Algorytmy równoległe 2015 (zad. 4)

Michał Liszcz

2016-01-04

Contents

1 Wstęp **2**

2 Opis algorytmu **2**

2.1 Wybór wartości *pivot* 2

3 Testy wydajności **3**

3.1 Wyniki 3

3.1.1 Sortowanie 16'000'000 liczb 3

3.1.2 Sortowanie 32'000'000 liczb 5

3.2 Wnioski 6

1 Wstęp

Celem zadania jest projekt i implementacja równoległego algorytmu sortowania. Rozwiązanie należy zbadać pod kątem wydajności (skalowalność i przespieszenie).

Opracowane przeze mnie rozwiązanie bazuje na rozproszonym algorytmie *Quick-sort* przedstawionym w [1].

2 Opis algorytmu

Rozproszony algorytm wykorzystuje schemat komunikacji oparty o hipersześcian. Dane jest $P = 2^k$ procesorów oraz zbiór $M \cdot P$ danych do posortowania.

1. W pierwszym kroku do każdego z procesorów przydzielona jest kolejna porcja M liczb. Przydziałem zajmuje się jeden, wyróżniony procesor, który przesyła dane do pozostałych procesorów. Jednocześnie każdy z procesorów przechowuje na stercie co najwyżej M liczb.
2. Utworzona zostaje *grupa* procesorów (zrealizowana poprzez komunikator w MPI), złożona ze wszystkich procesorów.
3. W grupie zostaje wyróżniony jeden procesor. Na podstawie swojej porcji liczb wybiera lokalnie *pivot* (liczbę użytą do podziału zbioru danych). Sposób wybrania tej liczby opisany jest dalej.
4. *pivot* zostaje rozgłoszony do wszystkich elementów w grupie.
5. Każdy z procesorów w grupie wyznacza indeks swojego sąsiada, poprzed odwrócenie najbardziej znaczącego bitu w zapisie binarnym swojego indeksu. Przykładowo, sąsiadem procesora $6 = 110$ jest procesor $2 = 010$.
6. Sąsiednie procesory wymieniają się danymi. Należy pamiętać że procesor może przesłać większą liczbę danych niż M (zwłaszcza w kolejnych krokach algorytmu). W MPI można wykorzystać procedurę `MPI_Probe` do określenia potrzebnego rozmiaru bufora.
7. Sąsiad o niższym indeksie zachowuje elementy mniejsze lub równe od *pivot*. Sąsiad o wyższym indeksie zachowuje elementy większe od *pivot*. Te zbiory stają się nowymi danymi w procesorach.
8. Po takiej wymianie wśród procesorów można wyodrębnić dwie rozłączne grupy: te które przechowują elementy mniejsze (bądź równe) *pivot* (indeksy $0xyz...$), i te o większych elementach (indeksy $1abc...$). Pierwsza połowa procesorów w komunikatorze MPI ma mniejsze elementy, druga - większe.
9. Jeżeli w grupie jest więcej niż dwa procesory, dzielimy ją na dwie grupy opisane w punkcie powyżej (z wykorzystaniem `MPI_Comm_split`). Dla każdej z tych grup powtarzamy rekurencyjnie procedurę, wracając od punktu 3.
10. Jeżeli w każdej grupie zostały tylko dwa procesory, algorytm równoległy jest zakończony. Każdy z procesorów posiada *przedział* początkowego zbioru liczb. Dodatkowo, wszystkie liczby w pierwszym przedziale procesora o indeksie i są większe od liczb procesora $i - 1$ i mniejsze od liczb procesora $i + 1$.
11. Liczby w każdym procesorze można posortować dowolnym algorytmem sekwencyjnym (wybrałem `std::sort` o złożoności $O(N \log N)$, dostępny w bibliotece standardowej C++).
12. Można złączyć listy z kolejnych procesorów, otrzymując posortowaną postać wejściowej sekwencji.

2.1 Wybór wartości *pivot*

Istnieje wiele sposobów na dobór wartości *pivot* przy podziale danych w algorytmach klasy *Quick-sort*. Przy jej wyborze należy kierować się kosztem czasowym jej wyznaczenia oraz stosunkiem w jakim podzieli zbiór liczb.

Przy ustalonej metodzie wyboru *pivot*-a, "jakość" podziału zależy od zbioru liczb. W przypadku trywialnego wyboru (na przykład wybór pierwszej wartości w tablicy), złożoność algorytmu zdegraduje się do $O(N^2)$ dla danych (częściowo) posortowanych.

