

OpenMP

1. Wprowadzenie	2
2. Model Programowania	3
3. Sterowanie równoległością	5
3.1 Pragma parallel	5
3.2 Pragmy sterujące podziałem pracy	8
4. Specyfikacje widoczności danych	13
4.1 Konstrukcje synchronizacyjne	18
4.2 Pragma master	18
4.3 Pragma critical	18
4.4 Pragma ordered	19
4.5 Pragma atomic	20
4.6 Pragma flush	20
4.7 Pragma barrier	21
5. Funkcje biblioteczne	23
5.1 Testowanie i ustawianie środowiska	23
5.2 Synchronizacji wątków	26
5.3 Funkcje pomiaru czasu	28
6. Zmienne środowiska	29
7. Przykład	30
8. Źródła	31

1.Wprowadzenie

OpenMP jest interfejsem aplikacji API (ang. *Application Program Interface*) opracowanym przez wiodących wytwórców sprzętu i oprogramowania. Definiuje on przenośny i skalowalny model programowania aplikacji równoległych dla maszyn złożonych z procesorów komunikujących się przez pamięć dzieloną.

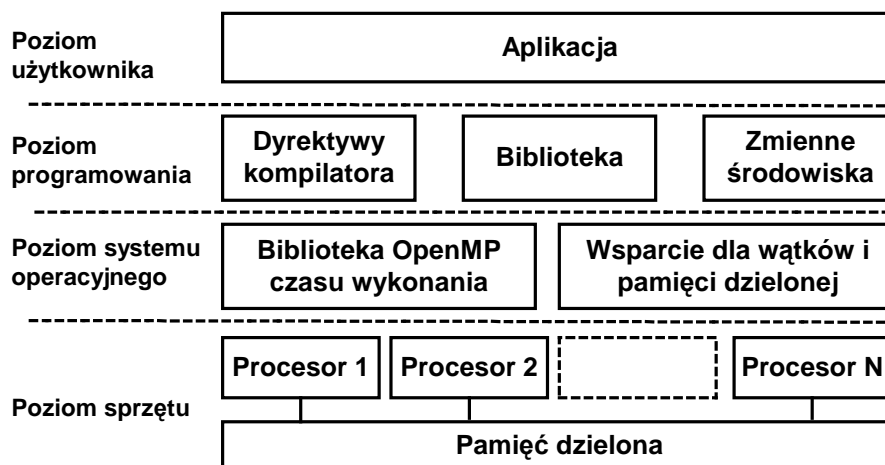
API opracowano dla języków C, C++ i Fortran dla wielu systemów operacyjnych włączając Windows, Unix i Linux.

Cechy:

- Przeznaczony dla programowania wielowątkowego maszyn z pamięcią dzieloną
- Przenośność – zdefiniowany dla C, C++, Fortran
- Standaryzacja – obecna wersja OpenMP 3.0 z 2008 roku
- Wspierany przez Intel, HP, IBM, Sun i innych

Główne elementy OpenMP

- Dyrektywy kompilatora
- Funkcje i biblioteki
- Zmienne środowiska



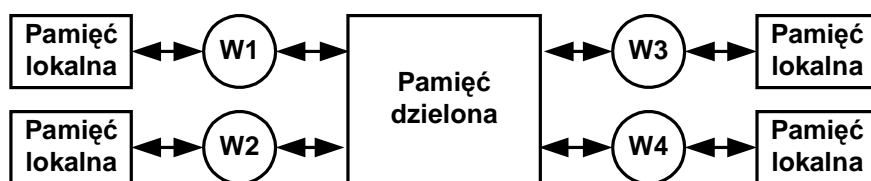
Rys. 1-1-1 Struktura OpenMP

Historia:

1997 – Wersja 1.0 dla Fortran
1998 – Wersja 1.0 dla C, C++
1999 - Wersja 1.1 dla Fortran
2000 - Wersja 2.0 dla Fortran
2002 - Wersja 2.0 dla C, C++
2005 - Wersja 2.5
2008 - Wersja 3.0

2.Model Programowania

OpenMP oparty jest na modelu wielu wykonujących się współbieżnie wątków komunikujących się przez pamięć dzieloną.



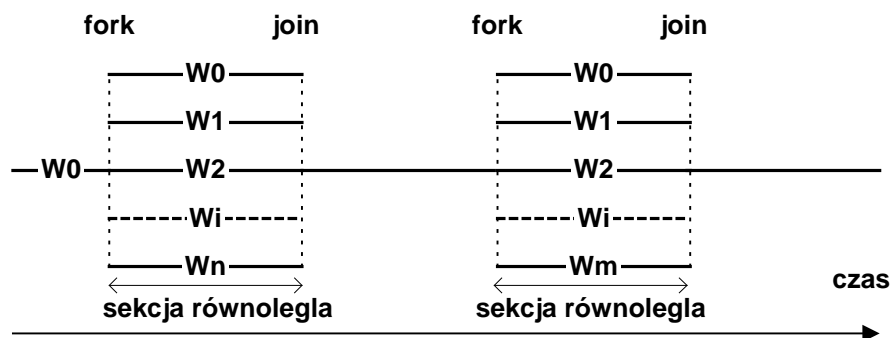
Rys. 2-2-1 Wątki komunikują się przez pamięć dzieloną

- Wszystkie wątki mają dostęp do tego samego obszaru pamięci dzielonej
- Każdy wątek może mieć pamięć lokalną dostępną tylko dla niego.

Równoległość określona jest bezpośrednio (ang. *Explicit Parallelism*) – nie stosuje się automatycznego dzielenia problemu na wątki.

