Παράλληλη επεξεργασία

ΠΕΤΡΑΚΗΣ ΚΩΣΤΑΝΤΙΝΟΣ ΑΜ:5878

ΠΕΤΡΑΚΗΣ ΕΥΑΓΓΕΛΟΣ ΑΜ:6339

ΟΜΑΔΑ ΧΡΗΣΤΩΝ 8

Περιγραφή συστήματος

Όλα τα πειράματα υλοποιήθηκαν σε φορητό υπολογιστή με τα εξής χαρακτηριστικά.

Επεξεργαστής Intel(R) Core™ i7-2630 QM CPU 2.00/2.90(Turbo boost) GHz

Ο εν λόγω επεξεργαστής διαθέτει 4 πυρήνες(8 εικονικούς)

Κρυφή μνήμη: Η κρυφή μνήμη, μεγέθους 6MB, αποτελείται από 3 επίπεδα L1,L2,L3

L1 cache: 256KB

L2 cache: 1MB

L3 cache: 6MB

Στα πλαίσια της άσκησης χρησιμοποιήθηκε εικονική μηχανή VirtualBox για την υλοποίηση των ερωτημάτων σε λειτουργικό σύστημα Ubuntu 18.04 LTS.

Για τα ερωτήματα που αφορούν την βελτίωση του σειριακού προγράμματος οπου αναφέρουμε ποσοστιαία χρονοβελτίωση στους πίνακες έχουμε χρησιμοποιήσει τον τύπο

Χρονοβελτιώση = $\frac{t_{αρχ}-t_{τελ}}{t_{αρχ}} * 100$ %. Η ίδια μετρική χρησιμοποιήθηκε και για την ποσοστιαία χρονοβελτίωση στους πίνακες που αφορούν την παραλληλοποίηση του προγράμματος. Ενώ στην στήλη που αναφέρουμε ως speedup έχουμε υπολογίσει κανονικά την χρονοβελτίωση που ζητήται από την άσκηση από την γνωστή σχέση speedup=$\frac{t_{σειριακό}}{t_{παράλληλο}}$.

(Να σημείωσουμε ότι τα 2 μέτρα συμφωνούν πλήρως όπως παρατηρούμε και από τους παρακάτω πίνακες και ο λόγος που συμπεριλάβαμε και τα ποσοστά στους πίνακες είναι διότι γίνεται πιο σαφές πια από τις 3 παράλληλες υλοποιήσεις είναι καλύτερη όταν οι τιμές για την μετρική speedup είναι πολύ κοντά αλλά και για λόγους πληρότητας των μετρήσεων.)

Για το ακολουθιακό πρόγραμμα που δίνεται οι μετρήσεις που προκύπτουν και χρησιμοποιούνται παρακάτω είναι

Παράμετροι	**Ακολουθιακό πρόγραμμα που δίνεται**
--n 1000 --r 350	371.124782
--n 2000 --r 700	1476.733133
--n 3000 --r 1000	3264.412432
--n 4000 --r 1300	5910.104436
--n 5000 –r 1600	8794.441821

1a) Βελτιστοποίηση αντιγραφών

Αντικαταστήσαμε την εντολή u[i]=uplus[i] στον δεύτερο βρόγχο for(i=0;i<n;i++) με τις εντολές

temp=u;

u=uplus;

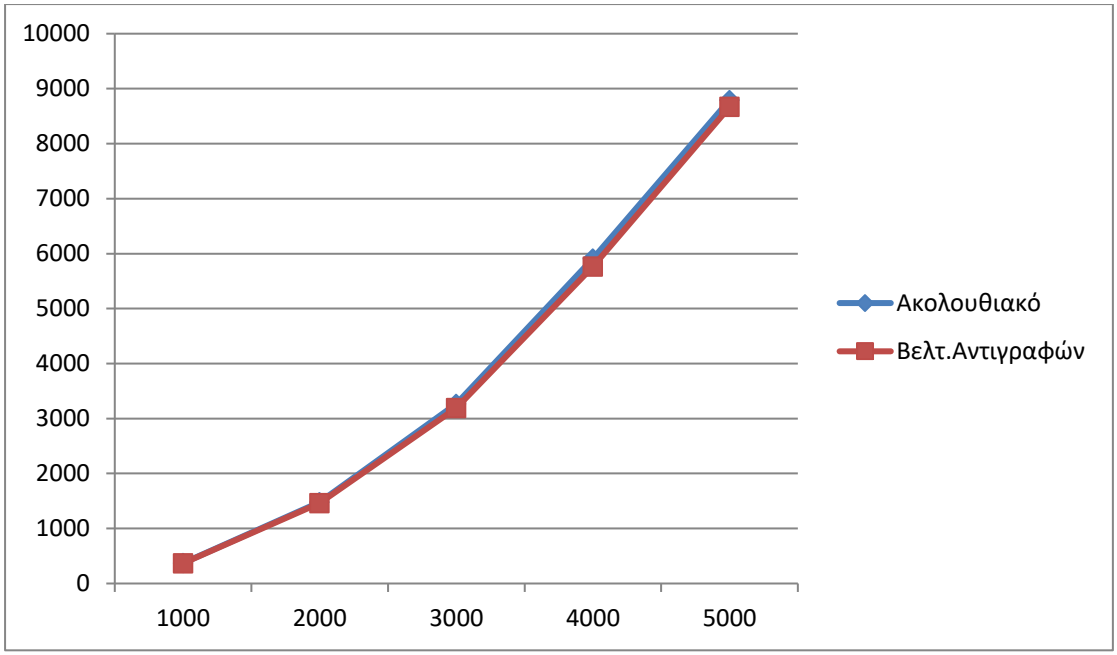
uplus=temp;

έξω από τον βρόγχο for(i=0;i<n;i++). Έτσι αντιγράφουμε τα περιεχόμενα του ένα δείκτη στον άλλον και όχι όλα τα στοιχεία του πίνακα ένα προς ένα.

