Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Συστήματα Παράλληλης Επεξεργασίας

9ο εξάμηνο

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

Δαζέα Ελένη 03114060
Καναβάκης Ελευθέριος 03114180

Μέρος 1.

Στο μέρος αυτό υλοποιήσαμε παράλληλες εκδόσεις για τις τρεις διαφορετικές υλοποιήσεις του αλγορίθμου Floyd-Warshall . Οι παράλληλες εκδόσεις αυτές έγιναν με την βοήθεια του εργαλείου Openmp . Στην συνέχεια παραθέτουμε τους κώδικες των παράλληλων εκδόσεων αυτών καθώς και γραφικές παραστάσεις με τις μετρήσεις που πήραμε από το μηχάνημα sandman .

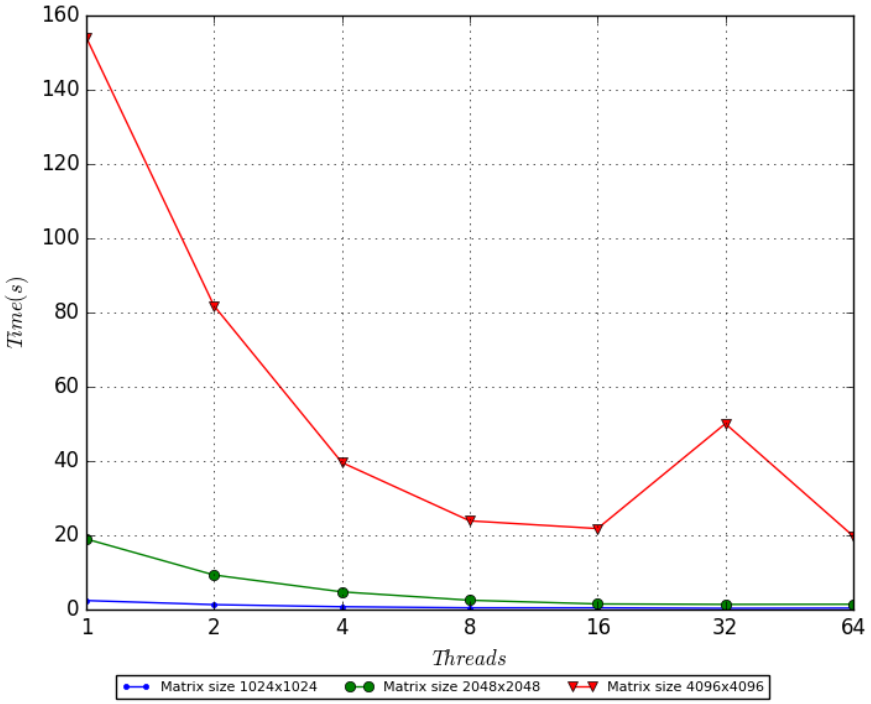
fw.c :

Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

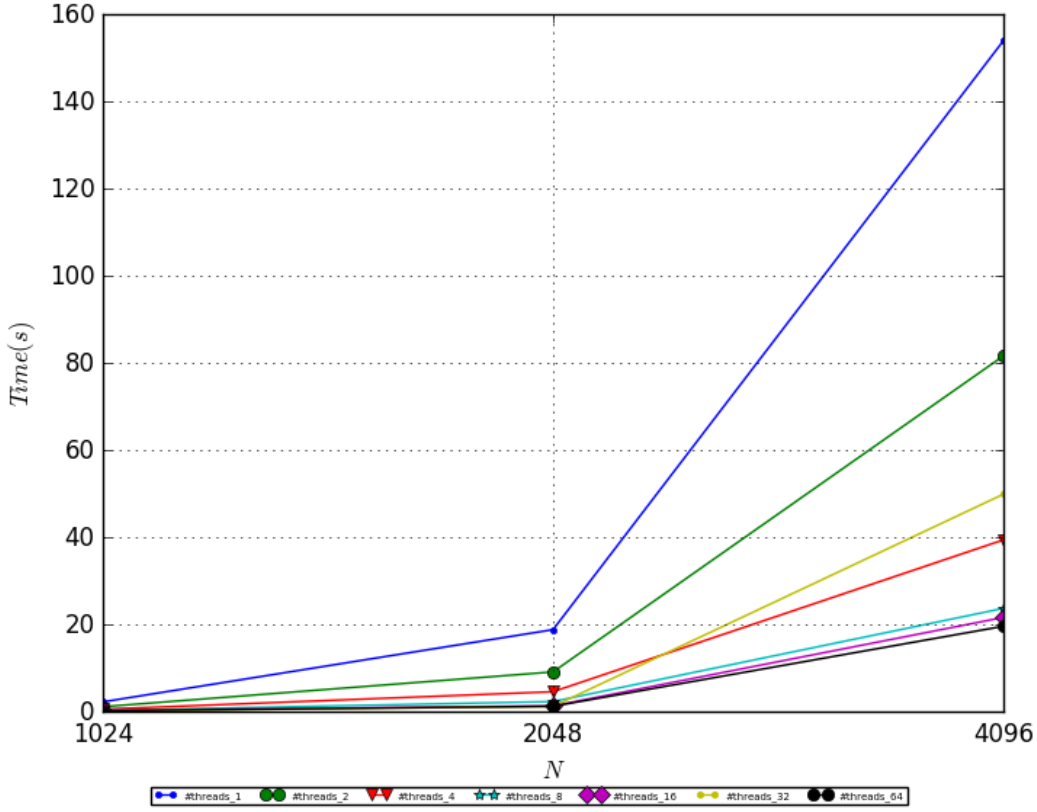
```
for(k=0;k<N;k++) {

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

        if ( i==k ) continue ;

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

    }

}
```

Στην συνέχεια παραθέτουμε γραφικές παραστάσεις για τον χρόνο καθώς και το speedup .

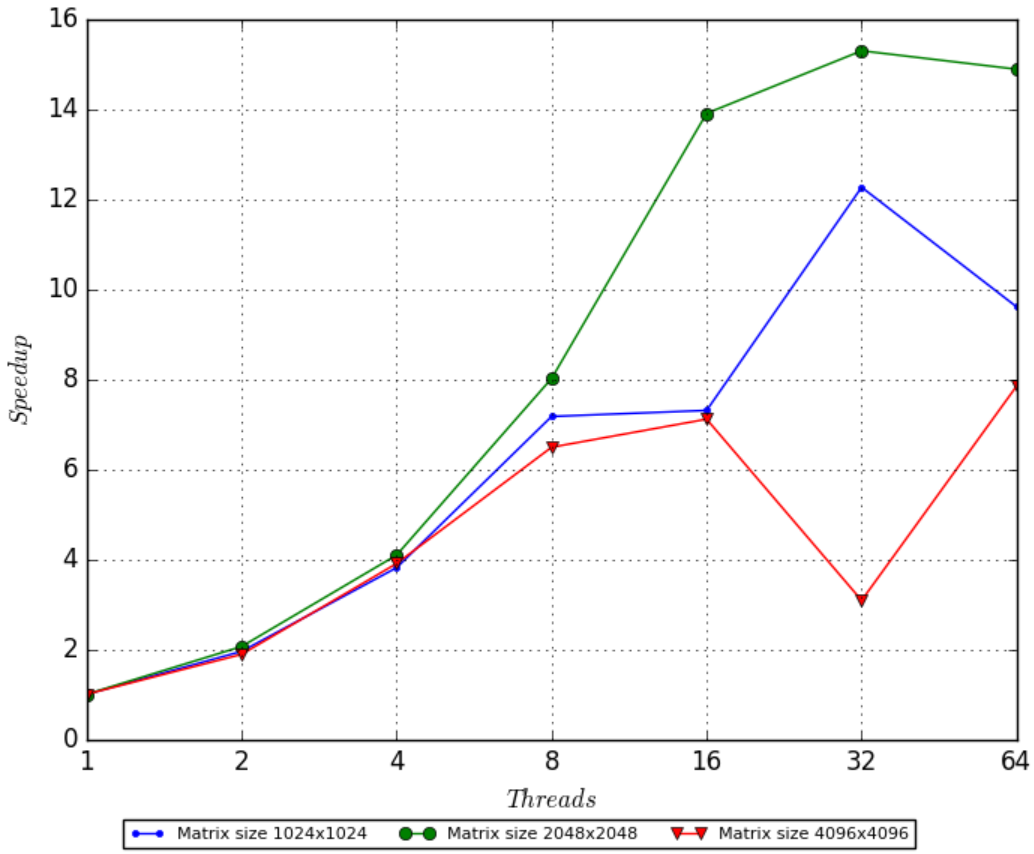
Time grouped by Matrix size :



Time grouped by number of threads :



Speedup :



Τέλος , παραθέτουμε αναλυτικά τους χρόνους για αριθμό νημάτων ίσο με 64.

```
#threads = 64

----------------------------------------

FW,1024,0.2368
FW,2048,1.2670
FW,4096,19.6297

----------------------------------------
```

Σχολιασμός :

Στον σειριακό αλγόριθμο προσθέσαμε απλά ένα parallel for για τα δύο εσωτερικά loop που μπορούν να παραλληλλοποιηθούν. Στην αρχή είχαμε ξεχωριστά τα i<k, i=k, i>k, αλλά επειδή η διαγώνιος είναι πάντα μηδέν, τα στοιχεία του σταυρού δεν αλλάζουν τιμή στο συγκεκριμένο iteration και άρα δεν υπάρχει κάποια εξάρτηση που να μας υποχρεώνει να τα κάνουμε ξεχωριστά. Βλέπουμε πως ο σειριακός, ακόμα και με μια τόσο απλή υλοποίηση κλιμακώνει όσο αυξάνουν τα thread, αφού μπορεί κάθε iteration να εκτελείται από διαφορετικό thread. Η συγκεκριμένη υλοποίηση βέβαια δεν εκμεταλλεύεται καθόλου την τοπικότητα, οπότε ο τελικός χρόνος δεν είναι τόσο καλός σε σχέση με άλλες υλοποιήσεις του αλγορίθμου Floyd-Warshall.

