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

```c
#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

```
przed bariera 1 - watek 0
przed bariera 1 - watek 1
przed bariera 1 - watek 2
przed bariera 1 - watek 3
po ostatniej barierze - watek 3
po ostatniej barierze - watek 1
po ostatniej barierze - watek 2
po ostatniej barierze - watek 0
```

3. Implemntacja Czytelni

```c
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);
}
```

```c
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);
}
```

Fragment output'u:

```
czytelnik 139638619706944 - wchodze
                                        czytam:  l_c 8, l_p 0
czytelnik 139638628099648 - przed zamkiem
czytelnik 139638628099648 - wchodze
                                        czytam:  l_c 9, l_p 0
czytelnik 139638661670464 - przed zamkiem
czytelnik 139638661670464 - wchodze
                                        czytam:  l_c 10, l_p 0
czytelnik 139638628099648 - wychodze
czytelnik 139638628099648 - po zamku
czytelnik 139638586136128 - wychodze
czytelnik 139638586136128 - po zamku
czytelnik 139638628099648 - przed zamkiem
czytelnik 139638628099648 - wchodze
```

4. Użycie *read_write_locks*

```c
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 licznikow
} 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 licznikow
  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 licznikow
  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;
}
```

```c
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);
}
```

```
czytelnik 140171959653952 - wchodze
                                    czytam:  l_c 10, l_p 0
czytelnik 140172026795584 - wychodze
czytelnik 140172026795584 - po zamku
pisarz 140172068759104 - przed zamkiem
czytelnik 140171968046656 - wychodze
czytelnik 140171968046656 - po zamku
czytelnik 140172018402880 - wychodze
czytelnik 140172018402880 - po zamku
pisarz 140172077151808 - przed zamkiem
czytelnik 140171968046656 - przed zamkiem
czytelnik 140171968046656 - wchodze
                                    czytam:  l_c 8, l_p 0
czytelnik 140172026795584 - przed zamkiem
czytelnik 140172026795584 - wchodze
```

5. Implementacja Bariery w Java

```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
        }
    }
}
```

6. Implementacja Czytelni w Java

```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. Uruchomienie 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);
  }
```

```
a[ 0]->W_0
a[ 1]->W_0
a[ 2]->W_0
a[ 3]->W_0
a[ 4]->W_0
a[ 5]->W_1
a[ 6]->W_1
a[ 7]->W_1
a[ 8]->W_1
a[ 9]->W_1
a[10]->W_2
a[11]->W_2
a[12]->W_2
a[13]->W_2
a[14]->W_3
a[15]->W_3
a[16]->W_3
a[17]->W_3
```

3. Przetestowanie 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);
  }
```

```
a[ 0]->W_0
a[ 1]->W_0
a[ 2]->W_0
a[ 3]->W_1
a[ 4]->W_1
a[ 5]->W_1
a[ 6]->W_2
a[ 7]->W_2
a[ 8]->W_2
a[ 9]->W_3
a[10]->W_3
a[11]->W_3
a[12]->W_0
a[13]->W_0
a[14]->W_0
a[15]->W_1
a[16]->W_1
a[17]->W_1
```

```
  { // 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);
  }
```

```
a[ 0]->W_0
a[ 1]->W_0
a[ 2]->W_0
a[ 3]->W_0
a[ 4]->W_0
a[ 5]->W_1
a[ 6]->W_1
a[ 7]->W_1
a[ 8]->W_1
a[ 9]->W_1
a[10]->W_2
a[11]->W_2
a[12]->W_2
a[13]->W_2
a[14]->W_3
a[15]->W_3
a[16]->W_3
a[17]->W_3
```

```
a[ 0]->W_1
a[ 1]->W_1
a[ 2]->W_2
a[ 3]->W_2
a[ 4]->W_3
a[ 5]->W_3
a[ 6]->W_0
a[ 7]->W_0
a[ 8]->W_1
a[ 9]->W_1
a[10]->W_2
a[11]->W_2
a[12]->W_3
a[13]->W_3
a[14]->W_0
a[15]->W_0
a[16]->W_1
a[17]->W_1
```

```
a[ 0]->W_2
a[ 1]->W_3
a[ 2]->W_0
a[ 3]->W_1
a[ 4]->W_2
a[ 5]->W_3
a[ 6]->W_0
a[ 7]->W_1
a[ 8]->W_2
a[ 9]->W_3
a[10]->W_0
a[11]->W_1
a[12]->W_2
a[13]->W_3
a[14]->W_0
a[15]->W_1
a[16]->W_2
a[17]->W_3
```

