Duże zadanie zaliczeniowe z C

Wymagane do: poniedziałek, 13 stycznia 2025, 23:59

Obliczeniowa weryfikacja hipotezy kombinatorycznej

Dla danego multizbioru liczb naturalnych A oznaczamy $\sum A = \sum_{a \in A} a$. Na przykład jeśli $A = \{1, 2, 2, 2, 10, 10\}$, to $\sum A = 27$. Dla dwóch multizbiorów piszemy $A \supseteq B$ jeśli każdy element występuje w A co najmniej tyle razy, co w B. Na cele zadania przyjmijmy następujące definicje.

Definicja. Multizbiór A nazywamy *d-ograniczonym*, dla liczby naturalnej d, jeśli jest skończony i wszystkie jego elementy należą do $\{1, \ldots, d\}$ (z dowolnymi powtórzeniami).

Definicja. Parę d-ograniczonych multizbiorów A, B nazywamy *bezspornymi*, jeśli dla wszystkich $A' \subseteq A$ i $B' \subseteq B$ zachodzi

$$\sum A' = \sum B' \iff A' = B' = \emptyset \vee (A' = A \wedge B' = B).$$

Innymi słowy $\sum A = \sum B$, ale poza tym sumy dowolnych niepustych podzbiorów A i B muszą się różnić.

Problem. Dla ustalonego $d \geq 3$ (mniejszych d nie będziemy rozważać) i multizbiorów A_0, B_0, chcemy znaleźć bezsporne d-ograniczone multizbiory $A \supseteq A_0$ i $B \supseteq B_0$, które maksymalizują wartość $\sum A$ (równoważnie $\sum B$). Oznaczmy tę wartość przez $\alpha(d, A_0, B_0)$. Przyjmijmy $\alpha(d, A_0, B_0) = 0$ jeśli A_0 i B_0 nie są d-ograniczone lub nie mają d-ograniczonych bezspornych nadmultizbiorów.

Przykład. $\alpha(d, \emptyset, \emptyset) \geq d(d-1)$.
Szkic dowodu. Zbiory $A = \{\underbrace{d, \ldots, d}_{d-1 \text{ razy}}\}$ oraz $B = \{\underbrace{d-1, \ldots, d-1}_{d \text{ razy}}\}$ spełniają warunki dla $\sum A = d(d-1) = \sum B$.

Przykład. $\alpha(d, \emptyset, \{1\}) \geq (d-1)^2$.
Szkic dowodu. Zbiory $A = \{1, \underbrace{d, \ldots, d}_{d-2 \text{ razy}}\}$ oraz $B = \{\underbrace{d-1, \ldots, d-1}_{d-1 \text{ razy}}\}$ spełniają warunki dla $\sum A = 1 + d(d-2) = (d-1)^2 = \sum B$.

Można udowodnić, że powyższe przykłady są najlepsze, tzn. $\alpha(d, \emptyset, \emptyset) = d(d-1)$ oraz $\alpha(d, \emptyset, \{1\}) = (d-1)^2$.

Niemniej w tym zadaniu będziemy chcieli zweryfikować to obliczeniowo, dla jak największych d, a także wyliczyć wartości α dla innych wymuszonych multizbiorów A_0, B_0.

Rekurencja z nawrotami

Wartości $\alpha(d, A_0, B_0)$ możemy wyliczyć rekurencyjnie, budując inkrementacyjnie multizbiory $A \supseteq A_0$ i $B \supseteq B_0$. Oznaczmy przez $A^\Sigma = \{\sum A' : A' \subseteq A\}$, czyli zbiór wszystkich możliwych sum jakie możemy uzyskać ze zbioru A (nie multizbiór, tzn. nie interesuje nas na ile sposobów można daną sumę uzyskać z elementów jednego multizbioru). Stosujemy poniższą rekurencję.

?

Solve(d, A, B):
  if $\sum A > \sum B$ **then** swap(A, B)
  $S \leftarrow A^\Sigma \cap B^\Sigma$
  if $\sum A = \sum B$ **then**
    if $S = \{0, \sum A\}$ **then return** $\sum A$
    else return 0
  else if $S = \{0\}$ **then**
    return $\max_{x \in \{\text{last} A, \ldots, d\} \setminus B^\Sigma} \text{Solve}(d, A \cup \{x\}, B)$
  else return 0

gdzie *last* A oznacza element ostatnio dodany do A, w przypadku $A = A_0$ przyjmujemy 1 (tzn. rekurencja dodaje do A_0 elementy niemalejąco).

