POLITECHNIKA ŚLĄSKA W GLIWICACH

WYDZIAŁ ELEKTRYCZNY

Katedra Energoelektroniki, Napędu Elektrycznego i Robotyki

PROJEKT INŻYNIERSKI

Równoległe algorytmy sortowania

Parallel sorting algorithms

Student: **Marek Jacek TOMCZYK**
Nr albumu: 265698

Studia: Stacjonarne I stopnia
Kierunek: Informatyka
Specjalność: Informatyka w systemach elektrycznych

Promotor: dr inż. Marcin POŁOMSKI

GLIWICE 2020

Streszczenie pracy

Wykonana praca została zrealizowana jako projekt inżynierski na wydziale Elektrycznym Politechniki Śląskiej w Gliwicach. Celem projektu jest analiza porównawcza wybranych równoległych algorytmów sortowania z wykorzystaniem języka C++ w standardzie C++17.

W pierwszym rozdziale określono cel i zakres projektu. Drugi rozdział opisuje wybrane równoległe algorytmy sortowania wraz z przykładową implementacją w języku C++. W kolejnym rozdziale przedstawiono wybrane zagadnienie implementacyjne. Rozdział czwarty to eksperymenty obliczeniowe – opisano w nim wyniki przeprowadzonych testów aplikacji. Ostatni rozdział to podsumowanie pracy, wnioski końcowe oraz możliwości rozwoju projektu. Utworzona aplikacja jest narzędziem do badania czasów pracy dla wybranego algorytmu sortowania równoległego.

Słowa kluczowe: algorytm, sortowanie, wielowątkowość, C++

Abstract

Performed thesis has been carried out as an engineering project at faculty of Informatics at Silesian University of Technology. The goal of the project is comparative analysis of paralel sorting algorithms using the C++17 programming language.

First part of the theis is Introduction, where the purpose and scope of the project was specified. Second part is Specification of selected sorting algorithms, where are descibed considered algorithms. Next part is Selected aspects of implementation of the program, where are shown issues related to application implementation and problems encountered during its implementation. Simulation effects is the fourth part of the thesis, where are descibed effects of performer investiagion and the last part is Conclusion where are presented possibilities of developing of application and the summation of the project.

Keywords: algorithm, sorting, multithreading, C++

SPIS TREŚCI

Rozdział 1 - Wstęp

1.1 – Cel projektu i zakres projektu

Praca *Równoległe algorytmy sortowania* została zrealizowana w ramach projektu inżynierskiego na kierunku Informatyka. Celem projektu jest poszerzenie wiedzy z zakresu algorytmiki, analiza porównawcza wybranych równoległych algorytmów sortowania, a także praktyczne wykorzystanie mechanizmów jakie oferuje standard języka C++ takie jak natywne wątki, system plików czy funkcje lambda. W ramach projektu opracowano aplikację, w której zaimplementowano wybrane oraz opisane w pracy równoległe algorytmy sortowania. Przy użyciu tego oprogramowaniu wykonano eksperymenty obliczeniowe, na podstawie których dokonano analizy porównawczej działania tych algorytmów pod kątem czasu potrzebnego na posortowanie różnych zbiorów danych. Podczas symulacji zmieniać będzie się algorytm sortujący, rozmiar zestawu danych do posortowania oraz liczba wątków biorąca udział w procesie sortowania.

1.2 – Układ pracy

W następnym rozdziale przedstawione zostaną podstawowe informacje dotyczące tematyki projektu, takie jak definicja algorytmu, procesu sortowania, wątku czy złożoności obliczeniowej, a także opis wybranych równoległych algorytmów sortowania wraz z ich przykładową implementacją w języku C++. Kolejne rozdziały przedstawiają założenia funkcjonalne programu, architekturę obiektową aplikacji, zastosowane mechanizmy ze standardu C++17 wraz z przykładami użycia. W dalszych rozdziałach zostaną opisane wyniki eksperymentów obliczeniowych, wnioski oraz pomysły na dalszy rozwój aplikacji.

Rozdział 2 – Wybrane algorytmy sortowania

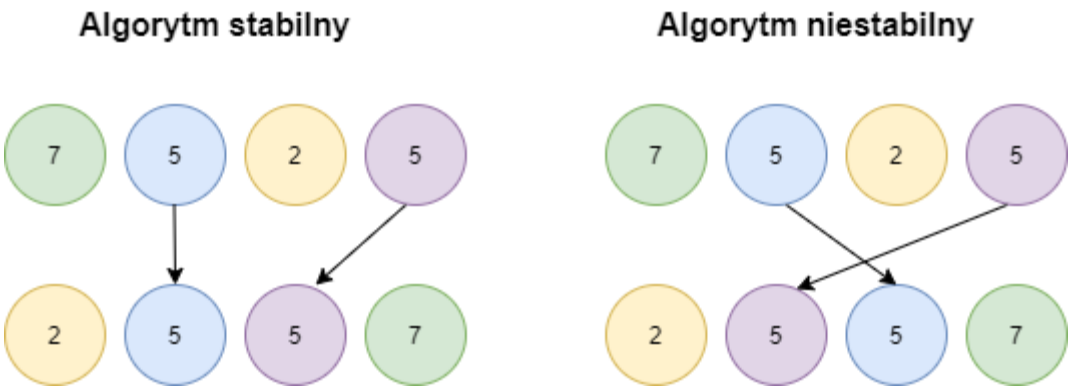
W niniejszym rozdziale zostaną przedstawione wybrane algorytmy sortowania. Aby w pełni zrozumieć jak działają opisane niżej algorytmy należałoby na początku zdefiniować czym jest algorytm oraz na czym polega proces sortowania. Temat projektu dotyczy równoległych algorytmów sortowania, a więc ważnymi pojęciami są również proces i wątek oraz różnice między nimi, których definicje znajdują się poniżej.

Algorytm *(ang. algorithm)* – jest to zbiór jasno określonych kroków prowadzących do wykonania zadania. Jak opisuje w swojej książce *Thomas H. Cormen „Algorytmy bez tajemnic”* [2] przykłady algorytmów można znaleźć w życiu codziennym np. proces mycia zębów czy dojazdu do pracy, jednak różnią się one od algorytmów komputerowych, które wymagają na tyle precyzyjnego opisu, że potrafi je wykonać komputer. Autor w swojej książce określa również wymagania algorytmów komputerowych:

- algorytm mając dane do problemu, powinien zawsze rozwiązać go poprawnie,
- algorytm rozwiązując problem powinien oszczędnie zużywać zasoby obliczeniowe.

Sortowanie *(ang. sorting)* – jest to proces, który polega na uporządkowaniu zbioru obiektów w określonym porządku względem pewnych cech charakterystycznych dla każdego z elementów tego zbioru. Sortowanie jest jednym z podstawowych problemów informatycznych, a swoje zastosowanie znajduje np. w serwisach internetowych w celu prezentacji danych dla użytkownika czy ułatwienia późniejszego wyszukiwania konkretnych elementów posortowanego zbioru. Algorytmy sortowania dzielimy na **stabilne** i **niestabilne**. Jak można zauważyć na **Rysunku 1** algorytmy, które dla

elementów o tej samej wartości zachowują w tablicy końcowej kolejność tablicy wejściowej, nazywamy algorytmami stabilnymi.



Rysunek 1. Graficzne ukazanie czym jest stabilność algorytmu (źródło: opracowanie własne)

Złożoność obliczeniowa algorytmu – jest określeniem ilości zasobów niezbędnych do wykonania algorytmu. Wyróżniamy złożoność czasową oraz złożoność pamięciową. Ilość potrzebnych zasobów może się różnić w zależności od danych wejściowych. Złożoność obliczeniowa nie zależy zwykle od rozmiaru danych, a wręcz może się drastycznie różnić dla zbiorów o tym samym rozmiarze. Często stosowanymi sposobami określania złożoności obliczeniowej algorytmu są:

- złożoność pesymistyczna – czyli rozpatrywanie najgorszych przypadków danych wejściowych,
- złożoność oczekiwana – czyli uśrednienie wszystkich możliwych przypadków.

Wątek *(ang. Thread)* [1] – nazywany również procesem lekkim *(ang. Lightweight Process),* jest podstawową jednostką wykorzystania procesora, w której skład wchodzą:

- licznik rozkazów,
- zbiór rejestrów,

- obszar stosu.

Wątek współdzieli wraz z innymi, równorzędnymi wątkami sekcję kodu, danych oraz zasoby systemu operacyjnego, takie jak otwarte pliki i sygnały. Inaczej rzecz ujmując, jest to fragment programu wykonywany współbieżnie w obrębie jednego procesu – czyli instancji programu o unikalnym identyfikatorze tzw. PID *(ang. Process Identifier).* Z uwagi na to, że wszystkie wątki działające w danym procesie współdzielą między sobą przestrzeń adresową oraz inne struktury systemowe, wymagają tym samym mniej zasobów do działania i szybszy jest czas ich tworzenia, a także mogą komunikować się między sobą.