Model programowania typu *fork – join*:

- Aplikacja zaczyna się jako jednowątkowa – wykonywany wątek główny W0.
- Gdy jest potrzebna operacja fork tworzy grupę wykonywanych współbieżnie wątków W1 – Wn. Wątek W0 jest wątkiem głównym.
- Zakończenie wątków jest synchronizowane – operacja join. Wszystkie wątki grupy czekają na zakończenie ostatniego wątku. Na końcu bloku synchronizacja typu bariera.
- Dalej wykonywany jest wątek główny.



Rys. 2-2-2 Model programowania fork – join

Sekcje równoległe określane przez dyrektywy języka C, C++, Fortran.

Możliwość zagnieżdżania sekcji równoległych zależy od implementacji ale standard dopuszcza taką możliwość.

```
#include <omp.h>
main () {
    int var1, var2, var3;
    // Kod sekwencyjny
    . . .
    // Początek części równoległej - podział na wątki
    // Specyfikacja zasięgu zmiennych
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        // Część wykonywana równoległe przez wszystkie wątki
        . . .
        // Wszystkie wątki dołączają do wątku głównego
    }
    // Wznowienie części sekwencyjnej
    . . .
}
```

Przykład 2-2-1 Przykładowa struktura programu w OpenMP

3. Sterowanie równoległością

Większość konstrukcji OpenMP ma postać dyrektyw.

```
#pragma omp dyrektywa [klauzula, [klauzula],...]
```

Przykład:

```
#pragma omp parallel num_threads(4)
```

Prototypy funkcji są w pliku nagłówkowym <omp.h>

Dyrektywy odnoszą się do bloku strukturalnego. Blok strukturalny jest blokiem języka C (ograniczonym klamrami {...}) mający jedno wejście i jedno wyjście. W bloku można stosować funkcję exit(...).

Kompilacja w gcc:

```
gcc mhello.c -o mhello -fopenmp
```

```
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    printf("Jestem watek %d, z %d\n",
        omp_get_thread_num(), omp_get_num_threads());
}
```

Przykład 3-3-1 Program Hello w OpenMP

```
Jestem watek 0 z 4
Jestem watek 1 z 4
Jestem watek 3 z 4
Jestem watek 2 z 4
```

Przykład 3-3-2 Wynik program Hello w OpenMP

3.1 Pragma parallel

Gdy wątek napotka konstrukcję `parallel` tworzy grupę wątków wykonywanych współbieżnie. Wątek początkowy nazwany jest wątkiem głównym (ang. *Master Thread*) i otrzymuje numer 0. Pozostałe wątki otrzymują kolejne numery 1,2,3,...

```
#pragma omp parallel [clause[ [, ]clause] ...] NL
Blok strukturalny
```

Gdzie klauzule są dane poniżej:

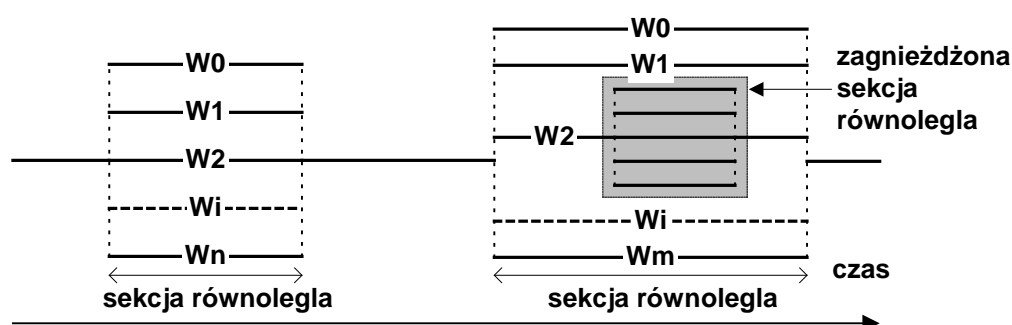
- `if(wyrażenie-skalarne)`
- `num_threads(wyrażenie-całkowite)`
- `default(shared | none)`
- `private(lista)`
- `firstprivate(lista)`
- `shared(lista)`
- `copyin(lista)`
- `reduction(operator: lista)`

Liczba wątków uruchomionych w sekcji równoległej zależy od:

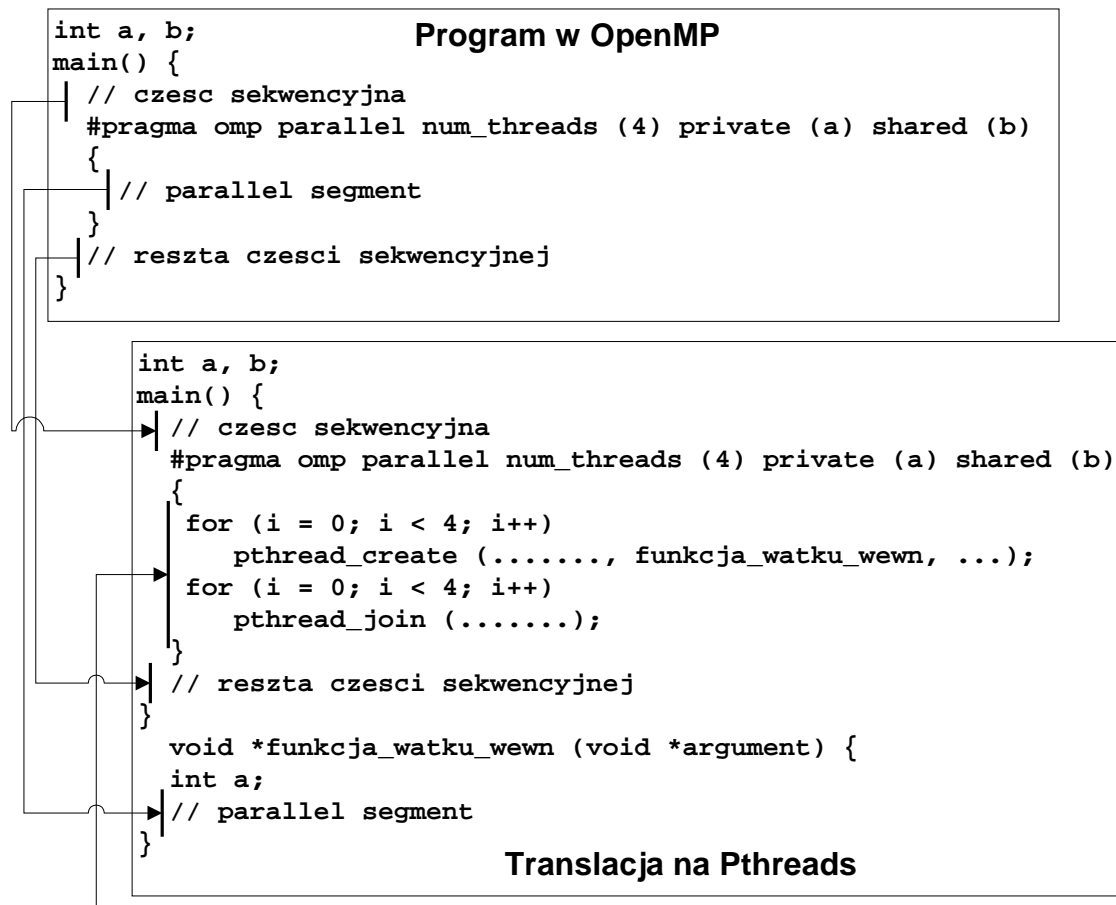
- Klauzuli `if(...)`
- Klauzuli `num_threads`
- Parametrów wywołania funkcji `omp_set_num_threads()`
- Ustawienia zmiennej środowiska `OMP_NUM_THREADS`
- Środowiska sprzętowego – liczby procesorów

Klauzula `if(wyrażenie)` – gdy wartość wyrażenia jest różna od 0 to utworzonych będzie wiele wątków, gdy nie – jeden.

Gdy wątek wykonywany w ramach konstrukcji `parallel` napotka inną konstrukcję `parallel` to może utworzyć nową zagnieżdżoną grupę wątków (zależy to od implementacji).



Rys. 3-3-1 Zagnieżdżone sekcje równoległe



**Wstawione przez
kompilator OpenMP**

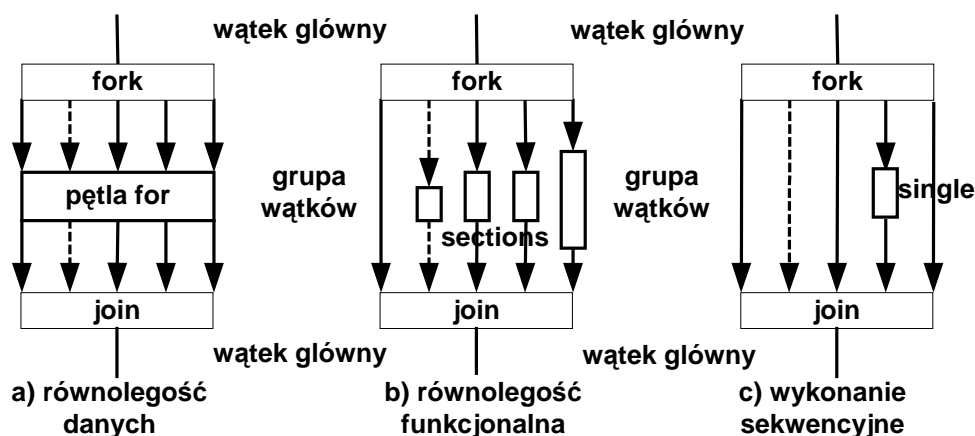
Przykład 3-3 Translacja programu OpenMP na program standardu Pthreads

3.2 Pragmy sterujące podziałem pracy

- Konstrukcje podziału zadania (ang. *Work Sharing Construction*) dzielą napotkane fragmenty kodu pomiędzy wątki grupy które napotkały tę konstrukcję.
- Konstrukcje podziału zadania nie tworzą nowych wątków – wykorzystują wątki istniejące w grupie.
- Nie ma bariery na początku konstrukcji ale jest na końcu.
- Konstrukcje podziału zadania muszą być umieszczone w sekcji **parallel**.