W praktyce, aby nie liczyć zbiorów sum A^Σ i B^Σ za każdym razem od nowa, przekazujemy A^Σ i B^Σ. Gdy dodajemy element x do A, to nowy A^Σ wynosi $A^\Sigma \cup (A^\Sigma + x)$, gdzie $A^\Sigma + x$ to zbiór powstały z A^Σ przez powiększenie każdego elementu o x. Zbiory sum A^Σ i B^Σ reprezentujemy wydajnie za pomocą tzw. bitsetów.

W folderze `reference` załączonego archiwum znajduje się szczegółowa referencyjna implementacja: względnie zoptymalizowana, ale sekwencyjna, rekurencyjna.

Zadanie

Zadanie polega na napisaniu dwóch innych implementacji tego samego obliczenia: nie-rekurencyjnej (ale dalej jedno-wątkowej), oraz równoległej (wielo-wątkowej), uzyskującej jak najlepszy współczynnik skalowalności. Dodatkowo, należy załączyć raport prezentujący skalowalność wersji równoległej na wybranych testach z różną liczbą wątków.

Celem zadania nie jest teoretyczna analiza matematycznego problemu ani mikro-optymalizacje operacji na bitach; dlatego implementacje operacji na multizbiorach są dane, nie należy ich zmieniać, a rozwiązanie powinno wykonać dokładnie te same operacje na multizbiorach co referencyjna implementacja , tylko równolegle (w szczególności może je wykonać w innej kolejności); szczegóły niżej w wymaganiach programu.

Współczynnik skalowalności. Załóżmy, że wersja referencyjna działa na zadanym wejściu w czasie t_s, a wersja równoległa uruchomiona na n wątkach pomocniczych zadziała w czasie t_n (przy założeniu, że na maszynie jest fizycznie co najmniej n rdzeni). Wtedy *współczynnik skalowalności* wynosi $\frac{t_s}{t_n}$. Im wyższy tym lepsza skalowalność, idealnie skalujące się rozwiązanie osiągało by współczynnik n (technicznie możliwe jest trochę więcej, jeśli rozwiązanie na jednym wątku zadziała szybciej niż referencyjna implementacja).

Sugerowane podejście. Sugerujemy zacząć od implementacji nie-rekurencyjnej (np. używając własnej implementacji stosu), a następnie ją zrównoleglić. W implementacji nie-rekurencyjnej należy zwrócić uwagę przede wszystkim na alokację i zwalnianie pamięci.

Trudność w zrównoleglaniu polega na tym, że trudno przewidzieć które rozgałęzienia rekurencji okażą się największe obliczeniowo, nie da się więc z góry podzielić pracy. Jednocześnie rozwiązania, które w każdej iteracji synchronizują się z innymi wątkami będą zbyt powolne. Sugerujemy więc, by każdy wątek pracował większość czasu nad własną gałęzią i tylko co jakiś czas rozważał oddanie jakiejś podgałęzi do wspólnej puli zadań. Wątki które skończą wyliczać swoją gałąź mogą pobrać kolejne zadania do wykonania ze wspólnej puli.

Ocenie podlega czas działania, poprawność wyników i (w mniejszym stopniu) raport.

?

Format wejścia/wyjścia

Funkcje implementujące poprawne parsowanie wejścia i formatowanie wyjścia znajdują się w plikach `io.h`, `io.c`.

Program na standardowym wejściu dostaje trzy wiersze. W pierwszym wejściu znajdują się cztery liczby t, d, n i m oznaczające odpowiednio liczbę wątków pomocniczych, parametr d oraz liczbę wymuszonych elementów A_0 i B_0. W drugim i trzecim wierszu znajduje się odpowiednio n i m liczb z przedziału od 1 do d: elementy odpowiednio multizbioru A_0 i B_0. Należy wyliczyć $\alpha(d, A_0, B_0)$ używając t wątków pomocniczych. (Można nie wliczać do t wątku głównego, o ile nie dzieją się w nim żadne obliczenia z `sumset.h`.)

