|  |  |  |
| --- | --- | --- |
| **Imię i Nazwisko**  Patryk Twardosz | **Kierunek**  Informatyka Techniczna | **Rok studiów i grupa**  I rok, Gr. 9 |
| **Data zajęć**  09.12.2024r. | **Numer i temat sprawozdania**  Lab 8, 9 i 10 – Przetwarzanie współbieżne i OpenMP | |

**Lab8:**

**Cel:**

* Doskonalenie umiejętności realizacji synchronizacji w języku C za pomocą zmiennych warunku oraz w programach obiektowych w Javie za pomocą narzędzi pakietu java.util.concurrency

**Zadanie:**

1. Przygotowanie projektu
2. Implementacja brakujących funkcji bariery

#include <pthread.h>

#include <stdio.h>

static int threads\_to\_wait;   // Liczba wątków, które mają osiągnąć barierę

static int threads\_waiting;   // Liczba wątków, które już dotarły do bariery

static pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; // Muteks do synchronizacji

static pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;   // Zmienna warunkowa

void bariera\_init(int n) {

  threads\_to\_wait = n;  // Ustaw liczbę wątków do oczekiwania

  threads\_waiting = 0;  // Na początku żaden wątek nie czeka

}

void bariera(void) {

  pthread\_mutex\_lock(&mutex);

  threads\_waiting++; // Zwiększ licznik wątków oczekujących

  if (threads\_waiting == threads\_to\_wait) {

    threads\_waiting = 0;

    pthread\_cond\_broadcast(&cond); // Obudź wszystkie oczekujące wątki

  } else {

    // Wątek czeka, aż wszystkie inne dotrą do bariery

    pthread\_cond\_wait(&cond, &mutex);

  }

  pthread\_mutex\_unlock(&mutex);

}

- Wynik działania zaimplementowanej bariery, wszystkie wątki czekają do momentu gdy czekają wszystkie

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

1. Implemntacja Czytelni

typedef struct {

  int l\_p; // liczba piszacych

  int l\_c; // liczba czytajacych

  // <- zasoby czytelni

  pthread\_mutex\_t mutex;

  pthread\_cond\_t czytelnicy\_cond;

  pthread\_cond\_t pisarze\_cond;

} cz\_t;

int my\_read\_lock\_lock(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->mutex);

  while (cz\_p->l\_p > 0)

    pthread\_cond\_wait(&cz\_p->czytelnicy\_cond, &cz\_p->mutex);

  cz\_p->l\_c++;

  pthread\_mutex\_unlock(&cz\_p->mutex);

}

int my\_read\_lock\_unlock(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->mutex);

  cz\_p->l\_c--;

  if (cz\_p->l\_c == 0)

    pthread\_cond\_signal(&cz\_p->pisarze\_cond);

  pthread\_mutex\_unlock(&cz\_p->mutex);

}

int my\_write\_lock\_lock(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->mutex);

  while (cz\_p->l\_c > 0 || cz\_p->l\_p > 0)

    pthread\_cond\_wait(&cz\_p->pisarze\_cond, &cz\_p->mutex);

  cz\_p->l\_p++;

  pthread\_mutex\_unlock(&cz\_p->mutex);

}

int my\_write\_lock\_unlock(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->mutex);

  cz\_p->l\_p--;

  pthread\_cond\_broadcast(&cz\_p->czytelnicy\_cond);

  pthread\_cond\_signal(&cz\_p->pisarze\_cond);

  pthread\_mutex\_unlock(&cz\_p->mutex);

}

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznieFragment output’u:

void inicjuj(cz\_t\* cz\_p){

  cz\_p->l\_p = 0;

  cz\_p->l\_c = 0;

  pthread\_mutex\_init(&cz\_p->mutex, NULL);

  pthread\_cond\_init(&cz\_p->czytelnicy\_cond, NULL);

  pthread\_cond\_init(&cz\_p->pisarze\_cond, NULL);

}

void czytam(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->mutex);

  printf("\t\t\t\t\tczytam:  l\_c %d, l\_p %d\n", cz\_p->l\_c, cz\_p->l\_p);

#if MY\_DEBUG

  if (cz\_p->l\_p > 1 || (cz\_p->l\_p == 1 && cz\_p->l\_c > 0) || cz\_p->l\_p < 0 || cz\_p->l\_c < 0) {

    printf("Błąd: nieprawidłowy stan liczników!\n");

    pthread\_mutex\_unlock(&cz\_p->mutex);

    exit(0);