4. 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);
  }
```

```
(0,0)-W_0 (0,1)-W_0 (0,2)-W_0 (0,3)-W_0 (0,4)-W_0 (0,5)-W_0 (0,6)-W_0 (0,7)-W_0 (0,8)-W_0 (0,9)-W_0
(1,0)-W_0 (1,1)-W_0 (1,2)-W_0 (1,3)-W_0 (1,4)-W_0 (1,5)-W_0 (1,6)-W_0 (1,7)-W_0 (1,8)-W_0 (1,9)-W_0
(2,0)-W_1 (2,1)-W_1 (2,2)-W_1 (2,3)-W_1 (2,4)-W_1 (2,5)-W_1 (2,6)-W_1 (2,7)-W_1 (2,8)-W_1 (2,9)-W_1
(3,0)-W_1 (3,1)-W_1 (3,2)-W_1 (3,3)-W_1 (3,4)-W_1 (3,5)-W_1 (3,6)-W_1 (3,7)-W_1 (3,8)-W_1 (3,9)-W_1
(4,0)-W_2 (4,1)-W_2 (4,2)-W_2 (4,3)-W_2 (4,4)-W_2 (4,5)-W_2 (4,6)-W_2 (4,7)-W_2 (4,8)-W_2 (4,9)-W_2
(5,0)-W_2 (5,1)-W_2 (5,2)-W_2 (5,3)-W_2 (5,4)-W_2 (5,5)-W_2 (5,6)-W_2 (5,7)-W_2 (5,8)-W_2 (5,9)-W_2
(6,0)-W_0 (6,1)-W_0 (6,2)-W_0 (6,3)-W_0 (6,4)-W_0 (6,5)-W_0 (6,6)-W_0 (6,7)-W_0 (6,8)-W_0 (6,9)-W_0
(7,0)-W_0 (7,1)-W_0 (7,2)-W_0 (7,3)-W_0 (7,4)-W_0 (7,5)-W_0 (7,6)-W_0 (7,7)-W_0 (7,8)-W_0 (7,9)-W_0
(8,0)-W_1 (8,1)-W_1 (8,2)-W_1 (8,3)-W_1 (8,4)-W_1 (8,5)-W_1 (8,6)-W_1 (8,7)-W_1 (8,8)-W_1 (8,9)-W_1
(9,0)-W_1 (9,1)-W_1 (9,2)-W_1 (9,3)-W_1 (9,4)-W_1 (9,5)-W_1 (9,6)-W_1 (9,7)-W_1 (9,8)-W_1 (9,9)-W_1
```

    2) 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);
  }
```

```
(0,0)-W_2 (0,1)-W_1 (0,2)-W_0 (0,3)-W_2 (0,4)-W_1 (0,5)-W_0 (0,6)-W_2 (0,7)-W_1 (0,8)-W_0 (0,9)-W_2
(1,0)-W_2 (1,1)-W_1 (1,2)-W_0 (1,3)-W_2 (1,4)-W_1 (1,5)-W_0 (1,6)-W_2 (1,7)-W_1 (1,8)-W_0 (1,9)-W_2
(2,0)-W_2 (2,1)-W_1 (2,2)-W_0 (2,3)-W_2 (2,4)-W_1 (2,5)-W_0 (2,6)-W_2 (2,7)-W_1 (2,8)-W_0 (2,9)-W_2
(3,0)-W_2 (3,1)-W_1 (3,2)-W_0 (3,3)-W_2 (3,4)-W_1 (3,5)-W_0 (3,6)-W_2 (3,7)-W_1 (3,8)-W_0 (3,9)-W_2
(4,0)-W_2 (4,1)-W_1 (4,2)-W_0 (4,3)-W_2 (4,4)-W_1 (4,5)-W_0 (4,6)-W_2 (4,7)-W_1 (4,8)-W_0 (4,9)-W_2
(5,0)-W_2 (5,1)-W_1 (5,2)-W_0 (5,3)-W_2 (5,4)-W_1 (5,5)-W_0 (5,6)-W_2 (5,7)-W_1 (5,8)-W_0 (5,9)-W_2
(6,0)-W_1 (6,1)-W_2 (6,2)-W_1 (6,3)-W_0 (6,4)-W_2 (6,5)-W_1 (6,6)-W_0 (6,7)-W_2 (6,8)-W_1 (6,9)-W_0
(7,0)-W_2 (7,1)-W_1 (7,2)-W_0 (7,3)-W_2 (7,4)-W_1 (7,5)-W_0 (7,6)-W_2 (7,7)-W_1 (7,8)-W_0 (7,9)-W_2
(8,0)-W_1 (8,1)-W_0 (8,2)-W_2 (8,3)-W_1 (8,4)-W_0 (8,5)-W_2 (8,6)-W_1 (8,7)-W_0 (8,8)-W_2 (8,9)-W_1
(9,0)-W_2 (9,1)-W_0 (9,2)-W_1 (9,3)-W_2 (9,4)-W_0 (9,5)-W_1 (9,6)-W_2 (9,7)-W_0 (9,8)-W_1 (9,9)-W_2
Suma wyrazów tablicy równolegle: 913.500000
```

