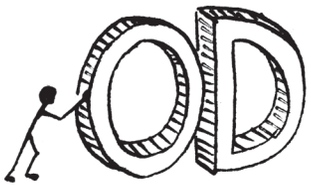
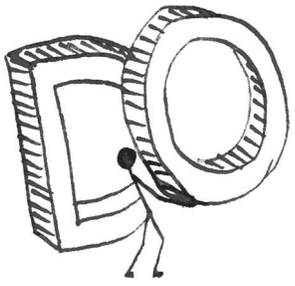
Kolejność ma znaczenie

Tomasz IDZIASZEK



Zakładamy, że Czytelnik zna podstawowe kwestie związane z pamięcią podręczną procesora (tzn. pamięcią *cache*). Można o nich przeczytać w artykule Krzysztofa Piecucha w *Delcie* 10/2009.



Z artykułu Wojciecha Śmietanki wypływa ważny morał: przystosowanie algorytmu do działania na maszynie równoległej wymaga często zupełnie innego spojrzenia na dany problem. Okazuje się jednak, że nawet w przypadku architektury jednoprocesorowej optymalizacja algorytmu może wymagać od nas całkiem pomysłowych przeróbek. W tym artykule podamy dwa przykłady, w których kluczową okaże się *kolejność*, w jakiej wykonujemy operacje.

Za pierwszy przykład niech posłuży wspomniany już algorytm mnożenia macierzy. Można go zapisać, na przykład, tak:

for $i := 1$ **to** n **do**
  for $k := 1$ **to** n **do**
    for $j := 1$ **to** n **do**
      $c[i,j] := c[i,j] + a[i,k] \cdot b[k,j]$;

Z uwagi na przemienność dodawania nie ma znaczenia, w jakiej kolejności wykonamy powyższe pętle. Ważne jest tylko to, aby ostatni wiersz został wykonany dla wszystkich trójek $(i,j,k) \in \{1,\ldots,n\}^3$. Okazuje się, że to, iż pętle występują w kolejności i,k,j (zamiast, wydawać by się mogło, bardziej naturalnej kolejności i,j,k), ma kluczowe znaczenie. Na moim komputerze ten program dla $n = 1000$ wykonuje się 2,1 s, natomiast program „naturalny" aż 8,2 s – prawie cztery razy wolniej! Wiąże się to z tym, że w powyższym programie wszystkie macierze są przeglądane wierszami, co dobrze wpływa na wykorzystanie pamięci podręcznej procesora.

Często, aby umożliwić wykonywanie operacji w lepszej kolejności, konieczne jest głębsze przebudowanie algorytmu. Za przykład posłuży nam tu wyznaczanie liczb pierwszych za pomocą sita Eratostenesa.

Będziemy wypełniać tablicę $p[2..n]$. Po zakończeniu algorytmu wartość $p[i]$ będzie wskazywała, czy liczba i jest pierwsza. Algorytm w pseudokodzie wygląda tak:

for $i := 2$ **to** n **do**
  $p[i] :=$ **true**;
for $i := 2$ **to** $\lfloor\sqrt{n}\rfloor$ **do**
  if $p[i]$ **then**
    $j := i^2$;
    while $j \leqslant n$ **do**
      $p[j] :=$ **false**;
      $j := j + i$;

Zauważmy, że dla każdej nowo znalezionej liczby pierwszej i przeglądamy prawie całą tablicę p, by wykreślić wielokrotności i. To powoduje, że nieefektywnie korzystamy z pamięci podręcznej procesora. Spróbujmy zatem tak przepisać powyższy algorytm, by zwiększyć lokalność odwołań do pamięci.

Zauważmy, że liczba pierwsza i zostanie użyta do wykreślania tylko, gdy $i \leqslant \sqrt{n}$. Na początek wykonajmy więc powyższy kod dla początkowego kawałka tablicy $p[2..\lfloor\sqrt{n}\rfloor]$, przy okazji zapamiętując w tablicy $pie[1..k]$ napotkane liczby pierwsze, a w tablicy $wiel[1..k]$ ich najmniejsze wielokrotności większe niż $\lfloor\sqrt{n}\rfloor$. Resztę tablicy podzielmy na bloki długości B i przeglądajmy te bloki kolejno, za każdym razem wykreślając (dla wszystkich $1 \leqslant i \leqslant k$) znajdujące się w tym bloku wielokrotności liczby $pie[i]$, odpowiednio uaktualniając w $wiel[i]$ najmniejszą niewykorzystaną jeszcze wielokrotność $pie[i]$. Poniższy pseudokod realizuje ten nowy algorytm:

for $i := 2$ **to** n **do**
  $p[i] :=$ **true**;
$k := 0$;
for $i := 2$ **to** $\lfloor\sqrt{n}\rfloor$ **do**
  if $p[i]$ **then**
    $k := k + 1$;
    $pie[k] := i$;
    $wiel[k] := i^2$;
    while $wiel[k] \leqslant \lfloor\sqrt{n}\rfloor$ **do**
      $p[wiel[k]] :=$ **false**;
      $wiel[k] := wiel[k] + i$;

$b := \lfloor\sqrt{n}\rfloor$; {$b$ to ostatni element poprzedniego bloku}
while $b < n$ **do**
  for $i := 1$ **to** k **do**
    while $wiel[i] \leqslant \min(b + B, n)$ **do**
      $p[wiel[i]] :=$ **false**;
      $wiel[i] := wiel[i] + pie[i]$;
  $b := b + B$;

Jeśli przyjmiemy $n = 10^8$, to pierwszy kod działa na moim komputerze 4,2 s. Pamięć podręczna w moim procesorze ma rozmiar 32 kB, zatem możemy przyjąć $B = 32\,000$. Wtedy drugi program działa w czasie 1,3 s, co daje ponadtrzykrotne przyspieszenie, mimo iż liczba operacji wykonywanych w pseudokodzie wzrosła!

Można, oczywiście, dalej optymalizować nasz algorytm, np. wyrzucić z tablicy p liczby parzyste i użyć tablicy bitowej. Można również przyjąć jako rozmiar bloku wielokrotność iloczynu małych liczb pierwszych i zauważyć, że wtedy wielokrotności tych liczb w każdym bloku są położone tak samo.