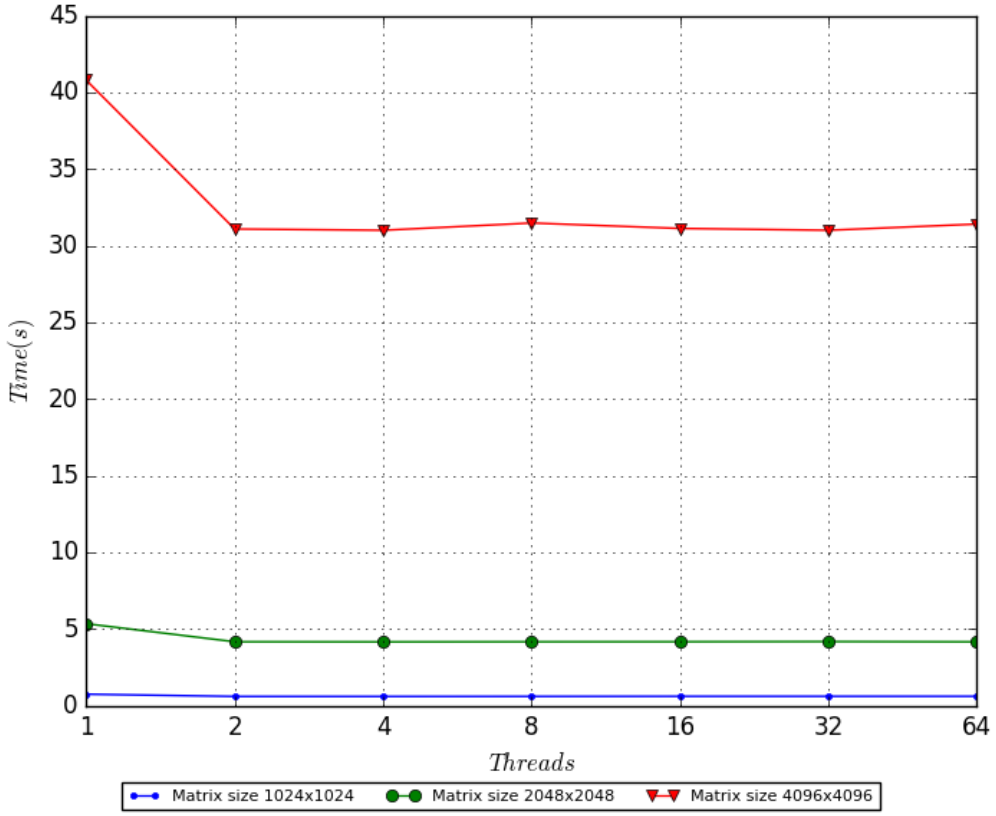
fw_sr.c :

Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

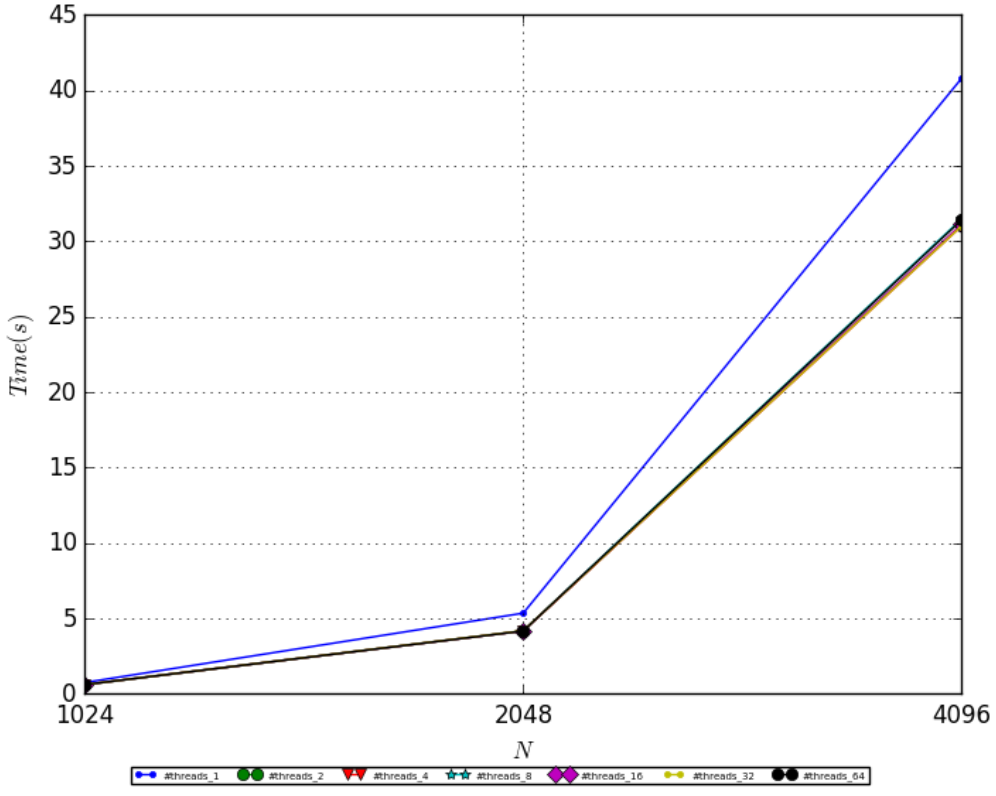
```
#pragma omp parallel
#pragma omp single
{
FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);

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

#pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN) shared(A,B,C,bsize)
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,bs
ize);

#pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN) shared(A,B,C,bsize)
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 firstprivate(arow,acol,brow,bcol,crow,ccol,myN) shared(A,B,C,bsize)
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);

}
```

Στην συνέχεια παραθέτουμε γραφικές παραστάσεις για τον χρόνο καθώς και το speedup .

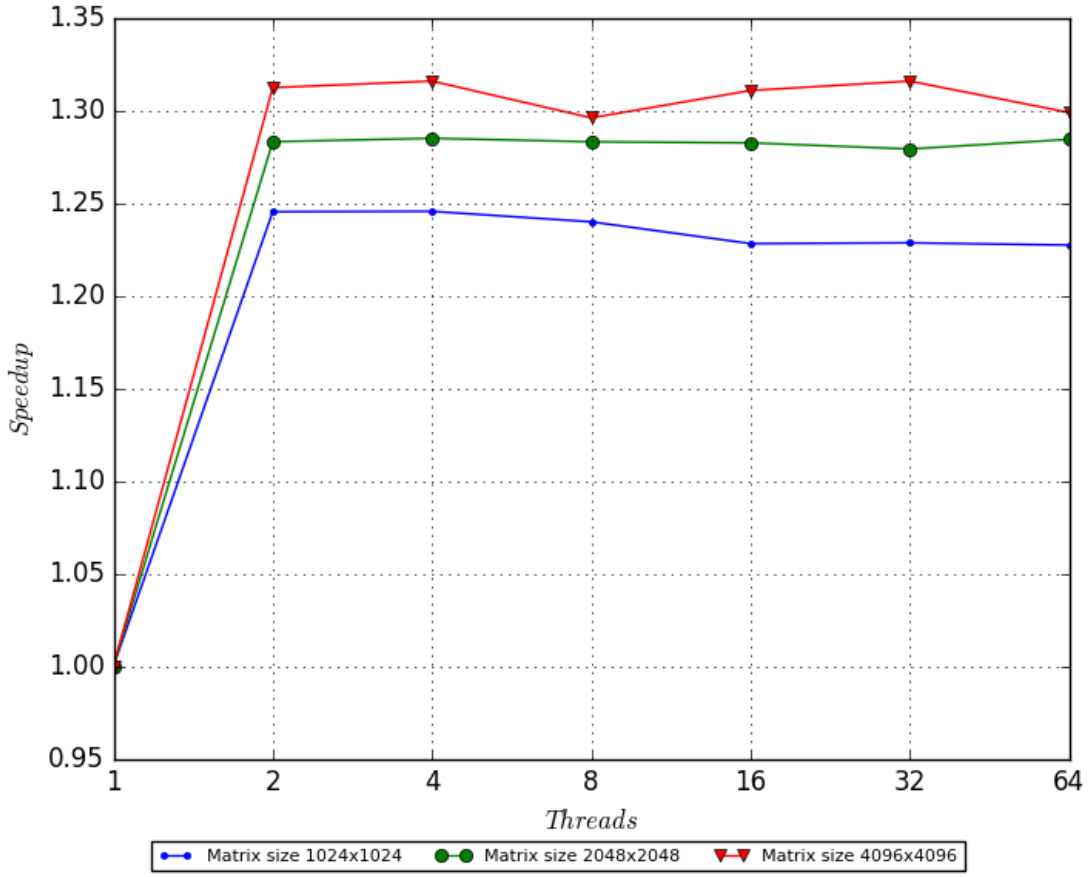
Time grouped by Matrix size :



Time grouped by number of threads :



Speedup :



Τέλος , παραθέτουμε αναλυτικά τους χρόνους για αριθμό νημάτων ίσο με 64.

```
#threads = 64

----------------------------------------

FW_SR,1024,128,0.6016
FW_SR,2048,128,4.1548
FW_SR,4096,128,31.4048

----------------------------------------
```

Σχολιασμός :

Για την παράλληλη έκδοση του αναδρομικού `Floyd-Warshall` χρησιμοποιήσαμε tasks, σύμφωνα με το task graph. Παρατηρήσαμε έπειτα από δοκιμές ότι ήταν πιο γρήγορος για B=128. Δοκιμάσαμε έπειτα και να παραλληλλοποιήσουμε το τελικό τριπλό loop αλλά επιδείνωσε την απόδοση. Παρατηρούμε τέλος πως για #threads > 2 ο αλγόριθμος δεν κλιμακώνει. Ο λόγος που συμβαίνει αυτό είναι πιθανά πως λόγω της δομής του προγράμματος, πολλά κομμάτια του κώδικα δεν γίνονται παράλληλα, όπως η

κλήση fw στην πρώτη γραμμή, και δεν μπορεί να εκμεταλλευτεί την ύπαρξη μεγάλου αριθμού νημάτων .

