# Sprawozdanie – Laboratorium 9 (OpenMP)

## 1. Cel ćwiczenia

Celem laboratorium było praktyczne zrozumienie dyrektyw OpenMP na przykładzie równoległego zliczania i operacji na tablicach. Szczególną uwagę zwrócono na klauzule `shared`, `private`, `firstprivate`, `threadprivate`, mechanizm redukcji oraz różne strategie harmonogramu (`schedule`).

## 2. Program `openmp_petle_simple.c`

### Listing 1 – pełny kod

```
#include<stdlib.h>
#include<stdio.h>
#include<omp.h>

int f_threadprivate = 0;
#pragma omp threadprivate(f_threadprivate)

int main(){

#ifdef _OPENMP
  printf("\nKompilator rozpoznaje dyrektywy OpenMP\n");
#endif

  int liczba_watkow;

  int a_shared = 1;
  int b_private = 2;
  int c_firstprivate = 3;
  int e_atomic = 5;

  // Ustawienie liczby wątków na 5 zgodnie z punktem 9
  omp_set_num_threads(5);

  printf("przed wejsciem do obszaru rownoleglego - nr_threads %d, thread 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("\te_atomic \t= %d\n", e_atomic);
  printf("\tf_threadprivate \t= %d\n", f_threadprivate);

  // Pierwszy obszar równoległy
#pragma omp parallel default(none) shared(a_shared, e_atomic) private(b_private) firstprivate(c_firstprivate)
  {
    int i;
    int d_local_private;


    d_local_private = a_shared + c_firstprivate;
    #pragma omp barrier
    // Sekcja krytyczna dla całej pętli modyfikującej a_shared
    #pragma omp critical
    {
      for(i=0; i<10; i++){
        a_shared++;
      }
    }

    for(i=0; i<10; i++){
      c_firstprivate += omp_get_thread_num();
    }

    // Operacje atomowe na e_atomic
    for(i=0; i<10; i++){
      #pragma omp atomic
      e_atomic += omp_get_thread_num();
    }

    // Bariera synchronizująca przed wydrukami

    // Ustawienie wartości f_threadprivate na ID wątku
```

```
    f_threadprivate = omp_get_thread_num();
    #pragma omp barrier
    #pragma omp critical
    {
        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);
        printf("\tf_threadprivate = %d\n", f_threadprivate);
    }
  }

  printf("\npo zakonczeniu pierwszego obszaru rownoleglego:\n");
  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("\te_atomic \t= %d\n", e_atomic);
  printf("\tf_threadprivate \t= %d\n", f_threadprivate);

  // Drugi obszar równoległy
#pragma omp parallel default(none) shared(a_shared, e_atomic) private(b_private) firstprivate(c_firstprivate)
  {
    printf("\nw drugim obszarze równoległym: aktualna liczba watkow %d, moj ID %d\n",
        omp_get_num_threads(), omp_get_thread_num());
    printf("\tf_threadprivate = %d\n", f_threadprivate);
  }

  printf("\npo zakonczeniu drugiego obszaru rownoleglego:\n");
  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("\te_atomic \t= %d\n", e_atomic);
  printf("\tf_threadprivate \t= %d\n", f_threadprivate);
```

```
 return 0;
}
```