Οι παρακάτω μετρήσεις προέκυψαν με τις βελτιστοποιήσεις από τον μεταφραστή ,δηλαδή το πρόγραμμα έγινε μεταγλώττιση με την εντολή gcc –O3 –Wall –Wextra.

Παράμετροι	**Ακολουθιακό πρόγραμμα που δίνεται**	**Ακολουθιακό πρόγραμμα με βελτιστοποίηση αντιγραφών**	**Ποσοστιάια Χρονοβελτίωση**
--n 1000 --r 350	371.124782	367.441527	1%
--n 2000 --r 700	1476.733133	1463.387830	0,8%
--n 3000 --r 1000	3264.412432	3192.770727	2,2%
--n 4000 --r 1300	5910.104436	5767.791000	2,41%
--n 5000 –r 1600	8794.441821	8667.081280	1,44%

Όλες οι μετρήσεις είναι σε δευτερόλεπτα. Παρατηρούμε μικρή σχετικά βελτίωση. Θα είχαμε μεγαλύτερη εξοικονόμηση χρόνου με την βελτιστοποίηση αντιγραφών αν η είσοδος ήταν πάρα πολύ μεγάλη(π.χ 50000 νευρώνες).Το διάγραμμα της χρονοβελτίωσης ανάλογα με τον αριθμό νευρώνων:



Παρατηρούμε ότι η μέση ποσοστιαία χρονοβελτίωση είναι κοντά στο 2%. Αναμένουμε η βελτίωση αντιγραφών να έχει καλύτερη συμπεριφορά για πολύ μεγαλύτερες εισόδους.

1b) Αναδιοργάνωση υπολογισμών

Για τη αναδιοργάνωση των υπολογισμών αρχικά εκχωρήσαμε τις τιμές κάποιων σταθερών που χρησιμοποιούνται επανειλημμένα στους υπολογισμού σε κάποιες σταθερές έξω από τους βρόγχους. Στη συνέχεια «σπάμε» το άθροισμα σε 2 αθροίσματα και αφού τα u[i] είναι σταθερά για κάθε χρονική στιγμή μπορούμε να υπολογίσουμε μια φορά το άθροισμα $\sum \sigma_{ij}$ έξω από τους βρόγχους επανάληψης και να αποθηκεύσουμε τα δεδομένα σε ένα διάνυσμα(temp_vec) και να διατρέχουμε αυτό το διάνυσμα όταν απαιτείται. Ο κώδικας που προκύπτει φαίνεται παρακάτω:

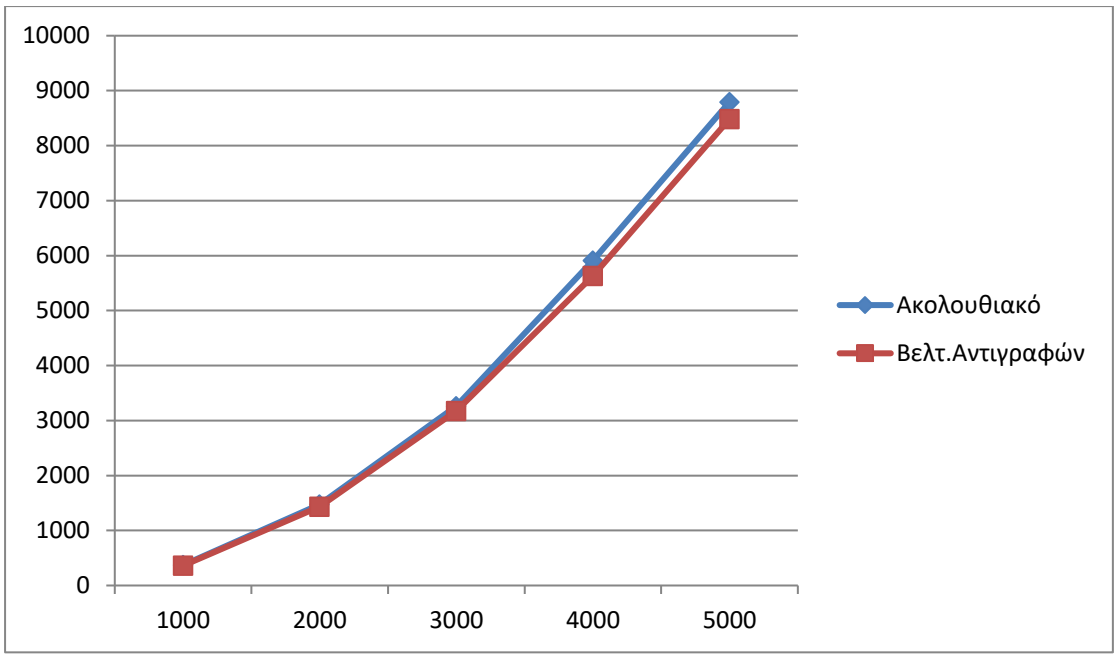
```
temp_vec = (double *)calloc(n, sizeof(double));
        if (temp_vec == NULL) {
                printf("Could not allocate memory for \"temp_vec\".\n");
                exit(1);
        }
        for(i=0;i<n;i++){
                temp_vec[i]=0;
                for(j=0;j<n;j++){
                        temp_vec[i]+=sigma[i*n+j];
                }
```

```
}


c1=dt*mu;

c2=dt/divide;

gettimeofday(&global_start, NULL);

for (it = 0; it < itime; it++) {

        /*

         * Iteration over elements.

         */

        for (i = 0; i < n; i++) {

                uplus[i] = u[i] + c1-dt*u[i];

                sum = 0.0;

                /*

                 * Iteration over neighbouring neurons.

                 */

                for (j = 0; j < n; j++) {

                        sum += sigma[i * n + j] * u[j];

                }

                uplus[i] += c2 * (sum-u[i]*temp_vec[i]);

        }
```

