0.1 Πρόσθεση διανυσμάτων αριθμών μικρής ακρίβειας - *SAXPY*

Μια από τις λειτουργίες που κατέχει θεμελιώδη θέση σε εφαρμογές της γραμμικής άλγεβρας, αποτελεί η πρόσθεση διανυσμάτων δεκαδικών αριθμών μικρής ακρίβειας (*floats*), γνωστή ως **SAXPY**.

Στο παράδειγμα *SAXPY*, ο λόγος του μεγέθους υπολογισμών προς το μέγεθος των δεδομένων που τελούν υπό επεξεργασία είναι μικρός. Ως εκτότου, αποτελεί πρόβλημα περιορισμένης επεκτασιμότητας. Παρόλα αυτά, πρόκειται για ένα χρήσιμο παράδειγμα που ανήκει στην κατηγορία προβλημάτων παραλληλοποίησης τύπου *map* και των εννοιών *uniform* και *varying parameters*[1].

0.1.1 Περιγραφή προβλήματος

Η λειτουργία SAXPY δέχεται ως δεδομένα δυο διάνυσματα δεκαδικών αριθμών. Το πρώτο διάνυσμα πολλαπλασιάζεται με μια σταθερά *a* και το αποτέλεσμα προστίθεται στο δεύτερο διάνυσμα *y*. Τα διανύσματα *x,y* πρέπει να έχουν το ίδιο μέγεθος.

Ο υπολογισμός αυτός εμφανίζεται συχνά στη γραμμική άλγεβρα, όπως για παράδειγμα στη διαγραφή σειρών για την απαλοιφή **Gauss**. Το όνομα *SAXPY* δόθηκε από την βιβλιοθήκη **BLAS** (*"Basic Linear Algrbra Subprograms"*) για δεκαδικούς αρθμούς μικρής ακρίβειας (*floats*). Ο αντίστοιχος αλγόριθμος διπλής ακρίβειας ονομάζεται *DAXPY*, ενώ για μιγαδικούς αριθμούς ονομάζεται *CAXPY*. Η μαθηματική διατύπωση του *SAXPY* είναι:

$$\mathbf{y} = a * \mathbf{x} + \mathbf{y}$$

όπου το διάνυσμα *x* χρησιμοποιείται ως είσοδος, το *y* ως είσοδος και έξοδος. Δηλαδή το αρχικό διάνυσμα *y* τροποποιείται. Εναλλακτικά, η λειτουργία *SAXPY* μπορεί να περιγραφεί ως συνάρτηση που δρα σε μεμονωμένα στοιχεία, οπως φαίνεται παρακάτω:

$$f(t, p, q) = tp + q$$

$$\forall_i : y_i \leftarrow f(a, x_i, y_i)$$

Οι συναρτήσεις τύπου f δεχονται ως ορίσματα, δύο είδη παραμέτρων. Τις παραμέτρους όπως την a που παραμένουν σταθερές και ονομάζονται *uniform*, οι παράμετροι που είναι μεταβλητές σε κάθε κλίση της f ονομάζονται *varying*. Το μοτίβο *map* καλεί τη συνάρτηση f τόσες φορές όσες και ο αριθμός των στοιχείων του διανύσματος.[1].

0.2 Περιγραφή κεντρικού τμήματος προβλήματος SAXPY

Το πρόβλημα ξεκινάει δημιουργώντας ένα στοιχείο τύπου *Containers*, που περιέχει τα διανύσματα που εισάγονται στον αλγόριθμο SAXPY. Ο ρόλος του Containers είναι για την διαχείριση της *heap* μνήμης. Τα διανύσματα και η σταθερά cons αρχικοποιούνται με τυχαίους αριθμούς μικρής ακρίβειας. Το μέγεθος των διανυσμάτων (μέγεθος προβλήματος) ορίζεται από τον χρήστη μέσω της γραμμής εντολών. Στη συνέχεια, καλείται ο αλγόριθμος SAXPY μόλις τελειώσει γίνεται επαλήθευση των αποτελεσμάτων, όπου αν επαληθευτούν σωστά, γίνεται εκτύπωση του χρόνουν εκτέλεσης της παραλλαγής.

Συμβ. 1: Κεντρικός κώδικας προβλήματος SAXPY

```
int main(int argc, char **argv) {
    Opts o;
    parseArgs(argc, argv, o);
    Containers c(o.size);
    c.setRandomValues();
    float cons = float(rand()) / float(RAND_MAX);
    auto start = omp_get_wtime();
    saxpy(c.m_size, cons, c.m_a, c.m_b);
    auto end = omp_get_wtime();
    verify(c.m_size, cons, c.m_a, c.m_b, c.m_verification);
    std::cout << "Execution Time : " << std::fixed
         << end - start << std::setprecision(5);
    std::cout << " sec " << std::endl;
    return 0;
}
```

Συμβ. 2: Κλάση *Containers*

```
struct Containers {
    explicit Containers(size_t containers_size);
    ~Containers();

    size_t m_size;
    float *m_a;
    float *m_verification;
    float *m_b;
};

Containers::Containers(size_t containers_size)
    : m_size(containers_size) {
    srand(time(nullptr));
    m_a = new float[containers_size];
    m_verification = new float[containers_size];
    m_b = new float[containers_size];
}

Containers::~Containers() {
    delete []m_a;
    delete []m_b;
    delete [] m_verification;
}

Containers::setRandomValues() {
    fill_random_arr(m_a, m_size);
    fill_random_arr(m_b, m_size);
}
```

Συμβ. 3: Συνάρτηση επαλήθευσης

```
static void verify(size_t size, float c, float *a, float *b,
                   float *verification) {
    for (size_t i = 0; i < size; ++i) {
        if (abs(c * a[i] + verification[i] - b[i]) >= 10e-6) {
            std::cout << "Failed index: " << i <<
             ". " << c * a[i] + verification[i] <<
             " =! " << b[i] << std::endl;
            exit(1);
        }
    }
}
```

Συμβ. 4: Συνάρτηση αρχικοποίησης τιμών

```
static void fill_random_arr(float *arr, size_t size) {
    for (size_t k = 0; k < size; ++k) {
        arr[k] = (float)(rand()) / RAND_MAX;
    }
}
```

0.3 Σειριακή εκτέλεση

Η υλοποίηση της σειριακής παραλλαγής της συνάρτησης saxpy περιλαμβάνει έναν επαναληπτικό βρόγχο στον οποίο γίνεται ο υπολογισμός για κάθε στοιχείο των διανυσμάτων.

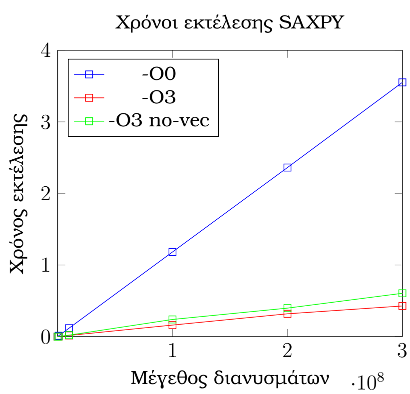
Συμβ. 5: Σειριακή υλοποίηση της SAXPY

```
void saxpy(size_t n, float a, const float *x, float *y) {
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Οι χρόνοι εκτέλεσης του αλγορίθμοι συναρτήσει του μεγέθους του προβλήματος παρατίθενται στον παρακάτων πίνακα. Το πρόγραμμα μεταγλωττίστηκε με επιλογή -O3 και -O0.

Πίνακας 1: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	-O0	**-O3**	**-O3 -fno-tree-vectorize**
100000	0.0012	0.00011	0.0002
1000000	0.0118	0.00171	0.0021
10000000	0.1179	0.01631	0.0203
100000000	1.1821	0.16124	0.2397
200000000	2.3612	0.31867	0.3981
300000000	3.5510	0.42806	0.6052
400000000	4.7291	0.6339	0.7884



Σχήμα 1: Σύγκριση αποτελεσμάτων

Μέγεθος	Επιτάχυνση (%)
100000	90.8
1000000	85.5
10000000	86.2
100000000	86.4
200000000	86.5
300000000	87.9

Πίνακας 2: Ποσοστό μείωσης με -O3

0.4 Παραλλαγή με οδηγία parallel for

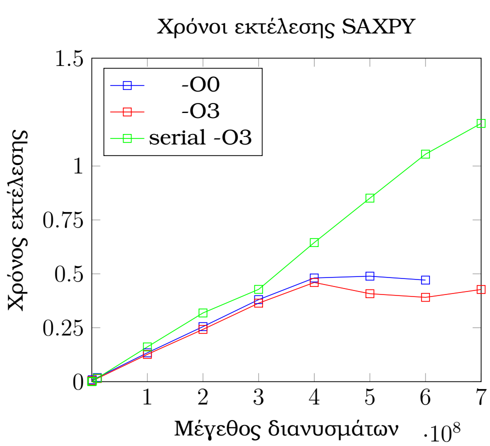
Η πρώτη υλοποίηση παραλλαγής με παραλληλισμό της συνάρτησης saxpy περιλαμβάνει τον επαναληπτικό βρόγχο ενσωματωμένο στην οδηγία παραλλελ φορ στον οποίο γίνεται ο υπολογισμός για κάθε στοιχείο των διανυσμάτων. Τα αποτελέσματα φαίνονται στον ακολουθούμενο πίνακα.

Συμβ. 6: Υλοποίηση παραλλαγής με parallel for

