότι το μέγεθος της cache προκειμένου αυτή η στρατηγική να αποδίδει πρέπει να είναι τουλάχιστον ίσο με **3** tiles x **B** στοιχεία / γραμμή tile x **B** στοιχεία / στήλη tile x **n** bytes / στοιχείο (προκειμένου να έχουμε μεταφορά όλων των απαιτούμενων στοιχείων με μια πρόσβαση).

β. Κατανόηση της αρχιτεκτονικής του μηχανήματος sandman

Με την εντολή “hwloc-ls” είδαμε την εσωτερική δομή του μηχανήματος Sandman.

Σχετικά με την αρχιτεκτονική του χρησιμοποιούμενου υπολογιστή (Sandman), παρατηρούμε ότι το μέγεθος της L3 cache είναι τέτοιο (16 MiB) ώστε να επαρκεί μόνο για την αποθήκευση του πίνακα 2048x2048 (και βεβαίως του πίνακα 1024x1024), αλλά όχι του 4096x4096. Τα 16 MiB της L3 κρυφής μνήμης ισούνται με $16x1024 = 2^4 \times 2^{10} = 2^{24}$ bytes. Ένας πίνακας 2048x2048 (με δεδομένο ότι κάθε ακέραιο στοιχείο του καταλαμβάνει χώρο ίσο με 4 bytes, κάτι που διαπιστώσαμε παίρνοντας την τιμή του sizeof(int)) καταλαμβάνει συνολικά χώρο ίσο με 2048 (στοιχεία ανά γραμμή) x 2048 (στοιχεία ανά στήλη) x 4 (bytes ανά στοιχείο) = $2^{11} \times 2^{11} \times 2^2 = 2^{24}$ bytes. Άρα ο πίνακας μεγέθους 2048x2048 χωράει ακριβώς στην κρυφή μνήμη ενός node οπότε αν τα threads που χρησιμοποιούμε είναι μέχρι 8 τότε η βέλτιστη λύση είναι η χρήση ενός μόνο node με 8 επεξεργαστές με την αποθήκευση ολόκληρου του πίνακα εκεί (προκειμένου να αποφευχθούν προσβάσεις σε “μακρινή” κρυφή μνήμη αλλά και το επιπλέον κόστος επικοινωνιών μεταξύ διεργασιών σε διαφορετικούς κόμβους).

Θεωρητικά βέβαια απαιτείται και κάποια κρυφή μνήμη για τις μεταβλητές που θα χρησιμοποιηθούν, τα εκτελέσιμα κλπ, οπότε ίσως δεν θα χωρέσει ολόκληρος ο πίνακας στον κόμβο και θα δημιουργηθούν κάποια (λίγα υποθέτουμε) misses.

Ο πίνακας μεγέθους 4096x4096 καταλαμβάνει συνολικά χώρο ίσο με 4096 (στοιχεία ανά γραμμή) x 4096 (στοιχεία ανά στήλη) x 4 (bytes ανά στοιχείο) = $2^{12} \times 2^{12} \times 2^2 = 2^{24}$ bytes. Ο πίνακας αυτός λοιπόν δεν χωράει σε έναν κόμβο αλλά το γεγονός ότι η μνήμη που απαιτεί είναι ακριβώς τετραπλάσια από την κρυφή μνήμη του κάθε κόμβου και το ότι ο υπολογιστής αποτελείται από συνολικά τέσσερις κόμβους σημαίνει ότι ο πίνακας μπορεί να κατατμηθεί σε 4 “τεταρτημόρια”, με το κάθε ένα από αυτά να αποστέλλεται προς υπολογισμό σε έναν κόμβο (με τους περιορισμούς για την cache που αναφέρθηκαν και πριν).

Στην αρχικοποίηση του πίνακα (μέσω του **util-na.c**) ορίσαμε παράλληλη περιοχή όπου ο εξωτερικός βρόχος for παραλληλοποιήθηκε με το compiler directive:

```
#pragma omp for schedule(static,128)
```

(δηλαδή round-robin διαμοιρασμός με τον αριθμό των γραμμών που ανατίθενται στο κάθε νήμα (chunk size) ίσο με 128). Για παράδειγμα ο πίνακας με μέγεθος 1024 μπορεί να διαμοιραστεί σε 8 νήματα (αν υπάρχουν), με το κάθε νήμα να αρχικοποιεί 128 γραμμές του πίνακα (παρομοίως ο πίνακας μεγέθους 2048 σε 16 νήματα και ο πίνακας μεγέθους 4096 σε 32 νήματα). Το κάθε νήμα παίρνει μνήμη όταν αρχικοποιεί κάποιο τμήμα του πίνακα (και όχι όταν γίνεται η κλήση της malloc). Υποθέτουμε λοιπόν ότι επειδή είναι πολύ πιθανόν κατά την εκτέλεση του

αλγορίθμου αυτό το νήμα να ξαναχρησιμοποιήσει αυτό το κομμάτι της μνήμης, θέλουμε το νήμα να “έχει κοντά του” αυτήν την μνήμη (με αυτόν τον τρόπο ευνοούμε την “τοπικότητα” της μνήμης).

Αυτό έγινε αφού με τη malloc δεν παίρνουμε μνήμη στην πραγματικότητα, απλώς ζητάμε από το λειτουργικό τη μνήμη που πιστεύουμε οτι χρειαζόμαστε, και το λειτουργικό αφού κοιτάξει το χάρτη μνήμης και έχει διαθέσιμη απλά μας δίνει κάποιες εικονικές διευθύνσεις. Μνήμη στην πραγματικότητα παίρνουμε σε φυσικούς πόρους όταν γράψουμε για πρώτη φορά σε αυτήν **(first touch policy)**.

Για να εκμεταλλευτούμε την αρχιτεκτονική του μηχανήματος ορίζαμε το κάθε νήμα σε ποιον επεξεργαστή θα τρέξει(affinity), με τις παρακάτω εντολές:

```
export OMP_NUM_THREADS=8              # Αίτημα για 8 νήματα
export GOMP_CPU_AFFINITY="0-7"        # Πρόσδεση των νημάτων στους πρώτους 8
                                      # επεξεργαστές (δηλαδή στο πρώτο node)
export OMP_PROC_BIND=true             # Ενεργοποίηση της σχετικής μεταβλητής
                                      # περιβάλλοντος
```

δηλαδή τα πρώτα 8 νήματα που γεννιούνται δένονται με τις CPU 0-7 του πρώτου κόμβου. Έτσι συνδυάσαμε την αρχιτεκτονική και τη first touch policy ώστε να κερδίσουμε χρόνο μειώνοντας το cache miss rate.

Util-na.c

Ο συγκεκριμένος κώδικας αρχικοποιεί τον πίνακα και συνδέεται με το fw και το fw_tiled. Ακολουθεί ο κώδικας του αρχείου.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include "util.h"
#include <omp.h>

void graph_init_random(int **adjm, int seed, int n,  int m)
{
    unsigned  int i, j;

    srand48(seed);
    #pragma omp parallel shared(n,adjm) private(i,j)
    {
      #pragma omp for schedule(static,128)
```

```
      for(i=0; i<n; i++)
            for(j=0; j<n; j++)
                  adjm[i][j] = abs((( int)lrand48()) % 1048576);