W OpenMP zdefiniowano trzy konstrukcje podziału pracy:

- Równoległość danych – **#pragma for**
- Równoległość funkcjonalna – **#pragma sections**
- Aplikacja sekwencyjna – **#pragma single**



Rys. 3-3-2 Konstrukcje podziału pracy w OpenMP

Pragma for

Pragma informuje system że iteracje w danym bloku muszą być wykonane równolegle przez grupę wątków. Pragma ta musi być już umieszczona w zdefiniowanym wcześniej bloku równoległym.

```
#pragma omp for[clause[ [, ]clause] ...] NL
  Blok strukturalny
```

Gdzie klauzule są dane poniżej:

- **schedule** (**typ** [,**chunk**])
- **private** (**lista_zmiennych**)
- **firstprivate** (**lista_zmiennych**)
- **lastprivate** (**list_zmiennych**)
- **shared** (**list_zmiennych**)
- **reduction** (**operator: lista**)

Klauzula **schedule** specyfikuje sposób podziału pętli na fragmenty wykonywane równolegle. Definiowane opcje: **static**, **dynamic**, **guided**, **runtime**, **auto**. Zmienna **chunk** określa wielkość równolegle przetwarzanego indeksu.

static	Iteracje są dzielone na fragmenty długości chunk i statycznie przypisywane do wątków. Gdy fragmentów jest więcej niż wątków przydział następuje według algorytmu karuzelowego (ang. <i>round robin</i>).
dynamic	Iteracje są dzielone na fragmenty długości chunk i statycznie przypisywane do wątków. Gdy wątek wykona zadanie pobiera następny fragment.
auto	Podział zadania dokonywany jest przez system operacyjny.

Tab. 3-3-1 Niektóre opcje klauzuli **schedule**

Na końcu bloku for umieszczana jest niejawnie bariera (o ile nie wystąpiła klauzula **nowait**). Gdy wątek zakończy pracę czeka na zakończenie pozostałych wątków.

Gdy wystąpiła klauzula **nowait**, to wątek który zakończył wykonywanie bloku przechodzi do następnych zadań.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 80

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

    /* Inicjalizacja */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk)
    private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Liczba watkow = %d\n", nthreads);
        }
        printf("Watek %d startuje...\n",tid);

        #pragma omp for schedule(dynamic,chunk)
        for (i=0; i<N; i++) {
            c[i] = a[i] + b[i];
            printf("Watek %d: c[%d]= %f\n",tid,i,c[i]);
        }
    } /* Koniec sekcji rownoleglej */
}
```

Przykład 3-3-4 Wykorzystanie pragmy for

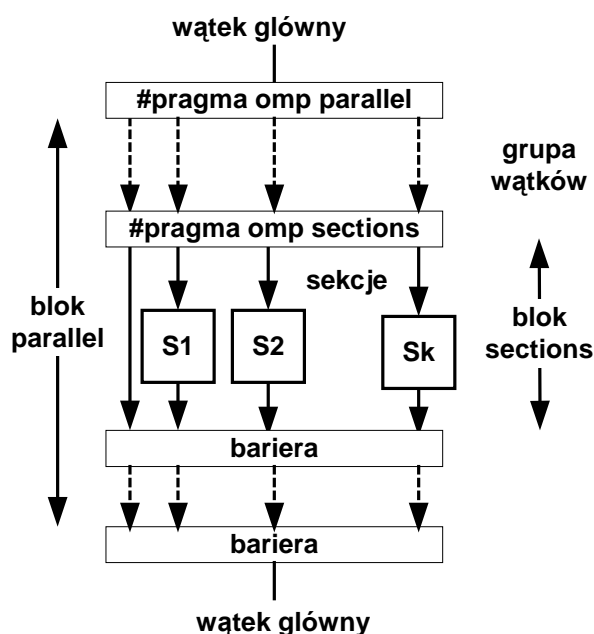
Ograniczenia dla pragmy for:

- Pętle for nie mogą zawierać instrukcji **break**
- Zmienna sterująca pętli musi być typu **int**
- Inicjalizacja zmiennej sterującej pętli musi być typu **int**
- Wyrażenia logiczne wewnątrz pętli mogą być tylko typu **<**, **<=**, **>**, **>=**
- Zmienna sterująca pętli może być zmniejszana (zwiększana) tylko o wielkość całkowitą

Pragma sections

Pragma section jest nie iteracyjną metodą podziału kodu pomiędzy wątki. Zdefiniowane sekcje mogą być wykonywane równolegle przez wątki.

```
#pragma omp sections [clause ...] newline
private (list) firstprivate (list) nowait
lastprivate (list) reduction (operator: list)
{
    #pragma omp section  newline
    blok_strukturalny
    #pragma omp section  newline
    blok_strukturalny
    ...
}
```



Rys. 3-3 Pragma sections implementuje równoległość funkcjonalną, sekcje S1, S2,..., Sk wykonywane są równoległe.

```
#include <omp.h>
#define N      1000

main (){
    int i;
    float a[N], b[N], c[N], d[N];
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
        } /* Koniec sekcji */
    } /* Koniec bloku rownoleglego */
}
```

Przykład 3-3-5 Dekompozycja funkcjonalna

Pragma single

Pragma **single** specyfikuje blok który powinien być wykonany tylko przez jeden wątek grupy. Może być wykorzystany do zabezpieczenia fragmentu kodu który musi być wykonany w trybie wyłącznym (ang. *thread safe*) jak np. operacja wejścia wyjścia.

```
#pragma omp single [clause ...] newline
private(lista_zmiennych)
firstprivate(lista_zmiennych) nowait
blok_strukturalny
```

4. Specyfikacje widoczności danych

Komunikacja między wątkami zachodzi poprzez zmienne dzielone.

Niewłaściwe ich użycie może (na skutek wyścigów) dać nieprzewidziane rezultaty. Stąd użycie zmiennych dzielonych powinno być synchronizowane.

Klauzule określające zakres widoczności danych (ang. *Data Scope Attribute Clauses*) są następujące:

- **private,**
- **firstprivate,**
- **lastprivate,**
- **shared,**
- **default,**
- **threadprivate**

Występują one z połączeniu z dyrektywami: **parallel**, **for**, **scope**

Definiują:

- Które zmienne z sekwencyjnej części programu mają być przekazane do części równoległej i w jaki sposób.
- Które zmienne mają być widoczne dla wszystkich wątków w części równoległej a które mają być dla wątków prywatne.

private(lista-zmiennych)

- Dla każdego wątku tworzona jest kopia zmiennej.
- Odniesienie do początkowej zmiennej zmieniane są na odniesienia do utworzonych zmiennych.
- Zmienne pozostają nie zainicjalizowane.

