Raport

Zrównoleglanie MergeSort

Franciszek Grzędzicki 291888 (grzedzicki@mat.umk.pl)

1. Problem

Wydajność algorytmy sortowania przez scalania przy wielordzeniowej maszynie.

2. Pomysł

Zrównoleglenie algorytmu, aby lepiej wykorzystywał dostępne zasoby komputera.

3. Rozwiązanie

Wykorzystanie wątków pthread z języka C.

4. Algorytm (kroki)

Sortowanie przez scalanie (Serial)

1.Podział tablicy liczb na dwie równe połowy.
2. Sortowanie przez scalanie dla każdej z nich oddzielnie.
3. Połączenie posortowanych podciągów w jeden posortowany ciąg.

Sortowanie przez scalanie (Parallel)

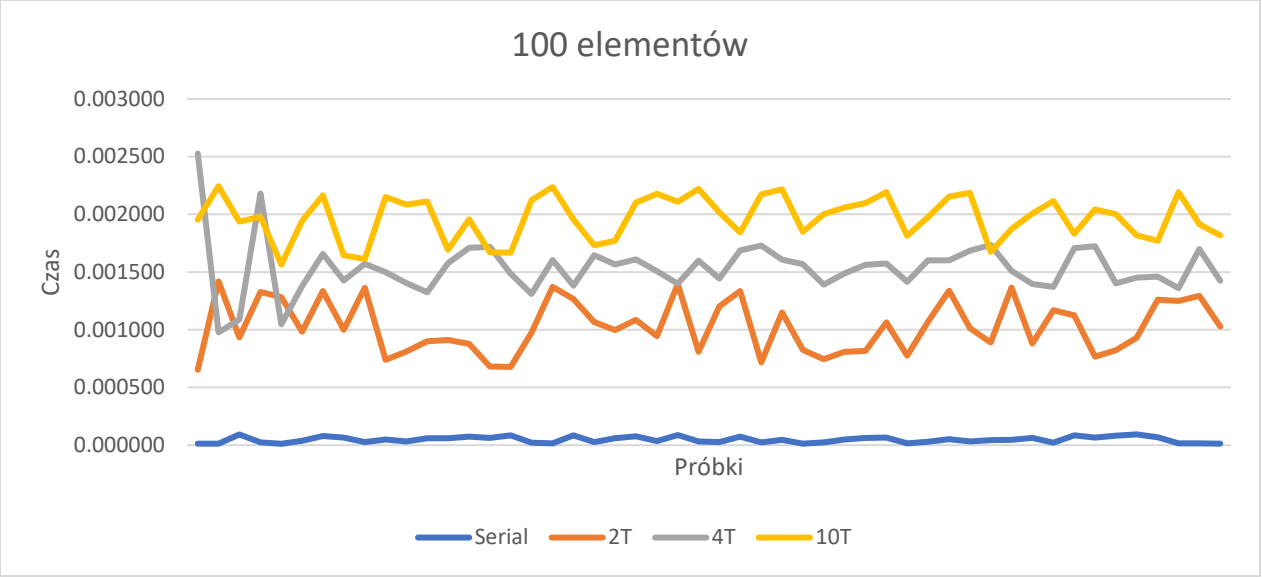
1. Podział tablicy liczb względem liczby wątków.
2. Sortowanie przez scalanie dla każdego wątku odbywa się równolegle.
3. Połączenia posortowanych tablic z wątków w jeden posortowany ciąg.

5. Metodologia testowania

Maszyna wirtualna Arch Linux dystrybucja Manjaro 20.1 XFCE
Procesor: i5-6600K 4.7GHz , 4 rdzenie, 4 wątki, 64bit
RAM: 16384MB 3000MHz
Plik stronicowania: 4096MB
Dysk: Crucial M550 Wirtualizcja: Hyper-V
Każdy test był wykonywany na tej samej konfiguracji sprzętowej z dostępem do całości zasobów.

