Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών Ε.Μ.Π.

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ
2016-2017

Άσκηση 2
Παράλληλη επίλυση του αλγορίθμου Floyd-Warshall σε πολυπύρηνες αρχιτεκτονικές

Βαρηά Χρυσούλα - 03112105
Κουτσανίτη Ειρήνη - 03112135

Όλες οι υλοποιήσεις έγιναν με χρήση του openMP. Τα πειραματικά αποτελέσματα των εκτελέσεων των σειριακών εκδόσεων παρουσιάζονται στη συνέχεια:

	64×64	1024×1024	2048×2048
fw.c	1.3739	11.22	93.5341
fw_sr.c, B=128	0.6625	4.8154	36.4538
fw_tiled.c, B=128	0.6928	4.8491	35.9686

Για τις γραφικές παραστάσεις του speedup χρησιμοποιούμε πάντα την καλύτερη τιμή για κάθε είσοδο και όχι την τιμή του αντίστοιχου σειριακού αλγορίθμου.

Όλες οι υλοποιήσεις που περιγράφονται παρουσιάζονται στο τέλος της αναφοράς.

Standard έκδοση αλγορίθμου Floyd-Warshall

Παραλληλοποίηση:

Ο αλγόριθμος που δίνεται στο αρχείο fw.c δεν είναι τόσο εύκολα παραλληλοποιήσιμος όσο η έκδοση (βλ. Διαφάνειες Υλοποίηση παράλληλων προγραμμάτων σελ. 16) που χρησιμοποιεί δύο πίνακες $N \times N$ (ή έναν $2 \times N \times N$) και για κάθε k υπολογίζει την τιμή A[(k+1)%2][i][j] με βάση τις τιμές A[k%2][i][j], A[k%2][i][k] και A[k%2][k][j], δηλαδή τιμές που υπολόγισε σε προηγούμενη επανάληψη του εξωτερικού loop k. Χρησιμοποιώντας δύο πίνακες, οι βρόχοι i και j είναι ανεξάρτητοι και μπορούν να υπολογιστούν παράλληλα για δεδομένο k.

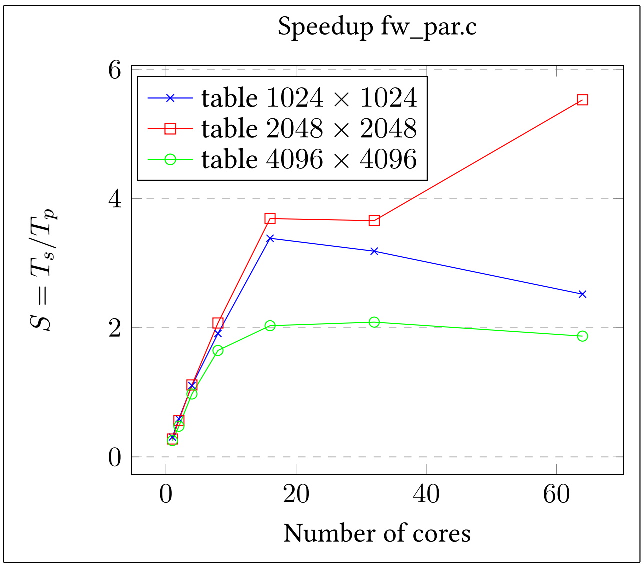
Αντίθετα, με έναν πίνακα προκύπτουν επιπλέον εξαρτήσεις. Για παράδειγμα, για k=2, i=5, j=1 χρειάζονται οι τιμές A[5][1], A[2][1] και A[5][2], όμως στη σειριακή υλοποίηση που δίνεται το A[2][1] έχει υπολογιστεί για k=2 ενώ σε αυτή που χρησιμοποιεί δύο πίνακες για k=1. Παραλληλοποιώντας τον βρόχο i δεν είναι σίγουρο αν το A[2][1] για k=2 θα έχει υπολογιστεί ακόμα. Ευτυχώς, τα k περάσματα του πίνακα διορθώνουν αυτή την ασάφεια και για το τελικό αποτέλεσμα δεν έχει σημασία ποια από τις δύο τιμές θα πάρουμε.

Επομένως, η παραλληλοποίηση της συγκεκριμένης έκδοσης μπορεί να επιτευχθεί με τη χρήση parallel for με private τιμές τις μεταβλητές i,j για κάθε νήμα και shared τη μεταβλητή k του εξωτερικού βρόχου.