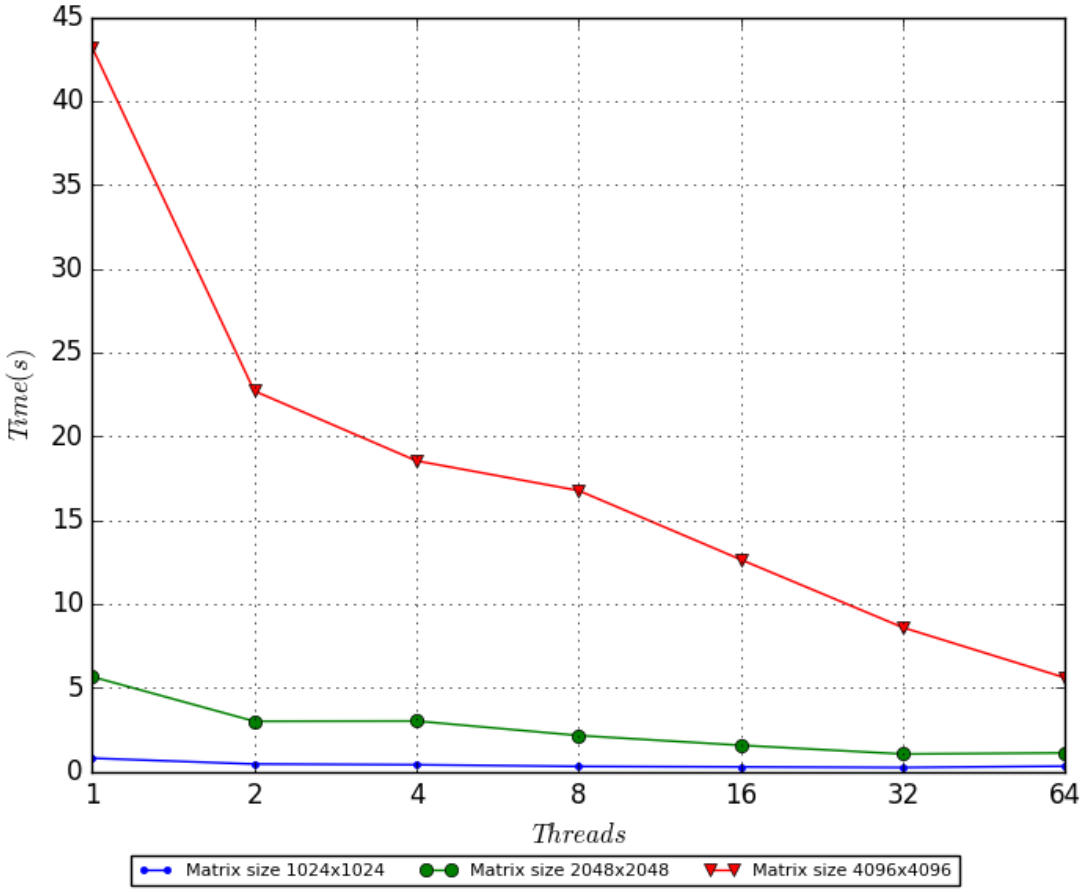
fw_tiled.c :

Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

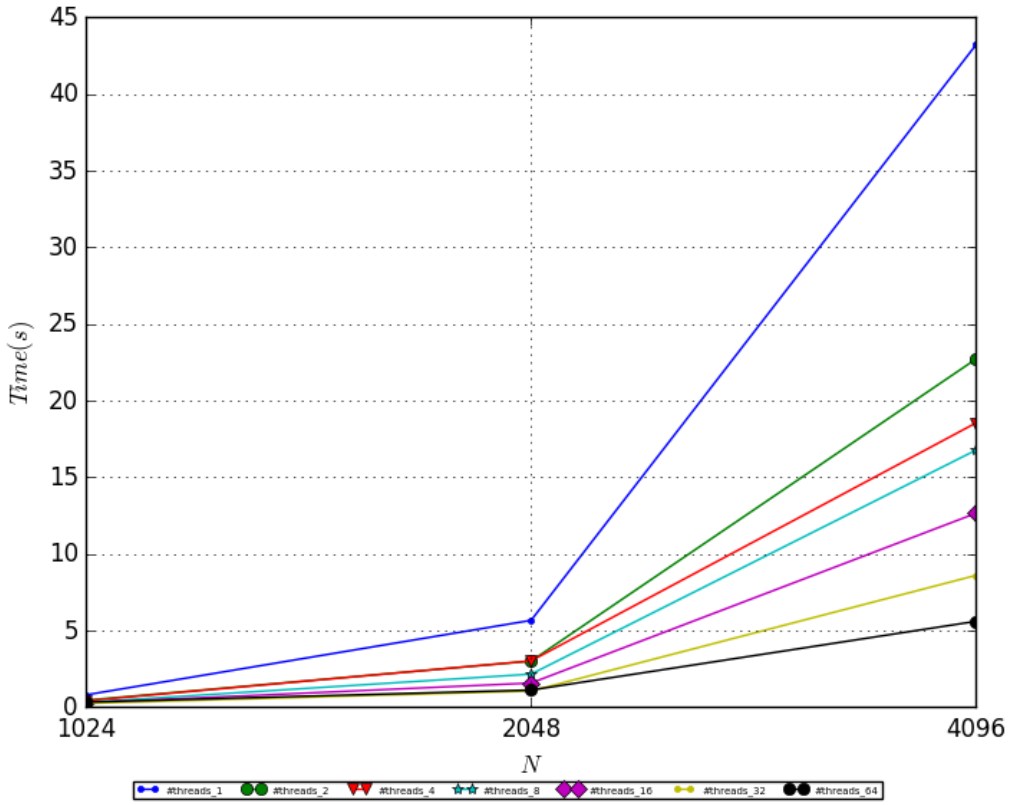
```
#pragma omp parallel
#pragma omp single
{
for(k=0;k<N;k+=B){


    #pragma omp task shared(A,k,B)
        FW(A,k,k,k,B);

    #pragma omp taskwait

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

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

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

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

    #pragma omp taskwait

    #pragma omp task shared(A,k,B) 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 task shared(A,k,B) 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 task shared(A,k,B) 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 task shared(A,k,B) 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);

    #pragma omp taskwait
    }
    }
```

Στην συνέχεια παραθέτουμε γραφικές παραστάσεις για τον χρόνο καθώς και το speedup .

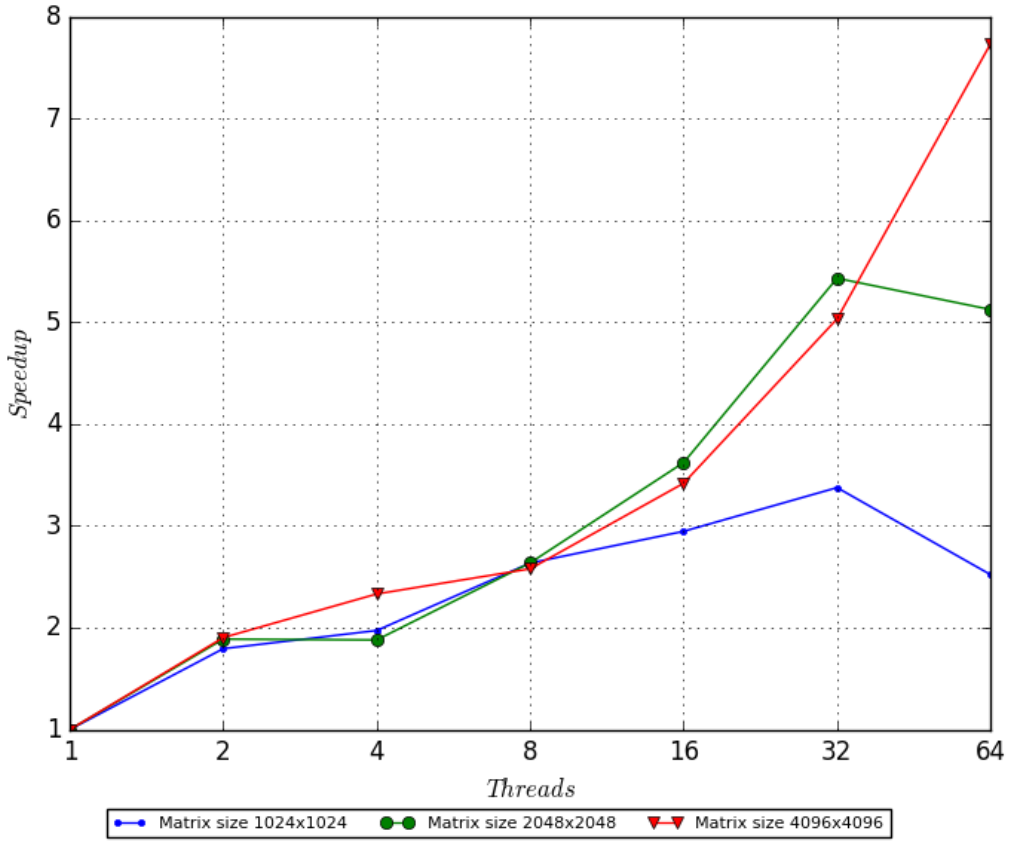
Time grouped by Matrix size :



Time grouped by number of threads :



Speedup :



Τέλος , παραθέτουμε αναλυτικά τους χρόνους για αριθμό νημάτων ίσο με 64.

```
#threads = 64

----------------------------------------

FW_TILED,1024,128,0.3134
FW_TILED,2048,128,1.1031
FW_TILED,4096,128,5.5875

----------------------------------------
```

Σχολιασμός :

Για την παράλληλη έκδοση της tiled υλοποίησης του αλγορίθμου Floyd-Warshall χρησιμοποιήσαμε και πάλι tasks. Η παράλληλη έκδοση αυτή υλοποιήθηκε σύμφωνα με το αντίστοιχο task graph που παραδώσαμε στην ενδιάμεση αναφορά. Πιο συγκεκριμένα , σύμφωνα με το task graph αυτό , εκτελείται πρώτα το μεσαίο tile, στην συνέχεια εκτελούνται παράλληλα τα tiles του σταυρού και τέλος εκτελούνται παράλληλα όλα τα υπόλοιπα tiles . Για την επιλογή της τιμής της παραμέτρου B διεξάγαμε δοκιμές . Πιο συγκεκριμένα, πήραμε διάφορες μετρήσεις από τον sandman για B ίσο με : 16,32,64,128,256,512 . Από τις μετρήσεις αυτές συμπεράναμε ότι η επίδοση της παράλληλης έκδοσης αυτής βελτιστοποιείται για B = 64. Τέλος , παρατηρώντας τις παραπάνω γραφικές παραστάσεις συμπεραίνουμε οτι ο tiled αλγόριθμος είναι πολύ πιο γρήγορος από τους άλλους δύο επειδή εκμεταλλεύεται τόσο την παραλληλία όσο και την τοπικότητα .

Μέρος 2.

Στο δεύτερο μέρος της άσκησης αυτής θα προσπαθήσουμε να βελτιστοποιήσουμε την επίδοση των παράλληλων εκδόσεων μας . Παρατηρώντας τα αποτελέσματα που παραθέσαμε παραπάνω εύκολα μπορεί να καταλάβει κανείς πως ο tiled αλγόριθμος όχι μόνο είναι ο πιο γρήγορος αλλά μάλλον είναι και αυτός που επιδέχεται τις περισσότερες βελτιώσεις . Αυτό το συμπεραίνουμε από το γεγονός ότι ο αλγόριθμος αυτός κλιμακώνει συνεχώς όσο αυξάνονται τα νήματα και πετυχαίνει τον καλύτερο χρόνο για αριθμό νημάτων ίσο με 64 σε σχέση με τις άλλες δύο εκδόσεις του αλγορίθμου Floyd-Warshall .

