

Wprowadzenie do OpenMP

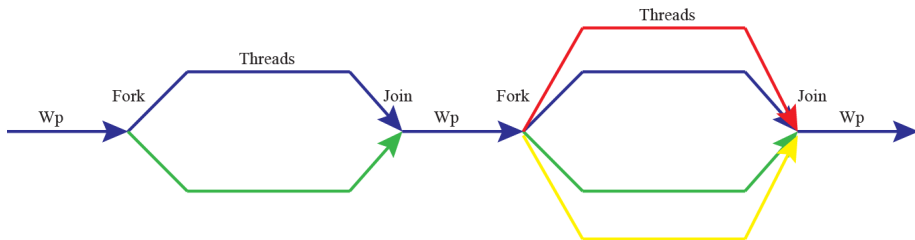
OZUKO

Kamil Dworak

OpenMP (ang. *Open Multi-Processing*)

- opracowany w 1997 przez radę Architecture Review Board,
- obliczenia równoległe z użyciem **pamięci wspólnej** (ang. *Shared Memory Computation*),
- rozszerzenie C/C++ i Fortran o nowe dyrektywy kompilatora, biblioteki funkcji oraz zmienne środowiskowe,
- obliczenia wykonywane są za pomocą **1 procesu**, składającego się z zespołu **wątków**.

Realizacja programu równoległego



Rysunek: Realizacja programu równoległego za pomocą OpenMP

Pierwszy program

```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    #pragma omp parallel
    {
        cout << "Watek " << omp_get_thread_num() << "pracuje \n";
        if(omp_get_thread_num() == 2) {
            cout << "Dodatkowe obliczenia\n";
        }
    }
    system("PAUSE");
    return 0;
}
```

Wynik działania powyższego programu

```
Watek 0 pracuje  
Watek 2 pracuje  
Dodatkowe obliczenia  
Watek 1 pracuje  
Watek 3 pracuje
```

Konstrukcja równoległa

Wszystkie funkcje OpenMP znajdują się w pliku:

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

Za pomocą dyrektyw preprocesora wskazujemy kompilatorowi fragmenty programu, które mają być realizowane równoległe.

Ogólna postać dyrektywy:

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

Przykład:

```
#pragma omp parallel shared(tab) lastprivate(i)
```

Kompilacja programu równoległego

Kompilacja programu równoległego może zostać zrealizowana za pomocą:

- GCC

```
gcc -fopenmp program_zrodlowy.c -o program_wynikowy
```

liczbę aktywnych wątków można ustawić wykorzystując komendę:

```
setenv OMP_NUM_THREADS 6
```

- Visual Studio → *ProjectPreferences* → *ConfigurationProperties* →
→ *C\C++* → *Language* → *OpenMPSupport* →
→ *Yes(/openmp)*.

Ustawienie liczby wątków

- z poziomu kody za pomocą funkcji `omp_set_num_threads()`

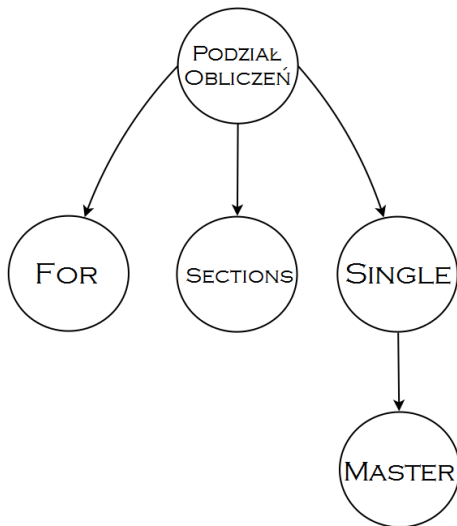
```
#include <iostream>
#include <omp.h>
using namespace std;

int main() {
    omp_set_num_threads(4);
    // ...
}
```

- klauzula `num_threads`:

```
#pragma omp parallel num_threads(2)
```

- zmienna środowiskowa Visual Studio - Project Preferences →
→ *Configuration Properties* → *Debugging* →
→ *Environment* → **OMP_NUM_THREADS=4**



Rysunek: Podział obliczeń

Konstrukcja iteracyjna for

```
//...
const int N = 10;
int main() {
    int i;
    int result[10] = {};
    int tab[N] = {0, 1, 2, 3, 4, 5, 6, 7 ,8 ,9 };
    omp_set_num_threads(4);
    #pragma omp parallel shared(result, tab, N) \
        private(i)
    {
        #pragma omp for
        for(i = 0; i < N; i ++) {
            result[i] = tab[i] * tab[i];
            printf("Wynik: %d (watek = %d) ", result[i],
                omp_get_thread_num());
        }
    } //...
```

