

OpenMP – część praktyczna

Zbigniew Koza

Wydział Fizyki i Astronomii
Uniwersytet Wrocławski



Wrocław, 17 marca 2011

Spis treści

1 Kompilacja

Spis treści

- 1 Kompilacja
- 2 Podział pracy

Spis treści

- 1 Kompilacja
- 2 Podział pracy
- 3 Synchronizacja

Część 1

- 1 Kompilacja
- 2 Podział pracy
- 3 Synchronizacja

Kompilacja

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

```
> g++ -fopenmp [...]
```

```
> icc -fopenmp [...]
```

- `-fopenmp` jest to opcja kompilatora i linkera

```
#ifdef _OPENMP  
...  
#endif
```

pragma omp parallel

- Kod zrównoleglony musi zawierać się w ramach bloku `#pragma omp parallel`
- Przykład:

```
#pragma omp parallel for
for (i = 0; i < rows; i++)
{
    for (j = 0; j < columns; j++)
        v[i] += A[i][j] * u[j];
}
```

pragma omp parallel

składnia ogólna

```
#pragma omp parallel
{
    #pragma omp nazwa_dyrektywy opcje_i_parametry
    {

        ...
    }
}
```


pragma omp parallel

składnia ogólna

```
#pragma omp parallel
{
    #pragma omp nazwa_dyrektywy opcje_i_parametry
    {

        ...
    }
}
```

składnia skrócona

```
#pragma omp parallel nazwa_dyrektywy opcje_i_parametry
```

Opcje private i shared

- Zmienne współdzielone przez wątki deklarujemy w opcji **shared**
 - Zmienne sterujące pętlą
- Zmienne lokalne w wątku deklarujemy w opcji **private**
 - Zmienne określające liczbę iteracji pętli
 - Zmienne i tablice „tylko do odczytu” w pętli
 - Tablice modyfikowane w pętli, jeżeli różne wątki modyfikują ich różne elementy

```
#pragma omp parallel for shared(A, v, u) private(i, j)
for (i = 0; i < rows; i++)
{
    for (j = 0; j < columns; j++)
        v[i] += A[i][j] * u[j];
}
```

Opcja default(none)

- Kompilator domyślnie w rozsądny sposób dzieli zmienne na prywatne i współdzielone
- Opcja **default (none)** wyłącza ten mechanizm
- Zaleca się używanie tej opcji

```
#pragma omp parallel for\  
    default(none) shared(A, v, u) private(i, j)  
for (i = 0; i < rows; i++)  
{  
    for (j = 0; j < columns; j++)  
        v[i] += A[i][j] * u[j];  
}
```

Opcja schedule

- Dyrektywa **schedule** służy do definiowania sposobu rozdziału pracy na wątki w pętli for.
- Składnia:

Składnia

```
#pragma omp for schedule(rodzaj [,rozmiar_segmentu])
```

- Możliwe rodzaje podziału pracy:
 - **static** – podział dokonany przed uruchomieniem pętli, najmniejszy narzut czasu wykonania
 - **dynamic** – wątki wykonują kolejno „pierwszy wolny” segment w przestrzeni instrukcji for
 - **guided** – trochę jak w dynamic, wielkość segmentu może ulegać zmniejszaniu
 - **runtime** – podział zależy od wartości zmiennej środowiskowej OMP_SCHEDULE

Opcja `schedule(static)`

- Podział statyczny

```
#pragma omp for schedule(static)
for (i = 0; i < n; i++) {
    id = omp_get_thread_num();
    if( i % 2 == 0 ) // parzyste sa coraz bardziej kosztowne
        for (k = 1; k < i*1000000000; k++) z += 1.0/k;
    printf("Iteracja  %d wykonana przez watek nr.  %d.\n", i, id);
}
```

```
Iteracja 5 wykonana przez watek nr. 1.
Iteracja 0 wykonana przez watek nr. 0.
Iteracja 1 wykonana przez watek nr. 0.
Iteracja 2 wykonana przez watek nr. 0.
Iteracja 3 wykonana przez watek nr. 0.
Iteracja 6 wykonana przez watek nr. 1.
Iteracja 7 wykonana przez watek nr. 1.
Iteracja 4 wykonana przez watek nr. 0.
Iteracja 8 wykonana przez watek nr. 1.
Iteracja 9 wykonana przez watek nr. 1.
czas: 13.486
```