      for(i=0; i<n; i++)adjm[i][i]=0;
   }
}
```

Util-sr.c

Ο συγκεκριμένος κώδικας αρχικοποιεί τον πίνακα για την περίπτωση της αναδρομικής εκτέλεσης και συνδέεται με το fw_sr. Ακολουθεί ο κώδικας του αρχείου.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include "util.h"
#include <omp.h>

void graph_init_random(int **adjm, int seed, int n,  int m)
{
   unsigned  int i, j, k;

   srand48(seed);

   #pragma omp parallel shared(n,adjm) private(i,j)
   {
     #pragma omp for schedule(static,1)
     for(k=0; k<4; k++)
     {
       if(k==0) {
       for(i=0; i<n/2; i++)
           for(j=0; j<n/2; j++)
                 adjm[i][j] = abs((( int)lrand48()) % 1048576);
       }

       else if(k==1) {
           for(i=n/2; i<n; i++)
           for(j=0; j<n/2; j++)
                 adjm[i][j] = abs((( int)lrand48()) % 1048576);
       }

       else if(k==2) {
       for(i=0; i<n/2; i++)
```

```
            for(j=n/2; j<n; j++)
                  adjm[i][j] = abs((( int)lrand48()) % 1048576);
        }

        else {
        for(i=n/2; i<n; i++)
            for(j=n/2; j<n; j++)
                  adjm[i][j] = abs((( int)lrand48()) % 1048576);
        }

      }
      for(i=0; i<n; i++)adjm[i][i]=0;
    }
}
```

Στην αναδρομική περίπτωση ακολουθήθηκε διαφορετικός τρόπος αρχικοποίησης από τα νήματα, ακολουθώντας τη συμμετρία που δημιουργεί η αναδρομή. Ο πίνακας μοιράζεται σε 4 κομμάτια (πάνω αριστερά, πάνω δεξιά, κάτω αριστερά, κάτω δεξιά).

Σημείωση:

Παρατηρήθηκε πως το αίτημα δέσμευσης μνήμης με την εντολή malloc() δεν ακολουθούνταν από αντίστοιχη εντολή free() ώστε να αποδεσμευτεί η μνήμη που είχε δεσμευτεί από τα στοιχεία του πίνακα που εγγράφονται εκεί. Έτσι η λειτουργικότητα αυτή προστέθηκε με τον κάτωθι κώδικα:

```
for(i=0; i<N; i++) free(A[i]);
    free(A);
```

(γ) Δομές παράλληλου προγραμματισμού OpenMP

Στην περίπτωση της tiled υλοποίησης παρατηρήθηκε ότι οι πολλοί βρόχοι for δημιουργούν μεγάλες ανάγκες για συγχρονισμό, τον οποίον όμως παρακάμπτουμε με την τοποθέτηση των “nowait”, εκεί που μπορούσαμε. Αυτό μας οδηγεί σε έναν πιο συντηρητικό γράφο από αυτόν του task graph που κατασκευάσαμε στη φάση της σχεδίασης του παράλληλου προγράμματος. Το fork-join μοντέλο του εργαλείου OpenMP δε μας επιτρέπει να περιγράψουμε πλήρως το γράφο.

Στην προηγούμενη (ενδιάμεση) αναφορά χρησιμοποιήθηκε dynamic scheduling αλλά παρατηρήθηκε σημαντικό overhead λόγω αυξημένων απαιτήσεων bookkeeping και συγχρονισμού. Με δεδομένο όμως ότι το κάθε νήμα έχει το ίδιο φορτίο με τα υπόλοιπα, κρίθηκε φρόνιμο να χρησιμοποιηθεί static scheduling.

Με το static scheduling και τον ορισμό του chunk size ίσο με 1, εξασφαλίζουμε ότι υπάρχει ίσος καταμερισμός φορτίου και διαμοιρασμός στους

εργάτες-επεξεργαστές. Ορισμός του chunk size ίσο με 1 σημαίνει ότι κάθε νήμα αναλαμβάνει την εκτέλεση μιας γραμμής από block στην tiled και την εκτέλεση μιας γραμμής του πίνακα στην απλή. Αυτό πλησιάζει τη data centric λογική, όπου σκεφτόμαστε το πώς μοιράζουμε τη μνήμη.