  }

#endif

  pthread\_mutex\_unlock(&cz\_p->mutex);

  usleep(rand() % 3000000);

}

void pisze(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->mutex);

  printf("\t\t\t\t\tpisze:   l\_c %d, l\_p %d\n", cz\_p->l\_c, cz\_p->l\_p);

#if MY\_DEBUG

  if (cz\_p->l\_p > 1 || (cz\_p->l\_p == 1 && cz\_p->l\_c > 0) || cz\_p->l\_p < 0 || cz\_p->l\_c < 0) {

    printf("Błąd: nieprawidłowy stan liczników!\n");

    pthread\_mutex\_unlock(&cz\_p->mutex);

    exit(0);

  }

#endif

  pthread\_mutex\_unlock(&cz\_p->mutex);

  usleep(rand() % 3000000);

}

1. Użycie *read\_write\_locks*

typedef struct {

  int l\_p; // liczba piszacych

  int l\_c; // liczba czytajacych

  // <- zasoby czytelni

  pthread\_rwlock\_t rwlock;  // zamek do odczytu i zapisu

  pthread\_mutex\_t licznik\_mutex; // mutex do ochrony liczników

} cz\_t;

int my\_read\_lock\_lock(cz\_t\* cz\_p){

  pthread\_rwlock\_rdlock(&cz\_p->rwlock); // Zamek do odczytu

  pthread\_mutex\_lock(&cz\_p->licznik\_mutex); // Ochrona liczników

  cz\_p->l\_c++;

  pthread\_mutex\_unlock(&cz\_p->licznik\_mutex);

  return 0;

}

int my\_read\_lock\_unlock(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->licznik\_mutex);

  cz\_p->l\_c--;

  pthread\_mutex\_unlock(&cz\_p->licznik\_mutex);

  pthread\_rwlock\_unlock(&cz\_p->rwlock); // Zwolnienie zamka

  return 0;

}

int my\_write\_lock\_lock(cz\_t\* cz\_p){

  pthread\_rwlock\_wrlock(&cz\_p->rwlock); // Zamek do zapisu

  pthread\_mutex\_lock(&cz\_p->licznik\_mutex); // Ochrona liczników

  cz\_p->l\_p++;

  pthread\_mutex\_unlock(&cz\_p->licznik\_mutex);

  return 0;

}

int my\_write\_lock\_unlock(cz\_t\* cz\_p){

  pthread\_mutex\_lock(&cz\_p->licznik\_mutex);

  cz\_p->l\_p--;

  pthread\_mutex\_unlock(&cz\_p->licznik\_mutex);

  pthread\_rwlock\_unlock(&cz\_p->rwlock); // Zwolnienie zamka

  return 0;

}

void inicjuj(cz\_t\* cz\_p){

  cz\_p->l\_p = 0;

  cz\_p->l\_c = 0;

  pthread\_rwlock\_init(&cz\_p->rwlock, NULL);

  pthread\_mutex\_init(&cz\_p->licznik\_mutex, NULL);

}

void czytam(cz\_t\* cz\_p){

  printf("\t\t\t\t\tczytam:  l\_c %d, l\_p %d\n", cz\_p->l\_c, cz\_p->l\_p);

  if(cz\_p->l\_p > 1 ||

    (cz\_p->l\_p == 1 && cz\_p->l\_c > 0) ||

    cz\_p->l\_p < 0 || cz\_p->l\_c < 0)

    printf("Blad: ....\n");

  usleep(rand()%3000000);

}

void pisze(cz\_t\* cz\_p){

  printf("\t\t\t\t\tpisze:   l\_c %d, l\_p %d\n", cz\_p->l\_c, cz\_p->l\_p);

  if(cz\_p->l\_p > 1 ||

    (cz\_p->l\_p == 1 && cz\_p->l\_c > 0) ||

    cz\_p->l\_p < 0 || cz\_p->l\_c < 0)

    printf("Blad: ....\n");

  usleep(rand()%3000000);

}

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

1. Implementacja Bariery w Java

class Bariera {

    private final int liczbaWymagana;

    private int liczbaOczekujacych = 0;

    public Bariera(int liczbaWymagana) {

        this.liczbaWymagana = liczbaWymagana;

