POLITECHNIKA OPOLSKA	**Politechnika Opolska** **Wydział Elektrotechniki, Automatyki i Informatyki** **Katedra Informatyki**
Rok akademicki	2023/2024
Przedmiot	Programowanie współbieżne i rozproszone
Forma zajęć	Wykład
Prowadzący zajęcia	dr inż. Artur Pala
Nr grupy	W1

Porównanie algorytmów sortowania

Nazwisko i imię	Nr indeksu
Panek Bartosz	**101394**

Uwagi

SPIS TREŚCI

1. **CEL I ZAKRES** 3
2. **CHARAKTERYSTYKA REALIZOWANEGO ZADANIA** 4
3. **OPIS ŚRODOWISKA** 15
4. **SCENARIUSZE TESTOWE I METODOLOGIA BADAŃ** 16
5. **WYNIKI BADAŃ** 17
6. **WNIOSKI** 35
7. **BIBLIOGRAFIA** 36

1. CEL I ZAKRES

Celem projektu jest porównanie efektywności różnych algorytmów sortowania: Bubble Sort, Quick Sort, Merge Sort, Insertion Sort, oraz Selection Sort, w kontekście ich implementacji sekwencyjnych i równoległych. Równoległość osiągnięto przy użyciu OpenMP oraz wątków C++. Problem badawczy polega na określeniu, które z tych algorytmów zyskują najwięcej na równolegleniu oraz jakie są limity i efektywność tych przyspieszeń.

Opis algorytmów:

- **Bubble Sort:** Algorytm sortowania przez porównanie sąsiadujących elementów i ich zamianę, jeżeli są w niewłaściwej kolejności. Powtarzany aż do uzyskania posortowanej listy.
- **Quick Sort:** Algorytm sortowania dziel i zwyciężaj, który wybiera element jako pivot i dzieli tablicę na dwie części, następnie sortuje je rekurencyjnie.
- **Merge Sort:** Algorytm dziel i zwyciężaj, który dzieli tablicę na dwie połówki, sortuje każdą z nich rekurencyjnie, a następnie łączy posortowane połówki.
- **Insertion Sort:** Algorytm sortowania, który działa przez budowanie posortowanej listy jeden element na raz, poprzez wstawianie każdego nowego elementu w odpowiednie miejsce.
- **Selection Sort:** Algorytm sortowania, który działa przez wielokrotne znajdywanie najmniejszego elementu z nieposortowanej części tablicy i zamianę go z pierwszym elementem tej części.

Algorytmy sortowania są podstawą wielu aplikacji komputerowych, w tym przetwarzania danych, analizy danych, systemów baz danych, oraz w systemach operacyjnych. Wydajność tych algorytmów jest kluczowa dla optymalizacji wielu systemów komputerowych.

2. CHARAKTERYSTYKA REALIZOWANEGO ZADANIA

Program został napisany w języku C++ z wykorzystaniem bibliotek standardowych, OpenMP oraz mechanizmu wątków C++. Implementacja każdego z algorytmów sortowania zawiera trzy wersje: sekwencyjną, równoległą z użyciem OpenMP, oraz równoległą z użyciem wątków C++. Celem zrównoleglenia było przyspieszenie sortowania poprzez równoczesne wykonywanie wielu operacji sortowania.

1. Bubble sort

Sekwencyjny:

```
void bubbleSort(std::vector<Data>& data) {
    int n = data.size();
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (data[j].value > data[j + 1].value) {
                std::swap(data[j], data[j + 1]);
            }
        }
    }
}
```

Klasyczna implementacja sortowania bąbelkowego. Iteruje po elementach tablicy, porównując sąsiadujące elementy i zamieniając je miejscami, jeśli są w niewłaściwej kolejności.

OpenMP:

```
void bubbleSortOMP(std::vector<Data>& data, int num_threads) {
    int n = data.size();
    for (int i = 0; i < n - 1; ++i) {
#pragma omp parallel for shared(data) schedule(static) num_threads(num_threads)
        for (int j = 0; j < n - i - 1; ++j) {
            if (data[j].value > data[j + 1].value) {
                std::swap(data[j], data[j + 1]);
            }
        }
    }
}
```

Zrównoleglenie sortowania bąbelkowego za pomocą OpenMP. Pętla wewnętrzna została zrównoleglona przy użyciu dyrektywy #pragma omp parallel for. shared(data) oznacza, że wszystkie wątki współdzielą dane. schedule(static) rozdziela iteracje pętli równomiernie między wątkami.

C++ Threads:

```
void bubbleSortThreads(std::vector<Data>& data, int num_threads) {
    std::vector<std::thread> threads;
    const int chunk_size = data.size() / num_threads;
    std::mutex mutex;
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back([&, i] {
            int start_index = i * chunk_size;
            int end_index = (i == num_threads - 1) ? data.size() : start_index + chunk_size;
            for (int j = start_index; j < end_index; ++j) {
                for (int k = start_index; k < end_index - 1; ++k) {
                    if (data[k].value > data[k + 1].value) {
                        std::lock_guard<std::mutex> lock(mutex);
                        std::swap(data[k], data[k + 1]);
                    }
                }
            }
            });
    }

    for (auto& t : threads) {
        t.join();
    }
}
```

Sortowanie bąbelkowe z wykorzystaniem wątków w C++. Dane są dzielone na fragmenty (chunk_size) i każdy wątek sortuje swój fragment. std::mutex jest używany do ochrony krytycznych sekcji podczas zamiany elementów, aby uniknąć problemów z współbieżnością.

2. Quick sort

Sekwencyjny:

```
void quicksort(std::vector<Data>& data, int left, int right) {
    if (left >= right) return;
    int pivot = data[left + (right - left) / 2].value;
    int i = left, j = right;
    while (i <= j) {
        while (data[i].value < pivot) i++;
        while (data[j].value > pivot) j--;
        if (i <= j) {
            std::swap(data[i], data[j]);
            i++;
            j--;
        }
    }
    quicksort(data, left, j);
    quicksort(data, i, right);
}
```

Klasyczna rekurencyjna implementacja QuickSort. Wybiera pivot i porządkuje elementy względem pivota, a następnie rekurencyjnie sortuje lewą i prawą część.

OpenMP:

```cpp
void quickSortOMP(std::vector<Data>& data, int left, int right, int depth) {
    if (left >= right) return;
    int pivot = data[left + (right - left) / 2].value;
    int i = left, j = right;
    while (i <= j) {
        while (data[i].value < pivot) i++;
        while (data[j].value > pivot) j--;
        if (i <= j) {
            std::swap(data[i], data[j]);
            i++;
            j--;
        }
    }

    if (depth > 0) {
#pragma omp parallel sections
        {
#pragma omp section
            {
                quickSortOMP(data, left, j, depth - 1);
            }
#pragma omp section
            {
                quickSortOMP(data, i, right, depth - 1);
            }
        }
    }
    else {
        quickSortOMP(data, left, j, depth);
        quickSortOMP(data, i, right, depth);
    }
}
```

QuickSort z zrównolegleniem przy użyciu OpenMP. Użyto #pragma omp parallel sections, aby równocześnie sortować lewą i prawą część. depth kontroluje głębokość rekurencji i liczba aktywnych wątków jest zmniejszana na kolejnych poziomach rekurencji.

C++ Threads:

```cpp
void quickSortThread(std::vector<Data>& data, int left, int right, int num_threads) {
    if (left >= right) return;
    int pivot = data[left + (right - left) / 2].value;
    int i = left, j = right;
    while (i <= j) {
        while (data[i].value < pivot) i++;
        while (data[j].value > pivot) j--;
        if (i <= j) {
            std::swap(data[i], data[j]);
            i++;
            j--;
        }
    }

    if (num_threads > 1) {
        std::thread left_thread(quickSortThread, std::ref(data), left, j, num_threads / 2);
        std::thread right_thread(quickSortThread, std::ref(data), i, right, num_threads / 2);

        left_thread.join();
        right_thread.join();
    }
    else {
        quickSortThread(data, left, j, num_threads);
        quickSortThread(data, i, right, num_threads);
    }
}
```

QuickSort z zrównolegleniem za pomocą wątków C++. Dzieli dane na dwie części i rekurencyjnie sortuje każdą część w osobnych wątkach. Liczba wątków jest dzielona między lewą i prawą część rekurencji.

3. Merge sort

Sekwencyjny:

```cpp
void merge(std::vector<Data>& data, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<Data> left_half(n1), right_half(n2);

    for (int i = 0; i < n1; ++i)
        left_half[i] = data[left + i];
    for (int j = 0; j < n2; ++j)
        right_half[j] = data[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (left_half[i].value <= right_half[j].value) {
            data[k] = left_half[i];
            ++i;
        }
        else {
            data[k] = right_half[j];
            ++j;
        }
        ++k;
    }

    while (i < n1) {
        data[k] = left_half[i];
        ++i;
        ++k;
    }

    while (j < n2) {
        data[k] = right_half[j];
        ++j;
        ++k;
    }
}
```

```cpp
void mergeSort(std::vector<Data>& data, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(data, left, mid);
        mergeSort(data, mid + 1, right);
        merge(data, left, mid, right);
    }
}
```

Klasyczna rekurencyjna implementacja MergeSort. Dzieli dane na dwie części, sortuje je rekurencyjnie i następnie łączy (merge) w jedną posortowaną tablicę.

OpenMP:

```cpp
void mergeSortOMP(std::vector<Data>& data, int left, int right, int num_threads) {
    if (left < right) {
        int mid = left + (right - left) / 2;
#pragma omp parallel sections
        {
#pragma omp section
            mergeSortOMP(data, left, mid, num_threads);
#pragma omp section
            mergeSortOMP(data, mid + 1, right, num_threads);
        }
        merge(data, left, mid, right);
    }
}
```

MergeSort z zrównolegleniem przy użyciu OpenMP. Użyto #pragma omp parallel sections do równoczesnego sortowania lewej i prawej części danych.

C++ Threads:

```cpp
void mergeSortThreads(std::vector<Data>& data, int left, int right, int num_threads) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        if (num_threads > 1) {
            std::thread left_thread(mergeSortThreads, std::ref(data), left, mid, num_threads / 2);
            std::thread right_thread(mergeSortThreads, std::ref(data), mid + 1, right, num_threads - num_threads / 2);
            left_thread.join();
            right_thread.join();
        }
        else {
            mergeSortThreads(data, left, mid, num_threads);
            mergeSortThreads(data, mid + 1, right, num_threads);
        }
        merge(data, left, mid, right);
    }
}
```

MergeSort z zrównolegleniem za pomocą wątków C++. Dane są dzielone na dwie części, które są sortowane w osobnych wątkach. Liczba wątków jest dzielona między lewą i prawą część rekurencji.