3) dekompozycja kolumnowa - zrównoleglenie pętli zewnętrznej

```
{ // 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);
}
```

```
(0,0)-W_0 (1,0)-W_0 (2,0)-W_0 (3,0)-W_0 (4,0)-W_0 (5,0)-W_0 (6,0)-W_0 (7,0)-W_0 (8,0)-W_0 (9,0)-W_0
(0,1)-W_0 (1,1)-W_0 (2,1)-W_0 (3,1)-W_0 (4,1)-W_0 (5,1)-W_0 (6,1)-W_0 (7,1)-W_0 (8,1)-W_0 (9,1)-W_0
(0,2)-W_0 (1,2)-W_0 (2,2)-W_0 (3,2)-W_0 (4,2)-W_0 (5,2)-W_0 (6,2)-W_0 (7,2)-W_0 (8,2)-W_0 (9,2)-W_0
(0,3)-W_0 (1,3)-W_0 (2,3)-W_0 (3,3)-W_0 (4,3)-W_0 (5,3)-W_0 (6,3)-W_0 (7,3)-W_0 (8,3)-W_0 (9,3)-W_0
(0,4)-W_1 (1,4)-W_1 (2,4)-W_1 (3,4)-W_1 (4,4)-W_1 (5,4)-W_1 (6,4)-W_1 (7,4)-W_1 (8,4)-W_1 (9,4)-W_1
(0,5)-W_1 (1,5)-W_1 (2,5)-W_1 (3,5)-W_1 (4,5)-W_1 (5,5)-W_1 (6,5)-W_1 (7,5)-W_1 (8,5)-W_1 (9,5)-W_1
(0,6)-W_1 (1,6)-W_1 (2,6)-W_1 (3,6)-W_1 (4,6)-W_1 (5,6)-W_1 (6,6)-W_1 (7,6)-W_1 (8,6)-W_1 (9,6)-W_1
(0,7)-W_2 (1,7)-W_2 (2,7)-W_2 (3,7)-W_2 (4,7)-W_2 (5,7)-W_2 (6,7)-W_2 (7,7)-W_2 (8,7)-W_2 (9,7)-W_2
(0,8)-W_2 (1,8)-W_2 (2,8)-W_2 (3,8)-W_2 (4,8)-W_2 (5,8)-W_2 (6,8)-W_2 (7,8)-W_2 (8,8)-W_2 (9,8)-W_2
(0,9)-W_2 (1,9)-W_2 (2,9)-W_2 (3,9)-W_2 (4,9)-W_2 (5,9)-W_2 (6,9)-W_2 (7,9)-W_2 (8,9)-W_2 (9,9)-W_2
```