Σύγκριση των 4 tiled υλοποιήσεων

parlab01@sandman:~/fw-par$ **./fw_tv2** 4096 64
FW_TILED,4096,64,**2.4399**
parlab01@sandman:~/fw-par$ **./fw_tiled** 4096 64
FW_TILED,4096,64,**3.1403**
parlab01@sandman:~/fw-par$ **./fw_cl3** 4096 64
FW_TILED,4096,64,**2.4455**
parlab01@sandman:~/fw-par$ **./fw_tv3** 4096 64
FW_TILED,4096,64,**2.6739**

Στην πρώτη (και καλύτερη από πλευράς αποτελεσμάτων) υλοποίηση της tiled έκδοσης (**fw_tv2.c**) σε κάθε στιγμιότυπο τα νήματα υπολογίζουν κατ' αρχήν όλα τα πλακίδια του “σταυρού” και μόλις οι υπολογισμοί αυτοί ολοκληρώνονται και τα νήματα συγχρονίζονται, ακολουθούν οι υπολογισμοί των πλακιδίων όλης της υπόλοιπης περιοχής του πίνακα για να ακολουθήσει και πάλι συγχρονισμός εν όψει του υπολογισμού του επόμενου στιγμιοτύπου του πίνακα. Η υλοποίηση αυτή αποδείχθηκε η βέλτιστη από τις 4 συνολικά υλοποιήσεις τις οποίες δημιουργήσαμε για την περίπτωση της tiled έκδοσης του αλγόριθμου καθώς περιορίζει το overhead του book-keeping και του συγχρονισμού στα ελάχιστα (σε σχέση με τις υπόλοιπες εκδόσεις) επίπεδα.

Στην έκδοση **fw_tv2_cl3.c** προσπαθήσαμε να επιτύχουμε την μέγιστη δυνατή παραλληλία δημιουργώντας και ένθετη παράλληλη περιοχή (μέσα στον διπλό βρόχο for) για τον υπολογισμό των πλακιδίων του υπόλοιπου (εκτός του “σταυρού”) τμήματος του κώδικα. Αυτό αποδείχθηκε ότι οδηγεί σε επιπλέον κόστος συγχρονισμού και book-keeping καθώς νέα νήματα δημιουργούνται συνεχώς οπότε το όφελος από την επιπλέον παραλληλία ακυρώνεται και οι χρόνοι που λάβαμε είναι ελαφρώς χειρότεροι σε σχέση με την προηγούμενη περίπτωση.

Η **fw_tv3.c** είναι η παράλληλη έκδοση του αρχικού σειριακού κώδικα του tiled αλγόριθμου (χωρίς τροποποίηση της δομής του αρχικού σειριακού κώδικα δηλαδή) με την προσθήκη της δομής “collapse” η οποία ενώνει τους 2 φωλιασμένους βρόχους for που υπάρχουν κάτω από το σημείο τοποθέτησής της. Δεν είχαμε αρκετά καλύτερα αποτελέσματα στην περίπτωση αυτή σε σχέση με την fw_tv2.c(βέβαια στην fw_tiled που μόνο το collapse τις διαφοροποιούσε έκανε

σημαντική διαφορά όπως φαίνεται από τα νούμερα παραπάνω) αλλά αυτό είναι implementation specific και δεν μπορούμε να κάνουμε κάτι περισσότερο. Σημειώνεται ότι για την εκτέλεση αυτής της περίπτωσης θέσαμε την OMP_NESTED true.

Η **fw_tiled.c** είναι η παράλληλη έκδοση του αρχικού σειριακού κώδικα του tiled αλγόριθμου (χωρίς τροποποίηση της δομής του αρχικού σειριακού κώδικα δηλαδή) χωρίς την προσθήκη της λειτουργίας “collapse” που είχαμε πριν. Είναι δηλαδή η πιο προφανής και κλασική υλοποίηση του αρχικού σειριακού κώδικα που θα μπορούσε να γίνει. Από άποψη απόδοσης σε απόλυτο χρόνο “υποφέρει” (όπως και η **fw_tv3.c**) σε σχέση με την βέλτιστη πρώτη υλοποίηση του κώδικα **fw_tv2.c** επειδή η προσπάθεια να μοιραστούν οι εργασίες μέσω περισσότερων βρόχων for οδηγεί σε αυξημένο κόστος book-keeping και αυτό παρά την προσπάθεια μέσω εντολών nowait να περιοριστεί το κόστος του συγχρονισμού στα απολύτως απαραίτητα επίπεδα).

