ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

Άσκηση 1: Παραλληλοποίηση αλγορίθμων σε πολυπύρηνες αρχιτεκτονικές κοινής μνήμης

Χειμερινό εξάμηνο 2019-20 - Ροή Υ

Αντωνιάδης, Παναγιώτης
el15009@central.ntua.gr

Μπαζώτης, Νικόλαος
el15739@central.ntua.gr

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers."

– Gene Amdahl, *American computer architect*

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

1.1 Σκοπός την ενότητας

Η ενότητα αυτή αποτελεί το βασικό κορμό της άσκησης. Στόχος της ενότητας είναι να αναπτύξουμε διαφορετικές παράλληλες εκδόσεις του αλγορίθμου Floyd-Warshall, να αξιολογήσουμε την παραγωγικότητα (productivity) ανάπτυξης παράλληλου κώδικα και την τελική επίδοση του παράλληλου προγράμματος, επιλέγοντας ένα από τα δύο προγραμματιστικά εργαλεία για αρχιτεκτονικής κοινής μνήμης: OpenMP ή Threading Building Blocks (TBBs).

1.2 Ο αλγόριθμος Floyd-Warshall

Ο αλγόριθμος των Floyd-Warshall (FW) υπολογίζει τα ελάχιστα μονοπάτια ανάμεσα σε όλα τα ζεύγη των N κόμβων ενός γράφου (all-pairs shortest path). Θεωρώντας το γράφο αποθηκευμένο στον πίνακα γειτνίασης A, ο αλγόριθμος έχει ως εξής:

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

Εκτός από την standard έκδοση του αλγορίθμου, έχουν προταθεί άλλες δύο εκδόσεις, μία αναδρομική (recursive) και μία tiled, προκειμένου να αξιοποιείται καλύτερα η κρυφή μνήμη [1].

1.3 Σχεδιασμός παραλληλοποίησης

Ο σχεδιασμός αναπτύχθηκε ως μέρος της ενδιάμεσης αναφοράς της άσκησης.

1.4 Υλοποίηση παράλληλων προγραμμάτων

Πριν την περιγραφή της υλοποίησης των παράλληλων προγραμμάτων, παρατίθονται οι γραφικές παραστάσεις των σειριακών αλγορίθμων (tiled και recursive). Σε κάθε γραφική, ο οριζόντιος άξονας αναπαριστά το μέγεθος του block και ο κάθετος τον χρόνος εκτέλεσης του προγράμματος.

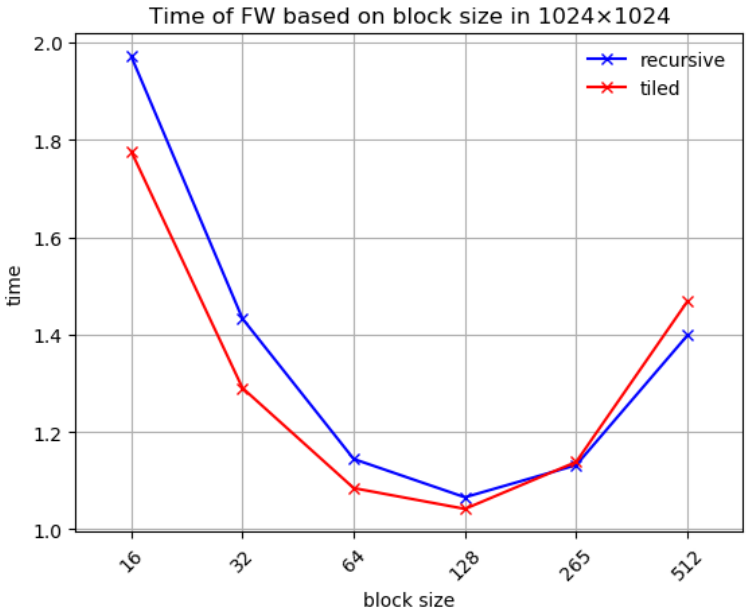

Figure 1: Χρόνοι εκτέλεσης για N=1024

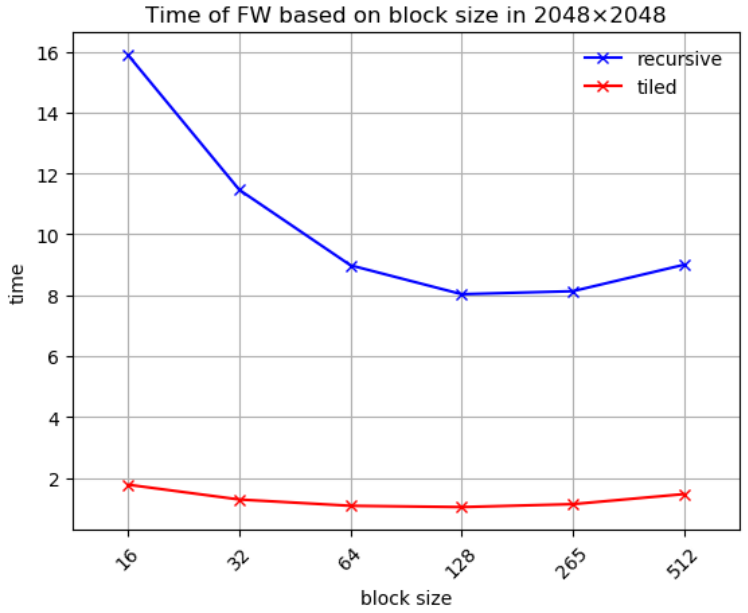

Figure 2: Χρόνοι εκτέλεσης για N=2048

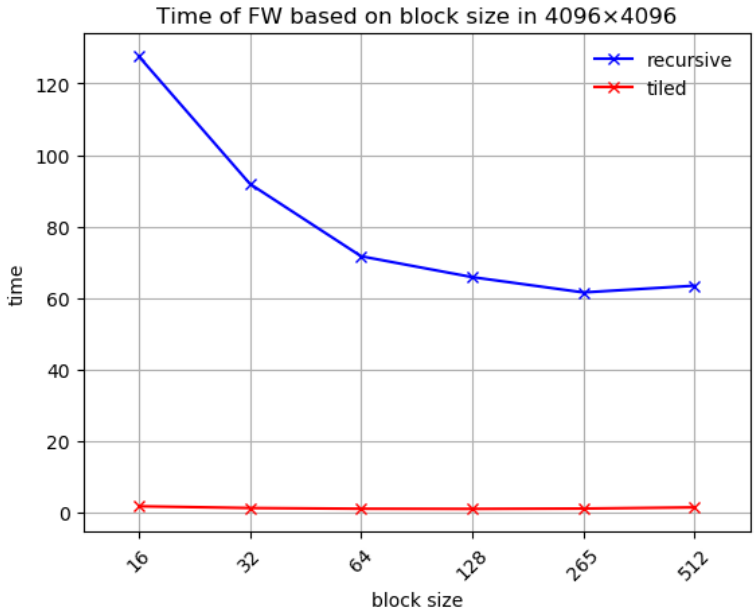

Figure 3: Χρόνοι εκτέλεσης για N=4096

1.4.1 Με χρήση των Threading Building Blocks

Τα Threading Building Blocks αποτελούν μία C++ template library για αποδοτικό και εύκολο παράλληλο προγραμματισμό σε πλατφόρμες μοιραζόμενης μνήμης. Στην περίπτωσή μας, ως βασικό συστατικό θα χρησιμοποιήσουμε 3 components της βιβλιοθήκης, το **parallel for**, τα **task groups** και τα **task graphs**. Το πρώτο προσφέρεται καθαρά και μόνο για την παραλληλοποίηση ενός loop, ενώ τα άλλα δύο βασίζονται στη δημιουργία του task dependency graph που έχει προκύψει κατά τον σχεδιασμό (δες ενδιάμεση αναφορά).