Εξετάζουμε την περίπτωση με τις βελτιστοποιήσεις του μεταφραστή δηλαδή η μεταγλώττιση του προγράμματος έγινε με την εντολή gcc –O3 –Wall –Wextra και οι χρόνοι που προκύπτουν είναι:

Παράμετροι	Ακολουθιακό πρόγραμμα που δίνεται	Ακολουθιακό πρόγραμμα με αναδιοργάνωση υπολογισμών	Ποσοστιάια Χρονοβελτίωση
--n 1000 --r 350	371.124782	361.499120	2.9%
--n 2000 --r 700	1476.733133	1433.365758	2.9%
--n 3000 --r 1000	3264.412432	3173.111676	2,7%
--n 4000 --r 1300	5910.104436	5630.593118	4,7%
--n 5000 –r 1600	8794.441821	8484.588924	3,5%

Και το διάγραμμα χρονοβελτίωσης είναι



Εδώ η ποσοστιαία χρονοβελτίωση είναι σταθερά μεγαλύτερη από ότι στην περίπτωση της βελτιστοποίησης των αντιγραφών. Και εδώ περιμένουμε ότι όσο μεγαλώνει η είσοδος(ο αριθμός των νευρώνων) τόσο θα αυξάνεται και η διαφορά των δυο γραμμών δηλαδή η χρονική βελτίωση που πετυχαίνουμε με την αναδιοργάνωση των υπολογισμών.

1c) Χρήση BLAS

(Χρησιμοποιήσαμε το πακέτο libblas-dev)

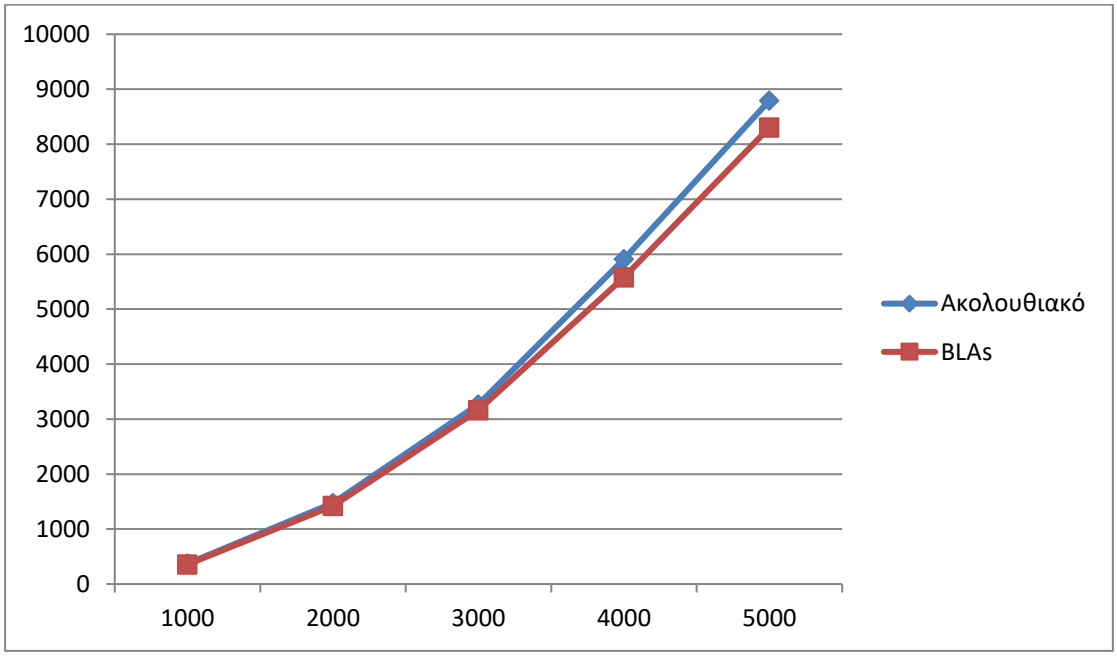
Εδώ απλά αντικαθιστούμε τον εσωτερικό βρόγχο for(j=0;j<n;j++) που υπολογίζει το γινόμενο $\sum \sigma_{ij} * u_j$. Ο υπολογισμός αντιστοιχεί σε πολλαπλασιασμό μητρώου με διάνυσμα επομένως χρησιμοποιούμε την συνάρτηση cblas_dgemv της βιβλιοθήκης BLAS. Ο κώδικας ακολουθεί(συμπεριλάβαμε στον κώδικα το header file <cblas.h>και στο διάνυσμα v αποθηκεύουμε το αποτέλεσμα):

```
v = (double *)calloc(n, sizeof(double));
        if (v == NULL) {
                printf("Could not allocate memory for \"v\".\n");
                exit(1);
        }
c1=dt*mu;
c2=dt/divide;
        gettimeofday(&global_start, NULL);
        for (it = 0; it < itime; it++) {
                /*
                 * Iteration over elements.
                 */
                cblas_dgemv(CblasRowMajor,CblasNoTrans,n,n,1.0,sigma,n,u,1,0,v,1);
                for (i = 0; i < n; i++) {
                        uplus[i] = u[i] + c1-dt*u[i] +c2 * (v[i]-u[i]*temp_vec[i]);
                        /*
                         * Iteration over neighbouring neurons.
                         */
                }
```

