Πανεπιστήμιο Μακεδονίας
Τμήμα Εφαρμοσμένης Πληροφορικής

Παράλληλος Προγραμματισμός με χρήση OpenMP

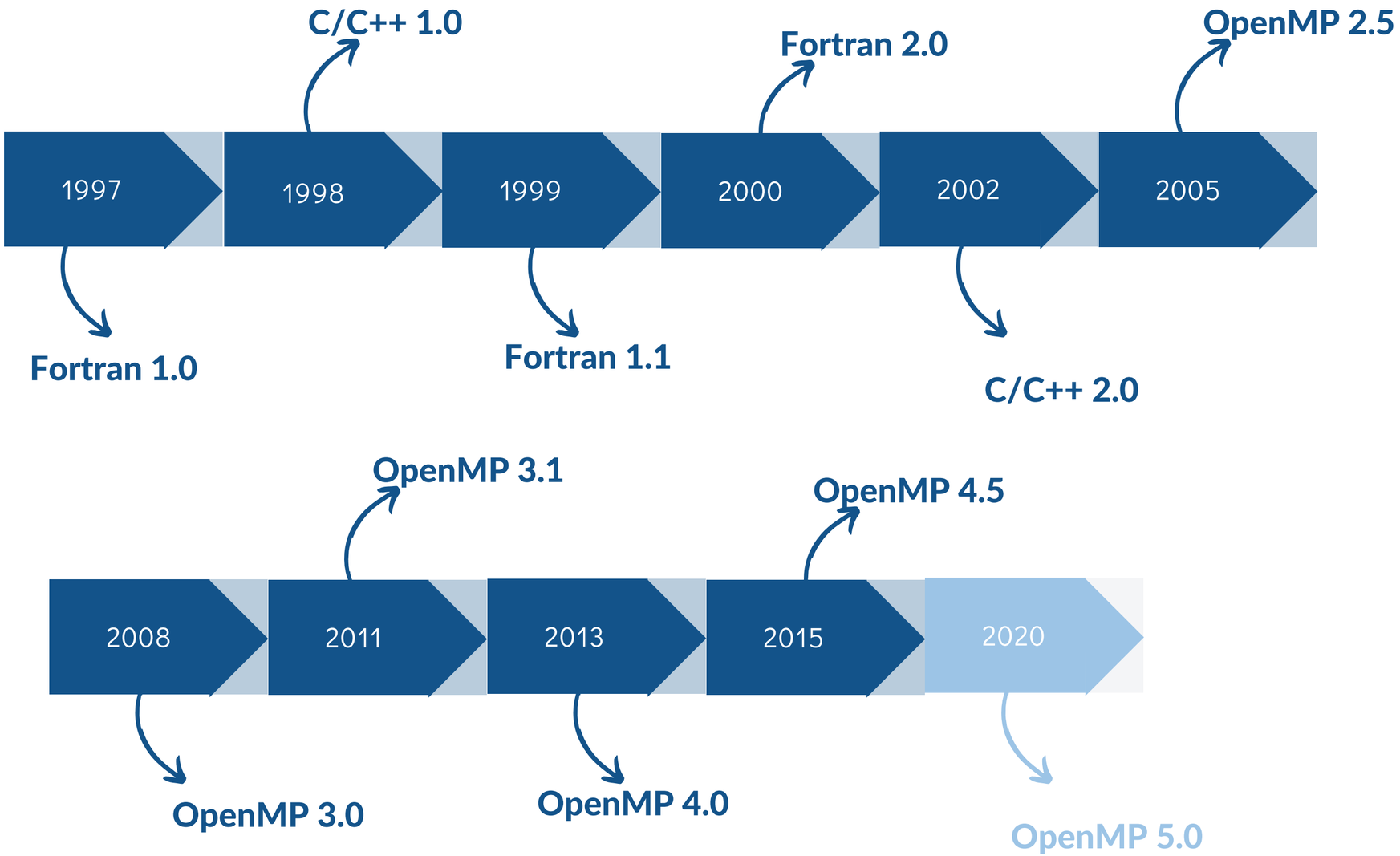
Κοντογιάννης Γεώργιος

Διάρθωση Εργασίας

Συνοπτική περιγραφή θεωρίας

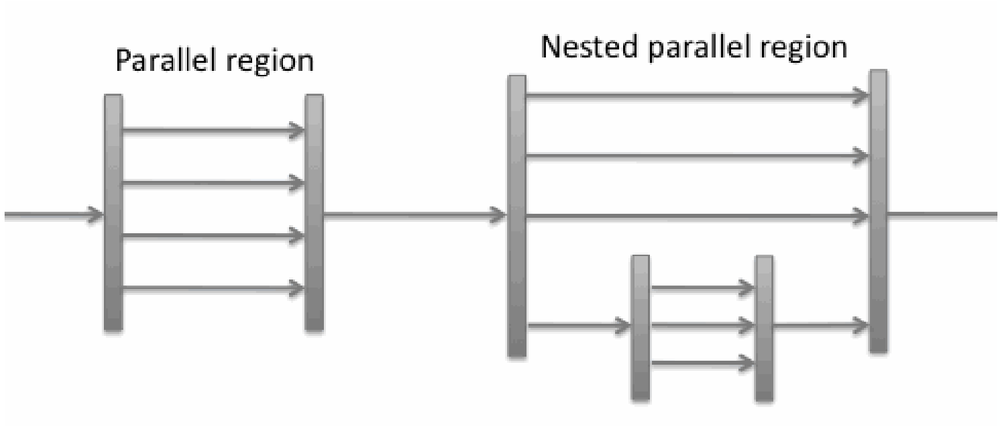
Σύνταξη και μελέτη παραδειγμάτων

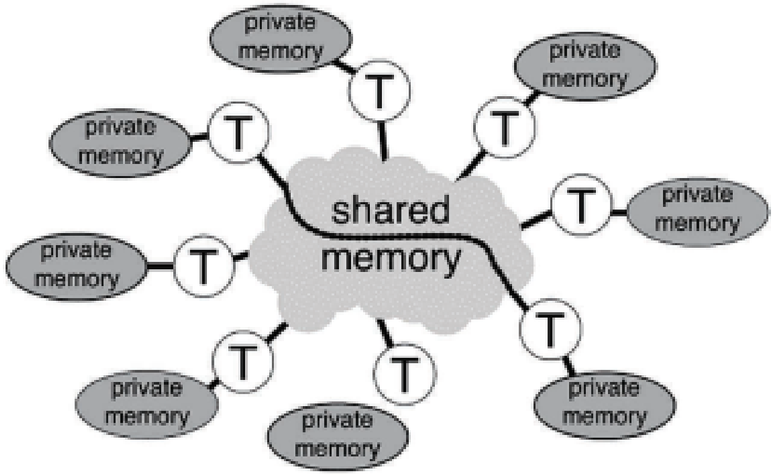
Ιστορική Αναδρομή



Διάρθωση Εργασίας

#pragma omp (directive) [clause[[,] clause] ...] new-line





!

Race Conditions

False sharing

Αποδόμηση βιβλιοθήκης OpenMP

- Μεταβλητές περιβάλλοντος

  OMP_NUM_THREADS, OMP_THREAD_LIMIT, OMP_CANCELLATION, OMP_WAIT_POLICY

- Runtime functions

  omp_set_num_threads, omp_get_max_threads, omp_set_dynamic, omp_init_nest_lock

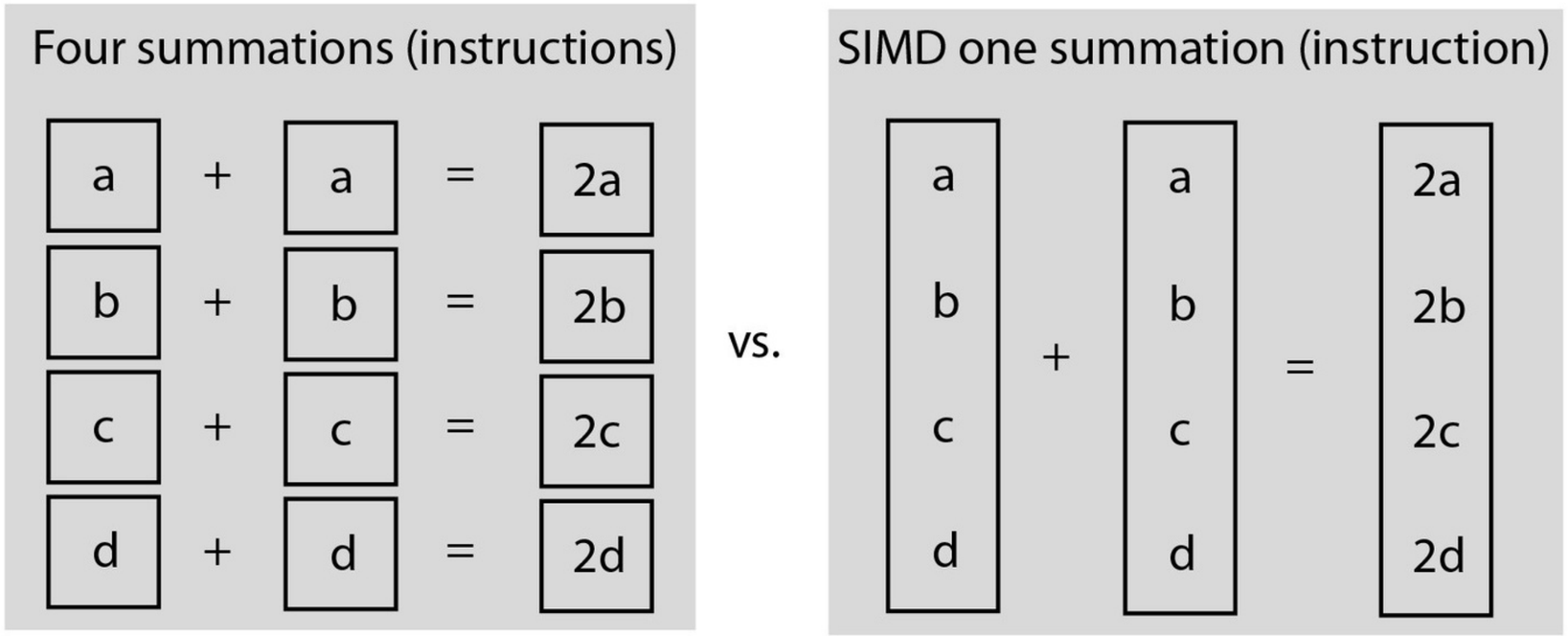
- Οδηγίες και φράσεις

  #pragma omp parallel for schedule(static), #pragma omp single , #pragma omp sections

Χαρακτηριστικά νεότερων εκδόσεων

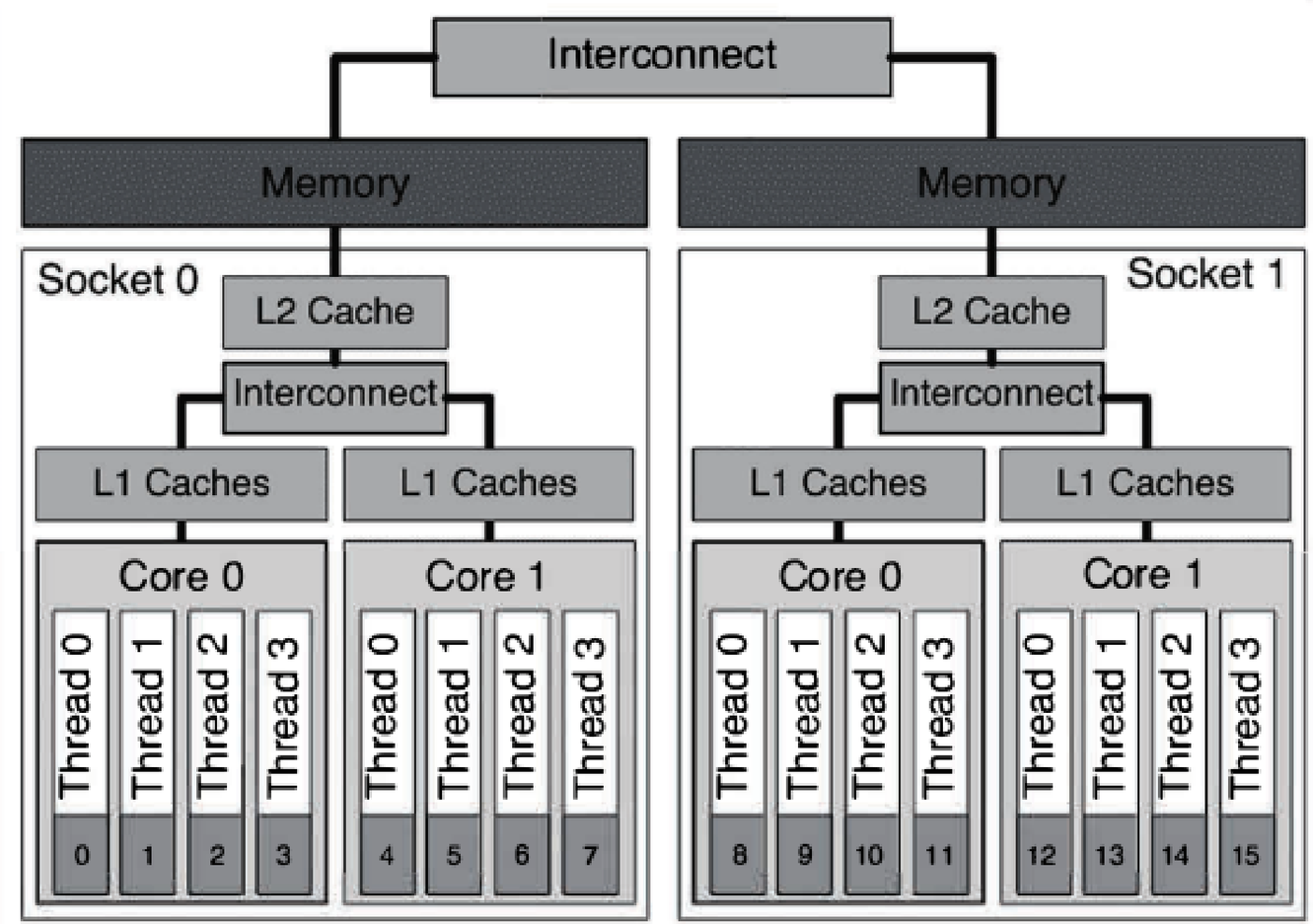
- Threads affinity
- Ετερογενείς προγραμματισμός
- Διανυσματικοποίηση SIMD
- Tasking
- Διαχείριση σφαλμάτων
- User Defined Reductions

Διανυσματικοποίηση SIMD

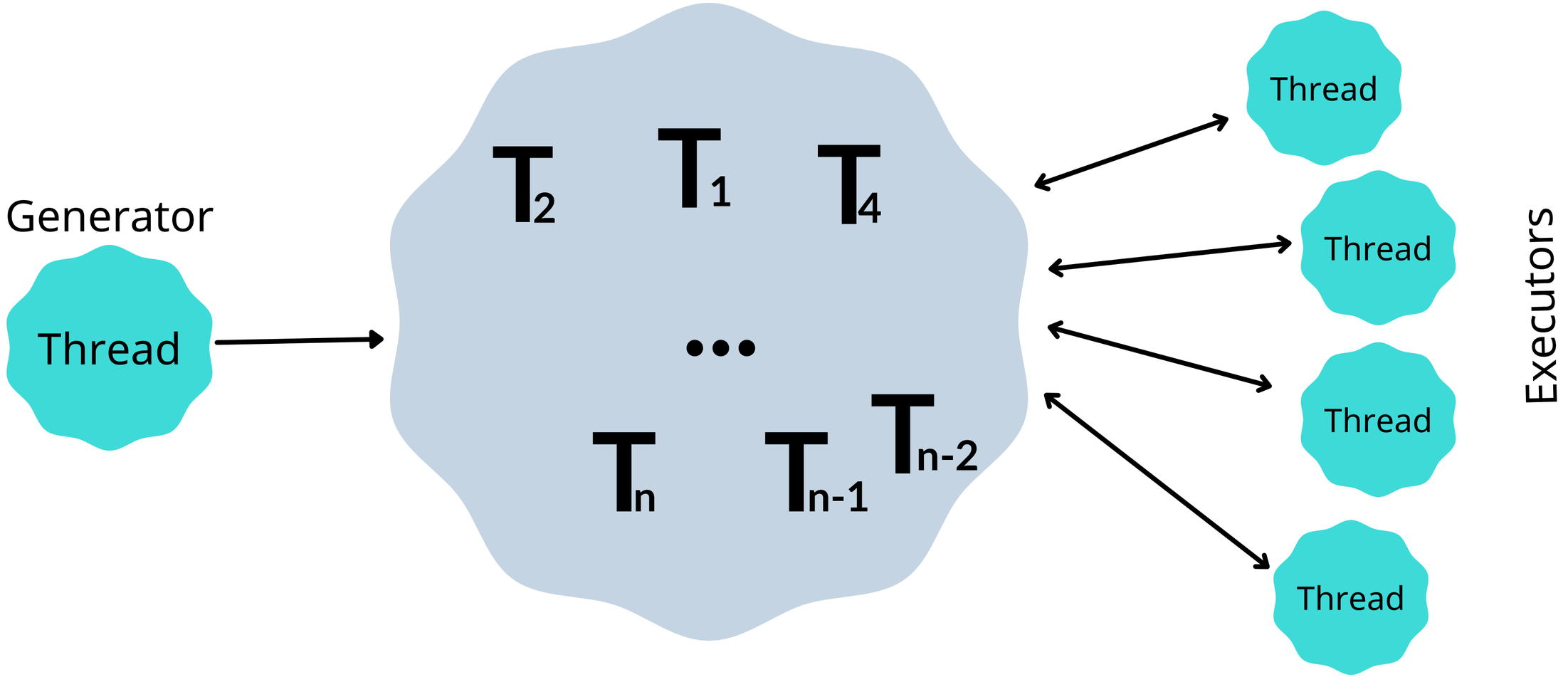


x4

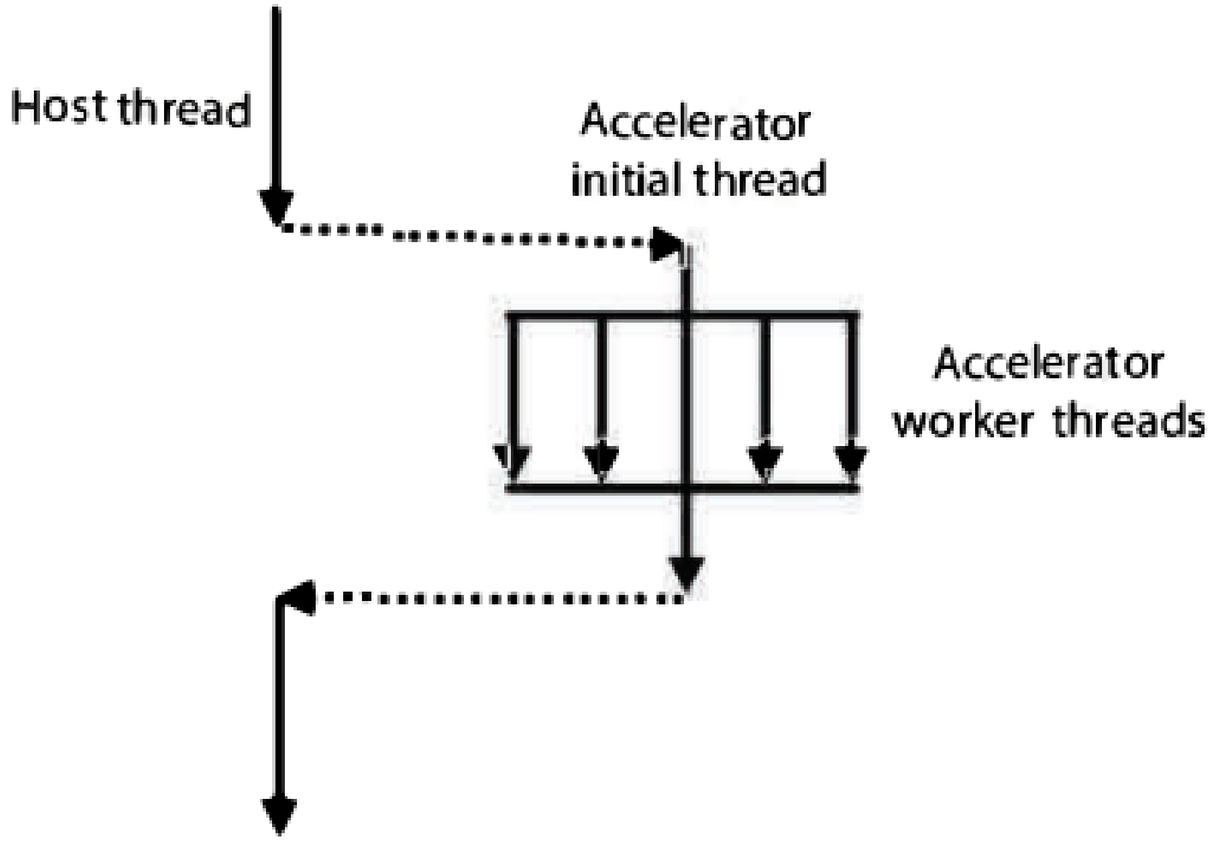
Thread Affinity



Tasking


Generator
Thread
T_2
T_1
T_4
...
T_n
T_{n-1}
T_{n-2}
Thread
Thread
Thread
Thread
Executors