4) 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);
  }
```

```
(0,0)-W_0,0 (0,1)-W_0,0 (0,2)-W_0,1 (0,3)-W_0,1 (0,4)-W_0,0 (0,5)-W_0,0 (0,6)-W_0,1 (0,7)-W_0,1 (0,8)-W_0,0 (0,9)-W_0,0
(4,0)-W_2,0 (4,1)-W_2,0 (4,2)-W_2,1 (4,3)-W_2,1 (4,4)-W_2,0 (4,5)-W_2,0 (4,6)-W_2,1 (4,7)-W_2,1 (4,8)-W_2,0 (4,9)-W_2,0
(2,0)-W_1,0 (2,1)-W_1,0 (2,2)-W_1,1 (2,3)-W_1,1 (2,4)-W_1,0 (2,5)-W_1,0 (2,6)-W_1,1 (2,7)-W_1,1 (2,8)-W_1,0 (2,9)-W_1,0
(1,0)-W_0,0 (1,1)-W_0,0 (1,2)-W_0,1 (1,3)-W_0,1 (1,4)-W_0,0 (1,5)-W_0,0 (1,6)-W_0,1 (1,7)-W_0,1 (1,8)-W_0,0 (1,9)-W_0,0
(3,0)-W_1,0 (3,1)-W_1,0 (3,2)-W_1,1 (3,3)-W_1,1 (3,4)-W_1,0 (3,5)-W_1,0 (3,6)-W_1,1 (3,7)-W_1,1 (3,8)-W_1,0 (3,9)-W_1,0
(5,0)-W_2,0 (5,1)-W_2,0 (5,2)-W_2,1 (5,3)-W_2,1 (5,4)-W_2,0 (5,5)-W_2,0 (5,6)-W_2,1 (5,7)-W_2,1 (5,8)-W_2,0 (5,9)-W_2,0
(6,0)-W_0,0 (6,1)-W_0,0 (6,2)-W_0,1 (6,3)-W_0,1 (6,4)-W_0,0 (6,5)-W_0,0 (6,6)-W_0,1 (6,7)-W_0,1 (6,8)-W_0,0 (6,9)-W_0,0
(8,0)-W_1,0 (8,1)-W_1,0 (8,2)-W_1,1 (8,3)-W_1,1 (8,4)-W_1,0 (8,5)-W_1,0 (8,6)-W_1,1 (8,7)-W_1,1 (8,8)-W_1,0 (8,9)-W_1,0
(7,0)-W_0,0 (7,1)-W_0,0 (7,2)-W_0,1 (7,3)-W_0,1 (7,4)-W_0,0 (7,5)-W_0,0 (7,6)-W_0,1 (7,7)-W_0,1 (7,8)-W_0,0 (7,9)-W_0,0
(9,0)-W_1,0 (9,1)-W_1,0 (9,2)-W_1,1 (9,3)-W_1,1 (9,4)-W_1,0 (9,5)-W_1,0 (9,6)-W_1,1 (9,7)-W_1,1 (9,8)-W_1,0 (9,9)-W_1,0
Suma wyrazów tablicy równolegle: 913.500000
```

5) rozważenie 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);
  }
```

```
(7,0)-W_2,0 (7,1)-W_2,0 (7,2)-W_2,0 (7,3)-W_2,0 (7,4)-W_2,1 (7,5)-W_2,1 (7,6)-W_2,1 (7,7)-W_2,2 (7,8)-W_2,2 (7,9)-W_2,2
(4,0)-W_1,0 (4,1)-W_1,0 (4,2)-W_1,0 (4,3)-W_1,0 (4,4)-W_1,1 (4,5)-W_1,1 (4,6)-W_1,1 (4,7)-W_1,2 (4,8)-W_1,2 (4,9)-W_1,2
(8,0)-W_2,0 (8,1)-W_2,0 (8,2)-W_2,0 (8,3)-W_2,0 (8,4)-W_2,1 (8,5)-W_2,1 (8,6)-W_2,1 (8,7)-W_2,2 (8,8)-W_2,2 (8,9)-W_2,2
(0,0)-W_0,0 (0,1)-W_0,0 (0,2)-W_0,0 (0,3)-W_0,0 (0,4)-W_0,1 (0,5)-W_0,1 (0,6)-W_0,1 (0,7)-W_0,2 (0,8)-W_0,2 (0,9)-W_0,2
(9,0)-W_2,0 (9,1)-W_2,0 (9,2)-W_2,0 (9,3)-W_2,0 (9,4)-W_2,1 (9,5)-W_2,1 (9,6)-W_2,1 (9,7)-W_2,2 (9,8)-W_2,2 (9,9)-W_2,2
(1,0)-W_0,0 (1,1)-W_0,0 (1,2)-W_0,0 (1,3)-W_0,0 (1,4)-W_0,1 (1,5)-W_0,1 (1,6)-W_0,1 (1,7)-W_0,2 (1,8)-W_0,2 (1,9)-W_0,2
(5,0)-W_1,0 (5,1)-W_1,0 (5,2)-W_1,0 (5,3)-W_1,0 (5,4)-W_1,1 (5,5)-W_1,1 (5,6)-W_1,1 (5,7)-W_1,2 (5,8)-W_1,2 (5,9)-W_1,2
(2,0)-W_0,0 (2,1)-W_0,0 (2,2)-W_0,0 (2,3)-W_0,0 (2,4)-W_0,1 (2,5)-W_0,1 (2,6)-W_0,1 (2,7)-W_0,2 (2,8)-W_0,2 (2,9)-W_0,2
(6,0)-W_1,0 (6,1)-W_1,0 (6,2)-W_1,0 (6,3)-W_1,0 (6,4)-W_1,1 (6,5)-W_1,1 (6,6)-W_1,1 (6,7)-W_1,2 (6,8)-W_1,2 (6,9)-W_1,2
(3,0)-W_0,0 (3,1)-W_0,0 (3,2)-W_0,0 (3,3)-W_0,0 (3,4)-W_0,1 (3,5)-W_0,1 (3,6)-W_0,1 (3,7)-W_0,2 (3,8)-W_0,2 (3,9)-W_0,2
Suma wyrazów tablicy równolegle: 913.500000
```

