Standard OpenMP

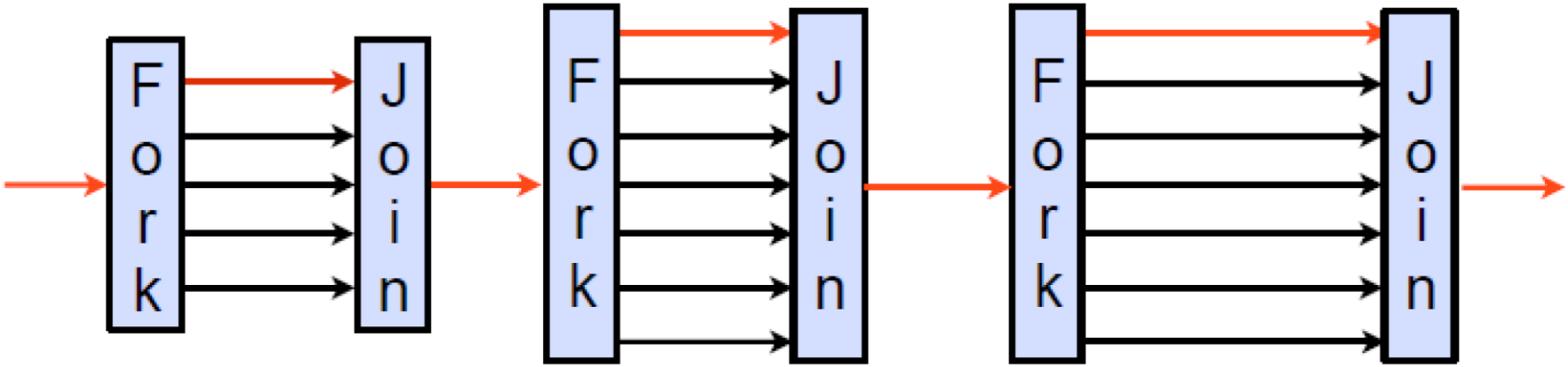
dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Model programowania OpenMP

- Współbieżność jest oparta na modelu współdzielonej pamięci i mechanizmie wątków
- Współbieżność jest kontrolowana przez programistę
- Definiowanie mechanizmów współbieżności jest oparte na dyrektywach kompilatora wbudowanych w kod programu.
  - Jeśli w programie jest wyłączone wsparcie OpenMP program wykonuje się sekwencyjnie.

Współbieżność typu Fork-Join

- Program zgodny z OpenMP rozpoczyna się jako pojedynczy proces. Główny wątek programu wykonuje się sekwencyjnie dopóki nie zostanie zainicjowany pierwszy współbieżny region.
- W chwili inicjacji regionu współbieżnego główny wątek:
  - Tworzy grupę wątków z zastosowaniem mechanizmu rozgałęzienia (fork)
  - Pozostaje nadrzędnym wątkiem (master) i otrzymuje identyfikator 0.
  - Instrukcje zawarte w regionie współbieżnym są wykonywane współbieżnie przez utworzone wątki.
  - Na wszystkich wątkach jest założona niejawna bariera. Na barierze następuje synchronizacja, zakończenie wykonywania obliczeń przez wątki, a następnie przekazanie sterowania do wątku nadrzędnego.



Czerwony to wątek główny.

Wejście/wyjście

- OpenMP nie specyfikuje współbieżnego wejścia/wyjścia
- Programista ma zapewnić poprawność operacji wej/wyj w kontekście programu wielowątkowego.

Model pamięci

- Wątki mogą „cash'ować" ich dane w czasie wykonywania i nie jest wymagane utrzymywanie spójności z rzeczywistą pamięcią przez cały czas
- Kiedy wartość danych ma być aktualna w realnej pamięci i widoczna dla wszystkich wątków, to programista ma o to zadbać.

Pierwszy program OpenMP

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

int main()
{
#pragma omp parallel
 {
   int ID = omp_get_thread_num();
   printf("Hello(%d) ", ID);
   printf("World(%d) \n", ID);
 }
return 0;
}
```

Wyjście:

```
Hello(0) Hello(1) World(0)
Hello(2) World(1)
World(2)
Hello(4) World(4)
Hello(3) World(3)
Hello(5) World(5)
Hello(6) World(6)
Hello(7) World(7)
```

- Fragment programu objęty zasięgiem dyrektywy „pragma omp parallel” jest automatycznie dekomponowany na zbiór wątków ustalony jako zmienna systemowa lub odpowiadający ilości rdzeni procesora.
- Funkcja „omp_get_thread_num()” zwraca bieżący numer wątku.

Czy można zrównoleglić wykonywanie pętli „for”?

Nie?

```
for (i=1; i<n; i++)
{
  a[i] = 2 * a[i-1];
}
```

Tak?

```
for (i=1; i<n; i++)
{
  b[i] = (a[i] - a[i-1]) * 0.5;
}
```

Iteracje są niezależne od siebie!

Przykład zrównoleglenia pętli „for”

```
#include <stdio.h>
#include <omp.h>
#define N 10
int main(void) {
float a[N], b[N];
int i;
#pragma omp parallel default(none), \
private(i), shared(a,b)
{
#pragma omp for
for (i = 0; i < N; i++) {
a[i] = (i + 1) * 1.0f;
b[i] = (i + 1) * 2.0f;
printf("%d, %f, %f \n", i + 1, a[i],
b[i]);
}
}
return 0;
}
```

Wyjście:

```
1, 1.000000, 2.000000
2, 2.000000, 4.000000
3, 3.000000, 6.000000
6, 6.000000, 12.000000
4, 4.000000, 8.000000
8, 8.000000, 16.000000
5, 5.000000, 10.000000
9, 9.000000, 18.000000
10, 10.000000, 20.000000
7, 7.000000, 14.000000
```

Przykład zrównoleglenia pętli „for” - komentarze

- „#pragma omp for” w prwadza region współbieżnego wykonywania pętli for
- Klauzule zastosowane w przykładowym programie:
  - „default”
    - Dostępne parametry: „shared” lub „none”
    - default(shared) – wszystkie zmienne w regionie są uważane za dzielone, chyba, że wskażemy, które mają być prywatne (należące tylko do danego wątku)
    - default(none) – programista musi jawnie wskazać dla wszystkich zmiennych, czy mają być współdzielone, czy prywatne dla wątków.
  - „shared”
    - Określa, które ze zmiennych we współbieżnym regionie mają być współdzielone
  - „private”
    - Określa, które ze zmiennych we współbieżnym regionie mają być prywatne dla wątku.

Fork – Join, Fork - Join

```
#include <stdio.h>
#include <omp.h>
#define N 10
int main(void) {
float a[N], b[N], c[N];
int i, TID, nthreads;
omp_set_num_threads(3);
#pragma omp parallel default(none),\
private(i), shared(a,b)
{
#pragma omp for
 for (i = 0; i < N; i++) {
 a[i] = (i + 1) * 1.0;
 b[i] = (i + 1) * 2.0;
 }
}
#pragma omp parallel default(none),\
private(i,TID), shared(a,b,c,nthreads)
{
TID = omp_get_thread_num();
if (TID == 0) {
nthreads = omp_get_num_threads();
printf("Number of threads = (%d) \n",
nthreads);
}
printf("Thread %d starting \n", TID);
```

```
#pragma omp for
for (i = 0; i < N; i++) {
c[i] = a[i] + b[i];
printf("%d, %d, %f, %f, %f \n", TID, i + 1,
a[i], b[i], c[i]);
}
}
}
```

Wyjście:

```
Thread 2 starting
Thread 1 starting
Number of threads = (3)
1, 5, 5.000000, 10.000000, 15.000000
2, 8, 8.000000, 16.000000, 24.000000
1, 6, 6.000000, 12.000000, 18.000000
Thread 0 starting
2, 9, 9.000000, 18.000000, 27.000000
1, 7, 7.000000, 14.000000, 21.000000
2, 10, 10.000000, 20.000000, 30.000000
0, 1, 1.000000, 2.000000, 3.000000
0, 2, 2.000000, 4.000000, 6.000000
0, 3, 3.000000, 6.000000, 9.000000
0, 4, 4.000000, 8.000000, 12.000000
```

Fork-Join, Fork-Join - komentarze

- Program wykonuje obliczenia w dwóch regionach współbieżnych
- omp_set_num_threads(2) – pozwala na ustalenie liczby działających wątków
- Funkcja „omp_get_num_threads()” podaje liczbę wątków w bieżącym regionie współbieżnym
- W programie pokazano, w jaki sposób zorganizować obliczenia powiązane z pojedynczym wątkiem, takie, które nie będą wykonywane w innych.

Problem sumowania wyników z współbieżnej pętli „for”

```
#include <stdio.h>
#include <omp.h>
#define N 10
int main() {
int i;
float a[N], b[N], partial_result,
result = 0;;
omp_set_num_threads(4);
#pragma omp parallel default(none),\
private(i), shared(a,b)
 {
  #pragma omp for
  for (i = 0; i < N; i++) {
  a[i] = (i + 1) * 1.0;
  b[i] = (i + 1) * 2.0;
  }
 }
#pragma omp parallel default(none),\
private(i,partial_result),
shared(a,b,result)
 {
  partial_result = 0.0;
  #pragma omp for
  for (i = 0; i < N; i++)
  partial_result =
    partial_result +  (a[i] * b[i]);
```

```
#pragma omp critical
  result = result + partial_result;
 }
printf("Final result= %f \n", result);
return 0;
}
```

Wyjście:

```
Final result= 770.000000
```

Problem sumowania wyników z współbieżnej pętli „for” - komentarze

- Rezultatem działania każdej iteracji pętli jest pewna wartość.
- Zadanie polega na prawidłowym zsumowaniu tych wartości:
  - Dla każdego wątku wprowadzania jest lokalna zmienna do sumowania rezultatów iteracji w obrębie wątku.
  - Ostateczna suma obliczana jest w sekcji krytycznej jako suma lokalnych zmiennych.
- W podobny sposób można rozwiązać problem dla operacji *, -, &, |, ^, &&, ||.

Problem sumowania wyników z współbieżnej pętli „for” - rozwiązanie z zastosowaniem klauzuli redukcji

```
#include <stdio.h>
#include <omp.h>
#define N 10
int main() {
int i;
float a[N], b[N], result;
omp_set_num_threads(4);
#pragma omp parallel default(none),\
private(i), shared(a,b)
{
#pragma omp for
for (i = 0; i < N; i++)
{
a[i] = (i + 1) * 1.0;
b[i] = (i + 1) * 2.0;
}
}
result = 0.0;
#pragma omp parallel default(none),\
private(i), shared(a,b)
reduction(+:result)
{
#pragma omp for
for (i = 0; i < N; i++)
result = result + (a[i] * b[i]);
}
printf("Final result= %f \n", result);
return 0;
}
```

Wyjście:

```
Final result= 770.000000
```

Sekwencyjne rozwiązanie sortowania przez zliczanie

```
#include <stdio.h>
#define N 5

float a[N] ={4.1f,3.2f,1.4f,5.2f,10.8f};
float b[N];
int w[N][N];
int ind[N];
int i, j;

int main(void)
{
 for (i = 0; i < N; i++){
  for (j = 0; j < N; j++) {
   if ((a[i] > a[j]) ||
       (a[i] == a[j] && i > j)
      )
      w[i][j] = 1;
   else
      w[i][j] = 0;
   }
 }
```

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

Proste współbieżne rozwiązanie sortowania przez zliczanie

```
#include <stdio.h>
#include <omp.h>
#define N 5

float a[N] ={4.1f,3.2f,1.4f,5.2f,10.8f};
float b[N] ={0.0f,0.0f,0.0f,0.0f,0.0f};
int w[N][N];
int ind[N];
int i, j;

int main(void)
{
#pragma omp parallel default(none),\
private(i,j), shared(a,b,w,ind)
 {
#pragma omp for
  for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
    if ((a[i] > a[j]) ||
       (a[i] == a[j] && i > j)
       )
     w[i][j] = 1;
    else
     w[i][j] = 0;
   }
  }
 }
```

```
#pragma omp parallel default(none),\
private(i,j), shared(a,b,w,ind)
 {
#pragma omp for
  for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
    ind[i] += w[i][j];
   }
   b[ind[i]] = a[i];
  }
}
return 0;
}
```

Współbieżne rozwiązanie sortowania przez zliczanie z zastosowaniem N^2 wątków

```
#include <stdio.h>
#include <omp.h>
#define N 5

float a[N] ={4.1f,3.2f,1.4f,5.2f,10.8f};
float b[N] ={0.0f,0.0f,0.0f,0.0f,0.0f};
int w[N][N];
int ind[N];
int i, j;

int main(void)
{
#pragma omp parallel for num_threads(N)
 for (i = 0; i < N; i++) {
#pragma omp parallel for num_threads(N)
  for (j = 0; j < N; j++) {
   if ((a[i] > a[j]) ||
      (a[i] == a[j] && i > j))
    w[i][j] = 1;
   else
    w[i][j] = 0;
  }
 }
```

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

Pragama firsprivate i lastprivate

```
#include <stdio.h>
#include <omp.h>
#define N 5
int main(void) {
int a[N];
int x=15,y,z;
int i = 0;
omp_set_num_threads(3);
#pragma omp parallel default(shared), \
private(i), firstprivate(x)
{
#pragma omp for lastprivate(z)
for (i = 0; i < N; i++) {
a[i] = x++;
printf("%d, %d, %d \n", i, a[i], x);
z = x;
}
}
y = x;
printf("\n%d, %d, %d, %d, %d \n", 0,
a[i], x, y, z);
return 0;
}
```

Wyjście:

```
0, 15, 16
2, 15, 16
1, 16, 17
3, 16, 17
4, 15, 16

0, 15, 15, 15, 16
```

Pragama firsprivate i lastprivate - komentarze

- „firstprivate”
  - Zmienne „private” są nieokreślone podczas startu regionu współbieżnego. Można je zainicjalizować wewnątrz bloku lub wskazać za pomocą klauzuli „firstprivate”, że początkową wartością zmiennej prywatnej w każdym wątku ma być jej wartość „sprzed” regionu współbieżnego.
- „lastprivate”
  - Zmiennej globalnej przypisuje się rezultat ostatniej iteracji współbieżnie wykonywanej instrukcji „for”.