    }

    public synchronized void czekaj() throws InterruptedException {

        liczbaOczekujacych++;

        if (liczbaOczekujacych < liczbaWymagana) {

            wait(); // Czekanie na spełnienie warunku

        } else {

            liczbaOczekujacych = 0; // Reset bariery

            notifyAll(); // Obudzenie wszystkich

        }

    }

}

1. Implementacja Czytelni w Java

public class Czytelnia {

    private int liczbaCzytelnikow = 0; // Liczba aktywnych czytelników

    private int liczbaPisarzy = 0;    // Liczba aktywnych pisarzy

    private final ReentrantLock lock = new ReentrantLock();

    private final Condition czytelnicyCondition = lock.newCondition();

    private final Condition pisarzeCondition = lock.newCondition();

    public void chceCzytac() throws InterruptedException {

        lock.lock();

        try {

            while (liczbaPisarzy > 0) // Czeka na brak pisarzy

                czytelnicyCondition.await();

            liczbaCzytelnikow++;

        } finally { lock.unlock(); }

    }

    public void koniecCzytania() {

        lock.lock();

        try {

            liczbaCzytelnikow--;

            if (liczbaCzytelnikow == 0) // Budzi pisarzy, jeśli nie ma aktywnych czytelników

                pisarzeCondition.signal();

        } finally { lock.unlock(); }

    }

    public void chcePisac() throws InterruptedException {

        lock.lock();

        try {

            while (liczbaCzytelnikow > 0 || liczbaPisarzy > 0) // Czeka na brak czytelników i pisarzy

                pisarzeCondition.await();

            liczbaPisarzy++;

        } finally { lock.unlock(); }

    }

    public void koniecPisania() {

        lock.lock();

        try {

            liczbaPisarzy--;

            if (lock.hasWaiters(czytelnicyCondition)) // Jeśli są czekający czytelnicy, budzi ich

                czytelnicyCondition.signalAll();

            else pisarzeCondition.signal();

        } finally { lock.unlock(); }

    }

}

**Wnioski:**

* Programowanie współbieżne wymaga skutecznego zarządzania dostępem do zasobów współdzielonych, aby uniknąć konfliktów między wątkami czy procesami.
* Mechanizm synchronizacji, taki jak semafory czy muteksy, umożliwia ochronę danych przed jednoczesnym dostępem wielu wątków, co jest kluczowe dla poprawności programu.
* Problem czytelników i pisarzy ilustruje konieczność równoważenia wydajności i bezpieczeństwa dostępu, gdzie ważne jest ograniczenie opóźnień dla czytelników bez narażania danych na niespójność.
* Implementacja algorytmu rozwiązującego ten problem wymaga precyzyjnego określenia priorytetów, aby zapobiec głodzeniu wątków (np. pisarzy).
* Testy pokazały, że nawet prosta zmiana w kolejności operacji synchronizacyjnych może znacząco wpłynąć na działanie aplikacji i wydajność.

**Lab9:**

**Cel:**

* Nabycie umiejętności tworzenia i implementacji programów równoległych z wykorzystaniem OpenMP

**Zadanie:**

1. Przygotowanie projektu
2. Obraz zawierający zrzut ekranu, typografia

   Opis wygenerowany automatycznieUruchomienie dla czterech wątków

{ // 4. bazujące na zmiennej środowiskoewej

    double suma\_parallel=0.0;

#pragma omp parallel for default(none) shared(a) reduction(+:suma\_parallel) ordered

    for(int i=0;i<WYMIAR;i++) {

      int id\_w = omp\_get\_thread\_num();

        suma\_parallel += a[i];

  #pragma omp ordered

        printf("a[%2d]->W\_%1d  \n",i,id\_w);

    }

    printf("\nSuma wyrazów tablicy równolegle (OMP\_NUM\_THREADS): %lf\n\n\n", suma\_parallel);

  }

1. Obraz zawierający zrzut ekranu

   Opis wygenerowany automatyczniePrzetestowanie 4 wersji klauzuli *schedule*:

{ // 5.1. static, porcja 3

    double suma\_parallel=0.0;

#pragma omp parallel for num\_threads(4) schedule(static, 3) default(none) shared(a) reduction(+:suma\_parallel) ordered

    for(int i=0;i<WYMIAR;i++) {

      int id\_w = omp\_get\_thread\_num();

        suma\_parallel += a[i];

  #pragma omp ordered

        printf("a[%2d]->W\_%1d  \n",i,id\_w);

    }

    printf("\nSuma wyrazów tablicy równolegle (static, porcja 3): %lf\n\n\n", suma\_parallel);

  }

Obraz zawierający zrzut ekranu

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu, tekst, typografia

Opis wygenerowany automatycznie**Obraz zawierający zrzut ekranu, sztuka

Opis wygenerowany automatycznie**

{ // 5.2. static, porcja default

    double suma\_parallel=0.0;

#pragma omp parallel for num\_threads(4) schedule(static) default(none) shared(a) reduction(+:suma\_parallel) ordered

    for(int i=0;i<WYMIAR;i++) {

      int id\_w = omp\_get\_thread\_num();

        suma\_parallel += a[i];

  #pragma omp ordered

        printf("a[%2d]->W\_%1d  \n",i,id\_w);

    }

    printf("\nSuma wyrazów tablicy równolegle (static, porcja default): %lf\n\n\n", suma\_parallel);

  }

  { // 5.3. dynamic, porcja 2

    double suma\_parallel=0.0;

#pragma omp parallel for num\_threads(4) schedule(dynamic, 2) default(none) shared(a) reduction(+:suma\_parallel) ordered

    for(int i=0;i<WYMIAR;i++) {

      int id\_w = omp\_get\_thread\_num();

        suma\_parallel += a[i];

  #pragma omp ordered

        printf("a[%2d]->W\_%1d  \n",i,id\_w);

    }

    printf("\nSuma wyrazów tablicy równolegle (dynamic, porcja 2): %lf\n\n\n", suma\_parallel);

  }

  { // 5.4. dynamic, porcja default

    double suma\_parallel=0.0;

#pragma omp parallel for num\_threads(4) schedule(dynamic) default(none) shared(a) reduction(+:suma\_parallel) ordered

    for(int i=0;i<WYMIAR;i++) {

      int id\_w = omp\_get\_thread\_num();

        suma\_parallel += a[i];

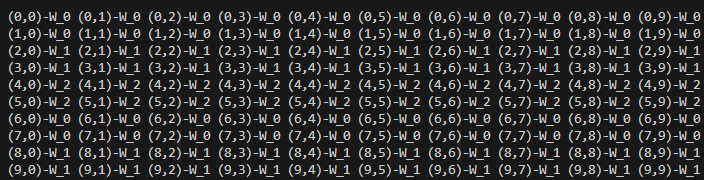
  #pragma omp ordered

        printf("a[%2d]->W\_%1d  \n",i,id\_w);

    }

    printf("\nSuma wyrazów tablicy równolegle (dynamic, porcja default): %lf\n\n\n", suma\_parallel);

  }

1. Zrównoleglenie sumowania tablicy 2D
   1. dekompozycja wierszowa – zrównoleglenie pętli zewnętrznej

{ // 7.1. dekompozycja wierszowa

    double sum\_parallel=0.0;

    #pragma omp parallel for schedule(static, 2) reduction(+:sum\_parallel) default(none) shared(a) ordered

    for(int i = 0; i < WYMIAR; i++) {

      #pragma omp ordered

      {

        for(int j = 0; j < WYMIAR; j++) {

          sum\_parallel += a[i][j];

          printf("(%1d,%1d)-W\_%1d ", i, j, omp\_get\_thread\_num());

        }

        printf("\n");

      }

    }

    printf("Suma wyrazów tablicy równolegle: %lf\n\n", sum\_parallel);

  }

* 1. dekompozycja kolumnowa - zrównoleglenie pętli wewnętrznej

{ // 7.2. dekompozycja kolumnowa wew (3.0)

    double sum\_parallel=0.0;

    for(int i = 0; i < WYMIAR; i++) {

      #pragma omp parallel for schedule(dynamic) reduction(+:sum\_parallel) default(none) shared(a, i) ordered

      for(int j = 0; j < WYMIAR; j++) {

        sum\_parallel += a[i][j];

        #pragma omp ordered

        printf("(%1d,%1d)-W\_%1d ", i, j, omp\_get\_thread\_num());

      }

      printf("\n");

    }

    printf("Suma wyrazów tablicy równolegle: %lf\n\n", sum\_parallel);