5. 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

```
Suma wyrazów tablicy równolegle: 913.500000
```

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

3. 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);
    }
```

```
        e_atomic        = 665

w obszarze równoległym: aktualna liczba watkow 12, moj ID 1
        a_shared        = 121
        b_private       = 0
        c_firstprivate  = 13
        d_local_private = 13
        e_atomic        = 665

w obszarze równoległym: aktualna liczba watkow 12, moj ID 0
        a_shared        = 121
        b_private       = 0
        c_firstprivate  = 3
        d_local_private = 34
        e_atomic        = 665
po zakonczeniu obszaru rownoleglego:
        a_shared        = 121
        b_private       = 2
        c_firstprivate  = 3
        e_atomic        = 665

================================================

Thread 1: f_threadprivate = 1
Thread 0: f_threadprivate = 0
Thread 2: f_threadprivate = 2
Thread 4: f_threadprivate = 4
Thread 3: f_threadprivate = 3
```

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

4. 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.

5. 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

```
suma 1459701.114868, czas obliczen 0.007483s
suma 1459701.114868, czas obliczen rownoleglych 0.004613s
```

6. Uruchomienie programu wyszukiwania wartości maksymalnej w tablicy
7. 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;
}
```

8. 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);
}
```

9. 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;
}
```

10. Wyniki sortowań:

```
maximal element 249999.950000
time for sequential linear search: 0.006035s
maximal element 249999.950000
time for parallel linear search: 0.006085s
maximal element 249999.950000
time for parallel linear search (tasks): 0.002859s
maximal element 249999.950000
time for sequential binary search: 0.016029s
maximal element 249999.950000
time for parallel binary search: 0.003003s
```

11. 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];
    }
  }
}
```

```c
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];
    }
  }
}
```

```
ROW MAJOR
time for one multiplication: 0.100637s, Gflop/s: 1.987340, GB/s: 7.949359
TEST ROW MAJOR
time for one multiplication: 0.095332s, Gflop/s: 2.097929, GB/s: 8.391715
COLUMN MAJOR
time for one multiplication: 0.056061s, Gflop/s: 3.567525, GB/s: 14.270101
TEST COLUMN MAJOR
time for one multiplication: 0.040029s, Gflop/s: 4.996400, GB/s: 19.985598
```

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.