Εξετάζουμε την περίπτωση με τις βελτιστοποιήσεις του μεταφραστή δηλαδή η μεταγλώττιση του προγράμματος έγινε με την εντολή gcc –O3 –Wall –Wextra –lblas και οι χρόνοι που προκύπτουν είναι:

Παράμετροι	Ακολουθιακό πρόγραμμα που δίνεται	Ακολουθιακό πρόγραμμα με αναδιοργάνωση υπολογισμών(χρήση BLAS)	Ποσοστιάια Χρονοβελτίωση
--n 1000 --r 350	371.124782	357.673937	4,0%
--n 2000 --r 700	1476.733133	1420.051471	3,8%
--n 3000 --r 1000	3264.412432	3160.228535	6,6%
--n 4000 --r 1300	5910.104436	5580.422055	5,5%
--n 5000 –r 1600	8794.441821	8304.952085	5,6%

Παρατηρούμε ποσοστιαία χρονοβελτίωση της τάξης του 2% με την χρήση συνάρτησης BLAS για τον υπολογισμό μητρώου επι διάνυσμα σε σχέση με την περίπτωση 1.b οπου χρησιμοποιούσαμε βρόγχο. Και μέση βελτίωση 5,5% σε σχέση με το ακολουθιακό πρόγραμμα που δίνεται. Ακολουθεί το διάγραμμα χρονοβελτιώσης.



2a) <u>Παραληλοποίηση του βρόγχου (it)</u>

Για την παραλληλοποίηση του βρόγχου it προσθέσαμε τις εντολές:

#pragma omp parallel(it) έξω από τον βρόγχο it

#pragma omp for private(j) έξω από τον βρόγχο i

#pragma omp barrier

#pragma omp single

{ temp=u;

u=uplus;

uplus=temp;} για να εκτελέσει την αντιγραφή δεικτών μόνο ένα νήμα

και την εντολή #pragma omp single πριν το κομμάτι του κώδικα που αφορά το γράψιμο των αποτελεσμάτων στο αρχείο.

Να σημειώσουμε εδώ ότι δεν χρειάστηκε να χρησιμοποιήσουμε και δεύτερη εντολή #pragma omp barrier πριν την εντολή #pragma omp single για το γράψιμο στο αρχείο καθώς αυτή ακολουθεί την προηγούμενη εντολή #pragma omp single, για την εναλλαγή των δεικτών, η οποία εγγυάται ότι όλα τα νήματα έχουν φτάσει σε αυτό το σημείο πριν προχωρήσουμε. Ο κώδικας του υπολογισμού είναι:

```
c1=dt*mu;
        c2=dt/divide;
        gettimeofday(&global_start, NULL);
        #pragma omp parallel private(it)
{
        for (it = 0; it < itime; it++) {
                /*
                 * Iteration over elements.
                 */
                #pragma omp for private(j)
                for (i = 0; i < n; i++) {
                        uplus[i] = u[i] + c1-dt*u[i];
                        sum = 0.0;
                        /*
                         * Iteration over neighbouring neurons.
                         */
                        for (j = 0; j < n; j++) {
```

```
                sum += sigma[i * n + j] * u[j];
        }
        uplus[i] += c2 * (sum-u[i]*temp_vec[i]);
}

/*
 * Update network elements and set u[i] = 0 if u[i] > uth
 */
#pragma omp barrier
#pragma omp single
{
temp=u;
u=uplus;
uplus=temp;     }
for (i = 0; i < n; i++) {

        if (u[i] > uth) {
                u[i] = 0.0;
                /*
                 * Calculate omega's.
                 */
                if (it >= ttransient) {
                        omega1[i] += 1.0;
                }
        }
}
```

```
                    /*
                     * Print out of results.
                     */
                    #pragma omp single
                    {
#if !defined(ALL_RESULTS)
                    if (it % ntstep == 0) {
#endif
                            printf("Time is %ld\n", it);
                            gettimeofday(&IO_start, NULL);
                            fprintf(output1, "%ld\t", it);
                            for (i = 0; i < n; i++) {
                                    fprintf(output1, "%19.15f", *(u+i));
                            }
                            fprintf(output1, "\n");

                            time = (double)it * dt;
                            fprintf(output2, "%ld\t", it);
                            for (i = 0; i < n; i++) {
                                    omega[i] = 2.0 * M_PI * omega1[i] / (time - ttransient * dt);
                                    fprintf(output2, "%19.15f", omega[i]);
                            }
                            fprintf(output2, "\n");
                            gettimeofday(&IO_end, NULL);
                            IO_usec += ((IO_end.tv_sec - IO_start.tv_sec) * 1000000.0 +
(IO_end.tv_usec - IO_start.tv_usec));
#if !defined(ALL_RESULTS)
                    }
```

```
#endif

}

        }

}
```

Εξετάζουμε την περίπτωση με τις βελτιστοποιήσεις του μεταφραστή δηλαδή η μεταγλώττιση του προγράμματος έγινε με την εντολή gcc –O3 – fopenmp -Wall –Wextra και οι χρόνοι για την εκτέλεση του προγράμματος με 1 νήμα είναι

Παράμετροι	**Ακολουθιακό πρόγραμμα που δίνεται**	**Παράλληλο Πρόγραμμα(βρόγχος it)threads=1**	**Ποσοστιάια Χρονοβελτίωση**	**speedup**
--n 1000 --r 350	371.124782	371.144688	-	1.00
--n 2000 --r 700	1476.733133	1422.057729	-	1.03
--n 3000 --r 1000	3264.412432	3202.320140	-	1.01
--n 4000 --r 1300	5910.104436	5534.359022	-	1.06
--n 5000 –r 1600	8794.441821	8786.761633	-	1.00

Η χρονοβελτίωση βλέπουμε ότι είναι πάρα πολύ κοντά στο 1 οπότε και συμπαιρένουμε ότι η βελτίωση που έχουμε με 1 νήμα είναι εξαιρετικά μικρή.