δ. Δυνατότητες του μεταγλωττιστή

Εντός του Makefile που δημιουργήσαμε χρησιμοποιήσαμε τις σημαίες βελτιστοποίησης “O3”, “pipe” και “Ofast”, σύμφωνα με την κάτωθι εντολή:

```
CFLAGS= -Wall -O3 -pipe -Ofast -Wno-unused-variable -fopenmp
```

5. Πραγματοποιήστε μετρήσεις για τις καλύτερες παράλληλες εκδόσεις, για μεγέθη πινάκων 1024x1024 , 2048x2048 και 4096x4096, για 1, 2, 4, 8, 16, 32 και 64 threads στο μηχάνημα sandman.

Σχόλια μετρήσεων

Όσον αφορά την παράλληλη υλοποίηση της στάνταρ εκδοχής του αλγόριθμου, παρατηρούμε ότι για τα μεγέθη πίνακα **1024** και **2048** η “απλή” έκδοση κλιμακώνει μέχρι και τα **64** νήματα ενώ για μέγεθος πίνακα **4096** η έκδοση αυτή κλιμακώνει μέχρι τα **32** νήματα. Βέβαια ο χρόνος εκτέλεσης για την συγκεκριμένη έκδοση ξεκινάει από μεγαλύτερες τιμές σε σχέση με τις άλλες δύο εκδόσεις αλλά η βελτίωση είναι τόσο μεγάλη ώστε για μεγάλο αριθμό threads ο χρόνος εκτέλεσης για τα μεγέθη **1024** και **2048** είναι καλύτερος από την tiled (και πολύ καλύτερος απο την recursive). Η αναδρομική (recursive) έκδοση του αλγορίθμου αποδείχθηκε η πιο δύσκολη στης παραλληλοποίηση με την κλιμάκωσή της σε όλες τις περιπτώσεις να σταματάει στα **2** ή **4** νήματα κιόλας.

Στην περίπτωση του hyperthreading ο χρόνος αυξάνεται, τουλάχιστον στην tiled, κάτι το οποίο είναι φυσιολογικό να συμβαίνει αφού η tiled προσπαθεί να εκμεταλλευτεί ό,τι μπορεί με αποτέλεσμα όταν υπάρχει δημιουργία περισσότερων νημάτων εκτέλεσης από φυσικούς πόρους να γίνεται overload.

Στην περίπτωση του block size=32 η διαφορά αυτή είναι μικρή, ενώ στο 64 μεγαλύτερη.

Η επιτάχυνση και η αποδοτικότητα μετρήθηκαν σε σχέση με τον χρόνο εκτέλεσης του σειριακού προγράμματος.