4. Insertion sort

Sekwencyjny:

```cpp
void insertionSort(std::vector<Data>& data) {
    int n = data.size();
    for (int i = 1; i < n; ++i) {
        Data key = data[i];
        int j = i - 1;
        while (j >= 0 && data[j].value > key.value) {
            data[j + 1] = data[j];
            --j;
        }
        data[j + 1] = key;
    }
}
```

Klasyczna implementacja sortowania przez wstawianie. Każdy element jest porównywany z poprzednimi i wstawiany na odpowiednią pozycję.

OpenMP:

```cpp
void insertionSortOMP(std::vector<Data>& data, int num_threads) {
    int n = data.size();
#pragma omp parallel for num_threads(num_threads) shared(data) schedule(static)
    for (int i = 1; i < n; ++i) {
        Data key = data[i];
        int j = i - 1;
        while (j >= 0 && data[j].value > key.value) {
            data[j + 1] = data[j];
            --j;
        }
        data[j + 1] = key;
    }
}
```

Zrównoleglenie sortowania przez wstawianie za pomocą OpenMP. Pętla zewnętrzna została zrównoleglona przy użyciu dyrektywy #pragma omp parallel for.

C++ Threads:

```cpp
void insertionSortThreads(std::vector<Data>& data, int num_threads) {
    int n = data.size();
    std::vector<std::thread> threads;
    const int chunk_size = n / num_threads;
    std::mutex mutex;
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back([&, i] {
            int start_index = i * chunk_size;
            int end_index = (i == num_threads - 1) ? n : start_index + chunk_size;
            for (int j = start_index + 1; j < end_index; ++j) {
                Data key = data[j];
                int k = j - 1;
                while (k >= start_index && data[k].value > key.value) {
                    std::lock_guard<std::mutex> lock(mutex);
                    data[k + 1] = data[k];
                    --k;
                }
                data[k + 1] = key;
            }
            });
    }

    for (auto& t : threads) {
        t.join();
    }
}
```

Sortowanie przez wstawianie z wykorzystaniem wątków w C++. Dane są dzielone na fragmenty (chunk_size) i każdy wątek sortuje swój fragment. Brak synchronizacji między wątkami, ponieważ każdy wątek działa na swojej części danych.

5. Selection sort

Sekwencyjny:

```cpp
void selectionSort(std::vector<Data>& data) {
    int n = data.size();
    for (int i = 0; i < n - 1; ++i) {
        int min_idx = i;
        for (int j = i + 1; j < n; ++j) {
            if (data[j].value < data[min_idx].value) {
                min_idx = j;
            }
        }
        std::swap(data[i], data[min_idx]);
    }
}
```

Klasyczna implementacja sortowania przez wybieranie. W każdym kroku wybierany jest najmniejszy element i zamieniany z pierwszym nieposortowanym elementem.

OpenMP:

```
void selectionSortOMP(std::vector<Data>& data, int num_threads) {
    int n = data.size();
    for (int i = 0; i < n - 1; ++i) {
        int min_idx = i;
#pragma omp parallel for num_threads(num_threads) shared(data, min_idx)
        for (int j = i + 1; j < n; ++j) {
#pragma omp critical
            {
                if (data[j].value < data[min_idx].value) {
                    min_idx = j;
                }
            }
        }
        std::swap(data[i], data[min_idx]);
    }
}
```

Zrównoleglenie sortowania przez wybieranie za pomocą OpenMP. Pętla wewnętrzna została zrównoleglona przy użyciu dyrektywy #pragma omp parallel for. #pragma omp critical zapewnia, że tylko jeden wątek na raz może modyfikować min_idx, aby uniknąć problemów z współbieżnością.

C++ Threads:

```cpp
void selectionSortThreads(std::vector<Data>& data, int num_threads) {
    int n = data.size();
    const int chunk_size = n / num_threads;
    std::vector<std::thread> threads;
    std::mutex mutex;
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back([&, i] {
            int start_index = i * chunk_size;
            int end_index = (i == num_threads - 1) ? n : start_index + chunk_size;
            for (int j = start_index; j < end_index - 1; ++j) {
                int min_index = j;
                for (int k = j + 1; k < n; ++k) {
                    if (data[k].value < data[min_index].value) {
                        min_index = k;
                    }
                }
                if (min_index != j) {
                    std::lock_guard<std::mutex> lock(mutex);
                    std::swap(data[j], data[min_index]);
                }
            }
            });
    }

    for (auto& t : threads) {
        t.join();
    }
}
```

Sortowanie przez wybieranie z wykorzystaniem wątków w C++. Każdy wątek przeszukuje fragment tablicy, aby znaleźć lokalne minimum. std::mutex jest używany do ochrony krytycznych sekcji podczas modyfikacji min_idx, aby uniknąć problemów z współbieżnością.

3. OPIS ŚRODOWISKA

Parametry procesora	Rodzina procesorów: **Intel Core** Seria procesora: **i7-1165G7** Taktowanie rdzenia: **2.80 GHz** Liczba rdzeni fizycznych: **4 rdzeni** Liczba wątków: **8 wątków**
Pamięć RAM	Pamięć RAM: **16.0 GB** Szybkość: **3200 MHz**
Dysk	Dysk: **INTEL SSDPEKNW512G8H PCI-E x4 (MZVLQ512HALU-00000)** Format dysku: **M.2** Interfejs: **PCI-E 3.0 x4** Pojemność dysku: **512GB** Szybkość odczytu: **1500 MB/s** Szybkość zapisu: **1000 MB/s**
Karta graficzna	**NVIDIA GeForce MX450**
Dodatkowe funkcjonalności	Hyper threading

4. SCENARIUSZE TESTOWE I METODOLOGIA BADAŃ

Testowanie czasu wykonania:

- **Pomiar czasu wykonania dla różnych wartości parametru n (wymiar problemu) oraz p (liczba wątków):** Celem badania jest ocena efektywności implementacji równoległych względem implementacji sekwencyjnej. Przypadki testowe różniły się wymiarem problemu (n) oraz liczbą pracujących wątków (p). Przyjęto 5 wymiarów problemu (100, 5000, 25000, 50000, 100000 – liczby całkowite z zakresu 0-999999999) i 10 wariantów liczby wątków (1, 2, 4, 6, 8, 10, 12, 14, 16, 18).

Analiza przyspieszenia:

- **Obliczenie przyspieszenia dla każdej wartości n i liczby wątków:** Przyspieszenie będzie obliczane jako stosunek czasu wykonania dla pojedynczego wątku do czasu wykonania dla wielu wątków.

Obliczenie opłacalności i prawa Amdahla:

- **Obliczenie opłacalności:** Ocena opłacalności zwiększania liczby wątków w zależności od czasu wykonania i przyspieszenia. Czasem koszt związany z synchronizacją wątków może przeważyć korzyści wynikające z równoległego przetwarzania.

- **Obliczenie prawa Amdahla:** Obliczenie i analiza prawa Amdahla, aby zrozumieć, jakie ograniczenia nakłada to prawo na możliwość zwiększenia wydajności poprzez równoległe przetwarzanie.

Procentowy udział części szeregowej i równoległej

- Procentowy udział części szeregowej i równoległej został wyznaczony empirycznie.

Ziarnistość problemu

5. WYNIKI BADAŃ

Tabela 5.1 Wyniki badań.

Liczba wątków	Algorytm sortowania	Technologia	Rozmiar problemu	Czas (s)	Przyśpieszenie (s)	Przyśpieszenie (%)
1	Bąbelkowy	Sekwencyjnie	100	0,000069	-	-
	Szybki			0,0000076	-	-
	Przez scalanie			0,0001983	-	-
	Przez wstawianie			0,0000076	-	-
	Przez wybieranie			0,0000344	-	-
	Bąbelkowy		5000	0,186061	-	-
	Szybki			0,0009123	-	-
	Przez scalanie			0,0159319	-	-
	Przez wstawianie			0,0001028	-	-
	Przez wybieranie			0,0735086	-	-
	Bąbelkowy		25000	4,64213	-	-
	Szybki			0,0025354	-	-
	Przez scalanie			0,0705201	-	-
	Przez wstawianie			0,000465	-	-
	Przez wybieranie			1,84262	-	-
	Bąbelkowy		50000	19,4884	-	-
	Szybki			0,005719	-	-
	Przez scalanie			0,1407503	-	-
	Przez wstawianie			0,0014771	-	-
	Przez wybieranie			7,67891	-	-
	Bąbelkowy		100000	79,9921	-	-
	Szybki			0,0095672	-	-
	Przez scalanie			0,24083	-	-
	Przez wstawianie			0,0020993	-	-
	Przez wybieranie			31,7511	-	-
2	Bąbelkowy	OpenMP	100	0,0037009	-0,0036319	-526362%
		Thread C++		0,0023793	-0,0023103	-334826%
	Szybki	OpenMP		0,0000108	-0,0000032	-4211%
		Thread C++		0,0012899	-0,0012823	-1687237%
	Przez scalanie	OpenMP		0,0002283	-0,00003	-1513%
		Thread C++		0,0010719	-0,0008736	-44054%
	Przez wstawianie	OpenMP		0,0000082	-6E-07	-789%
		Thread C++		0,0004278	-0,0004202	-552895%
	Przez wybieranie	OpenMP		0,0001969	-0,0001625	-47238%