Ετερογενείς Προγραμματισμός - Offloading



Υλοποιημένα παραδείγματα

- Discrete Fourier Transform
- Mergesort
- Quicksort
- Producer-Consumer
- π calculation
- Υπολογισμός πρώτων αριθμών
- Εσωτερικό γινόμενο
- Πολλαπλασιασμός πινάκων
- SAXPY
- Διάσχιση συνδεδεμένη λίστας

https://github.com/gkonto/openmp

SAXPY (1)

$$y = a * x + y$$

```
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];
    }
```

Serial

pragma omp simd
pragma omp declare simd uniform notinbranch

pragma omp target teams map(tofrom: y) map(to: x)

pragma omp parallel for

pragma omp target map(tofrom: y) map(to: x)

pragma omp target teams distribute map(tofrom: y) map(to: x)

pragma omp simd

pragma omp target parallel for map(tofrom: y) map(to: x)

pragma omp target teams distribute parallel for map(tofrom: y) map(to: x)

pragma omp parallel for simd

pragma omp target parallel for map(tofrom: y) map(to: x)

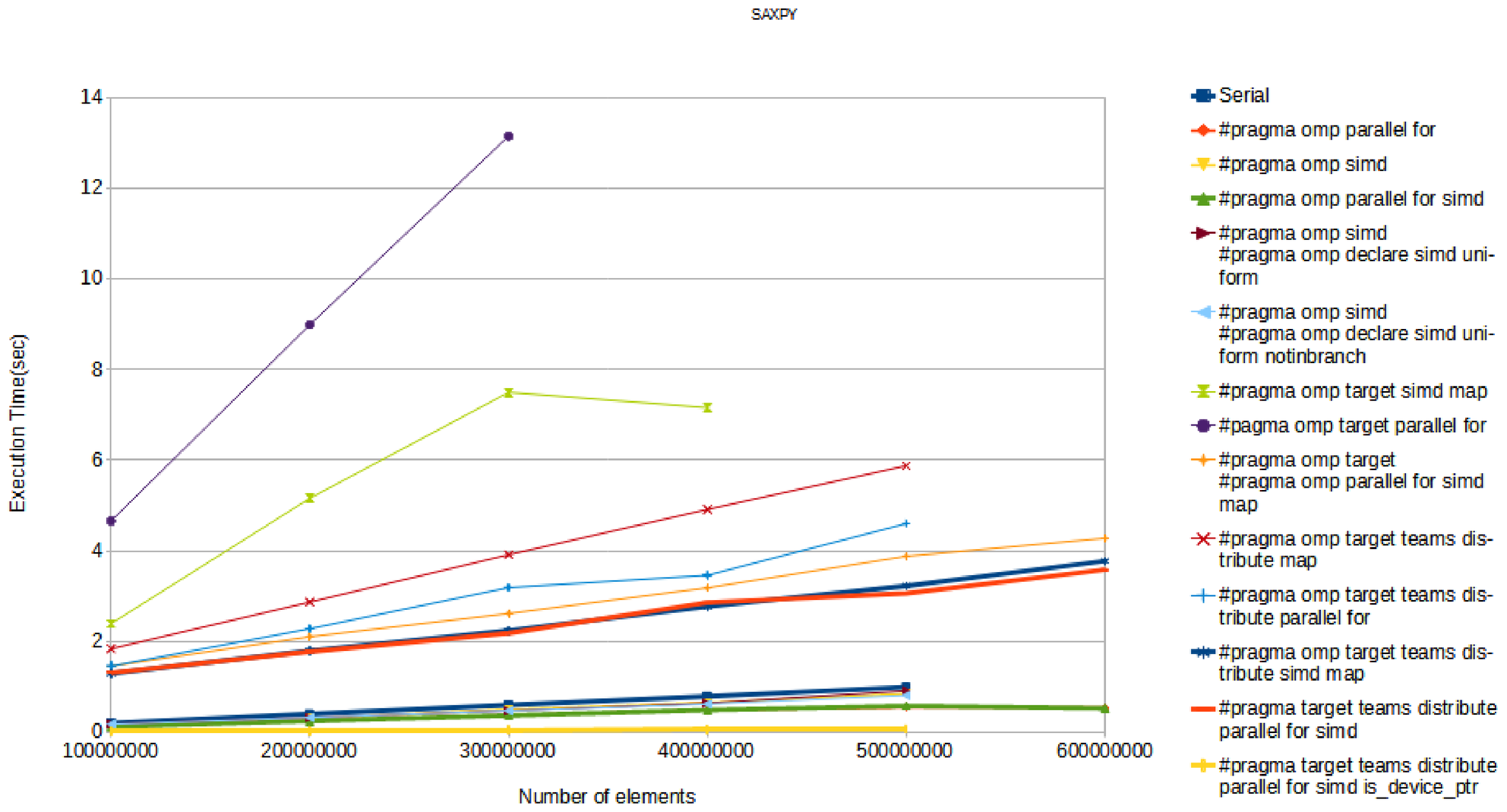
pragma omp target teams distribute parallel for simd map(tofrom: y) map(to: x)

pragma omp simd
pragma omp declare simd uniform

pragma omp target teams distribute parallel for simd is_device_ptr(y, x)

pragma omp target simd map(tofrom: y) map(to: x)

SAXPY (2)


SAXPY
Execution Time(sec)
0
2
4
6
8
10
12
14
100000000
200000000
300000000
400000000
500000000
600000000
Number of elements
Serial
#pragma omp parallel for
#pragma omp simd
#pragma omp parallel for simd
#pragma omp simd
#pragma omp declare simd uniform
#pragma omp simd
#pragma omp declare simd uniform notinbranch
#pragma omp target simd map
#pagma omp target parallel for
#pragma omp target
#pragma omp parallel for simd
map
#pragma omp target teams distribute map
#pragma omp target teams distribute parallel for
#pragma omp target teams distribute simd map
#pragma target teams distribute parallel for simd
#pragma target teams distribute parallel for simd is_device_ptr

SAXPY (3) - Παρατηρήσεις

- Η επιλογή μεταγλώττισης με -O2 δεν εφαρμόζει διανυσματικοποίηση
- Η χρήση #pragma omp simd εφαρμόζει διανυσματικοποίηση
- Η εκτέλεση του προγράμματος με χρήση offloading εμφανίζει χαμηλές επιδόσεις
- Απο το χρόνο εκτέλεσης με offloading, το 98% οφείλεται στην αντιστοίχιση των δεδομένων
- Η εκτέλεση με is_device_ptr εμφανίζει πολυ καλές επιδόσεις

Υπολογισμός Π (1)

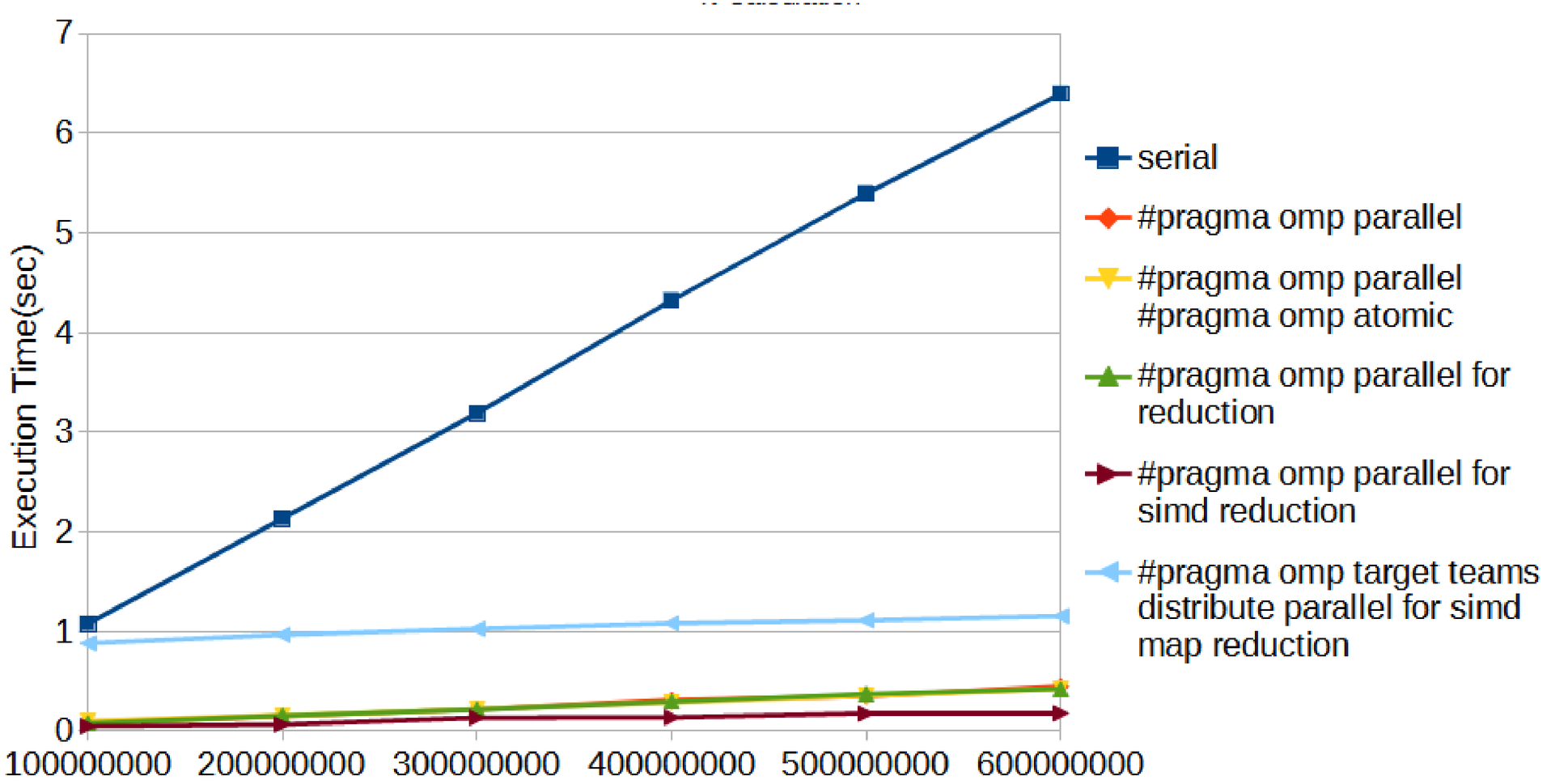
$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \sum_{n=1}^{\infty} \frac{4}{1+x^2}\Delta$$