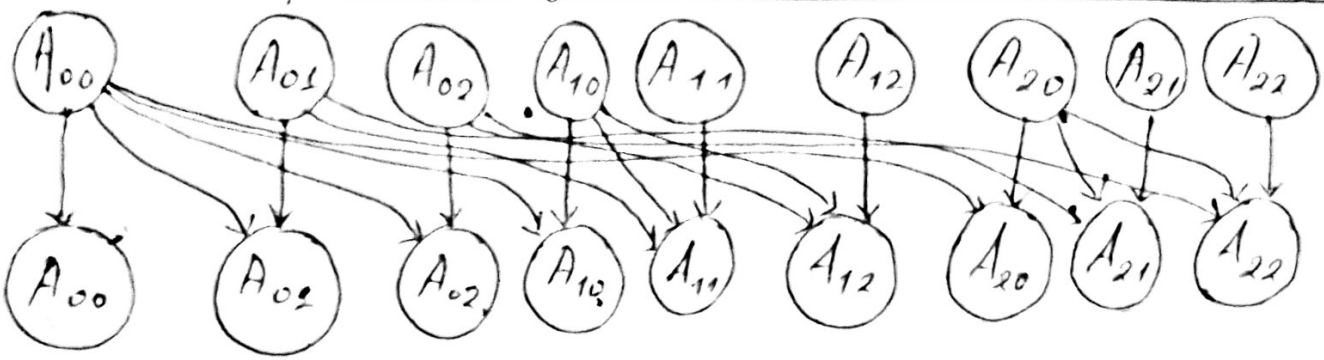
Η επιλογή αυτή έγινε επειδή στον κώδικα των άλλων δύο εκδόσεων έγιναν εκτεταμένες επεμβάσεις (εκτός από την εισαγωγή της παραλληλίας) οι οποίες έδωσαν πολύ καλύτερους χρόνους εκτέλεσης για την μονονηματική παράλληλη εκτέλεση ακόμα και σε σχέση με το αρχικό σειριακό πρόγραμμα. Το γεγονός αυτό θα οδηγούσε σε διαγράμματα που θα έδειχναν superlinear επιτάχυνση (κάτι που δεν θα έπρεπε λογικά να ισχύει). Λόγω λοιπόν των σημαντικών αλλαγών στην δομή του σειριακού κώδικα για τις περιπτώσεις της αναδρομικής και της tiled υλοποίησης επιλέχθηκε η σύγκριση να γίνει με τους αντίστοιχους χρόνους εκτέλεσης των αντίστοιχων σειριακών προγραμμάτων της νέας έκδοσης.

Οι εκτελέσεις έγιναν για δύο μεγέθη μπλοκ, 32 και 64. Όσον αφορά τον πίνακα με μέγεθος μπλοκ **32**, ο καλύτερος χρόνος εκτέλεσης για μέγεθος πίνακα **4096x4096** είναι τα **2.470 sec** και επιτεύχθηκε με την **Tiled** υλοποίηση, για αριθμό νημάτων **32**.

Με την εκτέλεση των προγραμμάτων με μέγεθος μπλοκ **64**, ο καλύτερος χρόνος εκτέλεσης όλων των περιπτώσεων για μέγεθος πίνακα **4096x4096** (και ο καλύτερος χρόνος που επιτεύχθηκε συνολικά) είναι τα **2.442 sec** και επιτεύχθηκε με την **Tiled** υλοποίηση, για αριθμό νημάτων επίσης **32**. Επιτύχαμε δηλαδή μείωση του βέλτιστου χρόνου εκτέλεσης κατά |(2.440-5.724)|/5.724 = **57.37%** σε σχέση με τον χρόνο που είχαμε επιτύχει στην ενδιάμεση αναφορά.

Δοκιμάστηκαν κάποιες υβριδικές υλοποιήσεις όπως παραλληλοποίηση και του base case αλλά αυτές δεν πήγαν και τόσο καλά, οπότε δεν τις συμπεριλάβαμε στην αναφορά.

Γράφος εξαρτήσεων για τη σειριακή έκδοση fw.c



Αρχικές τιμές

$k = 0$

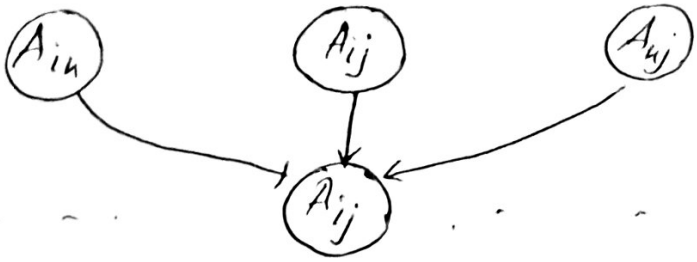
...

$k = 1$

...