  }

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

* 1. dekompozycja kolumnowa - zrównoleglenie pętli zewnętrznej

Obraz zawierający tekst, zrzut ekranu, design, sztuka

Opis wygenerowany automatycznie

{ // 7.3. dekompozycja kolumnowa zew (3.5)

    double sum\_parallel=0.0;

    #pragma omp parallel default(none) shared(a, sum\_parallel)

    {

      double local\_sum = 0;

      #pragma omp for schedule(static) ordered

      for(int j = 0; j < WYMIAR; j++) {

        for(int i = 0; i < WYMIAR; i++) {

          local\_sum += a[i][j];

          #pragma omp ordered

          printf("(%1d,%1d)-W\_%1d ", i, j, omp\_get\_thread\_num());

        }

        #pragma omp ordered

        printf("\n");

      }

      #pragma omp critical

      {

        sum\_parallel += local\_sum;

      }

    }

    printf("Suma wyrazów tablicy równolegle: %lf\n\n", sum\_parallel);

  }

* 1. dekompozycja 2D

{ // 7.4. dekompozycja 2D

    double sum\_parallel=0.0;

#pragma omp parallel for schedule(static, 2) num\_threads(3) default(none) shared(a) reduction(+:sum\_parallel) ordered

    for(int i = 0; i < WYMIAR; i++) {

      int id\_w = omp\_get\_thread\_num();

  #pragma omp parallel for schedule(static, 2) num\_threads(2) firstprivate(id\_w) reduction(+:sum\_parallel) ordered

      for(int j = 0; j < WYMIAR; j++) {

        sum\_parallel += a[i][j];

        #pragma omp ordered

        printf("(%1d,%1d)-W\_%1d,%1d ",i,j,id\_w,omp\_get\_thread\_num());

        if (j == WYMIAR - 1)

          printf("\n");

      }

    }

    printf("Suma wyrazów tablicy równolegle: %lf\n\n", sum\_parallel);

  }

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

* 1. Obraz zawierający tekst, zrzut ekranu, Czcionka, typografia

     Opis wygenerowany automatycznierozważenie wariantu dekompozycji 2D bez ustalania rozmiaru porcji

{ // 7.5. dekompozycja 2D bez rozmiaru porcji (4.5)

    double sum\_parallel=0.0;

#pragma omp parallel for schedule(static) default(none) shared(a) reduction(+:sum\_parallel) ordered

    for(int i = 0; i < WYMIAR; i++) {

      int id\_w = omp\_get\_thread\_num();

  #pragma omp parallel for schedule(static) firstprivate(id\_w) reduction(+:sum\_parallel) ordered

      for(int j = 0; j < WYMIAR; j++) {

        sum\_parallel += a[i][j];

        #pragma omp ordered

        {

          printf("(%1d,%1d)-W\_%1d,%1d ",i,j,id\_w, omp\_get\_thread\_num());

          if (j == WYMIAR - 1)

            printf("\n");

        }

      }

    }

  printf("Suma wyrazów tablicy równolegle: %lf\n\n", sum\_parallel);

}

1. Dekompozycja wierszowa

{ // 8. dekompozycja wierszowa - zrównoleglenie pętli wewnętrznej (5.0)

    double total\_sum = 0.0;

    double sums\_parallel[WYMIAR];

    #pragma omp parallel for schedule(static, 1) default(none) shared(a, sums\_parallel)

    for(int j = 0; j < WYMIAR; j++) {

      #pragma omp parallel for schedule(static, 1) default(none) shared(a, j, sums\_parallel)

      for(int i = 0; i < WYMIAR; i++)

        sums\_parallel[i] += a[i][j];

    }

    for (int i = 0; i < WYMIAR; i++)

      total\_sum += sums\_parallel[i];

    printf("Suma wyrazów tablicy równolegle: %lf\n\n", total\_sum);

  }

****Z racji na sposób obliczania zagnieżdżonych pętli, nie jesteśmy w stanie otrzymać takiego samego stylu wypisania do konsoli, poza poprawnie obliczonym wynikiem na koniec