Ακολουθούν τα αποτελέσματα των πειραματικών μετρήσεων και το διάγραμμα της επιτάχυνσης του προγράμματος:

fw_par.c

	64×64	1024×1024	2048×2048
1	2.1599	17.4256	143.0654
2	1.1254	8.5182	75.8905
4	0.6003	4.3261	36.9569
8	0.3469	2.324	21.8331
16	0.1958	1.3056	17.7156
32	0.2081	1.317	17.2449
64	0.2631	0.8713	19.2454



Παρατηρήσεις:

- Ο χρόνος εκτέλεσης του προγράμματος μειώνεται με την αύξηση των νημάτων, το οποίο είναι αναμενόμενο καθώς μεγαλύτερο μέρος του κώδικα εκτελείται παράλληλα. Αυτό φαίνεται και από το γεγονός ότι βελτίωση του χρόνου είναι εντονότερη σε μεγαλύτερα μεγέθη πίνακα εισόδου.

- Επίσης η επιτάχυνση βελτιώνεται με την αύξηση των νημάτων, όμως για μεγαλύτερα μεγέθη πίνακα (N=4096) η τιμή της μειώνεται. Αυτό μπορεί να οφείλεται στην αρχιτεκτονική μνήμης του μοντέλου sandman, καθώς μπορεί οι επεξεργαστές να απαιτούν δεδομένα, τα οποία είναι αποθηκευμένα σε απομακρυσμένες περιοχές. Επομένως μπορεί να προκληθεί αναμονή των μονάδων επεξεργασίας μέχρι να εξασφαλιστούν τα αντίστοιχα δεδομένα λόγω καθυστέρησης της μεταφοράς ή λόγω συμφόρησης στο data bus.

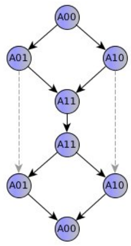
Recursive έκδοση αλγορίθμου Floyd-Warshall

Παραλληλοποίηση:

Για τη συγκεκριμένη έκδοση η διαδικασία εκτέλεσης/αναδρομής γίνεται ως εξής: Ο πίνακας A χωρίζεται σε 4 επιμέρους τετραγωνικούς πίνακες A00, A01, A10 και A11. Αρχικά υπολογίζονται οι τιμές του υποπίνακα A00, χρησιμοποιώντας μόνο δικά του δεδομένα. Έπειτα υπολογίζονται παράλληλα οι τιμές των A01 και A10, οι οποίοι εξαρτώνται από τιμές του A00 και των ίδιων. Αφού ενημερωθούν οι δύο επιμέρους υποπίνακες μπορεί να ξεκινήσει ο υπολογισμός του A11, ο οποίος εξαρτάται από τους προηγούμενους δύο πίνακες. Ο A11 ενημερώνεται σειριακά δύο φορές, χρησιμοποιώντας τις ίδιες του τις τιμές, επομένως δεν υπάρχει περιθώριο παραλληλοποίησης των συγκεκριμένων αναδρομών. Μετά τις παραπάνω εκτελέσεις, οι υποπίνακες A10 και A01 είναι σε θέση να ενημερωθούν ξανά. Επειδή δεν υπάρχουν εξαρτήσεις μεταξύ τους μπορούν να υπολογιστούν από διαφορετικά νήματα. Τέλος ενημερώνεται ο υποπίνακας A00 χρησιμοποιώντας τις τιμές των A01, A10 παραπάνω.

Η διαδικασία, η οποία μόλις περιγράφτηκε, παρουσιάζεται περιληπτικά στη συνέχεια και από το γράφο εξαρτήσεων για κάθε βήμα της αναδρομής:

1) A00 <- A00, A00
2) A01 <- A00, A01 και A10 <- A10, A00
3) A11 <- A10, A01
4) A11 <- A11, A11
5) A10 <- A11, A10 και A01 <- A01, A11
6) A00 <- A01, A10