		Thread C++		0,0003748	-0,0003404	-98953%
	Bąbelkowy	OpenMP	5000	0,154198	0,031863	1713%
		Thread C++		0,0528085	0,1332525	7162%
	Szybki	OpenMP		0,0003624	0,0005499	6028%
		Thread C++		0,0010771	-0,0001648	-1806%
	Przez scalanie	OpenMP		0,0152906	0,0006413	403%
		Thread C++		0,0146945	0,0012374	777%
	Przez wstawianie	OpenMP		0,0000963	6,5E-06	632%
		Thread C++		0,0008554	-0,0007526	-73210%
	Przez wybieranie	OpenMP		0,0576945	0,0158141	2151%
		Thread C++		0,080799	-0,0072904	-992%
	Bąbelkowy	OpenMP	25000	3,74973	0,8924	1922%
		Thread C++		1,29143	3,3507	7218%
	Szybki	OpenMP		0,0030461	-0,0005107	-2014%
		Thread C++		0,0033625	0,0671576	9523%
	Przez scalanie	OpenMP		0,0659653	0,0045548	646%
		Thread C++		0,0659752	-0,0154551	-3059%
	Przez wstawianie	OpenMP		0,0002936	0,0001714	3686%
		Thread C++		0,0007893	-0,0003243	-6974%
	Przez wybieranie	OpenMP		1,42357	0,41905	2274%
		Thread C++		1,74609	0,09653	524%
	Bąbelkowy	OpenMP	50000	13,7509	5,7375	2944%
		Thread C++		4,63551	14,85289	7621%
	Szybki	OpenMP		0,0044745	0,0012445	2176%
		Thread C++		0,0057715	-5,25E-05	-92%
	Przez scalanie	OpenMP		0,118502	0,0222483	1581%
		Thread C++		0,133894	0,0068563	487%
	Przez wstawianie	OpenMP		0,0005202	0,0009569	6478%
		Thread C++		0,0014359	4,12E-05	279%
	Przez wybieranie	OpenMP		5,18849	2,49042	3243%
		Thread C++		6,92078	0,75813	987%
	Bąbelkowy	OpenMP	100000	61,4841	18,508	2314%
		Thread C++		21,7793	58,2128	7277%
	Szybki	OpenMP		0,0134109	-0,0038437	-4018%
		Thread C++		0,0082692	0,001298	1357%
	Przez scalanie	OpenMP		0,223241	0,017589	730%
		Thread C++		0,222149	0,018681	776%
	Przez wstawianie	OpenMP		0,0008838	0,0012155	5790%
		Thread C++		0,001383	0,0007163	3412%
	Przez wybieranie	OpenMP		25,2596	6,4915	2044%
		Thread C++		27,5662	4,1849	1318%
4	Bąbelkowy	OpenMP	100	0,0008105	-0,0007415	-107464%
		Thread C++		0,0016967	-0,0016277	-235899%
	Szybki	OpenMP		0,0000172	-0,0000096	-12632%
		Thread C++		0,0009737	-0,0009661	-1271184%
	Przez scalanie	OpenMP		0,0005131	-0,0003148	-15875%

		Thread C++		0,0009363	-0,000738	-37216%
	Przez wstawianie	OpenMP		0,0000051	0,0000025	3289%
		Thread C++		0,0005847	-0,0005771	-759342%
	Przez wybieranie	OpenMP		0,0004136	-0,0003792	-110233%
		Thread C++		0,0008193	-0,0007849	-228169%
	Bąbelkowy	OpenMP	5000	0,11239	0,073671	3960%
		Thread C++		0,0258902	0,1601708	8609%
	Szybki	OpenMP		0,000694	0,0002183	2393%
		Thread C++		0,0007618	0,0001505	1650%
	Przez scalanie	OpenMP		0,0163132	-0,0003813	-239%
		Thread C++		0,0190834	-0,0031515	-1978%
	Przez wstawianie	OpenMP		0,0000441	0,0000587	5710%
		Thread C++		0,0006656	-0,0005628	-54747%
	Przez wybieranie	OpenMP		0,0518737	0,0216349	2943%
		Thread C++		0,0666139	0,0068947	938%
	Bąbelkowy	OpenMP	25000	2,74364	1,89849	4090%
		Thread C++		0,417668	4,224462	9100%
	Szybki	OpenMP		0,0049971	-0,0024617	-9709%
		Thread C++		0,0023665	0,0681536	9664%
	Przez scalanie	OpenMP		0,0762509	-0,0057308	-813%
		Thread C++		0,0893392	-0,0388191	-7684%
	Przez wstawianie	OpenMP		0,0001853	0,0002797	6015%
		Thread C++		0,0006912	-0,0002262	-4865%
	Przez wybieranie	OpenMP		1,2643	0,57832	3139%
		Thread C++		1,06077	0,78185	4243%
	Bąbelkowy	OpenMP	50000	10,4623	9,0261	4632%
		Thread C++		1,73042	17,75798	9112%
	Szybki	OpenMP		0,0074271	-0,0017081	-2987%
		Thread C++		0,0046314	0,0010876	1902%
	Przez scalanie	OpenMP		0,12943	0,0113203	804%
		Thread C++		0,149257	-0,0085067	-604%
	Przez wstawianie	OpenMP		0,000676	0,0008011	5423%
		Thread C++		0,0008853	0,0005918	4006%
	Przez wybieranie	OpenMP		4,86259	2,81632	3668%
		Thread C++		4,47442	3,20449	4173%
	Bąbelkowy	OpenMP	100000	47,0821	32,91	4114%
		Thread C++		6,68196	73,31014	9165%
	Szybki	OpenMP		0,011835	-0,0022678	-2370%
		Thread C++		0,0077444	0,0018228	1905%
	Przez scalanie	OpenMP		0,257583	-0,016753	-696%
		Thread C++		0,367017	-0,126187	-5240%
	Przez wstawianie	OpenMP		0,0008148	0,0012845	6119%
		Thread C++		0,001185	0,0009143	4355%
	Przez wybieranie	OpenMP		17,6859	14,0652	4430%
		Thread C++		18,7489	13,0022	4095%
6	Bąbelkowy	OpenMP	100	0,001165	-0,001096	-158841%

		Thread C++		0,0013451	-0,0012761	-184942%
	Szybki	OpenMP		0,0000322	-0,0000246	-32368%
		Thread C++		0,0006139	-0,0006063	-797763%
	Przez scalanie	OpenMP		0,0004101	-0,0002118	-10681%
		Thread C++		0,0014195	-0,0012212	-61583%
	Przez wstawianie	OpenMP		0,0000124	-0,0000048	-6316%
		Thread C++		0,0006448	-0,0006372	-838421%
	Przez wybieranie	OpenMP		0,000423	-0,0003886	-112965%
		Thread C++		0,0005422	-0,0005078	-147616%
	Bąbelkowy	OpenMP	5000	0,0906237	0,0954373	5129%
		Thread C++		0,0146164	0,1714446	9214%
	Szybki	OpenMP		0,0006526	0,0002597	2847%
		Thread C++		0,0008092	0,0001031	1130%
	Przez scalanie	OpenMP		0,0196937	-0,0037618	-2361%
		Thread C++		0,0185094	-0,0025775	-1618%
	Przez wstawianie	OpenMP		0,0000426	0,0000602	5856%
		Thread C++		0,0005479	-0,0004451	-43298%
	Przez wybieranie	OpenMP		0,0426208	0,0308878	4202%
		Thread C++		0,0590662	0,0144424	1965%
	Bąbelkowy	OpenMP	25000	1,64874	2,99339	6448%
		Thread C++		0,222661	4,419469	9520%
	Szybki	OpenMP		0,0026969	-0,0001615	-637%
		Thread C++		0,0013817	0,0691384	9804%
	Przez scalanie	OpenMP		0,0599664	0,0105537	1497%
		Thread C++		0,0637169	-0,0131968	-2612%
	Przez wstawianie	OpenMP		0,0001301	0,0003349	7202%
		Thread C++		0,0004644	6E-07	13%
	Przez wybieranie	OpenMP		0,768345	1,074275	5830%
		Thread C++		0,768412	1,074208	5830%
	Bąbelkowy	OpenMP	50000	6,8886	12,5998	6465%
		Thread C++		0,826352	18,662048	9576%
	Szybki	OpenMP		0,0046375	0,0010815	1891%
		Thread C++		0,0026413	0,0030777	5382%
	Przez scalanie	OpenMP		0,109276	0,0314743	2236%
		Thread C++		0,127985	0,0127653	907%
	Przez wstawianie	OpenMP		0,0002182	0,0012589	8523%
		Thread C++		0,0005597	0,0009174	6211%
	Przez wybieranie	OpenMP		3,2376	4,44131	5784%
		Thread C++		3,15667	4,52224	5889%
	Bąbelkowy	OpenMP	100000	31,047	48,9451	6119%
		Thread C++		4,07182	75,92028	9491%
	Szybki	OpenMP		0,0177665	-0,0081993	-8570%
		Thread C++		0,0071454	0,0024218	2531%
	Przez scalanie	OpenMP		0,28714	-0,04631	-1923%
		Thread C++		0,305125	-0,064295	-2670%
	Przez wstawianie	OpenMP		0,0005232	0,0015761	7508%

		Thread C++		0,0008088	0,0012905	6147%
	Przez wybieranie	OpenMP		16,138	15,6131	4917%
		Thread C++		15,5229	16,2282	5111%
8	Bąbelkowy	OpenMP	100	0,0010032	-0,0009342	-135391%
		Thread C++		0,0011183	-0,0010493	-152072%
	Szybki	OpenMP		0,0000226	-0,000015	-19737%
		Thread C++		0,0007781	-0,0007705	-1013816%
	Przez scalanie	OpenMP		0,0002436	-0,0000453	-2284%
		Thread C++		0,0009657	-0,0007674	-38699%
	Przez wstawianie	OpenMP		0,0000061	0,0000015	1974%
		Thread C++		0,0004761	-0,0004685	-616447%
	Przez wybieranie	OpenMP		0,0004584	-0,000424	-123256%
		Thread C++		0,0005167	-0,0004823	-140203%
	Bąbelkowy	OpenMP	5000	0,0586758	0,1273852	6846%
		Thread C++		0,0067885	0,1792725	9635%
	Szybki	OpenMP		0,0005267	0,0003856	4227%
		Thread C++		0,0009328	-0,0000205	-225%
	Przez scalanie	OpenMP		0,0134319	0,0025	1569%
		Thread C++		0,0130112	0,0029207	1833%
	Przez wstawianie	OpenMP		0,0001369	-0,0000341	-3317%
		Thread C++		0,000555	-0,0004522	-43988%
	Przez wybieranie	OpenMP		0,0287132	0,0447954	6094%
		Thread C++		0,0405552	0,0329534	4483%
	Bąbelkowy	OpenMP	25000	1,37532	3,26681	7037%
		Thread C++		0,170484	4,471646	9633%
	Szybki	OpenMP		0,0028049	-0,0002695	-1063%
		Thread C++		0,0016532	0,0688669	9766%
	Przez scalanie	OpenMP		0,0609635	0,0095566	1355%
		Thread C++		0,0838478	-0,0333277	-6597%
	Przez wstawianie	OpenMP		0,0001331	0,0003319	7138%
		Thread C++		0,0005629	-9,79E-05	-2105%
	Przez wybieranie	OpenMP		0,633876	1,208744	6560%
		Thread C++		0,70781	1,13481	6159%
	Bąbelkowy	OpenMP	50000	5,61599	13,87241	7118%
		Thread C++		0,669094	18,819306	9657%
	Szybki	OpenMP		0,0065699	-0,0008509	-1488%
		Thread C++		0,0028551	0,0028639	5008%
	Przez scalanie	OpenMP		0,109416	0,0313343	2226%
		Thread C++		0,178686	-0,0379357	-2695%
	Przez wstawianie	OpenMP		0,0002594	0,0012177	8244%
		Thread C++		0,0006919	0,0007852	5316%
	Przez wybieranie	OpenMP		2,67622	5,00269	6515%
		Thread C++		3,16	4,51891	5885%
	Bąbelkowy	OpenMP	100000	26,3034	53,6887	6712%
		Thread C++		2,90395	77,08815	9637%
	Szybki	OpenMP		0,0141985	-0,0046313	-4841%