Opcja `schedule(static, chunk_size)`

- Podział statyczny ze specyfikacją długości segmentu

```
#pragma omp for schedule(static, 1)
for (i = 0; i < n; i++) {
    id = omp_get_thread_num();
    if( i % 2 == 0 ) // parzyste sa coraz bardziej kosztowne
        for (k = 1; k < i*1000000000; k++) z += 1.0/k;
    printf("Iteracja  %d wykonana przez watek nr.  %d.\n", i, id);
}
```

```
Iteracja 1 wykonana przez watek nr. 1.
Iteracja 3 wykonana przez watek nr. 1.
Iteracja 5 wykonana przez watek nr. 1.
Iteracja 7 wykonana przez watek nr. 1.
Iteracja 9 wykonana przez watek nr. 1.
Iteracja 0 wykonana przez watek nr. 0.
Iteracja 2 wykonana przez watek nr. 0.
Iteracja 4 wykonana przez watek nr. 0.
Iteracja 6 wykonana przez watek nr. 0.
Iteracja 8 wykonana przez watek nr. 0.
czas: 20.808
```

Opcja `schedule(static, chunk_size)`

- Podział statyczny ze specyfikacją długości segmentu (2)

```
#pragma omp for schedule(static, 2)
for (i = 0; i < n; i++) {
    id = omp_get_thread_num();
    if( i % 2 == 0 ) // parzyste sa coraz bardziej kosztowne
        for (k = 1; k < i*1000000000; k++) z += 1.0/k;
    printf("Iteracja  %d wykonana przez watek nr.  %d.\n", i, id);
}
```

```
Iteracja 0 wykonana przez watek nr. 0.
Iteracja 1 wykonana przez watek nr. 0.
Iteracja 2 wykonana przez watek nr. 1.
Iteracja 3 wykonana przez watek nr. 1.
Iteracja 4 wykonana przez watek nr. 0.
Iteracja 5 wykonana przez watek nr. 0.
Iteracja 6 wykonana przez watek nr. 1.
Iteracja 7 wykonana przez watek nr. 1.
Iteracja 8 wykonana przez watek nr. 0.
Iteracja 9 wykonana przez watek nr. 0.
czas: 12.789
```

Opcja `schedule(dynamic)`

- Podział dynamiczny

```
#pragma omp for schedule(dynamic)
for (i = 0; i < n; i++) {
    id = omp_get_thread_num();
    if( i % 2 == 0 ) // parzyste sa coraz bardziej kosztowne
        for (k = 1; k < i*1000000000; k++) z += 1.0/k;
    printf("Iteracja  %d wykonana przez watek nr.  %d.\n", i, id);
}
```

```
Iteracja 0 wykonana przez watek nr. 1.
Iteracja 1 wykonana przez watek nr. 0.
Iteracja 3 wykonana przez watek nr. 0.
Iteracja 2 wykonana przez watek nr. 1.
Iteracja 5 wykonana przez watek nr. 1.
Iteracja 4 wykonana przez watek nr. 0.
Iteracja 7 wykonana przez watek nr. 0.
Iteracja 6 wykonana przez watek nr. 1.
Iteracja 9 wykonana przez watek nr. 1.
Iteracja 8 wykonana przez watek nr. 0.
czas: 12.672
```


Opcja `schedule(dynamic(chunk_size))`

- Podział dynamiczny ze specyfikacją długości segmentu

```
#pragma omp for schedule(dynamic, 3)
for (i = 0; i < n; i++) {
    id = omp_get_thread_num();
    if( i % 2 == 0 ) // parzyste sa coraz bardziej kosztowne
        for (k = 1; k < i*1000000000; k++) z += 1.0/k;
    printf("Iteracja  %d wykonana przez watek nr.  %d.\n", i, id);
}
```

```
Iteracja 0 wykonana przez watek nr. 1.
Iteracja 3 wykonana przez watek nr. 0.
Iteracja 1 wykonana przez watek nr. 1.
Iteracja 2 wykonana przez watek nr. 1.
Iteracja 4 wykonana przez watek nr. 0.
Iteracja 5 wykonana przez watek nr. 0.
Iteracja 9 wykonana przez watek nr. 0.
Iteracja 6 wykonana przez watek nr. 1.
Iteracja 7 wykonana przez watek nr. 1.
Iteracja 8 wykonana przez watek nr. 1.
czas: 15.23
```

Opcja schedule(guided)

- Podział „naprowadzany”

```
#pragma omp for schedule(guided)
for (i = 0; i < n; i++) {
    id = omp_get_thread_num();
    if( i % 2 == 0 ) // parzyste sa coraz bardziej kosztowne
        for (k = 1; k < i*1000000000; k++) z += 1.0/k;
    printf("Iteracja  %d wykonana przez watek nr.  %d.\n", i, id);
}
```

```
Iteracja 0 wykonana przez watek nr. 1.
Iteracja 5 wykonana przez watek nr. 0.
Iteracja 1 wykonana przez watek nr. 1.
Iteracja 2 wykonana przez watek nr. 1.
Iteracja 3 wykonana przez watek nr. 1.
Iteracja 4 wykonana przez watek nr. 1.
Iteracja 6 wykonana przez watek nr. 0.
Iteracja 7 wykonana przez watek nr. 0.
Iteracja 9 wykonana przez watek nr. 0.
Iteracja 8 wykonana przez watek nr. 1.
czas: 13.416
```