$k = 2$

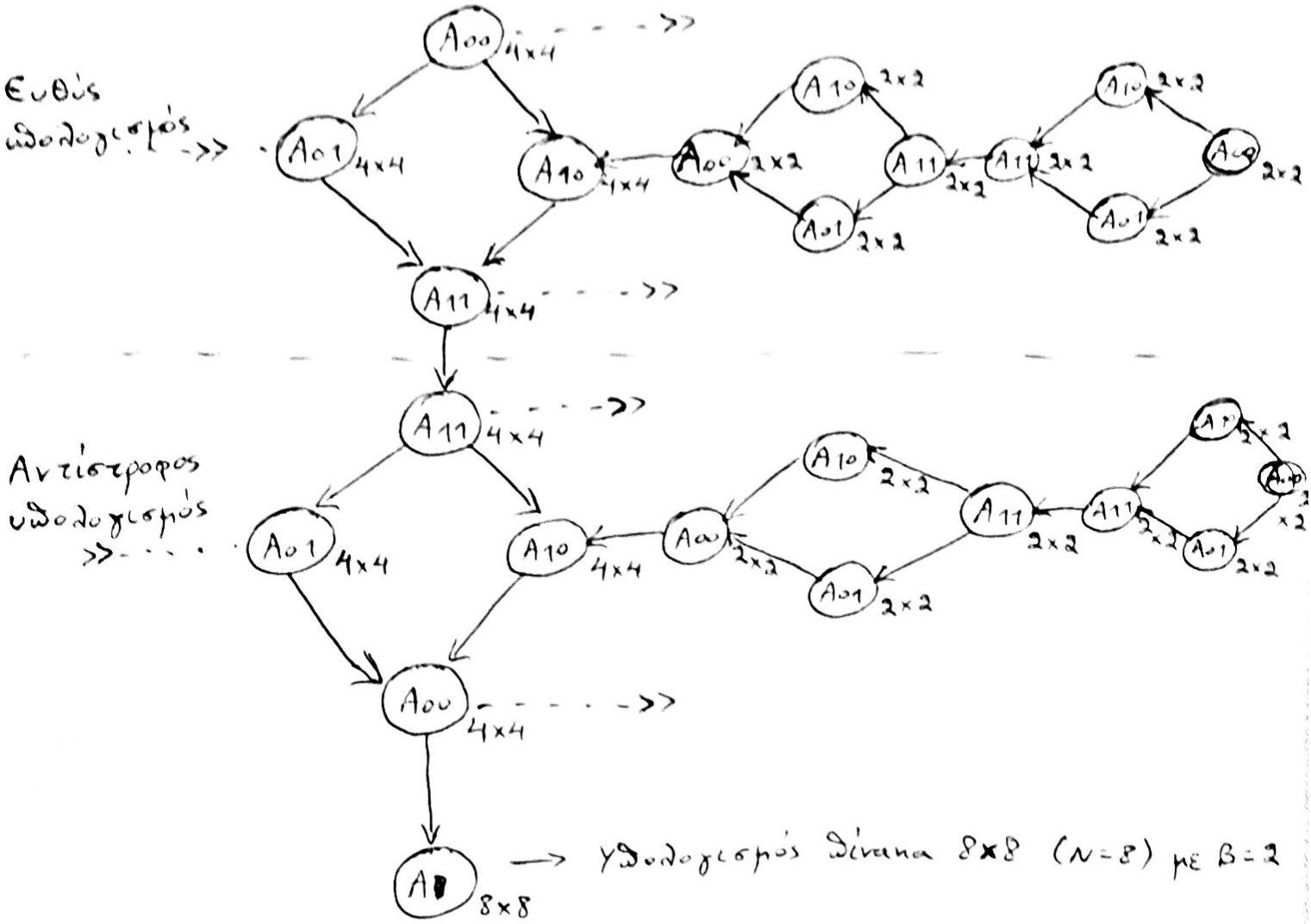
Γενικά: (Θεωρούμε την παρακάτω διάταξη των A_{iu}, A_{ij}, A_{uj})