		Thread C++		0,0053034	0,0042638	4457%
	Przez scalanie	OpenMP		0,228984	0,011846	492%
		Thread C++		0,366252	-0,125422	-5208%
	Przez wstawianie	OpenMP		0,000374	0,0017253	8218%
		Thread C++		0,0009003	0,001199	5711%
	Przez wybieranie	OpenMP		13,7242	18,0269	5678%
		Thread C++		13,982	17,7691	5596%
10	Bąbelkowy	OpenMP	100	0,0012385	-0,0011695	-169493%
		Thread C++		0,0013936	-0,0013246	-191971%
	Szybki	OpenMP		0,0000229	-0,0000153	-20132%
		Thread C++		0,0008868	-0,0008792	-1156842%
	Przez scalanie	OpenMP		0,000298	-0,0000997	-5028%
		Thread C++		0,0012788	-0,0010805	-54488%
	Przez wstawianie	OpenMP		0,0000082	-6E-07	-789%
		Thread C++		0,0005927	-0,0005851	-769868%
	Przez wybieranie	OpenMP		0,0005845	-0,0005501	-159913%
		Thread C++		0,0006222	-0,0005878	-170872%
	Bąbelkowy	OpenMP	5000	0,0837277	0,1023333	5500%
		Thread C++		0,0070625	0,1789985	9620%
	Szybki	OpenMP		0,0039077	-0,0029954	-32833%
		Thread C++		0,0010251	-0,0001128	-1236%
	Przez scalanie	OpenMP		0,0143962	0,0015357	964%
		Thread C++		0,0157354	0,0001965	123%
	Przez wstawianie	OpenMP		0,000041	0,0000618	6012%
		Thread C++		0,0006238	-0,000521	-50681%
	Przez wybieranie	OpenMP		0,0487746	0,024734	3365%
		Thread C++		0,0389949	0,0345137	4695%
	Bąbelkowy	OpenMP	25000	1,83558	2,80655	6046%
		Thread C++		0,149097	4,493033	9679%
	Szybki	OpenMP		0,0061589	-0,0036235	-14292%
		Thread C++		0,0018492	0,0686709	9738%
	Przez scalanie	OpenMP		0,0662668	0,0042533	603%
		Thread C++		0,0734663	-0,0229462	-4542%
	Przez wstawianie	OpenMP		0,0001244	0,0003406	7325%
		Thread C++		0,0006333	-0,0001683	-3619%
	Przez wybieranie	OpenMP		0,907282	0,935338	5076%
		Thread C++		0,693461	1,149159	6237%
	Bąbelkowy	OpenMP	50000	7,44118	12,04722	6182%
		Thread C++		0,549051	18,939349	9718%
	Szybki	OpenMP		0,0095261	-0,0038071	-6657%
		Thread C++		0,0026385	0,0030805	5386%
	Przez scalanie	OpenMP		0,117698	0,0230523	1638%
		Thread C++		0,146971	-0,0062207	-442%
	Przez wstawianie	OpenMP		0,000241	0,0012361	8368%
		Thread C++		0,0006594	0,0008177	5536%
	Przez wybieranie	OpenMP		3,75487	3,92404	5110%

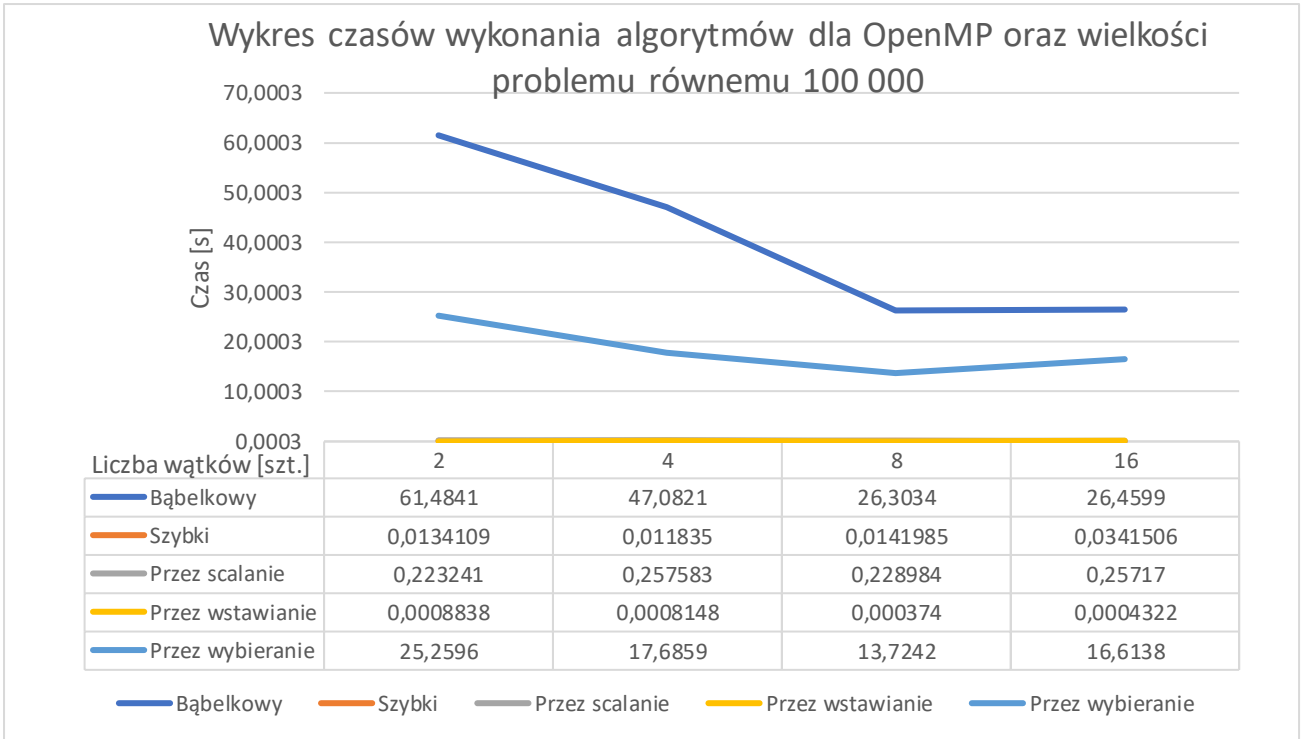
		Thread C++		3,05439	4,62452	6022%
	Bąbelkowy	OpenMP	100000	32,2433	47,7488	5969%
		Thread C++		2,75544	77,23666	9656%
	Szybki	OpenMP		0,0187997	-0,0092325	-9650%
		Thread C++		0,006384	0,0031832	3327%
	Przez scalanie	OpenMP		0,246085	-0,005255	-218%
		Thread C++		0,37027	-0,12944	-5375%
	Przez wstawianie	OpenMP		0,0004385	0,0016608	7911%
		Thread C++		0,0009249	0,0011744	5594%
	Przez wybieranie	OpenMP		17,454	14,2971	4503%
		Thread C++		14,3113	17,4398	5493%
12	Bąbelkowy	OpenMP	100	0,0014774	-0,0014084	-204116%
		Thread C++		0,0012683	-0,0011993	-173812%
	Szybki	OpenMP		0,0000227	-0,0000151	-19868%
		Thread C++		0,0008546	-0,000847	-1114474%
	Przez scalanie	OpenMP		0,0002674	-0,0000691	-3485%
		Thread C++		0,0012101	-0,0010118	-51024%
	Przez wstawianie	OpenMP		0,0000061	0,0000015	1974%
		Thread C++		0,0006747	-0,0006671	-877763%
	Przez wybieranie	OpenMP		0,0006161	-0,0005817	-169099%
		Thread C++		0,0006358	-0,0006014	-174826%
	Bąbelkowy	OpenMP	5000	0,0929662	0,0930948	5003%
		Thread C++		0,0059125	0,1801485	9682%
	Szybki	OpenMP		0,00425	-0,0033377	-36586%
		Thread C++		0,0008001	0,0001122	1230%
	Przez scalanie	OpenMP		0,0147651	0,0011668	732%
		Thread C++		0,0154485	0,0004834	303%
	Przez wstawianie	OpenMP		0,0000394	0,0000634	6167%
		Thread C++		0,0006907	-0,0005879	-57189%
	Przez wybieranie	OpenMP		0,0505969	0,0229117	3117%
		Thread C++		0,0314453	0,0420633	5722%
	Bąbelkowy	OpenMP	25000	1,73039	2,91174	6272%
		Thread C++		0,124927	4,517203	9731%
	Szybki	OpenMP		0,0068443	-0,0043089	-16995%
		Thread C++		0,0016827	0,0688374	9761%
	Przez scalanie	OpenMP		0,06686	0,0036601	519%
		Thread C++		0,0728221	-0,022302	-4414%
	Przez wstawianie	OpenMP		0,0001584	0,0003066	6594%
		Thread C++		0,0009769	-0,0005119	-11009%
	Przez wybieranie	OpenMP		0,893591	0,949029	5150%
		Thread C++		0,704222	1,138398	6178%
	Bąbelkowy	OpenMP	50000	6,91668	12,57172	6451%
		Thread C++		0,474101	19,014299	9757%
	Szybki	OpenMP		0,0101797	-0,0044607	-7800%
		Thread C++		0,0021385	0,0035805	6261%
	Przez scalanie	OpenMP		0,121535	0,0192153	1365%