Dyrektywa single

- Dyrektywa single identyfikuje kod wykonywany przez jeden wątek

```
#pragma omp parallel default(none)\
    private(i, id, k) shared(z, n, chunk_size)
{
    #pragma omp single
    {
        printf("Program jest wykonywany na %d watkach.\n",
            omp_get_num_threads());
    }
    #pragma omp for schedule(guided)
    for (i = 0; i < n; i++) {
        id = omp_get_thread_num();
        if( i % 2 == 0 )
            for (k = 1; k < i*100000000; k++)
                z += 1.0/k;
        printf("Iteracja  %d wykonana przez watek nr.  %d.\n", i, id);
    }
}
```

Blok sections

- Dyrektywa **sections** wyodrębniania fragmenty kodu wykonywane jednocześnie przez różne wątki
- Poszczególne zadania umieszcza się w blokach **section**

```
#pragma omp parallel sections shared(x,y)
{
# pragma omp section
{
    cout << "f(x) obliczy  watek " << omp_get_thread_num() << "\n";
    f(x);
}

# pragma omp section
{
    cout << "g(y) obliczy  watek " << omp_get_thread_num() << "\n";
    g(y);
}
}
```

Blok sections

- Kolejność wykonywania bloków **section** jest niekreślona, podobnie jak przydział wątków do ich realizacji
- Poszczególne zadania umieszczone w blokach **section** muszą być niezależne od siebie

```
#pragma omp parallel sections shared(x,y)
{
# pragma omp section
    f(x);
# pragma omp section
    g(x);
# pragma omp section
    h(x, x);
# pragma omp section
    j(x, 1);
}
```

Jak zrównoleglać sumy, iloczyny itp?

- Suma ma być shared czy private?

```
double suma_zla (int n)
{
    double suma = 0;
    int i;
    # pragma omp parallel for default(none)\
        shared(suma, n) private(i) schedule(static)
    for (i = 2; i <= n; i++)
        suma += (!(i & 1)) ? 1.0 / log(i) : -1.0 / log(i);
    return suma;
}
```

- Oba rozwiązania złe:

n = 100000000	
Bez OpenMP:	czas = 10.6 , suma = 0.951443
OpenMP, shared , schedule(static)	czas = 5.79 , suma = 148303
OpenMP, shared , schedule(dynamic)	czas = 34.0 , suma = 84.287
OpenMP, private , schedule(dynamic)	czas = 21.2 , suma = 0

Opcja reduction

- Opcja **reduction** służy do definiowania zmiennych jako wyników sum, iloczynów etc.

```
double suma_dobra (int n)
{
    double suma = 0;
    int i;
    # pragma omp parallel for \
        default(none) shared(n) private(i), reduction(+ : suma)
    for (i = 2; i <= n; i++)
        suma += (!(i & 1)) ? 1.0 / log(i) : -1.0 / log(i);
    return suma;
}
```

- Opcja **reduction** daje poprawny wynik w rozsądnym czasie:

```
n = 100000000
Bez OpenMP,  czas: 10.628,  suma = 0.951443
Z OpenMP,    czas: 5.582,   suma = 0.951443
```

Opcja reduction

- Opcję **reduction** można wykorzystywać z większością operatorów binarnych:

+, -, *, ^, |, &, &&, ||

```
# pragma omp parallel for \  
    default(none) shared(n) private(i) reduction(*: iloczyn)  
for (i = 0; i <= n; i++)  
    iloczyn *= 1/(i*i + 1.0);
```


Dyrektywa if

- Dyrektywa **if** służy do warunkowego zrównoleglania pętli

```
# pragma omp parallel for \  
    if (n > 100000) \  
        default(none) shared(n) private(i) reduction(*: iloczyn)  
for (i = 0; i <= n; i++)  
    iloczyn *= 1/(i*i + 1.0);
```

Dyrektywa `num_threads`

- Dyrektywa `num_threads` służy do określania liczby wątków obsługujących blok `parallel`

```
# pragma omp parallel for \  
    if (n > 100000) \  
        num_threads(2) \  
        default(none) shared(n) private(i) reduction(*: iloczyn)  
for (i = 0; i <= n; i++)  
    iloczyn *= 1/(i*i + 1.0);
```

Część 1

- 1 Kompilacja
- 2 Podział pracy
- 3 Synchronizacja**

Synchronizacja wątków

Do synchronizacji lub desynchronizacji pracy wątków służą m.in. dyrektywy:

- `barrier`
- `critical`
- `master`
- `atomic`

i opcje:

- `nowait`

Dyrektywa barrier

- Dyrektywa **barrier** wstrzymuje wątki, aż wszystkie wątki zespołu osiągną barierę

```
# pragma omp parallel
{
    ...
#   pragma omp barrier
    ...
}
```

- Bariery domyślnie ustawiane są na końcu bloków instrukcji objętych niektórymi dyrektywami, m.in. `for`, `sections` i `single`.

Opcja `nowait`

- Opcja `nowait` wyłącza domyślną barierę

```
# pragma omp parallel
{
    ...
#   pragma omp single nowait
    ...
#   pragma omp for nowait
    ...
#   pragma omp sections nowait
    ...
}
```

Dyrektywa `critical`

- Dyrektywa `critical` tworzy region („sekcję krytyczną”), który może być wykonywany przez co najwyżej jeden wątek naraz
- Nazwany region krytyczny obejmuje wszystkie sekcje krytyczne o tej samej nazwie
- Służy do unikania pościgu (*race condition*) przy dostępie do zmiennych
- Powinna zawierać kod, który wykonuje się szybko, by nie blokować pracy innych wątków
- Nie powinna zastępować innych konstrukcji, np. opcji `reduction`

```
# pragma omp parallel
{
    ...
    # pragma omp critical (sumowanie)
    {
        suma += suma_lokalna_w_watku;
    }
    ...
}
```

Dyrektywa `atomic`

- Dyrektywa `atomic` sygnalizuje, że objęta nią prosta instrukcja przypisania do zmiennej współdzielonej wykona się atomowo (wątek nie zostanie wywołany przed jej zakończeniem).

```
# pragma omp parallel shared(x)
{
    ...
    # pragma omp atomic
    {
        x++;
    }
    ...
}
```

- Dopuszczalne operacje atomowe: `++`, `--`, `+=`, `*=`, `-=`, `/=`, `&=`, `^=`, `|=`, `<<=`, `>>=`

Funkcje OpenMP

```
void omp_set_num_threads (int); // ustaw # watkow w nast. parallel
int omp_get_num_threads (void); // ile aktywnych watkow?
int omp_get_max_threads (void);
int omp_get_thread_num (void); // numer watku
int omp_get_num_procs (void); // liczba procesorow

double omp_get_wtime (void);
double omp_get_wtick (void);

int omp_in_parallel (void);

void omp_set_dynamic (int);
int omp_get_dynamic (void);

void omp_set_schedule (omp_sched_t, int);
void omp_get_schedule (omp_sched_t *, int *);
```

Zmienne środowiskowe

- OMP_SCHEDULE
- OMP_DYNAMIC
- OMP_NUM_THREADS
- OMP_NESTED

```
> env OMP_NUM_THREADS=4 ./a.out
```

OpenMP w bibliotece standardowej

- Przykład: sortowanie 50000000 liczb.

```
const int N = 50000000;  
std::vector<double> v(N);  
srand(0);  
for (int i = 0; i < N; i++)  
    v[i] = rand();  
std::sort(v.begin(), v.end());
```

- Kompilacja kompilatorem g++ 4.5

```
> g++ -O2 -D_GLIBCXX_PARALLEL -fopenmp test.cpp
```

- Przyspieszenie: 4 razy na 6-rdzeniowym AMD Phenom II X6 1055T

OpenMP w bibliotece standardowej

- Przykład: sortowanie 50000000 liczb.

```
const int N = 50000000;
std::vector<double> v(N);
srand(0);
for (int i = 0; i < N; i++)
    v[i] = rand();
std::sort(v.begin(), v.end());
```

- Kompilacja kompilatorem g++ 4.5

```
> g++ -O2 -D_GLIBCXX_PARALLEL -fopenmp test.cpp
```

- Przyspieszenie: 4 razy na 6-rdzeniowym AMD Phenom II X6 1055T
- Bez modyfikacji kodu źródłowego!!!

OpenMP w bibliotece standardowej

- Wśród algorytmów biblioteki standardowej zrównoleglonych w g++ znajdują się m.in.:
 - `accumulate`
 - `inner_product`
 - `partial_sum`
 - `adjacent_difference`
 - `find`
 - `for_each`
 - `sort`
 - `stable_sort`
 - `search`
 - `min_element`, `max_element`
 - `replace`

Literatura

- *OpenMP Application Program Interface Version 3.0 May 2008*
- www.openmp.org
- Barbara Chapman, Gabriele Jost, Ruud van der Pas, *Using OpenMP Portable Shared Memory Parallel Programming*, The MIT Press, 2008
- Paweł Przybyłowicz, *Kurs OpenMP - pierwsze kroki*
- The GNU C++ Library Manual