Αρχικά επιλέξαμε να δοκιμάσουμε να υλοποιήσουμε την παράλληλη έκδοση του tiled Floyd-Warshall με χρήση Threading Building Blocks (TBB) . Η επιλογή αυτή έγινε τόσο για να πειραματιστούμε με ένα διαφορετικό εργαλείο παράλληλου προγραμματισμού όσο και να ελέγξουμε κατά πόσο μια παρόμοια υλοποίηση σε TBB και Openmp είναι ισοδύναμη από άποψης επίδοσης . Στο σημείο αυτό πρέπει να σημειώσουμε ότι για τα TBB

αντικαταστήσαμε την κατάληξη του κώδικα μας από .c σε .cpp καθώς τα TBB λειτουργούν στην C++ !

Πριν όμως εστιάσουμε το ενδιαφέρον μας στην tiled έκδοση του αλγόριθμου Floyd-Warshall υλοποιήσαμε την recursive έκδοση αυτού με TBB tasks. Με την έκδοση αυτή πήραμε όχι μόνο αρκετά καλύτερους χρόνους εκτέλεσης αλλά παρατηρήσαμε και καλύτερη κλιμάκωση με την αύξηση του αριθμού των νημάτων .

fw_sr.cpp :

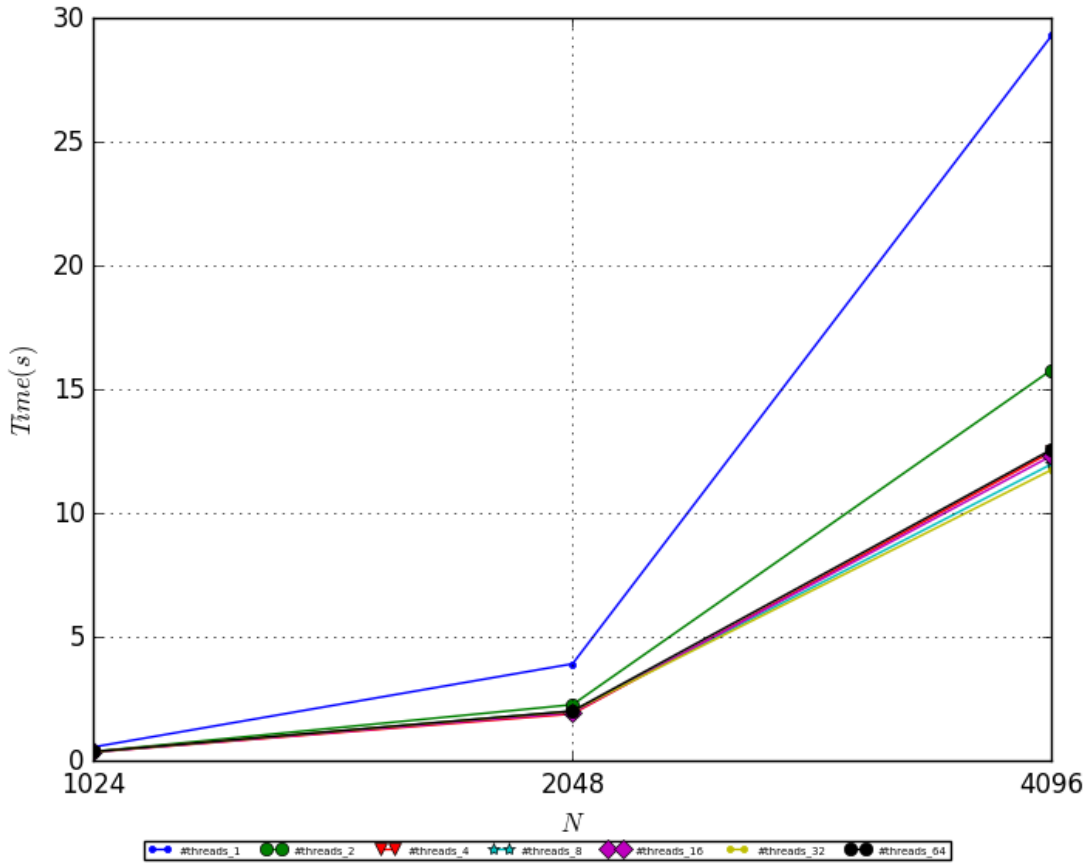
Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

```
    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++)
           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 {


        tbb::task_group g;

        g.run( [=] { FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol,
myN/2, bsize); } );
        g.wait();

        g.run( [=] { FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow,
ccol+myN/2, myN/2, bsize); } );
        g.run( [=] { FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
bcol,C,crow, ccol, myN/2, bsize); } );
        g.wait();

        g.run( [=] { FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2,
bcol,C,crow, ccol+myN/2, myN/2, bsize); } );
        g.wait();

        g.run( [=] { 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); } );
        g.wait();

        g.run( [=] {FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize); } );
        g.run( [=] { FW_SR(A,arow, acol+myN/2,B,brow,
bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize); } );
        g.wait();
```

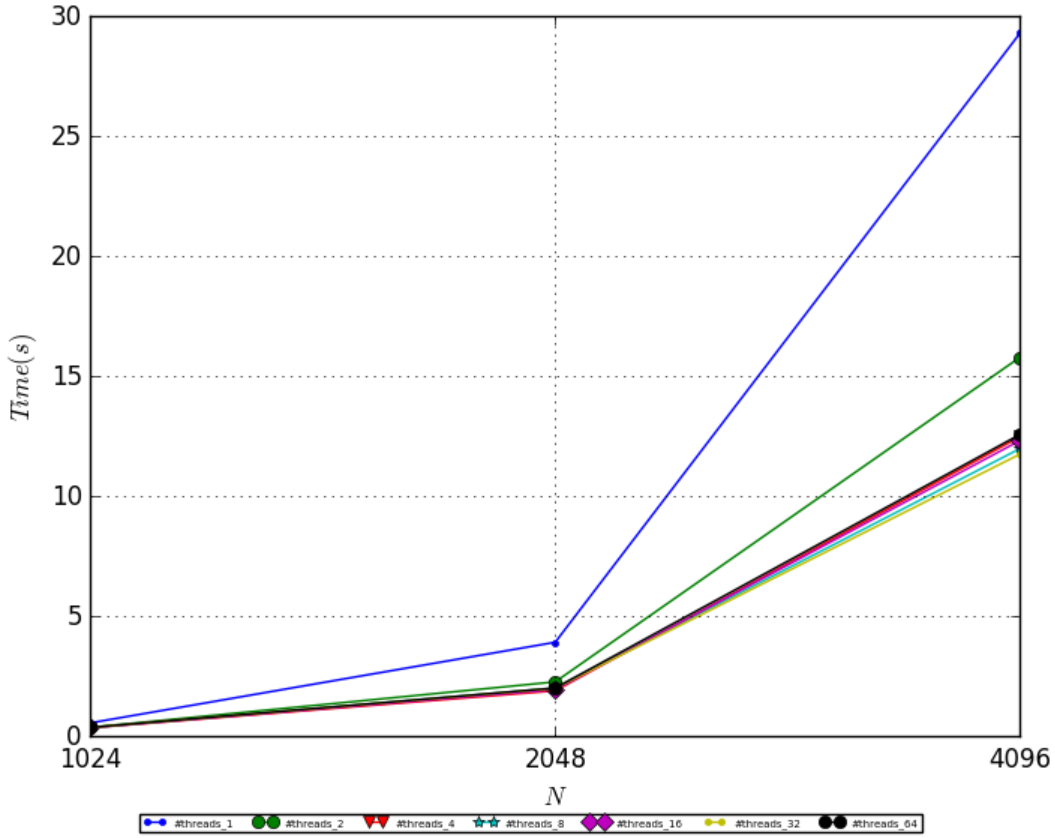
```
        g.run ( [=] { FW_SR(A,arow, acol,B,brow,
bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize); } );
        g.wait();

        }

}
```

Στην συνέχεια παραθέτουμε γραφικές παραστάσεις για τον χρόνο καθώς και το speedup .

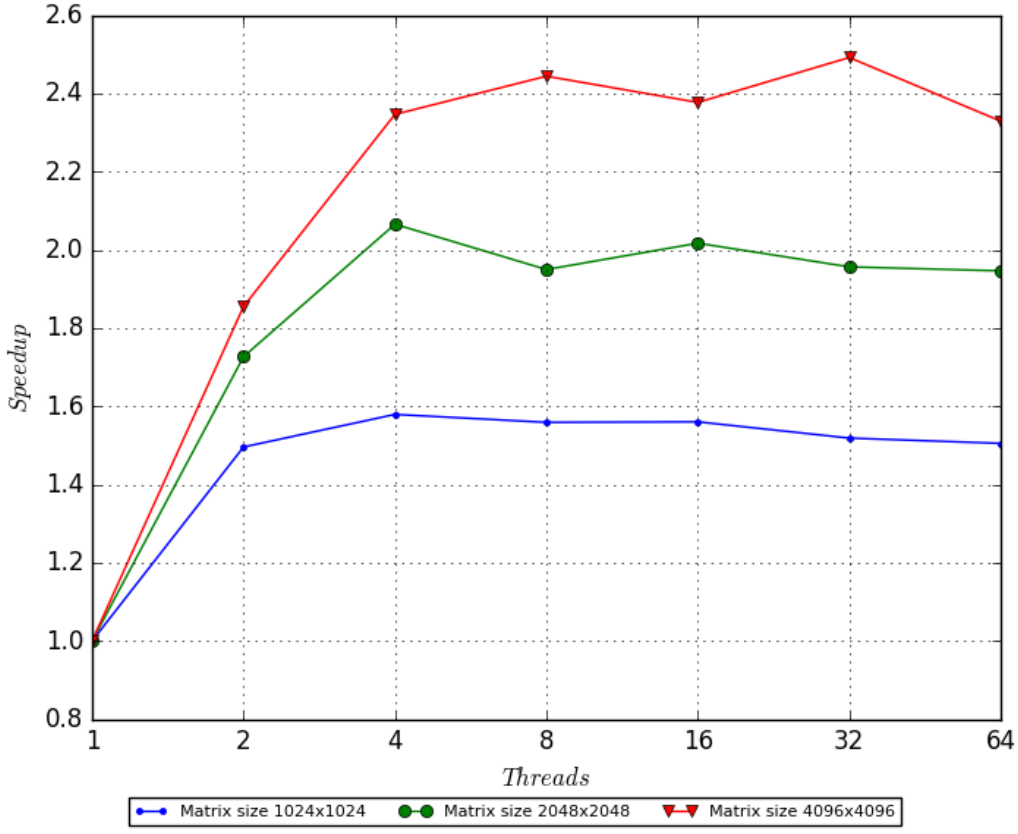
Time grouped by Matrix size :