**Wnioski:**

* OpenMP upraszcza równoległe przetwarzanie pętli, zapewniając łatwą w implementacji dyrektywę #pragma omp parallel for, która automatyzuje podział iteracji między wątki.
* Odpowiedni podział pracy (schedule) ma kluczowe znaczenie dla wydajności – różne strategie, takie jak static, dynamic czy guided, pozwalają dostosować program do specyfiki obciążenia.
* Testy wykazały, że przy dużej liczbie iteracji i odpowiednim rozkładzie zadań OpenMP może znacznie przyspieszyć wykonanie obliczeń w porównaniu do implementacji sekwencyjnej.
* Nadmierna liczba wątków może prowadzić do narzutu komunikacyjnego, co zmniejsza korzyści z równoległości – należy odpowiednio dostosować liczbę wątków do liczby dostępnych rdzeni.
* Analiza wyników pokazała, że optymalizacja kodu równoległego wymaga uwzględnienia kosztów synchronizacji i narzutu wynikającego z zarządzania wątkami.

**Lab10:**

**Cel:**

* Pogłębienie umiejętności pisania programów równoległych w środowisku OpenMP

**Zadanie:**

1. Przygotowanie projektu
2. Analiza *Opnemp\_watki\_zmienne*:

Wartości zmiennych nie są deterministyczne, dochodzi do modyfikacji danych krytycznych

1. Naprawa *Opnemp\_watki\_zmienne*:

// Stworzenie zmiennej prywatnej dla każdego wątku

int f\_threadprivate;

#pragma omp threadprivate(f\_threadprivate)

#pragma omp barrier // stworzenie bariery

    d\_local\_private = a\_shared + c\_firstprivate;

    #pragma omp critical // Oznaczenie ścieżki krytycznej

    for(i=0;i<10;i++){

      a\_shared ++;

    }

    for(i=0;i<10;i++){

      #pragma omp atomic // Oznaczenie operacji atomicznej

      e\_atomic+=omp\_get\_thread\_num();

    }

    #pragma omp critical

    {

      // Print w ścieżce krytycznej

      printf("\nw obszarze równoległym: aktualna liczba watkow %d, moj ID %d\n",

       omp\_get\_num\_threads(), omp\_get\_thread\_num());

      printf("\ta\_shared \t= %d\n", a\_shared);

      printf("\tb\_private \t= %d\n", b\_private);

      printf("\tc\_firstprivate \t= %d\n", c\_firstprivate);

      printf("\td\_local\_private = %d\n", d\_local\_private);

      printf("\te\_atomic \t= %d\n", e\_atomic);

    }

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

Po wykonaniu powyższych zmian, operacje się przewidywalne, a modyfikowanie zmiennych nie prowadzi do ich korupcji. Same print’y również są wykonywane pojedynczo.

1. Analiza zależności przenoszonych w pętli (*pde*):

A[i] = A[i] + A[i + 2] + sin(B[i])

Oznacza to, że obecny element A jest zależny obecnego A, 2. następnego elementu A oraz od obecnego elementu B.

1. Wprowadzenie tablicy tymczasowej, w celu pozbycia się zależności obecnego A od obecnego A

  double temp[N + 2];

  #pragma omp parallel for shared(A, B, temp) num\_threads(2)

  for(int i = 0; i < N; i++)

    temp[i] = A[i] + A[i+2] + sin(B[i]);

  #pragma omp parallel for shared(A, temp) num\_threads(2)

  for(int i = 0; i < N; i++)

    A[i] = temp[i];

Po pozbyciu się tej zależności, obliczania trwają o połowę mniej czasu



1. Uruchomienie programu wyszukiwania wartości maksymalnej w tablicy
2. Uzupełnienie funkcji *search\_max\_openmp\_task* obliczania maksimum w wersji równoległej OpenMP

#pragma omp task default(none) firstprivate(A, p\_task, k\_task) shared(a\_max)

{

  double local\_max = search\_max(A, p\_task, k\_task);

  #pragma omp critical (cs\_a\_max)

  if(a\_max < local\_max)

    a\_max = local\_max;

}

1. Modyfikacja funkcji *merge\_sort\_openmp\_2*:

int q1=(p+r)/2;

#pragma omp task default(none) firstprivate(A,p,r,q1,poziom) final(poziom > max\_poziom)

{

if(omp\_in\_final())

    sortowanie\_szybkie(A, p, q1);

  else

    merge\_sort\_openmp\_2(A,p,q1,poziom);

  }

#pragma omp task default(none) firstprivate(A,p,r,q1,poziom) final(poziom > max\_poziom)