2.1 – Algorytm Bitonic sort

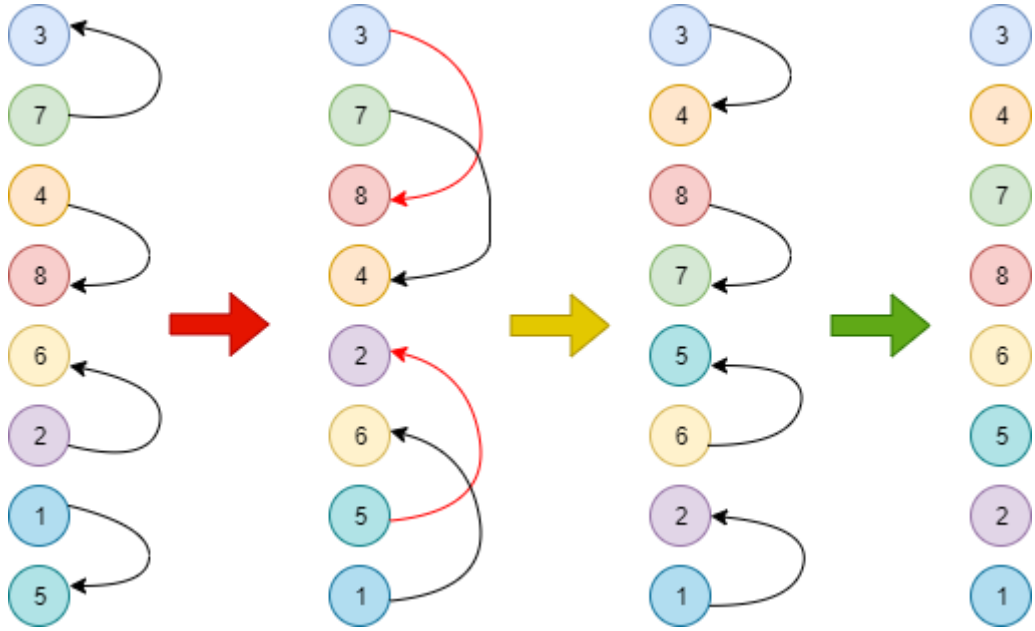
Bitonic sort to równoległy algorytm sortujący. Złożoność algorytmu wynosi $O(n\,log^2 n)$, a więc jest większa od popularnych algorytmów takich jak sortowanie przez scalanie lub sortowanie szybkie, które zostaną szerzej opisane w dalszej części pracy. Algorytm Bitonic sort jest jednak łatwiejszy w implementacji wielowątkowej ponieważ zawsze porównuje elementy w predefiniowanej sekwencji, która nie zależy od danych. Ta zależność umożliwia implementacje w macierzy procesorów sprzętowych i równoległych. Aby zrozumieć jak działa algorytm, w pierwszej kolejności należy określić czym jest sekwencja bitoniczna (ang. Bitonic sequence).

Sekwencją bitoniczną nazywamy ciąg, w którym elementy ułożone są najpierw rosnąco, a później malejąco lub odwrotnie. Innymi słowy ciąg nazywamy bitonicznym, gdy istnieje index i, gdzie $0 \leq i \leq n-1$ taki, że

$$x_o \leq x_1 \leq \cdots x_i \geq x_{i+1} \geq \cdots \geq x_{n-1}$$

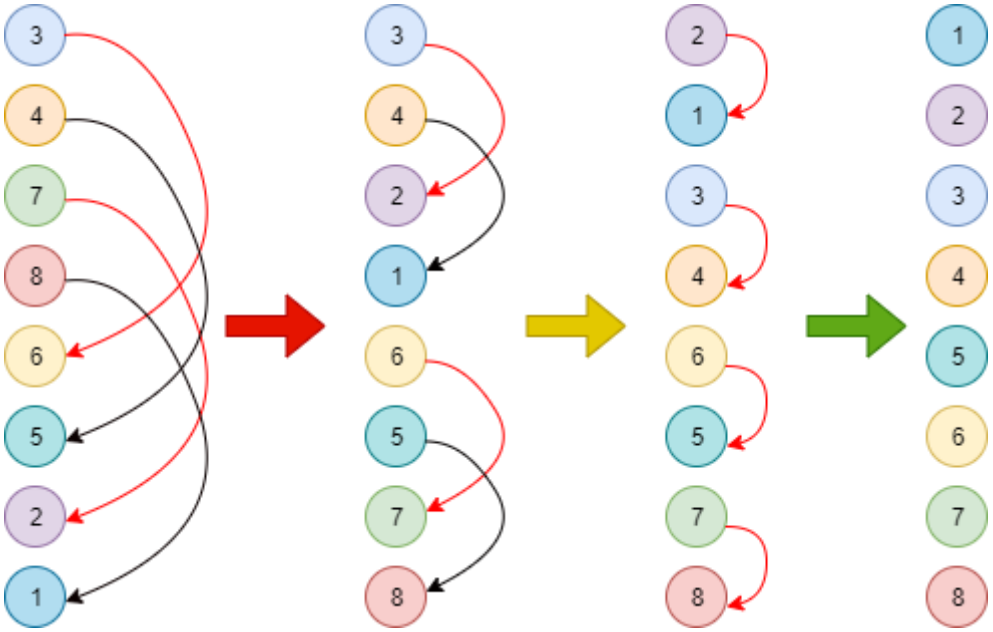
Na **rysunku 2** przedstawiono proces przekształcania losowego ciągu w sekwencję bitoniczną. Zaczynamy od utworzenia 4-elementowych sekwencji bitonicznych z kolejnych 2-elementowych sekwencji, które sortujemy naprzemiennie rosnąco oraz malejąco. W kolejnym kroku łączymy dwie pary tworząc 4-elementową sekwencję bitoniczną. Następnie rozważamy dwie 4-elementowe sekwencje bitoniczne, sortując

jedną w kolejności rosnącej, a drugą w kolejności malejącej. Proces powtarzamy aż do uzyskania sekwencji bitonicznej.



Rysunek 2. Proces przekształcania losowych danych w sekwencję bitoniczną (źródło: opracowanie własne)

Proces sortowania bitonicznego sprowadza się do dwóch kroków. Pierwszym z nich jest utworzenie sekwencji bitonicznej opisany powyżej – otrzymujemy wtedy dwa ciągi, gdzie jeden jest posortowany w porządku rosnącym, a drugi w porządku malejącym. Następnym krokiem jest połączenie dwóch posortowanych podciągów w jeden. W pierwszym kroku porównujemy każdy element z pierwszej tablicy, z odpowiadającym mu indeksem, elementem z drugiej tablicy – innymi słowy porównujemy pierwszy element z pierwszej tablicy, z pierwszym elementem z drugiej tablicy itd. Zamieniamy elementy miejscami, jeżeli element z pierwszej tablicy jest mniejszy. Po tym kroku otrzymujemy dwie sekwencje bitoniczne, każda o długości $n/2$, gdzie n jest liczbą elementów ciągu wejściowego. W pierwszej sekwencji wszystkie elementy są mniejsze od elementów z drugiej sekwencji bitonicznej. Powtarzamy ten proces dopóki nie uzyskamy n sekwencji bitonicznych, każda o długości równej 1. Cały proces został graficznie przedstawiony na **rysunku 3**.



Rysunek 3. Graficzne przedstawienie działania algorytmu Bitonic sort (źródło: opracowanie własne)

Ważnym podkreślenia ograniczeniem algorytmu Bitonic sort jest liczba danych do posortowania – musi być potęgą liczby 2. Wynika to właśnie z porównywania elementów w predefiniowanej sekwencji, która jest niezależna od danych. Bitonic sort jest równoległym algorytmem sortującym, jednak nie istnieje jedna, podstawowa implementacja równoległa. Dobrym miejscem do zrównoleglenia wydaje się operacja porównania oraz zamiany elementów. Sekwencyjna implementacja algorytmu Bitonic sort znajduje na listingu **Algorytm 2.1**.

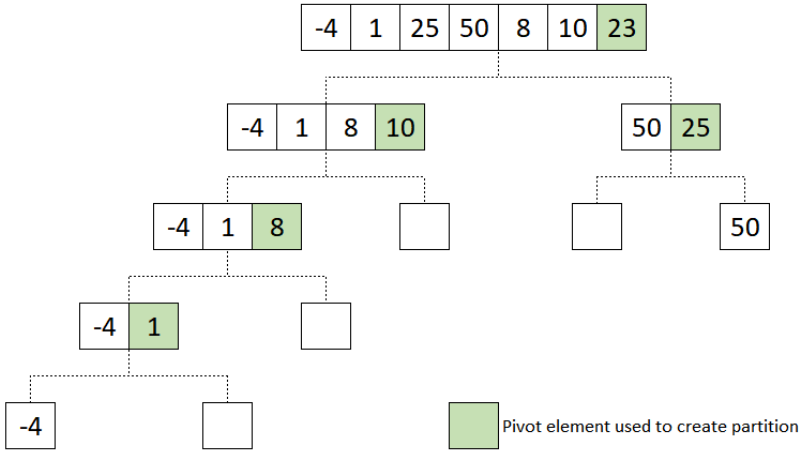
ALGORYTM 2.1: ALGORYTM BITONIC SORT

```
void bitonicMerge(int arr[], int low, int size, bool asc)
{
   if (size > 1)
   {
      int k = size / 2;
      for (int i = low; i < low + pivot; i++)
      {
         if (asc == (arr[i] > arr[i + pivot]))
         {
            int tmp = arr[i];
            arr[i] = arr[i + pivot];
            arr[i + pivot] = tmp;
         }
      }

      bitonicMerge(arr, low, pivot, asc);
      bitonicMerge(arr, low + pivot, pivot, asc);
   }
}

void bitonicSort(int arr[], int low, int size, bool asc)
{
   if (size > 1)
   {
      int pivot = size / 2;

      bitonicSort(arr, low, pivot, true);
      bitonicSort(arr, low + pivot, pivot, false);

      bitonicMerge(arr, low, size, asc);
   }
}
```