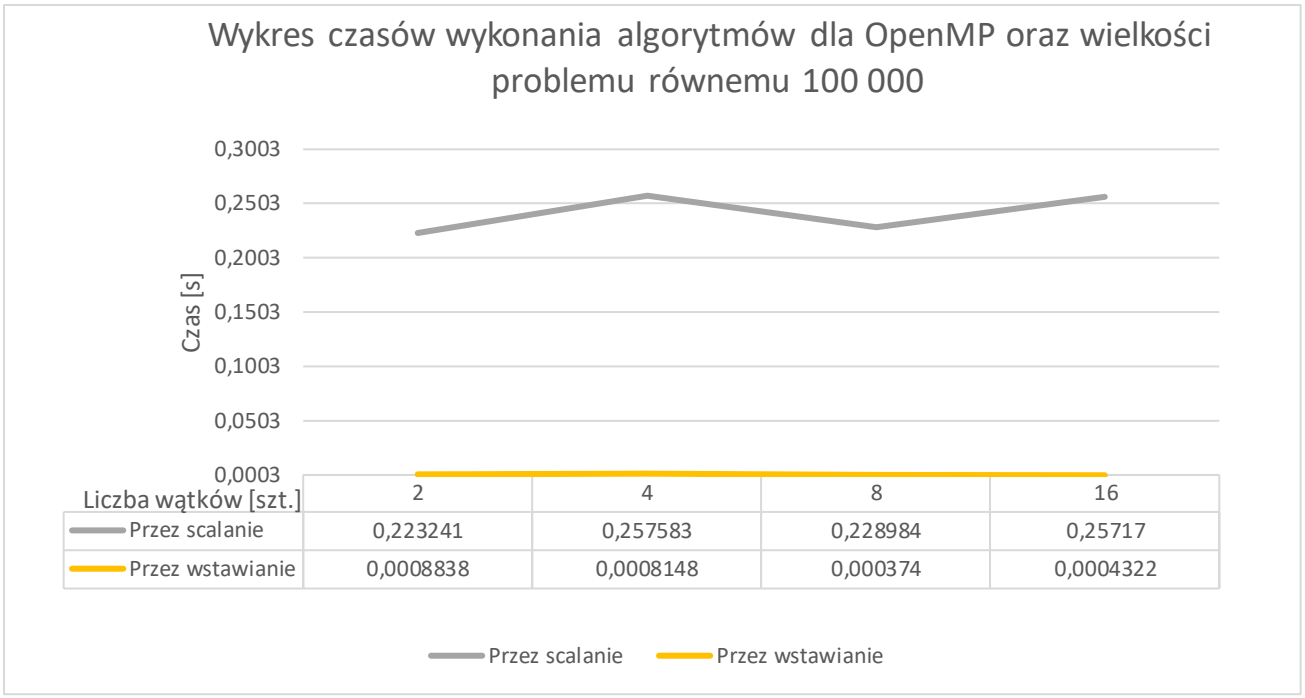
		Thread C++		0,140689	6,13E-05	4%
	Przez wstawianie	OpenMP		0,0002273	0,0012498	8461%
		Thread C++		0,00076	0,0007171	4855%
	Przez wybieranie	OpenMP		3,54275	4,13616	5386%
		Thread C++		2,76203	4,91688	6403%
	Bąbelkowy	OpenMP	100000	28,8104	51,1817	6398%
		Thread C++		1,88429	78,10781	9764%
	Szybki	OpenMP		0,0169283	-0,0073611	-7694%
		Thread C++		0,0041197	0,0054475	5694%
	Przez scalanie	OpenMP		0,250636	-0,009806	-407%
		Thread C++		0,307069	-0,066239	-2750%
	Przez wstawianie	OpenMP		0,0003863	0,001713	8160%
		Thread C++		0,001153	0,0009463	4508%
	Przez wybieranie	OpenMP		16,7817	14,9694	4715%
		Thread C++		13,8	17,9511	5654%
14	Bąbelkowy	OpenMP	100	0,0015158	-0,0014468	-209681%
		Thread C++		0,0013187	-0,0012497	-181116%
	Szybki	OpenMP		0,000032	-0,0000244	-32105%
		Thread C++		0,0009052	-0,0008976	-1181053%
	Przez scalanie	OpenMP		0,0001993	-1E-06	-50%
		Thread C++		0,0015167	-0,0013184	-66485%
	Przez wstawianie	OpenMP		0,0000092	-0,0000016	-2105%
		Thread C++		0,0008311	-0,0008235	-1083553%
	Przez wybieranie	OpenMP		0,000793	-0,0007586	-220523%
		Thread C++		0,0007911	-0,0007567	-219971%
	Bąbelkowy	OpenMP	5000	0,0900052	0,0960558	5163%
		Thread C++		0,004439	0,181622	9761%
	Szybki	OpenMP		0,0042627	-0,0033504	-36725%
		Thread C++		0,0009204	-8,1E-06	-89%
	Przez scalanie	OpenMP		0,016086	-0,0001541	-97%
		Thread C++		0,015345	0,0005869	368%
	Przez wstawianie	OpenMP		0,0000498	0,000053	5156%
		Thread C++		0,0007666	-0,0006638	-64572%
	Przez wybieranie	OpenMP		0,0529765	0,0205321	2793%
		Thread C++		0,0306911	0,0428175	5825%
	Bąbelkowy	OpenMP	25000	1,63121	3,01092	6486%
		Thread C++		0,113027	4,529103	9757%
	Szybki	OpenMP		0,0094508	-0,0069154	-27275%
		Thread C++		0,002048	0,0684721	9710%
	Przez scalanie	OpenMP		0,0593299	0,0111902	1587%
		Thread C++		0,0774855	-0,0269654	-5338%
	Przez wstawianie	OpenMP		0,0001239	0,0003411	7335%
		Thread C++		0,0009488	-0,0004838	-10404%
	Przez wybieranie	OpenMP		0,892533	0,950087	5156%
		Thread C++		0,69888	1,14374	6207%
	Bąbelkowy	OpenMP	50000	6,60621	12,88219	6610%

		Thread C++		0,387328	19,101072	9801%
	Szybki	OpenMP		0,0131598	-0,0074408	-13011%
		Thread C++		0,0024927	0,0032263	5641%
	Przez scalanie	OpenMP		0,122394	0,0183563	1304%
		Thread C++		0,156463	-0,0157127	-1116%
	Przez wstawianie	OpenMP		0,0002308	0,0012463	8437%
		Thread C++		0,0009075	0,0005696	3856%
	Przez wybieranie	OpenMP		3,32287	4,35604	5673%
		Thread C++		2,81159	4,86732	6339%
	Bąbelkowy	OpenMP	100000	28,0442	51,9479	6494%
		Thread C++		1,72909	78,26301	9784%
	Szybki	OpenMP		0,0270875	-0,0175203	-18313%
		Thread C++		0,0053085	0,0042587	4451%
	Przez scalanie	OpenMP		0,272328	-0,031498	-1308%
		Thread C++		0,395206	-0,154376	-6410%
	Przez wstawianie	OpenMP		0,0005695	0,0015298	7287%
		Thread C++		0,0014358	0,0006635	3161%
	Przez wybieranie	OpenMP		15,5498	16,2013	5103%
		Thread C++		13,6379	18,1132	5705%
16	Bąbelkowy	OpenMP	100	0,001745	-0,001676	-242899%
		Thread C++		0,001511	-0,001442	-208986%
	Szybki	OpenMP		0,000032	-0,0000244	-32105%
		Thread C++		0,0016164	-0,0016088	-2116842%
	Przez scalanie	OpenMP		0,0002572	-0,0000589	-2970%
		Thread C++		0,0016882	-0,0014899	-75134%
	Przez wstawianie	OpenMP		0,0000126	-0,000005	-6579%
		Thread C++		0,000873	-0,0008654	-1138684%
	Przez wybieranie	OpenMP		0,0008733	-0,0008389	-243866%
		Thread C++		0,0008697	-0,0008353	-242820%
	Bąbelkowy	OpenMP	5000	0,0874551	0,0986059	5300%
		Thread C++		0,005358	0,180703	9712%
	Szybki	OpenMP		0,0027006	-0,0017883	-19602%
		Thread C++		0,0016397	-0,0007274	-7973%
	Przez scalanie	OpenMP		0,0164007	-0,0004688	-294%
		Thread C++		0,0142959	0,001636	1027%
	Przez wstawianie	OpenMP		0,0000374	0,0000654	6362%
		Thread C++		0,0008964	-0,0007936	-77198%
	Przez wybieranie	OpenMP		0,056928	0,0165806	2256%
		Thread C++		0,0304362	0,0430724	5860%
	Bąbelkowy	OpenMP	25000	1,4696	3,17253	6834%
		Thread C++		0,0946826	4,5474474	9796%
	Szybki	OpenMP		0,0095206	-0,0069852	-27551%
		Thread C++		0,0023429	0,0681772	9668%
	Przez scalanie	OpenMP		0,0598781	0,010642	1509%
		Thread C++		0,0749653	-0,0244452	-4839%
	Przez wstawianie	OpenMP		0,0001579	0,0003071	6604%

