Programowanie współbieżne
Wstęp do OpenMP

Rafał Skinderowicz

Historia

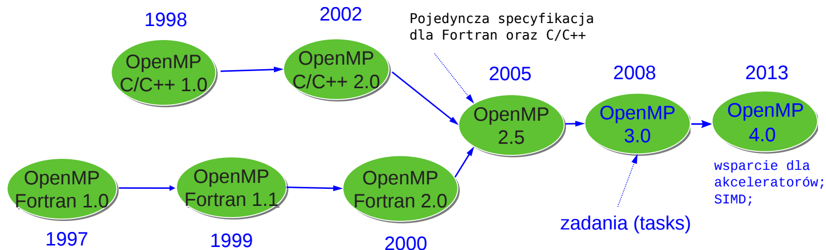
Czym jest OpenMP?

- OpenMP = Open Multi-Processing – interfejs programowania aplikacji (API) dla pisania aplikacji równoległych na komputery wieloprocesorowe ze współdzieloną pamięcią
- Składa się z:
  - dyrektyw kompilatora (np. #pragma w języku C)
  - biblioteki
  - zmiennych środowiskowych

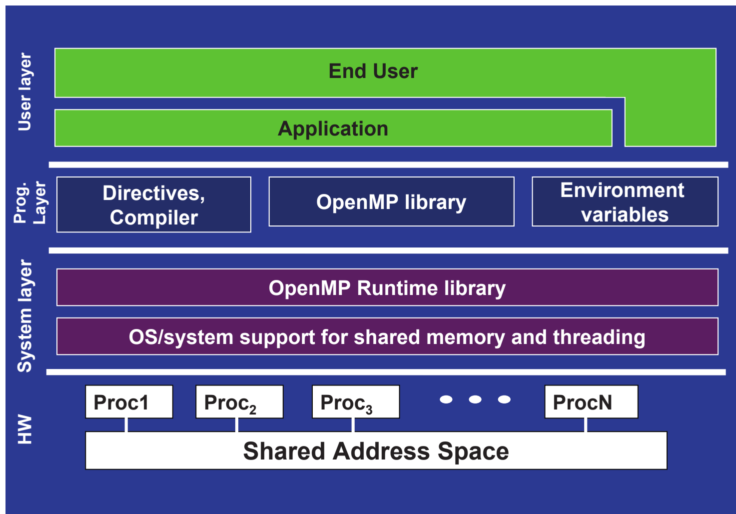
Czym jest OpenMP?

- Przenośny:
  - Dla języków Fortran oraz C/C++
  - Wspierany na wielu platformach włączając Windows, Unix/Linux
  - Kompilatory: GCC, Intel Fortran i C/C++, IBM XL C/C++, MS C++
  - Od wersji 4.0 również Nvidia, AMD
- Ustandaryzowany – specyfikacją zarządza OpenMP Architecture Review Board (http://www.openmp.org)
- W skład komitetu standaryzacyjnego wchodzą m.in. AMD, Convey, Cray, Fujitsu, HP, IBM, Intel, NEC, NVIDIA, Oracle, RedHat (GNU), ST Mircoelectronics, clang/llvm

OpenMP – historia



OpenMP – architektura



Rysunek 1: Źródło: Mattson T., Meadows L., A "Hands-on" Introduction to OpenMP

Podstawy

OpenMP – podstawy

Programista w sposób *jawny* wskazuje, które fragmenty programu powinny być wykonywane równolegle. Wskazanie odbywa się za pomocą **pragm**

```
#pragma omp nazwa_konstrukcji [klauzula1] [klauzula2]
```

Przykład:

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

Dodatkowo OpenMP udostępnia poprzez nagłówek `omp.h` zestaw funkcji bibliotecznych oraz makr

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

- Większość dyrektyw OpenMP stosowana jest do bloków:
    - blok = jedno lub więcej wyrażeń
    - pojedynczy punkt wejścia na początku bloku

Kompilacja

GNU GCC wymaga przełącznika -fopenmp

```
1 gcc -fopenmp program.c -o program
```

Kompilatory Intela

```
1 icc -openmp program.c -o program
```

W VisualStudio należy włączyć obsługę OpenMP: Configuration Properties → C/C++ → Language i ustawić opcję OpenMP Support na Yes