2.2 – Algorytm Quicksort

Quicksort jest jednym z najpopularniejszych algorytmów sortowania działających na zasadzie *„dziel i zwyciężaj”* co oznacza w praktyce dzielenie problemu na mniejsze, przypominające problem początkowy, podproblemy. Te podproblemy rozwiązywane są rekurencyjnie, po czym wyniki są łączone w celu rozwiązania pierwotnego problemu. **Rysunek 4** przedstawia krok po kroku schemat postępowania algorytmu. Na zielono został zaznaczony element osiowy *(ang. pivot)*, który rozdziela zestaw danych na dwie części, gdzie jeden podciąg zawiera elementy mniejsze od elementu osiowego, a drugi elementy większe od niego. Proces ten jest powtarzany

aż do uzyskania pojedynczego elementu, ponieważ jednoelementowy podciąg nie wymaga sortowania.



Rysunek 4. Schemat działania algorytmu Quick sort [6]

Złożoność algorytmu zależy w głównej mierze od wyboru elementu osiowego – jeżeli wybory są zrównoważone złożoność jest równa $O(n \log n)$, w przeciwnym wypadku złożoność może wynosić $O(n^2)$. Element osiowy może być elementem środkowym, ostatnim, pierwszym lub wyznaczonym za pomocą jakiegoś algorytmu dostosowanego do danych wejściowych. Sortowanie szybkie nie wymaga dodatkowej pamięci potrzebnej do przechowywania sortowanych danych, jednak wymaga pamięci potrzebnej na rekurencyjne wywołania funkcji, a rozmiar tej pamięci jest zależny, podobnie jak czas wykonania, zależeć będzie od głębokości stosu wywołań. Istnieje wiele modyfikacji algorytmu np. Sortowanie szybkie z ograniczeniem rekursji, czy Quicksort działający „w miejscu” [12]. W aplikacji wykorzystano klasyczną wersję algorytmu, którego przykładową implementację przedstawiono na listingu **Algorytm 2.2**.

ALGORYTM 2.2: ALGORYTM QUICK SORT

```
void quickSort(int arr[], int left, int right)
{
   if (left < right)
   {
      int pivot = arr[right];
      int i = (left - 1);

      for (int j = left; j <= (right - 1); j++)
      {
         if (arr[j] <= pivot)
         {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }

      int temp = arr[i + 1];
      arr[i + 1] = arr[right];
      arr[right] = temp;

      int partIndex = (i + 1);
      quickSort(arr, left, partIndex - 1);
      quickSort(arr, partIndex + 1, right);
   }
}
```

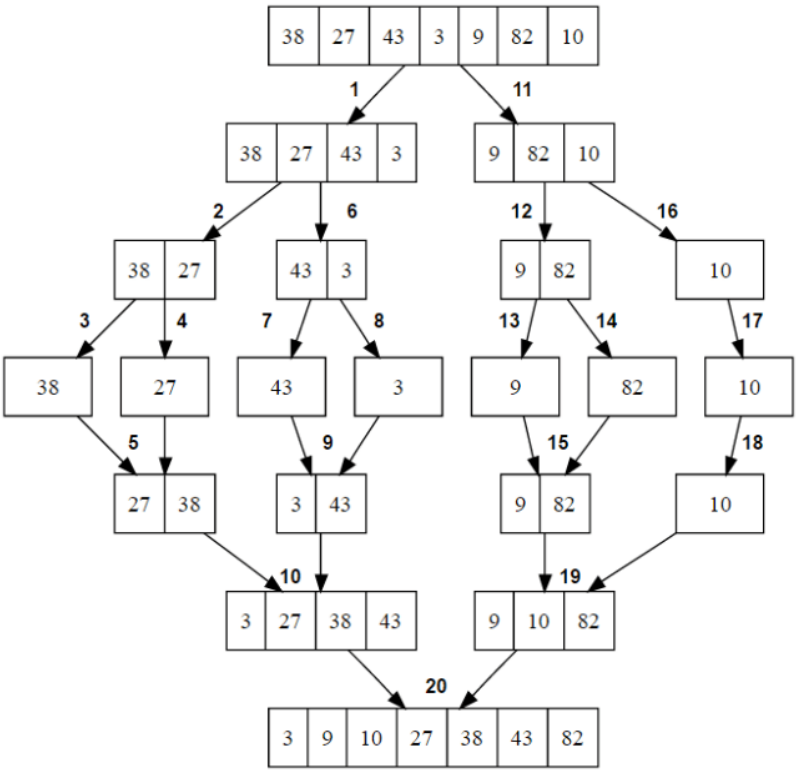
2.3 – Algorytm Merge sort

Sortowanie przez scalanie *(ang. Merge sort)* – rekurencyjny algorytm sortowania danych z rodziny algorytmów stosujących metodę *„dziel i zwyciężaj”*, który we wszystkich przypadkach ma czas działania wynoszący $O(n \lg n)$. Odkrycie algorytmu przypisuje się Johnowi von Neumannowi. Wyróżnić można dwa podstawowe kroki działania algorytmu:

1. Podziel ciąg wejściowy na n podciągów, każdy o rozmiarze równym 1,
2. Rekurencyjnie złączaj podciągi tworząc nowe posortowane podciągi dopóki ich ilość jest większa od 1.

Podstawową wersję algorytmu sortowania przez scalanie można uprościć poprzez odwrócenie procesu scalania serii. Ciąg danych możemy wstępnie podzielić na n serii długości 1, scalić je tak by otrzymać $\frac{n}{2}$ serii długości 2, scalić je otrzymując $\frac{n}{4}$ serii długości 4 itd. Złożoność obliczeniowa jest taka sama jak w przypadku klasycznym,

jednak dzięki zrezygnowaniu z rekursji oszczędzamy czas oraz pamięć potrzebną na jej obsłużenie.



Rysunek 5. Schemat działania algorytmu Merge sort [7]

Na **Rysunku 5** przedstawiono w proces sortowania tablicy siedmioelementowej przy użyciu algorytmu sortowania przez scalanie w klasycznej wersji. Zaletą algorytmu jest prostota implementacji, wydajność oraz stabilność. Jedyną wadą sortowania przez scalanie jest konieczność dodatkowego obszaru pamięci przechowywujący kopie podciągów do scalenia. Na listingu **Algorytm 2.3** przedstawiono przykładową implementację algorytmu Mergesort.

ALGORYTM 2.3: ALGORYTM MERGE SORT

```
void merge(int * arr, int low, int high, int mid)
{
   int i, j, k, c[50];
   i = low;
   k = low;
   j = mid + 1;
   while (i <= mid && j <= high)
   {
      if (arr[i] < arr[j])
      {
         c[k] = arr[i];
         k++;
         i++;
      }
      else
      {
         c[k] = arr[j];
         k++;
         j++;
      }
   }
   while (i <= mid)
   {
      c[k] = arr[i];
      k++;
      i++;
   }
   while (j <= high)
   {
      c[k] = arr[j];
      k++;
      j++;
   }
   for (i = low; i < k; i++)
   {
      arr[i] = c[i];
   }
}

void mergeSort(int * arr, int low, int high)
{
   if (low < high)
   {
      int mid = (low + high) / 2;
      mergeSort(arr, low, mid);
      mergeSort(arr, mid + 1, high);

      merge(arr, low, high, mid);
   }
}
```

Rozdział 3 – Wybrane zagadnienia implementacji aplikacji

W niniejszym rozdziale opisane zostaną narzędzia wykorzystywane w procesie tworzenia aplikacji oraz analiza wymagań funkcjonalnych. Opisana zostanie również architektura programu wraz z graficznym diagramem skonstruowanym w standardzie UML. W tym rozdziale przedstawione zostaną także problemy implementacyjne wraz z opisem rozwiązania.