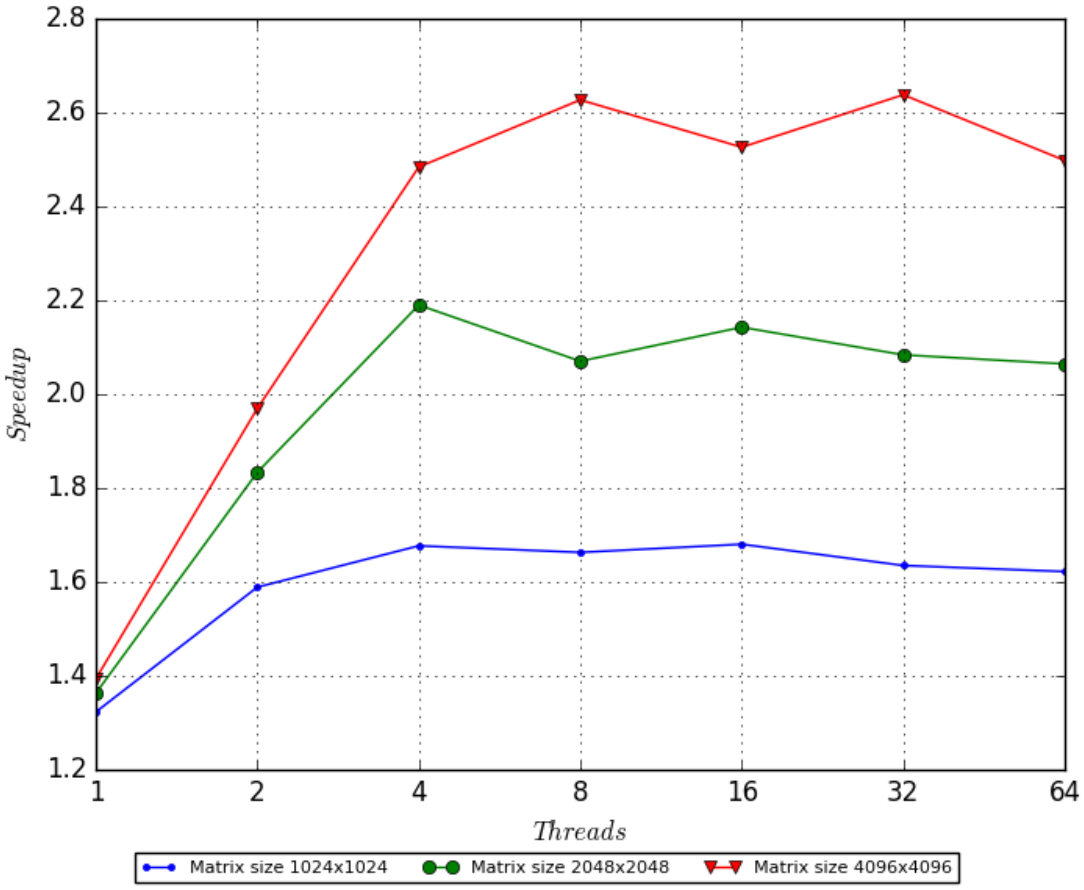
Time grouped by number of threads :



Speedup :



Στο σημείο αυτό παραθέτουμε και μια γραφική που μας δείχνει το speedup όχι όμως σε σχέση με το σειριακό χρόνο αλλά σε σχέση με τον χρόνο που έκανα το πρόγραμμα τους μέρους 1 για τον ίδιο αριθμό πυρήνων κάθε φορά.



Μετρήσεις :

```
#threads = 64

----------------------------------------

FW_SR,1024,128,0.3712
FW_SR,2048,128,2.0139
FW_SR,4096,128,12.5791

----------------------------------------
```

Από εδώ και κάτω θα ασχοληθούμε μόνο με την tiled έκδοση του αλγορίθμου FW !

fw_tiled.cpp :

Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

```
tbb::task_group g;

for(k=0;k<N;k+=B){

   g.run( [=] { FW(A,k,k,k,B); } );

   g.wait();

   for(i=0; i<k; i+=B)
      g.run( [=] { FW(A,k,i,k,B); } );

   for(i=k+B; i<N; i+=B)
      g.run( [=] { FW(A,k,i,k,B); } );

   for(j=0; j<k; j+=B)
      g.run( [=] { FW(A,k,k,j,B); } );

   for(j=k+B; j<N; j+=B)
      g.run( [=] { FW(A,k,k,j,B); } );

   g.wait();

   for(i=0; i<k; i+=B)
      for(j=0; j<k; j+=B)
         g.run( [=] { FW(A,k,i,j,B); } );

   for(i=0; i<k; i+=B)
      for(j=k+B; j<N; j+=B)
         g.run( [=] { FW(A,k,i,j,B); } );

   for(i=k+B; i<N; i+=B)
      for(j=0; j<k; j+=B)
         g.run( [=] { FW(A,k,i,j,B); } );

   for(i=k+B; i<N; i+=B)
      for(j=k+B; j<N; j+=B)
         g.run( [=] { FW(A,k,i,j,B); } );

   g.wait();

}
```

Στην συνέχεια , παραθέτουμε αναλυτικά τους χρόνους για αριθμό νημάτων ίσο με 64.

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.1889
FW_TILED,2048,64,1.0000
FW_TILED,4096,64,5.3478

----------------------------------------
```

Παρατηρούμε ότι η υλοποίηση αυτή είναι λίγο διαφορετική σε σχέση με την αντίστοιχη υλοποίηση που είχαμε κάνει με χρήση του Openmp . Η διαφορά έγκειται στο γεγονός ότι στα τελευταία 4 for loops δεν αναθέτουμε σε κάθε thread μια ολόκληρη γραμμή ή στήλη από tiles αλλά αναθέτουμε ένα tile σε κάθε thread . Η υλοποίηση αυτή πιθανώς να είναι χειρότερη από την προηγούμενη όσο αναφορά το locality αλλά βλέπουμε μια ελάχιστη βελτίωση σε σχέση με την αντίστοιχη υλοποίηση που είχαμε κάνει με Openmp . Το γεγονός αυτό δεν μας δίνει κάποιο ασφαλές συμπέρασμα σχετικά με αν τα TBB είναι πιο γρήγορα (καθώς η διαφορά είναι πολύ μικρή) αλλά μας παρακινεί να συνεχίσουμε να δουλεύουμε με τα TBB για την περαιτέρω βελτίωση της παράλληλης έκδοσης του tiled Floyd-Warshall .

Για να πετύχουμε την βελτίωση χρειάστηκε να μελετήσουμε αρκετά την τοπολογία του μηχανήματος sandman . Για την μελέτη αυτή δεν μείναμε στις πληροφορίες που μας δίνονται στης παρουσίαση της άσκησης αλλά προσπαθήσαμε να μάθουμε περισσότερα για το μηχάνημα αυτό . Για να πετύχουμε αυτό τρέξαμε δύο scripts στον sandman . Στο πρώτο εκτελέσαμε την εντολή lscpu ενώ στο δεύτερο εκτελέσαμε την εντολή cat /proc/cpuinfo . Στην συνέχεια θα παρουσιάσουμε τα αποτελέσματα της lscpu . Η άλλη εντολή θα παρουσιαστεί αργότερα όταν εισάγουμε στον κώδικα μας intrinsics .

Έτσι , με χρήση της εντολής lscpu πήραμε τις παρακάτω πληροφορίες για την τοπολογία του μηχανήματος sandman :

```
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  64
On-line CPU(s) list:     0-63
Thread(s) per core:      2
Core(s) per socket:      8
Socket(s):               4
NUMA node(s):            4
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   45
Model name:              Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20GHz
Stepping:                7
CPU MHz:                 2200.000
CPU max MHz:             2201.0000
CPU min MHz:             1200.0000
BogoMIPS:                4403.42
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                16384K
NUMA node0 CPU(s):       0-7,32-39
NUMA node1 CPU(s):       8-15,40-47
NUMA node2 CPU(s):       16-23,48-55
NUMA node3 CPU(s):       24-31,56-63
```

Από τα παραπάνω συμπεραίνουμε ότι ο sandman πρόκειται για ένα NUMA σύστημα 32 πυρήνων και 64 threads . Ο κάθε επεξεργαστής έχει μία L1 cache 32KB για δεδομένα και άλλη μία L1 cache 32KB για εντολές . Παράλληλα , ο κάθε επεξεργαστής διαθέτει μια L2 cache 256 KB . Στην cache αυτή μπορούν να χωρέσουν 16 tiles μεγέθους 64x64 καθώς ένα τέτοιο tile έχει μέγεθος $\frac{64x64x4}{1024} = 16\ KB$! Επιπρόσθετα , επειδή το μηχάνημα αυτό είναι NUMA , δεδομένα που βρίσκονται σε cache διαφορετικού κόμβου έχουν αρκετό κόστος για να προσπελαστούν . Έτσι , με χρήση των πληροφοριών αυτών θα προσπαθήσουμε να βελτιώσουμε περαιτέρω τον κώδικα μας.

Αυτό που δοκιμάσαμε να βελτιστοποιήσουμε πρώτα ήταν τα πρώτα loop. Αρχικά , ενώσαμε τα 4 loop του σταυρού σε ένα καθώς και τα επόμενα nested loops επίσης σε ένα. Στο πρώτο loop βάλαμε κάθε ένα tile της γραμμής και της στήλης στο ίδιο task . Στο nested loop , αναθέσαμε κάθε iteration του i σε διαφορετικό task για να διατηρηθεί ως ένα βαθμό η τοπικότητα στη γραμμή.

fw_tiled.cpp :

Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

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

    g.run( [=]{ FW(A,k,k,k,B); } );

    g.wait();

    for(i=0; i<N; i+=B) {

        if(i!=k) {

            g.run( [=] { FW(A,k,i,k,B);
            FW(A,k,k,i,B); } );
        }

     }

    g.wait();

    for(i=0; i<N; i+=B) {

        if(i==k) continue ;

        g.run( [=] {

            for(int j=0; j<N; j+=B) {

                if(  j!=k ) {

                    FW(A,k,i,j,B);
                }
            }
        });
    }


    g.wait();

}
```

Στην συνέχεια , παραθέτουμε αναλυτικά τους χρόνους για αριθμό νημάτων ίσο με 64.