- dla kompilacji w linii poleceń trzeba dodać przełącznik /openmp

OpenMP – przykład

```
1 void main()
2 {
3     int id = 0;
4     printf("hello(%d)", id);
5     printf("world(%d)\n", id
       );
6 }
```

```
1  #include <omp.h>
2  void main()
3  {
4      #pragma omp parallel
5      {
6          int id = omp_get_thread_num();
7          printf("hello(%d)", id);
8          printf("world(%d)\n", id);
9      }
10 }
```

Przykładowy wynik dla 4 procesorów:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

- Blok parallel wykonywany jest równolegle na dostępnych procesorach

OpenMP – przykład 2

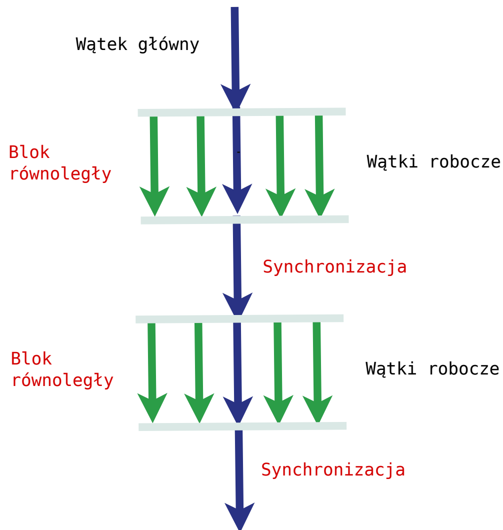
```c
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp task
        { printf("Hello "); }
        #pragma omp task
        { printf("World"); }
    }
    printf("\nThe end");
    return 0;
}
// Wynik: Hello Hello WorldHello WorldWorldHello World
```

- W bloku `parallel` tworzone są dwa zadania – dla 4 wątków będzie utworzonych 8 zadań

OpenMP – zarządzanie wątkami

- OpenMP wymaga jawnego definiowania, które fragmenty kodu mają wykonać się równolegle – pełna kontrola nad zrównolegleniem
- OpenMP stosuje model Fork-Join do równoległego wykonywania programu
  - po napotkaniu bloku równoległego w programie wątek główny odpowiedzialny jest za tworzenie grupy wątków roboczych (Fork)
  - każdy wątek wykonuje instrukcje w bloku
  - po zakończeniu wykonania bloku wątki robocze są synchronizowane i usuwane, pozostaje jedynie wątek główny (Join)
- Wątki tworzone są niejawnie przez OpenMP – domyślnie tyle ile dostępnych jest fizycznych procesorów/rdzeni

OpenMP – zarządzanie wątkami



OpenMP – zarządzanie wątkami

- Wątek główny ma zawsze ID $= 0$
- Wątki komunikują się przez współdzieloną pamięć, tj. współdzielone zmienne
  - Brak ostrożności w dostępie do współdzielonych zmiennych może prowadzić do wyścigów – zmienna jest jednocześnie modyfikowana przez kilka wątków
  - Dostępne są dyrektywy umożliwiające synchronizację dostępu wątków do sekcji krytycznej
  - Synchronizacja jest kosztowna, dlatego należy ją minimalizować
- Za pomocą instrukcji warunkowych można przekazać różnym wątkom różne fragmenty programu do wykonania

Przykład – współdzielenie pamięci

```
#include <omp.h>
#include <iostream>
int main(int argc, char *argv[]) {
    int licz = 0; // Ta zmienna jest współdzielona przez wątki
    #pragma omp parallel shared(licz)
    {
        #pragma omp single
        {
        std::cout << "Liczba wątków: " << omp_get_num_threads()
                  << std::endl;
        }
        // To będzie wykonane przez wszystkie wątki
        for (int i = 0; i < 100; ++i) {
            licz++;
        }
    }
    std::cout << "licz: " << licz << std::endl; // Chcemy 100*l. wątków
    return 0;
}
```

Przykład – współdzielenie pamięci

Przykładowy wydruk dla 4 wątków:

```
./example
Liczba wątków: 4
licz: 198
```

Przykład – współdzielenie pamięci

Przykładowy wydruk dla 4 wątków:

```
./example
Liczba wątków: 4
licz: 198
```

Poprawiony program:

```
// To będzie wykonane przez wszystkie wątki
for (int i = 0; i < 100; ++i) {
    #pragma omp critical
    licz++;
}
```

Przykład – współdzielenie pamięci

Przykładowy wydruk dla 4 wątków:

```
./example
Liczba wątków: 4
licz: 198
```

Poprawiony program:

```
// To będzie wykonane przez wszystkie wątki
for (int i = 0; i < 100; ++i) {
    #pragma omp critical
    licz++;
}
```

I wynik – zgodnie z oczekiwaniami:

```
./example
Liczba wątków: 4
licz: 400
```

Budowa dyrektywy OpenMP

#pragma omp	nazwa-dyrektywy	[klauzula, . . .]	newline
Obowiązkowa dla wszystkich dyrektyw	Nazwa dyrektywy – obowiązkowa, tylko jedna na jedną dyrektywę.	Opcjonalna lista klauzul, mogą występować w dowolnej kolejności	Dyrektywa kończy się wraz z końcem wiersza – w przypadku wielowierszowych dykrektyw należy używać znaku \

- Wielkość liter ma znaczenie
- Każda dyrektywa stosuje się tylko do jednego wyrażenia, które po niej następuje (w kolejnym wierszu)
- W przypadku większej liczby wyrażeń należy je ująć w blok, tj. { ... }, wtedy dyrektywa będzie miała zastosowanie dla całego bloku
- Wszystkie dyrektywy OpenMP mają zastosowanie, jedynie gdy zdefniowane jest makro _OPENMP

Organizacja obliczeń

Konstrukcja równoległa

- Konstrukcja równoległa wskazuje fragmenty programu do wykonania *rówoległego*

```
1 #pragma omp parallel [klauzula [[,] klauzula]...]
2 {
3     instrukcje wykonywane przez grupę wątków
4 } // niejawny punkt synchronizacji wątków (bariera synch.)
```

- Po *zakończeniu* wykonywania regionu równoległego tylko wątek *główny* kontynuuje działanie

Konstrukcja równoległa

- Dyrektywa **parallel** obsługuje następujące opcjonalne klauzule:
  - if (wyrażenie całkowitoliczbowe)
  - private (lista zmiennych)
  - shared (lista zmiennych)
  - default (`shared` albo `none`)
  - firstprivate (lista zmiennych)
  - reduction (operator: lista zmiennych)
  - copyin (lista zmiennych)
  - num_threads (wyrażenie całkowitoliczbowe)

```
#pragma omp parallel if(n > 1000)
{
    ...
}
```

Dyrektywa parallel – klauzule

- Klauzula `if (wyrażenie całkowitoliczbowe)` służy do określenia, czy blok kodu powinien być wykonywany *równolegle* – przez zespół wątków, czy *sekwencyjnie* – tylko przez wątek główny
- Klauzula `num_threads (wyrażenie całkowitoliczbowe)` określa liczbę wątków roboczych do utworzenia
- Klauzule `private, shared, firstprivate, lastprivate`, `default` są stosowane do określenia sposobu dostępu wątków do zmiennych współdzielonych – szczegóły w dalszej części

Konstrukcje organizacji obliczeń

- Konstrukcje te wskazują wątki odpowiedzialne za wykonanie oznaczonych fragmentów regionu równoległego

```
#pragma omp for
{
    ...
}
```

```
#pragma omp sections
{
    ...
}
```

```
#pragma omp single
{
    ...
}
```

- Obliczenia są dzielone między wszystkimi wątkami (głównym + roboczymi)
- Wymienione dyrektywy muszą znaleźć się wewnątrz konstrukcji równoległej (dyrektywa parallel)
- Niejawna bariera synchronizacyjna na wyjściu (można usunąć za pomocą klauzuli `nowait`)
- Wymienione dyrektywy nie uruchamiają nowych wątków – wykorzystują już utworzone przy definicji bloku równoległego

Przykład 1

Zadanie: zrównoleglić poniższy fragment programu:

```
for (i = 0; i < N; i++) {
    a[i] = a[i] + b[i];
}
```

Przykład 1

Dzięki konstrukcji **omp for** można łatwo zrównoleglić wykonanie pętli, jednak tylko **for**

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++) {
        a[i] = a[i] + b[i];
    }
}
```

Wersja skrócona

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    a[i] = a[i] + b[i];
}
```

Obliczenia dzielone są równo pomiędzy wątkami – można to zmienić za pomocą dodatkowych *klauzul*

Przykład 1 – uwagi

Wersja bez dyrektywy **omp for** – trzeba jawnie dokonać rozdziału obliczeń na podstawie identyfikatora wątku

Przykład:

```
1  #pragma omp parallel
2  {
3      const int id = omp_get_thread_num();
4      const int p = omp_get_num_threads();
5      // każdy wątek wykonuje obliczenia dla co p-tego
6      // el., począwszy od id
7      for(int i = id; i < n; i += p) {
8          a[i] = a[i] + b[i];
9      }
10 }
```

Przykład 1 – uwagi

Wersja bez dyrektywy **omp for** – trzeba jawnie dokonać rozdziału obliczeń na podstawie identyfikatora wątku

Przykład:

```
#pragma omp parallel
{
    const int id = omp_get_thread_num();
    const int p = omp_get_num_threads();
    // każdy wątek wykonuje obliczenia dla co p-tego
    // el., począwszy od id
    for(int i = id; i < n; i += p) {
        a[i] = a[i] + b[i];
    }
}
```

Zazwyczaj lepiej jest, gdy wątek wykonuje obliczenia dla elementów blisko siebie w pamięci, stąd powyższy podział pracy należałoby zmodyfikować

Pętla for – uwagi

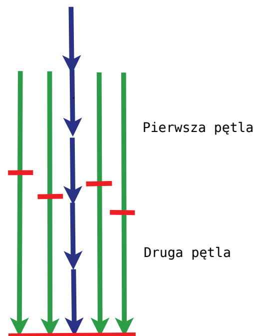
- Automatyczne zrównoleglanie pętli `for` działa tylko, gdy pętla ma określoną strukturę
- Musi być możliwe określenie *przed rozpoczęciem* wykonywania ile iteracji będzie wykonanych

```
for (i = całk-1; i op całk-2; inkr)
```

- `całk-1` i `całk-2` to wyrażenia typu całkowitego,
- op jest jednym z operatorów: $<, <=, >, >=$
- `inkr` może mieć jedną z postaci: `i++`, `i--`, `i += krok`, `i -= krok`

Dyrektywa for – przykład 2

```
#pragma omp parallel default (none) \
        shared (n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++) {
        b[i] = (a[i] + a[i+1]) / 2;
    } // <- tu nie ma bariery
    for (i=0; i<n; i++) {
        d[i] = 1.0/c[i];
    } // <- tu jest niejawna bariera
} // <- tu również
```



Klauzula nowait usuwa niejawną barierę synchronizacyjną na końcu pętli for

Zrównoleglanie pętli

- W wielu programach większość obliczeń wykonywana jest w niewielkim fragmencie kodu – reguła 90-10, czyli za 90% procent obliczeń odpowiada 10% kodu
- Często te 10% ma strukturę pętli, a więc na nich warto skupić się przy zrównoleglaniu programu
- W celu efektywnego zrównoleglenia potrzeba, aby kolejne iteracje pętli były od siebie niezależne

Zrównoleglanie pętli

Wersja z zależnością między iteracjami

```
int i, j, A[N];
j = 5;
for(i = 0; i < N; i++) {
    j += 2; // wart. j zależy
        // od poprz. iteracji
    A[i] = compute(j);
}
```

Wersja z usuniętą zależnością pomiędzy iteracjami

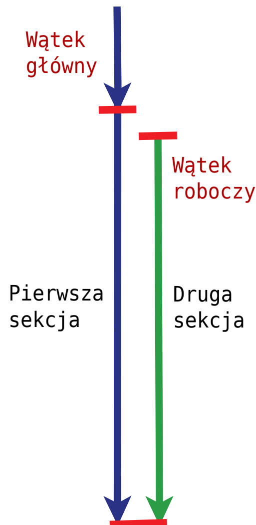
```
int i, j, A[N];
#pragma omp parallel for
for (i = 0; i < N; i++) {
    int j = 5 + 2*i;
    A[i] = compute(j);
}
```

Podział pracy za pomocą sekcji

- Nieiteracyjny sposób podziału obliczeń na bloki, dla których obliczenia wykonywane są w sposób równoległy
- Każda sekcja wykonywana jest przez dokładnie jeden wątek
- Sekcje oznaczone są dyrektywami `section` i muszą być zagnieżdżone wewnątrz bloku oznaczonego dyrektywną `sections`
- Może zdarzyć się, że pojedynczy wątek wykona więcej, niż jedną sekcję po zakończeniu poprzedniej
- Przydział sekcji do poszczególnych wątków zależy od danej implementacji OpenMP
- Jeżeli jest więcej wątków, niż sekcji równoległych, to część z nich pozostanie bezczynna

Podział pracy za pomocą sekcji – przykład

```
1  #pragma omp parallel default (none)
2          shared (n,a,b,c,d) private(i)
3  {
4    #pragma omp sections nowait
5    {
6      #pragma omp section
7      for (i=0; i<n-1; i++)
8        b[i] = (a[i] + a[i+1]) / 2;
9      #pragma omp section
10     for (i=0; i<n; i++)
11       d[i] = 1.0/c[i];
12   } // <- synchronizacja wyłączona
13 } // <- niejawna synchronizacja
```



Podział pracy za pomocą zadań – przykład

- Od wersji 3.0 OpenMP udostępnia możliwość podziału pracy za pomocą *zadań* – `task`
  - Konstrukcja ta przypomina sekcje (`section`), jednak wykonanie sekcji ograniczone jest do bloku `sections`
  - Wykonanie zadania może być odłożone w czasie, nie są również na stałe przypisane do jednego wątku

```
1  #include <cstdlib>
2  #include <cstdio>
3  using namespace std;
4  int main(int argc, char *argv[])
5  {
6      #pragma omp parallel
7      {
8          #pragma omp task
9          { printf("Hello "); }
10         #pragma omp task
11         { printf("World"); }
12     }
13     printf("\nThe end");
14     return 0;
15 }
```

Sekcje a zadania

```
...............[    sections     ]
Thread 0: -------< section 1 >---->*------
Thread 1: -------< section 2      >*------
Thread 2: ------------------------>*------
...                                 *
Thread N-1: ---------------------->*------
```

Sekcje a zadania

```
1 ...............+--+-->[ task queue ]--+
2                |  |                   |
3                |  |       +-----------+
4                |  |       |
5 Thread 0: --< single >-|  v  |-----
6 Thread 1: -------->|< foo() >|-----
7 Thread 2: -------->|< bar() >|-----
```

```
1 ...............+--+-->[ task queue ]--+
2                |  |                   |
3                |  |      +------------+
4                |  |      |
5 Thread 0: --< single >-| v             |---
6 Thread 1: --------->|< foo() >< bar() >|---
7 Thread 2: --------------------->|      |---
```

Każdy wątek ma swoją kolejkę zadań. Po jej opróżnieniu sięga po zadania innego wątku (ang. work-stealing).

Sekcje a zadania – przykład

- Dwa funkcjonalnie równoważne programy

```
// sections
...
#pragma omp sections
{
   #pragma omp section
   foo();
   #pragma omp section
   bar();
}
...
```

```
// tasks
...
#pragma omp single nowait
{
   #pragma omp task
   foo();
   #pragma omp task
   bar();
}
#pragma omp taskwait
...
```

Sekcje a zadania – przykład

Sekcje i zadania mogą być stosowane do wygodnego zrównoleglania algorytmów rekurencyjnych.

```
1  int fibOpenMP( int n ) {
2      int i, j;
3      if( n < 10 ) {
4          return fibSerial(n); // policz sekwencyjnie
5      } else {
6          // utwórz nowe zadanie dla fib(n-1)
7          #pragma omp task shared( i ), untied
8          i = fib( n - 1 );
9          // utwórz kolejne dla fib(n-2)
10         #pragma omp task shared( j ), untied
11         j = fib( n - 2 );
12         // poczekaj na zakończenie zadań
13         #pragma omp taskwait
14         return i + j;
15     }
16 }
```

Sekcje i zadania nie nadają się do zadań, które wykonują czasochłonne, blokujące operacje, np. związane z I/O.

Klauzula single

- Stosowana do wymuszenia wykonania wskazanego bloku tylko przez jeden wątek (niekoniecznie główny)
- Przydatna w przypadku konieczności wykonania operacji I/O
- Pozostałe wątki pomijają instrukcje w bloku i oczekują na jego końcu – niejawna bariera synchronizacyjna, którą można usunąć przez dodanie klauzuli `nowait`

```
1  #pragma omp parallel
2  {
3      do_many_things();
4      #pragma omp single
5      {
6          get_new_data();
7      }
8      #pragma omp barrier
9      do_many_other_things();
10 }
```

Klauzula master

- Stosowana do wymuszenia wykonania danego bloku tylko przez wątek główny
- Pozostałe wątki pomijają blok oznaczony tą klauzulą bez konieczności synchronizacji
- Nie wymaga pobierania, ani sprawdzania identyfikatora wątku

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        get_new_data();
    }
    #pragma omp barrier
    do_many_other_things();
}
```

Redukcje

Redukcja

Jak poradzić sobie z następującym przypadkiem?

```
double sr = 0, A[N];
for(int i = 0; i < N; i++)
{
    sr += A[i];
}
if (N > 0) {
    sr = sr / N;
}
```

- Nie da się łatwo usunąć zależności między iteracjami dla zmiennej `sr`
- Przedstawiona operacja sumowania należy do operacji **redukcji**
- Operacje redukcji występują bardzo często, dlatego opracowano dla nich odrębną klauzulę

Redukcja

- Klauzula dla operacji redukcji:
  `reduction (operator : lista_zmiennych)`
- Wewnątrz bloku równoległego, w którym wykonywana jest operacja redukcji:
  - tworzona jest lokalna kopia każdej zmiennej podanej w `lista_zmiennych` i inicjowana wartością początkową (0 dla "+", 1 dla "*")
  - Kompilator zastępuje zmienną globalną w wyrażeniach redukcji jej odpowiednikami lokalnymi
  - Przy wyjściu z bloku równoległego lokalne kopie zmiennej są łączone w jedną wartość zapisywaną w zmiennej globalnej

Redukcja

```c
double sr = 0, A[N];
#pragma omp parallel for reduction (+:sr)
for(int i = 0; i < N; i++)
{
    sr += A[i];
}
if (N > 0) {
    sr = sr / N;
}
```

Redukcja – operatory

Operator	Wartość początkowa
$+$	0
-	0
$*$	0
&	~ 0
|	0
^	0
&&	1
||	0

Wyrażenie redukcyjne wewnątrz pętli powinno mieć jedną z postaci:

```
x = x op expr (expr - to wyr. całkowitoliczbowe bez x)
x = expr op x (z wyjątkiem -)
x binop = expr
x++
++x
x--
--x
```

Synchronizacja

Synchronizacja

Synchronizacja jest stosowana do:

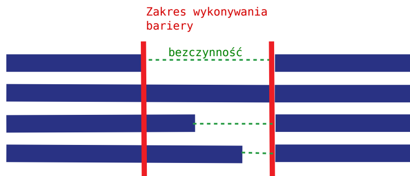
- wymuszenia określonej kolejności wykonania obliczeń wewnątrz bloku równoległego przez poszczególne wątki
- zabezpieczenia dostępu do sekcji krytycznych, w których wykonywane są operacje na współdzielonych zmiennych

Synchronizacja

Dostępne w OpenMP mechanizmy synchronizacyjne można podzielić na:

- Bariery niejawne – wstawiane automatycznie np. na zakończenie bloku równoległego dla pętli for
- Bariery jawne – wysokopoziomowe definiowane za pomocą klauzul:
  - critical
  - atomic
  - barrier
  - ordered
- Bariery jawne – niskopoziomowe:
  - flush
  - zamki (ang. locks)

Bariery niejawne




```
1 #pragma omp  for
2 for (i=0; i<N; i++) {
3     a[i] = c[i] + b[i];
4 } // niejawna bariera
5 #pragma omp for
6 for (i=0; i<N; i++) {
7     d[i] = a[i] + b[i];
8 } // niejawna bariera
```

Barierę niejawną można wyłączyć klauzulą `nowait` pod warunkiem, że istnieje gwarancja identycznego podziału pracy między wątkami w obu pętlach

Klauzula critical

Zapewnia wzajemne wykluczanie wykonanie **sekcji krytycznej**

```
1  float res;
2  #pragma omp parallel
3  {
4      float b;
5      int id = omp_get_thread_num();
6      int nthreads = omp_get_num_threads();
7      for(int i = id; i < N; i+=nthreads) {
8          b = compute(i);
9          #pragma omp critical
10         consume (b, res); // tylko jeden wątek wykonuje compute,
11                           // pozostałe czekają na swoją kolej
12     }
13 }
```

Klauzula atomic

Zapewnia **atomowe** wykonanie następującej po niej instrukcji – tylko wybrane

```
1 double X = 0; // zmienna współdzielona
2 #pragma omp parallel
3 {
4     double tmp, a;
5     a = compute();
6     tmp = compute_sth_else(a);
7     #pragma omp atomic
8     X += tmp;
9 }
```

Synchronizacja – klauzula barrier

Zapewnia jawny, wymuszony i wspólny dla wszystkich wątków punkt synchronizacji

```
1  #pragma omp parallel shared (A, B, C) private(id)
2  {
3      id=omp_get_thread_num();
4      A[id] = big_calc1(id);
5      #pragma omp barrier
6      #pragma omp for
7      for(i=0;i<N;i++) {
8          C[i]=big_calc3(i,A);
9      } // <- niejawna bariera dla klauzuli for
10     #pragma omp for nowait
11     for(i=0;i<N;i++){
12         B[i]=big_calc2(C, i);
13     } // <- bariera usunięta przez nowait
14     A[id] = big_calc4(id);
15 } // <- niejawna bariera dla bloku równoległego
```

Klauzula ordered

- Wymusza wykonanie oznaczonej instrukcji/bloku wewnątrz pętli dokładnie w takiej kolejności w jakiej byłyby wykonane w programie sekwencyjnym
- W danym momencie tylko jeden wątek może wykonywać oznaczoną instrukcję, pozostałe czekają
- Pętla zawierająca blok z klauzulą `ordered` musi również mieć tę klauzulę

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
for (i=0;i<N;i++){
    tmp = compute(i);
    #pragma ordered
    res += consume(tmp);
}
```

Współdzielenie pamięci

Współdzielenie danych

- OpenMP to interfejs do tworzenia programów równoległych dla modelu ze współdzieloną pamięcią – większość danych jest współdzielona przez wszystkie wątki domyślnie
- OpenMP stosuje złagodzony model spójności pamięci (ang. relaxed consistency)
- Zmienne globalne są współdzielone przez wątki
  - zmienne związane z obsługą plików
  - dynamicznie przydzielona pamięć – new, malloc
  - inne, np. stan generatora liczb losowych dla funkcji rand() (!)
- Nie są współdzielone:
  - zmienne automatyczne tworzone na stosie wewnątrz bloków równoległych
  - zmienne tworzone na stosie przez wywoływane w blokach równoległych funkcjach

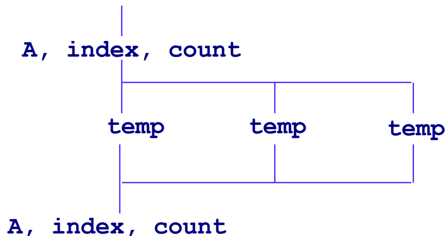
Przykład

```c
double A[10];
int main() {
  int index[10];
  #pragma omp parallel
  work(index);
  printf("%d ", index[0]);
}
```

```c
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

Zmienne index oraz count są współdzielone przez wątki

Każdy wątek ma własną kopię tablicy temp



Klauzule

- Można jawnie określić sposób dostępu przez wątki do zmiennych za pomocą klauzul:
  - `shared(zmienna1, zmienna2, ...)` – wyszczególnione zmienne są współdzielone między wątkami
  - `private(zmienna1, zmienna2, ...)` – podane zmienne są prywatne
- Domyślnie wszystkie zmienne są współdzielone (ang. shared)
- Można nadpisać to za pomocą klauzuli `default`
  - `default(shared)` – wszystkie zmienne są domyślnie współdzielone
  - `default(none)` – żadna zmienna nie jest domyślnie współdzielona – dobra praktyka programistyczna, ponieważ musimy jawnie określić listę zmiennych współdzielonych oraz prywatnych
    - wiemy, które zmienne są prywatne, a które mogą być modyfikowane przez wszystkie wątki – łatwiej zauważyć potencjalne źródła wyścigów

Klauzula private

- `private(x)` – każdy wątek ma własną prywatną kopię zmiennej `x`
- wartość początkowa zmiennej prywatnej jest domyślnie nieokreślona

```
1  void wrong() {
2      int tmp = 0;
3      #pragma omp parallel for private(tmp)
4      for (int j = 0; j < 1000; ++j) {
5          tmp += j; // <- zmienna tmp nie została zainicjowana
6      } // <- koniec bloku równoległego
7
8      printf("%d", tmp); // <- wart. tmp jest w
9                         // tym miejscu nieokreślona
10 }
```

Klauzula firstprivate

Dzięki klauzuli `firstprivate(x)` możemy określić, że każda prywatna zmienna `x` będzie miała wartość początkową jak zmienna (jej pierwowzór) w wątku głównym