3.1 – Analiza wymagań

W oparciu o przyjęte cele, opracowany został zbiór wymagań funkcjonalnych, znajdujący się poniżej:

- aplikacja konsolowa, uruchamiana z linii poleceń wraz z argumentami wejściowymi;

- aplikacja udostępnia usługę sortowania przy pomocy wybranego algorytmu;

- aplikacja pozwala posortować podany na wejście zbiór danych wszystkimi dostępnymi w aplikacji algorytmami;

- oprogramowanie umożliwia sortowanie rosnące lub malejące, zależne od podanego na wejście parametru;

- program na standardowym wyjściu podaje czas potrzebny na realizację sortowania, procesu łączenia plików tymczasowych oraz czas wczytywania i zapisywania danych;

- aplikacja umożliwia sortowanie jedynie liczb całkowitych wczytywanych z pliku binarnego;

3.3 – Opis działania aplikacji

Program uruchamiany jest z linii poleceń wraz z parametrami wejściowymi, które zostały przedstawione poniżej:

- -i [--input] – nazwa pliku wejściowego *(parametr wymagany)*;
- -o [--output] – nazwa pliku wyjściowego *(parametr wymagany)*;
- -t [--thread] – liczba wątków sortujących *(domyślnie 1)*;
- -l [--log] – nazwa pliku, do którego zapisane zostaną logi z przebiegu sortowania;
- -a [--algorithm] – nazwa algorytmu sortującego;
- -c [--comparison] – posortowanie pliku wszystkimi dostępnymi algorytmami;
- -v [--verify] – włączenie sprawdzania poprawności sortowania;
- -h [--help] – wyświetlenie pomocy użytkownikowi.

Aplikacja została napisana w sposób ograniczający zużycie pamięci RAM, tak aby można było jej używać również na słabszych komputerach. Poziom zużycia pamięci zależy od rozmiaru danych wejściowych, jednak jeżeli plik wejściowy przekracza *256MB* to aplikacja nie wczyta większej porcji danych z pliku niż *256MB*. To ograniczenie ma wpływ na prędkość sortowania jednak umożliwia sortowanie bardzo dużych zbiorów danych nie obciążając w znaczny sposób komputera. Istotnym ograniczeniem aplikacji jest również liczba danych wejściowych, mianowicie musi ona być potęgą o podstawie liczby dwa – wynika to z ograniczenia jakie niesie ze sobą opisany w rozdziale drugim algorytm *Bitonic sort*. Takie ograniczenia zostały założone podczas projektowania aplikacji, jednak mogą zostać w przyszłości zmienione, co szerzej opisane zostało w podsumowaniu pracy.

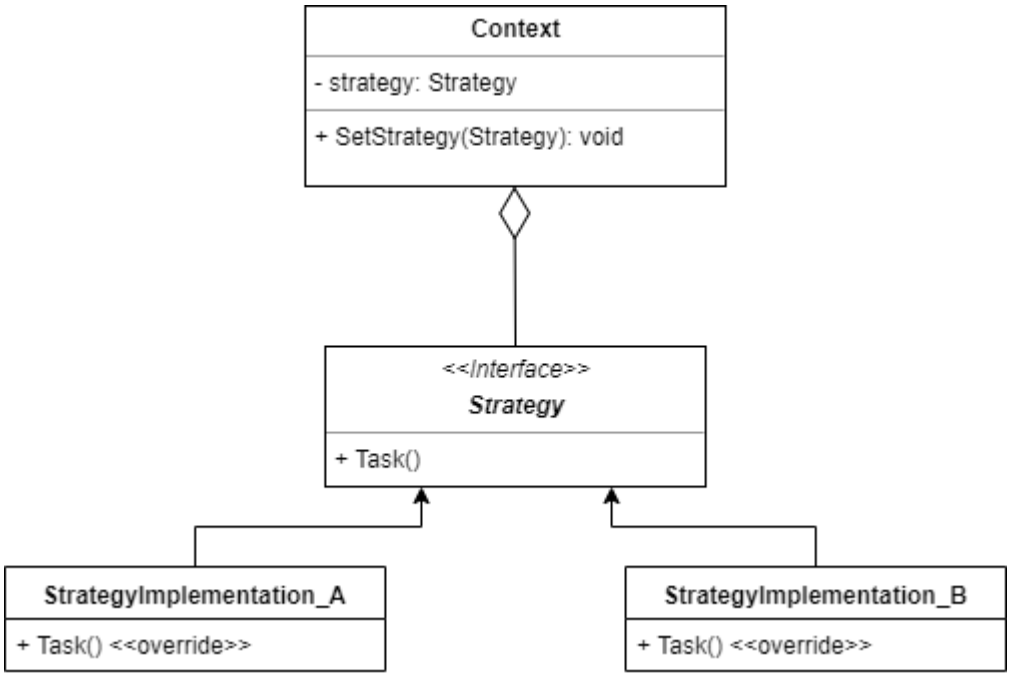
3.4 – Architektura obiektowa aplikacji

Aplikacja została napisana, jak wcześniej wspomniano, w języku C++17, który jest językiem obiektowym. Aplikację zaimplementowano zgodnie z paradygmatami

programowania obiektowego, a podczas tworzenia architektury obiektowej kierowano się zasadami SOLID, które opisują pięć podstawowych założeń programowania obiektowego. Program opisany jest jako zbiór obiektów, które komunikują się ze sobą. Każdy obiekt jest instancją klasy, która z kolei jest definicją obiektu. Każda klasa jest odpowiedzialna za jedną rzecz, otwarta na rozszerzenia ale zamknięta na modyfikacje, zgodnie z zasadami hermetyzacji. W procesie tworzenia aplikacji wykorzystano również znane wzorce projektowe, które określa się jako sprawdzone rozwiązania pewnych znanych problemów implementacyjnych. Wzorzec projektowy nie jest implementacją, a jedynie opisem rozwiązania, który można zastosować w projektach wykorzystujących programowanie obiektowe. W projekcie wykorzystano m. in. wzorce projektowe takie jak:

- **Singleton** – kreacyjny wzorzec projektowy, ograniczający możliwości tworzenia obiektów danej klasy do tylko jednej instancji. Singleton zapewnia także globalny dostęp do stworzonego obiektu. W projekcie swoje zastosowanie znalazł np. w klasie odpowiedzialnej parsowanie parametrów wejściowych.

- **Strategia** – wzorzec z rodziny czynnościowych wzorców projektowych, który definiuje grupę wymiennych zadań i zamyka je w postaci klas. Dzięki strategii mamy możliwość wymiennego stosowania każdego z zadań w trakcie działania aplikacji. Klasa zarządzająca procesem sortowania korzysta z tego wzorca projektowego. Ze strategii skorzystano ze względu na możliwość wymiennego stosowania algorytmów sortujących, a tym samym rozszerzalność aplikacji. Na **rysunku 6** znajduje się diagram UML, który graficznie przedstawia wzorzec projektowy strategia. Abstrakcyjna klasa lub interfejs **Strategy** posiada wirtualną metodę **Task**, którą klasy pochodne implementują. Klasa **Context**, która jako parametr przyjmuje instancje konkretnej implementacji interfejsu **Strategy**.