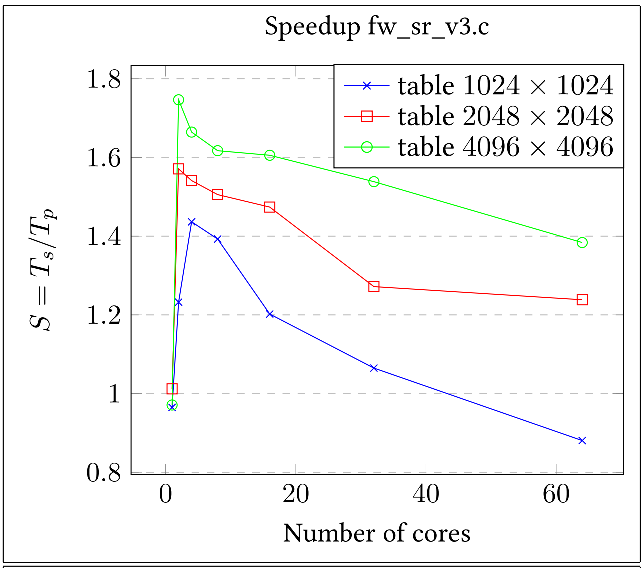


Πρέπει να σημειωθεί οτί το σχήμα δεν αντικατοπτρίζει τις πραγματικές εξαρτήσεις για όλες τις αναδρομές, αλλά είναι πιο συντηρητικό. Ακόμα, η συγκεκριμένη έκδοση δεν μπορεί να παραλληλοποιηθεί πλήρως και αξιοποιείται μικρό πλήθος νημάτων (2 το πολύ νήματα σε 1 αναδρομή).

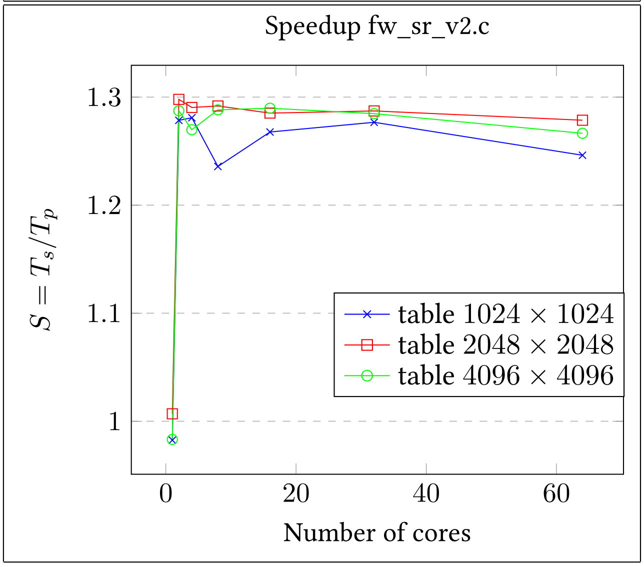
Για τη συγκεκριμένη σειριακή έκδοση υλοποιήθηκαν δύο παραλλαγές κώδικα. Οι συγκεκριμένες τροποποιήσεις είναι παρόμοιας λογικής, άλλα υλοποιούνται με διαφορετικό τρόπο. Η μία υλοποιείται με το μοντέλο fork-join/tasks(fw_sr_v2.c) και η άλλη με χρήση sections(fw_sr_v3.c).

Ακολουθούν τα αποτελέσματα των πειραματικών μετρήσεων και τα αντίστοιχα διαγράμματα επιτάχυνσης:

fw_sr_v3.c, B=128			
	64×64	1024×1024	2048×2048
1	0.659	4.5976	35.0384
2	0.4292	2.7382	19.2022
4	0.4292	2.8877	19.8917
8	0.4399	3.0415	20.1826
16	0.4777	3.1529	20.4648
32	0.6034	3.4597	21.9576
64	0.7596	3.991	24.1684



fw_sr_v2.c, B=128			
	64×64	1024×1024	2048×2048
1	0.6743	4.7822	36.5821
2	0.5182	3.7101	27.9387
4	0.5172	3.7317	28.3256
8	0.5361	3.7278	27.9216
16	0.5226	3.7469	27.8889
32	0.5189	3.7411	27.9974
64	0.5316	3.7662	28.4009



Παρατηρήσεις:

- Όπως και στην πρώτη έκδοση ο χρόνος εκτέλεσης του προγράμματος μειώνεται με την αύξηση των νημάτων, όμως στη συγκεκριμένη περίπτωση η μείωση είναι πολύ μικρή (ανεξαρτήτως διάστασης πίνακα). Αυτό ίσως οφείλεται στην καθυστέρηση της μεταφοράς των δεδομένων, τα οποία εγγράφονται, μεταξύ των L1, L2 caches (τήρηση συνάφειας μνήμης σε κάθε υποσύστημα Numa Node).