- **Παραλληλοποίηση for-loop με parallel for**

  Η τεχνική αυτή διασπάει το αρχικό range της επανάληψης και εφαρμόζει μία ανώνυμη συνάρτηση σε κάθε subrange [2]. Εφαρμόστηκε στην standard υλοποίηση του αλγορίθμου που βασίζεται στην επαναληπτική ανανέωση του πίνακα. Συγκεκριμένα, εφαρμόστηκαν οι εξής τεχνικές παραλληλοποίησης ενός for loop:

  1. Ανά γραμμή: Οι N γραμμές του ταμπλό χωρίζονται σε chunks και κάθε νήμα αναλαμβάνει την εκτέλεση ενός chunk.

```
for(k=0;k<N;k++){
    tbb::parallel_for(
    tbb::blocked_range<size_t>(0,N),
    [=](const tbb::blocked_range<size_t>& r) {
        for(size_t i = r.begin() ; i != r.end(); i++){
            for(int j=0; j!=N; j++){
                A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
```

```
            }
        }
    });
}
```

2. Ανά γραμμή με ορισμό κατωφλιού (grainsize): Οι N γραμμές του ταμπλό πάλι χωρίζονται σε chunks αλλά ορίζουμε ένα κατώφλι g, τέτοιο ώστε $g/2 \leq chunksize$.

```
for(k=0;k<N;k++){
    tbb::parallel_for(
    tbb::blocked_range<size_t>(0,N, N/nthreads),
    [=](const tbb::blocked_range<size_t>& r) {
        for(size_t i = r.begin() ; i != r.end(); i++){
            for(int j=0; j!=N; j++){
                A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
            }
        }
    });
}
```

3. Ανά γραμμή εκμεταλλεύοντας την cache (affinity_partitioner): Οι N γραμμές του ταμπλό πάλι χωρίζονται σε chunks, με τέτοιο τρόπο ώστε να εκμετταλευτούμε την τοπικότητα και να μεγιστοποιηθεί το cache affinity.

```
tbb::affinity_partitioner ap;
for(k=0;k<N;k++){
    tbb::parallel_for(
    tbb::blocked_range<size_t>(0,N),
    [=](const tbb::blocked_range<size_t>& r) {
        for(size_t i = r.begin() ; i != r.end(); i++){
            for(int j=0; j!=N; j++){
                A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
            }
        }
    }, ap );
}
```

4. Ανά block: Το N×N ταμπλό διαχωρίζεται σε block και κάθε block αποτελεί ένα chunk στο οποίο εφαρμόζεται η ανανέωση του ταμπλό.

```
for(k=0;k<N;k++){
    tbb::parallel_for(
    tbb::blocked_range2d<size_t>(0,N, 0, N),
    [=](const tbb::blocked_range2d<size_t>& r) {
        for(size_t i = r.rows().begin() ; i != r.rows().end(); i++){
            for(size_t j=r.cols().begin(); j!=r.cols().end(); j++){
                A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
            }
        }
    });
}
```

5. Ανά block με ορισμό καρωφλιού (grainisize): Το N×N ταμπλό διαχωρίζεται σε block όπως παραπάνω, αλλά εδώ ορίζουμε κατώφλι για τις διαστάσεις του block.

```
for(k=0;k<N;k++){
    tbb::parallel_for(
    tbb::blocked_range2d<size_t>(0,N, N/nthreads, 0, N, N/nthreads),
    [=](const tbb::blocked_range2d<size_t>& r) {
        for(size_t i = r.rows().begin() ; i != r.rows().end(); i++){
            for(size_t j=r.cols().begin(); j!=r.cols().end(); j++){
                A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
            }
        }
    });
}
```

Παρακάτω, βλέπουμε τις γραφικές παραστάσεις των παραπάνω μεθόδων για κάθε μέγεθος ταμπλό. Σε κάθε γραφική, ο οριζόντιος άξονας αναπαριστά αριθμό πυρήνων και ο κάθετος τον χρόνος εκτέλεσης του προγράμματος.

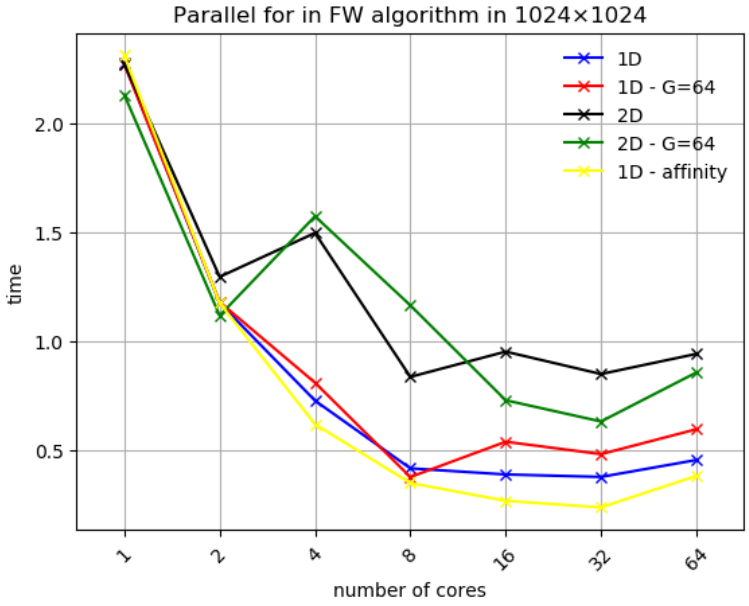

Figure 4: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=1024

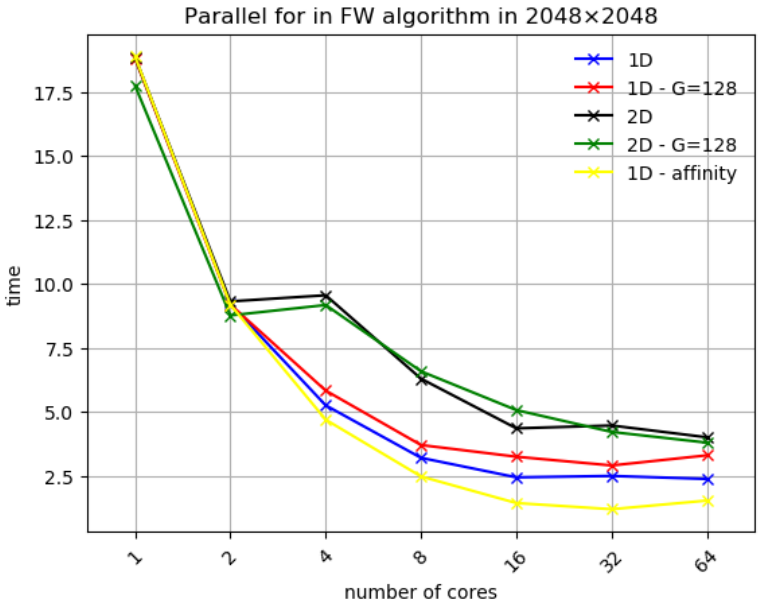

Figure 5: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=2048

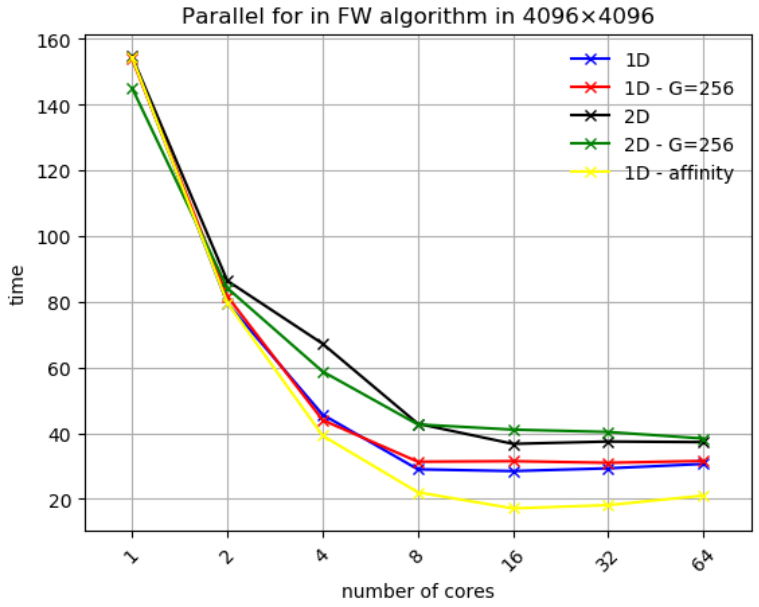

Figure 6: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=4096

Σχολιασμός:

Παρατηρούμε ότι σε όλες τις περιπτώσεις ο χωρισμός του ταμπλό ανά γραμμή είναι αποδοτικότερος από τον χωρισμό ανά block. Αυτό συμβαίνει γιατί κάθε χρονική στιγμή η ανανέωση ενός στοιχείου A_{ij} εξαρτάται από τα στοιχεία που βρίσκονται στην ίδια γραμμή A_{ik} και στην ίδια στήλη A_{kj} με αυτό. Συνεπώς, όταν χωρίζουμε το ταμπλό ανά γραμμή γνωρίζουμε ότι τουλάχιστον το στοιχείο της ίδιας γραμμής θα βρίσκεται πάντα στο ίδιο νήμα. Οπότε στη χειρότερη θα χρειαστεί επικοινωνία με άλλα νήματα για το A_{kj}. Αντίθετα, όταν χωρίζεται ανά block, η επικοινωνία των νημάτων μπορεί να είναι πολύ μεγάλη, καθώς σε ορισμένες επαναλήψεις τόσο το A_{ik} όσο και το A_{kj} θα βρίσκονται σε άλλο νήμα. Επίσης, το γεγονός ότι οι πίνακες αποθηκεύονται στη μνήμη ανά γραμμή ευνοεί τον χωρισμό του ταμπλό ανά γραμμή. Όσον αφορά την μέθοδο χωρισμού των chunks, ο affinity partitioner έδωσε το καλύτερο αποτέλεσμα καθώς επιλέγει το κατάλληλο g και εκμεταλλεύεται την τοπικότητα των δεδομένων, ελαχιστοποιώντας την επικοινωνία με την μνήμη. Συνολικά, η τεχνική **1D-affinity** είναι η καλύτερη χωρίς ωστόσο να κλιμακώνει, γεγονός που οφείλεται στην απλότητα και στους περιορισμούς που θέτει ο standard αλγόριθμος (και λύνουν οι recursive και tiled υλοποιήσεις).

Σημείωση:
Οι πρώτες 3 μέθοδοι θα μπορούσαν να εφαρμοστούν και στις στήλες του ταμπλό, αλλά από την στιγμή που οι πίνακες αποθηκεύονται στην μνήμη κατά γραμμές κάτι τέτοιο θα δημιουργούσε επιπλέον καθυστερήσεις για μεταφορά δεδομένων. Επιπλέον μέθοδοι θα μπορούσαν να προκύψουν και από την προσρμογή του partitioner αλλά οι περιορισμοί του standard αλγορίθμου δεν θα επέτρεπαν κάποια σημαντική μείωση του χρόνου.

- **Δημιουργία task dependency graph**

  Ο στόχος μας εδώ είναι η υλοποίηση του task dependency graph που σχεδιάσαμε στην ενδιάμεση αναφορά σε tbb, ώστε να έχουμε την μεγαλύτερη παραλληλία. Κάποιες φορές, βέβαια, η δημιουργία ενός παραπλήσιου task graph με μεγαλύτερη ευελιξία μπορεί να συμφέρει περισσότερο. Η δημιουργία μπορεί να γίνει με 2 τρόπους:

  - ☐ **Task groups**

    Εδώ , δημιουργούμε ομάδες από tasks (task group), πάνω στις οποία έχουμε δύο βασικές λειτουργίες [3]. Με την συνάρτηση run τοποθετούμε ένα task στην ουρά προς εκτέλεση από κάποιο thread και με την συνάρτηση wait περιμένουμε την ολοκλήρωση όλων των tasks του συγκεκριμένου taskgroup (συγχρονισμός). Η τεχνική αυτή, εφαρμόστηκε στον recursive και στον tiled αλγόριθμο, τα task dependency graph των οποίων έχουν υλοποιηθεί στην ενδιάμεση αναφορά.

  - ☐ **Task-Flow graph**

    Τα task groups δεν έχουν την δυνατότητα να περιγράψουν οποιοδήποτε task dependency graph. Λύση σε αυτό δίνουν τα Generic Task Graphs ή Flow graph [4], με τα οποία μπορείς να περιγράψεις οποιοδήποτε task dependency graph. Ουσιαστικά, το μόνο που χρειάζεται είναι να ορίσεις τους κόμβους και τις ακμές του γράφου.

1 Υλοποίηση και αποτελέσματα για τον recursive αλγόριθμο

- ✓ Κώδικας με χρήση task group:

```
1 if(myN<=bsize)
2     for(k=0; k<myN; k++)
3         for(i=0; i<myN; i++)
4             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 {
    FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);

    tbb::task_group g;

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

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

    g.wait();

    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);

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

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

    g.wait();

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

✓ Κώδικας με χρήση flow graph:

```
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 {
    graph g;
    continue_node <continue_msg> A11(g, [=] (const continue_msg &) {FW_SR(A, arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize); });

    continue_node <continue_msg> A12(g, [=] (const continue_msg &) {FW_SR(A, arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize); });

    continue_node <continue_msg> A21(g, [=] (const continue_msg &) {FW_SR(A, arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize); });

    continue_node <continue_msg> A22(g, [=] (const continue_msg &) {FW_SR(A, arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize); });

```

```
    continue_node <continue_msg> A22_(g, [=] (const continue_msg &) {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); });

    continue_node <continue_msg> A21_(g, [=] (const continue_msg &) {FW_SR(A,
    arow+myN/2, acol ,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol , myN/2, bsize
    ); });

    continue_node <continue_msg> A12_(g, [=] (const continue_msg &) {FW_SR(A,
    arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize
    ); });

    continue_node <continue_msg> A11_(g, [=] (const continue_msg &) {FW_SR(A,
    arow, acol ,B,brow, bcol+myN/2,C,crow+myN/2, ccol , myN/2, bsize); });

    make_edge(A11,A12);
    make_edge(A11,A21);
    make_edge(A12,A22);
    make_edge(A21,A22);
    make_edge(A22,A22_);
    make_edge(A22_,A12_);
    make_edge(A22_,A21_);
    make_edge(A12_,A11_);
    make_edge(A21_, A11_);

