AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

PI Numerical Integration

Autor:
Trudov Mykhailo

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Program ma na celu obliczenie przybliżonej wartości liczby PI za pomocą metody całkowania numerycznego całki oznaczonej z funkcji. Do obliczenia wykorzystany zostanie algorytm numerycznego całkowania (np. metoda trapezów) w celu obliczenia przybliżonej wartości całki.

1.1. Funkcje programu

Program powinien umożliwić użytkownikowi:

- **Wybór liczby podziałów całki:** Użytkownik może określić liczbę podziałów całki, co ma wpływ na dokładność obliczeń (np. 100'000'000, 1'000'000'000, 3'000'000'000).
- **Ustawienie liczby wątków:** Użytkownik może wybrać liczbę wątków, które będą wykorzystywane do zrównoleglenia obliczeń matematycznych.
- **Wynik obliczeń:** Program powinien wyświetlić wynik obliczeń oraz czas, jaki był potrzebny do uzyskania wyniku.

1.2. Zrównoleglenie obliczeń

Do zrównoleglenia obliczeń wykorzystana zostanie biblioteka `thread`, która działa w systemach z standardem POSIX. Dzięki tej bibliotece obliczenia będą podzielone na mniejsze zadania, które będą wykonywane równocześnie, co pozwoli na szybsze uzyskanie wyniku.

1.3. Wymagania funkcjonalne

Program powinien:

- Obliczać przybliżoną wartość liczby PI za pomocą metody numerycznego całkowania.
- Umożliwiać użytkownikowi wybór liczby podziałów całki oraz liczby wątków, co pozwala na dostosowanie programu do różnych warunków obliczeniowych.
- Wyświetlać wynik obliczeń oraz czas ich trwania w terminalu, co umożliwia monitorowanie wydajności programu.

1.4. Przykład

Program ma na celu obliczenie przybliżonej wartości liczby PI za pomocą metody całkowania numerycznego całki oznaczonej z funkcji. Program pozwala użytkownikowi ustawić liczbę podziałów całki oraz liczbę wątków, które będą wykorzystywane do równoległego obliczenia całki. Program powinien wyświetlać czas obliczeń oraz wynik obliczeń.

1.5. Funkcje programu

Program powinien umożliwić użytkownikowi:

- **Wybór liczby podziałów całki:** Użytkownik może określić liczbę podziałów całki (np. 100'000'000, 1'000'000'000, 3'000'000'000), co ma wpływ na dokładność obliczeń.
- **Ustawienie liczby wątków:** Użytkownik może wybrać liczbę wątków, które będą wykorzystywane do zrównoleglenia obliczeń matematycznych.
- **Wynik obliczeń:** Program wyświetli wynik obliczeń oraz czas, który był potrzebny do uzyskania wyniku.

1.6. Zrównoleglenie obliczeń

Do zrównoleglenia obliczeń wykorzystana zostanie biblioteka `thread` w języku C++. Dzięki tej bibliotece obliczenia będą dzielone na mniejsze zadania, które będą wykonywane równocześnie przez różne wątki.

1.7. Przykład kodu

Poniżej znajduje się przykład implementacji programu w języku C++:

```
1 #include <iostream>
2 #include <thread>
3 #include <vector>
4
5 double f(double x) {
6     return 4.0 / (1.0 + x * x);  // Funkcja, kt r   ca Ćkujemy (
    funkcja arctangens)
7 }
8
```

```
void integralPart(int start, int end, double step, double& result) {
    double local_sum = 0.0;
    for (int i = start; i < end; ++i) {
        double x = (i + 0.5) * step;
        local_sum += f(x);
    }
    result = local_sum * step;
}

int main() {
    int num_intervals = 1000000000;  // Liczba podzia Ć w
    int num_threads = 4;  // Liczba w tk w
    double step = 1.0 / num_intervals;

    std::vector<std::thread> threads;
    std::vector<double> results(num_threads, 0.0);

    int intervals_per_thread = num_intervals / num_threads;

    // Rozdzielenie oblicze na w tki
    for (int i = 0; i < num_threads; ++i) {
        int start = i * intervals_per_thread;
        int end = (i + 1) * intervals_per_thread;
        threads.push_back(std::thread(integralPart, start, end, step, std::ref(results[i])));
    }

    // Czekanie na zako czenie wszystkich w tk w
    for (auto& t : threads) {
        t.join();
    }

    // Sumowanie wynik w z ka dego w tku
    double pi = 0.0;
    for (const auto& res : results) {
        pi += res;
    }

    std::cout << "Przybli ona warto Ż PI: " << pi << std::endl;
    return 0;
}
```