```
...
#pragma omp parallel shared(a,b,c,chunk) private(i,j,k)
  #pragma omp for schedule (static, chunk)
  for (i=0; i<N; i++)
  {
    for(j=0; j<N; j++)
      for (k=0; k<N; k++)
        c[i][j] += a[i][k] * b[k][j];
  }
} /* Koniec obszaru rownoleglego */
```

Każdy wątek ma niezależną kopię zmiennych i,j,k

firstprivate (list)

Zmienna tak jak private ale na początku bloku inicjowana na wartość tej zmiennej z części sekwencyjnej która wystąpiła tuż przed wejściem do bloku równoległego.

lastprivate (list)

Zmienna tak jak private ale po zakończeniu bloku przyjmuje wartość nadaną przez wątek który wykonuje się jako ostatni w bloku.

shared(lista-zmiennych)

- Zmienne wymienione na liście są widoczne we wszystkich wątkach.
- Programista powinien zabezpieczyć ochronę sekcji krytycznej jeżeli jest taka potrzeba.

```
...
#pragma omp parallel shared(a,b,c,chunk) private(i,j,k)
  #pragma omp for schedule (static, chunk)
  for (i=0; i<N; i++)
  {
    for(j=0; j<N; j++)
      for (k=0; k<N; k++)
        c[i][j] += a[i][k] * b[k][j];
  }
} /* Koniec obszaru rownoleglego */
```

Wszystkie wątki mają dostęp do współdzielonych tablic a,b,c

default (shared | none)

Specyfikuje domyślny typ zmiennych wewnątrz bloku równoległego.

- **default (shared)** - zmienne zadeklarowane na wyższym poziomie są domyślnie widoczne w bloku jako zmienne dzielone.
- **default (none)** - Nie jest określony domyślny sposób traktowania zmiennych

reduction (operator: lista)

Działanie:

- Dla każdego wątku tworzona jest prywatna kopia zmiennej występującej na liście.
- Po zakończeniu bloku równoległego na wszystkich kopiach wykonywana jest operacje redukcji zadana przez operator.

```
#include <omp.h>
main () {
    int    i, n, chunk;
    float a[100], b[100], result;
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }

    #pragma omp parallel for default(shared) private(i) \
        schedule(static,chunk) reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
}
```

Przykład 4-4-1 Operacja redukcji

Ograniczenia:

Zmienne muszą być typu skalarnego (wykluczone są tablice i struktury). Operacje: dodawanie, odejmowanie, wyrażenia.

threadprivate(lista-zmiennych)

Zmienne są prywatne dla wątku ale pozostają trwałe dla wątku po przejściu przez wiele bloków równoległych.

```

#include <omp.h>
int  a, b, i, tid;
float x;
#pragma omp threadprivate(a, x)

main () {

/* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);

  printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
  {
    tid = omp_get_thread_num();
    a = tid;
    b = tid;
    x = 1.1 * tid +1.0;
    printf("Thread %d:  a,b,x= %d %d %f\n",tid,a,b,x);
  } /* end of parallel section */

  printf("*****\n");
  printf("Master thread doing serial work here\n");
  printf("*****\n");

  printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Thread %d:  a,b,x= %d %d %f\n",tid,a,b,x);
  } /* end of parallel section */

}

```

Przykład 4-2 Użycie klauzuli threadprivate

```

1st Parallel Region:
Thread 0:  a,b,x= 0 0 1.000000
Thread 2:  a,b,x= 2 2 3.200000
Thread 3:  a,b,x= 3 3 4.300000
Thread 1:  a,b,x= 1 1 2.100000
*****
Master thread doing serial work here
*****
2nd Parallel Region:
Thread 0:  a,b,x= 0 0 1.000000
Thread 3:  a,b,x= 3 0 4.300000
Thread 1:  a,b,x= 1 0 2.100000
Thread 2:  a,b,x= 2 0 3.200000

```

Przykład 4-3 Użycie klauzuli threadprivate - wyniki


```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
/* liczba wierszy i kolumn w macierzach a,b,c */
#define N 100

int main (int argc, char *argv[]) {
int    tid, nthreads, i, j, k, chunk;
double
    a[N][N],          /* A - macierz zrodlowa */
    b[N][N],          /* B - macierz zrodlowa */
    c[N][N];          /* C - macierz wynikowa */

/* ustalenie liczby iteracji przydzielanej jednorazowo
dla watka */
chunk = 10;

#pragma omp parallel shared(a,b,c,nthreads,chunk)
private(tid,i,j,k)
{
    tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Start - watekow %d\n",nthreads);
    }
    #pragma omp for schedule (static, chunk)
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            a[i][j]= i+j;
    #pragma omp for schedule (static, chunk)
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            b[i][j]= i*j;
    #pragma omp for schedule (static, chunk)
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            c[i][j]= 0;
    #pragma omp for schedule (static, chunk)
    for (i=0; i<N; i++)
    {
        printf("Watek =%d mnozy wiersz=%d\n",tid,i);
        for(j=0; j<N; j++)
            for (k=0; k<N; k++)
                c[i][j] += a[i][k] * b[k][j];
    }
} /* Koniec obszaru rownoleglego */
}
```

Przykład 4-4-4 Równoległe mnożenie macierzy

4.1 Konstrukcje synchronizacyjne

<code>#pragma omp master</code>	Wykonanie tylko przez wątek główny
<code>#pragma omp critical</code>	Wykonanie w trybie wzajemnego wykluczania
<code>#pragma omp ordered</code>	Wykonanie uporządkowane
<code>#pragma omp atomic</code>	Wykonanie jako instrukcji atomowej
<code>#pragma omp flush</code>	Synchronizacja pamięci
<code>#pragma omp barrier</code>	Bariera

Tabela 4-1 Dyrektywy synchronizacyjne w OpenMP

4.2 Pragma master