    A11.try_put(continue_msg());
    g.wait_for_all();
}
```

✓ Γραφικές παραστάσεις για κάθε μέγεθος ταμπλό:

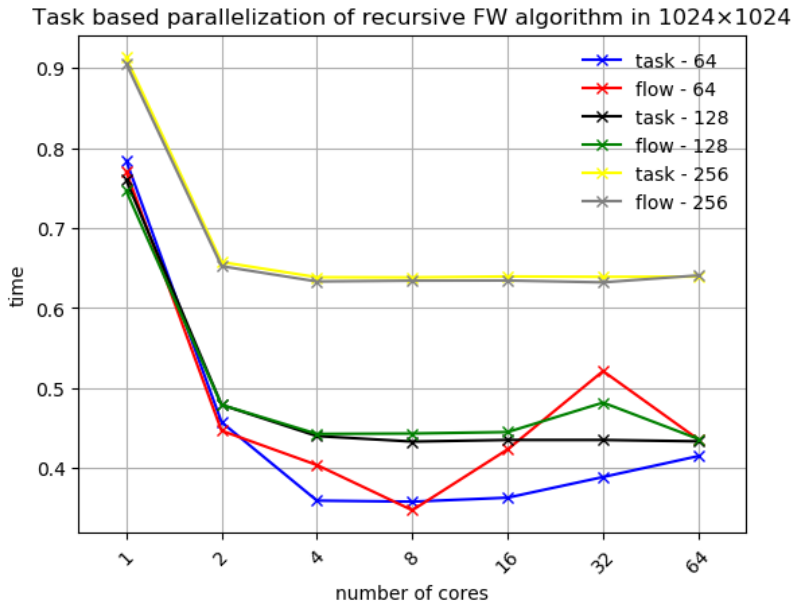

Figure 7: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=1024

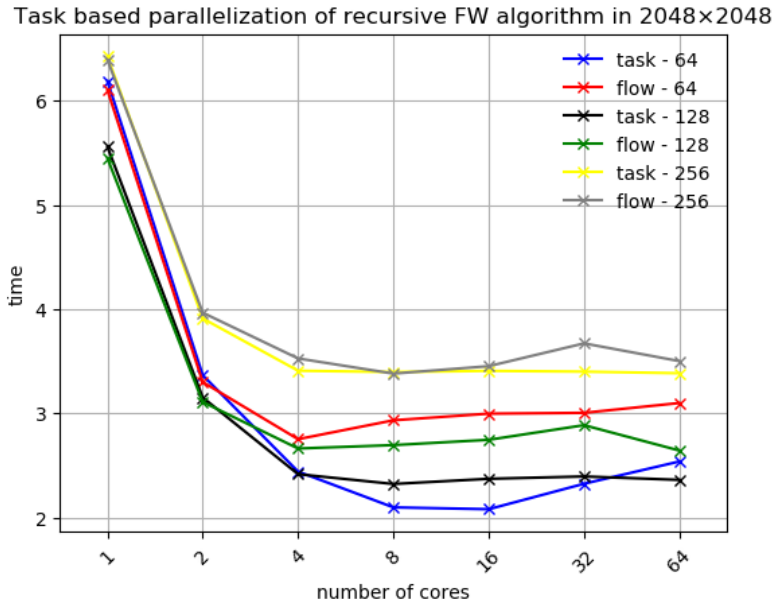

Figure 8: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=2048

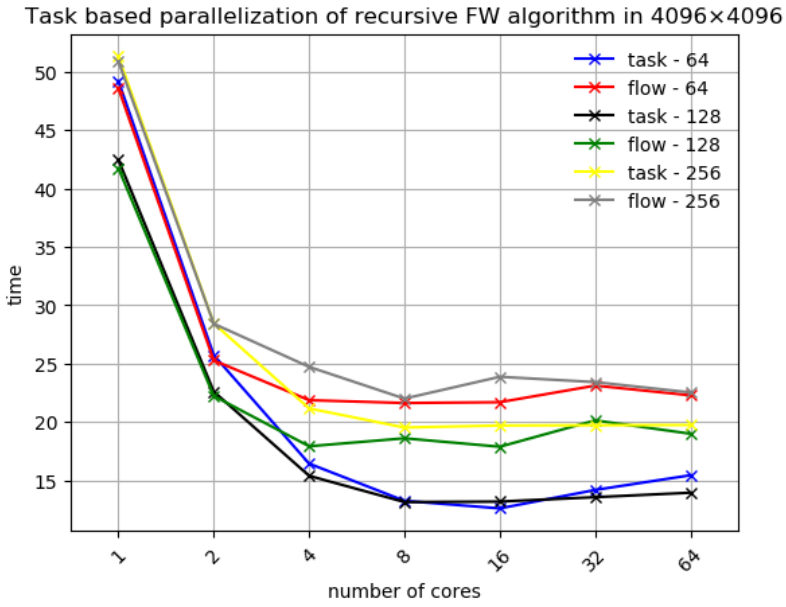

Figure 9: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=4096

✓ Σχολιασμός:
Αρχικά, παρατηρούμε ότι σε όλες τις περιπτώσεις τα task groups είναι λίγο πιο αποδοτικά από το flow graph. Αυτό λογικά οφείλεται στο γεγονός ότι στα task groups δημιουργείται δυναμικά ο γράφος ενώ στο flow graph πρώτα δημιουργείται όλος ο γράφος στατικά και μετά ξεκινάει η εκτέλεσή του. Οπότε η ευελιξία και η εκφραστικότητα των flow graph επηρεάζουν αρνητικά την επίδοσή τους. Όσο για το μέγεθος του block οι καλύτεροι χρόνοι επιτυγχάνονται για B=64. Όσο αυξάνεται το μέγεθος του ταμπλό η διαφορά του B=64 από το B=128 μειώνεται, καθώς όσο αυξάνουμε το ταμπλό και κρατάμε το B σταθερό η αναδρομή γίνεται όλο και σε μεγαλύτερο βάθος προκαλώντας καθυστερήσεις. Τέλος, δεν υπάρχει κλιμάκωση για περισσότερους από 16 πυρήνες. Για B=64, τα blocks που δημιουργούνται είναι 16 για N=1026, 32 για N=2048 και 64 για N=4096. Για τον λόγο αυτό βλέπουμε ότι η καλύτερη καμπύλη χρησιμοποιεί όλο και περισσότερους πυρήνες όσο αυξάνουμε το μέγεθος του ταμπλό. Για παράδειγμα, για N=1024, όπου έχουμε 16 blocks μεγέθους 64, οι 32 ή 64 πυρήνες δεν προσφέρουν περισσότεροι παραλληλία στο πρόγραμμα και δρουν μόνο αρνητικά στην απόδοση. Τέλος, να σημειωθεί ότι οι πίνακες περνάνε ως παράμετροι by reference ενώ οι υπόλοιπες μεταβλητές by value, για να έχουμε την ελάχιστη καθυστέρηση κάθε φορά.

2 Υλοποίηση και αποτελέσματα για τον tiled αλγόριθμο

✓ Κώδικας με χρήση task group:

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

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

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

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

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

   g.wait();

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

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

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

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

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

    g.wait();
}
```

✓ Κώδικας με χρήση flow graph:

```
for(k=0;k<N;k+=B){
    graph g;
    continue_node <continue_msg> Akk(g, [=] (const continue_msg &) { FW(A,k,k,
    k,B); });

    continue_node <continue_msg> Aik(g, [=] (const continue_msg &) { for(int i
    =0; i<k; i+=B) FW(A,k,i,k,B); });

    continue_node <continue_msg> Aki(g, [=] (const continue_msg &) { for(int i
    =k+B; i<N; i+=B) FW(A,k,i,k,B); });

    continue_node <continue_msg> Ajk(g, [=]  (const continue_msg &) { for(int
    j=0; j<k; j+=B) FW(A,k,k,j,B); });

    continue_node <continue_msg> Akj(g, [=]  (const continue_msg &) { for(int
    j=k+B; j<N; j+=B) FW(A,k,k,j,B); });

    continue_node <continue_msg> Aij1(g, [=]  (const continue_msg &) { for(int
     i=0; i<k; i+=B) for(int j=0; j<k; j+=B) FW(A,k,i,j,B); });

    continue_node <continue_msg> Aij2(g, [=]  (const continue_msg &) { for(int
     i=0; i<k; i+=B) for(int j=k+B; j<N; j+=B) FW(A,k,i,j,B); });

    continue_node <continue_msg> Aij3(g, [=]  (const continue_msg &) { for(int
     i=k+B; i<N; i+=B) for(int j=0; j<k; j+=B) FW(A,k,i,j,B); });

    continue_node <continue_msg> Aij4(g, [=]  (const continue_msg &) { for(int
     i=k+B; i<N; i+=B) for(int j=k+B; j<N; j+=B) FW(A,k,i,j,B); });

    make_edge(Akk,Aik);
    make_edge(Akk,Aki);
    make_edge(Akk,Ajk);
    make_edge(Akk,Akj);
    make_edge(Aik,Aij1);
    make_edge(Ajk,Aij1);
    make_edge(Aik,Aij2);
    make_edge(Akj, Aij2);
    make_edge(Aki, Aij3);
    make_edge(Ajk, Aij3);
    make_edge(Aki, Aij4);
    make_edge(Akj, Aij4);


    Akk.try_put(continue_msg());
    g.wait_for_all();
}
```

✓ Γραφικές παραστάσεις των παραπάνω μεθόδων για κάθε μέγεθος ταμπλό:

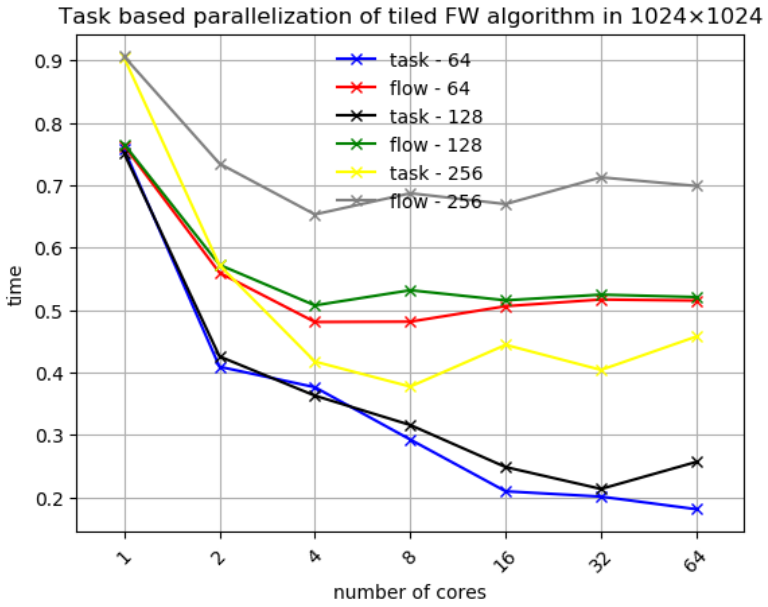

Figure 10: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=1024

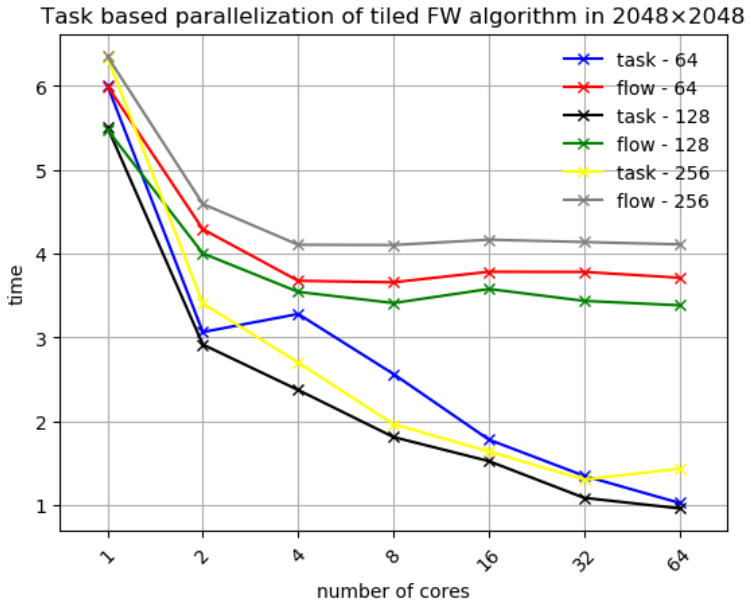

Figure 11: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=2048

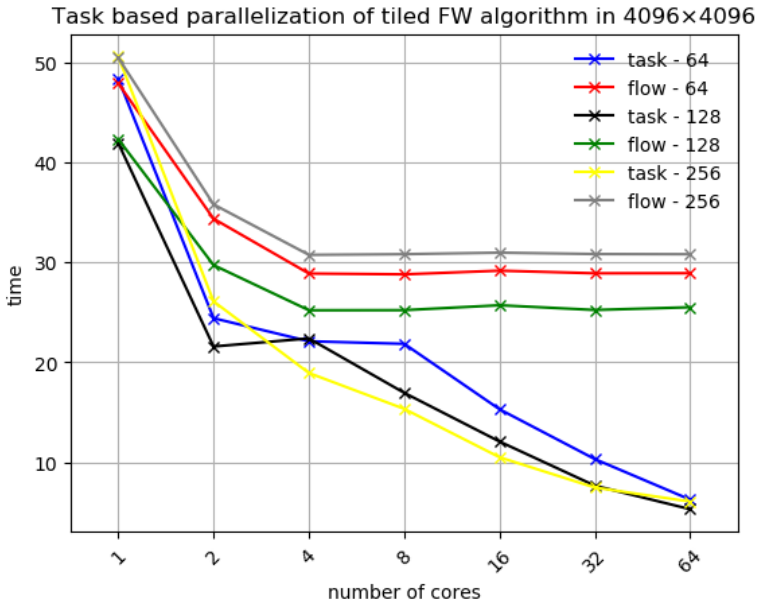

Figure 12: Χρόνοι εκτέλεσης των παραπάνω μεθόδων για N=4096

✓ Σχολιασμός:
Αυτό που επισημάνθηκε παραπάνω εδώ γίνεται πιο έντονο. Οι task group υλοποιήσεις είναι πολύ πιο αποδοτικές από τις flow graph. Η διαφορά εδώ είναι μεγαλύτερη καθώς ο γράφος που δημιουργούμε κάθε φορά είναι μεγαλύτερος (στον recursive οι κόμβοι δεν ήταν πολλοί γιατί κάθε task καλείται αναδρομικά) και έτσι η δημιουργία όλου του γράφου από την flow graph υλοποίηση καθυστερεί όλη την διαδικασία. Από την άλλη, η κλιμάκωση που περιμέναμε από την αύξηση των πυρήνων εδώ συμβαίνει με μεγάλη επιτυχία. Σε κάθε επανάληψη τα $\frac{N}{B}$ blocks που υπάρχουν μπορούν να χρησιμοποιηθούν (με κατάλληλο συγχρονισμό που έχουμε ορίσει) από τους πυρήνες που έχουμε. Συνολικά, βλέπουμε ότι μπορεί η αναδρομική υλοποίηση να είναι πιο κομψή αλλά αποδοτικότερη είναι η tiled, καθώς κλιμακώνει πιο πολύ. Να σημειωθεί ότι και εδώ στο task group περνάμε τον πίνακα by reference για μεγαλύτερη ταχύτητα.

1.4.2 Με χρήση του προγραμματιστικού εργαλείου OpenMP

Για τον παραλληλισμό προγραμμάτων σε αρχιτεκτονικές κοινής μνήμης, το OpenMP ορίζει μία συγκεκριμένη διεπαφή (API) με τους εξής τρόπους:

- οδηγίες σε μεταγλωττιστή (compiler directives)
- βιβλιοθήκη χρόνου εκτέλεσης (run-time library)
- μεταβλητές συστήματος (environment variables)

Το εργαλείο αυτό χρησιμοποιήθηκε για να υλοποιηθούν οι παράλληλες εκδόσεις 3 υλοποιήσεων του αλγόριθμου Floyd-Warshall, της standard υλοποίησης, της recursive και της tilled.

- **Παραλληλοποίηση for-loop με parallel for**

  Η τεχνική αυτή εφαρμόστηκε στην standard υλοποίηση του αλγορίθμου που βασίζεται στην επαναληπτική ανανέωση του πίνακα. Όπως μπορούμε να δούμε και στον γράφο εξαρτήσεων, το εξωτερικό for loop δεν είναι παραλληλοποιήσιμο. Αντίθετα, η ανανέωση του ταμπλό για ένα συγκεκριμένο χρόνικό βήμα είναι (είτε κατά γραμμές είτε κατά block). Δοκιμάστηκαν και οι 2 τρόποι αλλά ο πιο αποτελεσματικός ήταν αυτός με την παραλληλοποίηση κατά γραμμές (όπως αναφέρθηκε και στην αντίστοιχη tbb υλοποίηση).

  ✓ Κώδικας:

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

  ✓ Γραφικές παραστάσεις για κάθε μέγεθος ταμπλό:

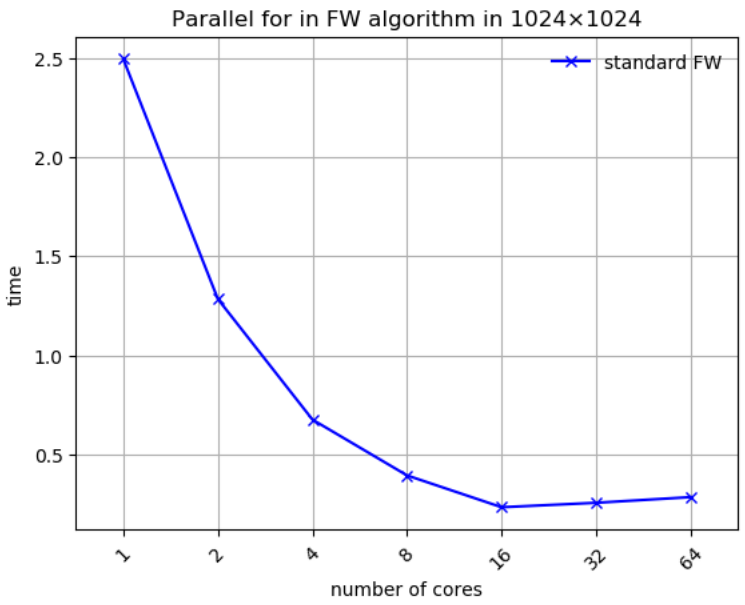

Figure 13: Χρόνοι εκτέλεσης για N=1024

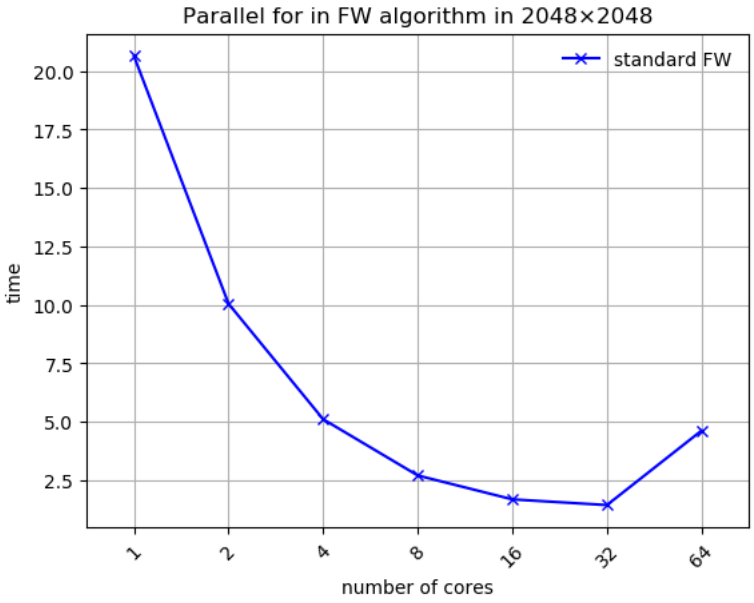

Figure 14: Χρόνοι εκτέλεσης για N=2048

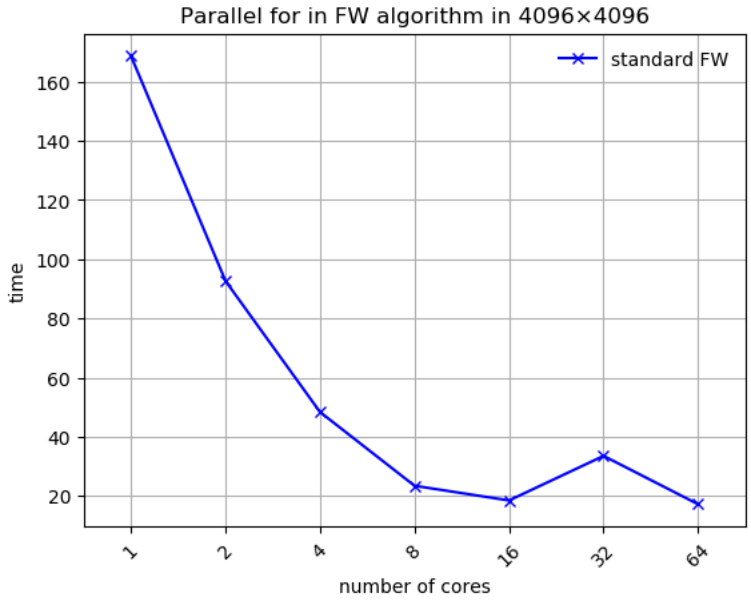

Figure 15: Χρόνοι εκτέλεσης για N=4096

✓ Σχολιασμός:

Γενικά, η συμπεριφορά είναι παρόμοια με την υλοποίηση του parallel for σε tbb, με όλα τα θετικά και αρνητικά που αναφέρθηκαν.

✓ Σημείωση:

Κατά την χρησιμοποίηση του parallel for δοκιμάστηκαν και οι πρακτικές Scheduling, Dynamic και Guided. Η πρώτη μέθοδος ορίζει ένα μέγεθος chunk και τα αναθέτει στα διαθέσιμα threads. Όταν ένα thread ολοκληρώσει το chunk που του ανατέθηκε ζητά καινούργιο έως ότου ολοκληρωθούν όλα τα iterations. Η δεύτερη μέθοδος είναι παρόμοια με την πρώτη αλλά διαφέρει στο ότι αναθέτει chunks ανάλογα των ελεύθερων νημάτων με αποτέλεσμα την εκθετική μείωση του μεγέθους των chunks. Παρόλο που η ανάθεση γίνεται πιο ευέλικτη με αυτούς τους τρόπους, στην περίπτωση που εξετάζεται δεν έχουν καμία επίδραση. Ο λόγος είναι ότι οι πρακτικές αυτές δίνουν καλύτερα αποτελέσματα όταν κάθε iteration δεν έχει τον ίδιο "φόρτο" εργασίας. Στις περιπτώσεις αυτές, αν κάποια threads παραμείνουν απασχολημένα για αρκετό διάστημα τα υπόλοιπα δεν μενουν ανενεργά. Στην standard υλοποίηση, όμως, του αλγόριθμου ο "φόρτος" είναι ίδιος οπότε χάνεται το πλεονέκτημα των μεθόδων αυτών.

- **Δημιουργία task dependency graph**

Η μέθοδος παραλληλοποίησης με tasks χρησιμοποιήθηκε στην recursive υλοποίηση, όπου η δημιουργία του task dependency graph είναι εύκολη με τις δυνατότητες των OpenMP tasks. Στην tiled υλοποίηση χρησιμοποιήθηκε άλλη τεχνική, όπως θα δούμε παρακάτω.

✓ Κώδικας:

```
1  FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
2
3  #pragma omp parallel
4  {
5      #pragma omp single
6      {
7          #pragma omp task shared (A, B, C)
8              FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize)
       ;
9          FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
10         #pragma omp taskwait
11     }
12 }
13
14 FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize
       );
15 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);
16
17 #pragma omp parallel
18 {
19     #pragma omp single
20     {
21         #pragma omp task shared(A, B, C)
22             FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol,
       myN/2, bsize);
23         FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN
       /2, bsize);
24         #pragma omp taskwait
25     }
26 }
```

```
27
28 FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
```

✓ Γραφικές παραστάσεις για κάθε μέγεθος ταμπλό:

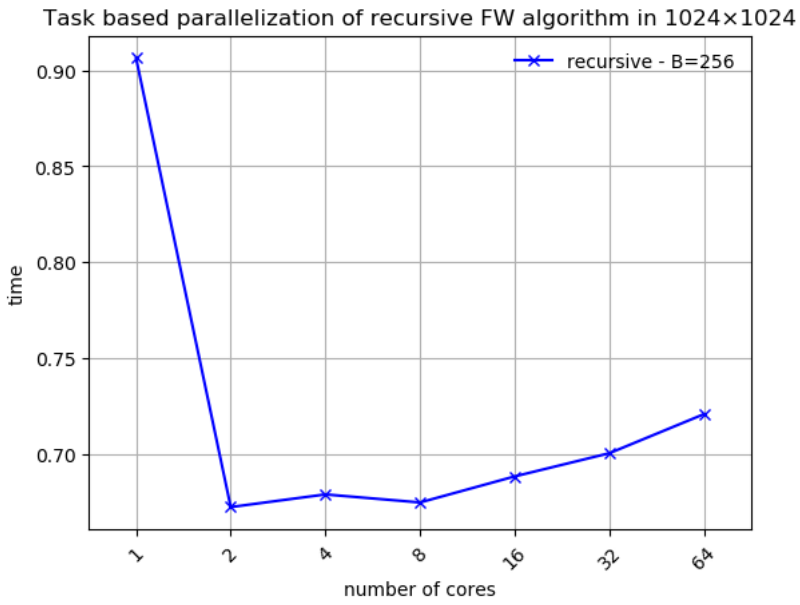

Figure 16: Χρόνοι εκτέλεσης για N=1024

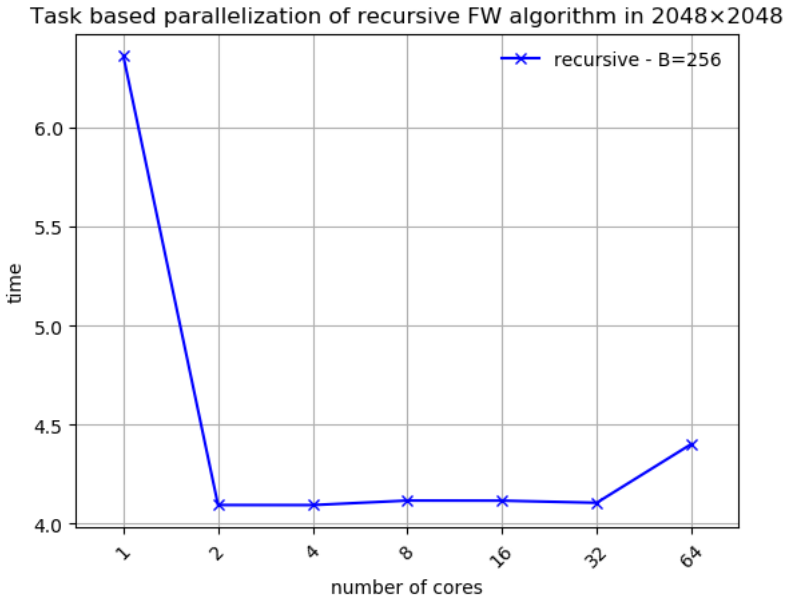

Figure 17: Χρόνοι εκτέλεσης για N=2048

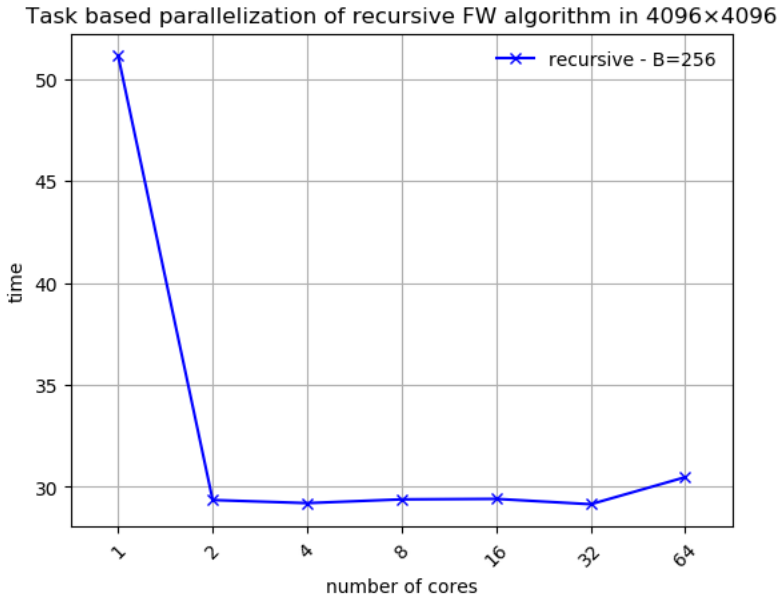

Figure 18: Χρόνοι εκτέλεσης για N=4096

✓ Σχολιασμός:

Τα αποτελέσματα δεν είναι καλά για τους λόγους που αναφέρθηκαν και στο tbb. Εδώ, βέβαια, είναι ακόμα χειρότερα από την task-based υλοποίηση του recursive αλγορίθμου σε tbb.

- **Συνδυασμός parallel for και dependency graph**

  Τέλος, για την παραλληλοποίηση του tiled αλγορίθμου σε OpenMP, χρησιμοποιήθηκε η τεχνική του parallel for. Ωστόσο, εκμεταλλευτήκαμε πληροφορίες για την παραλληλία του αλγορίθμου από το task dependency graph, οπότε μπορεί να θεωρηθεί ως συνδυασμός αυτών (χωρίς την χρήση tasks). Συγκεκριμένα, στον tiled αλγόριθμου (πέρα από τον υπολογισμό των διαγώνιων στοιχείων) έχουμε τον παράλληλο υπολογισμό των στοιχείων της k γραμμής και k στήλης και στη συνέχεια τον παράλληλο υπολογισμό των υπόλοιπων στοιχείων. Έτσι, οι δύο αυτές κατηγορίες υπολογισμών αποτελούν από μία παράλληλη περιοχή, στην οποία υπάρχουν παράλληλα for loop.

- Κώδικας:

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

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

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

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

    #pragma omp parallel shared(A, B, k)
    {
        #pragma omp for collapse(2) nowait
        for(i=0; i<k; i+=B)
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for collapse(2) 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 collapse(2) 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 collapse(2) nowait
        for(i=k+B; i<N; i+=B)
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);
```

```
43      }
44  }
```

- Γραφικές παραστάσεις για κάθε μέγεθος ταμπλό:

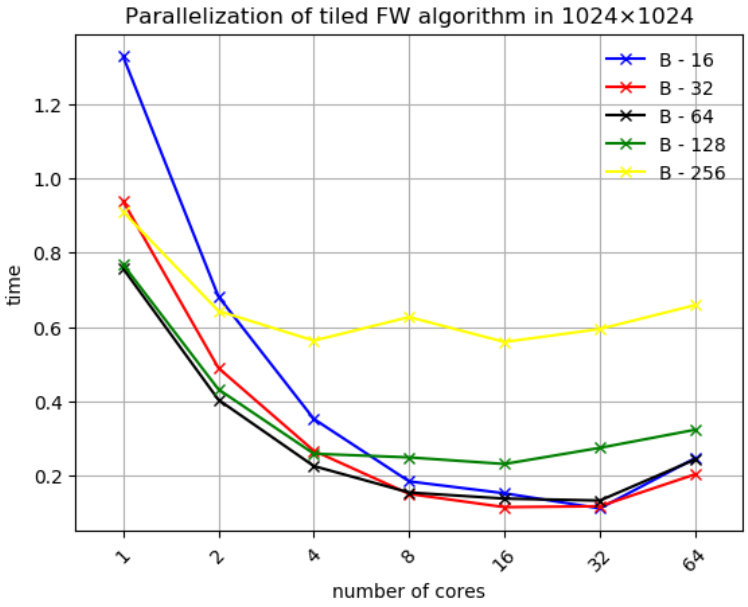

Figure 19: Χρόνοι εκτέλεσης για N=1024

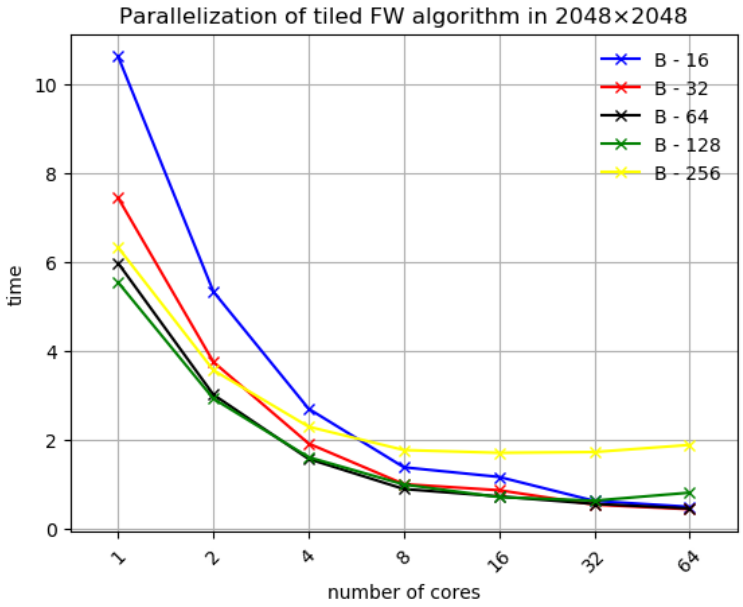

Figure 20: Χρόνοι εκτέλεσης για N=2048

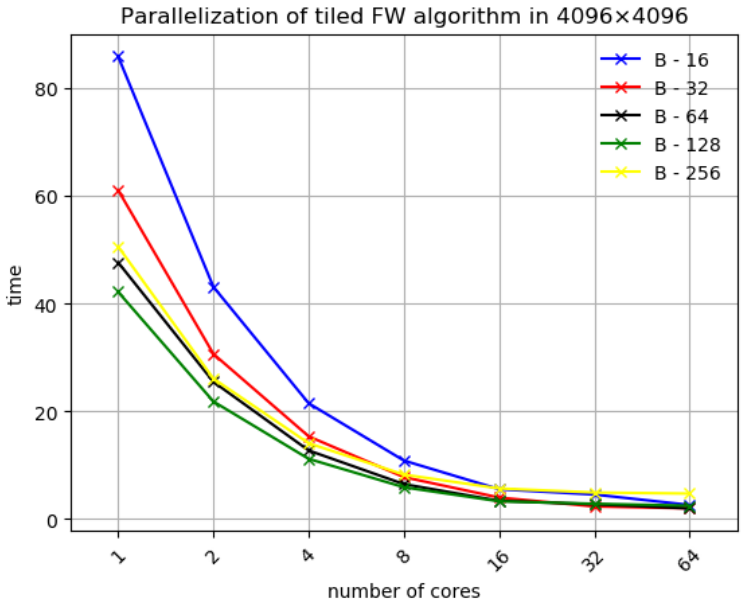

Figure 21: Χρόνοι εκτέλεσης για N=4096

- Σχολιασμός:

  Παρατηρούμε ότι η ευελιξία που μας δίνει το OpenMP μαζί με τον tiled αλγόριθμο μας δίνουν το καλύτερο αποτέλεσμα. Τα πλεονεκτήματα του tiled αλγορίθμου έχουν αναφερθεί και στο κομμάτι του tbb. Αυτό που κερδίζουμε παραπάνω εδώ είναι ότι συνδυάζουμε τα πλεονεκτήματα του parallel for και της task-based υλοποίησης. Από την μία, χάρη στον tiled αλγόριθμο, καταφέρνουμε να έχουμε ένα αρκετά παραλληλοποιήσιμο γράφο εξαρτήσεων (αποφεύγωντας παράλληλα τα αρνητικά του recursive) και από την άλλη σε κάθε ομάδα από εργασίες εφαρμόζουμε parallel for αντί να δημιουργούμε τον γράφο, πράγμα που επιτρέπει στο OpenMP να δρομολογεί πιο αποδοτικά τα νήματα που δημιουργούνται χωρίς να θυσιάζουμε παραπάνω χρόνο για συγχρονισμό (βλέπε και 1.6).

1.5 Σημειώσεις

- Όλες οι υλοποιήσεις μεταγλωτίστηκαν με όρισμα -O3 για compiler optimizations.
- Η ορθότητα κάθε υλοποίησης εξετάστηκε συγκρίνοντας το output που έδινε με το output του αντίστοιχου σειριακού αλγορίθμου.
- Μία ακόμη βελτιστοποίηση θα ήταν η αρχικοποίηση του ταμπλό να γίνει παράλληλα ώστε τα νήματα που θα κάνουνε malloc την περιοχή να είναι και αυτά που τελικά θα την χρησιμοποιήσουν. Έτσι, τα νήματα θα χρησιμοποιούν δεδομένα που θα είναι κοντά τους, περιορίζοντας έτσι τον χρόνο επικοινωνίας μνήμης-επεξεργαστή.

1.6 Παρουσίαση καλύτερων μετρήσεων

Στο κομμάτι αυτό, θα παρουσιάσουμε για κάθε μέγεθος ταμπλό τις 3 καλύτερες μετρήσεις κια θα γίνει μία προσπάθεια συνολικής ερμηνείας των αποτελεσμάτων μας.

- **Για N=1024:** Ο χρόνος του καλύτερου σειριακού είναι 1.0421 s, οπότε έχουμε speedup ίσο με 9.31.

Κατάταξη	Αλγόριθμος	Εργαλείο	Τεχνική	Πυρήνες	Μέγεθος Block	Χρόνος (s)
1	Tiled	OpenMP	Parallel For	32	16	0.1119
2	Tiled	OpenMP	Parallel For	16	32	0.1152
3	Tiled	OpenMP	Parallel For	32	32	0.1177

- **Για N=2048:** Ο χρόνος του καλύτερου σειριακού είναι 7.7952 s, οπότε έχουμε speedup ίσο με 17.8.

Κατάταξη	Αλγόριθμος	Εργαλείο	Τεχνική	Πυρήνες	Μέγεθος Block	Χρόνος (s)
1	Tiled	OpenMP	Parallel For	64	32	0.4380
2	Tiled	OpenMP	Parallel For	64	64	0.4541
3	Tiled	OpenMP	Parallel For	64	16	0.4906

- **Για N=4096:** Ο χρόνος του καλύτερου σειριακού είναι 62.5464 s, οπότε έχουμε speedup ίσο με 31.4.

Κατάταξη	Αλγόριθμος	Εργαλείο	Τεχνική	Πυρήνες	Μέγεθος Block	Χρόνος (s)
1	Tiled	OpenMP	Parallel For	64	64	1.9922
2	Tiled	OpenMP	Parallel For	64	32	2.0140
3	Tiled	OpenMP	Parallel For	64	128	2.4837

Παρατηρούμε, λοιπόν, ότι σε όλα τα μεγέθη η **OpenMP tiled υλοποίηση είναι και η καλύτερη**. Αυτό συμβαίνει γιατί, από την μία ο tiled αλγόριθμος είναι ο πιο αποδοτικά παραλληλοποιήσιμος και μας δίνει την δυνατότητα να τρέξουμε τον standard αλγόριθμο στα block από πολλά νήματα κάθε φορά. Από την άλλη, το OpenMP με την εφαρμογή parallel for σε όλες τα loops που μπορούν να γίνουν παράλληλα, μας έδωσε την δυνατότητα να περιορίσουμε όσο γίνεται περισσότερο την ανάγκη συγχρονισμό των νημάτων (γίνεται μόνο 2 φορές σε κάθε επανάληψη) και να έχουμε πολλά ενεργά νήματα κάθε φορά. Ως βελτιώση, θέσαμε την μεταβλητή περιβάλλοντος **OMP_PRO_BIND** σε true, ώστε οι διεργασίες να μην αλλάζουν επεξεργαστή στον οποίο τρέχουν και πετύχαμε χρόνο 1.6535 s.

Αναφορές

[1] Park, J-S., Michael Penner, and Viktor K. Prasanna. "Optimizing graph algorithms for improved cache performance." IEEE Transactions on Parallel and Distributed Systems 15.9 (2004): 769-782.

[2] "Parallel-for" `https://software.intel.com/en-us/node/506057`

[3] "Task group" `https://software.intel.com/en-us/node/506287`

[4] "Dependence graph" `https://software.intel.com/en-us/node/517349`

[5] "Σημειώσεις του μαθήματος" `http://www.cslab.ntua.gr/courses/pps/notes.go`