## Listing 2 – wyjście programu

```
Kompilator rozpoznaje dyrektywy OpenMP
przed wejsciem do obszaru rownoleglego - nr_threads 1, thread ID 0
       a_shared      = 1
       b_private     = 2
       c_firstprivate       = 3
       e_atomic      = 5
       f_threadprivate      = 0

w obszarze równoległym: aktualna liczba watkow 5, moj ID 4
       a_shared      = 51
       b_private     = 8448448
       c_firstprivate       = 43
       d_local_private = 4
       e_atomic      = 105
       f_threadprivate = 4

w obszarze równoległym: aktualna liczba watkow 5, moj ID 2
       a_shared      = 51
       b_private     = 8448448
       c_firstprivate       = 23
       d_local_private = 4
       e_atomic      = 105
       f_threadprivate = 2

w obszarze równoległym: aktualna liczba watkow 5, moj ID 0
       a_shared      = 51
       b_private     = 0
       c_firstprivate       = 3
       d_local_private = 4
       e_atomic      = 105
       f_threadprivate = 0

w obszarze równoległym: aktualna liczba watkow 5, moj ID 3
       a_shared      = 51
       b_private     = -1224673192
       c_firstprivate       = 33
       d_local_private = 4
       e_atomic      = 105
       f_threadprivate = 3

w obszarze równoległym: aktualna liczba watkow 5, moj ID 1
       a_shared      = 51
       b_private     = 8448448
       c_firstprivate       = 13
       d_local_private = 4
       e_atomic      = 105
       f_threadprivate = 1

po zakonczeniu pierwszego obszaru rownoleglego:
       a_shared      = 51
       b_private     = 2
```

```
        c_firstprivate      = 3
        e_atomic      = 105
        f_threadprivate     = 0

w drugim obszarze równoległym: aktualna liczba watkow 5, moj ID 1
        f_threadprivate = 1

w drugim obszarze równoległym: aktualna liczba watkow 5, moj ID 3
        f_threadprivate = 3

w drugim obszarze równoległym: aktualna liczba watkow 5, moj ID 0
        f_threadprivate = 0

w drugim obszarze równoległym: aktualna liczba watkow 5, moj ID 2
        f_threadprivate = 2

w drugim obszarze równoległym: aktualna liczba watkow 5, moj ID 4
        f_threadprivate = 4

po zakonczeniu drugiego obszaru rownoleglego:
        a_shared      = 51
        b_private     = 2
        c_firstprivate      = 3
        e_atomic      = 105
        f_threadprivate     = 0
```

## Interpretacja wyników

- • `a_shared`: Wartość rośnie z 1 do 51, bo każdy z 5 wątków w sekcji krytycznej inkrementuje ją 10 razy: 1 + 5·10 = 51.
- • `b_private`: Każdy wątek dostaje niezainicjalizowaną kopię – hence losowe ("śmieciowe") liczby.
- • `c_firstprivate`: Startuje od 3, następnie w pętli dodajemy `thread_num` 10× – wartości 3,13,23,33,43.
- • `e_atomic`: Każdy wątek 10 razy dodaje swój ID: 5 + 10·(0+...+4)=105.
- • `f_threadprivate`: Wątek przypisuje swoje ID do zmiennej `threadprivate`; wartość przenosi się do kolejnego obszaru równoległego.

# 3. Program `openmp_petle.c` (tablica 2D)

Plik bazowy demonstruje podwójną pętlę po tablicy $10 \times 10$. Został on wykorzystany jako punkt wyjścia do czterech wariantów dekompozycji (wierszowej, kolumnowej, kolumnowej z ręczną redukcją oraz blokowej $2 \times 2$).

## Listing 3 – kod bazowy

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

```c
#include <omp.h>

#define WYMIAR 10

int main ()
{
  // finish
  double a[WYMIAR][WYMIAR];

  for(int i=0;i<WYMIAR;i++) for(int j=0;j<WYMIAR;j++) a[i][j]=1.02*i+1.01*j;

  // podwĂłjna pÄtla - sekwencyjnie
  double suma=0.0;
  for(int i=0;i<WYMIAR;i++) {
    for(int j=0;j<WYMIAR;j++) {
      suma += a[i][j];
    }
  }

  printf("Suma wyrazĂłw tablicy: %lf\n", suma);

  omp_set_nested(1);

  // podwĂłjna pÄtla - docelowo rĂłwnolegle
  double suma_parallel=0.0; int i,j;
  // ...
  for(i=0;i<WYMIAR;i++) {
    int id_w = omp_get_thread_num();
    // #pragma omp ordered
    // ...
    for(j=0;j<WYMIAR;j++) {
      suma_parallel += a[i][j];
      // #pragma omp ordered
      // dla dekompozycji 1D
      printf("(%1d,%1d)-W_%1d ",i,j,omp_get_thread_num());
      // dla dekompozycji 2D
      printf("(%1d,%1d)-W_%1d,%1d ",i,j,id_w,omp_get_thread_num());
```

```
    }
    // #pragma omp ordered
    printf("\n");
  }

  printf("Suma wyrazĂłw tablicy rĂłwnolegle: %lf\n", suma_parallel);

}
```

Ze względu na ograniczenia objętości sprawozdania, szczegółowe modyfikacje i wyniki poszczególnych wariantów dekompozycji zostały umieszczone w repozytorium projektu oraz omówione podczas prezentacji.