Znacznie lepsze okazują się techniki niedeterministyczne, przykadowo *median-of-three*, polegająca na losowym wyborze trzech liczb z całego zbioru i przyjęciu ich mediany jako *pivot*. W taki sposób, przy dużej ilości liczb można uzyskać bardzo równomierne podziały. Zdarzają się jednak podziały $1:2$ lub gorsze (zwłaszcza kiedy liczb jest mało).

W moim rozwiązaniu *pivot* liczę jako średnią artymetyczną największego i najmniejszego elementu w tablicy [1]. Empirycznie sprawdziłem że generowane podziały są lepsze niż w przypadku *median-of-three* (dla sortowanych danych losowanych z rozkładem jednostajnym). Wyznaczenie *pivot*-a wymaga liniowego przejścia całej tablicy, jednak operacja ta zostanie wykonana tylko kilka razy (dokładnie $2P-1$, P - procesory), a każdy z procesorów i tak przegląda całe swoje dane przy wymianie z sąsiadem i filtrowaniu.

3 Testy wydajności

Pomiary wykonałem z wykorzystaniem klastra Zeus w ACK Cyfronet AGH.

Badałem trzy rozmiary problemów - sortowanie zbioru 3'200'000, 16'000'000 oraz 32'000'000 liczb wylosowanych z rozkładem jednostajnym. W pierwszym przypadku czasy są małe a wyniki obarczone bardzo dużym błędem - pomijam je.

Liczbę procesorów zwiększałem kolejno do 2, 4, 8, 16, i 32.

DLa każdej ilości procesorów i rozmiaru problemu wykonałem 10 pomiarów z liczbami wygenerowanymi z losowym ziarnem generatora oraz 10 pomiarów z losowym ziarnem. W obu wariantach wyniki były zgodne i dawały takie same charakterystyki przyspieszenia i efektywności.

3.1 Wyniki

3.1.1 Sortowanie 16'000'000 liczb

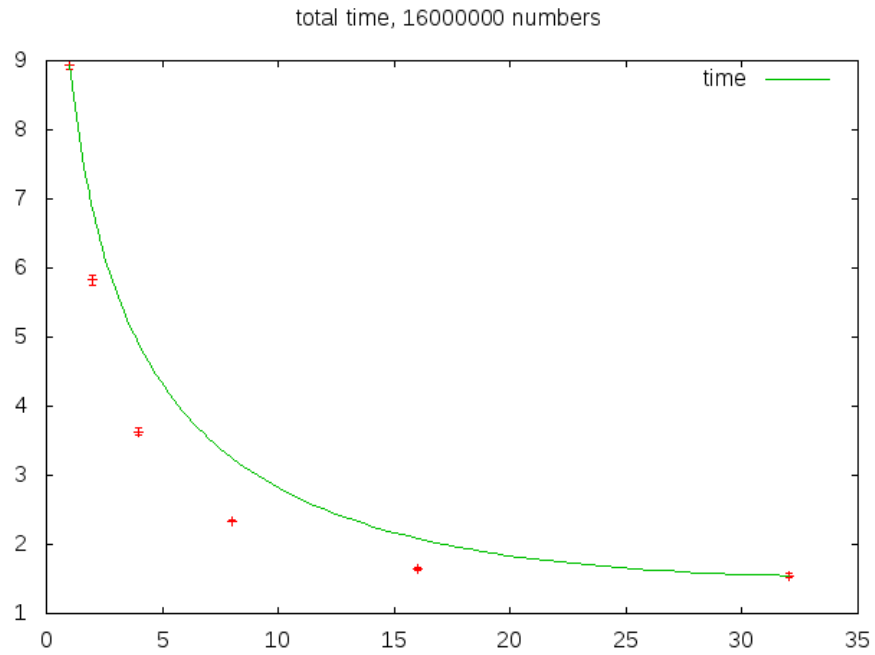

Figure 1: Czas wykonania - 16'000'000 liczb.

[1] Można również policzyć średnią wszystkich elementów

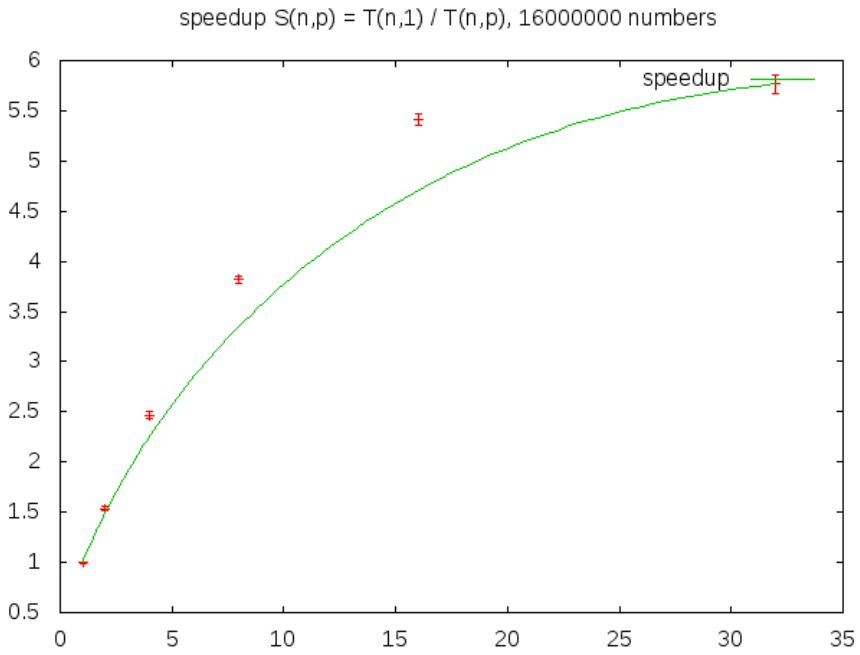

Figure 2: Przyspieszenie - 16'000'000 liczb.

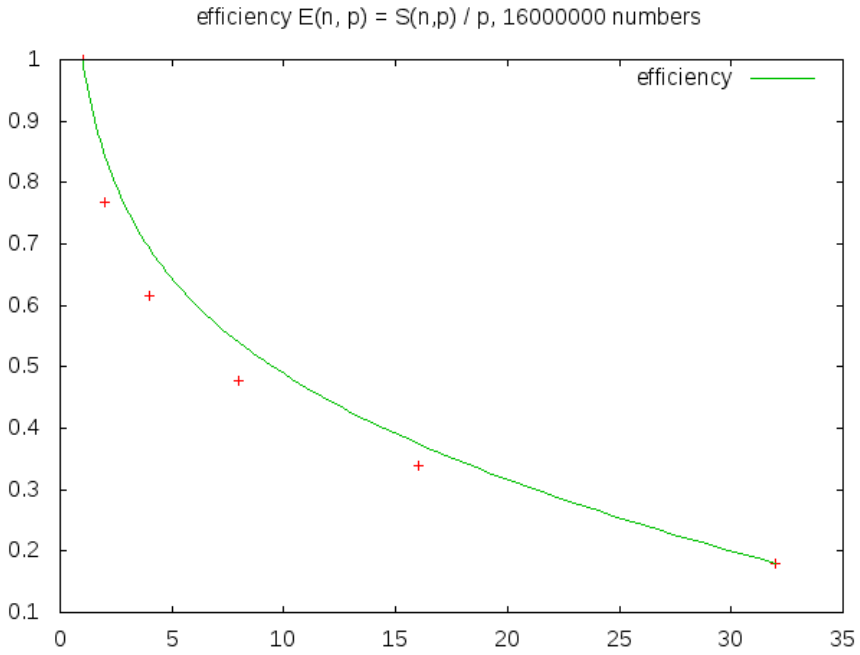

Figure 3: Efektywność - 16'000'000 liczb.

3.1.2 Sortowanie 32'000'000 liczb

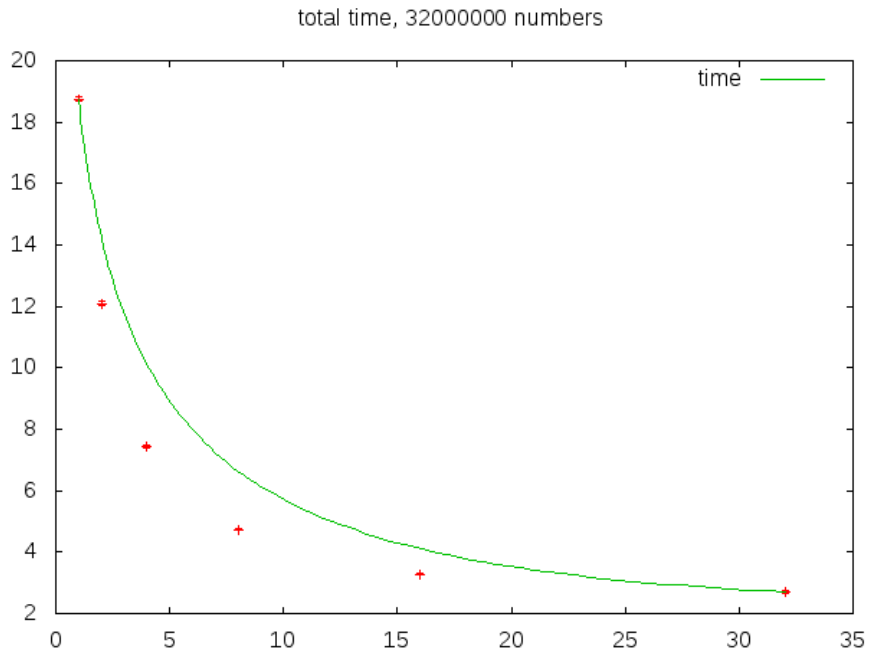

Figure 4: Czas wykonania - 32'000'000 liczb.

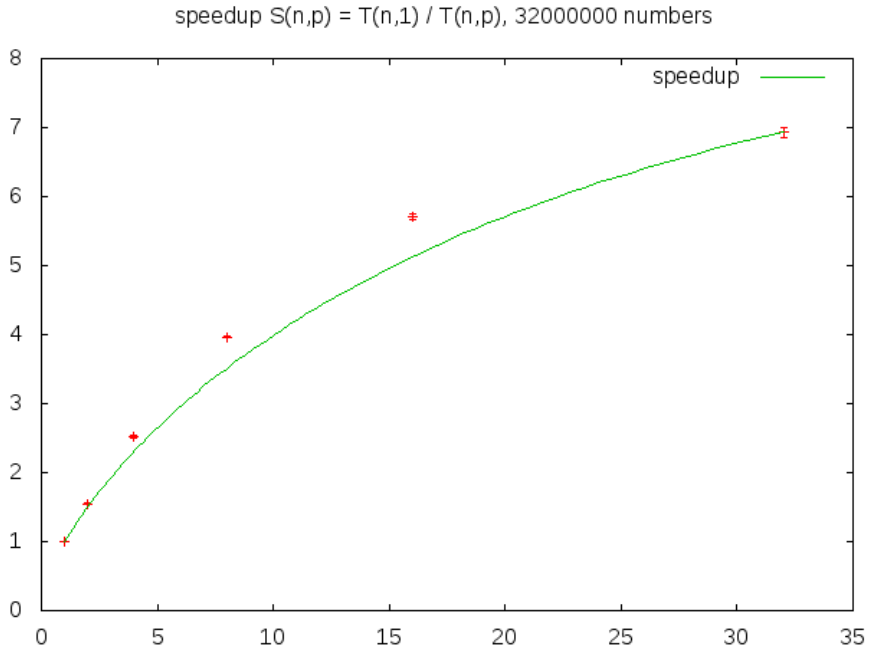

Figure 5: Przyspieszenie - 32'000'000 liczb.

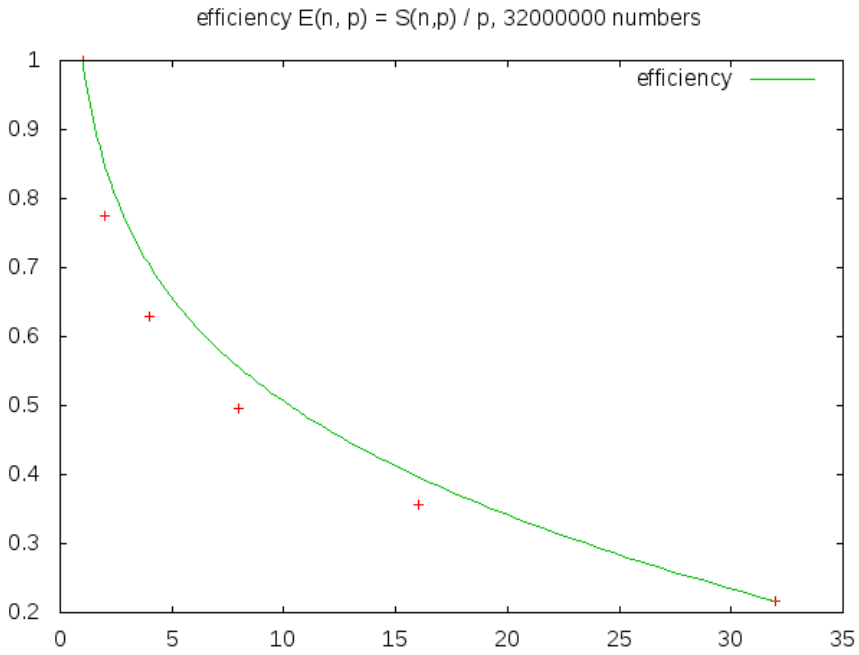

Figure 6: Efektywność - 32'000'000 liczb.

3.2 Wnioski

Czas obliczeń maleje zgodnie z oczekiwaniami. Dla większych rozmiarów problemu można wykorzystać jeszcze więcej procesorów.

Charakterystyka przyspieszenia nie jest liniowa a efektywność maleje ze wzrostem liczby procesorów.

References

[1] *Divide-and-conquer Parallelization Paradigm*, http://cacs.usc.edu/education/cs653/02-3DC.pdf.