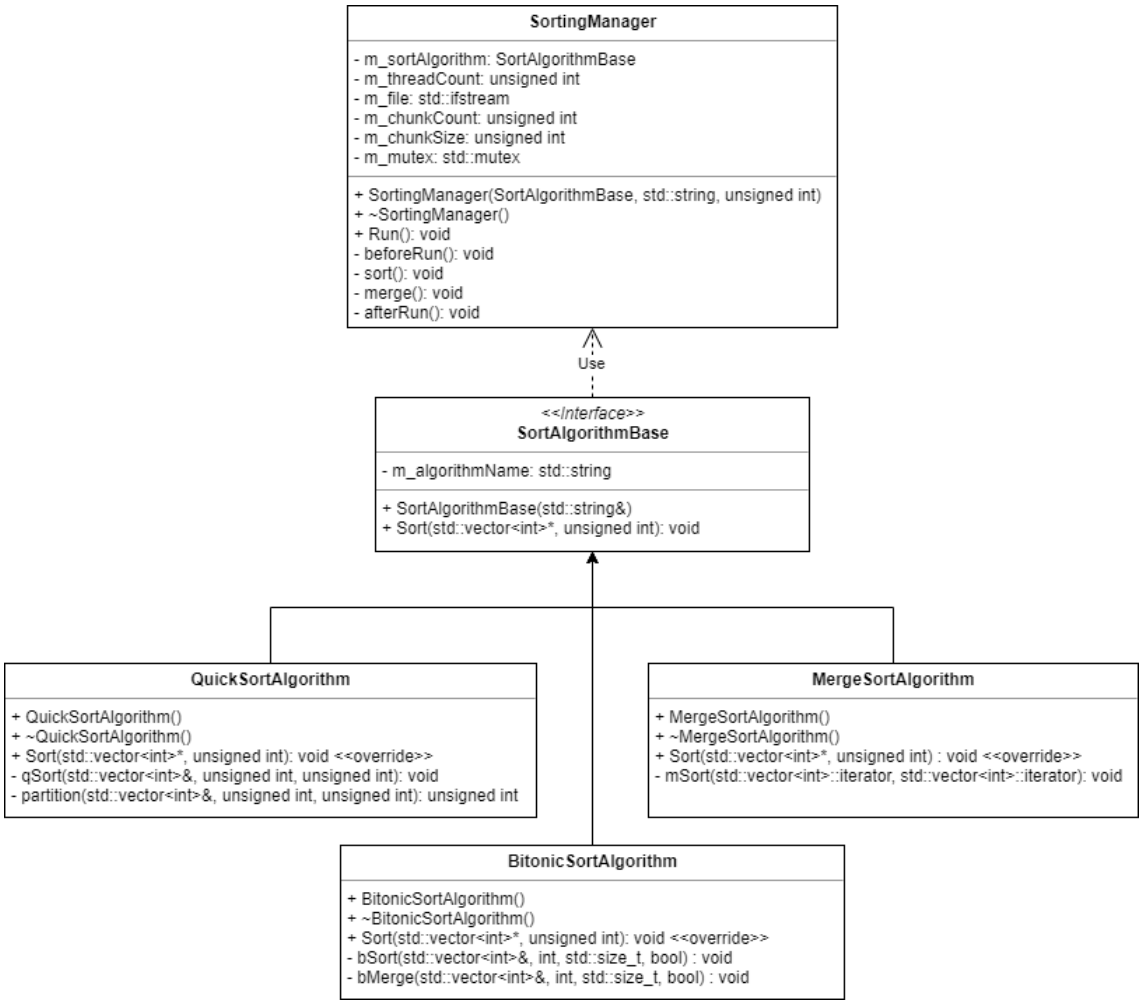


Rysunek 6. Diagram klas wzorca projektowego Strategia

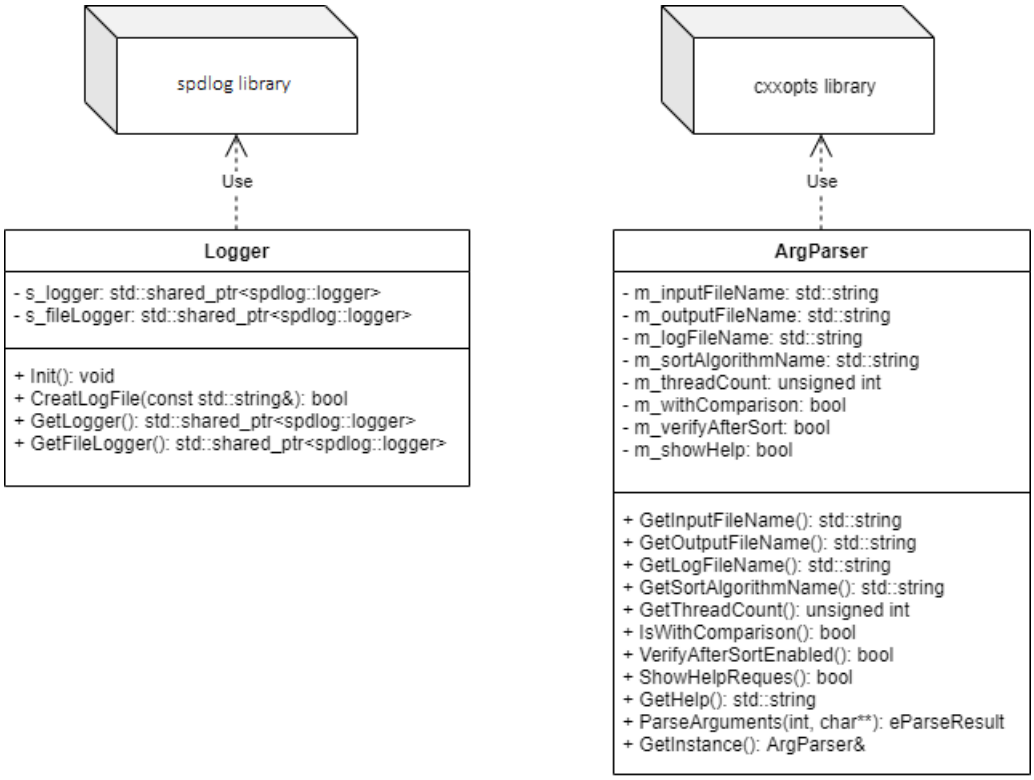
- **Pula obiektów** – wzorzec projektowy polegający na użyciu zainicjowanego zbioru obiektów, które są trzymane i gotowe do użycia – zamiast alokować je i zwalniać na żądanie. Klient pobiera obiekt z puli, wykonuje na nim żądane operacje, po czym, zamiast usuwać obiekt, zwraca go do puli. Pulę obiektów zastosowano w klasie ThreadPool, która zarządza określoną liczbą wątków. Użytkownik puli ma możliwość dodawania zadania do wykonania przez osobny wątek w tle, który jest zwracany do puli i może zostać wykorzystany ponownie dla kolejnych zadań.

Na **rysunku 7** przedstawiona została architektura obiektowa aplikacji. Na załączonym obrazie można zaobserwować klasę typu zarządca – *SortingManager*, która jest odpowiedzialna za sortowanie podanego zbioru danych. Klasa ta posiada inteligentny wskaźnik na obiekt typu *SortAlgorithmBase* ze standardu C++11 – *std::shared_ptr*, który jest klasą opakowującą (*ang. wrapper*) klasyczny wskaźnik. Twórcy języka zastosowali tutaj wzorzec projektowy **RAII** (*ang. Resource Acquisition Is Initialization*) – polega on na tym, że to obiekt sam zarządza zaalokowaną pamięcią, co oznacza, że wszelkie zasoby zaalokowane w klasie muszą zostać zwolnione w chwili niszczenia obiektu. Dodatkowo wskaźnik *std::shared_ptr* posiada licznik referencji, który jest inkrementowany z każdym skopiowaniem obiektu oraz dekrementowany z każdym wyjściem z zakresu. Obiekt jest usuwany tylko i wyłącznie wtedy, gdy licznik

referencji jest równy 0. Klasa *SortingManager* na zewnątrz udostępnia jedynie metodę *SortingManager::Sort*() według konwencji SOLID. Wykorzystując opisany wcześniej wzorzec projektowy Strategia, zapewniono rozszerzalność aplikacji w prosty i nie ingerujący w pozostałą część aplikacji sposób. Dzięki temu w przyszłości aplikacja może zostać rozbudowana o kolejne algorytmy sortujące. Oprócz klas głównych przedstawionych na diagramie z **rysunku 7**, zaimplementowane zostały także klasy poboczne, takie jak *Timer*, *ToolSet* czy szerzej opisana w dalszej części pracy pula wątków, których diagramy znajdują się na **rysunku 8**.



Rysunek 7. Architektura obiektowa aplikacji (źródło: opracowanie własne)



Rysunek 8. Architektura obiektowa aplikacji (źródło: opracowanie własne)

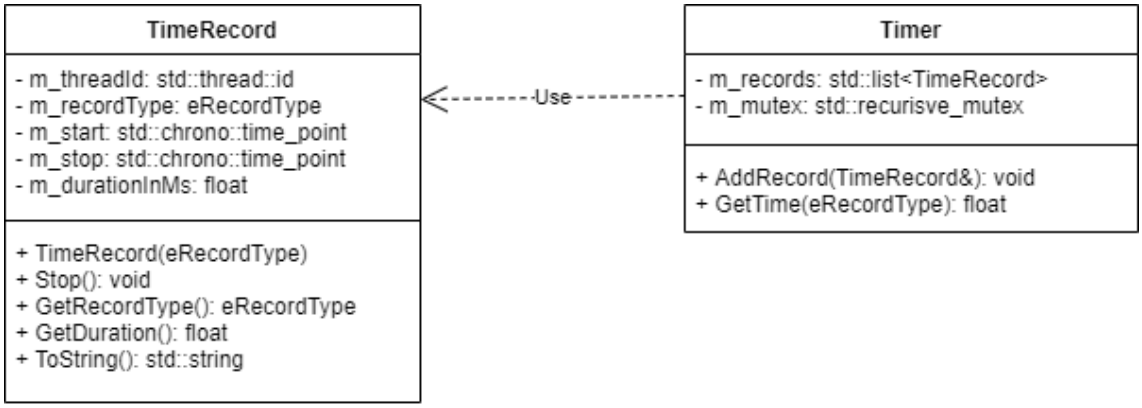
Na **rysunku 8** zostały przedstawione klasy, które stanowią interfejs do zewnętrznych bibliotek wykorzystanych w programie. Aplikacja korzysta z biblioteki służącej do logowania spdlog [9] która umożliwia przy użyciu różnych szablonów *(ang. pattern)* stworzyć swój format logowania. Dostępne typy logów to *trace*, *info*, *warn*, *error* – podział na typy ma ułatwić programiście proces wykrywania błędów w aplikacji. Biblioteka jest świetnie udokumentowana i przede wszystkim wspiera logowanie z wielu wątków. Biblioteka umożliwia wyróżnienie logów kolorami na standardowym wyjściu, a także w łatwy sposób udostępnia opcję logowania do pliku. Klasa *Logger* jest klasą, w której wszystkie pola oraz metody są statyczne. Metoda *Init()* tworzy logger, który loguje na standardowe wyjście programu, natomiast metoda *CreateLogFile()* tworzy logger, który loguje do pliku, którego nazwę przyjmuje jako parametr. Funkcjonalność logowania została zastąpiona makrami. Przykład jednego z dostępnych makr można znaleźć na **Listingu 3.1**.