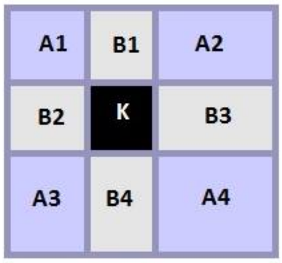
- Παρατηρείται επίσης ότι για πλήθος νημάτων μεγαλύτερο από 8, η επίδοση δυσχεραίνει. Επειδή ο αριθμός των νημάτων είναι αρκετά μεγάλος, μπορούν να χρησιμοποιηθούν επεξεργαστικές μονάδες από παραπάνω υποσυστήματα (Numa Nodes). Αυτό συνεπάγεται την ανάγκη για επικοινωνία μεταξύ των προαναφερθέντων κόμβων, το οποίο οδηγεί στη διαδοχική ανταλλαγή μηνυμάτων, με σκοπό τη μεταφορά των νέων δεδομένων, στα οποία έγινε κάποια εγγραφή. Αυτή η μορφή επικοινωνίας αυξάνει την αναμονή των επεξεργαστών και επομένως την καθυστέρηση της εκτέλεσης του προγράμματος. Παρόμοιες παρατηρήσεις έγιναν και στην πρώτη έκδοση υλοποίησης του αλγορίθμου, όμως στη συγκεκριμένη περίπτωση το φαινόμενο είναι εντονότερο.

- Τέλος ο μικρός βαθμός παραλληλοποίησης εκδηλώνεται επίσης μέσω των διαγραμμάτων της επιτάχυνσης, στα οποία φαίνεται ότι το παράλληλο πρόγραμμα είναι κατα μέγιστο 1.75 φορές γρηγορότερο από το αντίστοιχο σειριακό.

Tiled έκδοση αλγορίθμου Floyd-Warshall

Παραλληλοποίηση:

Σε αυτή την περίπτωση η διαδικασία υπολογισμού των δεδομένων μιας αναδρομής είναι η εξής: Ο αρχικός πίνακας A χωρίζεται σε 9 επιμέρους πίνακες όπως φαίνεται στο σχήμα.



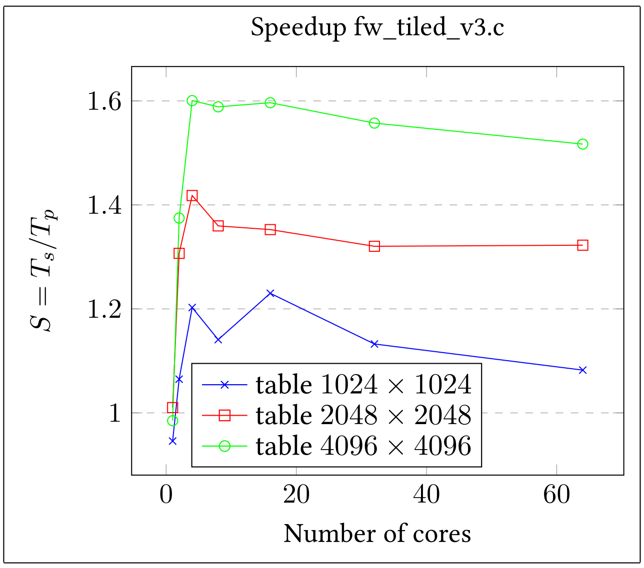
Αρχικά υπολογίζεται ο κεντρικός υποπίνακας K, ο οποίος εξαρτάται από τις δικές του τιμές. Έπειτα υπολογίζονται οι υποπίνακες B1, B2, B3 και B4, οι οποίοι για την ανανέωση των στοιχείων τους, χρησιμοποιούν τα δεδομένα του K, αλλά δεν έχουν καμία εξάρτηση μεταξύ τους (επομένως ο υπολογισμός τους μπορεί να παραλληλοποιηθεί). Τέλος μπορούν να υπολογιστούν τα στοιχεία των "άκρων" A1, A2, A3 και A4, οι οποίοι χρησιμοποιούν τις τιμές όλων των παραπάνω πινάκων. Σημειώνεται ότι τα επιμέρους for loops μπορούν να παραλληλοποιηθούν,καθώς για τον υπολογισμό 1 στοιχείου στο εσωτερικό του υποπίνακα δεν απαιτούνται νέες τιμές άλλων στοίχειων. Αυτή η διαδικασία ισχύει για δεδομένο k, δηλαδή ο εξωτερικός βρόχος εκτελείται σειριακά.

