Współbieżność jest nieintuicyjna

*Tomasz IDZIASZEK**

* Redaktor *Delty* w latach 2010–2016, współautor rubryki „Informatyczny Kącik Olimpijski”

O tym, jak różne rozwiązania techniczne stosowane w komputerach mają wpływ na wydajność uruchamianych na nich programów, pisaliśmy na łamach *Delty* niejednokrotnie.

Pierwsze komputery miały pojedynczy procesor. Można było zakładać, że każda z jego podstawowych operacji (tj. wczytanie z pamięci, wykonanie jednostki obliczeń i zapis do pamięci) trwa podobną ilość czasu.

O przetwarzaniu potokowym pisaliśmy w artykule *O sierotce, co chciała się mózgiem elektronowym wyręczyć*, Δ^{8}_{12}.

W kolejnych dekadach inżynierowie konstruowali coraz bardziej skomplikowane procesory: tranzystory były mniejsze, gęściej upakowane, co pozwalało na implementację wydajniejszych mechanizmów umożliwiających wykonywanie wielu instrukcji jednocześnie (np. przetwarzanie potokowe).

O szczegółach komputerowej pamięci można przeczytać w artykułach: *Pamięć cache w praktyce*, Δ^{10}_{09}, *Kolejność ma znaczenie*, Δ^{9}_{11}, *Pamięć w komputerze*, Δ^{5}_{16}.

Potrzebowano również coraz większych pamięci przechowujących dane w komputerze. Problem w tym, że duże pamięci są albo wolne, albo drogie. Poradzono sobie z nim, tworząc tzw. hierarchię pamięci: procesor ma dostęp do pamięci podręcznej (kilka poziomów *cache*), pamięci głównej (RAM), a w końcu pamięci masowej (dyski twarde).

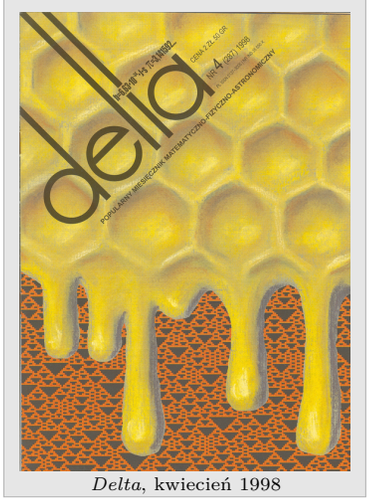
Delta, listopad 1997

Proste modele są pożądane, bo łatwo analizować, co się w nich dzieje. Model procesora typu „wykonuję po kolei pojedyncze instrukcje, kończąc wykonanie instrukcji, zanim zaczną następną” jest intuicyjny dla programisty i wygodny, jeśli chcemy formalnie udowodnić poprawność działania naszego programu. Jednak chęć wprowadzenia do modelu dodatkowych mechanizmów mających na celu zwiększenie wydajności systemu powoduje, że musimy dobrze przemyśleć, jak te mechanizmy będą współdziałały z modelem i z pozostałymi dodatkami. Za przykład niech posłużą tutaj dwa mechanizmy: zmiana kolejności instrukcji (*reordering*) oraz efektywne korzystanie z pamięci podręcznej.

Zmiana kolejności może być wykonana przez kompilator, który tłumaczy program z języka wysokiego poziomu na kod maszynowy, a następnie optymalizuje ten kod. Przykładowo w kodzie `a=c+1; b=1` kompilator może zamienić miejscami te dwie instrukcje, gdyż operują one na niezależnych miejscach w pamięci.

Ale ta zmiana może być również wykonana przez procesor, w ramach wykonywania instrukcji poza kolejnością (*out-of-order execution*). Procesor może, czekając na dostęp do pamięci dla zmiennej `c`, najpierw wykonać operację `b=1`, dla której ma wszystkie dane.

Podstawową zasadą jest tu, że zmiany te muszą być niewidoczne dla programisty. A konkretnie: wynik uruchomienia *jednowątkowego* programu przy zmianach kolejności musi być taki sam jak oryginalnego. Kluczowe jest tutaj założenie o jednowątkowości, tzn. program jest uruchamiany na jednym procesorze.



Delta, kwiecień 1998

Idea hierarchii pamięci jest taka, że jeśli potrzebujemy wczytać dane, to najpierw szukamy ich w mniejszej i szybszej **pamięci podręcznej**. Dopiero, jeśli ich tam nie znajdziemy, to czytamy je z pamięci głównej i zapisujemy również do pamięci podręcznej (aby kolejny odczyt był szybszy; przy okazji zapisujemy też trochę sąsiednich komórek pamięci, zakładając lokalność odwołań).

Działa to całkiem efektywnie w przypadku *odczytu* z pamięci. A co w przypadku *zapisu*? Możemy założyć, że zapisu dokonujemy jedynie w pamięci podręcznej, a jeśli będziemy musieli jakieś dane z niej usunąć (bo brakuje nam miejsca), to wtedy dopiero zostaną one zapisane również w pamięci głównej (o ile były modyfikowane). Ten sposób „leniwego zapisu” ma swoją konsekwencję taką, że pamięć główna niekoniecznie jest zsynchronizowana z pamięcią podręczną, przy czym znowu: dla programu jednowątkowego jest to niewidoczne.

Jeszcze jednym sposobem przyspieszenia pracy procesora jest stworzenie w nim dodatkowego bufora, w którym kolejkowane są operacje zapisu (*store buffer*).

Tak więc nawet pamięć podręczna nie musi być w pełni zsynchronizowana (każdy odczyt z pamięci najpierw sprawdza bufor zapisu, a dopiero potem pamięć podręczną).

O problemach, jakie niesie ze sobą wieloprocesorowość, wspominaliśmy w artykule *Programowanie na platformie CUDA*, Δ^{9}_{11}.

Upakowywanie coraz większej liczby tranzystorów ma swoje fizyczne granice. W związku z tym aktualną metodą na przyspieszenie komputerów jest instalowanie w nich **wielu procesorów** (lub procesorów o wielu rdzeniach). Idea jest taka, że można na nich niezależnie uruchamiać różne programy (czym automatycznie zajmuje się system operacyjny). Kłopot zaczyna się wtedy, kiedy chcemy wiele procesorów wykorzystać do przyspieszenia działania pojedynczego programu. Nie dostaniemy tego za darmo – nasz program musi być odpowiednio napisany, aby mógł być uruchomiony **współbieżnie**.

Problematyczna jest sytuacja, gdy wiele wątków (części naszego programu uruchamianych na różnych procesorach) próbuje uzyskać dostęp do tej samej komórki pamięci. Jeśli wszystkie z niej czytają – jest w porządku. Ale gdy choć jeden procesor do niej zapisuje, to możemy mieć do czynienia z wyścigiem (*race condition*). Zadaniem programisty jest zagwarantować, że taka sytuacja nigdy nie wystąpi.

Inną możliwością jest skorzystanie z wysokopoziomowych konstrukcji synchronizujących, tzw. muteksów (*mutex*).

Jednym ze sposobów jest użycie tzw. instrukcji atomowych (*atomic instructions*), które gwarantują na poziomie sprzętu, że na czas działania tej jednej instrukcji, procesor uzyskuje wyłączność na dostęp do komórki pamięci. Z wykorzystaniem takich instrukcji możemy zsynchronizować dostęp do pamięci, czego przykładem jest tzw. *message passing*:

```
T1: a=c+1          T2: while (ok != 1) { }
    ok=1               assert(a==c+1)
```

Mamy tu do czynienia z dwoma wątkami, T_1 i T_2, które są uruchomione na dwóch procesorach. Pierwszy wątek zapisuje dane do pamięci dzielonej (zmienna `a`), a następnie ustawia flagę, że już skończył, korzystając z instrukcji zapisu do atomowej zmiennej (`ok=1`). Drugi wątek czeka, dopóki flaga nie jest ustawiona, a następnie czyta dane.

W programie mamy dwa wątki i dwie zmienne dzielone, początkowo równe 0 (`x=y=0`):

```
T1: y=1      T2: x=1
    a=x          b=y
```

Gdyby model „przeplotu” był poprawny (tzn. prawdą byłoby, że wykonanie równoległe dwóch wątków odpowiada pewnemu przetasowaniu ich instrukcji), to instrukcja `b=y` wykonałaby się po instrukcji `y=1` albo instrukcja `a=x` po instrukcji `x=1`. To pokazuje, że na końcu działania programu co najmniej jedna ze zmiennych, `a` lub `b`, będzie równa 1.

W praktyce okazuje się jednak, że nawet jeśli kolejność instrukcji nie zostanie zmieniona ani przez kompilator, ani procesor, to nadal wynik `a=0, b=0` jest możliwy.
Jest to spowodowane buforem zapisu. Instrukcja `y=1` kolejkuje się w buforze; zatem zmienna `y` dla wątku T_1 ma wartość 1, a dla wątku T_2 wartość 0 (bo ani pamięć główna, ani pamięć podręczna wspólna dla procesorów nie została jeszcze zsynchronizowana).
Z symetrii zmienna `x` dla wątku T_2 ma wartość 1, a dla wątku T_1 wartość 0.
Zatem dwa wątki nie zgadzają się co do kolejności, w której następują przypisania do zmiennych `x` i `y`.

Powyższy program nie musi być niestety poprawny w przypadku agresywnych operacji optymalizacji kompilatora lub procesora. Co będzie bowiem, gdy kompilator zamieni kolejność operacji w T_1? Z perspektywy tego wątku nic się nie zmieni, ale program współbieżny zadziała nieprawidłowo: wątek T_2 uzyska dostęp do nie w pełni zainicjowanych danych. Również procesor może doprowadzić do błędu. Ponieważ zmienne `ok`, `a` i `c` są niezależne, może on w wątku T_2 wczytać zawartość zmiennej `a`, zanim jeszcze zakończy się pętla.

Zabezpieczenie się przed takimi błędami należy do programisty. Jednym z rozwiązań jest umieszczenie w kodzie programu **barier** (*memory barrier* lub *fence*), który informuje kompilator i procesor, że nie można zmieniać kolejności instrukcji, pomiędzy którymi znajduje się bariera:

```
T1: a=c+1          T2: while (ok != 1) { }
    mb()               mb()
    ok=1               assert(a==c+1)
```

Barier należy jednak używać rozważnie, bo ich nadpodaż ogranicza możliwe optymalizacje i powoduje spadek wydajności programu. To na programiście spoczywa trud zapewnienia minimalnej liczby barier potrzebnych do poprawnego działania programu, bez zbytniego nadwerężania jego wydajności. Jest to trudne zadanie, tym bardziej, że współbieżność wymaga od programisty wyrobienia sobie zupełnie innego rodzaju intuicji na temat programowania.

Przykładowo, typowa intuicja, że wynik działania programu wielowątkowego można uzyskać, rozważając pewien „przeplot” instrukcji uruchamianych wątków, nie jest w ogólności prawidłowa (patrz przykład na marginesie).