LISTING 3.1: PRZYKŁAD MAKRA UDOSTĘPNIAJĄCEGO FUKCJONALOŚĆ LOGOWANIA

```
#define LOG_INFO(...)                                  \
{                                                      \
  Logger::GetLogger()->info(__VA_ARGS__);              \
  if ( Logger::GetFileLogger() != nullptr )            \
      Logger::GetFileLogger()->info(__VA_ARGS__);      \
}
```

Klasa *ArgParser* odpowiada za rozpoznanie, inaczej mówiąc – sparsowanie argumentów wejściowych programu. *ArgParser* jest interfejsem dla biblioteki *cxxopts* [10]. W klasie wykorzystano wzorzec projektowy singleton, ponieważ lista parametrów wejściowych programu jest jedna i powinna zostać sparsowana tylko raz oraz dostępna z różnych części programu. Metoda *Parse()* przyjmuje jako argumenty tablicę argumentów wejściowych programu oraz jej długość, która zostaje przeanalizowana. Rozpoznane wartości zostają przypisane do pól prywatnych w klasie, do których dostęp użytkownik klasy ma jedynie poprzez metody dostępowe *(ang. getters)*. Operacja parsowania zwraca rezultat w postaci typu wyliczeniowego, który opisuje czy proces się udał, czy brakuje wymaganych do poprawnego działania programu parametrów wejściowych lub inny błąd zwracany przez bibliotekę *cxxopts*.

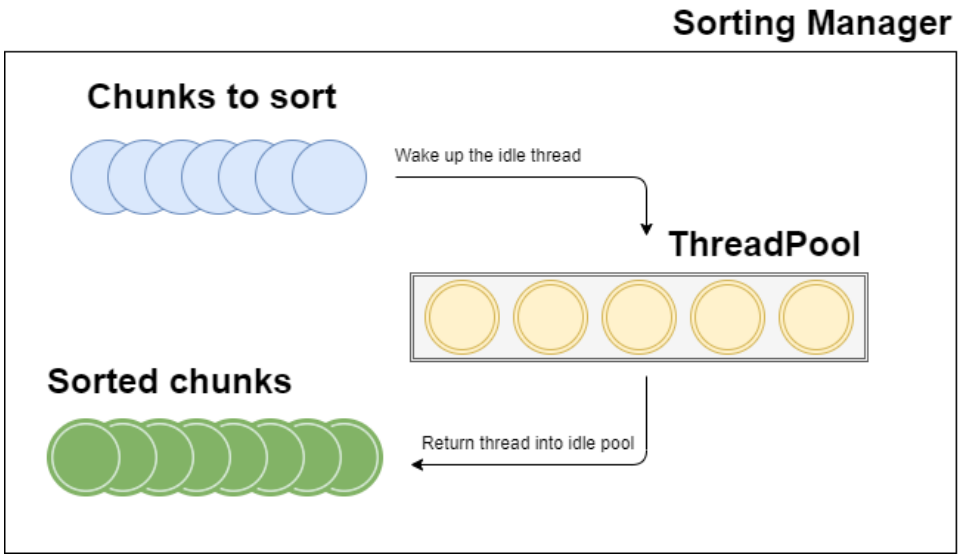
Jednym, z wymienionych na początku pracy, celów projektu jest analiza porównawcza wybranych algorytmów sortowania pod względem czasu potrzebnego na realizacje procesu sortowania. Na potrzeby realizacji tego celu została zaimplementowana klasa *Timer* oraz *TimeRecord*. Klasa *TimeRecord* przedstawiona na **rysunku 9** służy do pomiaru czasu jasno określonych - poprzez typ wyliczeniowy – procesów wykonywanych podczas sortowania zbioru danych. *TimeRecord* korzysta biblioteki *std::chrono*, która została wprowadzona wraz ze standardem C++11. Klasa *TimeRecord* została oparta o *high_resolution_clock*, dzięki czemu pomiary są maksymalnie dokładne. Na **rysunku 9** została przedstawiona również klasa *Timer,* która posiada listę obiektów typu *TimeRecord*. Udostępnia ona jedynie metodę dodania nowego rekordu oraz odczytania czasu według kategorii. Dostęp do listy rekordów został zabezpieczony mutexem rekursywnym, który zabezpiecza przed jednoczesnym dostęp przez wiele wątków.



Rysunek 9. Schemat działania menadżera wątków (źródło: opracowanie własne)

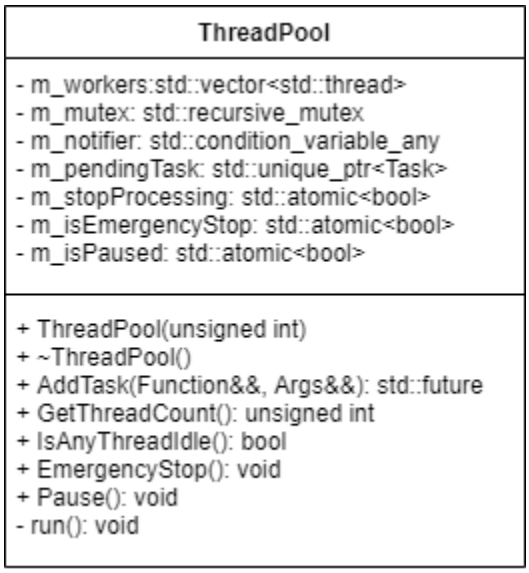
3.6 – Pula wątków

Na potrzeby aplikacji zaimplementowano klasę, która zarządza określoną liczbą wątków - *ThreadPool*. Niektóre języki programowania takie jak Java czy C#, posiadają implementację takiego rozwiązania w swoich bibliotekach standardowych, jednak w języku C++ ma zostać ona wprowadzona wraz ze standardem C++20. Pula wątków to klasa, która posiada kolejkę zadań oraz określoną liczbę wątków wykonawczych. Zadania wkładane są do kolejki oczekujących zadań przez klasę typu producent. W momencie włożenia zadania do kolejki zostaje poinformowany wolny wątek o takim zdarzeniu. Zadanie zostaje wyciągnięte z kolejki i od tej chwili realizowane jest przez osobny wątek. Po skończeniu zadania wątek wraca do puli wątków oczekujących na zadania. Do implementacji menadżera wątków zostały wykorzystane mechanizmy standardu C++17, takie jak: *std::unique_lock*, *std::atomic* oraz *std::feature*. Na **rysunku 10.** przedstawiono schemat działania puli wątków w projekcie. Tutaj zadaniem jest porcja danych do posortowania (*ang. chunk*).



Rysunek 10. Schemat działania menadżera wątków (źródło: opracowanie własne)

Jak wspomniano wcześniej, klasyczna implementacja puli wątków opiera się na kolejkowaniu zadań. Menadżer wątków został opracowany w sposób ograniczający zużycie zasobów – jeżeli wszystkie wątki są zajęte to tylko jedna porcja danych oczekuje w kolejce. Proces sortowania został oparty o mechanizm poolingu, czyli odpytywania, znany np. w protokole HTTP, gdzie klient odpytuje serwer czy jego żądanie zostało już wykonane. W tym przypadku główny wątek odpytuje pulę wątków co 10 ms, czy któryś z wątków zakończył pracę – jeżeli tak, to znaczy, że porcja, która czekała w kolejce została zużyta i należy wczytać kolejną porcję danych do posortowania. Takie podejście pozwala zminimalizować dane trzymane w pamięci RAM. Na **rysunku 11** znajduje się diagram UML przedstawiający klasę *ThreadPool*. Jak można zauważyć klasa udostępnia jedynie funkcjonalności dodania nowego zadania do wykonania oraz zatrzymania pracy puli, a także wspomniany wcześniej mechanizm poolingu w postaci metody *IsAnyThreadIdle()*, która zwraca typ *bool*. Na **Listingu 3.2** można znaleźć implementację metody odpowiedzialnej za dodanie nowego zadania do kolejki – wykorzystane zostały w niej wszystkie możliwości nowych standardów języka C++ takie jak szablony, typy generyczne, wskaźniki inteligentne i wiele innych. Typ zwracany przez metodą *AddTask* to *std::future*, który zapewnia dostęp do wyniku operacji asynchronicznych.



Rysunek 11. Diagram UML klasy ThreadPool (źródło: opracowanie własne)

LISTING 3.2: IMPLEMENTACJA DODAWANIA ZADANIA DO PULI WĄTKÓW

```
template < typename Function, typename... Args >
auto AddTask(Function&& fun, Args&& ... args)
   -> std::future< typename std::result_of< Function(Args...) >::type >
{
   using returnType = typename std::result_of< Function(Args...) >::type;

   auto task = std::make_shared< std::packaged_task< returnType() > >
   (
      std::bind
      (
         std::forward< Function >(fun),
         std::forward< Args >(args)...
      )
   );

   std::future< returnType > result = task->get_future();

   {
      std::unique_lock< std::recursive_mutex > lock{ m_mutex };

      if (m_stopProcessing || m_isEmergencyStop)
         assert(false);

      if (m_pendingTask == nullptr)
         m_pendingTask = std::make_unique< Task >([task]() { (*task)(); });
   }

   m_notifier.notify_one();

   return result;
}
```

Rozdział 4 – Eksperymenty obliczeniowe

W tym rozdziale zostaną przedstawione eksperymenty obliczeniowe, które zostały przeprowadzone przy użyciu stworzonego oprogramowania. Aby dane wejściowe były dla wszystkich algorytmów takie same - z uwagi na to, że algorytm Bitonic sort, wymaga danych wejściowych o ilości równej potędze liczby 2 - eksperymenty zostaną przeprowadzone na plikach wejściowych o rozmiarach 512 MB, 1GB, 2GB oraz 4GB. Porównana zostanie również liczba wątków wykorzystywanych w procesie sortowania – 1, 4 oraz 8 wątków. W celu dokładniejszych pomiarów czasowych wydzielono pomiary dla różnych procesów w trakcie sortowania pliku:

- sortowanie,
- łączenie plików tymczasowych,
  operacje wczytywania,
- operacje zapisywania

Z uwagi na to, że celem projektu jest analiza porównawcza algorytmów sortowania, w niniejszym rozdziale zostaną przedstawione i opisane jedynie wyniki pomiarów czasowych dla procesu sortowania danych.