```
void wrong() {
    int tmp = 3;
    #pragma omp parallel for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j) {
        tmp += j; // <- zmienna tmp ma początkową wart. 3
    } // <- koniec bloku równoległego
    printf("%d", tmp); // <- wart. tmp jest w
                       // tym miejscu nieokreślona
}
```

Klauzula lastprivate

- Dzięki klauzuli `lastprivate(x)` możemy określić, że zmienna `x` po zakończeniu pętli będzie miała wartość taką jak wartość zmiennej prywatnej w ostatniej iteracji pętli
- Działa, oczywiście, tylko dla dyrektywy `omp for`

```
int x = 0;
const int N = omp_get_max_threads();
#pragma omp parallel for firstprivate(x) lastprivate(x)
for (int i = 0; i < N; ++i) {
    int id = omp_get_thread_num();
    x += id;
}
printf("Final x: %d", x); // <- x ma wartość N-1
```

Klauzula threadprivate

- Umożliwia definiowanie zmiennych globalnych, dla których każdy wątek będzie miał własną kopię
- Różni się od klauzul `private`, `firstprivate`, `lastprivate`:
  - klauzula `private` powoduje „maskowanie” zmiennej globalnej zmienną prywatną
  - klauzula `threadprivate` powoduje, że nie ma jednej globalnej zmiennej, ale każdy wątek ma własną, tylko jedną i dostępną w każdym bloku równoległym

```
1 int counter = 0;
2 #pragma omp threadprivate(counter) // każdy wątek ma własny
3                                    // licznik (counter)
4 int increment_counter()
5 {
6     counter++;
7     return (counter);
8 }
```

Klauzula copyprivate

Dyrektywa `copyprivate` służy do rozgłaszania (ang. broadcast) wartości zmiennych lokalnych wątku, który wykonuje sekcję `single` do pozostałych wątków

```
#include <omp.h>
void wczytaj_parametry (int, int);
void oblicz(int, int);
void main()
{
    int rozmiar, dokladnosc;
    #pragma omp parallel private (rozmiar, dokladnosc)
    {
        #pragma omp single copyprivate (rozmiar, dokladnosc)
            wczytaj_parametry(rozmiar, dokladnosc);

        // teraz zmienne rozmiar i dokladnosc pozostałych
        // wątków mają takie same (pobrane) wartości
        oblicz(rozmiar, dokladnosc);
    }
}
```

Podsumowanie

```
int A = 1, B = 1, C = 1, D = 2;
#pragma omp parallel default(none) private(B) firstprivate(C)
{ blok równoległy }
```

Wewnątrz bloku równoległego:

- Zmienna `A` jest dzielona przez wszystkie wątki, wartość 1
- Zmienne `B` i `C` są prywatne – każdy wątek ma własną kopię `B` i `C`
  - Wartość początkowa `B` jest nieokreślona
  - Wartość początkowa `C` wynosi 1
- Zmienna `D` jest niedostępna wewnątrz bloku równoległego

Po zakończeniu wykonania bloku równoległego:

- Wartości `B` i `C` są nieokreślone

Dyrektywa flush

- W celu przekazania aktualnej wartości zmiennej współdzielonej między wątkami można wykorzystać dyrektywę `flush`
- Dyrektywa `flush` określa punkt, w którym istnieje gwarancja, że wszystkie wątki będą miały aktualny i spójny „widok” pamięci współdzielonej
- Scenariusz:
  - Wątek 1. zapisuje wartość zmiennej
  - Wątek 1. wykonuje dyrektywę flush dla tej zmiennej
  - Wątek 2. wykonuje dyrektywę flush dla tej zmiennej
  - Wątek 2. odczytuje aktualną wartość zmiennej

Dyrektywa flush – przykład

```
int data, flag = 0;
#pragma omp parallel sections num_threads(2)
{
  #pragma omp section
  {
     read(&data);
     #pragma omp flush(data)
     flag = 1;
     #pragma omp flush(flag)
     // ...
  }
  #pragma omp section
  {
     while (!flag) {
        #pragma omp flush(flag)
     }
     #pragma omp flush(data)
     process(&data);
  }
}
```

Pomiar czasu wykonania programu

- OpenMP udostępnia funkcję `double omp_get_wtime()`, która zwraca liczbę sekund od pewnego ustalonego momentu w przeszłości
- Rozdzielczość pomiaru można pobrać za pomocą funkcji `double omp_get_wtick()`, jest to najkrótszy mierzalny okres czasu

```
int main() {
    double start = omp_get_wtime( );
    oblicz_cos();
    double end = omp_get_wtime( );
    printf("start = %.16g end = %.16g diff = %.16g",
           start, end, end - start);
}
```