

Programowanie współbieżne

OpenMP – wybrane zagadnienia

Rafał Skinderowicz

OPENMP – NISKOPOZIOMOWA SYNCHRONIZACJA

- OpenMP udostępnia mechanizm zamków (lock) znany z typowych bibliotek programowania współbieżnego
 - zamki proste, metody: `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`
 - zamki zagnieżdżone (ang. nested) – pozwala na wielokrotne pozyskiwanie zamka przez ten sam wątek (podobnie do klasy `ReentrantLock` z Javy), metody: `omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`, `omp_test_nest_lock()`, `omp_destroy_nest_lock()`
- Każdorazowe pozyskanie (zamknięcie) bądź zwolnienie zamka powoduje wykonanie synchronizacji zmiennych lokalnych wątku z pamięcią główną

NISKOPOZIOMOWA SYNCHRONIZACJA – PRZYKŁAD

```
1 omp_lock_t lck;
2 omp_init_lock(&lck);
3 #pragma omp parallel private (tmp, id)
4 {
5     id = omp_get_thread_num();
6     tmp = obliczenia(id);
7     omp_set_lock(&lck); // czekaj na swoją kolej
8     printf("\%d \%d", id, tmp);
9     omp_unset_lock(&lck); // zwolnij lck
10 }
11 omp_destroy_lock(&lck);
```

- Jak widać zmienna lck jest współdzielona (shared), jednak nie wymaga dodatkowego odświeżania za pomocą dyrektywy `#pragma omp flush`

OPENMP – FUNKCJE BIBLIOTECZNE

Najczęściej stosowane funkcje biblioteczne OpenMP dotyczą identyfikacji wątku oraz pobierania i ustalania liczby wątków

```
1 #include <omp.h>
2 void main() {
3     int num_threads;
4     // wyłącza automatyczne ustalanie l. wątków:
5     omp_set_dynamic( 0 );
6     // zmiana liczby wątków na równą liczbie procesorów:
7     omp_set_num_threads( omp_num_procs() );
8     #pragma omp parallel shared(num_threads)
9     {
10         int id = omp_get_thread_num();
11         #pragma omp single
12             // pobranie rzeczywistej liczby wątków
13             num_threads = omp_get_num_threads();
14         obliczenia(id);
15     }
16 }
```

STEROWANIE PODZIAŁEM PRACY W PĘTLACH

- Jeżeli czasy wykonania obliczeń w kolejnych iteracjach pętli for **różnią się** istotnie, to być może równy przydział liczby iteracji dla każdego procesora może nie być najlepszy
- OpenMP udostępnia mechanizmy sterowania przydziałem poszczególnych zakresów indeksów w pętli for dla poszczególnych procesorów za pomocą klauzuli **schedule**

```
1 #pragma omp parallel for
2 for (int i = 0; i < N; ++i) {
3     int id = omp_get_thread_num();
4     obliczenia(id); // w zależności od id różny czas działania
5 }
```

STEROWANIE PODZIAŁEM PRACY W PĘTLACH

Warianty klauzuli schedule:

- `schedule(static [, porcja])` – podzieli liczbę iteracji na porcje o takiej samej lub podanej wielkości i przydzieli je „**z góry**” wszystkim wątkom
- `schedule(dynamic [, porcja])` – podobnie jak dla static, ale przydział porcji odbywa się w sposób dynamiczny – gdy wątek zakończy obliczenia dla jednej porcji, to sięga po kolejną dostępną
- `schedule(guided [, porcja])` – przydział następuje w sposób dynamiczny, ale rozmiary kolejnych porcji ulegają zmniejszeniu; minimalny rozmiar porcji określa parametr `porcja`
- `schedule(runtime)` – wariant podziału odczytywany jest ze zmiennej środowiskowej `OMP_SCHEDULE` w czasie działania programu

STEROWANIE PODZIAŁEM PRACY – PRZYKŁAD

Założmy, że liczba iteracji wynosi 16, a liczba wątków 4, wtedy podział iteracji wygląda następująco

porcja	id wątku			
	0	1	2	3
domyślnie	1-4	5-8	9-12	13-16
porcja = 2	1-2	3-4	5-6	7-8
	9-10	11-12	13-14	15-16

OPENMP – ZADANIA (TASKS)

