Programowanie współbieżne - Projekt

Piotr Kubicki

Wstęp

Projekt ma na celu zoptymalizowanie i uwspółbieżnienie rozwiązania problemu Świszcz z kursu ASD1. W zadaniu mowa jest o cięciu deski w wyznaczonych miejscach. Na wejściu dostajemy długość deski d i n współrzędnych cięć. Koszt każdego cięcia to większa z odległości od cięcia do końca ciętego kawałka deski. Należy podać minimalny koszt wykonania wszystkich cięć oraz ich odpowiednią kolejność.

Rozwiązanie zadania

Rozwiązanie bazuje na podejściu dynamicznym. Oznaczmy końce otrzymanych kawałków deski jako

$$a_0, a_1, ..., a_n, a_{n+1} \quad a_0 = 0,\ a_{n+1} = d$$

Niech $S(i, j)$ będzie kosztem pocięcia deski od a_i do a_j. Możemy zauważyć, że:

$$S(i, i+1) = 0$$

$$S(i, j) = \min_{k \in i...j} \cdot(\max(a_k - a_i,\ a_j - a_k) + S(i, k) + S(k, j))$$

To pozwala nam obliczyć dynamicznie wynik, czyli $S(0, n+1)$. W programie będziemy przechowywać kwadratową tablicę $(n + 1)$x$(n + 1)$. W i-tym wierszu wyliczanych będzie $n+1-i$ wartości $S(a, b)$ dla $|a - b| = i + 1$. W ostatnim wierszu znajdzie się wynikowa wartość.

Dla każdej wartości z i-tego wiersza trzeba wykonać i porównań. Obliczanie ostatecznego kosztu ma więc złożoność $O(n^3)$. Dla każdej wyliczonej wartości w bliźniaczej tablicy zapamiętamy dodatkowo współrzędną optymalnego podziału. To pozwoli odtworzyć optymalną kolejność cięć w złożoności $O(n)$.

Rozkład danych oraz wartość d w żaden sposób nie wpływają na algorytm, więc jedynie liczba cięć n ma znaczenie. Zadanie dopuszcza $n <= 400$, ale programy testowane są głównie na 10 zestawów o $n = 1000$. Pierwotny program oblicza rozwiązania w rzeczywistym czasie około $5.25s$.

Optymalizacja

Bazowy program, zaliczający testy na satori nosi nazwę `D.cpp`. W programie `D2.cpp` użyta jest pomocnicza zmienna podczas szukania minimum, co pomaga jedynie przy kompilacji z flagami `-O0, -O1`. Dla optymalizacji `-O3` program działa wolniej. W kodzie `D3.cpp` dwuwymiarowe tablice zostały zamieniona na jednowymiarowe, w których dane trzymane są w sposób spójny. Mimo kosztu obliczania indeksów dla odpowiednich wierszy, lepsze *cache*'owanie przyspieszyło czas działania programu do niecałych 5 sekund. Poprawa jest znacznie większa dla mniejszych wartości n.

Uwspółbieżnienie

Threads

W programie `Dt.cpp` zastosowane są wątki z `<thread>`. Wykonanie środkowej pętli obliczającej wartości dla jednego wiersza powierzone jest `t_number` wątkom. Każdy z nich czyta wartości z poprzednich wierszy i zapisuje wynik do własnych komórek, więc działają niezależnie od siebie. Testowane było liczenie sekwencyjne lub mniejsza liczba wątków dla krótkich wierszy, ale najlepsze wyniki daje użycie

zawsze `t_number` wątków. W tabeli widoczne są uśrednione czasy dla 10 zestawów o $n = 1000$, w zależności od liczby wątków:

`t_number`	1	2	3	4	5	6	7	8	9	10	11	12
`real time`	6.0	5.0	6.5	6.5	7.0	4.0	3.2	2.7	3.0	3.0	3.1	3.2

Koszt tworzenia jednego wątku nieco wpływa na działanie programu. Dla dwóch wątków program działa trochę szybciej. Kolejne wartości są nieoptymalne; dopiero dla sześciu wątków następuje większa poprawa. Widzimy, że optymalna jest liczba odpowiadająca liczbie wątków sprzętowych(4 rdzenie, każdy obsługuje 2 wątki). Dalsze jej zwiększanie stopniowo pogarsza realny czas wykonania.

Program `Dt2.cpp` jest próbą przydzielenia wewnętrznej pętli, liczącej minimalną wartość, między wątki. Każdy z nich oblicza minimum dla pewnego podzbioru. Po zakończeniu wybierana jest minimalna wartość z wyników pracy wątków. Czas działania jest nieporównywalnie dłuższy, dlatego ten sposób nie był dokładniej analizowany.

OpenMP

Programy `Do.cpp` i `Do2.cpp` to odpowiedniki powyższych, ale z zastosowaniem biblioteki `<omp.h>`. W pierwszym z nich wystarcza dodanie jednej linii nad odpowiednią pętlą; drugi naiwnie przydziela odpowiednie minimum w sekcji critical, jednak przez znaczne pogorszenie czasu, sposób synchronizacji nie był ulepszany. Tabela dla `Do.cpp`:

`num_threads`	1	2	4	8	12
`real time`	4.7	2.5	1.4	1.2	1.6

Przy wykorzystaniu wszystkich 8 wątków, efektywność jest ponad dwa razy większa niż dla `<thread>`. Ciekawym jest, że program `Do.cpp` działał lepiej dla spójnego układu pamięci, za to wątki definiowane w `Dt.cpp` lepiej radziły sobie przy dwuwymiarowych tablicach.

Serwer Miracle

Program `Do.cpp` dał jeszcze lepszy wynik, uruchomiony na serwerze Miracle dla 14 wątków: $1.1s$. Sąsiednie wartości dawały nieco gorsze wyniki. Jedynie dla 16 wątków program nie przestawał się wykonywać.

Wersja ze standardowymi wątkami z `<thread>` wciąż wypadała najlepiej dla 8 wątków, przy czym trwało to w najlepszym przypadku $3.5s$

Podsumowanie

Problem Świszcz okazał się być idealnym do zastosowania wątków. Jego złożoność $O(n^3)$ pozwala użyć wątków mających odpowiednio duże i niekolidujące ze sobą podzadania, podobnie jak w przypadku algorytmu Floyda-Warshalla, czy mnożenia macierzy. W przypadku wersji z `<omp.h>` pomogła optymalizacja wielkości pamięci. Ostatecznie dla używanych tutaj danych realny czasy wykonania z $5,3$ sekund spadł do 1.2 / 1.1 sekundy.