{

  if(omp\_in\_final())

    sortowanie\_szybkie(A, q1 + 1, r);

  else

    merge\_sort\_openmp\_2(A,q1+1,r,poziom);

}

1. Implementacja Binary Search’a

double bin\_search\_max\_task(

  double\* A,

  int p,

  int r,

  int level

         )

{

  if (p >= r)

    return A[p];

  int q = (p + r) / 2;

  double leftMax, rightMax;

  if (level <= max\_level) {

    #pragma omp task shared(leftMax) firstprivate(A, p, q, level) default(none)

    leftMax = bin\_search\_max\_task(A, p, q, level + 1);

    #pragma omp task shared(rightMax) firstprivate(A, q, r, level) default(none)

    rightMax = bin\_search\_max\_task(A, q + 1, r, level + 1);

    #pragma omp taskwait

  }

  else {

    leftMax = search\_max(A, p, q);

    rightMax = search\_max(A, q + 1, r);

  }

  if(leftMax < rightMax)

    return rightMax;

  else

    return leftMax;

}

1. Obraz zawierający tekst, zrzut ekranu, Czcionka

   Opis wygenerowany automatycznieWyniki sortowań:
2. Zrównoleglenie mnożenia Matrix \* Vector:

void mat\_vec\_row\_row\_decomp(double\* a, double\* x, double\* y, int n)

{

  #pragma omp parallel for default(none) shared(a, x, y, n)

  for(int i=0;i<n;i++){

    y[i]=0.0;

    for(int j=0;j<n;j++){

      y[i]+=a[n\*i+j]\*x[j];

    }

  }

}

void mat\_vec\_row\_col\_decomp(double\* a, double\* x, double\* y, int n)

{

  #pragma omp parallel for default(none) shared(a, x, y, n)

  for(int i=0;i<n;i++){

    y[i]=0.0;

    #pragma omp parallel for reduction(+:y[i]) default(none) shared(a, x, n, i)

    for(int j=0;j<n;j++){

      y[i]+=a[n\*i+j]\*x[j];

    }

  }

}

void mat\_vec\_col(double\* a, double\* x, double\* y, int n)

{

  for(int i=0;i<n;i++) y[i]=0.0;

  for(int j=0;j<n;j++){

    for(i=0;i<n;i++){

      y[i]+=a[i+j\*n]\*x[j];

    }

  }

}

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

void mat\_vec\_col\_col\_decomp(double\* a, double\* x, double\* y, int n)

{

  for(int i=0;i<n;i++) y[i]=0.0;

  #pragma omp parallel shared(a,x,y,n) default(none)

  {

    double\* y\_local = calloc(n, sizeof(double));

    #pragma omp for

    for(int i=0;i<n;i++)

      for(int j=0;j<n;j++)

        y\_local[i] += a[i + j \* n] \* x[j];

    #pragma omp critical

    for(int i = 0; i < n; ++i)

      y[i] += y\_local[i];

    free(y\_local);

  }

}

void mat\_vec\_col\_row\_decomp(double\* a, double\* x, double\* y, int n)

{

  for(int i=0;i<n;i++) y[i]=0.0;

  for(int j=0;j<n;j++){

    #pragma omp parallel for default(none) shared(a, x, y, n, j)

    for(int i=0;i<n;i++){

      y[i]+=a[i+j\*n]\*x[j];

    }

  }

}

**Wnioski:**

* OpenMP pozwala na elastyczne zarządzanie zmiennymi w kontekście współbieżnym, oferując różne specyfikatory, takie jak private, shared, firstprivate i lastprivate.
* Właściwy wybór specyfikatora zmiennych jest kluczowy dla uniknięcia błędów, takich jak niespójność danych czy nieoczekiwane wartości w poszczególnych wątkach.
* Mechanizm redukcji (reduction) umożliwia bezpieczne równoległe wykonywanie operacji na zmiennych agregujących, co znacząco upraszcza kod i zwiększa wydajność.
* Testy pokazały, że niepoprawne zrozumienie zakresów zmiennych prowadzi do trudnych do zidentyfikowania błędów, takich jak konflikty w dostępie do danych.
* Przy równoległym przetwarzaniu istotne jest rozważenie kosztów kopiowania zmiennych w kontekście private oraz narzutu wynikającego z operacji redukcji.