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.1296
FW_TILED,2048,64,0.4587
FW_TILED,4096,64,1.7509

----------------------------------------
```

Παρατηρούμε , ότι πετύχαμε ραγδαία μείωση του χρόνου εκτέλεσης . Πιο συγκεκριμένα , για τον πίνακα μεγέθους 4096x4096 από 5.3 δευτερόλεπτα πέσαμε στα 1.7 δευτερόλεπτα , δηλαδή ο χρόνος εκτέλεσης μειώθηκε κατά 3.6 δευτερόλεπτα !!

Στην συνέχεια αποφασίσαμε πραγματοποιήσουμε loop unrolling στο loop που υπολογίζει τον σταυρό. Πιο συγκεκριμένα , αλλάζοντας μόνο το κόμματι που υπολογίζεται ο σταυρός δοκιμάσαμε να εκτελούμε 4 , 6 ή 8 tiles σε κάθε iteration. Παρακάτω παραθέτουμε τόσο το κομμάτι κώδικα που αλλάξαμε όσο και τις μετρήσεις που πήραμε από τον sandman .

Για 4 tiles έχουμε :

```
for(i=0; i<N; i+=2*B) {

        if(i!=k) {

             g.run( [=] { FW(A,k,i,k,B);
             FW(A,k,k,i,B); } );

         }

         if(i+B!=k) {

             g.run( [=] { FW(A,k,i+B,k,B);
             FW(A,k,k,i+B,B); } );

         }

}
```

Μετρήσεις :

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.1344
FW_TILED,2048,64,0.4569
FW_TILED,4096,64,1.7095

----------------------------------------
```

Για 6 tiles έχουμε :

```
for(i=0; i<N; i+=3*B) {


        if(i!=k) {

                g.run( [=] { FW(A,k,i,k,B);
                FW(A,k,k,i,B); } );
        }

        if( (i+B<N)  && ( i+B!=k) ) {

                g.run( [=] { FW(A,k,i+B,k,B);
                FW(A,k,k,i+B,B); } );

        }

        if( (i+(2*B)<N) && (i+(2*B))!=k) {

                 g.run( [=] { FW(A,k,i+2*B,k,B);
                 FW(A,k,k,i+2*B,B); } );

         }
}
```

Μετρήσεις :

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.1433
FW_TILED,2048,64,0.4583
FW_TILED,4096,64,1.7259

----------------------------------------
```

Για 8 tiles έχουμε :

```
for(i=0; i<N; i+=4*B) {

          if(i!=k) {

               g.run( [=] { FW(A,k,i,k,B);
               FW(A,k,k,i,B); } );
          }

          if( (i+B<N)  && ( i+B!=k) ) {

               g.run( [=] { FW(A,k,i+B,k,B);
               FW(A,k,k,i+B,B); } );

          }

          if( (i+(2*B)<N) && (i+(2*B))!=k) {

               g.run( [=] { FW(A,k,i+2*B,k,B);
               FW(A,k,k,i+2*B,B); } );

          }

          if( (i+(3*B)<N) && (i+(3*B))!=k) {

               g.run( [=] { FW(A,k,i+3*B,k,B);
               FW(A,k,k,i+3*B,B); } );

          }

}
```

Μετρήσεις :

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.1287
FW_TILED,2048,64,0.4564
FW_TILED,4096,64,1.7262

----------------------------------------
```

Από τα παραπάνω παρατηρούμε ότι τον καλύτερο χρόνο τον επιτυγχάνουμε για αριθμό tiles ίσο με 4. Το γεγονός αυτό είναι λογικό καθώς έτσι μεγιστοποιείται η τοπικότητα του κώδικα μας. Φυσικά , παρατηρούμε ότι η διαφορά σε επίδοση είναι πολύ μικρή!

Στην συνέχεια , αντικαταστήσαμε το κομμάτι στο οποίο γίνεται malloc δυσδιάστατος πίνακας A με τον scalable allocator που διαθέτουν τα TBB .

Πιο συγκεκριμένα , αυτό το επιτύχαμε με τις παρακάτω γραμμές κώδικα .

```
#include <tbb/scalable_allocator.h>

A=(int **)scalable_malloc(N*sizeof(int *));
for(i=0; i<N; i++)A[i]=(int *)scalable_malloc(N*sizeof(int));
```

Ο scalable allocator είναι ένας allocator ο οποίος βοηθάει το πρόγραμμα μας να αποφεύγονται προβλήματα που σχετίζονται με scalability bottlenecks . Πιο συγκεκριμένα , η δέσμευση μνήμης με τον allocator αυτόν γίνεται με τρόπο τέτοιο ώστε να βελτιστοποιείται η επίδοση για παράλληλα προγράμματα , ειδικά όταν πολλά threads ανταγωνίζονται για το ποιο θα δεσμεύσει κοινή μνήμη.

Μετρήσεις :

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.1143
FW_TILED,2048,64,0.3724
FW_TILED,4096,64,1.4477

----------------------------------------
```

Όπως ήταν λογικό , πήραμε αισθητή βελτίωση στον χρόνο από την προσθήκη αυτή .

Στο σημείο αυτό η βελτιστοποίηση του κώδικα μας έφτασε σε τέλμα καθώς δοκιμάσαμε διάφορα πράγματα και τα περισσότερα από αυτά είτε δεν μας έδωσαν καμία βελτίωση είτε έκαναν την επίδοση του κώδικα μας χειρότερη ! Χαρακτηριστικό παράδειγμα αποτελεί η συνάρτηση min . Παρατηρώντας την συνάρτηση min θεωρήσαμε ότι η κλήση της συγκεκριμένης συνάρτησης μπορεί να παραληφθεί . Πιο συγκεκριμένα αντικαταστήσαμε την γραμμή :

```
A[i][j]=min(A[i][j], A[i][k]+A[k][j]);
```

με τις παρακάτω γραμμές :

```
temp=A[i][k]+A[k][j];
if(A[i][j]>temp) A[i][j]=temp;
```

Μετρήσεις :

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.1397
FW_TILED,2048,64,0.4654
FW_TILED,4096,64,1.7564

----------------------------------------
```

Οι παραπάνω μετρήσεις αποτελούν απογοήτευση καθώς οι γραμμές που προσθέσαμε όχι μόνο απλοποιούν την κατάσταση αλλά θα έπρεπε να ευνοούν και το auto vectorization που πραγματοποιείται από τον compiler καθώς οι κλήσεις συναρτήσεων δυσχεραίνουν την δουλεία του compiler όσο αναφορά το vectorization . Στο σημείο αυτό , θεωρούμε απαραίτητο να αναφέρουμε ότι το auto vectorization ενεργοποιείται στον compiler με την χρήση του flag -O3 καθώς αυτό ενεργοποιεί τα flags -ftree-vectorize -ftree-vectorizer-verbose=1 . Ο λόγος λοιπόν που πήραμε το παραπάνω αποτέλεσμα υποθέτουμε ότι μάλλον έχει να κάνει με το vectorization αυτό καθώς μάλλον ο compiler καταλαβαίνει τι κάνει η συνάρτηση min ενώ όταν βλέπει τις γραμμές που προσθέσαμε μάλλον γίνεται συντηρητικός . Η συγκεκριμένη ιδέα λοιπόν ναι μεν απορρίφθηκε αλλά μας έδωσε το έναυσμα να ασχοληθούμε με την συνάρτηση FW η οποία μέχρι τώρα δεν είχε υποστεί καμία επεξεργασία. Αρχικά , θεωρήσαμε ότι δεν είναι καλή ιδέα να παραλληλοποιήσουμε το συγκεκριμένο κομμάτι κώδικα καθώς το chunk που πραγματεύεται είναι αρκετά μικρό και μάλλον η εκτέλεση του από πολλά νήματα θα επιφέρει καθυστέρηση παρά επιτάχυνση . Ως εκ τούτου , στραφήκαμε στην κατεύθυνση του vectorization . Η πρώτη μας προσέγγιση ήταν να κάνουμε μία hybrid υλοποίηση με Openmp και TBB . Ο λόγος που μας οδήγησε στην απόφαση αυτή είναι πως τα TBB , σε αντίθεση με το Openmp , δεν υποστηρίζουν εντολή για vectorization . Μάλιστα, στην ιστοσελίδα της Intel προτείνεται η συγκεκριμένη hybrid υλοποίηση .

Σε συνέχεια των παραπάνω τροποποιήσαμε την συνάρτηση FW ως εξής:

```
inline void FW(int **A, int K, int I, int J, int N)

{
      int i,j,k;

      for(k=K; k<K+N; k++)

         #pragma omp simd
         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]);

}
```

Δυστυχώς , η προσθήκη αυτή δεν προκάλεσε καμία αλλαγή στους χρόνους εκτέλεσης και έτσι αναγκαστήκαμε να στραφούμε σε κάτι πιο δραστικό . Ως εκ τούτου , στραφήκαμε προς την κατεύθυνση των Intel intrinsics . Πριν , όμως γράψουμε οποιαδήποτε γραμμή κώδικα έπρεπε να σιγουρευτούμε για το ποιες βιβλιοθήκες υποστηρίζονται από τον επεξεργαστή του sandman . Έτσι , τρέξαμε την εντολή cat /proc/cpuinfo στον sandman και τα αποτελέσματα που πήραμε φαίνονται παρακάτω :

```
flags : sse sse2 sse4_1 sse4_2 avx
```

Δυστυχώς , ο επεξεργαστής αυτός δεν υποστηρίζει AVX2 , το οποίο άρχισε να υποστηρίζεται από επεξεργαστές με haswell αρχιτεκτονική και μετά . Στο ISA του AVX2 υποστηρίζονται εντολές πρόσθεσης και σύγκρισης ακεραίων αριθμών για καταχωρητές 256 bits . Οι καταχωρητές αυτοί είναι διαθέσιμοι και στο AVX αλλά διατείθονται πράξεις για floats και όχι για integers . Έτσι , σε πρώτη φάση θα ασχοληθούμε με καταχωρητές μεγέθους 128 bits . Με το vectorization που υλοποιήσαμε ουσιαστικά εκτελούμε 16 πράξεις σε κάθε iteration του j . Δηλαδή , εκτελούμε για κάθε γραμμή 4 υπολογισμούς (4 στήλες) και συνολικά εκτελούμε το παραπάνω για 4 γραμμές ταυτόχρονα. Αρχικά , δοκιμάσαμε το παραπάνω μόνο για μια γραμμή αλλά η βελτίωση στην επίδοση θα μπορούσε να χαρακτηριστεί μικρή . Βέβαια , όταν εφαρμόσαμε το παραπάνω για 4 γραμμές τα αποτελέσματα ήταν θεαματικά. Τέλος , θεωρούμε σκόπιμο να αναφέρουμε πως για να γίνει compile ο κώδικας αυτός πρέπει να προσθέσουμε το flag -msse4.l καθώς η εντολή `_mm_min_epi32` ανήκει στο sse4.1 και όχι στο sse2 που ανήκουν οι υπόλοιπες!

Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

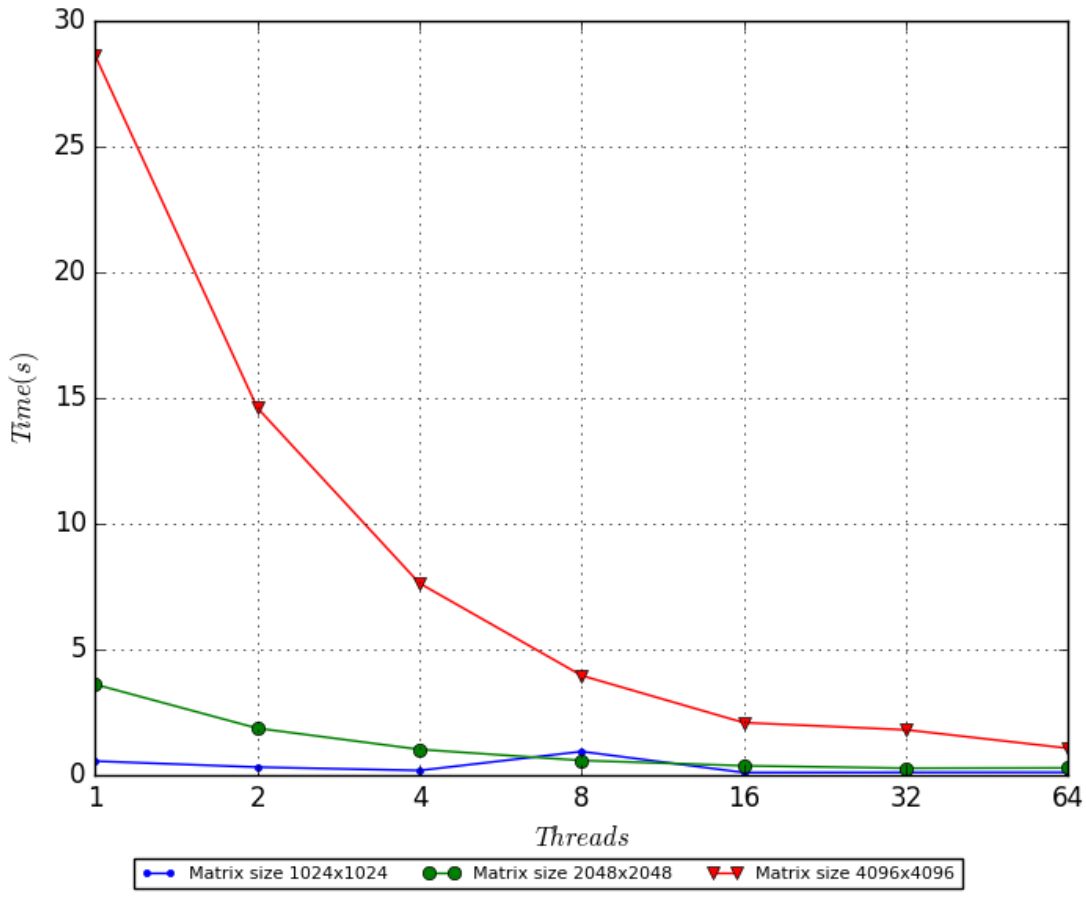
```
inline void FW(int **A, int K, int I, int J, int N) {

    int i,j,k;

    __m128i comp,Aij,Akj,Aik,Ai1k,Ai2k,Ai3k;

    for( k=K ; k<K+N ; k++) {

        for( i=I ; i<I+N ; i+=4) {

            Aik=_mm_set1_epi32(A[i][k]);
            Ai1k=_mm_set1_epi32(A[i+1][k]);
            Ai2k=_mm_set1_epi32(A[i+2][k]);
            Ai3k=_mm_set1_epi32(A[i+3][k]);

            for ( int j=J ; j<J+N ; j+=4) {


                    Akj=_mm_load_si128((__m128i*) &A[k][j]);

                    if(i!=k) {

                        Aij=_mm_load_si128((__m128i*) &A[i][j]);
                        comp =_mm_add_epi32(Aik,Akj);
                        Aij =_mm_min_epi32(Aij,comp);
                        _mm_store_si128((__m128i*) &A[i][j],Aij);

                    }

                    Aij=_mm_load_si128((__m128i*) &A[i+1][j]);
                    comp =_mm_add_epi32(Ai1k,Akj);
                    Aij =_mm_min_epi32(Aij,comp);
                    _mm_store_si128((__m128i*) &A[i+1][j],Aij);

                    Aij=_mm_load_si128((__m128i*) &A[i+2][j]);
                    comp =_mm_add_epi32(Ai2k,Akj);
                    Aij =_mm_min_epi32(Aij,comp);
                    _mm_store_si128((__m128i*) &A[i+2][j],Aij);

                    Aij=_mm_load_si128((__m128i*) &A[i+3][j]);
                    comp =_mm_add_epi32(Ai3k,Akj);
                    Aij =_mm_min_epi32(Aij,comp);
                    _mm_store_si128((__m128i*) &A[i+3][j],Aij);

            }
        }
    }
}
```

Μετρήσεις :

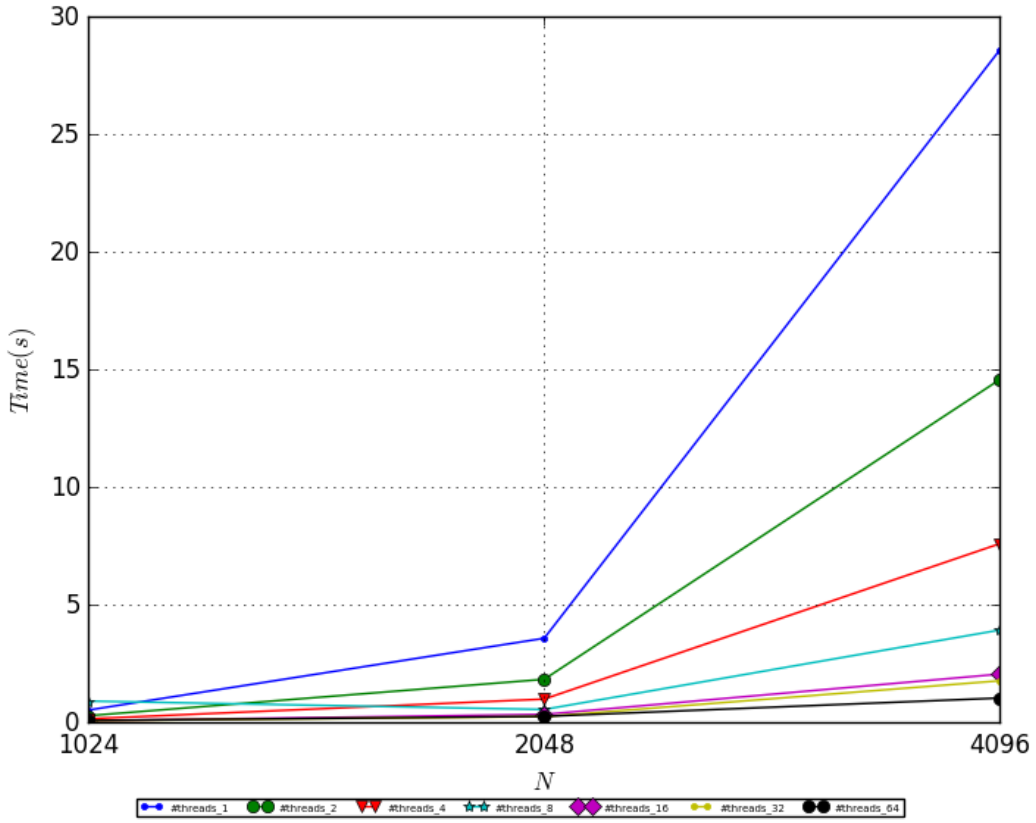
```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.0860
FW_TILED,2048,64,0.2613
FW_TILED,4096,64,1.0413

----------------------------------------
```

Στην συνέχεια παραθέτουμε γραφικές παραστάσεις για τον χρόνο καθώς και το speedup .

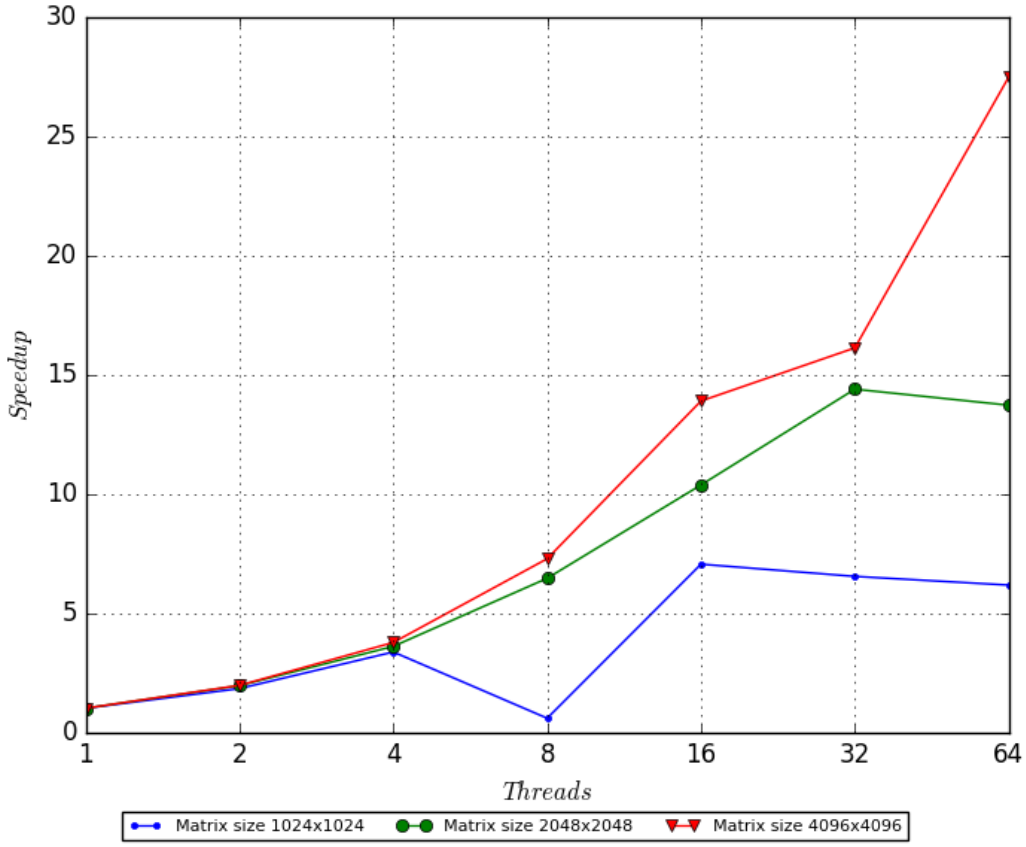
Time grouped by Matrix size :