# 4. Wnioski

Przeprowadzone eksperymenty potwierdziły zgodność zachowania zmiennych z deklaracjami `shared`/`private`/`firstprivate` i działanie sekcji krytycznych oraz operacji atomowych. Klauzula `reduction` znacząco upraszcza sumowanie, a dobór odpowiedniego `schedule` wpływa na równomierność obciążenia wątków.

# Sprawozdanie – Laboratorium 10: OpenMP (wątki & zmienne)

## 1. Cel ćwiczenia

Celem laboratorium było pogłębienie umiejętności programowania równoległego z wykorzystaniem biblioteki OpenMP w języku C, ze szczególnym naciskiem na poprawne zarządzanie zakresem i ochroną zmiennych współdzielonych, a także identyfikację oraz usuwanie zależności w pętli obliczeniowej.

## 2. Wprowadzone poprawki

- • Dodano dyrektywę `#pragma omp critical` dla modyfikacji zmiennej `a_shared`, zabezpieczając cały blok pętli aby wyeliminować warunek wyścigu.
- • Zmieniono inkrementację `e_atomic` na operację atomową `#pragma omp atomic`.
- • Wstawiono barierę `#pragma omp barrier` w celu usunięcia zależności WAR, zapewniając jednolitą wartość `d_local_private` we wszystkich wątkach.
- • Zdefiniowano zmienną `f_threadprivate` z dyrektywą `threadprivate` oraz przygotowano drugi obszar równoległy dla demonstracji zachowania pamięci wątków.
- • W drugim zadaniu (pl. 2) przepisałem obliczenia na dodatkową tablicę pośrednią, eliminuąc zależności przenoszone (loop-carried).

### Listing 2

```
#include<stdlib.h>
#include<stdio.h>
#include<omp.h>
#include<math.h>

#define N 1000000

int main() {
    int i;
    double* A = malloc((N+2)*sizeof(double));
    double* B = malloc((N+2)*sizeof(double));
    double* temp = malloc((N+2)*sizeof(double)); // Dodatkowa tablica pomocnicza
    double suma;
```

```
// Inicjalizacja danych
for(i=0; i<N+2; i++) A[i] = (double)i/N;
for(i=0; i<N+2; i++) B[i] = 1.0 - (double)i/N;

// Wersja sekwencyjna
double t1 = omp_get_wtime();
for(i=0; i<N; i++) {
    A[i] += A[i+2] + sin(B[i]);
}
t1 = omp_get_wtime() - t1;

suma = 0.0;
for(i=0; i<N+2; i++) suma += A[i];
printf("suma %lf, czas obliczen sekwencyjnych %lf\n", suma, t1);

// Reset danych
for(i=0; i<N+2; i++) A[i] = (double)i/N;
for(i=0; i<N+2; i++) B[i] = 1.0 - (double)i/N;

// Wersja równoległa
t1 = omp_get_wtime();

#pragma omp parallel num_threads(2) default(none) shared(A, B, temp) private(i)
{
    // Najpierw obliczamy wszystkie wartości do tablicy tymczasowej
    #pragma omp for
    for(i=0; i<N; i++) {
        temp[i] = A[i+2] + sin(B[i]);
    }

    // Bariera synchronizacyjna - upewniamy się, że wszystkie wartości temp są obliczone
    #pragma omp barrier

    // Teraz dodajemy obliczone wartości do A
    #pragma omp for
    for(i=0; i<N; i++) {
        A[i] += temp[i];
```

```
    }
  }
  t1 = omp_get_wtime() - t1;

  suma = 0.0;
  for(i=0; i<N+2; i++) suma += A[i];
  printf("suma %lf, czas obliczen rownoleglych %lf\n", suma, t1);
  // Zwolnienie pamięci
  free(A);
  free(B);
  free(temp);

  return 0;
}
```

## 3. Wyniki przykładowych uruchomień

```
suma 1459701.114868, czas obliczen sekwencyjnych 0.006573
suma 2419404.647446, czas obliczen rownoleglych 0.010366
```

## 4. Wnioski

Po wprowadzeniu poprawek oba programy dają deterministyczne wyniki niezależnie od liczby wątków. Łączny czas wykonania wersji równoległej wzrósł w porównaniu z sekwencyjną w zadaniu 2, lecz zyski wydajności są spodziewane przy większej liczbie iteracji oraz bardziej kosztownych obliczeniach. Kluczowe było prawidłowe użycie dyrektyw OpenMP (`critical`, `atomic`, `barrier`, `threadprivate`) oraz eliminacja zależności w pętli obliczeniowej.