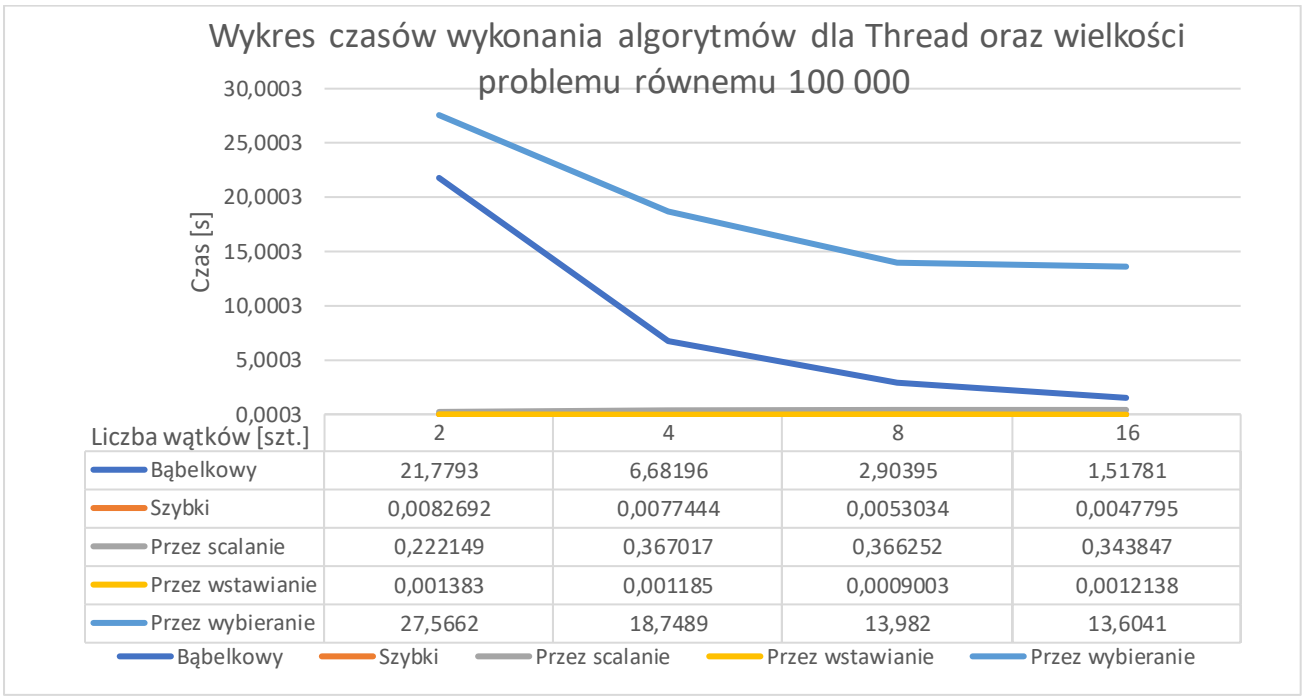
		Thread C++		0,0010069	-0,0005419	-11654%
	Przez wybieranie	OpenMP		0,927319	0,915301	4967%
		Thread C++		0,613606	1,229014	6670%
	Bąbelkowy	OpenMP	50000	6,52586	12,96254	6651%
		Thread C++		0,333177	19,155223	9829%
	Szybki	OpenMP		0,0173878	-0,0116688	-20404%
		Thread C++		0,0029661	0,0027529	4814%
	Przez scalanie	OpenMP		0,124991	0,0157593	1120%
		Thread C++		0,147618	-0,0068677	-488%
	Przez wstawianie	OpenMP		0,0002279	0,0012492	8457%
		Thread C++		0,0010261	0,000451	3053%
	Przez wybieranie	OpenMP		3,5576	4,12131	5367%
		Thread C++		2,74533	4,93358	6425%
	Bąbelkowy	OpenMP	100000	26,4599	53,5322	6692%
		Thread C++		1,51781	78,47429	9810%
	Szybki	OpenMP		0,0341506	-0,0245834	-25696%
		Thread C++		0,0047795	0,0047877	5004%
	Przez scalanie	OpenMP		0,25717	-0,01634	-678%
		Thread C++		0,343847	-0,103017	-4278%
	Przez wstawianie	OpenMP		0,0004322	0,0016671	7941%
		Thread C++		0,0012138	0,0008855	4218%
	Przez wybieranie	OpenMP		16,6138	15,1373	4767%
		Thread C++		13,6041	18,147	5715%
18	Bąbelkowy	OpenMP	100	0,0019126	-0,0018436	-267188%
		Thread C++		0,0015414	-0,0014724	-213391%
	Szybki	OpenMP		0,0000449	-0,0000373	-49079%
		Thread C++		0,0015183	-0,0015107	-1987763%
	Przez scalanie	OpenMP		0,0002306	-0,0000323	-1629%
		Thread C++		0,0016996	-0,0015013	-75709%
	Przez wstawianie	OpenMP		0,0000225	-0,0000149	-19605%
		Thread C++		0,0008602	-0,0008526	-1121842%
	Przez wybieranie	OpenMP		0,0010194	-0,000985	-286337%
		Thread C++		0,0010151	-0,0009807	-285087%
	Bąbelkowy	OpenMP	5000	0,103966	0,082095	4412%
		Thread C++		0,0041448	0,1819162	9777%
	Szybki	OpenMP		0,0042685	-0,0033562	-36788%
		Thread C++		0,0016819	-0,0007696	-8436%
	Przez scalanie	OpenMP		0,0132382	0,0026937	1691%
		Thread C++		0,0155802	0,0003517	221%
	Przez wstawianie	OpenMP		0,0000364	0,0000664	6459%
		Thread C++		0,0010703	-0,0009675	-94115%
	Przez wybieranie	OpenMP		0,0761835	-0,0026749	-364%
		Thread C++		0,0305392	0,0429694	5845%
	Bąbelkowy	OpenMP	25000	1,75536	2,88677	6219%
		Thread C++		0,101843	4,540287	9781%
	Szybki	OpenMP		0,0099717	-0,0074363	-29330%

	Thread C++		0,0023171	0,068203	9671%
Przez scalanie	OpenMP		0,065228	0,0052921	750%
	Thread C++		0,0722916	-0,0217715	-4309%
Przez wstawianie	OpenMP		0,0001593	0,0003057	6574%
	Thread C++		0,0010219	-0,0005569	-11976%
Przez wybieranie	OpenMP		0,932702	0,909918	4938%
	Thread C++		0,667206	1,175414	6379%
Bąbelkowy	OpenMP	50000	6,94739	12,54101	6435%
	Thread C++		0,28234	19,20606	9855%
Szybki	OpenMP		0,0156157	-0,0098967	-17305%
	Thread C++		0,0031323	0,0025867	4523%
Przez scalanie	OpenMP		0,159253	-0,0185027	-1315%
	Thread C++		0,150669	-0,0099187	-705%
Przez wstawianie	OpenMP		0,0002164	0,0012607	8535%
	Thread C++		0,0011636	0,0003135	2122%
Przez wybieranie	OpenMP		3,32727	4,35164	5667%
	Thread C++		2,49188	5,18703	6755%
Bąbelkowy	OpenMP	100000	28,1116	51,8805	6486%
	Thread C++		1,43091	78,56119	9821%
Szybki	OpenMP		0,042862	-0,0332948	-34801%
	Thread C++		0,0071707	0,0023965	2505%
Przez scalanie	OpenMP		0,266814	-0,025984	-1079%
	Thread C++		0,376235	-0,135405	-5622%
Przez wstawianie	OpenMP		0,0004478	0,0016515	7867%
	Thread C++		0,0014657	0,0006336	3018%
Przez wybieranie	OpenMP		16,0449	15,7062	4947%
	Thread C++		13,3654	18,3857	5791%



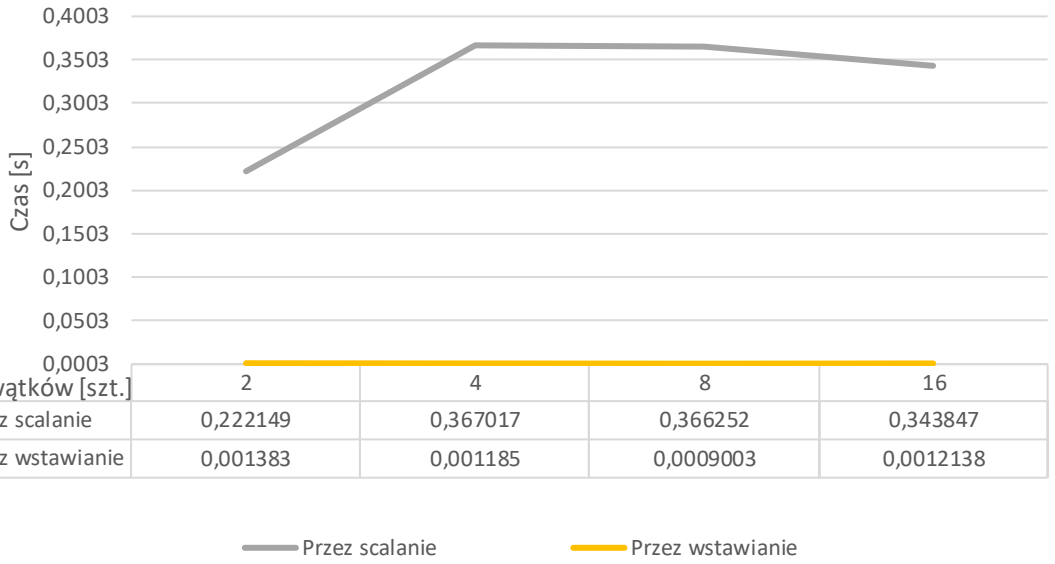


Liczba wątków [szt.]	2	4	8	16
Przez scalanie	0,223241	0,257583	0,228984	0,25717
Przez wstawianie	0,0008838	0,0008148	0,000374	0,0004322



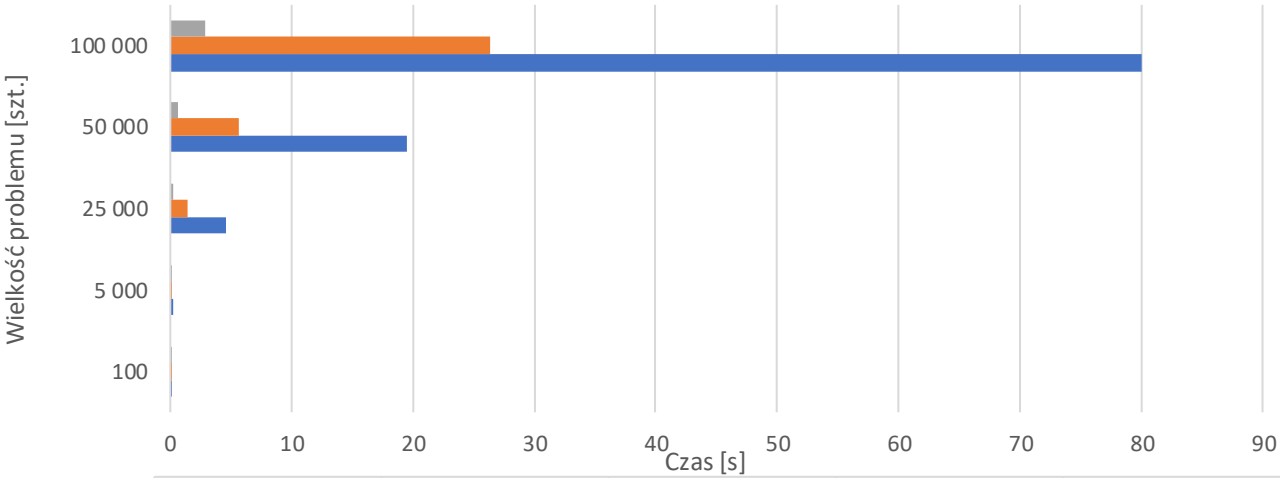
Liczba wątków [szt.]	2	4	8	16
Bąbelkowy	21,7793	6,68196	2,90395	1,51781
Szybki	0,0082692	0,0077444	0,0053034	0,0047795
Przez scalanie	0,222149	0,367017	0,366252	0,343847
Przez wstawianie	0,001383	0,001185	0,0009003	0,0012138
Przez wybieranie	27,5662	18,7489	13,982	13,6041

Wykres czasów wykonania algorytmów dla Thread oraz wielkości problemu równemu 100 000

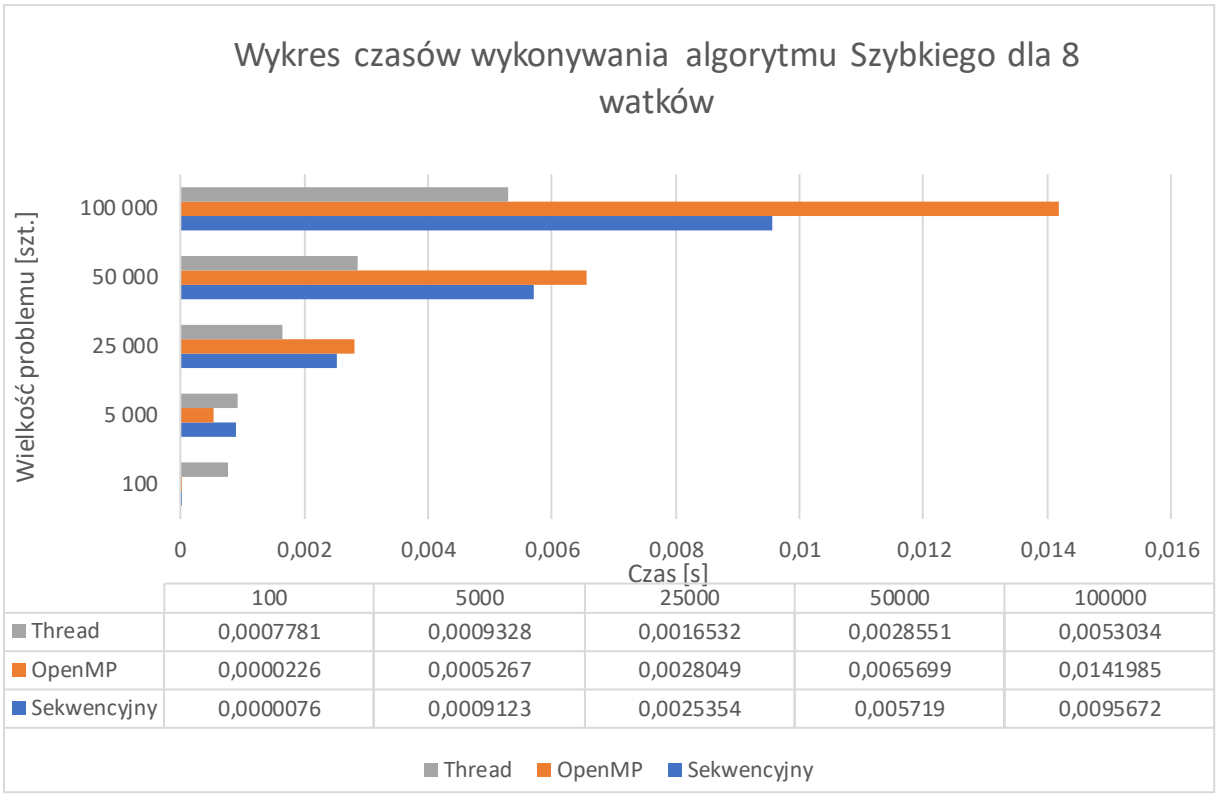


Liczba wątków [szt.]	2	4	8	16
Przez scalanie	0,222149	0,367017	0,366252	0,343847
Przez wstawianie	0,001383	0,001185	0,0009003	0,0012138

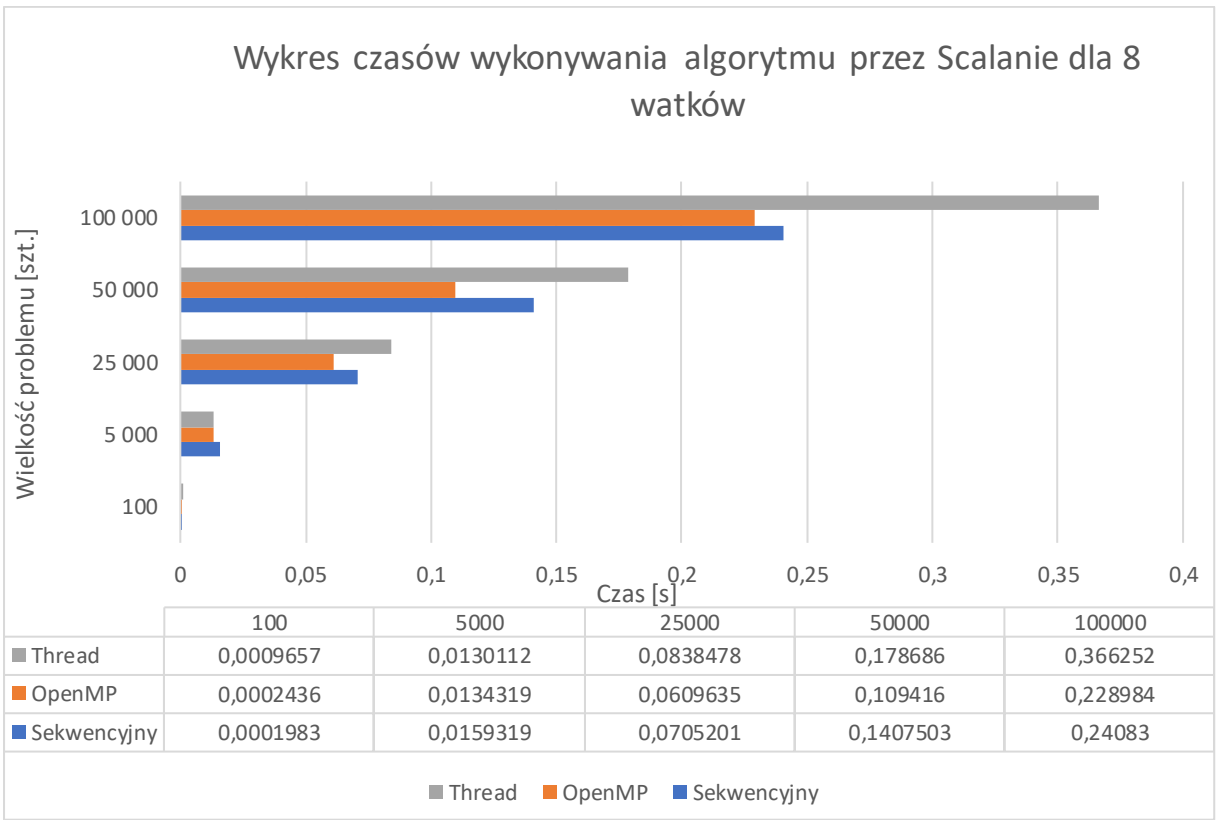
Wykres czasów wykonywania algorytmu Bąbelkowego dla 8 watków



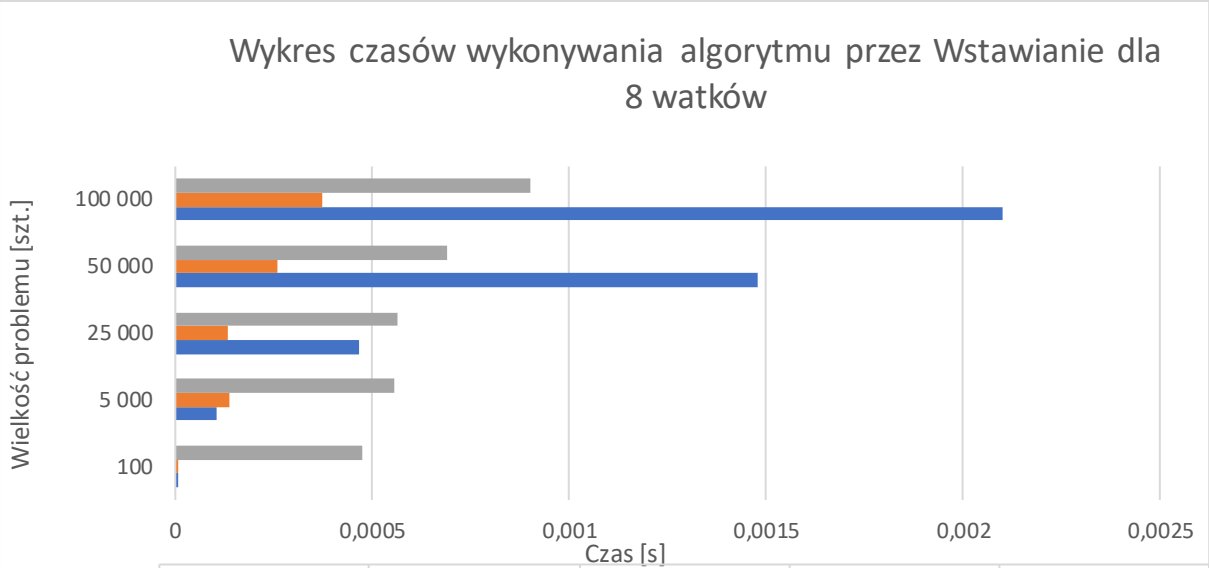
	100	5000	25000	50000	100000
Thread	0,0011183	0,0067885	0,170484	0,669094	2,90395
OpenMP	0,0010032	0,0586758	1,37532	5,61599	26,3034
Sekwencyjny	0,000069	0,186061	4,64213	19,4884	79,9921