$$\Delta = \frac{1}{N}$$

$$x_i = \frac{i-0.5}{\Delta}$$

```
double pi(long num_steps) {
    int upper_limit = 1;
    double step = upper_limit/(double)num_steps;
    double sum = .0, pi = 0.0;

    for (int i = 0; i < num_steps; ++i){
        double x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x*x);
    }
    pi = step * sum;
    return pi;
}
```

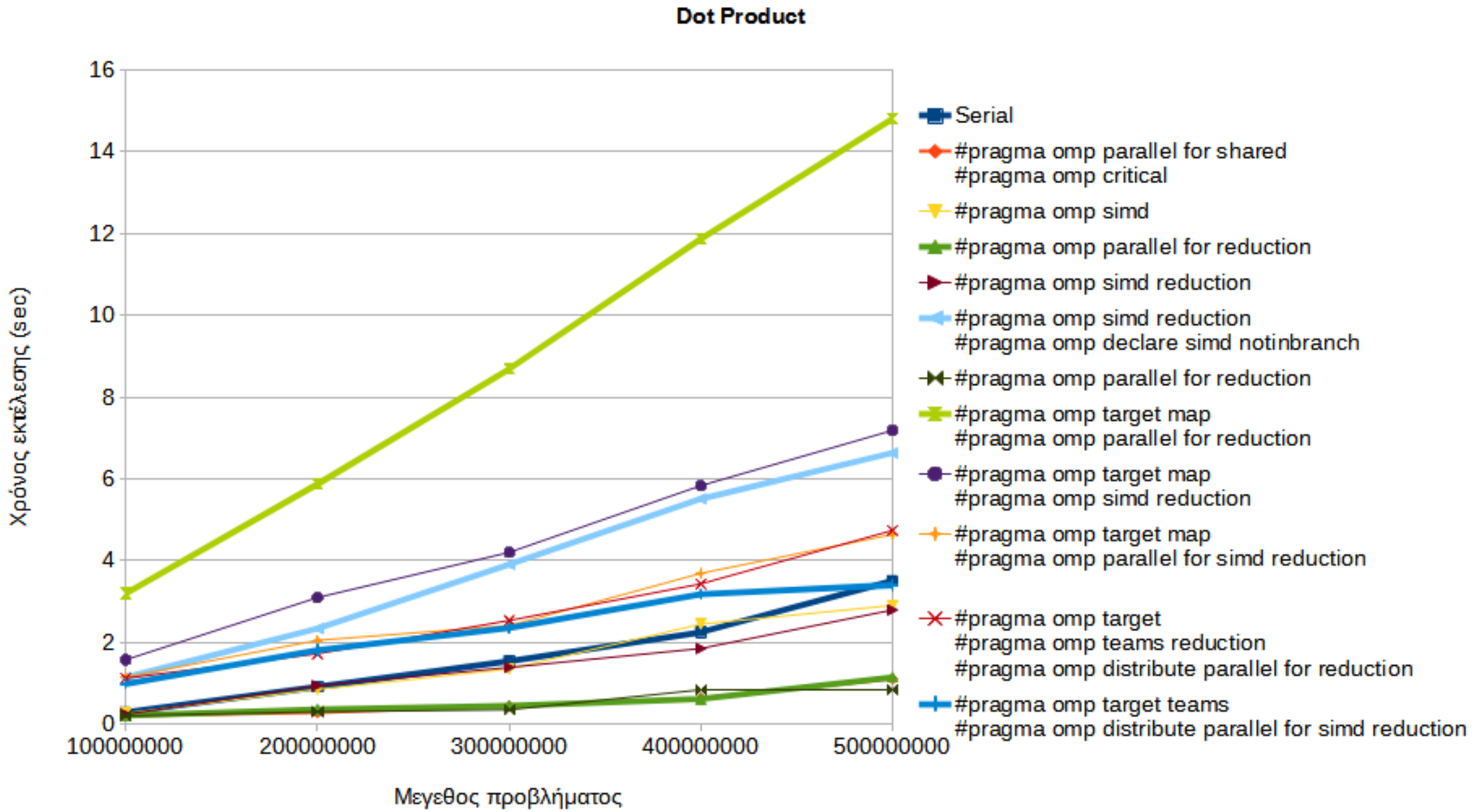
Υπολογισμός Π (2)



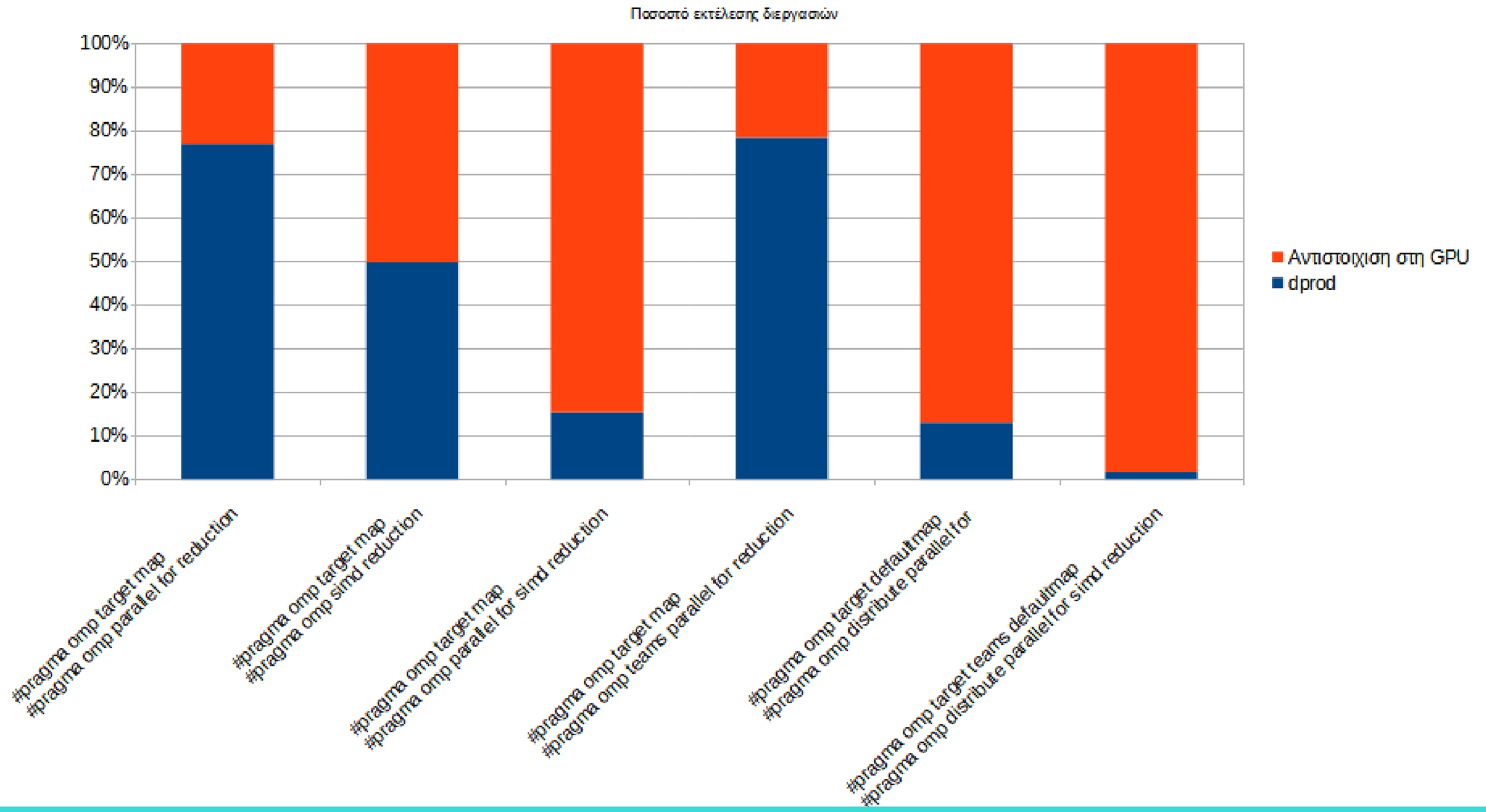
Εσωτερικό γινόμενο(1)

$$x = \sum a_i * b_i$$

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
    for (size_t i = 0; i < num; ++i) {
        res += a[i] * b[i];
    }
    return res;
}
```

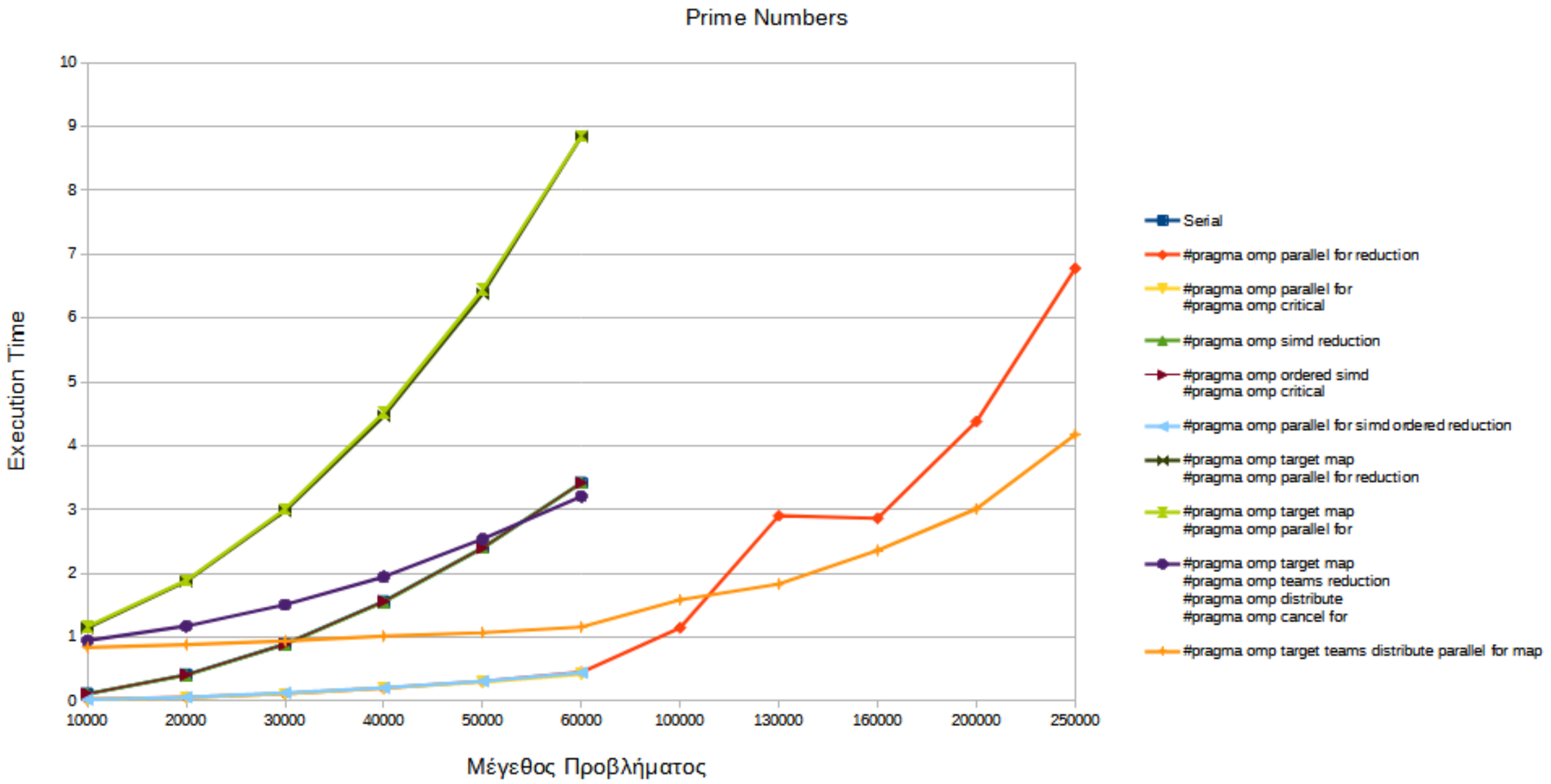


Εσωτερικό γινόμενο(3)



Υπολογισμός πρώτων αριθμών(1)

```
nt prime_number(int n) {
    int prime = 0;
    int total = 0;

    for (int i = 2; i <= n; i++) {
        prime = 1;
        for (int j = 2; j < i; j++) {
            if ((i % j) == 0) {
                prime = 0;
                break;
            }
        }
        total += prime;
    }
    return total;
```



Linked-List traversal(1)

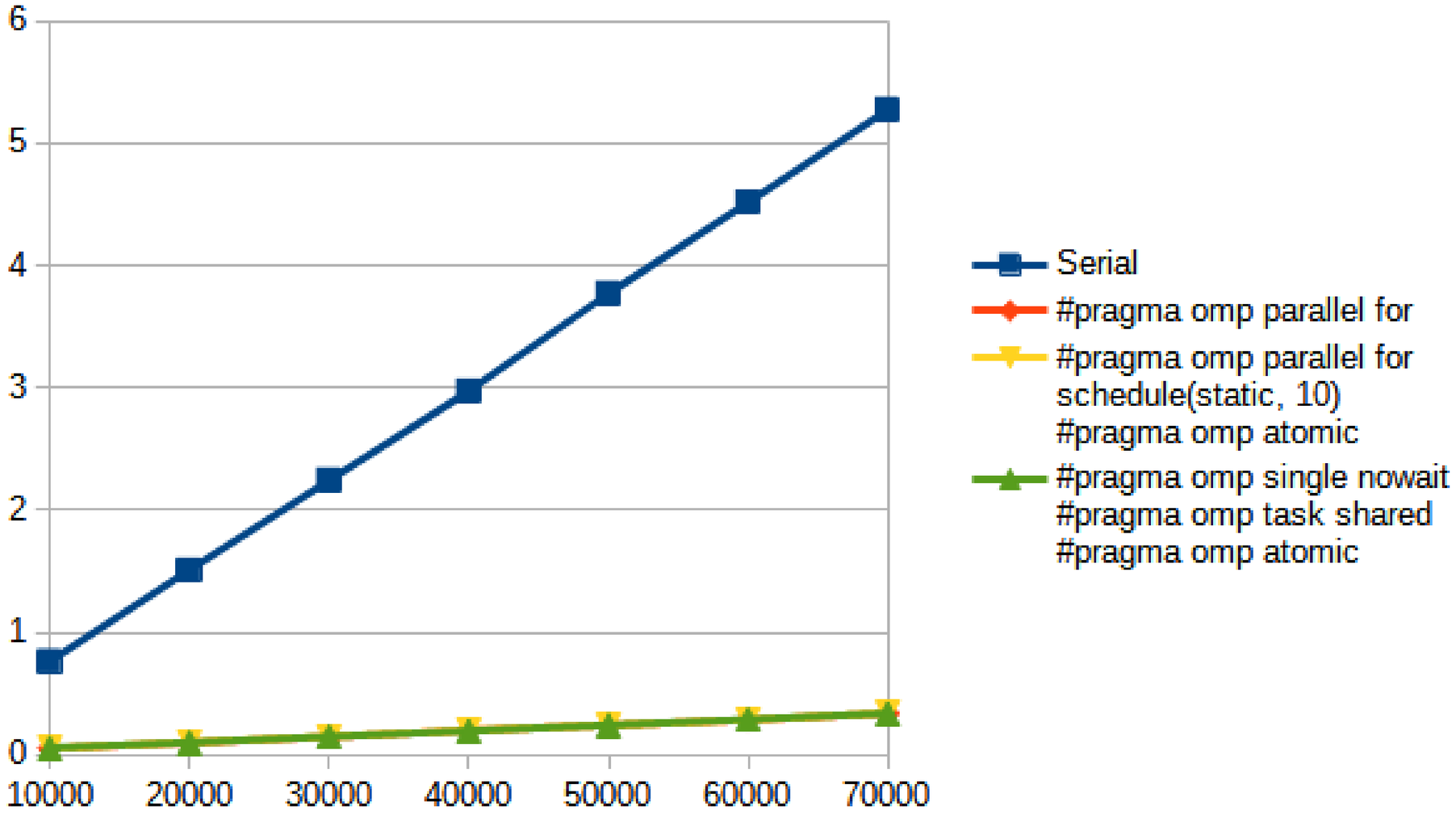
```
using namespace std;
void dowork(Lnode<int> &node, void *args) {
      int *num_nodes = static_cast<int *>(args);
      this_thread::sleep_for(
         chrono::nanoseconds(rand() % 5 + 1));
      *num_nodes += node.data();
 }

 int calculate(Llist<int> &l) {

      int number_of_nodes = 0;
      l.forEveryNode(dowork, &number_of_nodes);

      return number_of_nodes;
 }
```

```
using namespace std;

int calculate(Llist<int> &l) {

    Lnode<int> *node = l.head();
    size_t nodes_num = l.size();
    std::vector<int> nodes(nodes_num);

    for (size_t i = 0; i < nodes_num; ++i) {
        nodes[i] = node->data();
        node = node->next();
    }


    int num_nodes = 0;
#pragma omp parallel for shared(num_nodes)
    for (size_t i = 0; i < nodes_num; ++i) {
        this_thread::sleep_for(
                        chrono::milliseconds(randI()));
#pragma omp atomic
        num_nodes += nodes[i];
    }

    return num_nodes;
}
```

```
int calculate(Llist<int> &l) {
    Lnode<int> *node = l.head();
    size_t nodes_num = l.size();
    std::vector<int> nodes(nodes_num);

    for (size_t i = 0; i < nodes_num; ++i) {
        nodes[i] = node->data();
        node = node->next();
    }
    int num_nodes = 0;

#pragma omp parallel for schedule(static, 10) shared(num_nodes)
    for (size_t i = 0; i < nodes_num; ++i) {
        this_thread::sleep_for(chrono::nanoseconds(randI()));
#pragma omp atomic
        num_nodes += nodes[i];
    }

    return num_nodes;
}
```

```
using namespace std;

int calculate(Llist<int> &l) {
    Lnode<int> *node = l.head();

    int num_nodes = 0;
#pragma omp parallel shared(num_nodes)
    {
#pragma omp single nowait
        {
            while (node) {
#pragma omp task shared(node) mergeable
                {
                    this_thread::sleep_for(
                        chrono::nanoseconds(randI()));
#pragma omp atomic
                    num_nodes += node->data();
                }
                node = node->next();
            }
        }
    }
    return num_nodes;
}
```

Linked-List traversal(2)



Producer-Consumer(1)

