Środowisko OpenMP

Łukasz J. Wielebski <lukaszjw@po.onet.pl>
Maciej Polewczyński <polew@spider.pl>
Rafał Bisingier <ravbc@man.poznan.pl>
Maciej Regulski <macius@fanthom.math.put.poznan.pl>

1 czerwca 2001 r.

Streszczenie

Niniejszy artykuł stanowi próbę przyjrzenia się środowisku *OpenMP*, które jest implementacją modelu programowania równoległego z „przezroczystym" dostępem do pamięci współdzielonej.

W dalszej części zawarto również propozycje pytań egzaminacyjnych.

Spis treści

1 Wprowadzenie

OpenMP jest przede wszystkim:

- Application Program Interface (API) które może służyć do tworzenia wielowątkowych rozwiązań równoległego dostępu do współdzielonej pamięci,
- Połączeniem trzech elementów API:
  - dyrektyw kompilatora,
  - procedur (runtime library routines),
  - zmiennych środowiskowych,
- przenośne:
  - API zostało zdefiniowane dla C/C++ i Fortrana,
  - OpenMP zostało zaimplementowane na wielu platformach, zarówno na różbnych UNIX'ach, jak również pod Windows NT
- ustandaryzowane:
  - zdefiniowane i zaaprobowane wspolnie przez głównych dostawcow sprzetu i oprogramowania,
  - spodziewane jest ogloszenie OpenMP jako standardu ANSI.

Zastrzeżenie jest takie, że to użytkownik jest przede wszystkim zobowiązany do troszczenia się o takie aspekty (problemy) programowania, jak np.:

- zależności,
- konflikty,
- zakleszczenia,
- „wyścigi" do danych,
- inne drobiazgi ;).

OpenMP używa modelu równoległego wykonania *fork–join*. Chociaż model *fork–join* może być użyteczny dla rozwiązywania różnych problemów, to jest zaprojektowany przede wszystkim dla aplikacji korzystających z dużych tablic. Ten model pozwala tak pisać programy, aby rozdzielały się na wiele wątków wykonawczych. Możliwe jest takie napisanie programu, który przez cały czas działa w więcej niż jednym wątku (np. sytuacja, gdzie wyniki uzyskane przy wykonaniu sekwencyjnym są nieprawidłowe). Możliwe jest też takie napisanie programu, że ten wykonuje się sekwencyjnie (np. ingorując dyrektywy *OpenMP*) lub też równolegle – zależnie od dyrektyw kompilacji.

Program napisany z użyciem *OpenMP* C/C++ API rozpoczyna swoje wykonanie jako pojedynczy wątek, zwany *wątkiem głównym*. Główny wątek wykonuje się jako liniowy program („region"), aż do napotkania pierwszej instrukcji równoległej. W *OpenMP* C/C++ API, konstrukcja równoległa jest oznaczana dyrektywą „parallel". Kiedy zostanie napotkana konstrukcja równoległa, wątek główny tworzy zbiór wątków, jednocześnie stając się wątkiem głównym tego zbioru. Każdy wątek w zbiorze wykonuje instrukcje w ramach obszaru („regionu") równoległego, za wyjątkiem konstrukcji „współpracy" – współdzielenia pracy (ang. *work–sharing*). W konstrukcjach „współpracy" następuje pewna synchronizacja – wszystkie wątki ze zbioru muszą osiągnąć ten punkt, zanim zostaną wykonane następujące po nim instrukcje – przez jeden lub więcej wątków należących do grupy. „Bariera" na końcu konstrukcji „współpracy" (bez klauzuli „nowait") jest wykonywana przez wszystkie wątki ze zbioru.

Ogólnie – krótka charakterystyka OpenMP (czym jest OpenMP?):

- zestaw dyrektyw kompilatora i pragm
- występuje watek master, który w zależności od potrzeb powołuje odpowiednią liczbę wątków slave (liczba wątków slave może być różna w różnych częściach programu)
- OpenMP wykorzystuje się głównie do zrównoleglenia pętli (np. przez dodanie #pragma omp parallel for przed wywołaniem klasycznej pętli for)
- działa w oparciu o model pamięci współdzielonej, wątki komunikują się poprzez współdzielone zmienne
- większość konstrukcji w OpenMP odnosi się do bloków strukturalnych (jedyne rozgałęzienia jakie mogą się pojawić to STOP w Fortranie i exit() w C/C++)