W powyższym przykładzie:

- Funkcja `f(x)` to funkcja, którą całkujemy. W tym przypadku jest to funkcja $\frac{4}{1+x^2}$, która jest wykorzystywana do przybliżenia liczby PI.
- Funkcja `integralPart` oblicza część całki w obrębie określonych przedziałów.
- Wątki są tworzone przy użyciu klasy `std::thread`, a każdemu wątkowi przypisywane są różne przedziały do obliczeń.
- Program sumuje wyniki z poszczególnych wątków i wyświetla końcowy wynik.

1.8. Pomiar czasu

Pomiar czasu obliczeń jest ważnym elementem, który pozwala ocenić wydajność programu przy różnych liczbach wątków. Poniżej znajduje się przykład pomiaru czasu obliczeń za pomocą biblioteki `chrono`:

```
#include <chrono>

auto start = std::chrono::high_resolution_clock::now();

// Kod oblicze

auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::
    milliseconds>(end - start);

std::cout << "Czas oblicze : " << duration.count() << " ms" << std
    ::endl;
```

Dzięki tej funkcji, program mierzy czas wykonania całego procesu obliczeniowego, co umożliwia ocenę, jak szybko program działa przy różnej liczbie wątków.

1.9. Podsumowanie

Program umożliwia obliczenie przybliżonej wartości liczby PI za pomocą metody całkowania numerycznego, przy zastosowaniu zrównoleglenia obliczeń. Dzięki tej technice obliczenia są szybsze, a użytkownik ma pełną kontrolę nad liczbą wątków i dokładnością obliczeń.

2. Analiza problemu

Celem projektu jest opracowanie programu, który przybliży wartość liczby PI za pomocą metody całkowania numerycznego. W celu obliczeń numerycznych wybieramy jedną z popularnych metod – całkowanie prostokątami, które jest szybkie, ale daje przybliżone wyniki. Rozwiązywanie tego zadania za pomocą programowania wielowątkowego ma na celu optymalizację czasową i zwiększenie efektywności obliczeń.

2.1. Wybór metody obliczeniowej

Do przybliżenia wartości liczby PI stosujemy metodę całkowania numerycznego, w której przyjmujemy prostokątne przybliżenia pod funkcją:

$$\pi \approx \int_0^1 \frac{4}{1+x^2} dx$$

Zadanie polega na obliczeniu całki oznaczonej z funkcji $\frac{4}{1+x^2}$ w przedziale $[0, 1]$. Metoda, którą wybrano do obliczeń, polega na podzieleniu przedziału całkowania na małe części (prostokąty), a następnie na obliczeniu sumy pól tych prostokątów, co stanowi przybliżenie wartości całki.

2.2. Problemy związane z wydajnością

Z uwagi na dużą liczbę obliczeń wymaganych do uzyskania dokładnych wyników, konieczne stało się zrównoleglenie obliczeń, aby przyspieszyć proces. Zrównoleglenie pozwala na podział pracy na mniejsze fragmenty, które mogą być przetwarzane równocześnie przez różne rdzenie procesora. Ostateczny wynik uzyskiwany jest przez zsumowanie wyników obliczeń przeprowadzonych w poszczególnych wątkach.