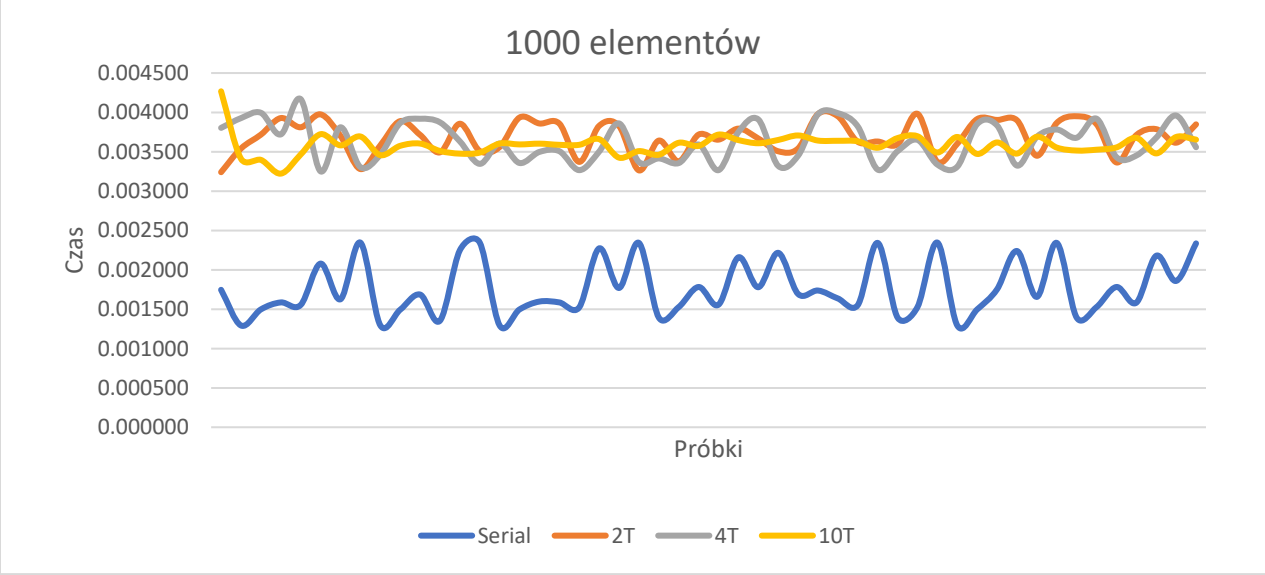
6. Wyniki

I.100 elementowa tablica danych



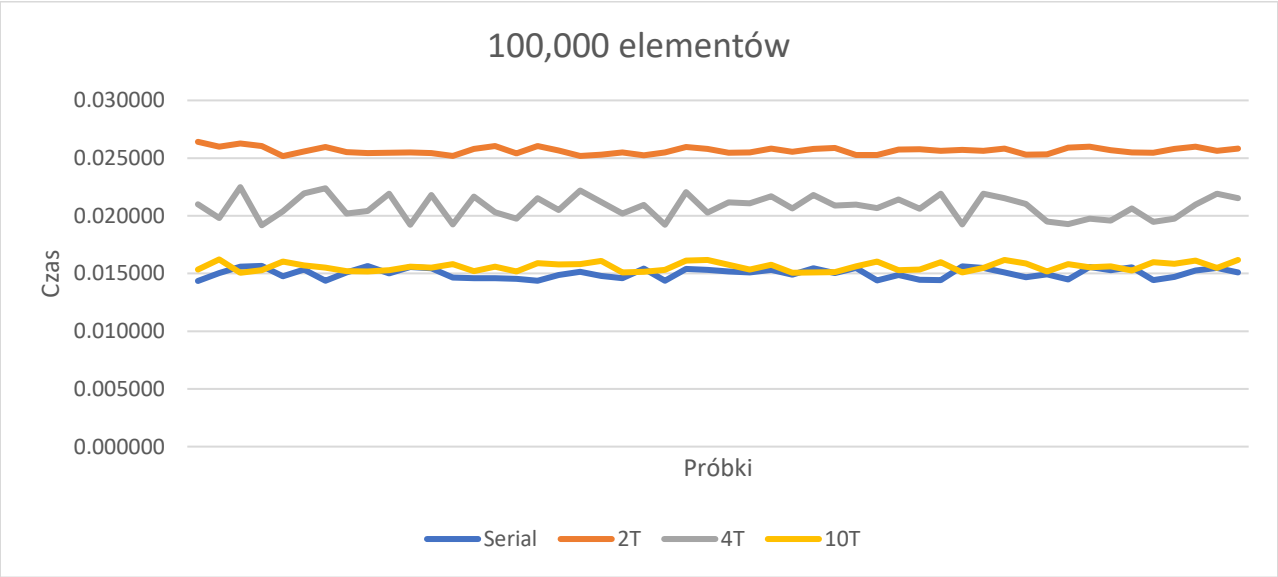
Jak można było przewidzieć przy 100 elementowej tablicy, sortowanie zrównoleglone nie ma sensu, ponieważ zajmuje o wiele więcej czasu niż zwykły MergeSort.