```
#pragma omp master  newline  
    blok_strukturalny
```

Pragma **master** mówi że dany blok instrukcji ma być wykonany tylko przez wątek główny. Inne wątki pomijają ten fragment kodu.

4.3 Pragma critical

```
#pragma omp critical [nazwa] newline  
    blok_strukturalny
```

Pragma **critical** mówi że dany blok instrukcji ma być wykonany w trybie wyłącznym przez tylko jeden wątek. Inne wątki będą wstrzymane do czasu opuszczeniu bloku przez zajmujący blok wątek.

Opcja *nazwa* pozwala na istnienie wielu sekcji krytycznych. Gdy kilka sekcji ma tę samą nazwę traktowane są jak ta sama sekcja krytyczna.

Uwaga!

Blok strukturalny zawierający sekcję krytyczną nie może zawierać rozgałęzień.

```
#include <omp.h>
main(){
  int x;
  x = 0;
  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
  } /* koniec sekcji równoległej */
}
```

Przykład 4-4-5 Ochrona sekcji krytycznej przez pragme **critical**

4.4 Pragma ordered

```
#pragma omp ordered newline
  blok_strukturalny
```

Pragma **ordered** stosowana jest wewnątrz bloku równoległego **for**. Nakazuje że dany blok instrukcji ma być wykonany w taki sposób w jakim byłby wykonywany na jednym procesorze. Wątki będą czekały z rozpoczęciem następnych kroków iteracji do zakończenia przez inne wątki danego etapu iteracji. Używana wewnątrz bloku **for**.

4.5 Pragma atomic

```
#pragma omp atomic newline  
wyrażenie
```

Pragma **atomic** mówi że dany obszar pamięci ma być aktualizowany w trybie wyłącznym.

- Nie jest dopuszczalne aby różne wątki pisały do tego obszaru naprzemiennie.
- Pragma dotyczy tylko jednej najbliższej instrukcji.

Jest to mini sekcja krytyczna.

Przykład:

```
#pragma omp atomic  
flag++;
```

4.6 Pragma flush

```
#pragma omp flush(lista_zmiennyc) newline  
wyrażenie
```

Pragma **flush** mówi że w danym punkcie wykonania programu powinien być uzyskany spójny obszar pamięci. Pamięci podręczne procesorów powinny być zapisane do pamięci głównej. Zmienne które mają być zapisane do pamięci głównej można wyliczyć w postaci listy zmiennych aby uniknąć zapisu wszystkich zmiennych.

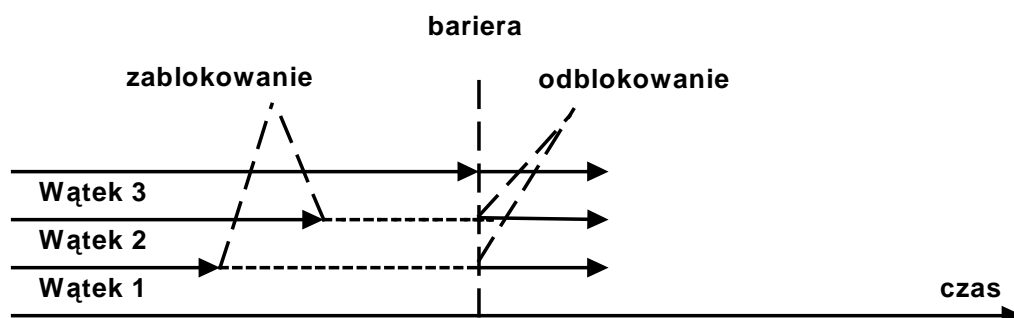
Przykład:

```
#pragma omp flush(work,synch)  
synch[iam] = 1;  
#pragma omp flush(synch)
```

4.7 Pragma barrier

```
#pragma omp barrier newline
```

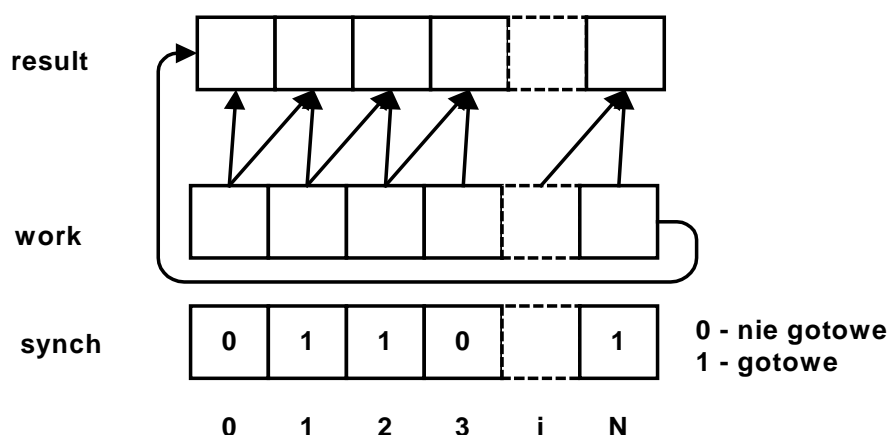
Pragma **barrier** powoduje że wątek będzie czekał aż wszystkie wątki nie osiągną danego punktu synchronizacyjnego.



Rys. 4-4-1 Działanie bariery

Przykład obliczeń komórkowych

- Wynik zapisywany w komórce i tablicy result zależy od wyniku pośredniego w komórce i tablicy work[i] i wyniku w komórce poprzedniej work[i-1].
- Obliczenia prowadzone przez oddzielne wątki dla każdej komórki.
- Zmienna synch[i] sygnalizuje zakończenie obliczeń dla komórki work[i]