2 Konstrukcje OpenMP

Konstrukcje OpenMP można podzielić na 5 kategorii:

1. Regiony równoległe
2. Współdzielenie pracy
3. Środowisko danych
4. Synchronizacja
5. Funkcje i zmienne środowiskowe

2.1 Regiony równoległe

Wątki tworzymy pragmą „omp parallel", np. aby utworzyć 4-wątkowy region równoległy:

```
omp_set_num_threads (4);
#pragma omp parallel
{
      int ID = omp_thread_num();
     some_function(ID,A);
}
```

Każdy wątek wykonuje ten sam kod! Pojedyncza kopia zmiennej A jest współdzielona pomiędzy wszystkie wątki. Wątek po zakończeniu działania czeka aż pozostałe zakończą pracę (niejawnie jest wywoływana bariera). Występują dwa tryby: dynamiczny (domyślny) i statyczny. W dynamicznym trybie liczba wątków może być różna w różnych równoległych regionach, a ustawienie liczby wątków ustala ich maksymalną liczbę – może zostać wykorzystane mniej. W statycznym trybie liczba wątków jest stała i jest kontrolowana przez programistę. Regiony równoległe mogą być zagnieżdżane.

2.2 Współdzielenie pracy

Współdzielenie pracy za pomocą for dzieli iteracje pętli pomiędzy wątki:

```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++) {
                neat_stuff(i); }
```

Domyślnie na końcu omp for występuje bariera. Używając `nowait` wyłączamy barierę. Przykład zastosowania:

Kod sekwencyjny:

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

Region równoległy.

```
#pragma omp parallel
{
int id, i, Nthrds, istart, iend;
id = omp_get_thread_num();
Nthrds = omp_get_num_threads();
istart = id * N/Nthrds;
iend = (id+1) * N/Nthrds;
for(i=istart;i<iend;i++) { a[i] = a[i] + b[i]; }
}
```

Region równoległy oraz konstrukcja współpracy `for`:

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

Wystepuje tutaj m.in. dyrektywa schedule. Służy ona do sterowania rozdziałem iteracji pętli (wykonywanej w bloku równoległym) pomiędzy poszczególne wątki. Możliwe są 4 sposoby użycia dyrektywy "schedule".

1. schedule (static [,chunk])

   Rozdaje bloki po „chunk” iteracji pomiędzy wątkami.

2. schedule (dynamic [,chunk])

   Każdy blok otrzymuje po „chunk” iteracji po kolei, aż nie zostanie wyczerpana "kolejka" iteracji.

3. schedule (guided [,chunk])

   Wątki dynamicznie otrzymują „porcje” iteracji. Rozmiar bloku początkowo jest duży (bliżej nieokreślony) i stopniowo się zmniejsza do wartości "chunk", w miarę postępu obliczeń.

4. schedule (runtime)

Zastosowanie `#pragma omp section` (section work-sharing construct) umożliwia wykonywanie przez poszczególne wątki różnych bloków, np.:

```
#pragma omp parallel
#pragma omp sections
{
        x_calculation();
#pragma omp section
        y_calculation();
#pragma omp section
        z_calculation();
}
```

Podpowiedź:

```
#pragma omp parallel
#pragma omp for
```

Można zastąpić

```
#pragma omp parallel for
```

Istnieje również konstrukcja parallel sections.

2.3 Środowisko danych

Domyślnie większość zmiennych jest współdzielona. Zmienne globalne są współdzielone pomiędzy poszczególne wątki. Zmienne lokalne i automatyczne są prywatne dla każdego wątku.

Uwaga! Wszystkie poniższe konstrukcje odnoszą się do konstrukcji parallel regions i worksharing poza „shared", który odnosi się tylko do konstrukcji parallel regions.