II. 1,000 elementowa tablica



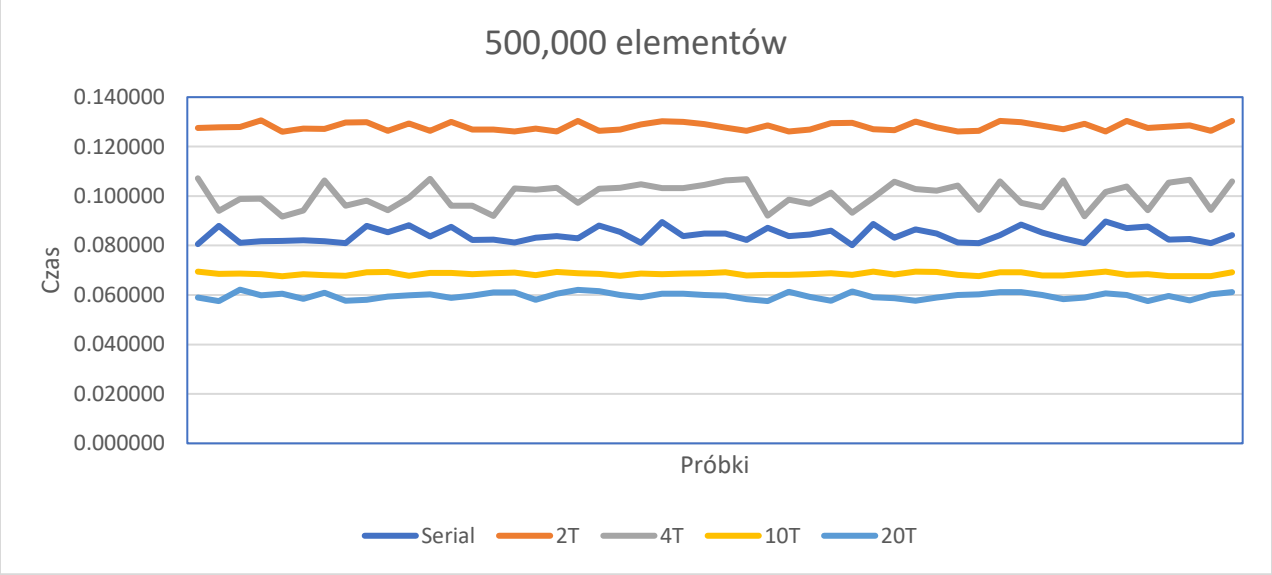
Przy 1000 elementów zrównoleglenie dalej nie ma sensu.