```
#include <omp.h>
#define NUMBER_OF_THREADS 256
int synch[NUMBER_OF_THREADS];

float work[NUMBER_OF_THREADS];
float result[NUMBER_OF_THREADS];

float fn1(int i){ return i*2.0; }
float fn2(float a, float b){ return a + b;}

int main() {
    int iam, neighbor;
    #pragma omp parallel private(iam,neighbor)
    shared(work,synch)
    {
        iam = omp_get_thread_num();
        // Gdy synch[i] = 1 wątek i zakończył pracę
        synch[iam] = 0;
        #pragma omp barrier
        //Każdy wątek wykonuje swoją część obliczeń
        work[iam] = fn1(iam);
        // Ogłoś że wątek wykonał swoją pracę
        // flush - wyniki widoczne dla innych
        #pragma omp flush(work,synch)
        synch[iam] = 1;
        // flush - zmienna synch widoczna dla innych
        #pragma omp flush(synch)
        // Wyznaczenie numeru sąsiada
        if(iam == 0) neighbor = omp_get_num_threads() - 1;
        else neighbor = iam -1;
        // Czekamy na wyniki sąsiada
        while (synch[neighbor] == 0) {
            // wartość zmiennej synch odczytywana z pamięci
            #pragma omp flush(synch)
        }
        // wartość zmiennej work odczytywana z pamięci
        #pragma omp flush(work,synch)
        // Odczyt wyniku pracy sąsiada z tablicy work
        result[iam] = fn2(work[neighbor], work[iam]);
    }
    return 0;
}
```

Przykład 4-6 Użycie dyrektyw flush i barrier dla obliczeń komórkowych

5. Funkcje biblioteczne

OpenMP definiuje szereg funkcji bibliotecznych. Służą one do następujących celów.

- Ustawianie środowiska wykonawczego i zmiennych środowiska.
- Zapewnienie synchronizacji wątków – semaforey
- Funkcje pomiaru czasu

5.1 Testowanie i ustawianie środowiska

Funkcja `omp_set_num_threads()`

```
void omp_set_num_threads(int num_threads)
```

`num_threads` Liczba wątków > 0

Funkcja służy do ustawienia liczby wątków które użyte będą w następnej sekcji równoległej.

Funkcja `omp_get_num_threads()`

```
int omp_get_num_threads(void)
```

Funkcja zwraca liczbę wątków w danej sekcji równoległej.

Funkcja `omp_get_max_threads()`

```
int omp_get_max_threads(void)
```

Funkcja odzwierciedla wartość zmiennej środowiska
`OMP_NUM_THREADS`

Funkcja `omp_get_thread_num()`

```
omp_get_thread_num().
```

Poprzez wywołanie tej funkcji wątek może otrzymać swój numer w sekcji równoległej.

Funkcja `omp_num_procs()`

```
int omp_get_num_procs(void)
```

Funkcja zwraca liczbę procesorów zainstalowanych w danym komputerze.

Funkcje `omp_set_dynamic()` i `omp_get_dynamic()`

`void omp_set_dynamic(int dynamic_threads)`

`dynamic_threads` Gdy != 0 dynamiczne ustawianie liczby wątków w sekcji dozwolone 0 gdy nie

Funkcja pozwala na ustawianie parametru decydującego o dynamicznym ustawianiu liczby wątków w sekcji równoległej. Testowanie za pomocą funkcji:

`int omp_get_dynamic(void)`

Jeżeli zezwolimy na dynamiczne przydzielanie, to ilość wątków używanych do wykonywania obszarów równoległych może zmieniać się automatycznie, tak aby jak najlepiej wykorzystać zasoby komputera. Ilość wątków podana jako parametr funkcji jest ilością maksymalną.


```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
int nthreads, tid, procs, maxt, inpar, dynamic, nested;
/* Początek części równoległej */
#pragma omp parallel private(nthreads, tid)
{

    /* Numer wątku bieżącego */
    tid = omp_get_thread_num();

    /* Wykonywane przez wątek główny */
    if (tid == 0) {
        printf("Wątek %d informacje \n", tid);
        procs = omp_get_num_procs();
        nthreads = omp_get_num_threads();
        maxt = omp_get_max_threads();
        inpar = omp_in_parallel();
        dynamic = omp_get_dynamic();
        nested = omp_get_nested();

        printf("Liczba procesorow = %d\n", procs);
        printf("Liczba wątkow = %d\n", nthreads);
        printf("Max wątkow = %d\n", maxt);
        printf("Czy sekc. rowolegla = %d\n", inpar);
        printf("Dynamiczne wątki dozw.? = %d\n", dynamic);
        printf("Zagnieżdżanie rownol. = %d\n", nested);
    }
}
```

Przykład 5-5-1 Testowanie środowiska OpenMP

5.2 Synchronizacji wątków

Inicjacja zamka