```
void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 3: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.003	0.004
1000000	0.009	0.004
10000000	0.018	0.014
100000000	0.133	0.126
200000000	0.255	0.244
300000000	0.380	0.364
400000000	0.480	0.460
500000000	0.489	0.408



Σχήμα 2: Σύγκριση αποτελεσμάτων

Μέγεθος	Επιτάχυνση (%)
100000	-33.3
1000000	55.5
10000000	22.22
100000000	5.26
200000000	4.31
300000000	4.2
400000000	4.16
500000000	16.56

Πίνακας 4: Ποσοστό μείωσης με -O3

0.4.1 Παρατηρήσεις

Με τη χρήση της οδηγίας *pragma omp parallel for* επετέφχθει μείωση του χρόνου εκτέλεσης του αλγορίθμου όταν το πρόγραμμα μεταγλωττίζεται με -O0 σε σύγκριση με την αντίστοιχη σειριακή. Οι χρόνοι εκτέλεσης της συγκεκριμένης περίπτωσης ωστόσο, μοιάζουν με τους αντίστοιχους της μεταγλώττισης σειριακού κώδικα με -O3 για διανύσματα μέχρι 3e8 στοιχείων, ενώ για μεγαλύτερες τιμές η παράλληλη εκτέλεση έχει καλύτερες επιδόσεις. Τέλος, από το παραπάνω διάγραμμα δε προκύπτει κάποια διαφοροποίηση ανάμεσα στη μεταγλώττιση με -O0 και -O3.

0.5 Παραλλαγή με *parallel for* και padding

Σε αυτή την περίπτωση, ο αλγόριθμος παραμένει ίδιος, χρησιμοποιείται δηλαδή η οδηγία *pragma omp parallel for*. Ωστόσο στη συνάρτηση εισάγονται ως ορίσματα δομές που εμπεριέχουν μία μεταβλητή αριθμού μικρής ακρίβειας και ένα τεχνητό κενό *padding*. Το μέγεθος της είναι 64bytes και έχει ως στόχο την αποφυγή του φαινομένου **false sharing**.

Συμβ. 7: Υλοποίηση παραλλαγής με parallel for

```
void saxpy(size_t n, float a, const float64 *x, float64 *y) {
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i) {
        y[i].val = a * x[i].val + y[i].val;
    }
}
```

Πίνακας 5: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.004	0.0046
1000000	0.021	0.0196
10000000	0.182	0.185
100000000	killed	killed

0.5.1 Παρατηρήσεις

Το πρόβλημα *false sharing* δεν ήταν δυνατό να εντοπισθεί στη συγκεκριμένη παραλλαγή. Μάλιστα, ο έλεγχος για μεγέθη διανυσμάτων μεγαλύτερων των 1e8 στοιχείων, ήταν ανεπιτυχής λόγω έλλειψης υπολογιστικών πόρων.