- W wersji 3.0 OpenMP wprowadzono mechanizm zadań (ang. tasks) znacznie upraszczający zrównoleglanie programów, w których nie można w łatwy sposób zastosować `#pragma omp parallel for`
- Każde zadanie opisane jest przez **instancję zadania** składającą się z:
 - kodu do wykonania
 - powiązanych zmiennych inicjowanych w momencie tworzenia instancji zadania
 - wewnętrznych zmiennych kontrolnych
- Wątek, który napotka na blok zadania tworzy nową instancję zawierającą kod oraz dane do wykonania
- Jeden z grupy wątków wykonuje obliczenia
- Wykonanie zadania, którego definicja zawarta jest w instancji zadania może być **odłożone w czasie**

OPENMP – ZADANIA (TASKS)

- Zadania stanowią uogólnienie różnych konstrukcji równoległych dostępnych we wcześniejszych wersjach OpenMP
 - dyrektywa `parallel` tworzy zbiór *niejawnych* zadań, po jednym dla wątku
 - każdy wątek w grupie otrzymuje porcję obliczeń do wykonania
- Każda część programu OpenMP jest częścią jakiegoś zadania – sekwencyjna stanowi zadanie dla wątku głównego
- Zadania mogą być swobodnie **zagnieżdżane** wewnątrz innych zadań, bloków równoległych

DYREKTYWA TASK

```
1 #pragma omp task [klauzula [ [,]klauzula] ...]  
2     blok
```

gdzie *klauzula* może przyjmować wartość:

```
1 if (expression)  
2 untied  
3 shared (list)  
4 private (list)  
5 firstprivate (list)  
6 default( shared | none )
```

KLAUZULA IF DLA ZADAŃ

- Jeżeli wartość wyrażenia jest nieprawdziwa (fałsz) to:
 - kod zadania (task) jest wykonywany od razu, bez generowania instancji zadania do późniejszego wykonania
 - zmienne lokalne dla zadania są tworzone podobnie, jak w przypadku generowania instancji zadania
- Klauzula `if` ma na celu umożliwienie optymalizacji tworzenia zadań – odrębna instancja jest tworzona jedynie wtedy, gdy koszt wykonania zadania znacząco przekracza koszt utworzenia nowej instancji zadania

ZADANIA – PRZYKŁAD 1.

Zadania można stosować do wygodnego zrównoleglania pętli o elastycznej strukturze, np. przeglądającej listę elementów.

```
1 #pragma omp parallel
2 {
3     #pragma omp single private(p)
4     {
5         p = listhead ;
6         while (p) {
7             #pragma omp task
8             // zmienna p jest firstprivate wewnątrz zadania
9             process (p)
10            p=next (p) ;
11        }
12    }
13 }
```

Uwaga, zmienne domyślnie są jako firstprivate wewnątrz zadań

ZADANIA – PRZYKŁAD 2.

Zadania można również stosować do łatwego zrównoleglenia funkcji rekurencyjnych, np. przeglądania drzewa:

```
1 void postorder(node *p) {
2     if (p->left)
3         #pragma omp task
4         postorder(p->left);
5     if (p->right)
6         #pragma omp task
7         postorder(p->right);
8     // czekaj na zakończenie wyżej utworzonych zadań
9     #pragma omp taskwait
10    process(p->data);
11 }
12 ...
13 #pragma omp parallel
14 #pragma omp single
15 postorder(tree_root);
```

Zadanie–rodzic kończy się po zakończeniu wszystkich zadań potomnych.

ZADANIA – PRZYKŁAD 2. WERSJA 2.

Na szczęście obliczenia rekurencyjne można również zrównoleglić bez zadań

```
1 void postorder(node *p) {
2     #pragma omp parallel sections
3     {
4         if (p->left)
5             #pragma omp section
6             postorder(p->left);
7         if (p->right)
8             #pragma omp section
9             postorder(p->right);
10    } // <- niejawna bariera synchronizacyjna
11    process(p->data);
12 }
13 ...
14 postorder(tree_root);
```

ZADANIA A SEKCJE

- Zadania (ang. tasks) i sekcje (ang. sections) są do siebie podobne, jednak istnieje między nimi kilka różnic
- Sekcje definiowane są wewnątrz bloku sections i żaden z wątków roboczych nie opuści bloku dopóki wszystkie sekcje nie zostaną zakończone

```
1  [   Sekcje (sections)                               ]
2  Thread 0: -----< section 1 >---->|-----
3  Thread 1: -----< section 2.....>|-----
4  Thread 2: ----->|-----
5  .....|
6  Thread N-1: ----->|-----
7                                     ^- bariera synchronizacyjna
```