2.3. Metody zrównoleglenia

W programie zastosowano wielowątkowość z użyciem standardowej biblioteki C++ `thread`, która pozwala na równoległe obliczanie poszczególnych części całki. Wybór tej metody pozwala na szybsze wykonanie obliczeń, ponieważ zadania są rozdzielane między różne rdzenie procesora, co redukuje czas oczekiwania na uzyskanie końcowego wyniku.

2.4. Podział całki na mniejsze części

Aby poprawnie podzielić zadanie na mniejsze części, całkowity przedział $[0, 1]$ dzielimy na n podprzedziałów. Każdy wątek oblicza część całki dla swojego fragmentu przedziału. Liczba podziałów oraz liczba wątków mają istotny wpływ na dokładność oraz czas obliczeń:

- **Liczba podziałów:** Im większa liczba podziałów, tym większa dokładność obliczeń. Jednak wzrasta także czas potrzebny do wykonania obliczeń.

- **Liczba wątków:** Wzrost liczby wątków nie zawsze prowadzi do skrócenia czasu obliczeń. Zbyt duża liczba wątków może powodować przeciążenie procesora, co skutkuje spadkiem wydajności.

2.5. Wyzwania w implementacji

Podczas implementacji programu napotkaliśmy na kilka wyzwań, które zostały rozwiązane poprzez:

- **Zarządzanie pamięcią:** Użycie odpowiednich mechanizmów synchronizacji, aby uniknąć błędów związanych z dostępem do współdzielonych zasobów przez różne wątki.

- **Zarządzanie wątkami:** Odpowiednie przydzielanie zadań poszczególnym wątkom oraz synchronizacja wyników w celu obliczenia finalnej wartości całki.

- **Optymalizacja:** Próba znalezienia optymalnej liczby wątków, która zapewniła minimalny czas obliczeń. Zbyt wiele wątków może prowadzić do przeciążenia procesora, a zbyt mała liczba wątków – do niskiej efektywności obliczeń.

2.6. Testowanie i optymalizacja

Po napisaniu programu przeprowadzono szereg testów, których celem było sprawdzenie wpływu liczby wątków na czas obliczeń. Zmierzono czas wykonania programu przy różnych liczbach wątków, a wyniki zapisano w formie tabeli oraz wykresu. Na podstawie wyników testów sprawdzono, jaka liczba wątków daje najlepszą wydajność i minimalizuje czas obliczeń.

2.7. Podsumowanie analizy

Wybór metody całkowania numerycznego oraz zrównoleglenie obliczeń z wykorzystaniem biblioteki `thread` pozwoliły na skuteczne przybliżenie wartości liczby PI w krótkim czasie. Przeprowadzone analizy i testy wykazały, że optymalna liczba wątków, przy której czas obliczeń jest najmniejszy, zależy od liczby podziałów oraz specyfikacji sprzętowej komputera. Zoptymalizowany program działa szybko i dokładnie, a dzięki zastosowaniu zrównoleglenia możemy uzyskać lepszą wydajność przy dużych liczbach podziałów całki.

3. Projektowanie

Projektowanie programu składa się z kilku kluczowych etapów, które pozwalają na skuteczne zaplanowanie i realizację zadania. Poniżej przedstawiono etapy projektowania, począwszy od analizy wymagań, przez wybór algorytmu, po strukturę kodu.

3.1. Wybór algorytmu

W pierwszym etapie projektowania wybrano algorytm do obliczania przybliżonej wartości liczby PI. Algorytm bazuje na metodzie całkowania numerycznego z funkcją $\frac{4}{1+x^2}$, gdzie przedział całkowania wynosi $[0, 1]$. Aby uzyskać możliwie jak najdokładniejsze wyniki, przedział ten jest dzielony na małe fragmenty, a każdy fragment jest obliczany w osobnym wątku.

3.2. Struktura programu

Program został zaprojektowany w taki sposób, aby umożliwić łatwe dostosowanie liczby podziałów całkowania oraz liczby wątków, które będą wykonywały obliczenia. Struktura programu jest następująca:

- **main.cpp** – główny plik programu, który zarządza wejściem użytkownika, tworzeniem wątków oraz obliczaniem wartości całki.
- **function.h** – plik nagłówkowy, w którym znajdują się definicje funkcji matematycznych, takich jak funkcja do obliczania wartości całki.
- **integral**$_c alculator.cpp – plik zawierajcy implementacj logikiobliczaniacakinumerycznej.Zawiera_$

3.3. Zarządzanie wątkami

Kluczową częścią projektu jest wykorzystanie wielowątkowości do równoległego obliczania poszczególnych części całki. Zastosowano bibliotekę `<thread>` w języku C++ do tworzenia i zarządzania wątkami. Każdy wątek wykonuje obliczenia dla swojego fragmentu przedziału, a po zakończeniu pracy wątki są synchronizowane, aby uzyskać ostateczny wynik.

- **Podział obliczeń** – przedział $[0, 1]$ dzielony jest na n małych przedziałów. Każdy wątek oblicza wartość funkcji w swoim przedziale.
- **Synchronizacja** – po zakończeniu pracy wątków, ich wyniki są sumowane, a wynik końcowy jest wyświetlany na ekranie użytkownika.

3.4. Interfejs użytkownika

Interfejs użytkownika jest prosty, umożliwia wprowadzenie liczby podziałów oraz liczby wątków. Program akceptuje dwa parametry wejściowe:

- Liczba podziałów n, która decyduje o dokładności obliczeń (im większa liczba podziałów, tym dokładniejsze przybliżenie).
- Liczba wątków w, która określa, na ile części zostaną podzielone obliczenia.

Program powinien umożliwiać użytkownikowi elastyczną konfigurację parametrów, co pozwala na testowanie programu z różnymi ustawieniami.

3.5. Bezpieczeństwo i synchronizacja

Z racji tego, że wątki mają dostęp do wspólnych zasobów, konieczne było zastosowanie mechanizmów synchronizacji, aby uniknąć problemów z dostępem do zmiennych współdzielonych. Do synchronizacji użyto standardowych mechanizmów C++:

- **Mutex** – do zapewnienia, że tylko jeden wątek będzie mógł w danym momencie modyfikować zmienną przechowującą wynik całkowania.
- **Condition Variables** – do zapewnienia odpowiedniej synchronizacji między wątkami, aby nie były wykonywane jednocześnie operacje na tych samych zasobach.

3.6. Testowanie i optymalizacja

Po zaprojektowaniu programu, przeprowadzono szereg testów mających na celu optymalizację liczby wątków oraz liczby podziałów. Testy wykazały, że wzrost liczby wątków nie zawsze prowadzi do skrócenia czasu obliczeń. Zbyt duża liczba wątków powoduje przeciążenie procesora, dlatego przeprowadzono optymalizację polegającą na znalezieniu najbardziej efektywnej liczby wątków.

4. Implementacja

W tej sekcji opisano implementację programu, w tym kluczowe fragmenty kodu oraz sposób realizacji wielowątkowości za pomocą biblioteki `<thread>`.

4.1. Struktura kodu

Program został podzielony na kilka modułów, co ułatwia zarządzanie kodem i jego czytelność:

- **main.cpp** – główny plik programu odpowiedzialny za zarządzanie przepływem programu i interakcję z użytkownikiem.
- **function.h** – plik nagłówkowy definiujący funkcję matematyczną $\frac{4}{1+x^2}$.
- **integral_calculator.cpp** – plik implementujący logikę obliczania wartości całki numerycznej.
- **thread_manager.cpp** – plik zarządzający wątkami i synchronizacją wyników.

4.2. Główne fragmenty kodu

4.2.1. Główna funkcja programu (`main.cpp`)

Na listingu 1 znajduje się kod funkcji głównej, która inicjalizuje program, wczytuje dane wejściowe od użytkownika oraz uruchamia obliczenia:

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
#include "function.h"
#include "integral_calculator.h"

std::mutex result_mutex; // Mutex do synchronizacji

int main() {
    int n; // Liczba podzia Ć w
    int threads_count; // Liczba w tk w
    double result = 0.0;

    // Wczytanie danych wej Żciowych
    std::cout << "Podaj liczbe podzialow: ";
    std::cin >> n;
```

```
18      std::cout << "Podaj liczbe watkow: ";
19      std::cin >> threads_count;
20
21      // Wywo Ćanie funkcji obliczaj cej ca Ćk Ź
22      auto start_time = std::chrono::high_resolution_clock::now();
23      result = calculate_integral(n, threads_count);
24      auto end_time = std::chrono::high_resolution_clock::now();
25
26      // Obliczenie czasu wykonania
27      std::chrono::duration<double> elapsed_time = end_time -
        start_time;
28
29      // Wy Żwietlenie wynik w
30      std::cout << "Wynik: " << result << std::endl;
31      std::cout << "Czas wykonania: " << elapsed_time.count() << "
        sekund" << std::endl;
32
33      return 0;
34  }
```

Listing 1. Główna funkcja programu

4.2.2. Funkcja obliczająca całkę (`integral_calculator.cpp`)

Funkcja `calculate_integral` dzieli przedział $[0, 1]$ na części i przypisuje każdą część do wątku, na listingu 2 jest pokazany przykład.

```
1   #include "integral_calculator.h"
2   #include <functional>
3   #include <thread>
4   #include <mutex>
5   #include <vector>
6
7   extern std::mutex result_mutex;
8
9   double calculate_integral(int n, int threads_count) {
10      double result = 0.0;
11      double dx = 1.0 / n; // Szeroko Ż  przedzia Ću
12      std::vector<std::thread> threads;
13
14      // Funkcja do obliczania cz Ź Żci ca Ćki
15      auto partial_integral = [&](int start, int end, double&
        partial_result) {
16          for (int i = start; i < end; i++) {
17              double x = (i + 0.5) * dx;
```

```
18              partial_result += (4.0 / (1.0 + x * x)) * dx;
19          }
20          // Dodanie wyniku cz Ź Żciowego do globalnego wyniku
21          std::lock_guard<std::mutex> lock(result_mutex);
22          result += partial_result;
23      };
24
25      // Podzia Ć pracy mi Ździy w tki
26      int range = n / threads_count;
27      for (int i = 0; i < threads_count; i++) {
28          int start = i * range;
29          int end = (i == threads_count - 1) ? n : start + range;
30          double partial_result = 0.0;
31
32          threads.emplace_back(partial_integral, start, end, std::ref
    (partial_result));
33      }
34
35      // Oczekiwanie na zako czenie pracy w tk w
36      for (auto& thread : threads) {
37          thread.join();
38      }
39
40      return result;
41  }
```

Listing 2. Funkcja calculate_integral

4.2.3. Definicja funkcji matematycznej (`function.h`)

Funkcja $\frac{4}{1+x^2}$ została zdefiniowana w pliku nagłówkowym, na listingu 3 znajduje się przykład.

```
1  #ifndef FUNCTION_H
2  #define FUNCTION_H
3
4  inline double function(double x) {
5      return 4.0 / (1.0 + x * x);
6  }
7
8  #endif
```

Listing 3. Funkcja matematyczna

4.3. Kompilacja i uruchomienie programu

Aby skompilować program, użyto kompilatora `g++`:

```
g++ -std=c++11 main.cpp integral_calculator.cpp -o pi_calculator -pthread
```

Po kompilacji program można uruchomić komendą:

```
./pi_calculator
```

4.4. Testowanie programu

Testowanie programu obejmowało sprawdzenie poprawności wyników oraz pomiar czasu działania dla różnych wartości n i liczby wątków. Wyniki zostały zapisane w tabelach i wykorzystane do analizy wydajności programu.

4.5. Problemy napotkane podczas implementacji

Podczas implementacji pojawiły się następujące problemy:

- Synchronizacja wyników z różnych wątków – rozwiązano za pomocą mutexów.
- Skalowanie programu – zwiększanie liczby wątków nie zawsze skraca czas wykonania, co wymagało optymalizacji algorytmu.

5. Wnioski

5.1. Analiza wyników

Podczas realizacji projektu przeprowadzono szczegółowe testy programu w celu analizy jego wydajności. Wyniki testów można podsumować następująco:

- Czas wykonywania programu ulegał skróceniu wraz ze wzrostem liczby wątków, co potwierdza poprawność zaimplementowanej wielowątkowości.
- Dla dużej liczby podziałów ($n > 10^8$) i liczby wątków przekraczającej liczbę fizycznych rdzeni procesora, zauważono spadek efektywności. Wynika to z narzutu związanego z przełączaniem kontekstu wątków.
- Program działał poprawnie na różnych konfiguracjach sprzętowych, jednak czas wykonania zależał od liczby dostępnych rdzeni procesora oraz ich częstotliwości taktowania.
- Optymalna liczba wątków zależy od liczby rdzeni fizycznych – zazwyczaj najlepsze wyniki osiągnięto przy liczbie wątków równej liczbie rdzeni.

5.2. Zalety i wady rozwiązania

5.2.0.1. Zalety:

- Wielowątkowość znacząco zwiększyła wydajność programu przy dużych obciążeniach obliczeniowych.
- Użycie mutexów zapewniło poprawną synchronizację wątków i uniknięcie błędów związanych z dostępem współbieżnym.
- Program jest elastyczny i pozwala użytkownikowi na dostosowanie liczby podziałów oraz wątków, co umożliwia testowanie wydajności na różnych konfiguracjach.

5.2.0.2. Wady:

- Skalowalność programu jest ograniczona przez sprzętową liczbę rdzeni procesora.
- Narzut związany z zarządzaniem wątkami powoduje, że przy małej liczbie podziałów ($n < 10^6$) wielowątkowość nie przynosi korzyści.

5.3. Refleksja nad implementacją

Proces implementacji tego projektu dostarczył cennych doświadczeń:

- Wielowątkowość, choć potężna, wymaga starannego zarządzania synchronizacją oraz zasobami.
- Biblioteka `<thread>` w standardzie POSIX jest wygodnym i wydajnym narzędziem do implementacji wielowątkowości w języku C++.
- Analiza wydajności programu pozwoliła lepiej zrozumieć wpływ architektury sprzętowej na czas wykonania i efektywność algorytmów.

5.4. Przyszłe usprawnienia

Na podstawie uzyskanych wyników zaproponowano następujące możliwe usprawnienia:

- Implementacja bardziej zaawansowanych metod synchronizacji, takich jak użycie atomów zamiast mutexów, co mogłoby zmniejszyć narzut.
- Wprowadzenie dynamicznego podziału pracy między wątkami w celu bardziej równomiernego obciążenia procesorów.
- Optymalizacja algorytmu poprzez zastosowanie bardziej wydajnych metod całkowania numerycznego, takich jak metoda Simpsona lub kwadratura Gaussa.

5.5. Znaczenie projektu

Realizacja tego projektu pozwoliła lepiej zrozumieć mechanizmy wielowątkowości oraz znaczenie optymalizacji algorytmów w kontekście współczesnych procesorów wielordzeniowych. Wyniki testów pokazują, że odpowiednio zaprojektowane i zaimplementowane rozwiązania wielowątkowe mogą znacząco zwiększyć wydajność aplikacji.

Spis rysunków

Spis tabel

Spis listingów