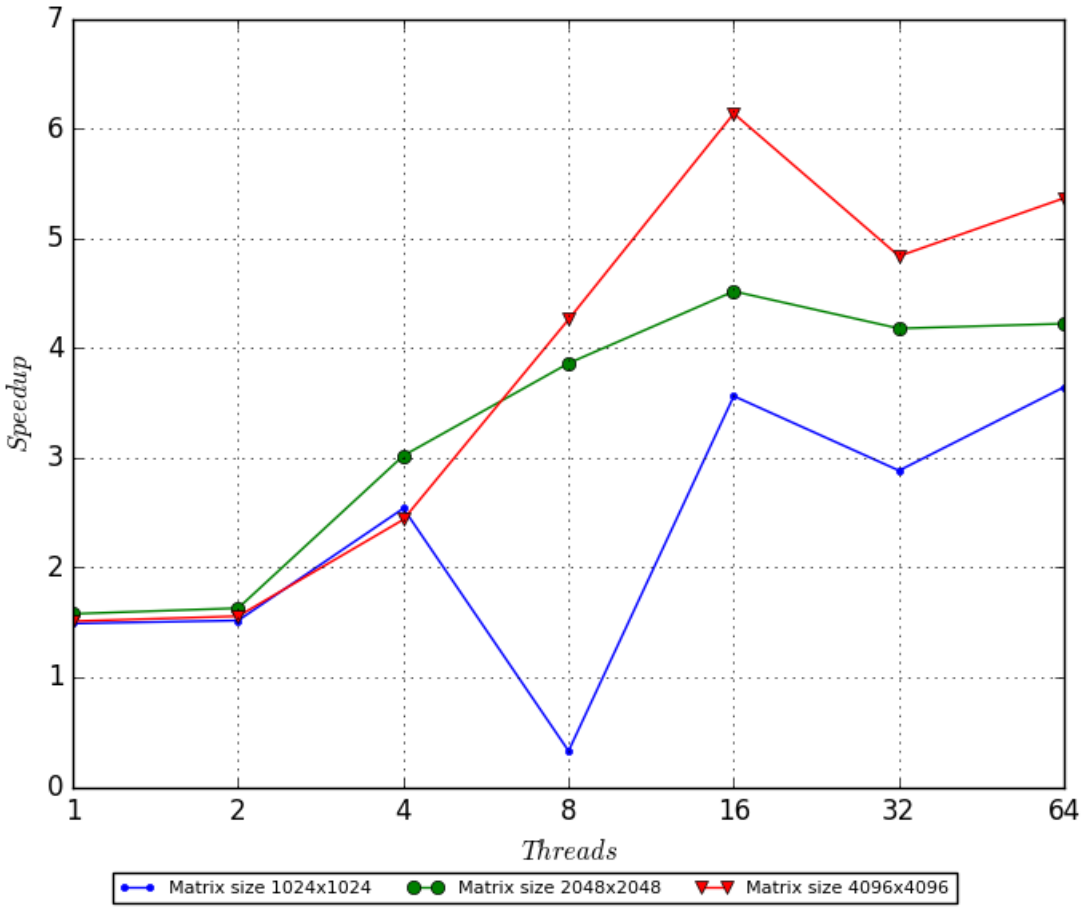
Time grouped by number of threads :



Speedup :



Στο σημείο αυτό παραθέτουμε και μια γραφική που μας δείχνει το speedup όχι όμως σε σχέση με το σειριακό χρόνο αλλά σε σχέση με τον χρόνο που έκανα το πρόγραμμα τους μέρους 1 για τον ίδιο αριθμό πυρήνων κάθε φορά.



Στο σημείο αυτό πετύχαμε τον καλύτερο χρόνο μας . Όποια προσπάθεια κάναμε από εδώ και πέρα δεν μας έδωσε καμία βελτίωση . Παρολαυτά ακόμα και αν δεν πήραμε κάποια βελτίωση θεωρούμε ενδιαφέρον να παρουσιάσουμε δύο διαφορετικές προσεγγίσεις που πραγματοποιήσαμε . Αρχικά , προσπαθήσαμε να εκμεταλλευτούμε τους καταχωρητές των 256 bit που υποστηρίζει ο επεξεργαστής μας ακόμα και αν δεν μπορούμε να κάνουμε πράξεις με ακεραίους . Για να πετύχουμε αυτό κάναμε load 8 ints και στην συνέχεια κάναμε cast τον πίνακα αυτό σε δύο πίνακες των 128 bits (έναν πίνακα με τα high και έναν πίνακα με τα low bits) . Στην συνέχεια , κάναμε τις πράξεις ξεχωριστά για τον κάθε πίνακα όπως και παραπάνω και μετά κάναμε cast τους πίνακες αυτούς και πάλι σε ένα καταχωρητή 256 bits των οποίο και κάναμε store στην μνήμη . Με την παραπάνω υλοποίηση μας επιτράπηκε να εκτελούμε 64 πράξεις σε κάθε iteration του j (αντί για 16 προηγουμένως) . Δυστυχώς το γεγονός ότι αναγκαστήκαμε να κάνουμε cast τις cast σε πίνακες 128 bits μας κόστισε σε χρόνο και έτσι ο κώδικας αυτός

κατέληξε να κάνει τον ίδιο χρόνο . Τέλος , θεωρούμε σκόπιμο να αναφέρουμε πως για να γίνει compile ο κώδικας αυτός πρέπει να προσθέσουμε το flag -mavx .

Στην συνέχεια παραθέτουμε ένα μέρος του κώδικα αυτού καθώς ο κώδικας ουσιαστικά επαναλαμβάνεται οπότε δεν εξυπηρετεί να τον παραθέσουμε ολόκληρο .

```
.
.
.
#include <emmintrin.h>
#include <immintrin.h>


#define _mm256_set_m128i(v0, v1)
_mm256_insertf128_si256(_mm256_castsi128_si256(v1), (v0), 1)

.
.
.

Akj=_mm256_load_si256((__m256i *) (&A[k][j]));
Akj1=_mm256_castsi256_si128 (Akj);
Akj2=_mm256_extractf128_si256(Akj,1);

Aij=_mm256_load_si256((__m256i *) (&A[i][j]));
Aij1=_mm256_castsi256_si128 (Aij);
Aij2=_mm256_extractf128_si256(Aij,1);

comp = _mm_add_epi32(Aik,Akj1);
Aij1=_mm_min_epi32(comp,Aij1);

comp = _mm_add_epi32(Aik,Akj2);
Aij2=_mm_min_epi32(comp,Aij2);

Aij=_mm256_set_m128i(Aij2,Aij1);

_mm256_store_si256((__m256i *) (&A[i][j]),Aij);

.
.
.
```

Τέλος , δοκιμάσαμε να υλοποιήσουμε τον κώδικα μας με την χρήση parallel for καθώς όπως είπαμε στο μάθημα τα parallel for δημιουργούν λιγότερο κώδικα στο εκτελέσιμο όποτε συνήθως ο τελικός κώδικας είναι πιο γρήγορος. Μάλιστα , δοκιμάσαμε να χρησιμοποιήσουμε τόσο τον auto_partitioner ο οποίος είναι default όσο και τον affinity_partinioner ο οποίος μοιράζει το workload με τρόπο τέτοιο ώστε να αυξάνεται το cache locality .

Ο κώδικας της υλοποίησης μας φαίνεται παρακάτω :

```
    tbb::task_group g;
    tbb::affinity_partitioner ap;
    gettimeofday(&t1,0);

     for(k=0;k<N;k+=B){

            g.run( [=]{ FW(A,k,k,k,B); } );

            g.wait();

        end=(N/(2*B));
        tbb::parallel_for(
        tbb::blocked_range<int>(0,end), [=](const
tbb::blocked_range<int>& r) {
            for(int i=r.begin() ,i_end=r.end() ,i1=i*(2*B) ; i<i_end;
i++) {
                if((i1!=k)&&(i1+B)!=k) {
                    FW(A,k,i1,k,B);
                    FW(A,k,k,i1,B);
                    FW(A,k,k,i1+B,B);
                    FW(A,k,i1+B,k,B);
                }
                else if(i1==k) {
                    FW(A,k,k,i1+B,B);
                    FW(A,k,i1+B,k,B);
                }
                else {
                    FW(A,k,k,i1,B);
                    FW(A,k,i1,k,B);
                }


            }
        },
        ap );

            end=(N/(B));
            tbb::parallel_for(
            tbb::blocked_range2d<int>(0,end,0,end) , [=](const
tbb::blocked_range2d<int>& t) {
            for(int i=t.rows().begin(),iend=t.rows().end(),i2=i*B;
i<iend ; i++) {
                for(int
j=t.cols().begin(),j_end=t.cols().end(),j2=j*B; j<j_end; j++) {

                    if( (i2 != k) && ( j2 != k ) ) {

                        FW(A,k,i2,j2,B);
                    }
                }
            }
            },
            ap);


    }
```

Μετρήσεις με auto_partinioner :

```
#threads = 64
```

```
----------------------------------------

FW_TILED,1024,64,0.0949
FW_TILED,2048,64,0.2554
FW_TILED,4096,64,1.3426

----------------------------------------
```

Μετρήσεις με affinity_partinioner :

```
#threads = 64

----------------------------------------

FW_TILED,1024,64,0.0892
FW_TILED,2048,64,0.2594
FW_TILED,4096,64,1.1345

----------------------------------------
```

Συνοψίζοντας , καταλήγουμε στο ότι τον καλύτερο χρόνο τον πήραμε με την χρήση των TBB tasks και των intrinsics . Οι **καλύτεροι** μας χρόνοι για αριθμό νημάτων ίσο με 64 είναι οι παρακάτω :

```
----------------------------------------

FW_TILED,1024,64,0.0860
FW_TILED,2048,64,0.2613
FW_TILED,4096,64,1.0413

----------------------------------------
```