4.1 – Specyfikacja sprzętowa urządzenia

Poniżej znajduje się specyfikacja sprzętowa urządzenia wykorzystywanego w eksperymentach obliczeniowych. W zależności od sprzętu, na którym uruchomiony zostanie program wyniki mogą się różnić.

Procesor: Intel Core i5-7300HQ 2.50 GHz

Dysk: Samsung CM871A 128GB M2 SSD

Liczba rdzeni: 4

Pamięć RAM: 8GB

System operacyjny: Windows 10 Home

Typ systemu: x64

4.2 – Sortowanie zbiorów liczbowych

Przy użyciu autorskiego oprogramowania zostały wygenerowane pliki binarne o rozmiarach 512MB, 1GB, 2GB oraz 4GB zawierające nieujemne liczby typu *integer*. Każda liczba została wylosowana przy biblioteki *random* wprowadzonej w standardzie C++11. Funkcja generująca liczby pseudo-losowe została przedstawiona na **listingu 4.1**.

LISTING 4.1: FUNKCJA GENERUJĄCA LICZBY PSEUDO-LOSOWE

```
int getRandom()
{
   std::random_device r;
   std::default_random_engine e1(r());
   std::uniform_int_distribution<int> uniform_dist(0, INT_MAX);
   return uniform_dist(e1);
}
```

Każdy z wygenerowanych plików został posortowany wszystkimi dostępnymi aplikacji algorytmami sortującymi oraz przy użyciu 1, 4, 8 oraz 16 wątków. Wykonano minimum 20 prób na każdym z plików, a wyniki pomiarów czasów potrzebnych na ich sortowanie zostały przedstawione na **tabeli 1**. W procesie sortowania pliku wyróżniono operacje wczytywania danych z pliku oraz zapisywania posortowanych porcji do plików tymczasowych, operację łączenia wszystkich plików tymczasowych w jeden plik wynikowy oraz sam proces sortowania porcji. Tabela 1 przestawia jedynie sumę czasu sortowania wszystkich porcji. Cały proces sortowania trwa znacznie dłużej, w zależności od rozmiaru pliku wejściowego. Na podstawie przeprowadzonych pomiarów wyliczono przyspieszenie każdego z algorytmów określane wzorem:

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

gdzie:

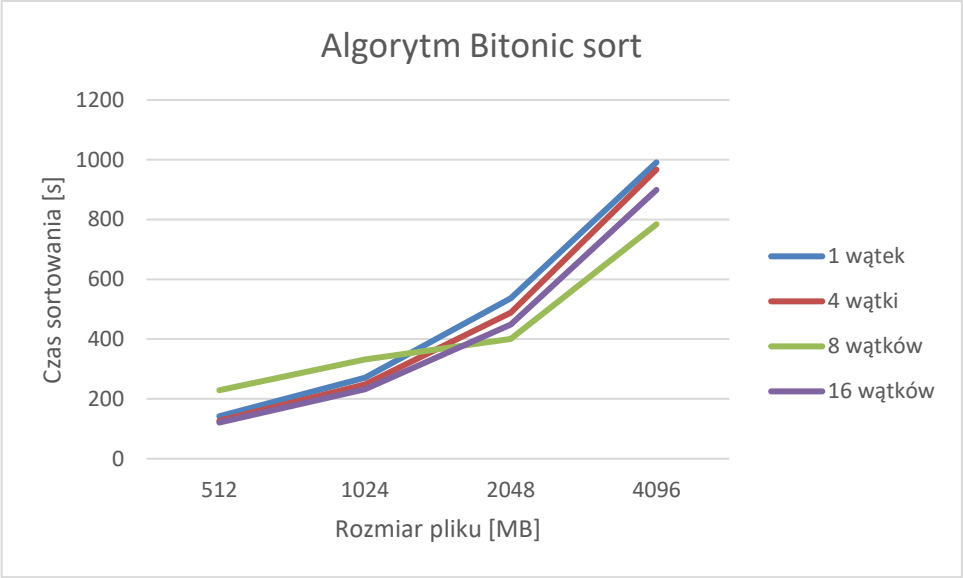
- p – liczba wątków,
- T_1 – czas wykonania algorytmu sekwencyjnego,
- T_p – czas wykonania algorytmu równoległego przy użyciu p wątków

Przyspieszenie zostało obliczone na podstawie czasów potrzebnych na sortowanie pliku o rozmiarze 512MB oraz liczby wątków równej cztery.

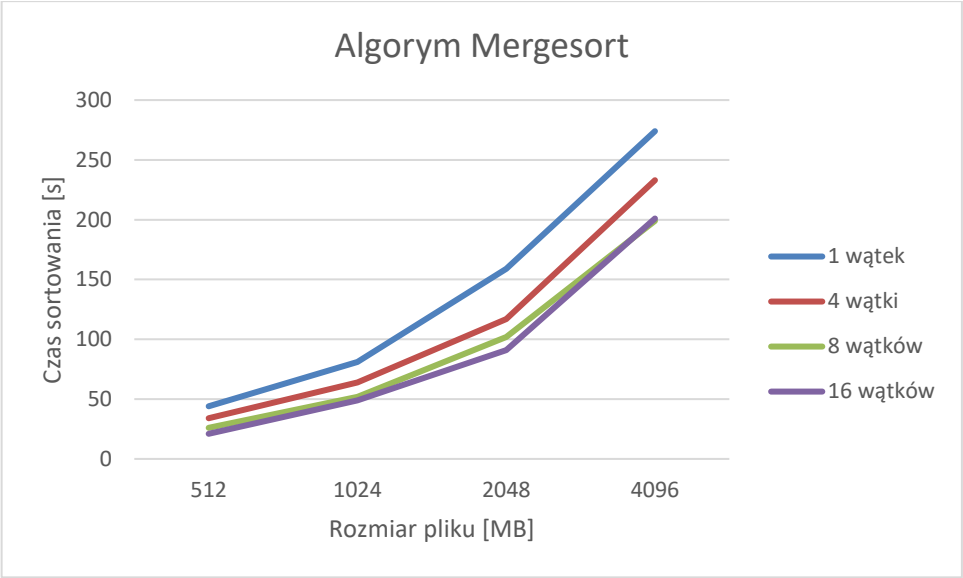
TABELA 1: WYNIKI POMIARÓW CZASU SORTOWANIA

Liczba wątków	Nazwa algorytmu Rozmiar pliku	Quick sort	Bitonic sort	Merge sort
1	512 MB	19 s	142 s	44 s
	1024 MB	42 s	271 s	81 s
	2048 MB	78 s	536 s	159 s
	4096 MB	121 s	991 s	274 s
4	512 MB	15 s	127 s	34 s
	1024 MB	27 s	249 s	64 s
	2048 MB	51 s	488 s	117 s
	4096 MB	97 s	967 s	233 s
8	512 MB	10 s	118 s	26 s
	1024 MB	19 s	221 s	52 s
	2048 MB	38 s	401 s	102 s
	4096 MB	81 s	784 s	199 s
16	512 MB	8s	121 s	21s
	1024 MB	16 s	232 s	49 s
	2048 MB	29 s	449 s	91 s
	4096 MB	63 s	899 s	201 s