- N wątków napotyka blok sections zawierający jedynie dwie sekcje
- Dwa wątki wykonują sekcje
- Pozostałe N-2 wątków musi czekać

ZADANIA A SEKCJE

- Zadania (tasks) są **kolejkowane** i wykonywane we wskazanych punktach synchronizacji
- Dodatkowo dopuszczalna jest **migracja** zadań między wątkami roboczymi, tzn. może się zdarzyć, że zadanie w trakcie wykonywania zostanie przeniesione np. z wątku 1 na 2

ZADANIA A SEKCJE

```
1 // sections
2 ...
3 #pragma omp sections
4 {
5     #pragma omp section
6     foo();
7     #pragma omp section
8     bar();
9 }
```

```
1 // tasks
2 ...
3 #pragma omp single nowait
4 {
5     #pragma omp task
6     foo();
7     #pragma omp task
8     bar();
9 }
10 #pragma omp taskwait
```

- taskwait działa podobnie do barrier – powoduje, że wątki robocze zanim przejdą dalej muszą najpierw wykonać wszystkie zakolejkowane zadania
- dyrektywa single powoduje, że zadania zostaną wygenerowane tylko przez jeden wątek roboczy

ZADANIA A SEKCJE

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

- W przykładzie zadania tworzone są przez pierwszy wątek, pozostałe dwa dzięki klauzuli `nowait` kontynuują obliczenia do napotkania klauzuli `taskwait`
- Wtedy pobierają zadania z kolejki i je wykonują
- Jeżeli nie ma jawnie określonych punktów synchronizacji, to OpenMP może wykonywać zadania w dowolnym momencie wewnątrz bloku `parallel`

GENEROWANIE LICZB PSEUDOLOSOwych

W symulacjach typu Monte Carlo, czy algorytmach heurystycznych niezbędne jest korzystanie z generatorów liczb pseudolosowych.

- Generator liczb pseudolosowych ma **stan** na podstawie którego wyznaczana jest kolejna wartość
- Zrównoleglając program tego typu należy o tym pamiętać – chcemy żeby każdy wątek / procesor miał **osobny** ciąg liczb losowych
- Funkcja `rand()` z biblioteki standardowej języka C nie nadaje się, ponieważ korzysta z globalnego stanu – współdzielonego przez wszystkie wątki
- Rozwiązanie – zewnętrzny generator

GENEROWANIE LICZB PSEUDOLOSOWYCH – PRZYKŁAD

Prosty liniowy generator kongruentny (ang. linear congruential generator)

```
1 static unsigned long MULTIPLIER = 1366;
2 static unsigned long ADDEND = 150889;
3 static unsigned long PMOD = 714025;
4 unsigned long random_last = 0;
5 double random () {
6     unsigned long random_next;
7     random_next = (MULTIPLIER * random_last + ADDEND) % PMOD;
8     random_last = random_next;
9     return ((double)random_next/(double)PMOD);
10 }
11 void init_random(long seed) {
12     random_last = seed;
13 }
```

GENEROWANIE LICZB PSEUDOLOSOWYCH – PRZYKŁAD

Każdy wątek musi mieć własny stan generatora

```
1 static unsigned long MULTIPLIER = 1366;
2 static unsigned long ADDEND = 150889;
3 static unsigned long PMOD = 714025;
4 unsigned long random_last = 0;
5 #pragma omp threadprivate(random_last)
6 double random () {
7     unsigned long random_next;
8     random_next = (MULTIPLIER * random_last + ADDEND) % PMOD;
9     random_last = random_next;
10    return ((double)random_next/((double)PMOD));
11 }
```

GENEROWANIE LICZB PSEUDOLOSOWYCH – PRZYKŁAD

Każdy wątek musi mieć inny stan początkowy (ziarno). Prymitywna metoda:

```
1 void init_random_par()  
2 {  
3     int seed = time(NULL);  
4     #pragma omp parallel  
5     {  
6         int id = omp_get_thread_num();  
7         init_random(seed + id);  
8     }  
9 }
```