Wynik działania powyższego programu:

```
Wynik: 0 (watek = 0) Wynik: 1 (watek = 0) Wynik: 36(watek = 2)
Wynik: 64(watek = 3) Wynik: 81(watek = 3) Wynik: 49(watek = 2)
Wynik: 4 (watek = 0) Wynik: 9 (watek = 1) Wynik: 16(watek = 1)
Wynik: 25(watek = 1)
```

- Wersja oryginalna:

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

- Dopuszczalna wersja skrócona:

```
#pragma omp parallel for [klauzule]
for(int i = 0; i < N; i++)
```

Problem redukcji

```
// ...  
int buffor;  
#pragma omp parallel shared(suma, tab, N) private (buffor)  
{  
    buffor = 0;  
    #pragma omp for  
    for(int i = 0; i < N; i++) {  
        buffor += tab[i];  
    }  
  
    #pragma omp critical  
    suma += buffor;  
}  
// ...
```

Problem redukcji

Klauzula reduction:

```
// ...  
#pragma omp parallel for shared(N, tab) reduction(+:suma)  
for(int i = 0; i < N; i ++) {  
    suma += tab[i];  
}  
// ...
```

Operatory redukcji:

+ - * & | ^ && ||

Konstrukcja sekcji sections

Dyrektywa sections:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        dodaj(tab1, tab2);

        #pragma omp section
        odejmij(tab3, tab4);

        #pragma omp section
        pomnoz(tab5, tab6);
    }
}
```

Konstrukcja pojedynczego wątku

Dyrektywa single:

```
#pragma omp parallel
{
    #pragma omp single
    printf("Start - watek: %d\n", omp_get_thread_num());
    // obliczenia...

    #pragma omp single
    printf("Koniec - watek: %d\n", omp_get_thread_num());
}
```


Podział obliczeń (3) - konstrukcja pojedynczego wątku

Dyrektywa master:

```
#pragma omp parallel
{
    #pragma omp master
    printf("Start - watek: %d\n", omp_get_thread_num());
    // obliczenia...

    #pragma omp master
    printf("Koniec - watek: %d\n", omp_get_thread_num());
}
```

Wyniki działania (3) - konstrukcja pojedynczego wątku

Wyniki działania dla konstrukcji pojedynczego wątku:

- Single:

```
Start - watek: 0  
Koniec - watek: 2
```

- Master:

```
Start - watek: 0  
Koniec - watek: 0
```

Master - Przykład Drugi

```
#pragma omp parallel
{
    #pragma omp for shared(N, wyniki)
    for(int i = 0; i < N; i ++){
        wyniki = obliczWartoscFunkcji(i);

    #pragma omp master
    for(int i = 0; i < N; i ++){
        cout << wyniki[i] << endl;

    #pragma omp for
    // kolejne obliczenia...
    }
```

- firstprivate:

```
#pragma omp parallel for firstprivate(a) private(i)
    shared(tab)
    for(i = 0; a > 0; i++) {
        result[i] = tab[i] * a;
        a--;
    }
```

- lastprivate:

```
#pragma omp parallel for lastprivate(i) \
    shared(tab1, tab2, N)
    for(i = 0; i < N; i++) {
        result[i] = tab1[i] * tab2[i];
    }
```

- default:

```
#pragma omp parallel for default(shared)
for(int i = 0; i < N; i++) {
    result[i] = tab1[i] * tab2[i];
}
```

- nowait:

```
#pragma omp sections nowait
{
    #pragma omp section
    func1();

    #pragma omp section
    func2();
}
```

- if:

```
#pragma omp parallel if(N > 100000) shared(tab, N)
{
    #pragma omp for
    for(int i = 0; i < N; i ++) {
        // ...
    }
```

- num_threads:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    cout << "Start!\n";
    // ...
}
```

- barrier:

```
#pragma omp parallel private(id) shared(N)
{
    id = omp_get_thread_num();
    duzaLiczbaObl(id);
    #pragma omp barrier
    #pragma omp for
    // ...
```

- atomic:

```
#pragma omp for private(i) shared(wyniki, czynnik, N)
for(i = 0; i < N; i++) {
    wyniki[i] = i * czynnik;
    #pragma omp atomic
    czynnik -= 2;
    // dopuszczalne operacje: + - * / & ^ | << >>
```

- ordered:

```
#pragma omp parallel for shared(result, N) ordered
for(int i = 0; i < N; i++) {
    result[i] = i + i;
    #pragma omp ordered
    cout << result[i] << endl;
    // 0, 2, 4, 6, 8, 10...
}
```


Pomiar czasu wykonywania programu możemy zrealizować za pomocą:

- **omp_get_wtime**

```
double start = omp_get_wtime();  
// obliczenia...  
double end = omp_get_wtime();  
cout << "Czas trwania programu: " << end - start << endl;
```

- **clock_t**

```
#include <ctime>  
// ...  
clock_t start = clock();  
// obliczenia...  
clock_t end = clock();  
cout << "Czas trwania programu: " << end - start << endl;
```