Można zmienić domyślne zachowanie przez zastosowanie poniższych konstrukcji:

`shared(var)` – pojedyncza zmienna jest współdzielona pomiędzy wszystkie wątki

`private(var)` – tworzy lokalną kopie zmiennej dla każdego wątku (uwaga: zmienna nie jest inicjowana żadną wartością - uninitialized; kopia nie jest skojarzona z wartością oryginalną)

`firstprivate(var)` – podobne do powyższego, z tą różnicą, że każda kopia zostaje zainicjowana wartością zmiennej globalnej z głównego wątku sprzed wykonywania części równoległej (w dalszym ciągu nie ma skojarzenia z wartością oryginalną)

`lastprivate(var)` – każda kopia zostaje zainicjalizowana, a zmienna globalna przyjmuje na końcu wartość zmiennej prywatnej z ostatniej iteracji pętli

Domyślne zachowanie może być zmienione poprzez zastosowanie default (private | shared |none) Przy wybraniu default(none) trzeba dla każdej zmiennej wyspecyfikować sposób przechowywania w pamięci. Zauważ, że domyślnie jest przyjęte default(shared), więc nie trzeba tego specyfikować.

Tylko Fortran API wspiera default(private) C/C++ wspiera tylko default(shared) albo default(none).

Kolejną klauzulą, która wykorzystuje współdzielenie zmiennych jest redukcja: reduction (op: list) Zmienne zawarte w list muszą być współdzielone w następującym regionie równoległym. Wewnątrz regionu równoległego: tworzona jest lokalna kopia każdej zmiennej z list i jest inicjowana w zależności od operatora (np. 0 dla "+") wartość jest obliczana dla lokalnych wartości i na końcu lokalne wartości są redukowane do jednej globalnej kopii np.

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
        int i;
        double ZZ, func(), res=0.0;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:res) private(ZZ)
        for(i=0;i<1000;i++)
        {
                ZZ = func(i);
                res = res + ZZ;
        }
}
```

2.4 Synchronizacja

Występują następujące konstrukcje wspierające synchronizację:

`atomic` – jest specjalnym przypadkiem sekcji krytycznej, która może być używana w niektórych prostych wyrażeniach; ma zastosowanie tylko w przypadku uaktualnienia pamięci (np. x = x + b;)

`critical section` – bez komentarza (domyślnie jest bariera na końcu - nie można jej wyłączyć)

`barrier` – barierę można wywołać jawnie (#pragma omp barier); domyślnie jest na końcu pętli for (można zrezygnować z bariery poprzez użycie nowait (#pragma omp for nowait)

`flush` – jest stosowana do oznaczania sekwencji konkretnych "punktów" w programie, w których wątek próbuje stworzyć (wymusić) spójny stan pamięci.

- Wszystkie operacje na pamięcy (zarówno zapis, jak i odczyt) zdefiniowane przed sekwencją "flush" muszą zostać zakończone.
- Wszystkie operacje na pamięci (zarówno zapis, jak i odczyt) zdefiniowane po sekwencji "flush" mogą wykonać się dopiero po jej zakończeniu.
- Zmienne przechowywane w rejestrach lub buforach zapisu muszą być uaktualnione w pamięci.

Jako argument dla „flush" podaje się, które zmienne mają być „uspójnione". Jeśli nie podamy argumentów, wówczas system spowoduje „uspójnienie" wszystkich zmiennych.

`ordered` – wymusza sekwencyjne wykonanie pewnych części bloku np.:

```
#pragma omp parallel private(tmp)
#pragma omp for ordered
for(i=0;i<1000;i++) {
tmp = NEAT_STUFF(i);
#pragma ordered
res = consum(tmp);
}
```

`single` – konstrukcja single wydziela kawałek kodu jaki ma być wyliczony tylko przez jeden wątek (jawnie na końcu bloku single jest wywoływana bariera).

`master` – konstrukcja master oznacza kawałek bloku strukturalnego do wykonania tylko prze wątek master; pozostałe wątku pomijają ten fragment (Uwaga: nie jest wywoływana bariera - pozostałe wątki nie czekają na mastera; trzeba jawnie wywołać barierę – jeśli jest potrzebna).

Dwa ostatnie tak naprawdę nie są konstrukcjami służącymi do synchronizacji. Są konstrukcjami zapewniającymi współdzielenie pracy (work sharing) zawierającymi synchronizację.

2.5 Funkcje i zmienne środowiskowe

Funkcje i zmienne środowiskowe zostały opisane bardziej szczegółowo w załączonych dokumentach, do których odsyłamy po konkretne informacje.

3 Propozycje pytań

1. Scharakteryzuj krótko system OpenMP.
2. Do czego służy dyrektywa "schedule" w OpenMP? Podaj przykłady zastosowania.
3. Opisz konstrukcje "flush" stosowaną w synchronizacji procesów w OpenMP.