Przetwarzanie równoległe – Zadanie domowe II

Jarosław Marek Gliwiński, #indeksu 74839

Założenia wstępne

Należy zaznaczyć, że należący do danych zadania kod źródłowy poddałem pewnym modyfikacjom już na etapie poprzedzającym analizę. Mianowicie dostrzegłem dwa zapisy, które można uznać za co najmniej specyficzne, a z różnym prawdopodobieństwem błędne.

1. Układ wcięć sugeruje, że obydwie pętle są zagnieżdżone (indeksowana „po j" wewnątrz „po i", co jest niespójne z brakiem klamr zewnętrznej pętli. Założyłem, że jest to zapis błędny i analizy dokonałem dla przypadku pętli zagnieżdżonych („z klamrami")

2. Zewnętrzna pętla jest iterowana po $i \in \langle 1; 1024)$. Jest to o tyle dziwne, że kolumny w tablicy są numerowane w zakresie $\langle 0; 1024)$. Prowadzi to do wniosku, że tu także należałoby dokonać korekty, rozszerzając obliczenia o indeks 0. Jeśli jednak okazałoby się, że pierwotny zapis był zamierzony, poniższe rozważania są na tyle ogólne, że zmiana liczby iteracji z 1024 na 1023 nie powinna ich zaburzyć.

Właściwa analiza problemu

Pierwotnie rozpatrywałem rozwiązanie zadania w kategoriach modyfikacji układu kodu przy zachowaniu innej formy, operując głównie na poziomie manipulacji dyrektywami OpenMP i kodu programu. Jednak w tym podejściu udostępnione w zadaniu dane dotyczące pamięci podręcznej okazywały się zbędne – co nie tylko wzbudziło mój niepokój na poziomie „metazadaniowym", ale też naprowadziło mnie na wniosek, że istotnie rozważania na poziomie pomijającym organizację pamięci powodują spłycenie ich uniemożliwiające wyciągnięcie znaczących wniosków. W związku z tym te relatywnie jałowe rozważania pomijam, przechodząc do właściwego opisu.

Właściwa analiza opiera sie na spostrzeżeniu, że zadane obliczenia są o tyle specyficzne, iż jedynie pozornie wykonywane są dla szeregu kolejnych liczb w macierzy. Załóżmy macierz $m \times n$. O ile z „ludzkiej" perspektywy liczby te są obok siebie, to z organizacji pamięci narzucanej przez kompilator wynika, że kolejne komórki pamięci to kolejne wartości w ramach *wiersza*, a zatem kolejne wartości danej *kolumny* są od siebie oddalone o n („szerokość" macierzy). Aby uniknąć niejednoznaczności, poniżej przedstawiam ogólną zależność na lokalizację (adres) w pamięci komórki odpowiadającej dowolnemu elementowi typu T we wspomnianej macierzy:

```
address(matrix[i][j]) = address(matrix[0][0]) + (i * n + j) * size(T)
```

A zatem w obliczeniach „kolumnowych", danych sprowadzonych do *cache* dotyczyć będą zależności[1] określające efektywność użycia *cache*:

- jeżeli $CLL \leqslant n$ „zmarnuje się" $n - 1$ wartości
- jeżeli $CLL > n$ „zmarnuje się" $n - \lfloor \frac{CLL}{n} \rfloor + 1$ wartości

[1] Dla uproszczenia relacji mniejszości/większości/równości zakładam tu, że długość linii CLS jest wyrażona w jednostkach odpowiadających rozmiarem zmiennej typu *float*, co nie zmienia ogólnego sensu.

Dlaczego? Otóż odwzorowanie sekcyjno-skojarzeniowe jest zwykle przeprowadzane zgodnie z zasadą lokalności – czyli sąsiedztwo danych w RAM przekłada się na sąsiedztwo w *cache*. A więc w pamięci podręcznej procesora umieszczany jest ciąg wartości znajdujących się pod kolejnymi adresami. Nie można założyć, że wszystkie używane w programie dane zmieszczą się w *cache* (w tym przypadku zadanie byłoby z punktu widzenia pamięci podręcznej trywialne[2], a rozważania mało ogólne). Bez tego założenia musimy uznać, że na pewnym etapie działania programu konieczne będzie przeładowanie części, a w końcu i całości, pamięci podręcznej. Część danych zostanie zatem niewykorzystana, ponieważ zostanie nadpisana, choć teoretycznie te wartości będą jeszcze używane w dalszym toku programu (kazdy element macierzy jest elementem jakiejś sumy).