0.6 Παραλλαγή με *omp simd*

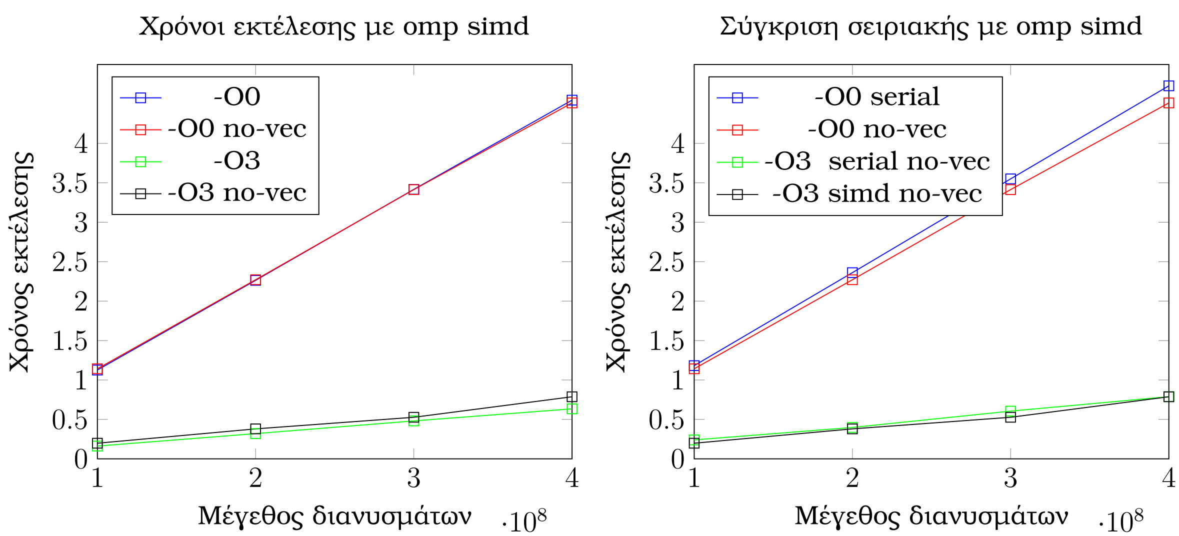
Συμβ. 8: Υλοποίηση παραλλαγής με omp simd

```
void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 6: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)			
	-O0	-O0 -fno-tree-vectorize	-O3	-O3 -fno-tree-vectorize
100000	0.001	0.001	0.000	0.003
1000000	0.011	0.011	0.002	0.002
10000000	0.113	0.114	0.016	0.020
100000000	1.126	1.142	0.162	0.198
200000000	2.263	2.271	0.320	0.380
300000000	3.417	3.414	0.481	0.527
400000000	4.548	4.513	0.634	0.787

ϑελω να συγκρινω το εςτοριζατιν με την σεριαλ εκδοση. Να ξαναδω τη σειριαλ, για να παρε αυτο με την επιλογη -φνο-τρεε-εςτοριζε. Να φτιαξω διαγραμματα και να κανω πειραματα.



Σχήμα 3: Λεφτ: Νο Ιντεραςτιον. Ριγητ: Ιντεραςτιον

0.7 Παραλλαγή με *omp declare simd uniform*

Συμβ. 9: Υλοποίηση παραλλαγής με omp declare simd uniform

```
#pragma omp declare simd uniform(a)
float do_work(float a, float b, float c)
{
    return a * b + c;
}

void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = do_work(a, x[i], y[i]);
    }
}
```

Πίνακας 7: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**			
	-O0	**-O0 -fno-tree-vectorize**	**-O3**	**-O3 -fno-tree-vectorize**
100000	0.001	0.001	0.001	0.0003
1000000	0.017	0.014	0.002	0.002
10000000	0.143	0.139	0.017	0.022
100000000	1.424	1.421	0.148	0.193
200000000	2.792	2.78	0.294	0.422
300000000	4.206	4.174	0.459	0.630
400000000	5.705	5.59	0.651	0.833

TODO Μπορω να βαλω 4 διαγραμματα που να συγκρινουν το προηγουμενο με αυτο. για να δειξω οτι η πιθανη καθυστερηση σε αυτο ειναι το οερηεαδ απο το φυνςτιον ςαλλ.

0.8 Παραλλαγή με *omp declare simd uniform notinbranch*

Συμβ. 10: Υλοποίηση παραλλαγής με omp declare simd uniform notinbranch

```
#pragma omp declare simd uniform(a) notinbranch
float do_work(float a, float b, float c)
{
    return a * b + c;
}

void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i]; //TODO
    }
}
```

Πίνακας 8: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**			
	-O0	**-O0 -fno-tree-vectorize**	**-O3**	**-O3 -fno-tree-vectorize**
100000	0.001	0.001	0.001	0.001
1000000	0.012	0.011	0.002	0.002
10000000	0.115	0.115	0.016	0.021
100000000	1.154	1.145	0.161	0.212
200000000	2.308	2.299	0.322	0.378
300000000	3.473	3.446	0.479	0.633
400000000	4.621	4.604	0.632	0.827

Να αναφερω οτι δε βλέπω κάποια διαφορα σε σχέση με το προηγούμενο παραλλαγή.

0.9 Παραλλαγή με *omp parallel for simd*

Συμβ. 11: Υλοποίηση παραλλαγής με omp parallel for simd

```
void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp parallel for simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 9: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)			
	-O0	-O0 -fno-tree-vectorize	-O3	-O3 -fno-tree-vectorize
100000	0.004	0.005	0.002	0.005
1000000	0.007	0.007	0.001	0.004
10000000	0.019	0.019	0.017	0.014
100000000	0.127	0.125	0.125	0.124
200000000	0.256	0.249	0.247	0.250
300000000	0.372	0.383	0.369	0.365
400000000	0.494	0.525	0.482	0.485

να αναφέρω οτι δεν παιζει ρολο το -O3 οπτιμιζατιον :(Επίσης να συγκρίνω με την παραλλελ φορ χωρις σιμδ!

0.10 Παραλλαγή με *target map*

Συμβ. 12: Υλοποίηση παραλλαγής με target map

```
void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp target map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 10: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.005	0.004
1000000	0.016	0.006
10000000	0.123	0.019
100000000	1.187	0.269
200000000	2.381	0.334
300000000	3.555	0.512
400000000	4.731	0.622

Συγκριση με σειριακή.

0.11 Παραλλαγή με *target simd map*

Συμβ. 13: Υλοποίηση παραλλαγής με target simd map

```
void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp target simd map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 11: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**			
	-O0	**-O0 -fopenmp-simd**	**-O3**	**-O3 -fopenmp-simd**
100000	0.005	0.005	0.004	0.004
1000000	0.015	0.016	0.006	0.006
10000000	0.120	0.120	0.020	0.021
100000000	1.154	1.161	0.157	0.167
200000000	2.298	2.314	0.332	0.333
300000000	3.485	3.491	0.486	0.490
400000000	4.620	4.613	0.650	0.654

Συγκριση με σιμδ.

0.12 Παραλλαγή με *target parallel for*

Συμβ. 14: Υλοποίηση παραλλαγής με target parallel for

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target parallel for map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 12: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.011	0.011
1000000	0.009	0.013
10000000	0.035	0.021
100000000	0.150	0.127
200000000	0.264	0.251
300000000	0.387	0.369
400000000	0.500	0.490

0.13 Παραλλαγή με *target parallel for simd*

Συμβ. 15: Υλοποίηση παραλλαγής με target parallel for simd

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target parallel for simd map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 13: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**			
	-O0	**-O0 -fopenmp-simd**	**-O3**	**-O3 -fopenmp-simd**
100000	0.010	0.010	0.011	0.011
1000000	0.012	0.014	0.011	0.009
10000000	0.027	0.026	0.020	0.020
100000000	0.140	0.154	0.128	0.130
200000000	0.257	0.271	0.249	0.247
300000000	0.378	0.385	0.370	0.365
400000000	0.513	0.505	0.450	0.489

0.14 Παραλλαγή με *target teams map*

Συμβ. 16: Υλοποίηση παραλλαγής με target teams map

```
void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp target teams map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 14: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.005	0.004
1000000	0.016	0.006
10000000	0.132	0.025
100000000	1.186	0.221
200000000	2.374	0.420
300000000	3.543	0.652
400000000	4.729	0.821

0.15 Παραλλαγή με *target teams distribute map*

Συμβ. 17: Υλοποίηση παραλλαγής με target teams distribute map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute map(from: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 15: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.005	0.004
1000000	0.015	0.006
10000000	0.117	0.025
100000000	1.142	0.214
200000000	2.278	0.428
300000000	3.433	0.625
400000000	4.575	0.851

0.16 Παραλλαγή με *target teams distribute parallel for map*

Συμβ. 18: Υλοποίηση παραλλαγής με target teams distribute parallel for

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute parallel for map(from: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 16: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.010	0.011
1000000	0.016	0.011
10000000	0.023	0.020
100000000	0.139	0.127
200000000	0.257	0.250
300000000	0.389	0.369
400000000	0.511	0.490

0.17 Παραλλαγή με *target teams distribute simd map*

Συμβ. 19: Υλοποίηση παραλλαγής με target teams distribute simd map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute simd map(from: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 17: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**			
	-O0	**-O0 -fopenmp-simd**	**-O3**	**-O3 -fopenmp-simd**
100000	0.005	0.050	0.004	0.004
1000000	0.015	0.015	0.006	0.006
10000000	0.120	0.119	0.021	0.021
100000000	1.159	1.153	0.159	0.168
200000000	2.311	2.305	0.326	0.327
300000000	3.487	3.472	0.482	0.495
400000000	4.620	4.610	0.647	0.644

0.18 Παραλλαγή με *target teams distribute parallel for simd map*

Συμβ. 20: Υλοποίηση παραλλαγής με teams distribute parallel for simd map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute parallel for simd\
                        map(from: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 18: Καταγραφή χρόνων εκτέλεσης

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	-O0	**-O3**
100000	0.010	0.010
1000000	0.012	0.010
10000000	0.025	0.021
100000000	0.146	0.127
200000000	0.263	0.246
300000000	0.389	0.371
400000000	0.519	0.489

References

[1] J. R. Michael McCool, Arch D.Robison. *Structural Parallel Programming*, pages 124–125. Morgan Kaufmann, 2012.