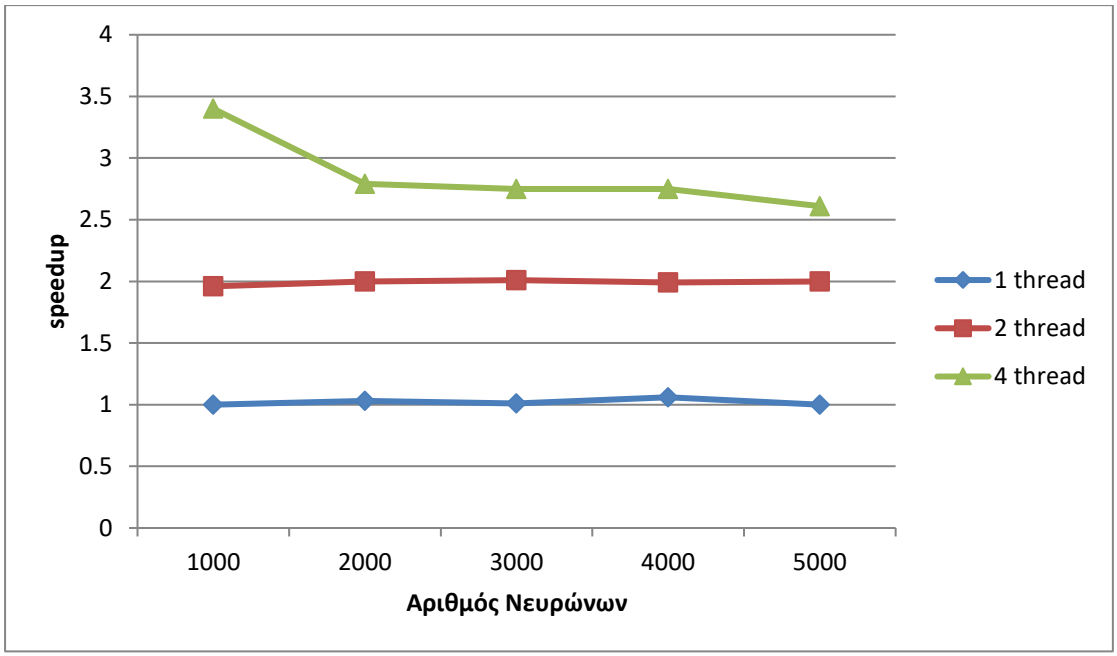
Οι χρόνοι που προκύπτουν με χρήση 2 νημάτων είναι:

Παράμετροι	**Ακολουθιακό πρόγραμμα που δίνεται**	**Παράλληλο Πρόγραμμα(βρόγχος it)threads=2**	**Ποσοστιαία Χρονοβελτίωση**	**Speedup**
--n 1000 --r 350	371.124782	189.791419	49.05%	1.96
--n 2000 --r 700	1476.733133	737.472861	50.06%	2.00
--n 3000 –r 1000	3264.412432	1616.823539	50.49%	2.01
--n 4000 --r 1300	5910.104436	2969.380247	49.76%	1.99
--n 5000 –r 1600	8794.441821	4390.851095	50.07%	2.00

Οι χρόνοι που προκύπτουν με 4 νήματα είναι:

Παράμετροι	**Ακολουθιακό πρόγραμμα που δίνεται**	**Παράλληλο Πρόγραμμα(βρόγχος it)threads=4**	**Ποσοστιαία Χρονοβελτίωση**	**speedup**
--n 1000 --r 350	371.124782	109.753229	70.61%	3.40
--n 2000 --r 700	1476.733133	529.595218	64.15%	2.79
--n 3000 --r 1000	3264.412432	1184.313168	63.75%	2.75
--n 4000 --r 1300	5910.104436	2149.049042	63.63%	2.75
--n 5000 –r 1600	8794.441821	3369.236441	61.68%	2.61

Το διάγραμμα χρονοβελτίωσης για τον βρόγχο (it) ανάλογα με τα threads που χρησιμοποιούμε:



Η χρονοβελτίωση για 1 thread είναι όπως θα αναμέναμε αμελητέα ενώ με χρήση 2 thread έχω χρονοβελτίωση σχεδόν σταθερή με τιμή 2. Για 4 thread η χρονοβελτίωση ξεκινά από την τιμή 3.4 και βλέπουμε ότι όσο αυξάνεται ο αριθμός των νευρώνων τείνει στην τιμή 2.5. Σημειώνουμε ότι σύμφωνα με την θεωρία θα έπρεπε να είναι speedup≤p όπου p ο αριθμός των επεξεργαστών που χρησιμοποιουμε για παραλληλία. Εδώ όπως βλέπουμε με 4 νήματα ισχύει η σχέση αλλά για 2 νήματα και 3000 νευρώνες παίρνουμε speedup=2.01>2. Και πάλι από την θεωρία γνωρίζουμε οτι μπορεί να συμβεί αυτό όταν έχουμε υπεργραμμική βελτίωση(superlinear speedup) με τον παράλληλο αλγόριθμο αλλά στην περίπτωση μας επειδή η τιμή είναι πολύ κοντά στο 2 δεν μπορούμε να θεωρήσουμε ότι πετύχαμε κάτι τέτοιο.

2b) <u>Παραληλοποίηση του βρόγχου (i)</u>

Για την παραλλολοποίηση του βρόγχου (i) προσθέσαμε τις εντολές ,

#pragma omp parallel private(j)

#pragma omp for

ακριβώς πριν από τον βρόγχο (i).Ο κώδικας του υπολογισμού είναι:

```
c1=dt*mu;
```

```
c2=dt/divide;
gettimeofday(&global_start, NULL);
for (it = 0; it < itime; it++) {
        /*
         * Iteration over elements.
         */
        #pragma omp parallel private(j)
        #pragma omp for
        for (i = 0; i < n; i++) {
                uplus[i] = u[i] + c1-dt*u[i];
                sum = 0.0;
                /*
                 * Iteration over neighbouring neurons.
                 */
                for (j = 0; j < n; j++) {
                        sum += sigma[i * n + j] * u[j];
                }
                uplus[i] += c2 * (sum-u[i]*temp_vec[i]);
        }
```

Εξετάζουμε την περίπτωση με τις βελτιστοποιήσεις του μεταφραστή δηλαδή η μεταγλώττιση του προγράμματος έγινε με την εντολή gcc –O3 – fopenmp -Wall –Wextra και οι χρόνοι για την εκτέλεση του προγράμματος με 1 νήμα είναι

Παράμετροι	Ακολουθιακό πρόγραμμα που δίνεται	Παράλληλο Πρόγραμμα(βρόγχος i)threads=1	Ποσοστιάια Χρονοβελτίωση	speedup
--n 1000 --r 350	371.124782	353.883967	-	1.05
--n 2000 --r 700	1476.733133	1412.312007	-	1.04
--n 3000 --r 1000	3264.412432	3180.341977	-	1.02
--n 4000 --r 1300	5910.104436	5492.321131	-	1.07
--n 5000 –r 1600	8794.441821	8702.902310	-	1.01

Η χρονοβελτίωση(speedup) βλέπουμε ότι είναι πολύ κοντά στο 1, δηλαδή η βελτίωση που παίρνουμε με 1 νήμα και σε αυτήν την περίπτωση είναι ελάχιστη.

και οι χρόνοι που προκύπτουν με χρήση 2 νημάτων είναι:

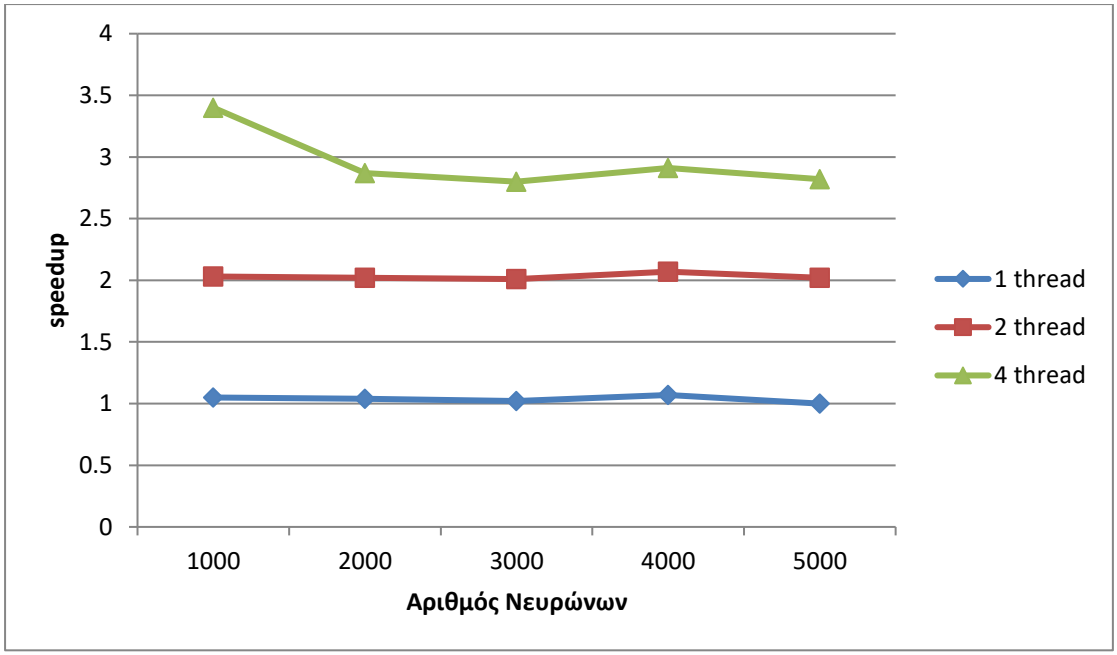
Παράμετροι	Ακολουθιακό πρόγραμμα που δίνεται	Παράλληλο Πρόγραμμα(βρόγχος i)threads=2	Ποσοστιαία Χρονοβελτίωση	Speedup
--n 1000 --r 350	371.124782	182.382405	50,9%	2.03
--n 2000 --r 700	1476.733133	730.500194	50,5%	2.02
--n 3000 --r 1000	3264.412432	1618.546500	50,4%	2.01
--n 4000 --r 1300	5910.104436	2843.910741	51,8%	2.07
--n 5000 –r 1600	8794.441821	4350.204056	50,5%	2.02

Με 4 νήματα έχουμε:

Παράμετροι	Ακολουθιακό πρόγραμμα που δίνεται	Παράλληλο Πρόγραμμα(βρόγχος i)threads=4	Ποσοστιαία Χρονοβελτίωση	speedup
--n 1000 --r 350	371.124782	109.440188	70,6%	3.40
--n 2000 --r 700	1476.733133	514.874181	65%	2.87
--n 3000 --r 1000	3264.412432	1164.450012	65%	2.80
--n 4000 --r 1300	5910.104436	2029.276094	64,3%	2.91
--n 5000 –r 1600	8794.441821	3112.880583	64,6%	2.82

(Και στους 3 παραπάνω πίνακες η χρονοβελτίωση είναι σε σχέση με το αρχικό ακολουθιακό πρόγραμμα.)

Τα διαγράμματα χρονοβελτίωσης για τον βρόγχο (i) ανάλογα με τα threads που χρησιμοποιούμε:



Παρατηρούμε ότι η χρονοβελτίωση για 1 και 2 threads είναι σχεδόν σταθερή, ενώ στην περίπτωση των 4 threads βλέπουμε μια μικρή πτώση όσο αυξάνεται το στιγμιότυπο της εισόδου αλλά και πάλι η χρονοβελτίωση παραμένει κοντά στην τιμή 3. Και εδώ οι τιμές που λαμβάνουμε για την χρονοβελτίωση συμφωνούν με την θεωρία(ότι δηλ. speedup≤p) για 4 νήματα, αλλά για την περίπτωση των 2 νημάτων παίρνουμε σε όλες τις περιπτώσεις τιμές μεγαλύτερες του 2 . Και πάλι δεν μπορούμε να θεωρήσουμε ότι πετύχαμε υπεργραμμική βελτίωση γιατί οι τιμές είναι πολύ κοντα στην τιμή 2 απλά είναι τέτοια η φύση του προγράμματος που επιδέχεται παραλληλοποίηση με πάρα πολύ καλά αποτελέσματα.

2c) Παραλληλοποίηση του βρόγχου (j)

Για την παραλληλοποίηση του βρόγχου (j) χρησιμοποιήσαμε τις εντολές

#pragma omp parallel

#pragma omp for reduction(+:sum)

Και ο κώδικας του υπολογισμού για την παραλληλοποίηση του βρόγχου (j) που προκύπτει είναι:

```
c1=dt*mu;
        c2=dt/divide;
        gettimeofday(&global_start, NULL);
        for (it = 0; it < itime; it++) {
                /*
```

```
 * Iteration over elements.
 */
for (i = 0; i < n; i++) {
        uplus[i] = u[i] + c1-dt*u[i];
        sum = 0.0;
        /*
         * Iteration over neighbouring neurons.
         */
        #pragma omp parallel
        #pragma omp for reduction(+:sum)
        for (j = 0; j < n; j++) {
                sum += sigma[i * n + j] * u[j];
        }
        uplus[i] += c2 * (sum-u[i]*temp_vec[i]);
}
```

Με χρήση του παραπάνω κώδικα και χρησιμοποιώντας τις βελτιστοποιήσεις του μεταφραστή, δηλαδή η μεταγλώττιση του προγράμματος έγινε με την εντολή gcc –O3 –fopenmp -Wall –Wextra, για 4 threads(που αναμένουμε και την καλύτερη επίδοση με 4 νήματα) και για είσοδο 1000 νευρώνες ο χρόνος εκτέλεσης του προγράμματος είναι 823.593438 sec. Βλέπουμε δηλαδή ότι το πρόγραμμα που προκύπτει είναι 2,2 φορές πιο αργό από ότι το αρχικό ακολουθιακό πρόγραμμα ακόμη και με την χρήση 4 νημάτων. Η συμπεριφορά του προγράμματος θα ήταν ακόμη χειρότερη με χρήση 2 ή ενός νήματος. Συνεπώς επειδή βλέπουμε ότι οι χρόνοι εκτέλεσης ξεφεύγουν πολύ γρήγορα δεν παραθέτουμε μετρήσεις για τα υπόλοιπα στιγμιότυπα εισόδου.

Αξίζει να σημειωθεί ότι η κακή αυτή χρονική συμπεριφορά του προγράμματος όταν παραλληλοποιούμε τον βρόγχο j είναι αναμενόμενη καθώς σε αυτήν την περίπτωση δημιουργούνται νήματα για κάθε μια εκτέλεση των εξωτερικών βρόγχων, δηλαδή 20*n φορές. Επίσης ο αριθμός των φορών που θα γίνουν join τα νήματα και τα barriers (φράγματα) που θα εκτελεστούν θα είναι σημαντικά μεγαλύτερος όταν παραλληλοποιούμε τον πιο εσωτερικό βρόγχο j.

2d)Προσθήκη βιβλιοθήκης BLAS στο παράλληλο πρόγραμμα

Από τους πίνακες παραπάνω συμπεραίνουμε ότι έχουμε λίγο καλύτερη συμπεριφορά(speedup κατά 0.1-0.2 μονάδες μεγαλύτερο) όταν παραλληλοποιούμε τον βρόγχο (i) .Έτσι επαναφέρουμε την χρήση της συνάρτησης BLAS για την εκτέλεση του γινόμενου πίνακα επι διάνυσμα στο πρόγραμμα που υλοποιήσαμε στο ερώτημα 2b παραπάνω. Βέβαια για να εφαρμόσουμε εδώ την συνάρτηση BLAS πρέπει να κάνουμε κάποιες τροποποιήσεις ώστε αυτή να εκτελείται παράλληλα. Η προσέγγιση μας εδώ είναι η εξής. Καλούμε την συνάρτηση BLAS μέσα στην παράλληλη περιοχή όπως και πριν και έξω από τον βρόγχο (i) αλλά για να μοιραστεί η εργασία της συνάρτησης στο κάθε νήμα αλλάξαμε το 7^{ο} όρισμα της συνάρτησης που δηλώνει πόσες γραμμές του πίνακα θα χρησιμοποιήσει για τον πολλαπλασιασμό. Η τιμή που είχαμε πριν στο 7^{ο} όρισμα ήταν η συνολική διάσταση του πίνακα δηλαδή ζητούσαμε από την συνάρτηση να επεξεργαστεί όλον τον πίνακα. Τώρα όπως φαίνεται και στον κώδικα που ακολουθεί σ'αυτό το όρισμα περνάμε την τιμή numofLines, που προκύπτει ανάλογα με τον αριθμό των threads που έχουμε και δηλώνει πόσες γραμμές του πίνακα θα χρησιμοποιήσει για τον πολλαπλασιασμό το κάθε thread. Ο κώδικας του υπολογισμού μετά και την προσθήκη της συνάρτησης BLAS είναι:

```
c1=dt*mu;
        c2=dt/divide;
        for (it = 0; it < itime; it++) {
                /*
                 * Iteration over elements.
                 */
                int numofThreads,numofLines,id;
                #pragma omp parallel private(numofThreads,numofLines,id)
                id=omp_get_thread_num();
                numofThreads=omp_get_num_threads();
                numofLines=n/numofThreads;


        cblas_dgemv(CblasRowMajor,CblasNoTrans,n,n,1.0,sigma,numofLines,u,1,0,v,1);
                #pragma omp for
                for (i = 0; i < n; i++) {
                        uplus[i] = (1-dt-c2*temp_vec[i])*u[i] + c1+c2*v[i];
```

}

Είναι αυτονόητο ότι όπως και όταν χρησιμοποιήσαμε την συνάρτηση BLAS στην σειριακή έκδοση του προγράμματος αυτή αντικαθιστά τον υπολογισμό που εκτελούνταν στον βρόγχο (j) γι αυτό και ο βρόγχος αυτός δεν υπάρχει πλέον στον κώδικα.

Πήραμε μετρήσεις χρησιμοποιώντας τις βελτιστοποιήσεις του μεταφραστή δηλαδή η μεταγλώττιση έγινε με την εντολή gcc –O3 – fopenmp -Wall –Wextra –lblas και οι χρόνοι που προκύπτουν είναι:

Με 2 νήματα:

Παράμετροι	**Ακολουθιακό πρόγραμμα που δίνεται**	**Παράλληλο Πρόγραμμα(βρόγχοι και BLAS) threads=2**	**Ποσοστιαία Χρονοβελτίωση**	**speedup**
--n 1000 --r 350	371.124782	353.716805	4%	1.05
--n 2000 --r 700	1476.733133	1436.056170	0.8%	1.02
--n 3000 --r 1000	3264.412432	3156.673992	3%	1.03
--n 4000 --r 1300	5910.104436	5731.293102	3%	1.03
--n 5000 –r 1600	8794.441821	8625.417176	1%	1.01

Με 4 νήματα:

Παράμετροι	**Ακολουθιακό πρόγραμμα που δίνεται**	**Παράλληλο Πρόγραμμα(βρόγχοι και BLAS) threads=4**	**Ποσοστιαία Χρονοβελτίωση**	**speedup**
--n 1000 --r 350	371.124782	394.492944	-	-
--n 2000 --r 700	1476.733133	1445.675525	2%	1.02
--n 3000 --r 1000	3264.412432	3160.636041	3%	1.03
--n 4000 --r 1300	5910.104436	5651.404194	3%	1.04
--n 5000 –r 1600	8794.441821	8657.177490	1%	1.01

Εδώ ενώ αναμέναμε να έχουμε μεγάλη βελτίωση από την παραλληλοποίηση της συνάρτησης BLAS, παρατηρούμε ότι το παράλληλο πρόγραμμα μας είναι τώρα πολύ πιο αργό σε σχέση με πριν. Η χρονοβελτίωση τώρα είναι αμελητέα της τάξης του 1.02% κατά μέσο όρο(Ειδικά για 1000 νευρώνες βλέπουμε ότι έχουμε ακόμη χειρότερη χρονική συμπεριφορά από την ακολουθιακή έκδοση του προγράμματος) ενώ βλέπουμε ότι είτε με 2 είτε με 4 threads η χρονική συμπεριφορά του προγράμματος είναι λιγο πολύ ίδια. Η εκτίμηση μας είναι ότι αυτό οφείλεται στο συνολικό overhead για την διάσπαση του μητρώου και τον διαμοιρασμό εργασίας στο κάθε νήμα αλλά ίσως ενά μέρος αυτής της κακής χρονικής συμπεριφοράς να οφείλεται και στην ίδια την συνάρτηση cblas_dgemv για την οποία δεν ξέρουμε πως υλοποιείται(ίσως οι βελτιστοποιήσεις της BLAS να έρχονται σε σύγκρουση με τις δικές μας).

Συμπεραίνουμε λοιπόν πως η παραλληλοποιήση οποιουδήποτε τμήματος κώδικα χωρίς έλεγχο δεν επιφέρει αυτόματα και ταχύτερη εκτέλεση του προγράμματος.