Obliczenia dotyczące sumy wierszy również znacznie upraszczałyby program. Prowadzi to do prostego wniosku, że gdyby ewentualne operacje na macierzy w „wykropkowanym" fragmencie kodu zastąpić operacjami z zamienionymi miejscami odwołaiami do współrzędnych (i, j). Z różnych powodów (na przykład optymalizacji „wykropkowanych" operacji) takie podejście może nie być jednak możliwe. Kolejną możliwością jest wtedy dokonanie jawnej i rzeczywistej transpozycji przed i po analizowanym fragmencie. Mozliwość zastosowania takiego wariantu zależy od wydajności operacji transpozycji, a zatem głównie od szybkości pamięci RAM. Możliwe jest tez bardziej ogólne rozwiązanie, które postaram sie przedstawić poniżej.

Otóż konieczna będzie zmiana struktury programu. Zauważmy, że poprzednio „marnujące się" dane sprowadzone do *cache* można przecież jakoś wykorzystać, póki nie zostały nadpisane, skoro wiemy, że *na pewno* będą potrzebne w dalszych obliczeniach.

W ogólności można sprowadzić tę metodę do następującego postępowania: w pojedynczej iteracji obliczana jest suma częściowa dla każdej kolumny – ilość elementów sumy zależna od tego, ile wierszy mieści się w pamięci podręcznej. A więc iteracja odpowiada obliczeniu sum częściowych składających sie z w wierszy dla $k = \max(n, CLL)$ kolumn. Dzięki temu zawsze wykorzystujemy całą (bądź *prawie* całą) zawartość sprowadzoną do *cache*. Dzięki temu nie musimy też ponownie sprowadzać tego fragmentu RAMu w pozostałych procesorach.

W szczegółach rozwiązanie można sprowadzić osobnych przypadków:

1. w zależności od tego, czy $CLL < n$, czy nie
2. oraz w zależności od ilości dostępnych w systemie procesorów.

Rozpatrzmy najpierw system o niewielu procesorach.[3] W takim wypadku przetwarzanie odbywa się w odbywa się następująco. Najpierw wykonywane są zrównoleglone iteracje opisane wcześniej (zalecany podział *dynamic* z parametrem `chunk=1`). Po zakończeniu tego kroku otrzymujemy częściową (w pierwszych wierszy) sumę w k kolumnach. Ten krok wykonywany jest wiele razy, za każdym razem część macierzy objęta już sumowaniem jest „coraz większa", aż do ostatecznego wyniku obejmującego całą macierz.

Jednak problemem efektywnościowym może okazać się oczekiwanie w punkcie synchronizacji na kolejne składniki sum stanowiących ostateczny wynik, podczas gdy nie jest to tak naprawdę konieczne, ponieważ poszczególne sumy wynikowe są od siebie niezależne. W praktyce jest to realizowalne przy zachowaniu atomowości dostępu do wynikowej tablicy sum i współdzieleniu jej pomiędzy procesami (dyrektywy `atomic` i `shared`).

Rozwiązanie takie ma moim zdaniem potencjał osiągnąć efektywność (wyrażoną czasem wykonania) zbliżoną do $\frac{T_s}{p}$, gdzie T_s jest czasem wykonania sekwencyjnego programu. Dokładność przybliżenia jest zależna od dopasowania wymiarów macierzy do liczby procesorów p[4] i czasów dostępu do pamięci.

[2] W przypadku rozmiaru linii pamieci podręcznej równego bądź większego od wiersza macierzy, przypuszczam, że program w obecnej postaci byłby optymalny – przy założeniu krótkiego względem sumarycznego czasu przetwarzania dostępu do pamięci

[3] Liczba procesorów $p \leqslant \lceil \frac{n}{k} \rceil$

[4] Powyżej limitu zrównoleglenia część procesorów zawsze pozostanie bezczynna