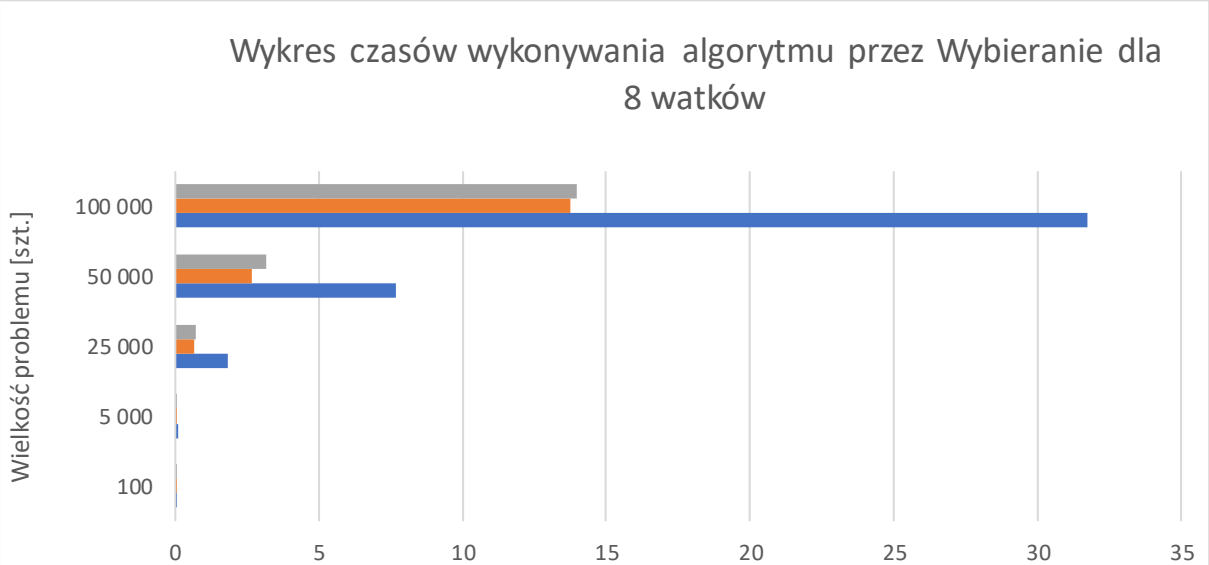
	100	5000	25000	50000	100000
Thread	0,0007781	0,0009328	0,0016532	0,0028551	0,0053034
OpenMP	0,0000226	0,0005267	0,0028049	0,0065699	0,0141985
Sekwencyjny	0,0000076	0,0009123	0,0025354	0,005719	0,0095672



	100	5000	25000	50000	100000
Thread	0,0009657	0,0130112	0,0838478	0,178686	0,366252
OpenMP	0,0002436	0,0134319	0,0609635	0,109416	0,228984
Sekwencyjny	0,0001983	0,0159319	0,0705201	0,1407503	0,24083



	100	5000	25000	50000	100000
Thread	0,0004761	0,000555	0,0005629	0,0006919	0,0009003
OpenMP	0,0000061	0,0001369	0,0001331	0,0002594	0,000374
Sekwencyjny	0,0000076	0,0001028	0,000465	0,0014771	0,0020993



	100	5000	25000	50000	100000
Thread	0,0005167	0,0405552	0,70781	3,16	13,982
OpenMP	0,0004584	0,0287132	0,633876	2,67622	13,7242
Sekwencyjny	0,0000344	0,0735086	1,84262	7,67891	31,7511

Procentowy udział części szeregowej i równoległej

Procentowy udział części szeregowej i równoległej został oszacowany i przedstawiony dla każdego algorytmu w tabeli 5.2.

Tabela 5.2 Procentowy udział części szeregowej i równoległej

Algorytm	Część szeregowa [%]	Część równoległa [%]
Bąbelkowy	40%	60%
Szybki	25%	75%
Przez scalanie	20%	80%
Przez wstawianie	30%	70%
Przez wybieranie	30%	70%

Prawo Amdahla

$$S_p = \frac{1}{(1-f) + \frac{f}{p}}$$

gdzie:

- S_{max} - maksymalne przyspieszenie
- f - frakcja czasu wykonania, która nie może być zrównoleglona (część sekwencyjna)
- p - liczba procesorów lub wątków

Tabela 5.3 Wyniki prawa Amdahla dla wartości maksymalnych (tzn. 18 wątków) oraz rozmiarze problemu równym 100000

Technologia	Algorytm	Wartość Prawa Amdahla
OpenMP	Bąbelkowy	2,85
Thread		55,9
OpenMP	Szybki	0,22
Thread		1,33
OpenMP	Przez scalanie	0,68
Thread		0,48
OpenMP	Przez wstawianie	4,69
Thread		1,43
OpenMP	Przez wybieranie	1,98
Thread		2,38

Przyśpieszenie na podstawie danych empirycznych

Przyśpieszenie (S) można obliczyć jako stosunek czasu wykonania programu na pojedynczym wątku (T_1) do czasu wykonania programu na wielu wątkach (T_p), gdzie p to liczba wątków:

$$S = \frac{T_1}{T_p}$$

Wyniki dla każdego z algorytmów dla różnych wielkości problemu oraz różnej liczbie wątków zostały przedstawione w tabeli 5.1 w wersji czasowej (s) oraz wersji procentowej (%).

Próg opłacalności

Opłacalność zrównoleglenia problemu została oszacowana dla każdego algorytmu oraz każdej technologii w tabeli 5.4 na podstawie wyników przedstawionych w tabeli 5.1.

Tabela 5.4 Wyniki oszacowania progu opłacalności

Technologia	Algorytm	Liczba wątków	Próg opłacalności (wielkość problemu)
OpenMP	Bąbelkowy	2	1000
Thread		2	500
OpenMP	Szybki	2	5000
Thread		2	65000
OpenMP	Przez scalanie	2	4000
Thread		2	4000
OpenMP	Przez wstawianie	2	3500
Thread		2	35000
OpenMP	Przez wybieranie	2	2000
Thread		2	15000

Ziarnistość problemu

Ziarnistość problemu odnosi się do stosunku pracy równoległej do pracy szeregowej w algorytmie. Mówi o tym, jak duże są kawałki pracy, które można wykonywać równolegle w porównaniu do kawałków pracy, które muszą być wykonywane szeregowo. Istnieją dwa rodzaje ziarnistości:

- Gruboziarnista: Duże kawałki pracy są wykonywane równolegle.
- Drobnoziarnista: Małe kawałki pracy są wykonywane równolegle.

Wyznaczanie ziarnistości problemu wymaga analizy kodu i zrozumienia, jak długo trwają różne sekcje programu w kontekście ich potencjału do równoległego wykonania. Tabela 5.5 przedstawia szczegółową analizę ziarnistości dla każdego z pięciu algorytmów.

Tabela 5.5 Ziarnistość problemu

Algorytm	Wartość ziarnistości
Bąbelkowy	**Drobnoziarnisty** - Sekcje krytyczne sprawiają, że efektywne zrównoleglenie jest trudne, a znacząca część czasu jest spędzana w sekcjach, które muszą być wykonywane szeregowo.

Szybki	**Gruboziarnisty** - Znaczna część pracy może być równoległa dzięki rekursywnemu podziałowi problemu, ale funkcja ‘partition’ pozostaje szeregowa.
Przez scalanie	**Gruboziarnisty** - Większość pracy może być równoległa dzięki rekursywnemu podziałowi problemu, ale funkcja ‘merge’ pozostaje szeregowa.
Przez wstawianie	**Drobnoziarnisty** - Sekcje krytyczne w pętli wewnętrznej znacznie ograniczają efektywność zrównoleglenia. Większość pracy jest wykonywana w sekcji równoległej, ale konieczność synchronizacji zmniejsza korzyści.
Przez wybieranie	**Drobnoziarnisty** - Sekcje krytyczne w pętli wewnętrznej znacznie ograniczają efektywność zrównoleglenia. Większość pracy jest wykonywana w sekcji równoległej, ale konieczność synchronizacji zmniejsza korzyści.

6. WNIOSKI

Najlepsze rezultaty czasowe uzyskano przy wykorzystaniu biblioteki **C++ Thread Library**, szczególnie dla dużych wartości problemu. Natomiast wydajność sekwencyjna okazała się słabsza w porównaniu do **OpenMP** i **C++ Thread Library**. Większość testowanych wielkości problemu powyżej 100 potwierdziły wolniejsze tempo działania algorytmu sekwencyjnego, co podkreśla korzyści płynące z równoległego przetwarzania.

Na podstawie analizy danych empirycznych zaobserwowano przyspieszenie S w przedziale od 0,22 do ponad 55,9 dla różnych kombinacji wielkości problemu i liczby wątków. Najlepsze wyniki przyspieszenia osiągnięto dla wysokich wielkości problemu i większej liczby wątków.

Maksymalne przyspieszenie, obliczone na podstawie prawa Amdahla, wyniosło około 55,9 dla 18 wątków. Jest to wartość teoretyczna, która wskazuje na istnienie ograniczeń związanych z częścią sekwencyjną algorytmu. Dane empiryczne pokazały, że przyspieszenie (S) osiągnęło około 55,9 dla 18 wątków i wielkości problemu 50000, mieszcząc się tym samym w przewidywanych maksymalnych wartościach według prawa Amdahla.

Prognozowany próg opłacalności zrównoleglenia problemu dla każdego z algorytmów jest inny, najmniejsza próg oszacowano dla algorytmu bąbelkowego z wykorzystaniem technologii Thread C++ i wynosi on 500, natomiast największy próg oszacowano dla algorytmu szybkiego również w technologii Thread C++, wynosi on 65000.

Podsumowując, przeprowadzone badania potwierdziły skuteczność zrównoleglania problemu sortowania przy użyciu różnych technologii równoległego przetwarzania. Wartości przyspieszenia zbliżają się do teoretycznych maksymalnych wartości, co podkreśla efektywność zastosowanych technik równoległego przetwarzania.

7. BIBLIOGRAFIA

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", The MIT Press, 2009.

2. OpenMP Documentation, https://www.openmp.org/resources/openmp-compilers-tools/

3. Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley, 2013.

4. Herb Sutter, Andrei Alexandrescu, "C++ Coding Standards: 101 Rules, Guidelines, and Best Practices", Addison-Wesley, 2004.