Για αυτήν την έκδοση του αλγορίθμου έγιναν δύο υλοποιήσεις, στις οποίες χρησιμοποιήθηκαν οι μηχανισμοί tasks του openMP. Στην πρώτη περίπτωση (fw_tiled_v3.c) χωρίζεται κάθε κομμάτι του πίνακα σε ένα task ώστε να τρέχουν παράλληλα. Στην δεύτερη (fw_tiled_v4.c), για να γίνεται καλύτερη κατανομή του όγκου εργασίας, δημιουργείται ένα task για κάθε κλήση της συνάρτηση FW.

Ακολουθούν τα αποτελέσματα των πειραματικών μετρήσεων και τα αντίστοιχα διαγράμματα επιτάχυνσης:

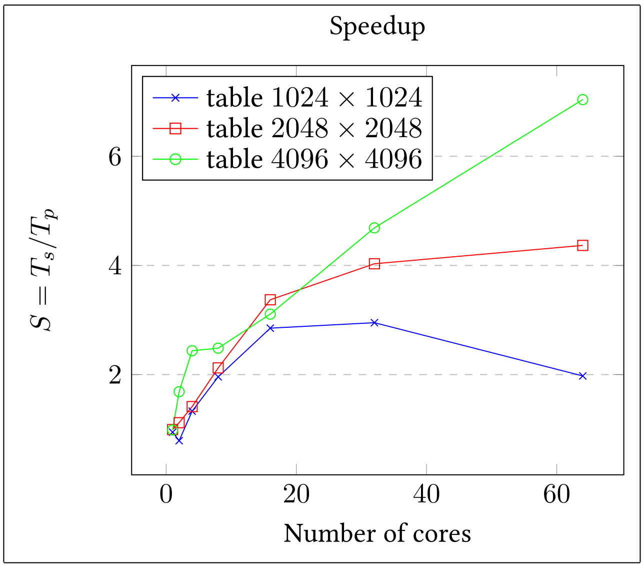
fw_tiled_v3.c, B=128

	64×64	1024×1024	2048×2048
1	0.7005	4.7669	36.5207
2	0.6222	3.685	26.1649
4	0.5509	3.3958	22.4724
8	0.5807	3.5418	22.6417
16	0.5386	3.5604	22.53
32	0.585	3.647	23.0965
64	0.6121	3.6412	23.7114



fw_tiled_v4.c, B=128

	64 × 64	1024 × 1024	2048 × 2048
1	0.7016	4.8493	36.5177
2	0.8401	4.3056	21.3278
4	0.4981	3.4067	14.7587
8	0.3382	2.2688	14.4806
16	0.2324	1.4285	11.574
32	0.2245	1.1947	7.6722
64	0.3354	1.1027	5.1102



Σημειώνεται ότι για αυτήν την έκδοση του σειριακού κώδικα υλοποιήθηκαν διάφορες τροποποιήσεις, όμως η επίδοση τους είχε ελάχιστη διαφορά, επομένως δε συμπεριλαμβάνονται στην αναφορά. Συγκεκριμένα παραλληλοποιώντας το for της συνάρτησης FW() η επίδοση είναι καλύτερη μόνο στην περίπτωση B=N,ενώ για τις υπόλοιπες εισόδους ελάχιστα καλύτερη επίδοση παρουσιάζει η υλοποίηση fw_par.c. Επίσης επιχειρήθηκε ο διαμοιρασμός των tasks στις επεξεργαστικές μονάδες μέσω της παραλληλοποίησης των for loops στα οποία γίνεται η κλήση της FW(), το οποίο όμως οδηγεί σε επίδοση όμοια με αυτή της fw_tiled_v4.c υλοποίησης.

Παρατηρήσεις:

- Όπως ήταν αναμενόμενο, η υλοποίηση fw_tiled_v4.c είχε πολύ καλύτερο χρόνο, αφού στην fw_tiled_v3.c τα tasks, τα οποία δημιουργούνται δεν είναι το ίδιο απαιτητικά.
- Οι παραπάνω υλοποιήσεις εκτελέστηκαν επίσης για μέγεθος B=256. Οι επιδόσεις, οι οποίες προέκυψαν είναι όμοιες με αυτές των εκτελέσεων για B=128 (για B=128 η επίδοση παρουσιάζει μικρή βελτίωση).

Γενικές Παρατηρήσεις:

- Με βάση τα παραπάνω διαγράμματα επιτάχυνσης προκύπτει ότι η πρώτη και η τρίτη έκδοση του αλγορίθμου είναι αρκετά παραλληλοποιήσιμες, ενώ από πλευράς χρόνου εκτέλεσης, η tiled έκδοση είναι η πιο αποδοτική για μεγαλύτερες διαστάσεις πίνακα (για N=1024 καλύτερη επίδοση σημειώνεται στην εκτέλεση της πρώτης έκδοσης).
- Επίσης η εκτέλεση του αλγορίθμου στις δύο τελευταίες εκδόσεις, επηρεάζεται άμεσα από το μέγεθος του υποπίνακα B, το οποίο εισάγει ο χρήστης κάθε φορά. Συγκεκριμένα παρουσιάζεται παρακάτω η εκτέλεση της fw_sr_v1.c έκδοσης για B ίσο με 64 και 256. Η καλύτερη επίδοση επιτυγχάνεται για μέγεθος 128, δηλαδή όχι πολύ μικρό ούτε πολύ μεγάλο. Αυτό πιθανώς να οφείλεται στη χωρητικότητα των caches L1 (32KB) και L2(256KB), καθώς όσο μεγαλύτερο είναι το μέγεθος του πίνακα, μπορεί να απαιτούνται περισσότερες αντικαταστάσεις/μεταφορές δεδομένων από και πρός την L2 (τόσο τοπικά όσο και σε άλλους επεξεργαστές του κόμβου).

fw_sr_v3.c, B=64

	64×64	1024×1024	2048×2048
1	0.6825	5.0661	40.2866
2	0.5944	3.7938	29.2861
4	0.5348	3.6456	24.7316
8	0.5321	3.7792	25.3017
16	0.5543	3.7642	25.04
32	0.5712	3.8326	25.7236
64	0.6602	3.9645	26.6987

fw_tiled_v3.c, B=128

	64×64	1024×1024	2048×2048
1	0.659	4.5976	35.0384
2	0.4292	2.7382	19.2022
4	0.4292	2.8877	19.8917
8	0.4399	3.0415	20.1826
16	0.4777	3.1529	20.4648
32	0.6034	3.4597	21.9576
64	0.7596	3.991	24.1684

fw_sr_v3.c, B=256

	64×64	1024×1024	2048×2048
1	0.8119	5.3882	42.8581
2	0.5987	3.6258	25.3271
4	0.5686	3.6183	25.6233
8	0.5669	3.5566	25.845
16	0.6084	3.699	26.5595
32	0.742	4.2653	27.8529
64	0.794	4.5134	28.8298

Κώδικας

fw_par.c

```
/*
 Parallel implementation of the Floyd-Warshall Algorithm
*/

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

inline int min(int a, int b);

int main(int argc, char **argv)
{
    int **A, **B, **C;
    int i,j,k;
    struct timeval t1, t2;
    double time;
    int N=1024;

    if (argc != 2) {
        fprintf(stdout,"Usage: %s N\n", argv[0]);
        exit(0);
    }

    N=atoi(argv[1]);

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

    B = (int **) malloc(N*sizeof(int *));
    for(i=0; i<N; i++) B[i] = (int *) malloc(N*sizeof(int));

    graph_init_random(A,-1,N,128*N);

    gettimeofday(&t1,0);
    for(k=0;k<N;k++) {
        #pragma omp parallel for private(i, j)
            for(i=0; i<N; i++)
                for(j=0; j<N; j++)
                    B[i][j]=min(A[i][j], A[i][k] + A[k][j]);
    }

    gettimeofday(&t2,0);
```

```
    time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.
       tv_usec)/1000000;
    printf("FW␣parallel,%d,%.4f\n", N, time);

    /*
    for(i=0; i<N; i++)
        for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
    */

    return 0;
}

inline int min(int a, int b)
{
    if(a<=b)return a;
    else return b;
}
```

fw_sr_v2.c

```
/*
  Recursive implementation of the Floyd-Warshall algorithm.
  command line arguments: N, B
  N = size of graph
  B = size of submatrix when recursion stops
  works only for N, B = 2^k
*/

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

inline int min(int a, int b);
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 main(int argc, char **argv)
{
    int **A;
    int i,j;
    struct timeval t1, t2;
    double time;
    int B=16;
    int N=1024;
```

```
    if (argc !=3){
        fprintf(stdout, "Usage␣%s␣N␣B␣\n", argv[0]);
        exit(0);
    }

    N=atoi(argv[1]);
    B=atoi(argv[2]);

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

    graph_init_random(A,-1,N,128*N);

    gettimeofday(&t1,0);
    FW_SR(A,0,0, A,0,0,A,0,0,N,B);
    gettimeofday(&t2,0);

    time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.
       tv_usec)/1000000;
    printf("FW_SR,%d,%d,%.4f\n", N, B, time);

    /*
    for(i=0; i<N; i++)
       for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
    */

    return 0;
}

inline int min(int a, int b)
{
    if(a<=b)return a;
    else return b;
}

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 {
```