```
Buffer *gl_buffer = 0;

int consume()
{
    int num = gl_buffer->buf_[--gl_buffer->len_].x_;
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    return num;
}

void produce(int key)
{
    gl_buffer->buf_[gl_buffer->len_++].x_ = key;
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}

int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);

    int total = 0;

    for (int i = 0; i < iterations; ++i) {
        produce(i);
        total += consume();
    }

    return total;
}
```

```
Buffer *gl_buffer = 0;

int consume()
{
    int num = 0;
#pragma omp critical
    {
        num = gl_buffer->buf_[--gl_buffer->len_].x_;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    return num;
}

void produce(int key)
{
#pragma omp critical
    {
        gl_buffer->buf_[gl_buffer->len_++].x_ = key;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}

int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    long long int total = 0;

#pragma omp parallel for reduction(+: total)
    for (int i = 0; i < iterations; ++i) {
        produce(i);
        total += consume();
    }

    return total;
}
```

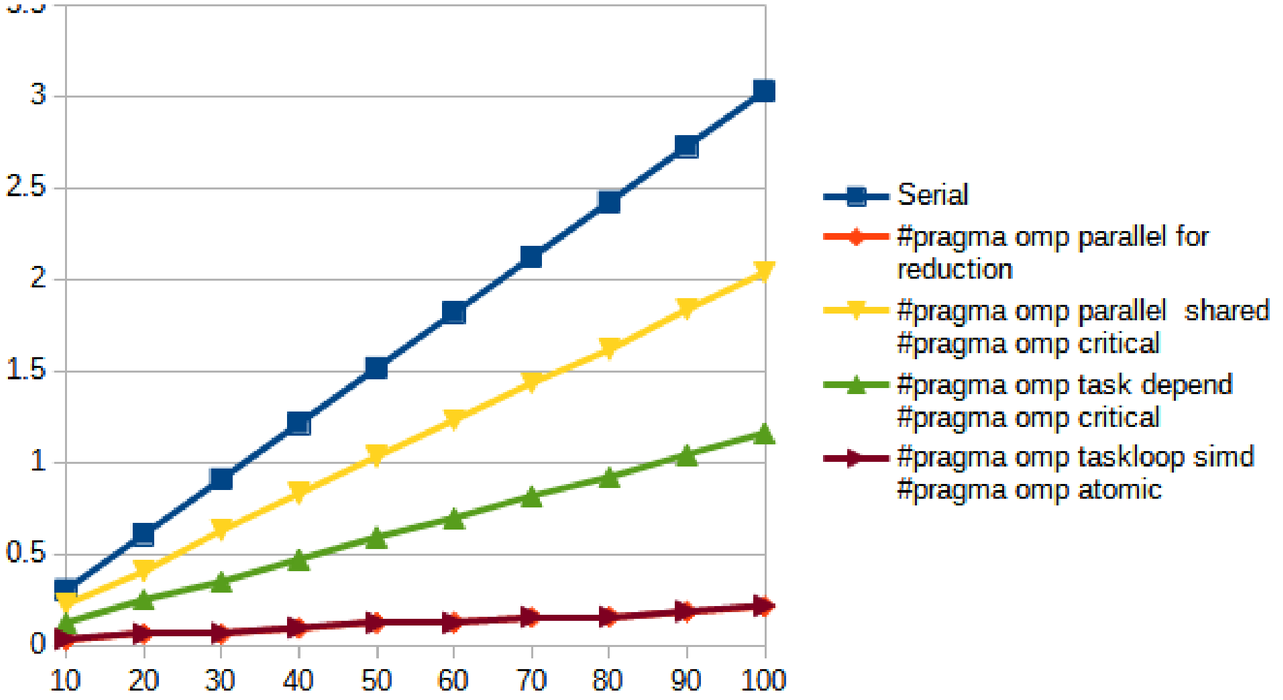
Producer-Consumer(2)

```
Buffer *gl_buffer = 0;

int consume()
{
    int num = 0;
#pragma omp critical
    {
        if (gl_buffer->len_ > 0) {
            num = gl_buffer->buf_[--gl_buffer->len_].x_;
        }
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    return num;
}

void produce(int key)
{
#pragma omp critical
    {
       gl_buffer->buf_[gl_buffer->len_++].x_ = key;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
}

int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    int total = 0;
    int finished_production = 0;
    int abort = 0;

#pragma omp parallel shared(finished_production) firstprivate(abort)
    {
        if (omp_get_thread_num() == 0) {
            for (int i = 0; i < iterations; ++i) {
                produce(i);
            }
            finished_production = 1;
        } else {
            while (!abort) {
                int temp = consume();
#pragma omp critical
                {
                    total += temp;
                    if (finished_production && gl_buffer->len_ == 0)
                    {
                        abort = 1;
                    }
                }
            }
        }
    }

    return total;
}
```

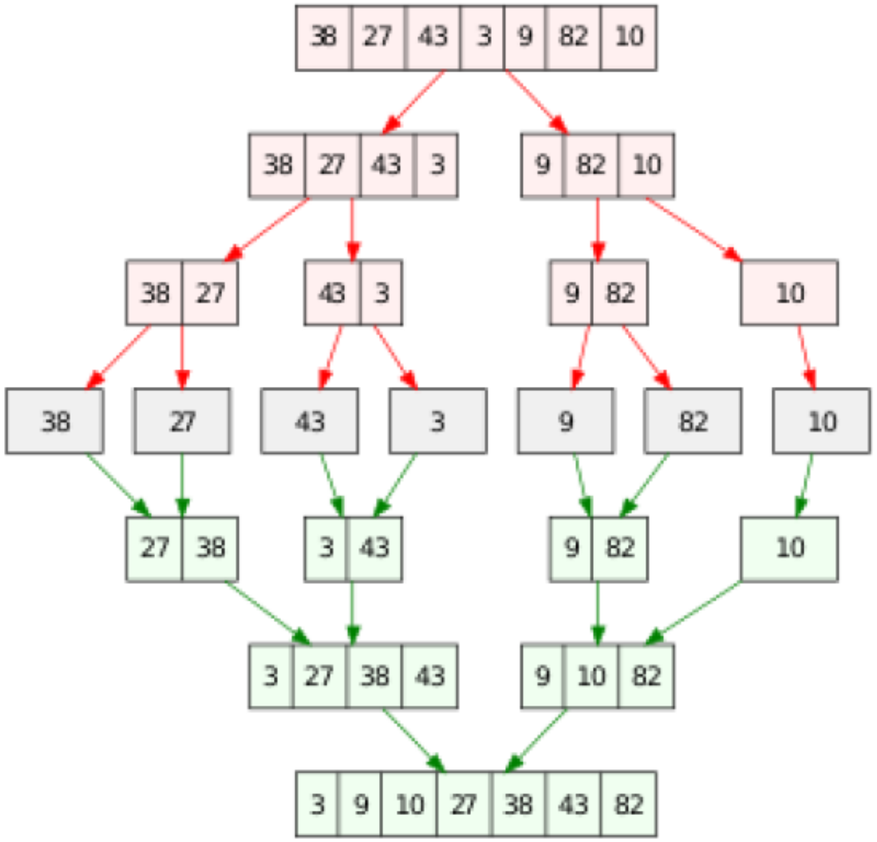
```
int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    int total = 0;
    int x = 0;

#pragma omp parallel
    {
#pragma omp single
        {
            for (int i = 0; i < iterations; ++i) {
#pragma omp task depend(out : x)
                {
                    produce(i);
                    x = 1;
                }
            }

            for (int i = 0; i < iterations; ++i) {
#pragma omp task depend(in : x)
                {
                    int temp = consume();
#pragma omp critical
                    {
                        total += temp;
                    }
                }
            }
        }
    }
    return total;
}
```

Producer-Consumer(3)

```
int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    int total = 0;
    int x = 0;
#pragma omp parallel
    {
#pragma omp single
        {
#pragma omp taskloop simd
            for (int i = 0; i < iterations; ++i) {
                produce(i);
            }
#pragma omp taskloop
            for (int i = 0; i < iterations; ++i) {
                int temp = consume();
#pragma omp atomic
                total += temp;
            }
        }
    }
    return total;
}
```



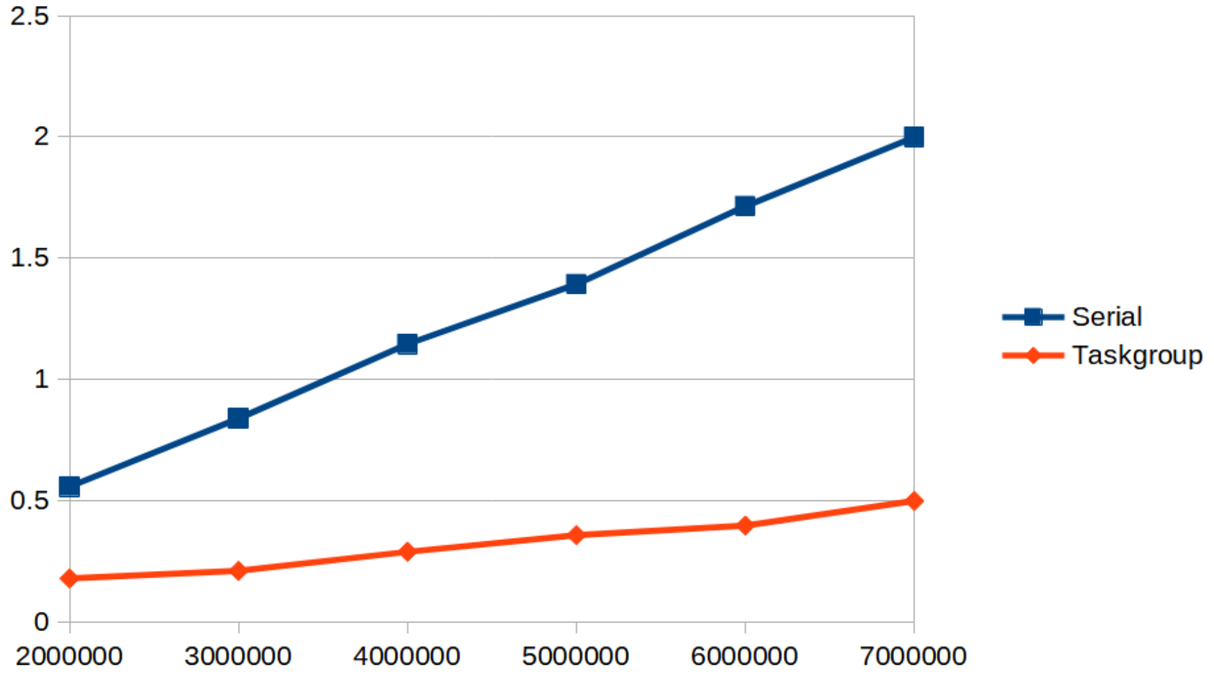
Mergesort(1)




```
void mergeSort(int arr[], int l, int r)
{

    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void mergeSort_wrapper(int *arr, int lhs, int rhs)
{
    mergeSort(arr, lhs, rhs);
}
```

Mergesort(2)

```
#define WORKLOAD 16000

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{

    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;
#pragma omp taskgroup
        {
#pragma omp task shared(arr) untied if (r - l >= WORKLOAD)
            mergeSort(arr, l, m);
#pragma omp task shared(arr) untied if (r - l >= WORKLOAD)
            mergeSort(arr, m+1, r);
#pragma omp taskyield
        }
        merge(arr, l, m, r);
    }
}

void mergeSort_wrapper(int *arr, int lhs, int rhs)
{
#pragma omp parallel
#pragma omp single
    mergeSort(arr, lhs, rhs);
}
```



Quicksort(1)

```
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment index of smaller element
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void qsort(int array[], int low, int high)
{
    if (low < high) {
        size_t pi = partition(array, low, high);
        qsort(array, low, pi - 1);
        qsort(array, pi + 1, high);
    }
}

void qsort_wrapper(int array[], int low, int high)
{
    qsort(array, low, high);
}
```

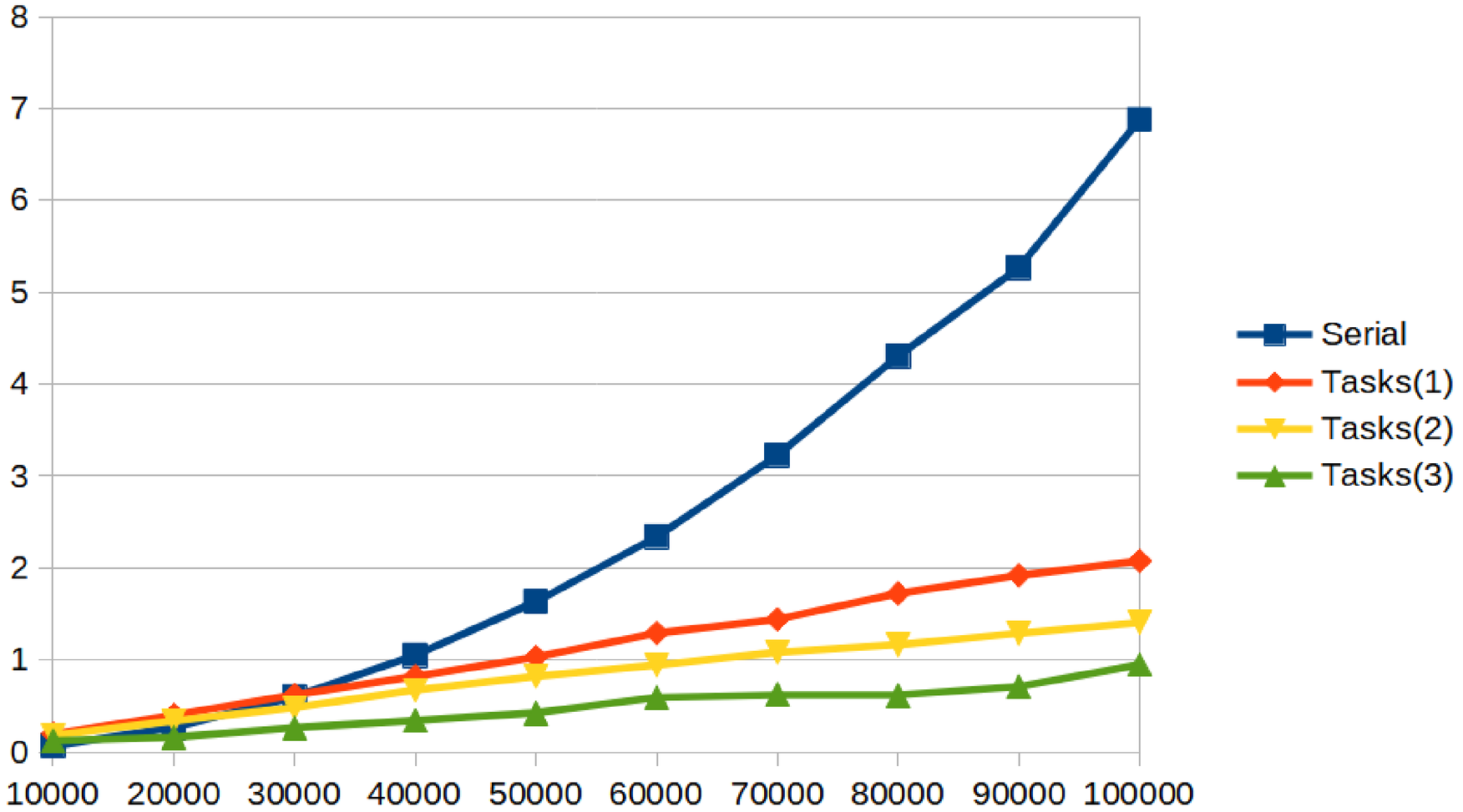
```
int partition(int array[], int low, int high)
{
    //select picot element
    int pivot = array[high];
    int i = (low - 1);

    // Put the elements smaller  than pivot on the lef
    // and greater than pivot  on the right of pivot.
    for (int j = low; j <= high; j++) {
        if (array[j] < pivot) {
            ++i;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i + 1], array[high]);
    return (i + 1);
}

void qsort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
#pragma omp task
        {
            qsort(array, low, pi - 1);
        }
#pragma omp task
        {
            qsort(array, pi + 1, high);
        }
    }
}

void qsort_wrapper(int array[], int low, int high)
{
#pragma omp parallel
#pragma omp single
    {
        qsort(array, low, high);
    }
}
```

Quicksort(2)

```
int partition(int array[], int low, int high)
{
    //select picot element
    int pivot = array[high];
    int i = (low - 1);

    // Put the elements smaller  than pivot on the left
    // and greater than pivot  on the right of pivot.
#pragma omp parallel for
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            ++i;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i + 1], array[high]);
    return (i + 1);
}

void qsort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
#pragma omp task
        {
            qsort(array, low, pi - 1);
        }
#pragma omp task

        {
            qsort(array, pi + 1, high);
        }
    }
}

void qsort_wrapper(int array[], int low, int high)
{
#pragma omp parallel
#pragma omp single
    {
        qsort(array, 0, high);
    }
}
```

```
int partition(int array[], int low, int high)
{
    //select picot element
    int pivot = array[high];
    int i = (low - 1);

    // Put the elements smaller  than pivot on the left
    // and greater than pivot  on the right of pivot.
    for (int j = low; j <= high; j++) {
        if (array[j] < pivot) {
            ++i;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i + 1], array[high]);
    return (i + 1);
}

void qsort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
#pragma omp task
        {
            qsort(array, low, pi - 1);
        }
#pragma omp task
        {
            qsort(array, pi + 1, high);
        }
#pragma omp taskwait
    }
}

void qsort_wrapper(int array[], int low, int high)
{
#pragma omp parallel
#pragma omp single
    {
        qsort(array, low, high);
    }
}
```

Quicksort(3)


8
7
6
5
4
3
2
1
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Serial
Tasks(1)
Tasks(2)
Tasks(3)

Πολλαπλασιασμός πινάκων(1)

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)

    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
```