```
void omp_init_lock(omp_lock_t *lock)
```

lock Identyfikator zamka

Funkcja inicjalizuje zamek lock. Początkowy stan – otwarty. Inicjalizacja zamka powinna mieć zawsze miejsce przed jego użyciem.

Zamknięcie zamka

```
void omp_set_lock(omp_lock_t *lock)
```

lock Identyfikator zamka

Działanie funkcji zależy od stanu zamka. Gdy jest otwarty następuje jego zajęcie przez bieżący wątek. Gdy jest zamknięty to wątek bieżący jest blokowany do czasu zwolnienia zamka.

Otwarcie zamka

```
void omp_unset_lock(omp_lock_t *lock)
```

lock Identyfikator zamka

Funkcja powoduje odblokowanie zamka. Gdy jakiś wątek jest zablokowany w oczekiwaniu na jego zwolnienie to będzie on wznowiony.

Skasowanie zamka

```
void omp_destroy_lock(omp_lock_t *lock)
```

lock Identyfikator zamka

Funkcja powoduje skasowanie zamka.

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

omp_lock_t my_lock;

int main() {
    omp_init_lock(&my_lock);

    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num( );
        int i, j;
        for (i = 0; i < 5; ++i) {
            omp_set_lock(&my_lock);
            printf_s("Watek %d - wejscie \n", tid);
            printf_s("Watek %d - wyjscie\n", tid);
            omp_unset_lock(&my_lock);
        }
    }
    omp_destroy_lock(&my_lock);
}
```

Przykład 5-5-2 Ilustracja użycia zamka

5.3 Funkcje pomiaru czasu

Pobranie czasu bieżącego

`double omp_get_wtime(void)`

Funkcja zwraca liczbę sekund jaka upłynęła od pewnego punktu startowego. Wykorzystywana jest w postaci dwukrotnego wywołania w celu obliczenia różnicy.

Pobranie rozdzielczości zegara

`double omp_get_wtick(void)`

Funkcja zwraca liczbę sekund pomiędzy dwoma kolejnymi przerwaniem zegarowymi.

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

int main() {
    double start = omp_get_wtime( );
    Sleep(1000);
    double end = omp_get_wtime( );
    double wtick = omp_get_wtick( );
    printf("start = %.16g koniec = %.16g roznica = %.16g\n", start, end, end - start);
    printf("wtick = %.16g 1/wtick = %.16g\n", wtick, 1.0 / wtick);
}
```

Przykład 5-5-3 Funkcje obliczania czasu

6. Zmienne środowiska

OpenMP używa zmiennych środowiska do kontrolowania opcji pracy systemu.

OMP_NUM_THREADS – Zmienna określa maksymalną liczbę wątków jaka może być użyta w aplikacji.

Przykład:

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC – Zmienna specyfikuje czy dozwolone jest dynamiczne dostosowanie liczby wątków w sekcjach równoległych. Dopuszczalne wartości TRUE lub FALSE.

Przykład:

```
setenv OMP_DYNAMIC TRUE
```

OMP_NESTED – Opcja specyfikuje czy dozwolone są zagnieżdżone bloki równoległe. Dopuszczalne wartości TRUE lub FALSE.

Przykład:

```
setenv OMP_NESTED TRUE
```

W systemie Windows informację o liczbie procesorów zawiera zmienna środowiska **NUMBER_OF_PROCESSORS**

7. Przykład

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

int main(int argc, char *argv[]) {
    int i,tid,chunk,ilw;
    int total = 0;
    int N = atoi(argv[1]);
    int nthreads, procs, maxt;
    double t1,t2; chunk = 100;
    procs = omp_get_num_procs();
    maxt = omp_get_max_threads();
    if(argc <3) ilw = procs; else ilw = atoi(argv[2]);
    printf("Liczba procesorow = %d\n", procs);
    printf("Max watkow = %d\n", maxt);
    t1 = omp_get_wtime();
    omp_set_num_threads(ilw);

    #pragma omp parallel shared(N,chunk) private(i,tid)
    {
        nthreads = omp_get_num_threads();
        tid = omp_get_thread_num();
        if(tid == 0) printf("Watkow = %d\n", nthreads);

        #pragma omp for schedule(dynamic,chunk)
        reduction(+:total)
        for (i = 2; i < N; i++) {
            if ( is_prime(i) ) {
                total++;
            }
        }
    }
    printf("L. pierw. pom 2 i %d jest: %d\n", N, total)
    t2 = omp_get_wtime();
    printf("Czas obliczen= %g\n",t2-t1);
    return 0;
}

int is_prime(int v) {
    int i;
    for (i = 2; i*i <= v; i++) {
        if (v % i == 0) {
            return 0; /* v nie jest l. pierwsza */
        }
    }
    return 1; /* v jest l. pierwsza */
}
```

Przykład 7-1 Znajdowanie liczb pierwszych w przedziale

```
Liczba procesorow = 4
Max watkow = 4
Liczba watkow = 4
Liczba pierwszych pomiedzy 2 i 10000000 jest: 664579
Czas obliczen= 2.00263
```

Liczba wątków	Czas w sek
4	2.01
3	2.66
2	3.96
1	7.98

Tab. 7-1 Czasy obliczeń przykładu dla zakresu od 2 do 10 000 000 na procesorze 4 rdzeniowym

8. Źródła

- [1] Blaise Barney, OpenMP, <https://computing.llnl.gov/tutorials/openMP/>
- [2] Paul Graham, OpenMP a Parallel Programming Model for Shared memory Architectures.
- [3] OpenMP Application Program Interface, Version 3.0 May 2008, <http://www.openmp.org>