III. 100,000 elementowa tablica



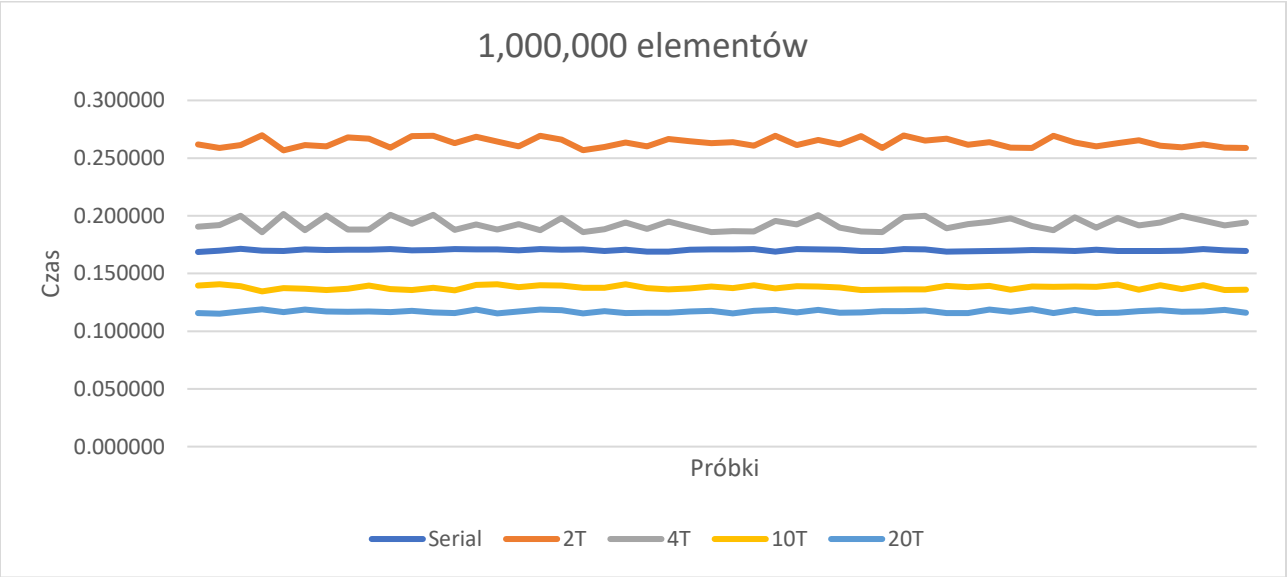
Przy 100,000 robi się już ciekawiej, ponieważ jak można zaobserwować wersja równoległa z użyciem 10 wątków przeplata się z czasem sortowanie zwykłego algorytmu. Sugeruje to, że przy większej ilości danych zrównoleglony algorytm ma sens istnienia.

IV. 500,000 elementowa tablica

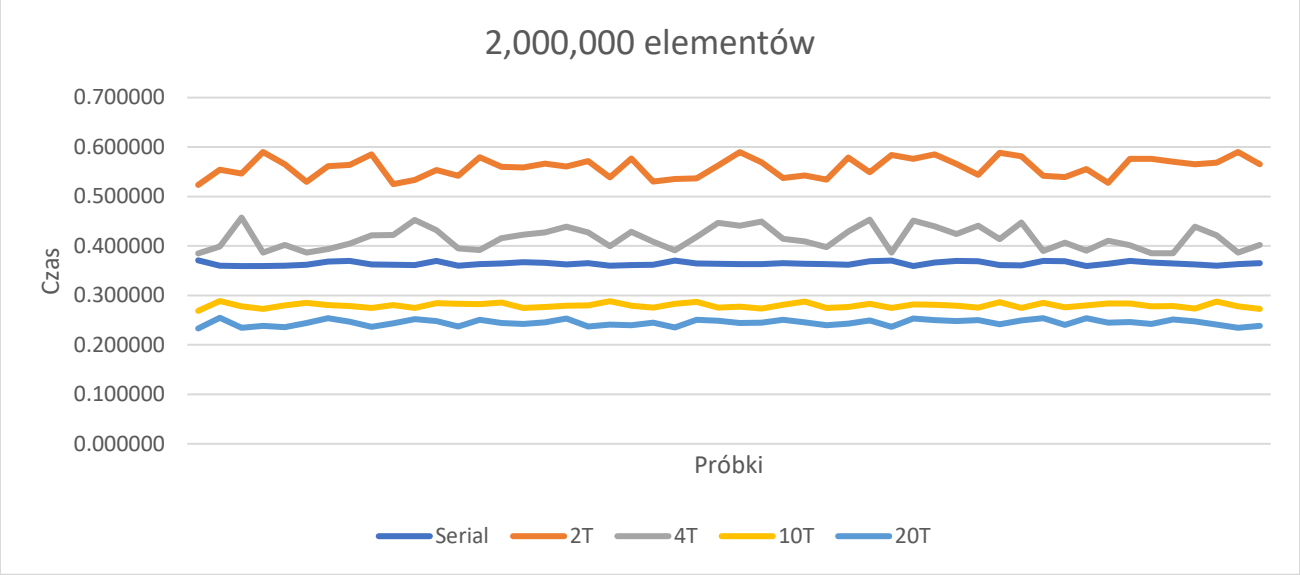


Tablica zawiera 500,000 elementów i z wykresu można łatwo odczytać, że algorytm zrównoleglony z 10 wątkami, a także ze względu na ilość danych również z 20 wątkami, jest szybszy od zwykłego MergeSorta.