Przykład wejścia

```
8 10 0 1

1
```

oznacza wyliczenie $\alpha(10, \emptyset, \{1\})$ używając ośmiu wątków pomocniczych.

Na wyjściu należy wypisać rozwiązanie A, B (czyli bezsporne d-ograniczone multizbiory $A \supseteq A_0, B \supseteq B_0$ maksymalizujące $\sum A = \sum B$). W pierwszym wierszu należy wypisać jedną liczbę $\sum A$. W drugim i trzecim wierszu opis multizbioru odpowiednio A i B. Opis multizbioru składa się z wypisania elementów wraz z krotnościami pooddzielanymi pojedynczymi spacjami. Jeżeli dany element występuje w krotności 1, to wypisujemy tylko ten element bez krotności. Wpp., jeżeli dany element a występuję k razy, to wypisujemy k`x`a (krotność, iks i element). Elementy wypisujemy rosnąco (niezależnie od krotności).

W przypadku gdy rozwiązania nie ma, należy wypisać sumę zero i dwa puste zbiory, tj. `0\n\n\n`.

Przykład wyjścia

```
81
9x9
1 8x10
```

jest poprawnym wyjściem dla powyższego przykładu (niekoniecznie jedynym).

Wymagania

Można założyć, że:

- Wejście jest zgodne z podanym formatem.
- $1 \leq t \leq 64,\ \ 3 \leq d \leq 50,\ \ 0 \leq n \leq 100,\ \ 0 \leq m \leq 100$
- $\alpha(d, A_0, B_0) \leq d(d-1)$ dla wszystkich d, A_0, B_0.

Program

- Rozwiązanie powinno przejść wszystkie rozgałęzia rekurencji, tak jak podana wersja referencyjna. W szczególności wymagamy, by multizbiór par (A^Σ, x) występujących w wywołaniach `sumset_add(a, i)` był nadzbiorem tego z załączonej rekurencyjnej wersji (istotna jest tylko wartość x i elementy w zbiorze A^Σ, nie pole `last`, itp.). Chodzi o to by nie optymalizować samych obliczeń (np. tablicując wyniki), tylko skupić się na ich zrównoleglaniu.
- Nie wolno implementować własnych bitsetów. Można jedynie używać dostarczonej biblioteki. Nie wolno sięgać do bitów w polu `sumset` struktury `Sumset` inaczej niż przez funkcje z `sumset.h`. Wolno kopiować strukturę `Sumset` (np. `*a = *b`), wolno dowolnie zmieniać pozostałe pola.
- Równoległość powinna być zrealizowana za pomocą wątków, czyli biblioteki `pthreads`. Nie należy tworzyć procesów.
- Nie wolno używać bibliotek innych niż standardowe (w tym systemowe) i

?

samodzielnie zaimplementowane.

- Rozwiązanie będzie kompilowane przy użyciu gcc ≥ 12.2 z flagami `-std=gnu17 -march=native -O3 -pthread`. Nie wolno używać C++.
- Rozwiązanie będzie ograniczone do 128 MiB przestrzeni adresowej razy liczba wątków (wliczając wątek główny). Np.:

```
echo '8 32 1 0 1' | prlimit --as=$((9*128*1024*1024)) time -f'%e
seconds, %M kb (max RSS),  exit=%x' timeout --foreground 10s ./
parallel
```

- Program nie powinien wypisywać nic poza rozwiązaniem na standardowe wyjście. Może pisać na standardowe wyjście błędów, natomiast pogorszy to wydajność (można takie rzeczy schować w `#ifndef NDEBUG`; rozwiązanie będzie testowane wyłącznie po kompilacji w trybie Release, czyli w szczególności ze zdefiniowaną stałą `NDEBUG`).
- W przypadku błędu funkcji systemowych, np. alokacji pamięci, można po prostu zakończyć działanie programu `exit(1)`.

Raport

- Powinien to być dokument w formacie `pdf`.
- Powinien zawierać wykres (lub wykresy) przedstawiające skalowalność równoległego rozwiązania dla następujących danych wejściowych: $d \in \{5, 10, 15, 20, 25, 30, 32, 34\}$, $A_0 = \emptyset$, $B_0 = \{1\}$; dla zmiennej liczby wątków będącej potęgą dwójki, od 1 do co najmniej 8 (co najwyżej 64).
- