Politechnika Opolska
Wydział Elektrotechniki, Automatyki i Informatyki
Instytut Informatyki

Programowanie współbieżne i rozproszone
Informatyka, studia stacjonarne II stopnia
Laboratorium

Sprawozdanie

Algorytmy sortowania

Prowadzący:
dr hab. inż. Jan Sadecki, prof. PO

Ćwiczenie zrealizował:
Marcin Kuchnia
nr indeksu 93210
grupa L1

Opole, maj 2020

Metody obliczeniowe

W ramach ćwiczenia zmierzono czasy sortowania tablic. Pierwotnie planowano dokonać większej ilości badań, jednak wyniki uzyskane już w pierwszym podejściu wydają się właściwie obrazować własności algorytmów. Każdy algorytm wielowątkowy przetestowano na od 1 do 12 wątków. Wszystkie algorytmy przetestowano na tablicach wielkości: 1250, 2500, 5000, 10000 elementów. Dla każdej z tych tablic przeprowadzono po 10 pomiarów.

Funkcje losujące i generujące macierze

Algorytmy działały na wygenerowanych losowotablicach. Funkcja *fRand()* zwraca wartość zmiennoprzecinkową o wartości od -1000 do 1000. Podobna funkcja *intRand()* zwraca liczbę całkowitą od -3 do 3 i wykorzystywana jest tylko w celach testowych, weryfikujących poprawność działania algorytmu. Funkcja *generateRandomArray()* pobiera jako parametry rozmiar tablicy oraz wskaźnik do pierwszego jej elementu. Korzystając z odpowiedniej funkcji losującej wypełnia ona tablicę losowymi wartościami. Funkcja *generatePermutationArray()* umożliwia wygenerowanie losowej permutacji kolejnych liczb naturalnych. Taki zbiór został wykorzystany w badaniu, ponieważ niektóre algorytmy sortujące są ograniczone właśnie do takiego zbioru. Test przeprowadzono z myślą, by wszystkie algorytmy potraktować identycznie.

Implementacja funkcji

```
float fRand()
{
    float fMin = -1000;
    float fMax = 1000;
    float f = (float)rand() / RAND_MAX;
    return fMin + f * (fMax - fMin);
}

int intRand()
{
    int fMin = -3;
    int fMax = 3;
    int f = rand() % (fMax - fMin) + fMin;
    return f;
}

void generateRandomArray(int n, float* Array)
{
    for (int i = 0; i < n; i++)
    {
        Array[i] = fRand();
    }
}

void generatePermutationArray(int n, float* Array)
{
    for (int i = 0; i < n; i++)
    {
        Array[i] = i;
    }
    shuffle(Array, Array+n, default_random_engine(time(NULL)));
}
```

Funkcje pomocnicze do tablic

Dodatkowo stworzone funkcje ułatwiają pewne elementarne operacje na tablicach. Funkcja *copyArray()* kopiuje tablicę przekazaną w parametrze, alokuje jej nowy egzemplarz oraz zwraca wskaźnik do nowoutworzonego egzemplarza. Funkcja *printArray()* umożliwia wyświetlenie zadanej macierzy w celu sprawdzenia jej zawartości. Daje to możliwość weryfikacji poprawności obliczeń. Funkcja *printArrays()* działa w bardzo podobny sposób, wyświetla jednak dwie tablice w dwóch kolumnach.

Implementacja funkcji

```
float* copyArray(int n, float* Array)
{
    float* newArray = new float[n];
    for (int i = 0; i < n; i++)
    {
        newArray[i] = Array[i];
    }
    return newArray;
}

void printArray(int n, float* Array)
{
    for (int i = 0; i < n; i++)
    {
        cout << Array[i] << endl;;
    }
}

void printArrays(int n, float* Array, float* sortedArray)
{
    for (int i = 0; i < n; i++)
    {
        cout << setw(20) << left << Array[i] << setw(20) << left << sortedArray[i] << endl;;
    }
}
```

Sortowanie bąbelkowe

Sortowanie bąbelkowe jest algorytmem sortowania bardzo prostym w założeniu. Porównując kolejne elementy tablicy parami porządkujemy te pary, zamieniając je wartościami w razie potrzeby. Po jednym przejściu takiego algorytmu przez tablicę w efekcie otrzymujemy największą wartość na swoim miejscu. Jest to jednak algorytm o dość dużej złożoności obliczeniowej $O(n^2)$. W efekcie sortowanie to jest efektywne tylko dla relatywnie niewielkich zbiorów. W implementacji utworzono także przeciążoną wersję funkcji *bubbleSort()*, która pozwala wygodnie określić przedział, w jakim ma zostać posortowana tablica podana jako argument.

Implementacja algorytmu

```
void bubbleSort(int n, float* Array)
{
    for (int i = 0; i < n -1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (Array[j] > Array[j + 1])
            {
                float temp = Array[j];
                Array[j] = Array[j+1];
                Array[j + 1] = temp;
            }
        }
    }
}

void bubbleSort(float* Array, int left, int right)
{
    bubbleSort(right - left, &Array[left]);
}
```

Stworzono także dwie wersje równoległe algorytmu bąbelkowego. Funkcja *pipeBubbleSort()* przydziela wątkom kolejne iteracje algorytmu sekwencyjnego. Oznacza to, że kolejna iteracja może rozpocząć się znacznie wcześniej niż w wersji sekwencyjnej i odbywać się równolegle do poprzedniej. Konieczne jest jednak monitorowanie postępów, aby nie nastąpił konflikt w dostępie do danych. Zastosowano więc tablicę postępu *doneTable*.

Implementacja algorytmu

```
void pipeBubbleSort(int n, float* Array)
{
    //tablica postępu
    int* doneTable;
    doneTable = new int[n];
    for (int i = 0; i < n; i++) doneTable[i] = 0;

    #pragma omp parallel for schedule(dynamic,1) firstprivate(n) shared(doneTable, Array)
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            while (doneTable[j] != i || doneTable[j] != doneTable[j + 1]) { Sleep(0); }
            if (Array[j] > Array[j + 1])
            {
                float temp = Array[j];
                Array[j] = Array[j + 1];
                Array[j + 1] = temp;
            }
            doneTable[j] = i + 1;
        }
    }
    delete[] doneTable;
}
```

Kolejną modyfikacją sortowania bąbelkowego jest sortowanie nieparzyste-parzyste. Tutaj w każdej iteracji również zachodzi operacja porównania i wymiany parami. Różnicą jest to, że każdy element jest porównywany w iteracji tylko raz. Powoduje to, że w pierwszej iteracji porównywane są wartości na miejscach nieparzystych ze swoimi sąsiadami z prawej, w następnej analogiczna sytuacja zachodzi dla miejsc parzystych. Zrównoleglenie operacji porównania i wymiany zachodzi w ramach iteracji.

Implementacja algorytmu

```
void oddEvenSort(int n, float* Array)
{
    //tablica postępu
    int* doneTable;
    doneTable = new int[n];
    for (int i = 0; i < n; i++) doneTable[i] = 0;

    //kolejne fazy
    for (int i = 0; i < n; i++)
    {
        if (i % 2 == 0)
        //jeśli faza parzysta
        {
            #pragma omp parallel for firstprivate(n) shared(doneTable, Array)
            for (int j = 0; j < n;j += 2)
            {
                while (doneTable[j] != i || doneTable[j] != doneTable[j + 1]) { Sleep(0); }
                if (Array[j] > Array[j + 1])
                {
                    float temp = Array[j];
                    Array[j] = Array[j + 1];
                    Array[j + 1] = temp;
                }
                doneTable[j] = i + 1;
                doneTable[j + 1] = i + 1;
            }
            //jeśli nieparzysta liczba elementów ustaw ostatni jako obsłużony
            if (n % 2 == 1) {
                while (doneTable[n - 1] != i) { Sleep(0); }
                doneTable[n - 1] = i + 1;
            }
        }
        else
        //jeśli faza nieparzysta
        {
            //ustaw pierwszy element jako obsłużony
            while (doneTable[0] != i) { Sleep(0); }
            doneTable[0] = i + 1;
            //#pragma omp parallel for schedule(static) firstprivate(n) shared(doneTable, Array)
            for (int j = 1; j < n-1;j += 2)
            {
                while (doneTable[j] != i || doneTable[j] != doneTable[j + 1]) { Sleep(0); }
                if (Array[j] > Array[j + 1])
                {
                    float temp = Array[j];
                    Array[j] = Array[j + 1];
                    Array[j + 1] = temp;
                }
                doneTable[j] = i + 1;
                doneTable[j + 1] = i + 1;
            }
            //jeśli parzysta liczba elementów ustaw ostatni jako obsłużony
            if (n % 2 == 0) {
                while (doneTable[n - 1] != i) { Sleep(0); }
                doneTable[n - 1] = i + 1;
            }
        }
    }
    delete[] doneTable;
}
```

Sortowanie szybkie

Efektywny algorytm sortowania szybkiego jest oparty na regule „dziel i zwyciężaj”. Wybrany element staje się „osią” sortowania, z którą porównywane są kolejne elementy tablicy. Elementy mniejsze przenoszone są na lewą stronę, większe na prawą. W efekcie otrzymujemy oś znajdującą się na docelowej pozycji. Kolejnym krokiem staje się posortowanie podzbiorów osobno elementów mniejszych oraz większych od osi. Dzieje się to poprzez rekurencyjne wywołanie tego samego algorytmu dla ograniczonych zbiorów. Dodatkowe przeciążenie funkcji *quickSort()* pozwala wygodnie wywołać sortowanie szybkie dla całej tablicy, w sposób podobny do pozostałych funkcji sortujących. Próby zrównoleglenia algorytmu nie doszły do skutku, ponieważ w środowisku Visual Studio 2019 nadal obsługiwany jest standard OpenMP 2.0 z 2002 roku. Nie pozwala on w logiczny sposób zaimplementować rekurencyjnego wywoływania kolejnych podprocesów.

Implementacja algorytmu

```
void quickSort(float* Array, int left, int right)
{
    float v = Array[(left + right) / 2];
    int i, j;
    float x;
    i = left;
    j = right;
    do
    {
        while (Array[i] < v) i++;
        while (Array[j] > v) j--;
        if (i <= j)
        {
            x = Array[i];
            Array[i] = Array[j];
            Array[j] = x;
            i++; j--;
        }
    } while (i <= j);
    if (j > left) quickSort(Array, left, j);
    if (i < right) quickSort(Array, i, right);
}

void quickSort(int n, float* Array)
{
    quickSort(Array, 0, n-1);
}
```

Rank sort

Ten algorytm sortowania jest ograniczony do zbioru niepowtarzających się wartości. Dla kolejnych elementów tablicy zliczane są elementy o wartości mniejszej od niego. W efekcie uzyskujemy miejsce na którym ów element powinien się finalnie znaleźć. Wymagane jest operowanie na dodatkowej tablicy, do której zapisywane są elementy już posortowane. Przeciążona funkcja *rankSort()* pozwala sortowanie przeprowadzić na ograniczonym zakresie zadanej tablicy.

Implementacja algorytmu

```
void rankSort(int n, float* Array)
{
    float* localArray = copyArray(n, Array);
    for (int i = 0;i < n;i++)
    {
        int x = 0;
        for (int j = 0;j < n;j++)
        {
            if (localArray[i] > localArray[j]) x++;
        }
        Array[x] = localArray[i];
    }
    delete[] localArray;
}

void rankSort(float* Array, int left, int right)
{
    rankSort(right - left, &Array[left]);
}
```

Koncepcja równoległego wykorzystania algorytmu rank sort zakłada, że zliczanie elementów może zostać zrównoleglone. Pozwala to efektywnie wykorzystać wiele procesorów.

Implementacja algorytmu

```
void rankSortMulti(int n, float* Array)
{
    float* localArray = copyArray(n, Array);
    #pragma omp parallel for firstprivate(n) shared(localArray, Array)
    for (int i = 0;i < n;i++)
    {
        int x = 0;
        for (int j = 0;j < n;j++)
        {
            if (localArray[i] > localArray[j]) x++;
        }
        Array[x] = localArray[i];
    }
    delete[] localArray;
}
```

Sortowanie przez zliczanie

Założeniem dla tego algorytmu jest że sortowane liczby są kolejnymi liczbami naturalnymi. Dla wszystkich wartości w pierwszej kolejności tworzony jest ich histogram. Na jego podstawie wyliczana jest pozycja na której powinna znaleźć się dana wartość. Powtarzające się wartości są wstawiane na odpowiednie miejsca od prawej strony. Podobnie jak w poprzednich funkcjach stworzona jest wersja przeciążona.

Implementacja algorytmu

```
void countingSort(int n, float* Array)
{
    float* localArray = copyArray(n, Array);

    int* histogram = new int[n];
    for (int i = 0; i < n; i++) histogram[i] = 0;
    for (int i = 0; i < n; i++) histogram[(int)localArray[i]]++;
    //suma prefiksowa
    for (int i = 1; i < n; i++) histogram[i] = histogram[i] + histogram[i - 1];
    //umieszczanie elementów
    for (int i = n - 1;i >= 0;i--)
    {
        Array[histogram[(int)localArray[i]] - 1] = localArray[i];
        histogram[(int)localArray[i]]--;
    }
    delete[] localArray;
    delete[] histogram;
}

void countingSort(float* Array, int left, int right)
{
    countingSort(right - left, &Array[left]);
}
```

Równoległa implementacja umożliwia wykonanie kolejnych etapów algorytmu na wielu procesorach: zerowanie histogramu, tworzenie go, obliczanie sumy prefiksowej jak i wstawianie elementów na pozycje.

Implementacja algorytmu

```
void countingSortMulti(int n, float* Array)
{
    float* localArray = copyArray(n, Array);

    int* histogram = new int[n];
    #pragma omp parallel for firstprivate(n) shared(histogram)
    for (int i = 0; i < n; i++) histogram[i] = 0;
    #pragma omp parallel for firstprivate(n) shared(histogram, localArray)
    for (int i = 0; i < n; i++) histogram[(int)localArray[i]]++;
    //suma prefiksowa
    for (int i = 1; i < n; i++) histogram[i] = histogram[i] + histogram[i - 1];
    //umieszczanie elementów
    #pragma omp parallel for firstprivate(n) shared(histogram, localArray, Array)
    for (int i = n - 1;i >= 0;i--)
    {
        Array[histogram[(int)localArray[i]] - 1] = localArray[i];
        histogram[(int)localArray[i]]--;
    }
    delete[] localArray;
    delete[] histogram;
}
```

Zrównoleglenie względem danych

Zgodnie ze strategią „dziel i zwyciężaj” można podzielić jeden duży zbiór danych na podzbiory mniejsze, sortowane osobno. Aby uzyskać końcowy zbiór posortowany wymagana jest dodatkowa operacja połączenia posortowanych podzbiorów. W implementacji wątkom przydzielana jest odpowiednia liczba zadań – niekoniecznie musi ona być równa, ponieważ musi być liczbą całkowitą, może się więc zdarzyć, że niektóre wątki mogą mieć ją o jeden większą od reszty. Na podstawie liczby zadań wyliczane są lewe granice w ramach których wątki mają sortować tablicę. W ramach kilku wersji zaimplementowanej funkcji zastosowano trzy sekwencyjne funkcje sortujące: *rankSort*, *bubbleSort* oraz *quickSort* (miejsce zmiany w kodzie zaznaczono kolorem czerwonym) Sortowanie przez zliczanie nie mogło zostać wykorzystane, ponieważ wymaga ono ciągłego zbioru, a podział go na podzbiory tego nie zapewnia. Ponadto stworzono wersje funkcji niewykorzystujące wielu wątków, aby sprawdzić wpływ samej strategii „dziel i zwyciężaj” (miejsca w kodzie zaznaczone niebieskim).

Po posortowaniu konieczna jest agregacja podciągów. Agregacji musi się dokonać tyle ile wykorzystano procesów – 1. Algorytm agregacji jest podobny jak w sortowaniu przez scalanie: porównując kolejne elementy podciągów wybierz mniejszy i wstaw na pozycję. Ważne jest aby aktualizować tablice przechowujące liczbę elementów przypisanych do procesu. Po zakończeniu scalania jeden z nich przyjmuje wartość 0 a drugi staje się właścicielem posortowanego ciągu. W ramach kolejnych pętli scalane są pozostałe podciągi, aż zostanie tylko jeden posortowany ciąg wynikowy.

W implementacji równoległej poszczególne wątki otrzymują do sortowania pewne zakresy tablicy. Scalanie kolejnych podciągów również zachodzi równolegle, z ograniczonym dostępem do zmiennej *aggregations* określającej ile operacji scalania pozostało do zakończenia całego procesu.

Implementacja algorytmu

```
void divMultiRank(int n, float* Array)
{
    //przydzielenie liczby zadań procesom
    int p = omp_get_max_threads();
    int* nPerThread = new int[p];
    for (int i = 0; i < p;i++) nPerThread[i] = n / p;
    if (n % p != 0) for (int i = 0; i < n % p; i++) nPerThread[i]++;
    //dolne granice procesów
    int* leftThread = new int[p];
    leftThread[0] = 0;
    for (int i = 1; i < p; i++) leftThread[i] = leftThread[i - 1] + nPerThread[i - 1];
    //sortowanie podciągów
#pragma omp parallel for schedule(static, 1) firstprivate(p) shared(Array, leftThread, nPerThread)
    for (int i = 0; i < p; i++) {
        //posortuj
        rankSort(Array, leftThread[i], leftThread[i] + nPerThread[i]);
    }
    //agregacja podciągów
    //scalanie drzewiaste, dopóki nie uzyska się jednego ciągu
    int aggregations = p;
    for (int i = 2; aggregations != 1; i *= 2) {
        float* localArray = copyArray(n, Array);
#pragma omp parallel for firstprivate(i, n) shared(localArray, Array, aggregations, nPerThread,
leftThread) schedule(static, 1)
        for (int j = 0; j < p - i / 2; j += i) {
            //scal
            int ArrayLeftIndex = leftThread[j]; int ArrayLeftMax = leftThread[j] + nPerThread[j];
            int ArrayRightIndex = leftThread[j + i / 2]; int ArrayRightMax = leftThread[j + i / 2] +
nPerThread[j + i / 2];
            int MainArrayIndex = ArrayLeftIndex;
            while ((ArrayLeftIndex < ArrayLeftMax || ArrayRightIndex < ArrayRightMax) && MainArrayIndex
< n)
            {
                if (ArrayLeftIndex >= ArrayLeftMax) {
                    Array[MainArrayIndex] = localArray[ArrayRightIndex];
                    MainArrayIndex++;
                    ArrayRightIndex++;
                }
                else if (ArrayRightIndex >= ArrayRightMax) {
                    Array[MainArrayIndex] = localArray[ArrayLeftIndex];
                    MainArrayIndex++;
                    ArrayLeftIndex++;
                }
                else {
                    if (localArray[ArrayLeftIndex] <= localArray[ArrayRightIndex]) {
                        Array[MainArrayIndex] = localArray[ArrayLeftIndex];
                        MainArrayIndex++;
                        ArrayLeftIndex++;
                    }
                    else {
                        Array[MainArrayIndex] = localArray[ArrayRightIndex];
                        MainArrayIndex++;
                        ArrayRightIndex++;
                    }
                }
            }
            nPerThread[j] += nPerThread[j + i / 2];
            nPerThread[j + i / 2] = 0;
#pragma omp atomic
            aggregations--;
        }
        delete[] localArray;
    }
    delete[] nPerThread;
    delete[] leftThread;
}
```

Funkcja testująca

W celu pomiaru czasu obliczeń zaprojektowano funkcję testującą *multipleTestExecution*, która oprócz parametrów dla funkcji realizujących algorytm przyjmuje także ilość testów do wykonania oraz samą funkcję sortującą jaka ma być wykorzystana. Kolejne pomiary zapisywane są w wektorze *executeTimes*. Jednostką pomiaru czasu jest mikrosekunda. Mierzony jest tylko czas wywołania funkcji. W czasie badania wyświetlany jest progres, w postaci *wykonanoTestów/liczbaTestów*. Po zakończeniu testów wyświetlana oraz zapisywana do pliku (za pomocą strumienia *fs*) jest liczba wątków oraz pojedyncze czasy posortowane rosnąco.

Jeśli program uruchomiony jest w trybie testu to nie jest wyświetlany progres obliczeń oraz zmierzony czas, tylko wyświetlane są tablice przed i po sortowaniu, co pozwala zweryfikować poprawność działania.

```
typedef void (*sortFunctionT)(int, float*); //wskaźnik na funkcję sortującą

void multipleTestExecution(int n, float* Array, sortFunctionT sortFunction, int maxTests)
{
    //badanie czasu obliczania
    vector < long long > executeTimes;
    for (int i = 1; i <= maxTests; i++)
    {
        if (!TEST) cout << '\r' << i << '/' << maxTests;
        //kopia lokalna wygenerowanej tablicy
        float* localArray = copyArray(n, Array);
        //pomiar czasu obliczania
        chrono::steady_clock::time_point start = chrono::steady_clock::now();
        sortFunction(n, localArray);
        chrono::steady_clock::time_point end = chrono::steady_clock::now();
        //END pomiar czasu obliczania
        executeTimes.push_back(chrono::duration_cast<chrono::microseconds>(end - start).count());
        //pokaż wyniki
        if (TEST) printArrays(n, Array, localArray);
        delete[] localArray;
    }
    if (!TEST) cout << "\r                      \r";
    //wyświetlanie posortowanych czasów obliczania
    sort(executeTimes.begin(), executeTimes.end());
    if (!TEST) cout << setw(10) << left << omp_get_max_threads(); fs << setw(10) << left <<
omp_get_max_threads();
    if (!TEST) for (const auto& i : executeTimes)
    {
        cout << setw(10) << left << i; fs << setw(10) << left << i;
    }
    cout << endl; fs << endl;
    //END wyświetlanie posortowanych czasów obliczania

    //END badanie czasu obliczania
}
```

Program główny

Funkcja *main()* realizująca główne zadania korzysta z zaimplementowanych funkcji. Na początku ustawiany jest punkt czasu pozwalający mierzyć czas całego badania. Następnie ustawiane są opcje językowe oraz ustalane jest ziarno generatora liczb pseudolosowych na podstawie czasu. Otwierany jest także plik do którego zapisywane są wyniki. Dalej ustawiane są parametry programu: liczba testów do wykonania na jednym algorytmie na jednej wielkości tablicy, liczby wątków dla których mają zostać uruchomione algorytmy, wektory zawierające funkcje sortujące i ich nazwy, wielkości sortowanych tablic.

Dla każdego rozmiaru tworzone i generowane są nowe macierze. Dla każdego algorytmu i dla wszystkich liczb wątków uruchamiana jest funkcja badająca czasy. Funkcje jednowątkowe są badane tylko dla jednego wątku. Po zakończeniu badania dynamiczne tablice są usuwane. Na zakończenie programu wyświetlany jest sumaryczny czas jego działania.

Implementacja

```
int main() {
    chrono::steady_clock::time_point start = chrono::steady_clock::now();
    setlocale(LC_CTYPE, "Polish");
    omp_set_dynamic(0);
    srand(time(NULL));
    ostringstream filenameStream;
    filenameStream << "wyniki\\out" << time(NULL) << ".txt";
    fs.open(filenameStream.str(), fstream::in | fstream::out | fstream::trunc);
    //ZMIENNE OGÓLNE
    //wektor wątków
    vector < int > threads;
    //liczba testów
    int maxTests = 10;
    if (TEST) maxTests = 1;
    //wektory funkcji
    vector < sortFunctionT > sortFunctions;
    vector < string > sortFunctionsNames;
    //wektor wielkości
    vector < int > n;
    //END ZMIENNE OGÓLNE
    //SORTOWANIE PERMUTACJI, WSZYSTKIE ALGORYTMY
    threads.clear();
    for (int i = 1; i <= 12; i++) threads.push_back(i);     //i <= 50 dla divFunkcji
    //wektory funkcji
    sortFunctions.clear();
    sortFunctionsNames.clear();
    { sortFunctions.push_back(rankSortMulti); sortFunctionsNames.push_back("rankSortMulti"); }
    { sortFunctions.push_back(countingSortMulti); sortFunctionsNames.push_back("countingSortMulti"); }
    { sortFunctions.push_back(bubbleSort); sortFunctionsNames.push_back("bubbleSort"); }
    { sortFunctions.push_back(pipeBubbleSort); sortFunctionsNames.push_back("pipeBubbleSort"); }
    { sortFunctions.push_back(oddEvenSort); sortFunctionsNames.push_back("oddEvenSort"); }
    { sortFunctions.push_back(quickSort); sortFunctionsNames.push_back("quickSort"); }
    { sortFunctions.push_back(divSingleRank); sortFunctionsNames.push_back("divSingleRank"); }
    { sortFunctions.push_back(divSingleBubble); sortFunctionsNames.push_back("divSingleBubble"); }
    { sortFunctions.push_back(divSingleQuick); sortFunctionsNames.push_back("divSingleQuick"); }
    { sortFunctions.push_back(divMultiRank); sortFunctionsNames.push_back("divMultiRank"); }
    { sortFunctions.push_back(divMultiBubble); sortFunctionsNames.push_back("divMultiBubble"); }
    { sortFunctions.push_back(divMultiQuick); sortFunctionsNames.push_back("divMultiQuick"); }
    //wektor wielkości
    n.clear();
    n = { 1250, 2500, 5000, 10000 };    //małe wartości, dla najsłabszych algorytmów
    if (TEST) n = { 10 };
    //dla każdej wielkości danych
    for (int i = 0; i < n.size(); i++) {
        cout << n[i] << endl; fs << n[i] << endl;
        float* Array = new float[n[i]];
        generatePermutationArray(n[i], Array);
        //dla każdego algorytmu
        for (int j = 0; j < sortFunctions.size(); j++) {
            cout << sortFunctionsNames[j] << endl; fs << sortFunctionsNames[j] << endl;
            //dla każdej liczby wątków
            for (int k = 0; k < threads.size(); k++) {
                omp_set_num_threads(threads[k]);
                multipleTestExecution(n[i], Array, sortFunctions[j], maxTests);
                //funkcje jednowątkowe wykonaj tylko na jednym wątku
                if (sortFunctionsNames[j] == "bubbleSort") break;
                if (sortFunctionsNames[j] == "quickSort") break;
            }
        }
        delete[] Array;
    }
    //END SORTOWANIE PERMUTACJI, WSZYSTKIE ALGORYTMY
    ////////////////////////////////////////////////////////////////////////////////

    fs.close();
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    int min = chrono::duration_cast<chrono::minutes>(end - start).count();
    int sec = chrono::duration_cast<chrono::seconds>(end - start).count()%60;
    int milisec = chrono::duration_cast<chrono::milliseconds>(end - start).count()%1000;
    cout << "Zakończono: " << min << "m " << sec << "s " << milisec << "ms " << endl;
    getchar();
}
```

Pomiary

Program został stworzony i skompilowany w programie Visual Studio 2019. Uruchomiony został na urządzeniu o parametrach określonych w tabeli:

LENOVO	Lenovo B50-80 Procesor Intel(R) Core(TM) i3-5020U CPU @ 2.20GHz, 2200 MHz, Rdzenie: 2, Procesory logiczne: 4 Nazwa systemu operacyjnego Microsoft Windows 10 Education

Wszystkie otrzymane wartości pozostawiono w arkuszu kalkulacyjnym dołączonym do sprawozdania.

Wśród otrzymanych wyników zdarza się, że najdłuższe czasy wykonania odstają od średniej wartości. Odrzucenie 20% najdłuższych czasów poprawia stabilność otrzymanych wyników, są one bardziej spójne. W jednym przypadku pozwoliło to zredukować średnie odchylenie aż 660 razy, a w innym średnią wartość 20 razy. Do dalszych obliczeń brano więc pod uwagę tylko 8 z 10 uzyskanych wyników.

Wszystkie czasy w tabelach pozostawiono w mikrosekundach, aby nie umniejszać różnicom czasowym jakie między nimi zachodzą. Wymagało to zmniejszenia rozmiaru fontu, a co za tym idzie nieco ucierpiała czytelność tabel.

Strony przedstawiające wyniki zostały przedzielone stronami zawierającymi ich analizę, aby ograniczyć konieczność wertowania sprawozdania w celu uzyskania powiązania między analizą słowną a graficzną.

Analiza wyników

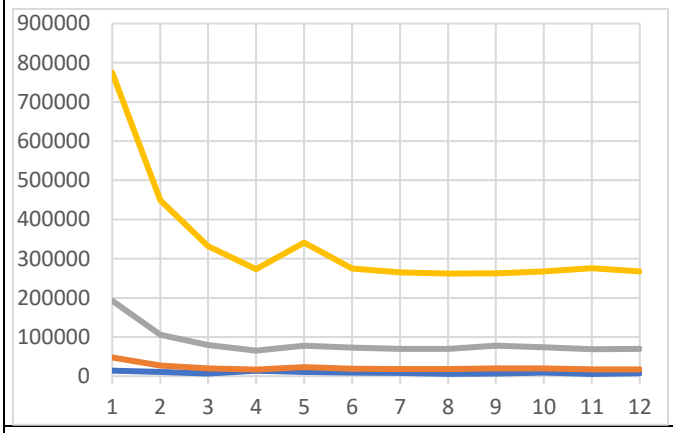
Średni czas wykonania
Oś pozioma: liczba wątków 1-12
Oś pionowa: średni czas w µs

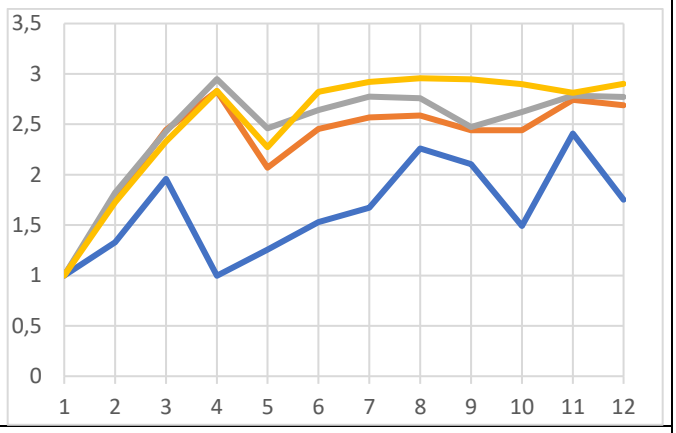
Rozmiar macierzy
10000
5000
2500
1250

Przyspieszenie
Oś pozioma: liczba wątków 1-12
Oś pionowa: przyspieszenie

rankSortMulti

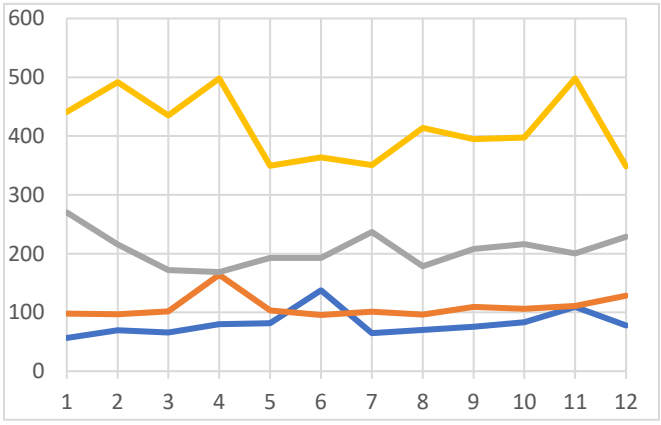
Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	774595,13	448730,88	332273,00	273549,50	340698,88	274571,75	265184,13	261945,88	262911,00	267195,38	275348,75	267064,25
Przyspieszenie	1,00	1,73	2,33	2,83	2,27	2,82	2,92	2,96	2,95	2,90	2,81	2,90
Średni czas	192745,50	105776,50	79354,38	65375,25	78316,50	72990,50	69462,00	69879,13	77959,13	73473,88	69222,75	69522,38
Przyspieszenie	1,00	1,82	2,43	2,95	2,46	2,64	2,77	2,76	2,47	2,62	2,78	2,77
Średni czas	47931,88	27045,25	19611,75	16943,25	23155,75	19525,75	18658,50	18521,38	19625,25	19624,75	17473,63	17825,75
Przyspieszenie	1,00	1,77	2,44	2,83	2,07	2,45	2,57	2,59	2,44	2,44	2,74	2,69
Średni czas	14059,38	10582,50	7174,25	14100,63	11198,38	9198,38	8405,75	6222,63	6676,13	9437,00	5838,13	8030,25
Przyspieszenie	1,00	1,33	1,96	1,00	1,26	1,53	1,67	2,26	2,11	1,49	2,41	1,75

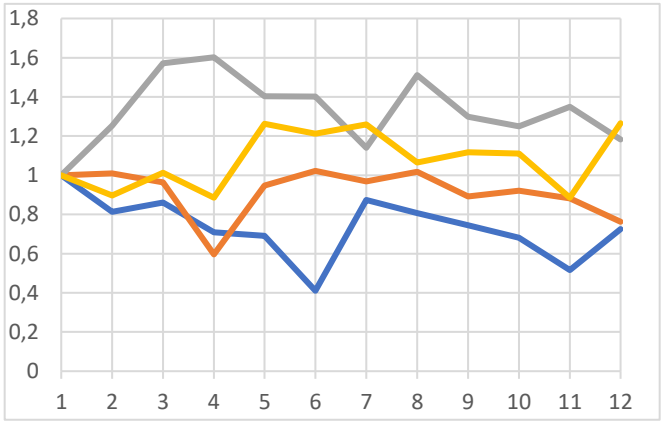




countingSortMulti

Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	441,00	491,75	435,25	498,00	349,38	363,75	350,38	413,88	394,88	397,25	498,00	348,50
Przyspieszenie	1,00	0,90	1,01	0,89	1,26	1,21	1,26	1,07	1,12	1,11	0,89	1,27
Średni czas	270,13	215,50	171,88	168,63	192,50	192,75	237,00	178,75	208,00	216,13	200,25	228,50
Przyspieszenie	1,00	1,25	1,57	1,60	1,40	1,40	1,14	1,51	1,30	1,25	1,35	1,18
Średni czas	97,75	96,88	101,50	164,00	103,13	95,63	101,00	96,13	109,63	106,13	110,75	128,25
Przyspieszenie	1,00	1,01	0,96	0,60	0,95	1,02	0,97	1,02	0,89	0,92	0,88	0,76
Średni czas	56,50	69,50	65,63	79,75	81,88	137,63	64,63	70,00	75,88	83,00	109,38	78,00
Przyspieszenie	1,00	0,81	0,86	0,71	0,69	0,41	0,87	0,81	0,74	0,68	0,52	0,72





Algorytm *Rank sort* pozwala bardzo dobrze wykorzystać wiele wątków. Dla większych zbiorów wykresy przyspieszenia niemal się nakładają, co pokazuje jego niezależność od wielkości sortowanej tablicy. Maksymalne wartości uzyskuje dla liczby wątków równej liczbie rdzeni. Względem innych algorytmów sortowania algorytm nie uzyskuje dobrych wyników. Jego czasy są jednymi z najdłuższych uzyskanych w trakcie badania.

Count sort jest jednym z najszybszych algorytmów sortowania. W jego implementacji nie ma zagnieżdżonych pętel co przekłada się na niską złożonością obliczeniową. Jest to jednak algorytm o dość ograniczonych możliwościach, ponieważ pozwala sortować tylko liczby naturalne z konkretnego zakresu. Wykorzystanie wielowątkowości w jego przypadku niekoniecznie musi skutkować poprawą wyników. W przypadku najmniejszej tablicy dla sześciu wątków przyspieszenie osiąga wartość zaledwie 0,4, co jest znaczącym regresem w stosunku do czasu wyjściowego. Maksymalne przyspieszenie wynosi z kolei 1,6 dla trzech i czterech wątków dla tablicy zawierającej 5000 elementów. Ogólnie wykresy przyspieszenia są bardzo niestabilne, a czasy utrzymują stosunkowo stały poziom.

Sekwencyjny algorytm *sortowania bąbelkowego* jest bardzo wrażliwy na zwiększanie się zbioru sortowanego. Wynika to z jego złożoności $O(n^2)$. Podwojenie rozmiaru tablicy sortującej wydłuża około czterokrotnie czas sortowania.

Wykorzystanie równoległości w funkcji *pipeBubbleSort()* pozwala przyspieszyć algorytm sortowania bąbelkowego około dwukrotnie. Najlepsze rezultaty i stabilność wykresu uzyskuje się dla największych rozmiarów tablic. Dla małej tablicy większa ilość wątków sprawia, że przyspieszenie oscyluje wokół wartości 1. Finalnie algorytm ten uzyskuje czasy na poziomie *Rank sort*, ale nie jest ograniczony do zbioru niepowtarzających się wartości.

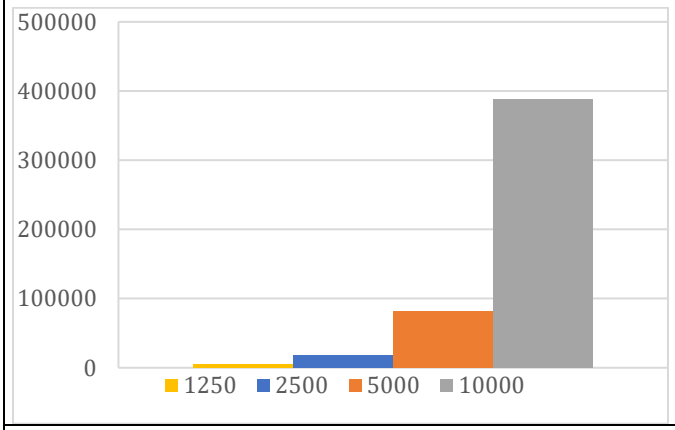
Średni czas wykonania
Oś pozioma: liczba wątków 1-12
Oś pionowa: średni czas w µs

Rozmiar macierzy
10000
5000
2500
1250

Przyspieszenie
Oś pozioma: liczba wątków 1-12
Oś pionowa: przyspieszenie

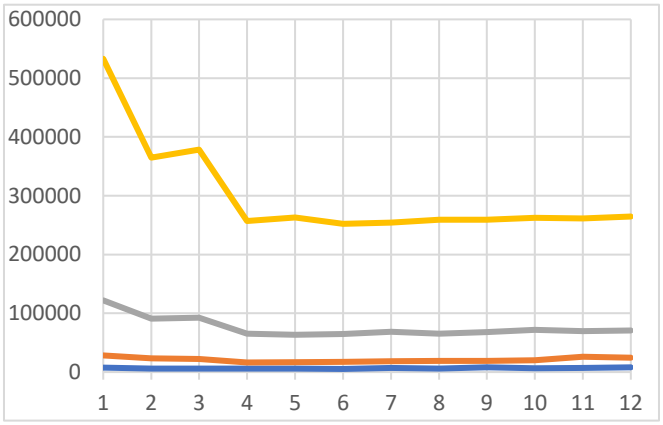
bubbleSort

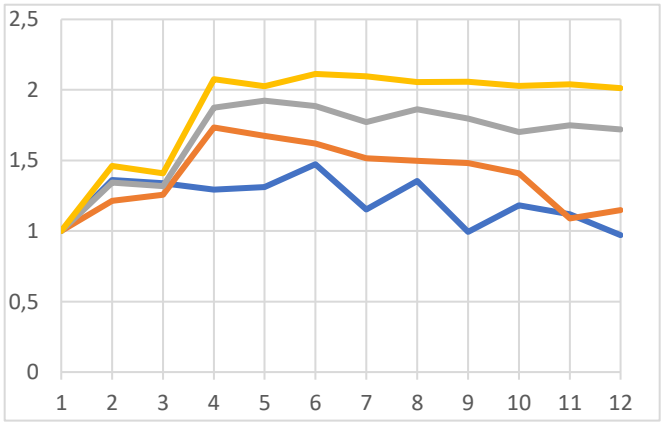
Wątków	1
Średni czas	387952,25
Przyspieszenie	
Średni czas	81673,13
Przyspieszenie	
Średni czas	18802,63
Przyspieszenie	
Średni czas	5298,38
Przyspieszenie	



pipeBubbleSort

Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	532952,88	364994,88	378464,88	256789,88	263153,50	252192,00	254183,13	259244,00	259041,00	262623,75	261238,63	264846,13
Przyspieszenie	1,00	1,46	1,41	2,08	2,03	2,11	2,10	2,06	2,06	2,03	2,04	2,01
Średni czas	121990,63	90891,25	92561,25	65080,63	63413,88	64673,25	68819,50	65458,75	67851,38	71645,25	69781,75	70975,38
Przyspieszenie	1,00	1,34	1,32	1,87	1,92	1,89	1,77	1,86	1,80	1,70	1,75	1,72
Średni czas	28296,38	23299,75	22501,75	16320,38	16903,38	17464,00	18661,50	18891,38	19100,63	20084,38	25978,75	24657,00
Przyspieszenie	1,00	1,21	1,26	1,73	1,67	1,62	1,52	1,50	1,48	1,41	1,09	1,15
Średni czas	7824,88	5749,00	5843,63	6051,50	5967,63	5312,25	6785,25	5777,25	7883,13	6624,13	6998,50	8062,50
Przyspieszenie	1,00	1,36	1,34	1,29	1,31	1,47	1,15	1,35	0,99	1,18	1,12	0,97





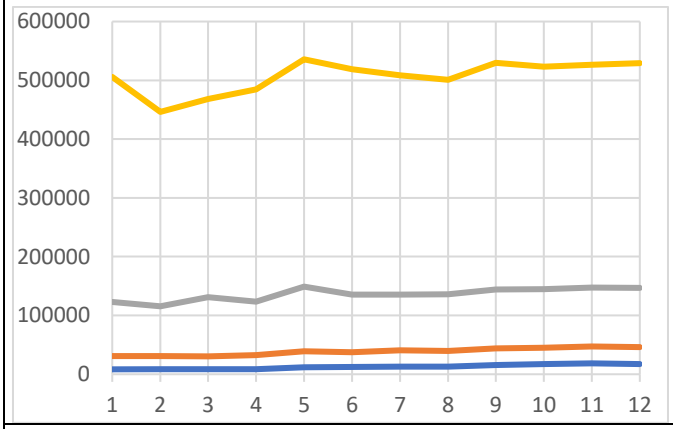
Średni czas wykonania
Oś pozioma: liczba wątków 1-12
Oś pionowa: średni czas w µs

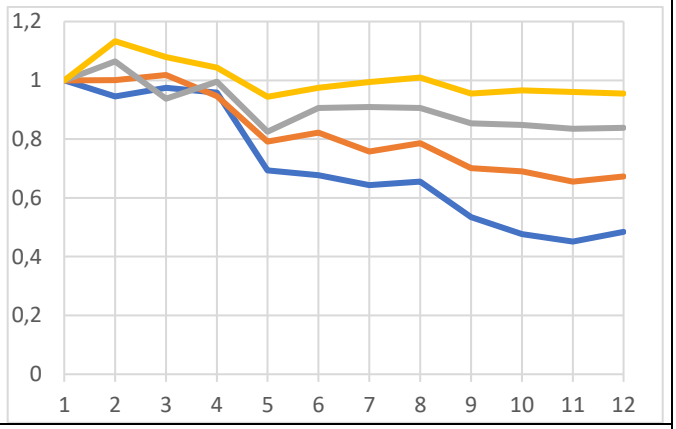
Rozmiar macierzy
10000
5000
2500
1250

Przyspieszenie
Oś pozioma: liczba wątków 1-12
Oś pionowa: przyspieszenie

oddEvenSort

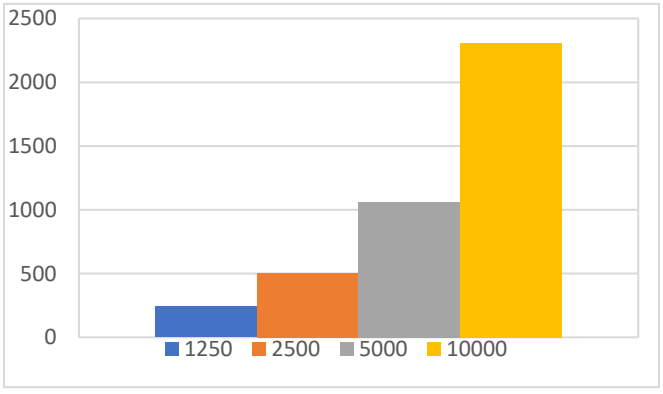
Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	505651,88	446292,38	468433,13	484800,38	535711,63	518710,75	508501,75	500993,75	529487,88	523461,25	526403,63	529255,13
Przyspieszenie	1,00	1,13	1,08	1,04	0,94	0,97	0,99	1,01	0,95	0,97	0,96	0,96
Średni czas	122989,38	115562,38	131155,75	123566,88	149037,50	135725,25	135307,50	135738,38	144097,63	144927,50	147335,00	146647,13
Przyspieszenie	1,00	1,06	0,94	1,00	0,83	0,91	0,91	0,91	0,85	0,85	0,83	0,84
Średni czas	30938,25	30923,88	30388,75	32685,75	39069,25	37621,13	40812,50	39356,75	44117,25	44843,63	47211,38	45959,38
Przyspieszenie	1,00	1,00	1,02	0,95	0,79	0,82	0,76	0,79	0,70	0,69	0,66	0,67
Średni czas	8342,75	8832,00	8556,25	8706,63	12037,63	12323,63	12975,13	12737,88	15603,88	17511,13	18483,25	17236,75
Przyspieszenie	1,00	0,94	0,98	0,96	0,69	0,68	0,64	0,65	0,53	0,48	0,45	0,48





quickSort

Wątków	1
Średni czas	2308,75
Przyspieszenie	
Średni czas	1059,75
Przyspieszenie	
Średni czas	505,38
Przyspieszenie	
Średni czas	240,88
Przyspieszenie	



Kolejna wariacja na temat sortowania bąbelkowego, czyli *sortowanie nieparzyste-parzyste* może nieznacznie poprawić czasy wykonania. Zachodzi to jednak tylko dla niewielkiej liczby wykorzystywanych wątków. Powyżej czterech wątków następuje w zasadzie tylko pogarszanie uzyskiwanych wyników, ewentualnie utrzymywanie w miarę stałego poziomu. Algorytm ten posiada także wadę jaką posiadało sortowanie bąbelkowe. Dwukrotne zwiększenie sortowanego zbioru wydłuża czas sortowania cztery razy. Otrzymane czasy są bodaj najgorsze pośród wszystkich badanych algorytmów.

Sortowanie szybkie jest jednym z najlepszych algorytmów. Nie posiada ograniczeń co do sortowanego zbioru oraz jest szybki, co wynika z zastosowania filozofii „dziel i rządź". Nieco problemów stwarza zastosowana w nim rekurencja. Czasy uzyskiwane przez Quick sort należą do najlepszych oraz cechują się liniową zależnością względem wielkości sortowanego zbioru.

Wynikający z strategii „dziel i rządź" podział dużego zbioru sortowanego zbioru na posortowane podciągi a następnie ich agregacja daje bardzo dobre rezultaty. Dla dzielonego Rank sort *divSingleRank* następuje wręcz superliniowe przyspieszenie nawet bez wykorzystania obliczeń wielowątkowych. Przyspieszenie nie jest zależne od wielkości sortowanego zbioru.

Podobne wyniki uzyskano dla dzielonego sortowania bąbelkowego *divSingleBubble*. W tym wypadku uwidoczniła się jednak niedogodność związana z kwadratową zależnością czasu sortowania od wielkości tablicy. Ogólna tendencja pokazuje, że im większy zbiór tym lepsze przyspieszenie. Zwiększanie liczby podciągów wpływa liniowo na poprawę przyspieszenia. Dla największego zbioru uzyskane przyspieszenie jest lepsze od algorytmu *divSingleRank*.

Średni czas wykonania
Oś pozioma: liczba wątków 1-12
Oś pionowa: średni czas w µs

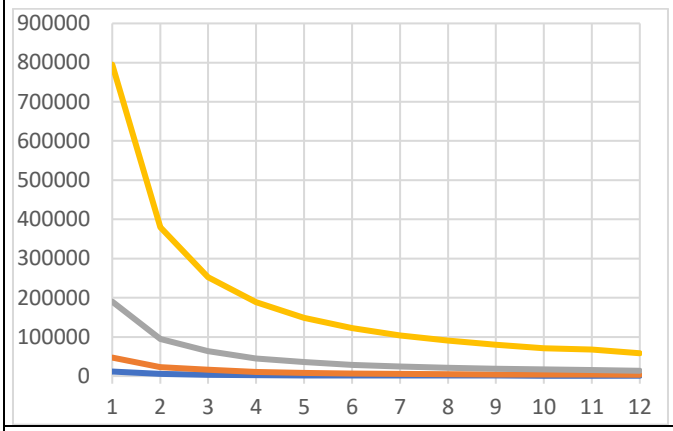
Rozmiar macierzy
10000
5000
2500
1250

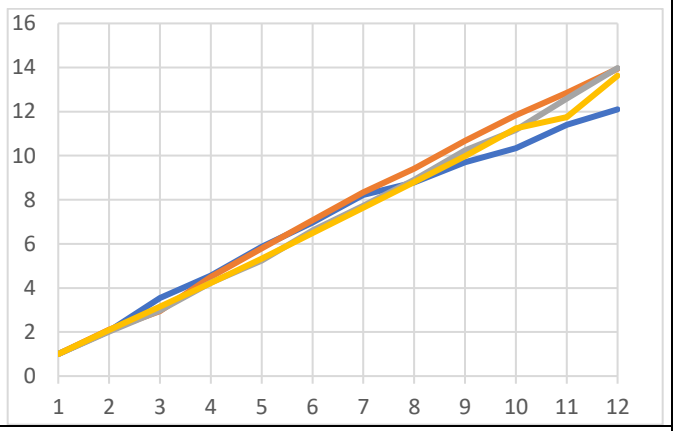
Przyspieszenie
Oś pozioma: liczba wątków 1-12
Oś pionowa: przyspieszenie

divSingleRank

uwaga: liczba wątków określa tylko na ile podciągów został podzielony zbiór, nie były one wykorzystywane do obliczeń

Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	794954,38	379727,13	252498,25	188324,38	149010,25	122344,75	104004,25	90267,00	79732,88	70716,63	67659,75	58328,38
Przyspieszenie	1,00	2,09	3,15	4,22	5,33	6,50	7,64	8,81	9,97	11,24	11,75	13,63
Średni czas	190001,50	94341,38	63961,75	44738,63	36212,63	28779,25	24585,63	21336,63	18550,50	17005,88	15085,13	13596,38
Przyspieszenie	1,00	2,01	2,97	4,25	5,25	6,60	7,73	8,90	10,24	11,17	12,60	13,97
Średni czas	47244,13	22494,00	16030,63	10501,50	8144,50	6678,63	5667,63	5024,50	4426,13	3991,63	3677,00	3388,00
Przyspieszenie	1,00	2,10	2,95	4,50	5,80	7,07	8,34	9,40	10,67	11,84	12,85	13,94
Średni czas	11584,25	5653,88	3276,13	2538,13	1976,38	1661,38	1409,63	1319,25	1193,75	1121,13	1015,75	957,38
Przyspieszenie	1,00	2,05	3,54	4,56	5,86	6,97	8,22	8,78	9,70	10,33	11,40	12,10

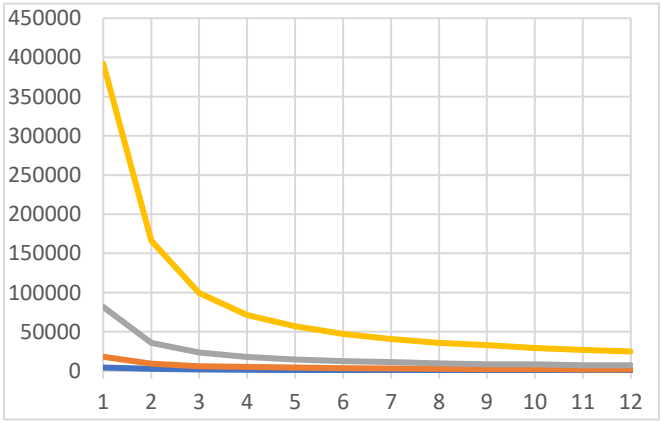


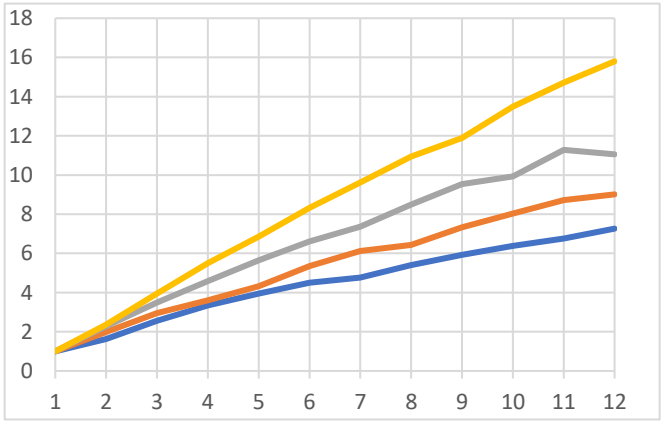


divSingleBubble

uwaga: liczba wątków określa tylko na ile podciągów został podzielony zbiór, nie były one wykorzystywane do obliczeń

Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	392106,38	166091,25	99437,25	71301,63	57229,75	47059,75	40794,75	35828,13	32967,38	29075,13	26653,38	24816,38
Przyspieszenie	1,00	2,36	3,94	5,50	6,85	8,33	9,61	10,94	11,89	13,49	14,71	15,80
Średni czas	81638,13	35751,75	23402,50	17812,00	14471,00	12334,38	11096,38	9623,88	8565,25	8225,88	7236,13	7381,38
Przyspieszenie	1,00	2,28	3,49	4,58	5,64	6,62	7,36	8,48	9,53	9,92	11,28	11,06
Średni czas	18079,50	9120,63	6142,25	5020,25	4178,50	3375,63	2951,75	2808,88	2465,75	2250,00	2072,13	2006,25
Przyspieszenie	1,00	1,98	2,94	3,60	4,33	5,36	6,13	6,44	7,33	8,04	8,73	9,01
Średni czas	4331,38	2649,00	1687,88	1295,00	1095,38	961,75	910,13	803,00	730,88	678,38	640,38	596,38
Przyspieszenie	1,00	1,64	2,57	3,34	3,95	4,50	4,76	5,39	5,93	6,38	6,76	7,26





Średni czas wykonania
Oś pozioma: liczba wątków 1-12
Oś pionowa: średni czas w µs

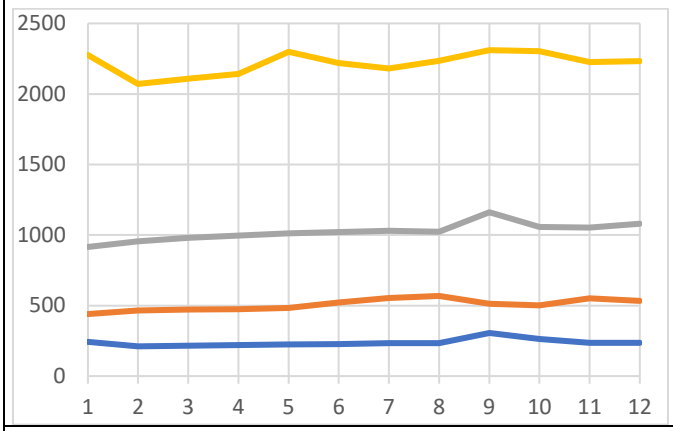
Rozmiar macierzy
10000
5000
2500
1250

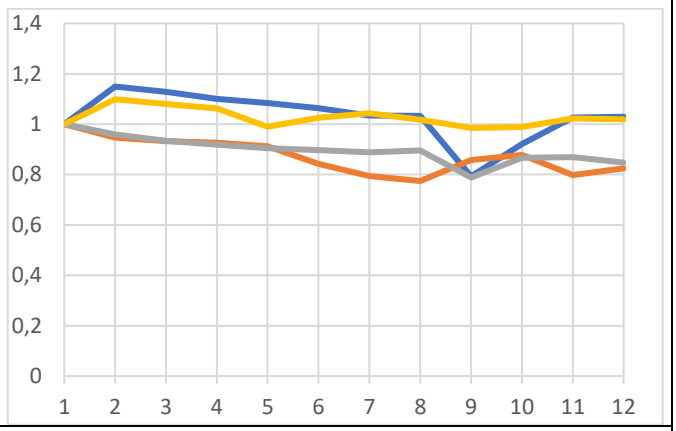
Przyspieszenie
Oś pozioma: liczba wątków 1-12
Oś pionowa: przyspieszenie

divSingleQuick

uwaga: liczba wątków określa tylko na ile podciągów został podzielony zbiór, nie były one wykorzystywane do obliczeń

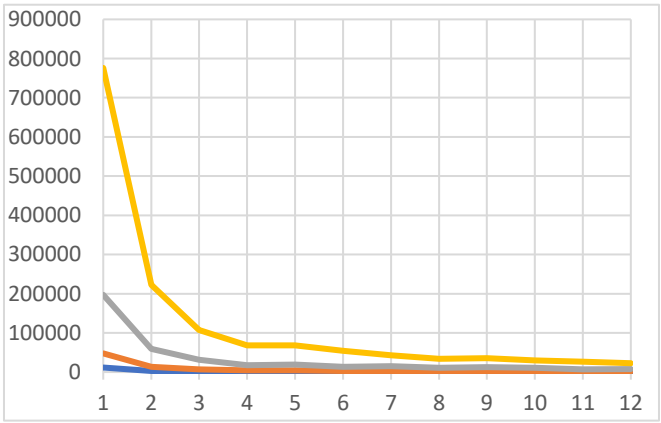
Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	2276,00	2071,00	2107,25	2141,63	2298,88	2218,25	2181,13	2235,88	2310,50	2302,75	2226,38	2233,50
Przyspieszenie	1,00	1,10	1,08	1,06	0,99	1,03	1,04	1,02	0,99	0,99	1,02	1,02
Średni czas	915,50	954,50	980,13	996,75	1011,38	1021,25	1030,13	1022,63	1161,50	1056,50	1052,88	1079,63
Przyspieszenie	1,00	0,96	0,93	0,92	0,91	0,90	0,89	0,90	0,79	0,87	0,87	0,85
Średni czas	440,00	464,88	471,63	475,00	482,25	522,38	553,75	568,13	513,38	501,25	551,38	533,88
Przyspieszenie	1,00	0,95	0,93	0,93	0,91	0,84	0,79	0,77	0,86	0,88	0,80	0,82
Średni czas	242,38	210,88	214,88	220,38	223,75	227,88	234,38	234,63	305,63	263,25	236,13	235,50
Przyspieszenie	1,00	1,15	1,13	1,10	1,08	1,06	1,03	1,03	0,79	0,92	1,03	1,03

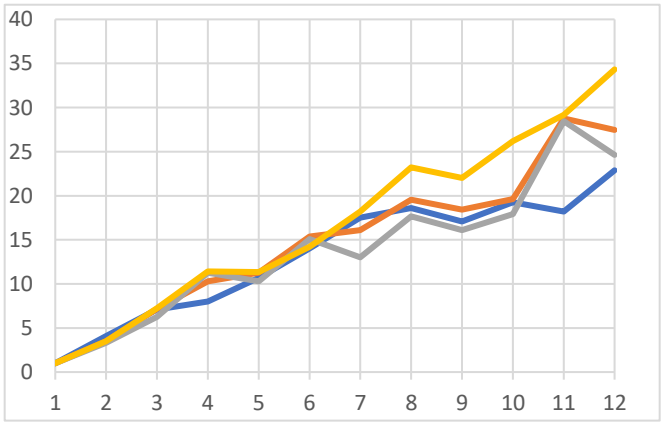




divMultiRank

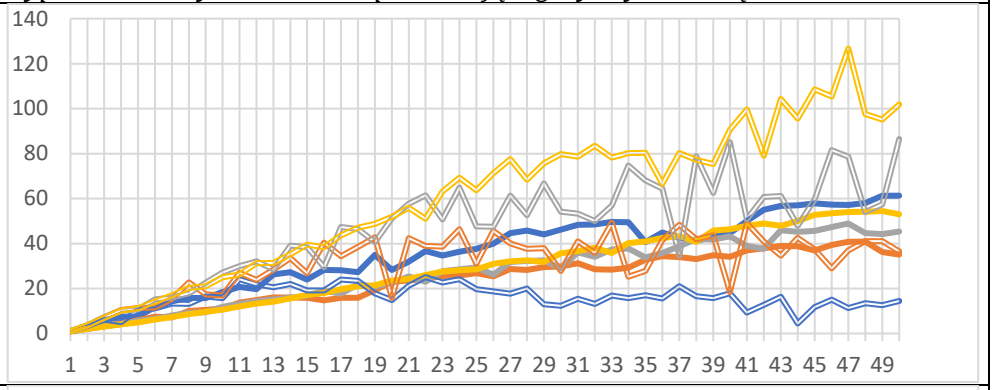
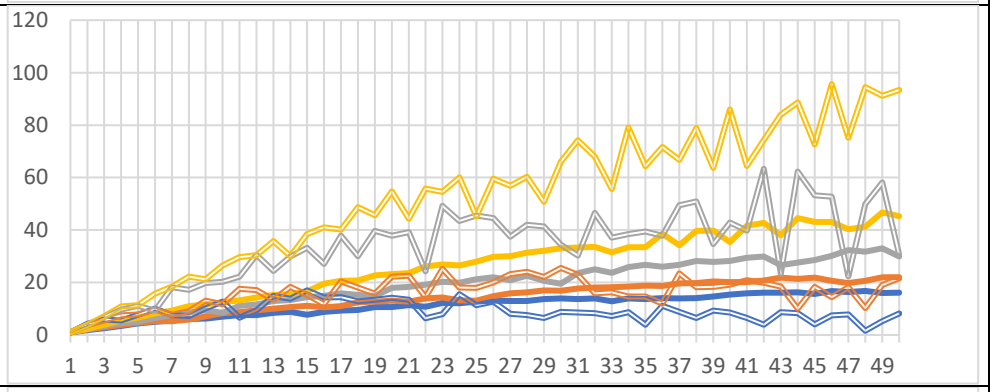
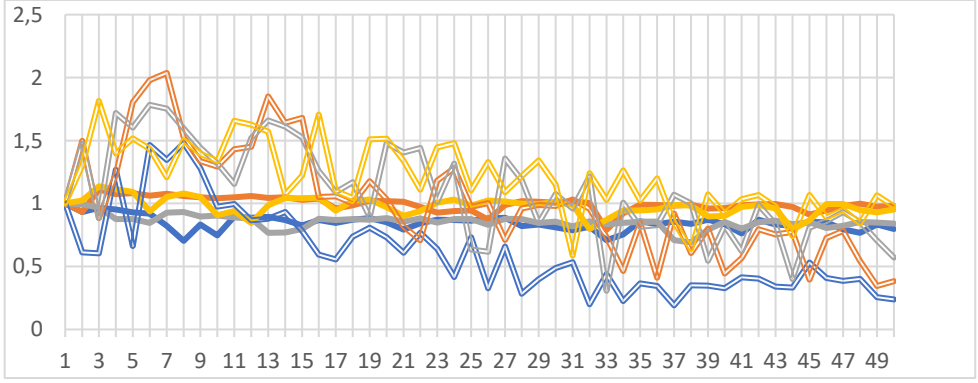
Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	776113,75	222626,13	107132,13	68050,25	68465,50	54627,50	42622,38	33438,50	35227,50	29639,63	26614,63	22612,00
Przyspieszenie	1,00	3,49	7,24	11,41	11,34	14,21	18,21	23,21	22,03	26,19	29,16	34,32
Średni czas	197058,50	59159,50	31488,13	17505,63	19177,38	13065,88	15158,75	11156,63	12228,38	10993,63	6924,38	8003,88
Przyspieszenie	1,00	3,33	6,26	11,26	10,28	15,08	13,00	17,66	16,11	17,92	28,46	24,62
Średni czas	47665,63	13672,63	6654,88	4638,63	4235,38	3103,38	2962,88	2440,75	2589,25	2427,88	1657,13	1735,50
Przyspieszenie	1,00	3,49	7,16	10,28	11,25	15,36	16,09	19,53	18,41	19,63	28,76	27,47
Średni czas	11547,00	2820,63	1629,38	1443,63	1080,88	822,50	659,38	620,38	675,63	599,88	634,25	504,63
Przyspieszenie	1,00	4,09	7,09	8,00	10,68	14,04	17,51	18,61	17,09	19,25	18,21	22,88





Podział na podciągi nie ma większego wpływu na algorytm *Quick sort* który już w swej idei przewiduje dzielenie danego zbioru na część mniejszą i większą od zadanej „osi”. Poprawa wyników zaszła wyłącznie dla najmniejszej i największej tablicy. Uzyskane czasy są najwyżej o kilkanaście procent lepsze od czasu wyjściowego. Zwiększanie liczby podciągów zwiększa koszt obliczeniowy operacji scalania, co powoduje pogorszenie kolejno uzyskiwanych czasów. Relatywnie jest to nadal algorytm bardzo szybki oraz zachowujący swoje właściwości.

Wykorzystanie wielowątkowości w czasie dzielenia i scalania tablicy pozwala jeszcze bardziej przyspieszyć badane algorytmy. Aby zbadać wpływ obliczeń równoległych na już podzielone zbiory przeprowadzono dodatkowe badanie na liczbie wątków sięgającej 50. Poniżej przedstawiono uzyskane wykresy:

linie wypełnione białym kolorem reprezentują algorytmy wielowątkowe	
Rank sort	
Bubble sort	
Quick sort	

Zauważono, że algorytmy wielowątkowe są dużo bardziej podatne na zakłócenia, ich wykresy są mniej stabilne. Ogólnie odwzorowują jednak tendencje występujące w jednowątkowych wersjach algorytmów. Dla małych tablic widoczne jest szybkie pogorszenie przyspieszenia, wynikające z narzutów związanych z scalaniem zbioru, które przewyższają korzyści płynące z sortowania mniejszych podciągów. Wyciągnięto z tego wniosek, że korzystne jest zastosowanie takich algorytmów tylko dla większych zbiorów sortowanych. Najlepsze przyspieszenie superliniowe wynoszące ok. 126 uzyskano dla 47 wątków w algorytmie *Rank sort*.

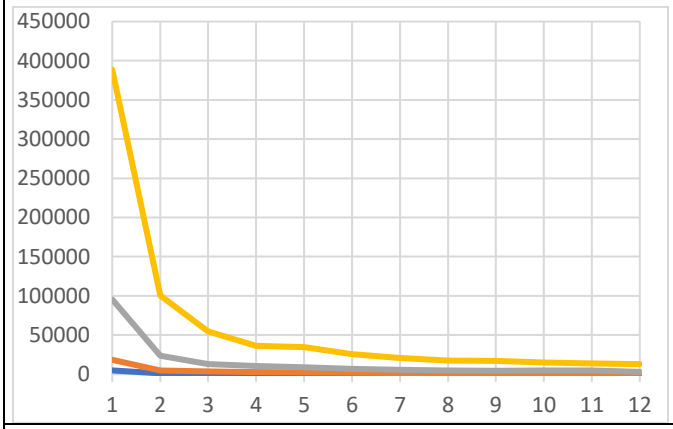
Średni czas wykonania
Oś pozioma: liczba wątków 1-12
Oś pionowa: średni czas w µs

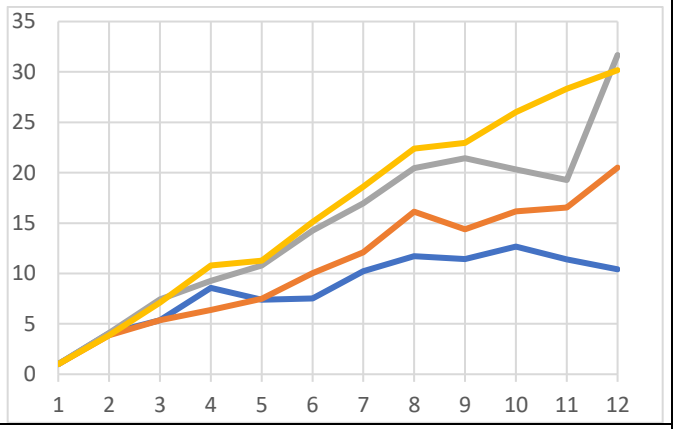
Rozmiar macierzy
10000
5000
2500
1250

Przyspieszenie
Oś pozioma: liczba wątków 1-12
Oś pionowa: przyspieszenie

divMultiBubble

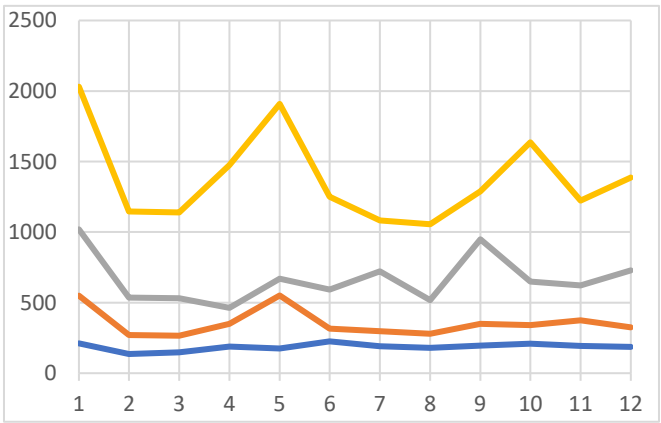
Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	388547,25	100494,38	54763,00	36034,88	34480,50	25721,63	20887,38	17350,63	16923,25	14936,38	13721,38	12877,38
Przyspieszenie	1,00	3,87	7,10	10,78	11,27	15,11	18,60	22,39	22,96	26,01	28,32	30,17
Średni czas	95408,13	23381,25	12825,75	10307,25	8852,25	6694,00	5630,00	4663,63	4451,75	4692,63	4953,63	3012,00
Przyspieszenie	1,00	4,08	7,44	9,26	10,78	14,25	16,95	20,46	21,43	20,33	19,26	31,68
Średni czas	18361,13	4754,50	3420,63	2872,38	2447,13	1831,50	1519,38	1137,50	1275,75	1135,38	1110,00	895,38
Przyspieszenie	1,00	3,86	5,37	6,39	7,50	10,03	12,08	16,14	14,39	16,17	16,54	20,51
Średni czas	4851,00	1191,75	906,00	566,50	655,50	645,75	474,38	414,13	424,88	383,00	426,25	465,50
Przyspieszenie	1,00	4,07	5,35	8,56	7,40	7,51	10,23	11,71	11,42	12,67	11,38	10,42

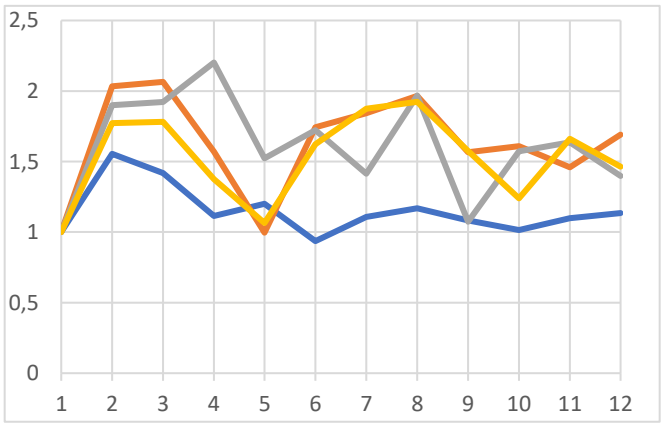




divMultiQuick

Wątków	1	2	3	4	5	6	7	8	9	10	11	12
Średni czas	2030,75	1146,00	1140,38	1475,38	1908,50	1251,63	1082,88	1055,75	1289,38	1637,50	1222,75	1386,88
Przyspieszenie	1,00	1,77	1,78	1,38	1,06	1,62	1,88	1,92	1,57	1,24	1,66	1,46
Średni czas	1019,50	537,00	530,38	463,00	669,75	591,63	721,13	518,13	949,00	648,25	623,13	729,63
Przyspieszenie	1,00	1,90	1,92	2,20	1,52	1,72	1,41	1,97	1,07	1,57	1,64	1,40
Średni czas	548,75	269,75	265,75	349,50	550,88	314,75	297,88	279,13	350,50	341,25	375,75	324,50
Przyspieszenie	1,00	2,03	2,06	1,57	1,00	1,74	1,84	1,97	1,57	1,61	1,46	1,69
Średni czas	211,25	135,88	148,88	189,38	175,88	225,63	190,88	180,75	195,25	208,13	192,25	186,25
Przyspieszenie	1,00	1,55	1,42	1,12	1,20	0,94	1,11	1,17	1,08	1,02	1,10	1,13





Mnogość algorytmów sortowania oraz możliwości ich równoległych implementacji powoduje, że zagadnienie to jest niezwykle obszerne. W badaniu ujęto jedynie kilka z nich, a i tak uzyskano bardzo interesujące wyniki. W powszechnym zastosowaniu prawdopodobnie najlepiej zachowuje się algorytm *sortowania szybkiego*. Jego prostota implementacji jak i skuteczność idei powodują, że może być zastosowany prawie zawsze. Nie działa on jednak w miejscu oraz nie zachowuje kolejności elementów o tych samych wartościach, więc jego działanie może być niepożądane.

W specyficznych zastosowaniach wykorzystanie sortowania równoległego może dać dużo lepsze rezultaty od sortowania sekwencyjnego. Może mieć to znaczenie dla bardzo dużych zbiorów danych, które powinny być posortowane jak najszybciej.

Wykorzystanie doświadczeń nabytych w trakcie zrównoleglania algorytmów sortowania można przenieść także na podobne w implementacji algorytmy, niekoniecznie sortujące. Nie w każdej dziedzinie da się wykoncypować algorytm odpowiadający *sortowaniu szybkiemu*.

Odrzucenie 20% najdłuższych czasów wykonania pozwala uzyskać bardziej spójne wyniki. Wygenerowane wykresy cechują się mniejszą liczbą wahań, co może świadczyć, że wartości najdłuższe mogły zostać uzyskane w wyniku zakłóceń działania programu. Pokazuje to wrażliwość obliczeń na zakłócenia pracy.

Otrzymane wyniki mogłyby być bardziej reprezentatywne, gdyby program został uruchomiony w lepiej przygotowanym środowisku. Odciążenie systemu poprzez wyłączenie wszystkich niepotrzebnych programów i funkcji pozostawiłoby procesor niemal w całości do dyspozycji badanego zadania. Tak otrzymane wartości mogłyby być bardziej zbliżone do teoretycznych.

Wyniki

- wyniki uzyskane w badaniu 1-12 wątków: 

- wyniki uzyskane w dodatkowym badaniu 1-50 wątków: