Konrad Lisiecki 291649

Zadanie 3 - MPI: Problem maksymalnej podtablicy

Programowanie współbieżne i Rozproszone

Semestr letni 2013/14

Warszawa, 2014-05-27

Spis treści

Rozdział 1

Wprowadzenie

Problem maksymalnej podtablicy Celem zadania jest zaimplementowanie algorytmu rozwiązującego problem maksymalnej podtablicy. Pojawia się on w wielu zastosowaniach współczesnej nauki, jak np. w grafice komputerowej czy eksploracji danych. Ponieważ operacje te są zazwyczaj kosztowne obliczeniowo, warto poszukać sposobu na przyspieszenie ich działania, np. poprzez zrównoleglenie.

Rozdział 2

Schemat algorytmu

Algorytm Użyty w rozwiązaniu algorytm został zaczerpnięty z pracy autorstwa Bae pt. „Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem" [1]. Algorytm ten można przedstawić w postaci następującego pseudokodu: Schemat algorytmu wygląda następująco:

Algorithm 1 Schemat algorytmu rozwiązania problemu maksymalnej podtablicy

```
1:  procedure FIND
2:      sum[0][1..n] ← 0, sum[1..m][0] ← 0
3:      for i in range(1, M) do
4:          for j in range(1, N) do
5:              sum[i][j] ← sum[i − 1][j] + sum[i][j − 1] + A[i][j] − sum[i − 1][j − 1]
6:          end for
7:      end for
8:      for g in range(1, m) do
9:          for i in range(g, m) do                    ▷ przypadek jednowymiarowy
10:             Min ← 0
11:             for j in range(1, n) do
12:                 s ← sum[i][j] − sum[g − 1][j]
13:                 cond ← s − Min
14:                 Max ← MAX(Max, cond)
15:                 Min ← MIN(Min, s)
16:             end for
17:         end for
18:     end for
19: end procedure
```

Opis implementacji algorytmu Jak wynika z powyższego pseudokodu, algorytm składa się z dwóch zasadniczych części. W pierwszej części każdy proces oblicza, na podstawie podanej macierzy wejściowej, tablicę zawierającą sumy prefiksowe dla tej macierzy. W drugiej fazie, każdy proces lokalnie przegląda przyporządkowany mu fragment tablicy i na tej podstawie wyznacza lokalną fragment macierzy o największej sumie. Po tym, każdy z tych procesów wysyła swój wynik do pewnego wyróżnionego procesu arbitra, który to sprawdza, który z wyników jest maksymalny (a zatem który jest maksimum globalnym).

[1] http://www.cosc.canterbury.ac.nz/research/reports/PhdTheses/2007/phd_0705.pdf

Rozdział 3

Wyniki testów

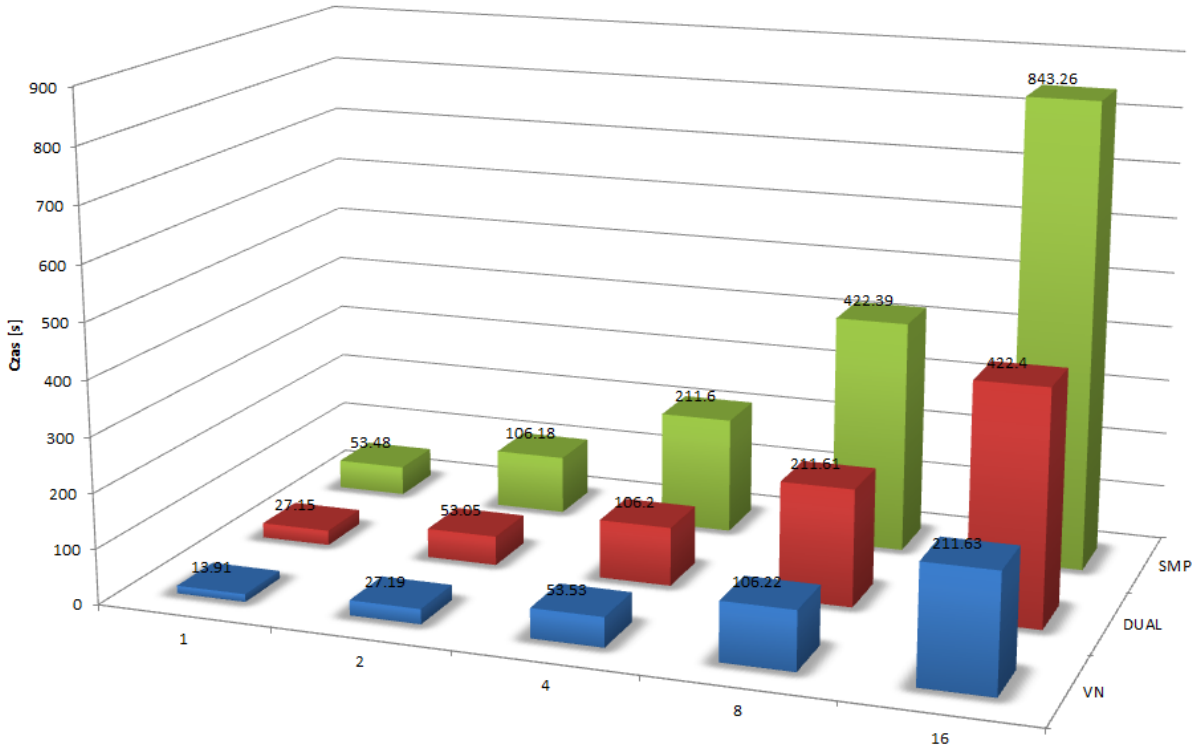
Klaster obliczeniowy NOTOS W niniejszym rozdziale zostaną przedstawione wyniki z wykonania testów. Były one wykonywane na klastrze **notos** znajdującym się w centrum obliczeniowym ICM. Jest to klaster o następującej specyfikacji:

- Producent: IBM
- Model: Blue Gene/P
- Architektura: ppc
- Liczba węzłów: 1024
- Sumaryczna liczba rdzeni: 4096
- Sumaryczna pamięć operacyjna: 4 TB
- Technologia sieciowa: trójwymiarowy torus
- System operacyjny: Blue Gene/P Linux 2.6
- System plików: GPFS
- System kolejkowy: LoadLeveler

i następujących parametrach:

- Procesor: Quad-Core PowerPC 450
- Częstotliwość: 850 MHz
- Pamięć operacyjna: 4 GB
- Dysk twardy: brak

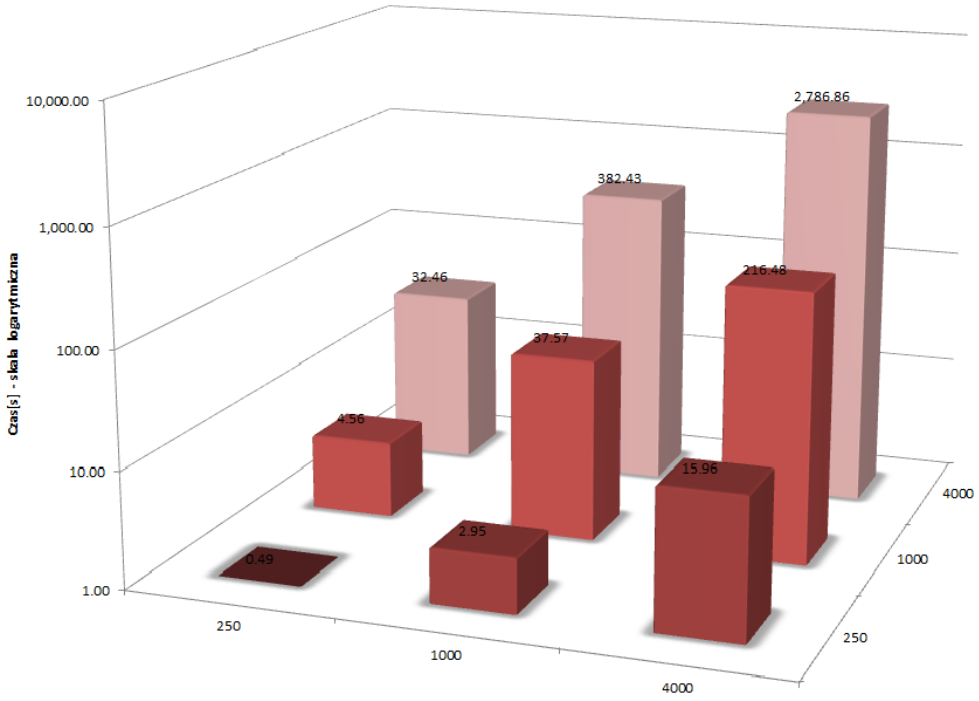
Testy wydajnościowe z użyciem 1,2,4,8,16 węzłów i 1,2,4 procesów na węzei Na obrazku nr 4.2 przedstawiono wykres porównujący czasy wykonania algorytmu dla różnych wartości węzłów i procesorów na węzeł. Wynika z nich, że podwojenie liczby węzłów lub podwojenie liczby procesorów powoduje prawie dwukrotne przyspieszenie działania algorytmu.



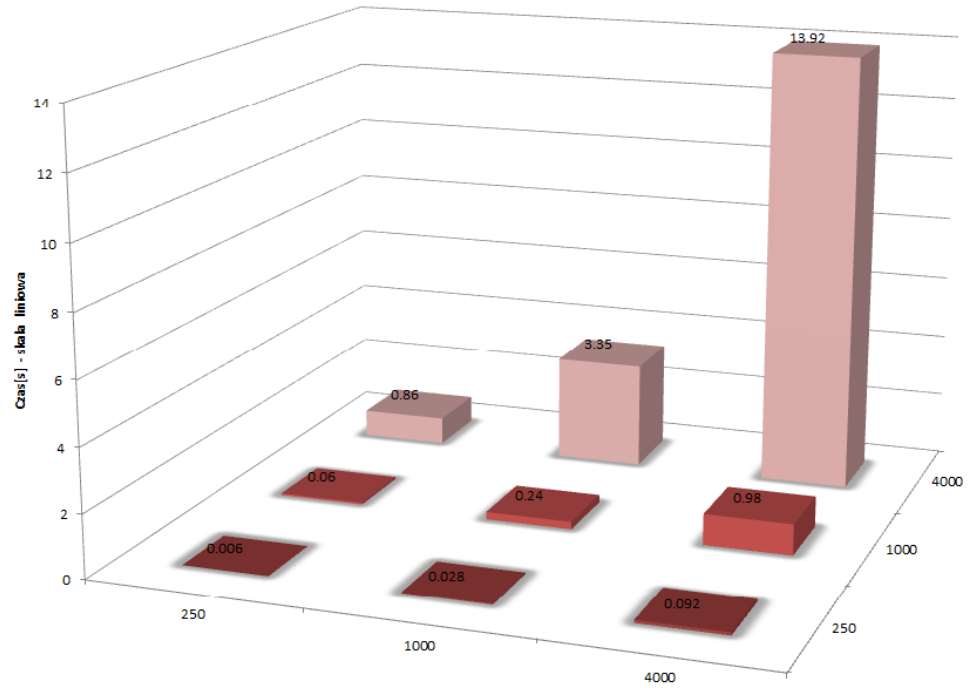
Rysunek 3.1: Porównanie szybkości algorytmu Brandesa na GPU i CPU dla dużych grafów

Porównanie prędkości działania naiwnej implementacji i implementacji na klaster Na rysunku 3.2. możemy zobaczyć wyniki dla algorytmu naiwnego. Z powodu logarytmicznej skali na osi pionowej wzrost długości działania wraz z rosnącym rozmiarem macierzy jest liniowy. Oznacza, to że wzrost długości działania jest ekspotencjalny. Podony rezultat otrzymano dla implementacji na klaster (wzrost ekspotencjalny jest wprost widoczny na rysunku 3.3, ze względu na zastosowaną skalę liniową).

Testy poprawnościowe Testy poprawnościowe wykonywano na małych macierzach, których rozmiar nie przekraczał 200 wierszy i kolumn. Wyniki, które dawały testy, wykazywały zgodność z wynikami dawanymi przez program z algorytmem naiwnym.



Rysunek 3.2: Skalowalność algorytmu w zależności od liczby węzłów i liczby procesorów na węzeł. W skali logarytmicznej widać zależność liniową



Rysunek 3.3: Skalowalność algorytmu w zależności od liczby węzłów i liczby procesorów na węzeł. W skali liniowej widać zależność wykładniczą

Rozdział 4

Tabela z wynikami

Tabela z czasami dla algorytmu naiwnego Poniższa tabela przedstawia wyniki jakie osiągnięto przy pomocy algorytmu naiwnego:

Naive	250	1000	4000
250	0.49	2.95	15.96
1000	4.56	37.57	216.48
4000	32.46	382.43	2786.86

Tabela z czasami dla implementacji na klaster Poniższa tabela przedstawia wyniki jakie udało się osiągnąć na klastrze NOTOS:

Notos	250	1000	4000
250	0.006	0.028	0.092
1000	0.06	0.24	0.98
4000	0.86	3.35	13.92

Speedup Przyspieszenie było liczone wg następującego wzoru:

$$speedup = \frac{\texttt{Wynik dla testu dla algorytmu naiwnego}}{\texttt{Wynik dla testu na klastrze}}$$

Poniższa tabela przedstawia przyspieszenia, które wylioczono na podstawie podanego wzoru:

Speedup	250	1000	4000
250	81.10	105.24	173.47
1000	75.98	156.52	220.89
4000	37.74	114.15	200.20

Widać więc, że dla największych testów osiągnięto przyspieszenie rzędu 200 razy.

Rozdział 5

Zawartość rozwiązania

Pliki z kodami źródłowymi Dołączony do opracowania folder z kodami źródłowymi zawiera:

1. report.pdf.
2. Makefile - Plik makefile kompilujący program równoległy do msp-par.exe.
3. msp-seq-naive.c - Oryginalny sekwencyjny program implementujący algorytm naiwny.
4. msp-par.c - Państwa wersja programu równoległego.
5. msp-par.ll - Specyfikacja zadania dla M = 1000 i N = 1000 przy 32 procesach (4 procesy na maszynę).
6. matgen.h - Plik zawierający funkcje nagłówkowe i struktury do inicjalizacji tablicy.
7. matgen-mt.c - Plik implementujący funkcje do inicjalizacji tablicy.