V. 1,000,000 elementowa tablica



VI. 2,000,000 elementowa tablica



Podsumowanie

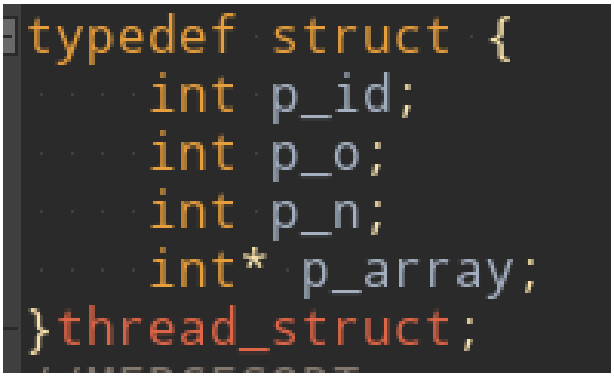
Bazując na czasach uzyskanych przez algorytmy niezależnie od siebie stwierdzam, że algorytm zrównoleglania MergeSort działa poprawnie oraz ma sens istnienia w przypadku:

- **odpowiedniej ilości danych** – na małej tablicy algorytm ten jest za wolny względem zwykłego

- **odpowiedniej liczby wątków** – z trendów znajdujących się na wykresach można zauważyć, że algorytm 2 i 4 wątkowy nie ma sensu być w ogóle implementowany

- **mocy maszyny** – mimo wszystko, zrównoleglony algorytm wymaga więcej zasobów, aby móc wykonać kilka operacji na raz

Opis kodu



Po przeanalizowaniu pomysłu zrównoleglenia sortowania przez scalania zdecydowałem się na użycie struktur, które będą zawierały wszystkie ważne informacje o danej części tablicy przydzielonej wątkowi.

p_id – id wątku

p_o – reszta z dzielenia, pomaga przy ustalaniu ostatniego indeksu tablicy

p_n – wielkość tablicy

*** p_array** – dynamiczna tablica int, zdecydowałem się na nią, ponieważ umożliwia łatwiejsze oraz szybsze inicjowanie dużych tablic, w przeciwieństwie do statycznej tablicy

```
void merge_sorting(void* arg){
    int a,pom_id;
     thread_struct *t_s = arg;
     if(t_s->p_o == 0) a=1;
     else a=2;
     int l = 0;
     int r = t_s->p_n-a;
     int id = t_s->p_id;
     if( l < r){
     merge_sort(t_s->p_array,l,r,id);
     if( id !=0) pom_id = 100 +id;
     else pom_id=100;
     writefile(t_s->p_array,r+1,pom_id);
     }
}
```

Funkcja wywoływana przez każdy wątek.
Następuje tutaj przygotowanie do algorytmu sortowania przez scalania oraz wejście do samego algorytmu.
thread_struct *ts - tworzy strukturę która pozwala na dostęp do struktur uzyskanych przez wątek
if(t_s->p_o) - sprawdza czy reszta dzielenia przypisana wątkowi wynosi 0, pomaga to przy ustaleniu prawego indeksu tablicy
W następnej kolejności przydzielany jest lewy oraz prawy indeks.
Inicjalizacja id pomaga w utworzeniu przez dany wątek pliku ze swoją posortowaną tablicą.

```
void merge_sort(int *array, int l, int r){
    if( l < r){
        int m = l + ( r - l) / 2;
        merge_sort(array,l,m);
        merge_sort(array,m+1,r);
        merge_function(array,l,m,r);
    }
}
```

Rekurencja dla sortowania przez scalania. Może być zarazem użyta dla równoległego sortowania jak i zwykłego.

```
void merge_function(int *array, int l, int m, int r){
    int tmp_left = m - l +1;
    int tmp_right = r - m;
    int array_left[tmp_left];
    int array_right[tmp_right];


    for(int i=0; i< tmp_left;i++)
        array_left[i] = array[l + i];

    for(int i=0; i<  tmp_right;i++)
        array_right[i] = array[m + 1 + i];
    int i=0,j=0,k=0;

    while(i < tmp_left && j < tmp_right){
        if(array_left[i] <= array_right[j]){
            array[l + k] = array_left[i];
            i++;
        }else{
            array[l + k] = array_right[j];
            j++;
        }
    k++;
    }

    while(i< tmp_left){
        array[l + k] = array_left[i];
        k++;
        i++;
    }
    while(j < tmp_right){
        array[l + k] = array_right[j];
        k++;
        j++;
    }
```

Zwykły algorytm sortowania przez scalanie.
Tablica zostaje podzielona na dwie części i przydzielona do lewej oraz prawej tymczasowej tablicy. Następnie zostaje sprawdzone czy liczba z lewej tablicy jest mniejsza bądź równa prawej, jeśli tak to zostaje przypisana do oryginalnej tablicy, jeśli nie to przypisana zostaje wartość z prawej tablicy. Jeśli tablica nie mogła zostać podzielona na równe części dwie ostatnie pętle dodają resztę liczb do tablicy.

```
void final_merge(int *array, int n, int ag, int len,int s){
    int l,r,m;
    for(int i =0; i<n; i = i+2){
        l = i * (len * ag);
        r = ((i + 2) * len * ag) - 1;
        m = l + (len * ag) -1;
        if(r >= s)
            r = s -1;
        merge_function(array,l,m,r);
    }
    if(n/ 2 >= 1)
        final_merge(array, n/2,ag*2,len,s);
}
```

Scalanie wszystkich podciągów w jeden.
Podciągi zostają rekurencyjnie scalone ze sobą.
Pętla jest wykonywana tyle razy, ile zostało wykorzystanych w algorytmie wątków.

n – ilość wątków
ag – zmienna pomocnicza przy łączeniu podciągów
len – wielkość tablicy wątku
s – wielkość całej tablicy

```
void testsort(int *array,int n,int s){
    for(int i=0;i<n-1;i++)
        if(!(array[i] <= array[i+1])){
            switch(s){
        case 2:
            printf("Out of order in Parallel Merge Sort! %d %d\n", array[i], array[i+1]);
            break;
        case 3:
            printf("Out of order in Serial Merge Sort! %d %d\n", array[i], array[i+1]);
            break;
        }
        return -1;
        }
    printf("Sorting is in order.");
}
```

Funkcja sprawdza, czy liczby są posortowane rosnąco.

```
void writefile(int* array,int n, int s){
    FILE *fp;
    int num;
    switch(s){
        case 0:
            fp = fopen ("array.txt","w");
            break;
        case 1:
            fp = fopen ("bubble.txt","w");
            break;
        case 100:
            fp = fopen ("merge1.txt","w");
            break;
```

Funkcja służąca do zapisu wyników sortowania oraz samej tablicy. Przygotowana jest ona do sortowań typu 2-4-10 wątkowych.

```
void init(int n,int *array){
    for(int i=0; i<n;i++){
        array[i]= rand();
    }
    writefile(array,n,0);
    display(n,array,0);
}
```

Funkcja inicjalizująca tablicę.

```
void display(int n, int *array,int s){
    switch(s){
        case 0:
            printf("Array: ");
            break;
        case 2:
            printf("Parallel MergeSort: ");
            break;
        case 3:
            printf("\nSerial MergeSort: ");
            break;

    }
    printf("");
    for(int i=0; i<n;i++){
        printf("%d ",array[i]);
    }
    printf("\n");
}
```

Funkcja odpowiedzialna za wyświetlanie tablicy.

```c
int main(int argc, char **argv)
{
    int n,seed;
    printf("Wielkosc tablicy oraz seed:\n");
    scanf("%d %d", &n,&seed);
    srand(seed);
    int* array= (int*)malloc(sizeof(int)*n);
    init(n,array);
    pthread_t threads[THREADS];
    int divide,mod,div;
    printf("\n_____\n");
    struct timespec start, finish;
    double elapsed;
    int* arraytomerge= (int*)malloc(sizeof(int)*(n));
    clock_gettime(CLOCK_MONOTONIC, &start);
```

Pobranie wielkości tablicy oraz wartości seed od użytkownika.
srand na podstawie wartości od użytkownika

***array** – utworzenie tablicy dynamicznej na podstawie wielkości n
init(n,array) – wywołanie funkcji wypełniającej tablicę
pthread_t – utworzenie wątków na podstawie zadeklarowanej wartości

struct timespec – służą do zapisu początku oraz zakończenia działania algorytmów
clock_gettime – pobranie aktualnego czasu

```
for(int i=0;i<THREADS;i++){
    divide = n/THREADS;
    thread_struct args[THREADS];
    args[i].p_id=i;
    int mod=n%THREADS;
    if(mod !=0) div = divide + mod;
    else div = divide + mod;
    args[i].p_n=div;
    args[i].p_o=mod;

    if(i == THREADS-1) {args[i].p_array = (int*)malloc(sizeof(int) * div);
    memcpy(args[i].p_array,array+(i*divide),(div  * sizeof(int)));
    }
    else if( i==0) {args[i].p_array = (int*)malloc(sizeof(int) * divide);
    memcpy(args[i].p_array,array,(divide * sizeof(int)));
    }
    else {args[i].p_array = (int*)malloc(sizeof(int) * divide);
    memcpy(args[i].p_array,array+(i*divide),(divide * sizeof(int)));
    }

if (pthread_create(&threads[i], NULL, &merge_sorting, (void *)&args[i]) != 0) {
    printf("pthread error[%d]", i);
    return -1;
}
}
```

Pętla zarządzająca przydzielaniem danych do struktury każdego wątku.
divide – wielkość tablica podzielona na wszystkie wątki (całkowita)
thread_struct – struktura do przechowywania danych dla wątku
div – jeśli modulo tablicy przez ilość wątków nie wynosi 0 to ostatni wątek ma otrzymać niepasujące liczby
Zerowy wątek ma otrzymać tablicę wielkości części całkowitej z dzielenia przez ilość wątków.
Każdy kolejny wątek poza ostatnim otrzymuje taką samą wielkość tablicy jak wątek zerowy, lecz otrzymuje dane z tablicy liczby przesunięte o wartość poprzednich wątków.
Ostatni wątek dostaje wielkość taką samą jak poprzednie wątki lecz w razie potrzeby zwiększoną o ilość liczb z reszty z dzielenia przez ilość wątków.

```
thread_struct rec[THREADS];
for(int i=0;i<THREADS;i++)
pthread_join(threads[i], &rec[i]);
```

Zakończeni wszystkich wątków oraz zebranie ich danych do struktur, aby móc z nich dalej korzystać.

```
memcpy(arraytomerge,rec[0].p_array,(divide * sizeof(int)));
for(int i=1;i<THREADS;i++){
    if(i!= THREADS-1) memcpy(arraytomerge + (divide * i),rec[i].p_array,(divide * sizeof(int)));
    else memcpy(arraytomerge + (divide * i),rec[i].p_array,(div * sizeof(int)));
}
```

Skopiowanie wartości tablic z wątków do odpowiednich miejsc w nowej tablicy arraytomerge.

```
final_merge(arraytomerge, THREADS,1,divide,n);
if(THREADS == 2) writefile(arraytomerge,n,102);
else if(THREADS == 4) writefile(arraytomerge,n,104);
else if(THREADS == 10) writefile(arraytomerge,n,110);
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed = (finish.tv_sec - start.tv_sec);
elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
printf("\nElapsed Pararrel Merge Sort Time: %lf\n",elapsed);
display(n,arraytomerge,2);
testsort(arraytomerge,n,2);
```

final_merge - wywołanie funkcji, która ustali poprawną kolejność między tablicami pochodzącymi z wątków.
if(THREADS) - na podstawie odpowiedniej ilości wątków, wykonaj zapis ostatecznego wyniku do pliku.
clock_gettime – uzyskanie czasu zakończenia działania algorytmu
elapsed – ustalenie czasu działania algorytmu
display – wyświetlenie posortowanej tablicy
testsort – wywołanie funkcji sprawdzającej poprawność sortowania

```
clock_gettime(CLOCK_MONOTONIC, &start);
merge_sort(array,0,n-1);
writefile(array,n,120);
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed = (finish.tv_sec - start.tv_sec);
elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
printf("\nElapsed Serial Merge Sort Time: %lf\n",elapsed);
display(n,array,4);
testsort(array,n,4);
return 0;
```

clock_gettime – uzyskanie czasu zakończenia działania algorytmu
merge_sort - wywołanie mergesort na danej tablicy
elapsed – ustalenie czasu działania algorytmu
display – wyświetlenie posortowanej tablicy
testsort – wywołanie funkcji sprawdzającej poprawność sortowania