$k-1$

k

ΕΞΑΡΤΗΣΕΙΣ ΑΝΑΔΡΟΜΙΚΗΣ ΠΕΡΙΠΤΩΣΗΣ

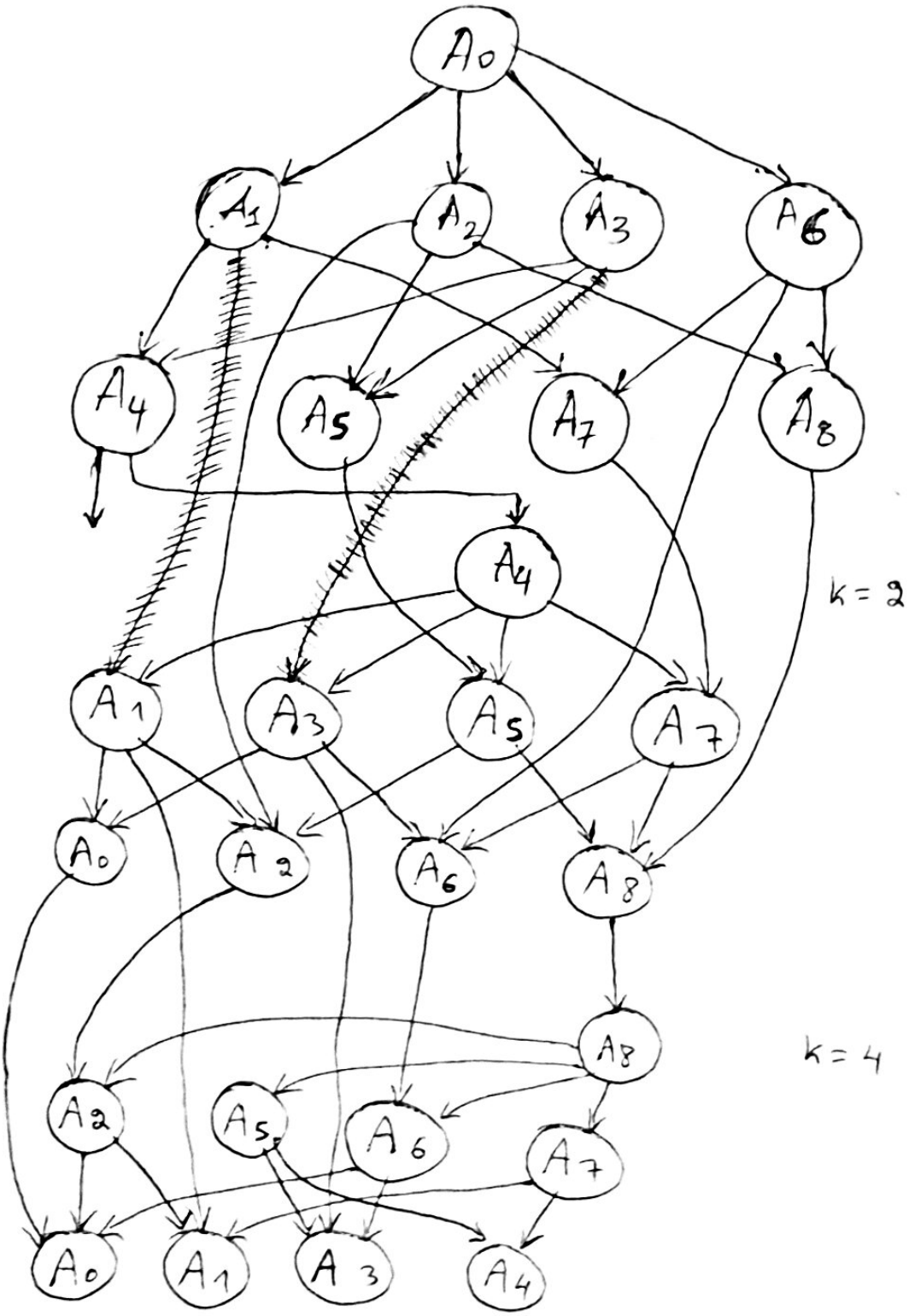


	1	2	3	4	5	6
1						
2	A_0		A_2		A_9	
3						
4	A_3		A_4		A_5	
5						
6	A_6		A_7		A_8	

$\begin{cases} k=0 \\ k=2 \\ k=4 \end{cases}$

$N=6$

$B=2.$



$k=0$

$k=2$

$k=4$

Tiled version