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

                #pragma omp task
                    FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow,
                       ccol+myN/2, myN/2, bsize);
                #pragma omp task if (0)
                    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 if (0)
                    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);
            }
        }
    }
}
```

fw_sr_v3.c

```
/*
  Recursive implementation of the Floyd-Warshall algorithm.
  command line arguments: N, B
  N = size of graph
  B = size of submatrix when recursion stops
  works only for N, B = 2^k
*/

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <sys/time.h>
#include "util.h"

inline int min(int a, int b);
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 main(int argc, char **argv)
{
    int **A;
    int i,j;
    struct timeval t1, t2;
    double time;
    int B=16;
    int N=1024;


    if (argc !=3){
        fprintf(stdout, "Usage␣%s␣N␣B␣\n", argv[0]);
        exit(0);
    }

    N=atoi(argv[1]);
    B=atoi(argv[2]);

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

    graph_init_random(A,-1,N,128*N);

    gettimeofday(&t1,0);
    FW_SR(A,0,0, A,0,0,A,0,0,N,B);
    gettimeofday(&t2,0);

    time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.
       tv_usec)/1000000;
    printf("FW_SR,%d,%d,%.4f\n", N, B, time);

    /*
    for(i=0; i<N; i++)
       for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
    */

    return 0;
}

inline int min(int a, int b)
```

```
{
    if(a<=b)return a;
    else return b;
}

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

    if(myN<=bsize)
        for(k=0; k<myN; k++)
            // #pragma omp parallel for private (i, j)
            for(i=0; i<myN; i++)
                for(j=0; j<myN; j++)
                    A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+
                       i][bcol+k]+C[crow+k][ccol+j]);
    else {
        FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);

        #pragma omp parallel
        {
            #pragma omp single
            {
               #pragma omp task
               FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+
                  myN/2, myN/2, bsize);
               #pragma omp task if (0)
               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 parallel
        {
            #pragma omp single
            {
               #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 if (0)
```

```
                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);


    }
}
```

fw_tiled_v3.c

```
/*
  Tiled version of the Floyd-Warshall algorithm.
  command-line arguments: N, B
  N = size of graph
  B = size of tile
  works only when N is a multiple of B
*/

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

inline int min(int a, int b);
inline void FW(int **A, int K, int I, int J, int N);

int main(int argc, char **argv)
{
    int **A;
    int i,j,k;
    struct timeval t1, t2;
    double time;
    int B=64;
    int N=1024;

    if (argc != 3){
        fprintf(stdout, "Usage %s N B\n", argv[0]);
        exit(0);
    }

    N=atoi(argv[1]);
    B=atoi(argv[2]);

    A=(int **)malloc(N*sizeof(int *));
```

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

    graph_init_random(A,-1,N,128*N);

    gettimeofday(&t1,0);

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

        /* 1 */
        FW(A,k,k,k,B);

        #pragma omp parallel
        {
            #pragma omp single
            {
                #pragma omp task private(i, j)
                {
                    /* 2 */
                    for(i=0; i<k; i+=B) {
                            FW(A,k,i,k,B);
                    }
                }

                #pragma omp task private(i, j)
                {
                    /* 3 */
                    for(i=k+B; i<N; i+=B) {
                        FW(A,k,i,k,B);
                    }
                }

                #pragma omp task private(i, j)
                {
                    /* 4 */
                    for(j=0; j<k; j+=B) {
                            FW(A,k,k,j,B);
                    }
                }

                #pragma omp task private(i, j)
                {
                    /* 5 */
                    for(j=k+B; j<N; j+=B) {
                            FW(A,k,k,j,B);
                    }
                }

                #pragma omp taskwait
```

```
                #pragma omp task private(i, j)
                {
                    /* 6 */
                    for(i=0; i<k; i+=B)
                        for(j=0; j<k; j+=B) {
                            FW(A,k,i,j,B);
                        }
                }

                #pragma omp task private(i, j)
                {
                    /* 7 */
                    for(i=0; i<k; i+=B)
                        for(j=k+B; j<N; j+=B) {
                            FW(A,k,i,j,B);
                        }
                }

                #pragma omp task private(i, j)
                {
                    /* 8 */
                    for(i=k+B; i<N; i+=B)
                        for(j=0; j<k; j+=B) {
                            FW(A,k,i,j,B);
                        }
                }

                #pragma omp task private(i, j)
                {
                    /* 9 */
                    for(i=k+B; i<N; i+=B)
                        for(j=k+B; j<N; j+=B) {
                            FW(A,k,i,j,B);
                        }
                }

            }

        }

    }
    gettimeofday(&t2,0);

    time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.
       tv_usec)/1000000;
    printf("FW_TILED,%d,%d,%.4f\n", N,B,time);

    /*
    for(i=0; i<N; i++)
```

```
        for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
    */

    return 0;
}

inline int min(int a, int b)
{
    if (a<=b) return a;
    else return b;
}

inline void FW(int **A, int K, int I, int J, int N)
{
    int i,j,k,nthreads;

    for(k=K; k<K+N; k++)
        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]);
        }
}
```

fw_tiled_v4.c

```
/*
  Tiled version of the Floyd-Warshall algorithm.
  command-line arguments: N, B
  N = size of graph
  B = size of tile
  works only when N is a multiple of B
*/

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

inline int min(int a, int b);
inline void FW(int **A, int K, int I, int J, int N);

int main(int argc, char **argv)
{
    int **A;
    int i,j,k;
    struct timeval t1, t2;
    double time;
    int B=64;
    int N=1024;
```

```
if (argc != 3){
    fprintf(stdout, "Usage␣%s␣N␣B\n", argv[0]);
    exit(0);
}

N=atoi(argv[1]);
B=atoi(argv[2]);

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

graph_init_random(A,-1,N,128*N);

gettimeofday(&t1,0);

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

    /* K */
    FW(A,k,k,k,B);

    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task private(i, j)
            {
                /* 2 */
                for(i=0; i<k; i+=B) {
                    #pragma omp task firstprivate (i)
                        FW(A,k,i,k,B);
                }
            }

            #pragma omp task private(i, j)
            {
                /* 3 */
                for(i=k+B; i<N; i+=B)
                    #pragma omp task firstprivate (i)
                        FW(A,k,i,k,B);
                }
            }

            #pragma omp task private(i, j)
            {
                /* 4 */
                for(j=0; j<k; j+=B) {
                    #pragma omp task firstprivate (j)
                        FW(A,k,k,j,B);
```

```
        }
    }

    #pragma omp task private(i, j)
    {
        /* 5 */
        for(j=k+B; j<N; j+=B) {
            #pragma omp task firstprivate (j)
                FW(A,k,k,j,B);
        }
    }

    #pragma omp taskwait

    #pragma omp task private(i, j)
    {
        /* 6 */
        for(i=0; i<k; i+=B)
            for(j=0; j<k; j+=B) {
                #pragma omp task firstprivate (i, j)
                    FW(A,k,i,j,B);
            }
    }

    #pragma omp task private(i, j)
    {
        /* 7 */
        for(i=0; i<k; i+=B)
            for(j=k+B; j<N; j+=B) {
                #pragma omp task firstprivate (i, j)
                    FW(A,k,i,j,B);
            }
    }

    #pragma omp task private(i, j)
    {
        /* 8 */
        for(i=k+B; i<N; i+=B)
            for(j=0; j<k; j+=B) {
                #pragma omp task firstprivate (i, j)
                    FW(A,k,i,j,B);
            }
    }

    #pragma omp task private(i, j)
    {
        /* 9 */
        for(i=k+B; i<N; i+=B)
            for(j=k+B; j<N; j+=B) {
```

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

            }

        }

    }
    gettimeofday(&t2,0);

    time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.
       tv_usec)/1000000;
    printf("FW_TILED,%d,%d,%.4f\n", N,B,time);

    /*
    for(i=0; i<N; i++)
        for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
    */

    return 0;
}

inline int min(int a, int b)
{
    if (a<=b) return a;
    else return b;
}

inline void FW(int **A, int K, int I, int J, int N)
{
    int i,j,k,nthreads;

    for(k=K; k<K+N; k++)
        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]);
        }
}
```