- Wada: generowane ciągi będą się nakładać, być może dość szybko – duża część generowanych liczb będzie taka sama
- Inne rozwiązania
 - zastosowanie generatora o bardzo dużym okresie
 - współdzielenie sekwencji przez wszystkie wątki – pewność, że każdy wątek ma unikalny podciąg liczb losowych

OPENMP – WSKAZÓWKI

- Zazwyczaj należy zrównoleglać najwyższy możliwy poziom kodu, np. pętlę zewnętrzną a nie wewnętrzne – pozwala uniknąć zbyt małych porcji obliczeń
- Stosować jak najmniejszą liczbę bloków równoległych (`#pragma omp parallel`):
 - w jednym bloku równoległym można umieścić wiele pętli
 - nie tworzyć bloków równoległych wewnątrz pętli – zazwyczaj można przenieść przed pętlę
- Stosować jak najmniejszą liczbę barier synchronizacyjnych

OPENMP – WSKAZÓWKI C.D.

- Rozmiar sekcji krytycznych powinien być tak mały, jak to możliwe
- Nie stosować klauzuli `ordered`, chyba że jest konieczna
- Usuwać niezamierzone współdzielenie (ang. false sharing) zmiennych między wątkami (o tym w dalszej części)
 - lepiej stosować zmienne prywatne
- Eksperymentować z różnymi sposobami podziału iteracji w pętlach i różnymi wielkościami porcji, np.
`schedule(dynamic, 10)`
- Klauzula `nowait` może być pomocna w wyeliminowaniu zbędnej synchronizacji, tam gdzie nie jest ona potrzebna

PAMIĘĆ PODRĘCZNA OGÓLNIE

- Sposób dostępu do pamięci przez program ma istotne znaczenie dla jego wydajności
- Dostęp do pamięci głównej (odczyt, zapis) wymaga znacznie dłuższego czasu, niż czas wykonania pojedynczego rozkazu przez CPU
- Oczywiście, z pomocą przychodzi pamięć podręczna – dane często używane powinny zostać przepisane do pamięci podręcznej, żeby procesor nie musiał czekać na dane potrzebne do wykonania kolejnego rozkazu
- Problem: dane muszą być przepisywane do pamięci podręcznej **z wyprzedzeniem**

LOKALNOŚĆ ODNIESIEŃ

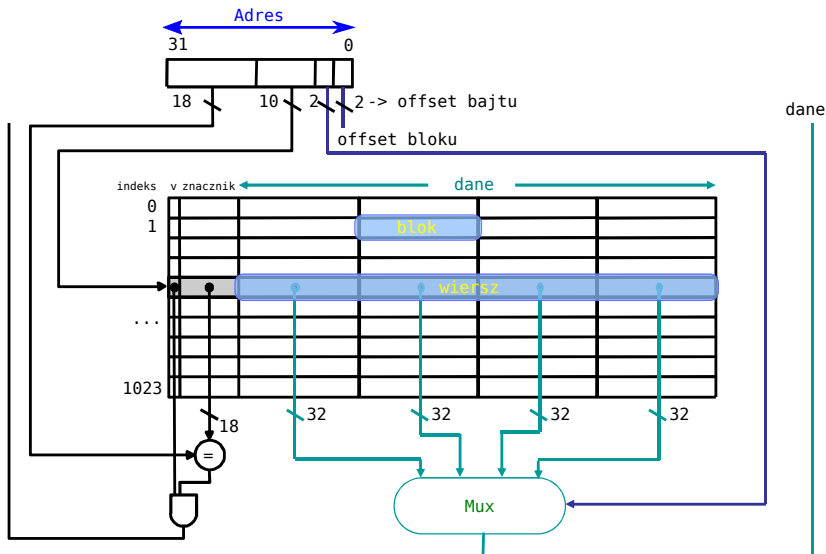
- CPU potrzebuje zarówno **kodu rozkazu** do wykonania, jak i **danych** dla tego rozkazu
- Zazwyczaj w programach występuje zjawisko **lokalności odniesień** (ang. locality)
 - **I. czasowa** – odniesienia powtarzające się w czasie
 - **I. przestrzenna** – jeżeli nastąpiło odniesienie do konkretnego adresu, to prawdopodobnie nastąpi do sąsiedniego
- Lokalność instrukcji w kodzie programu wynika z ich sekwencyjnego charakteru
 - Zaburzeniem lokalności są rozkazy skoków, np. dla instrukcji warunkowych, wywołań procedur
- Największy wpływ mamy na lokalność w dostępie do danych

HIERARCHIA PAMIĘCI PODRĘCZNYCH

- Jeżeli dana / kod rozkazu potrzebny CPU jest dostępny w pamięci podręcznej, to mówimy o **trafieniu** (ang. cache hit), w przeciwnym razie o **chybieniu** (ang. cache miss)
- Pamięć podręczna dzieli się na poziomy:
 - **L-1** – najszybsza, typowy rozmiar 64KB = 32KB instrukcje + 32KB dane
 - **L-2** – wolniejsza od L-1, ale większy rozmiar – np. 256KB / rdzeń w Core i5
 - **L-3** – nie zawsze występuje; Core i5 – rozmiar 4MB (2 rdzenie) lub 8MB (4 rdzenie) – współdzielona

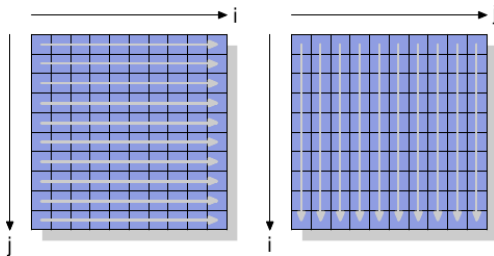
PAMIĘĆ PODRĘCZNA – SCHEMAT

Pam. jednoznaczna, 16KB ($1024 \times 4 \times 4B$)



PAMIĘĆ CACHE A WYDAJNOŚĆ

- Sposób dostępu do danych w programie ma istotny wpływ na wydajność programu – należy, o ile to możliwe, zapewnić lokalność w dostępie do danych
- Przykładowo, przy przetwarzaniu macierzy o dużych rozmiarach, lepiej odwoływać się do komórek w kolejności pokazanej na rys. po lewej



PRZYKŁAD – MNOŻENIE MACIERZY

Założmy, że należy pomnożyć dwie macierze, A i B, o rozmiarach $N \times N$ ($N=1000$). Wynikiem jest macierz C, które elementy dane są następująco:

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} b_{k,j}$$

Prosta implementacja:

```
1 for (int i = 0; i < N; ++i)
2     for (int j = 0; j < N; ++j)
3         for (int k = 0; k < N; ++k)
4             C[i][j] += A[i][k] * B[k][j];
```

PRZYKŁAD – MNOŻENIE MACIERZY

Problem – o ile dostęp do kolejnych komórek macierzy A jest wg wierszy, o tyle dostęp do B jest wg kolumn – brak lokalności odwołań do B

```
1 for (int i = 0; i < N; ++i)
2   for (int j = 0; j < N; ++j)
3     for (int k = 0; k < N; ++k)
4       C[i][j] += A[i][k] * B[k][j];
```

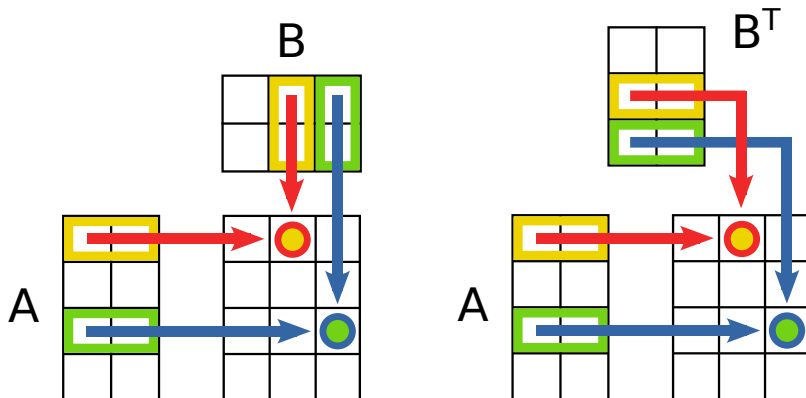
Rozwiązanie – mnożenie przez transponowaną macierz B:

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} b_{j,k}^T$$

zamiast

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} b_{k,j}$$

PRZYKŁAD – MNOŻENIE MACIERZY



PRZYKŁAD – ZMODYFIKOWANA WERSJA

Implementacja uwzględniająca transpozycję

```
1 int **BT = create_matrix(N, N);
2 // Transpozycja B
3 for (int i = 0; i < N; ++i)
4     for (int j = 0; j < N; ++j)
5         BT[i][j] = B[j][i];
6 // Właściwe mnożenie
7 for (int i = 0; i < N; ++i)
8     for (int j = 0; j < N; ++j)
9         for (int k = 0; k < N; ++k)
10            C[i][j] += A[i][k] * BT[j][k];
```

PORÓWNANIE

- Rozmiar macierzy 1000×1000 ($N = 1000$)
- Procesor Intel Core i5 2.27GHz 3MB Cache L2
- Kompilator: g++ 4.6.1
- Przełączniki: -O2 -funroll-loops

PORÓWNANIE

- Rozmiar macierzy 1000×1000 ($N = 1000$)
- Procesor Intel Core i5 2.27GHz 3MB Cache L2
- Kompilator: g++ 4.6.1
- Przełączniki: -O2 -funroll-loops

Wyniki:

- Wersja oryginalna: 12.78 sek.
- Wersja zmodyfikowana: 1.25 sek.
- Przyspieszenie: $12.78/1.25 \approx 10$ razy

PAMIĘĆ CACHE A PROGRAMY RÓWNOLEGŁE

- W przypadku wykonania programu (w modelu ze współdzieloną pamięcią) na wielu procesorach konieczne jest zapewnienie **spójności** między pamięciami podręcznymi procesorów oraz pamięcią główną
- Za zachowanie spójności pamięci odpowiedzialne są kontrolery pamięci podręcznej działające wg ustalonego protokołu
- Jednym z najczęściej stosowanych jest protokół MESI

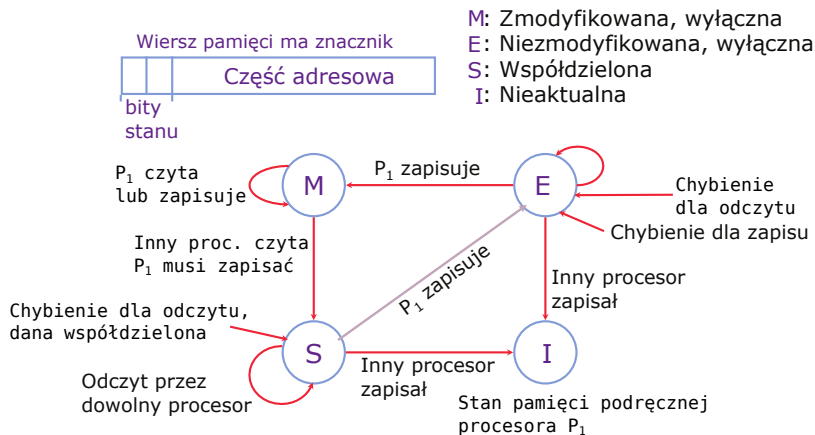
MESI

Z każdym wierszem w pamięci podręcznej związany jest jeden z 4 atrybutów (2 bity):

- Exclusive – niezmodyfikowana kopia znajdująca się tylko w jednej pamięci podręcznej
- Modified – zmodyfikowana kopia znajdująca się tylko w jednej pamięci podręcznej
- Shared – niezmodyfikowana kopia znajdująca się w kilku pamięciach
- Invalid – nieaktualna kopia – każdy kolejny dostęp do danej wymaga pobrania z pamięci głównej

Kontrolery pamięci podręcznych muszą „nasłuchiwać” zmian stanów dla przechowywanych wierszy danych

MESI



PAMIĘĆ CACHE A WYDAJNOŚĆ PROGRAMÓW RÓWNOLEGLYCH

Wydażność pamięci podręcznej zależy od:

- Liczby chybień do liczby żądań dla pojedynczego procesora
- Dodatkowego narzutu komunikacyjnego związanego z koniecznością zachowania spójności danych w pamięciach podręcznych różnych procesorów
- Chybień wynikające ze spójności (ang. coherency misses)
 - 1 **Prawdziwe** (ang. true sharing misses)
 - Unieważnienie współdzielonej (shared) kopii ze względu na zapis
 - Następnie odczyt tej danej w stanie zmodyfikowanym (modified) przez inny procesor
 - 2 **Fałszywe** inaczej niezamierzone (ang. false sharing misses)
 - Wynikające z rozmieszczenia zmiennych w pamięci operacyjnej i szerokości wiersza pamięci podręcznej

FALSE SHARING

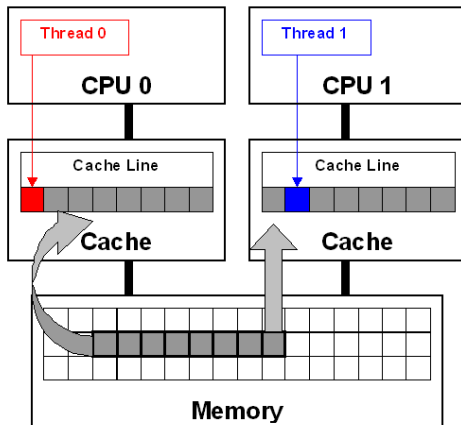
- Najmniejszą jednostką pamięci podręcznej jest jeden wiersz (zazwyczaj 64B), co może prowadzić do zjawiska **niezamierzonego współdzielenia pamięci** (ang. false sharing)

Przykład:

```
1 double sum=0.0, sum_local[NUM_THREADS];
2 #pragma omp parallel num_threads(NUM_THREADS)
3 {
4     int me = omp_get_thread_num();
5     sum_local[me] = 0.0;
6
7     #pragma omp for
8     for (i = 0; i < N; i++)
9         sum_local[me] += x[i] * y[i];
10
11     #pragma omp atomic
12     sum += sum_local[me];
13 }
```


FALSE SHARING

Zjawisko niezamierzonego współdzielenia występuje, gdy wątki modyfikują jednocześnie różne zmienne, ale znajdujące się blisko siebie – w ramach pojedynczego wiersza pamięci cache



FALSE SHARING

- Zjawisko niezamierzonego współdzielenia może istotnie wpłynąć na wydajność programu, dlatego należy je eliminować:
 - ręcznie:
 - korzystanie ze zmiennych prywatnych w wątkach
 - rozmieszczanie (ang. alignment) zmiennych odpowiednio „daleko” od siebie, by nie mieściły się w jednym wierszu cache
 - automatycznie – kompilatory w ramach optymalizacji starają się wykrywać i eliminować to zjawisko
- Wykrywanie za pomocą zewnętrznych narzędzi, jak np. Intel VTune przez analizę miejsc w pamięci, dla których następuje duża liczba chybień

FALSE SHARING

Przykład z wyeliminowanym niezamierzonym współdzieleniem

```
1 double sum=0.0;
2 double sum_local;
3 #pragma omp parallel num_threads(NUM_THREADS) private(sum_local)
4 {
5     int me = omp_get_thread_num();
6     sum_local = 0.0;
7
8     #pragma omp for
9     for (i = 0; i < N; i++)
10         sum_local += x[i] * y[i];
11
12     #pragma omp atomic
13     sum += sum_local;
14 }
```

FALSE SHARING – PRZYKŁAD 2.

Przykład opisany na stronie

<http://www.pgroup.com/lit/articles/insider/v2n2a4.htm>

Generowanie liczb pseudolosowych za pomocą alg. Mersenne Twister wymaga, oczywiście, przechowywania stanu generatora. Stan dla pojedynczego wątku:

```
1 static uint32_t      state[MT_NN];
```

Stan dla maks. 32 wątków:

```
1 static uint32_t      state[32][MT_NN];
```

Wyniki dla 4 rdzeniowego procesora Intel Core i7 (2.66 GHz):

```
1 CPU Mersenne Twister with OpenMP (1 threads)
2 exec time   = 1.327572 s
3 throughput  = 404.400599 [MB/s]
4 CPU Mersenne Twister with OpenMP (4 threads)
5 exec time   = 0.909803 s
6 throughput  = 590.095781 [MB/s]
```

FALSE SHARING – PRZYKŁAD 2.

Przyczyną kiepskiej wydajności było właśnie niezamierzone współdzielenie stanów generatora przez wątki. Usunięcie polegało na rozszerzeniu stanu, tak by przekroczył długość wiersza pamięci podręcznej

```
1 static uint32_t          state[32][MT_NN*16];
```

Wynik:

```
1 CPU Mersenne Twister with OpenMP (1 threads)
2 exec time   = 1.327572 s
3 throughput  = 404.400599 [MB/s]
```

```
1 CPU Mersenne Twister with OpenMP (4 threads)
2 exec time   = 0.349923 s
3 throughput  = 1534.254427 [MB/s]
```