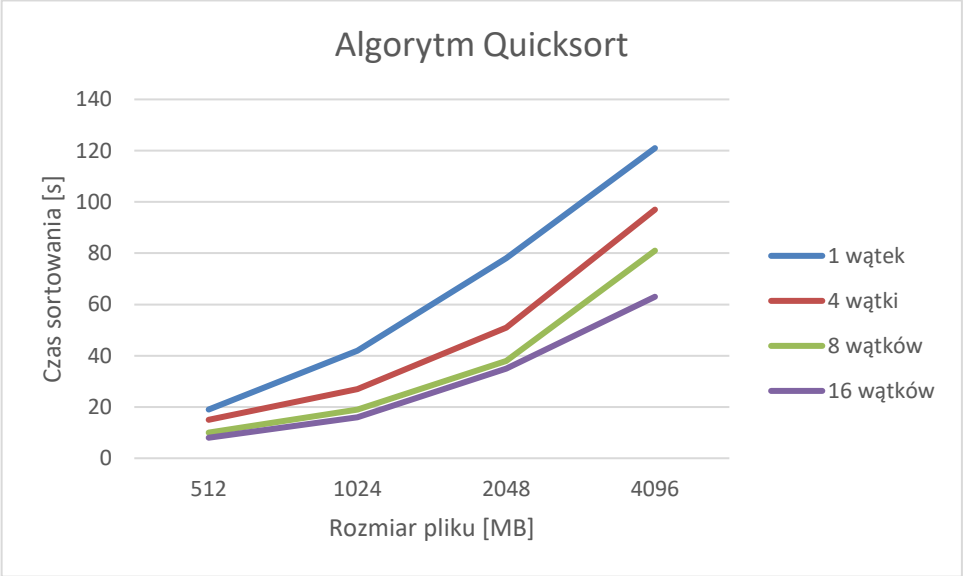
Jak można zauważyć na tabeli 1 najlepszym algorytmem okazał się Quick sort, który uzyskał najlepsze czasy sortowania wygenerowanych plików. Algorytm sortowania przez scalanie okazał się niewiele gorszy od sortowania szybkiego, jednak algorytm Bitonic sort wypadł znacznie gorzej od pozostałych algorytmów. Wynika to najprawdopodobniej z nieoptymalnej implementacji algorytmu – jak wspomniano w rozdziale 1, nie istnieje jedna, podstawowa implementacja równoległego algorytmu bitonicznego. Mimo wielu prób optymalizacji algorytmu w dalszym ciągu znacząco odstawał w eksperymentach od pozostałych algorytmów. Na **rysunkach 12-14** zobrazowano jak dla każdego z algorytmów skraca się czas sortowania wraz ze zwiększeniem liczby wątków biorących udział w procesie sortowania.



Rysunek 12. Wykres zależności czasu sortowania od wielkości sortowanego pliku dla algorytmu Bitonic sort (źródło: opracowanie własne)



Rysunek 13. Wykres zależności czasu sortowania od wielkości sortowanego pliku dla algorytmu Merge sort (źródło: opracowanie własne)



Rysunek 14. Wykres zależności czasu sortowania od wielkości sortowanego pliku dla algorytmu Quick sort (źródło: opracowanie własne)

Poprzez zastosowanie **wzoru 7.1** zostały obliczone przyspieszenia wszystkich dostępnych w aplikacji algorytmów, obliczenia znajdują się poniżej.

- Przyspieszenie algorytmu Quick sort

$$T_1 = 19\ [s]$$

$$T_4 = 15\ [s] \quad T_8 = 10\ [s] \quad T_{16} = 8\ [s]$$

$$S_4 = \frac{T_1}{T_4} = \frac{19}{15} = 1{,}27\ [s]$$

$$S_8 = \frac{T_1}{T_8} = \frac{19}{10} = 1{,}9\ [s]$$

$$S_{16} = \frac{T_1}{T_{16}} = \frac{19}{8} = 2{,}37\ [s]$$

- Przyspieszenie algorytmu Bitonic sort

$$T_1 = 142\ [s]$$

$$T_4 = 127[s] \quad T_8 = 118\ [s] \quad T_{16} = 121\ [s]$$

$$S_4 = \frac{T_1}{T_4} = \frac{142}{127} = 1{,}12\ [s]$$

$$S_8 = \frac{T_1}{T_8} = \frac{142}{118} = 1{,}2\ [s]$$

$$S_{16} = \frac{T_1}{T_{16}} = \frac{142}{121} = 1{,}17\ [s]$$

- Przyspieszenie algorytmu Merge sort

$$T_1 = 44\ [s]$$

$$T_4 = 34\ [s] \quad T_8 = 26\ [s] \quad T_{16} = 21\ [s]$$

$$S_4 = \frac{T_1}{T_4} = \frac{44}{34} = 1{,}29\ [s]$$

$$S_4 = \frac{T_1}{T_8} = \frac{44}{26} = 1{,}69[s]$$

$$S_8 = \frac{T_1}{T_4} = \frac{44}{21} = 2{,}09\ [s]$$

4.3 – Wnioski

Przy użyciu autorskiego oprogramowania zostały przeprowadzone symulacje, dzięki którym można ocenić możliwości realnego i praktycznego użycia oprogramowania oraz doboru algorytmu sortującego. Na podstawie przeprowadzonych eksperymentów symulacyjnych stwierdzono, że najlepszym i najbardziej wszechstronnym algorytmem sortującym jest **algorytm QuickSort**. Jego szybkość działania, nawet przy użyciu tylko jednego wątku, jest zadowalająca, a zwiększając liczbę wątków można przyspieszyć jego działanie. W przeprowadzonych badaniach najgorzej sprawdził się **algorytm BitonicSort** co może wynikać z jego implementacji lub założeń jakie zostały przyjęte w sortowaniu dużych plików – mowa tutaj o stałej ilości zużycia pamięci. Dla każdego z algorytmów zostały wyliczone przyspieszenia na podstawie pomiarów czasu sortowania dla pliku o rozmiarze 512MB. Wynik idealny nazywamy przyspieszeniem liniowym, które uzyskujemy gdy przyspieszenie jest równe liczbie wątków/procesów w obliczeniach równoległych. Takie zjawisko można wyrazić wzorem:

$$S_p = p \tag{7.2}$$

gdzie:

- p – liczba procesów.

Dla żadnego z przetestowanych algorytmów nie udało się osiągnąć takiego wyniku, a najlepszym przyspieszeniem jakie udało się uzyskać jest przyspieszenie dla algorytmu Quick sort.

Rozdział 5 – Zakończenie

5.1 – Podsumowanie

Realizacja projektu inżynierskiego o tematyce sortowania, porównania algorytmów oraz zrównolegleniem procesu sortowania przy użyciu wielu wątków, pozwoliła zagłębić się w temat algorytmiki, poszerzyć widzę z zakresu algorytmów sortowania oraz poznać możliwości jakie oferuje standard języka C++17. Dzięki charakterystyce porównawczej wybranych, współcześnie znanych i stosowanych algorytmów, można zaobserwować, w których przypadkach wykorzystać je w praktyce.

Realizacja oprogramowania umożliwiła zapoznanie się z nowymi, ciekawymi narzędziami oraz technologiami współcześnie używanych na rynku. Własna implementacja algorytmów, na podstawie książki *„Algorytmy bez tajemnic”* [2] w połączeniu z najnowszym standardem C++ okazała się być ciekawym wyzwaniem. Autor C++ *Bjarne Stroustrup* projektując język kierował się zasadą *Zero Cost Abstraction* i tą zasadą kieruje się również komitet standaryzacyjny *ISO C++*. Dlatego wprowadzone modyfikacje pochodzące z najnowszych standardów nie wpłynęły negatywnie na działanie samych algorytmów, a jedynie uprościły ich implementacje.

5.3 - Możliwości rozwoju projektu

Założenia postawione przed rozpoczęciem pracy nad programem zostały spełnione, jednak program można rozwinąć o nowe funkcjonalności. W przyszłości można rozszerzyć gamę algorytmów sortujących oraz ich modyfikacje.

Dodatkowo, program można rozszerzyć o kilka dodatkowych opcji podawanych jako parametry linii poleceń np. wielkość sortowanych porcji, maksymalne zużycie pamięci.

Kolejnym aspektem, o który można rozszerzyć istniejący program jest interfejs użytkownika. Program jest uruchamiany z linii poleceń wraz z argumentami, dlatego dobrym pomysłem byłoby napisanie drugiej odrębnej aplikacji, przy użyciu dowolnej technologii np. biblioteki Qt, dzięki której użytkownik wybierze odpowiednie parametry, a ta aplikacja uruchomi w tle istniejący już program.

Innym bardziej zaawansowanym pomysłem, jest benchmarking. Popularne przeglądarki z silnikiem Chrominium, takie jak Google Chrome, Brave czy nawet Microsoft Edge posiadają ciekawą funkcję, która jest dostępna pod adresem **chrome://tracing**. Umożliwia ona załadowanie pliku w formacie JSON, który jest renderowany w postaci przejrzystych wykresów z podziałem na procesy/wątki. Dzięki temu narzędziu analiza porównawcza była by prostsza, szybsza oraz dokładniejsza, a implementacja tego rozwiązania byłaby ciekawym wyzwaniem programistycznym.

Rozdział 6 – Bibliografia

6.1 – Literatura

[1] Abraham Silberschatz, Peter B. Galvin, Greg Gagne: *Podstawy systemów operacyjnych*, Wydawnictwo Naukowo-Techniczne, 2003

[2] Thomas H. Cormen: *Algorytmy bez tajemnic*, Helion, 2013 [str. 50 – 50]

[3] Darko Božidar, Tomaž Dobravec, *Comparison of paralel sorting algorithms*, Faculty of Computer and Information Science, University of Ljubljana, 2015

[4] Radosław Sokół, *Microsoft Visual Studio 2012 Programowanie w C i C++*, Helion, 2014

6.2 – Internet

[5] https://en.wikipedia.org/wiki/Bitonic_sorter#/media/File:BitonicSort1.svg

[6] https://www.alphacodingskills.com/algo/img/quick-sort.PNG

[7] https://www.techiedelight.com/merge-sort/

[8] https://www.tiobe.com/tiobe-index/

[9] https://github.com/gabime/spdlog

[10] https://github.com/jarro2783/cxxopts

[11] https://en.cppreference.com/w/

[12] https://pl.wikipedia.org/wiki/Sortowanie_szybkie