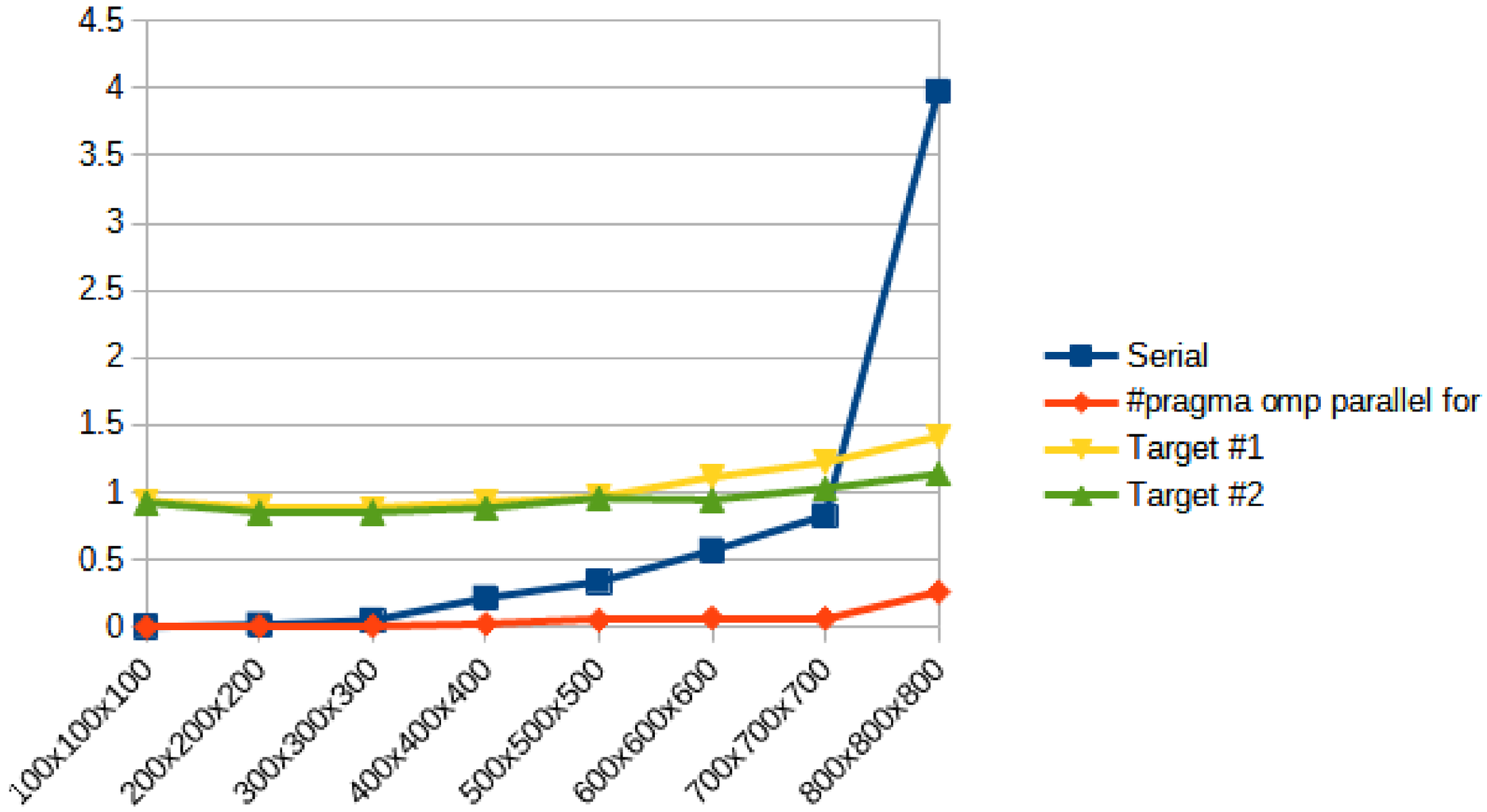
```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

#pragma omp parallel for
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

Πολλαπλασιασμός πινάκων(2)

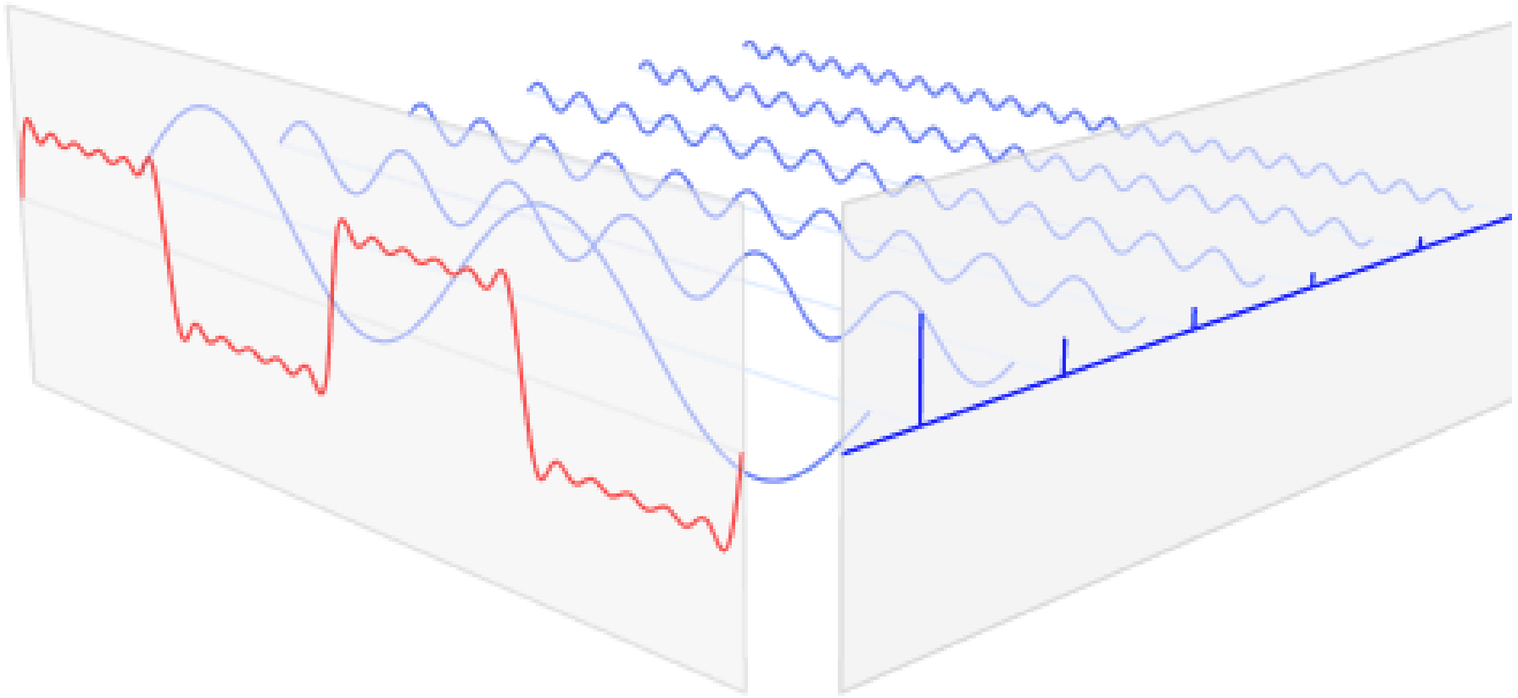
```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

    int a_size = r1 * c1;
    int b_size = r2 * c2;
    int c_size = r3 * c3;

#pragma omp target map(to: a[0:a_size], b[0: b_size]) \
                                        map(from: c[0:c_size])
#pragma omp parallel for collapse(2)
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
#pragma omp simd reduction(+ : temp)
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
 {
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

    int a_size = r1 * c1;
    int b_size = r2 * c2;
    int c_size = r3 * c3;

 #pragma omp target teams distribute parallel for simd\
            map(to:a[:a_size], b[:b_size])\
                map(from:c[:c_size])
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
 }
```

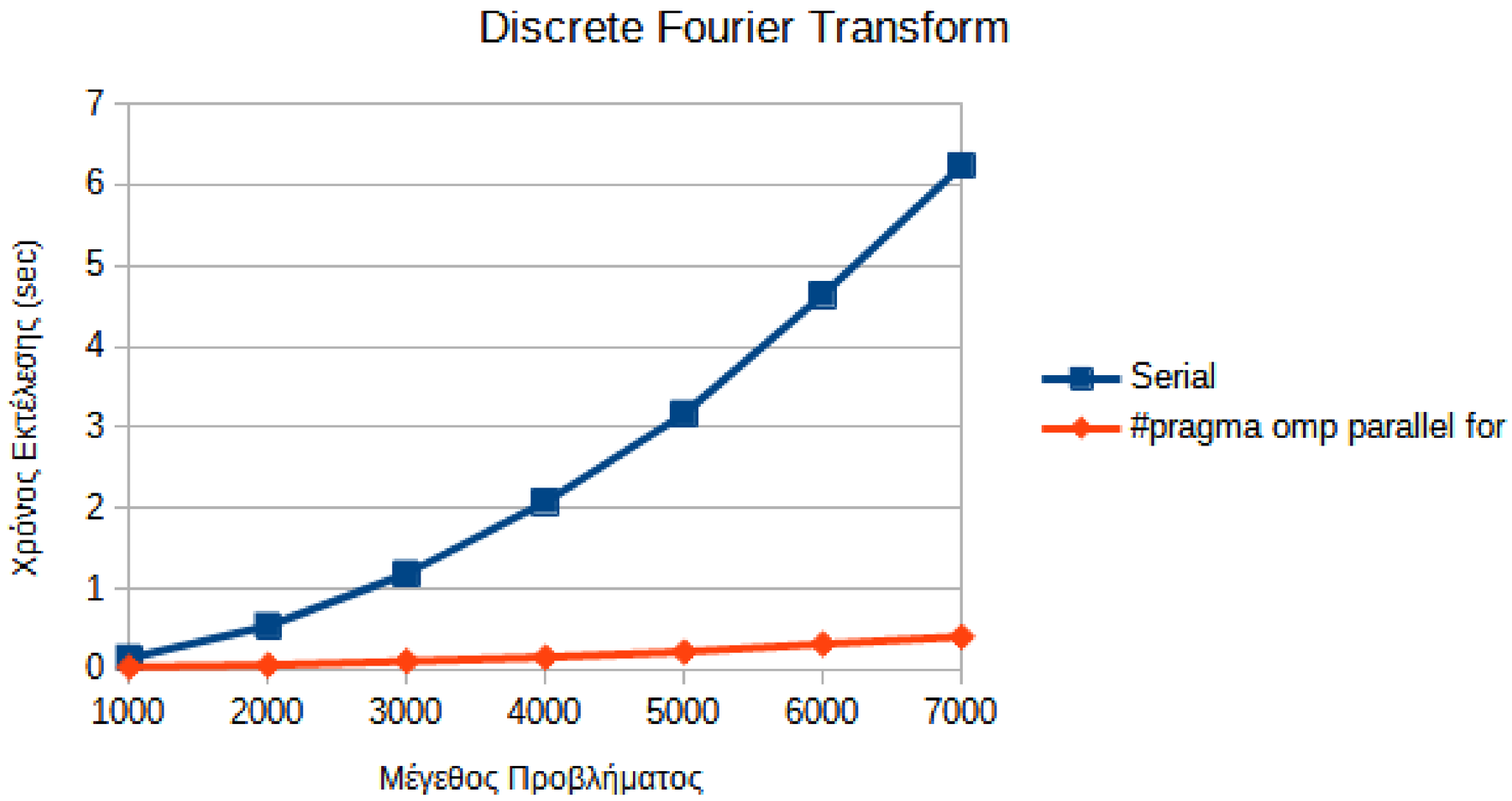
Πολλαπλασιασμός πινάκων(3)



Μετασχηματισμός Fourier(1)



Μετασχηματισμός Fourier(2)



Ευχαριστώ πολυ για το χρόνο σας!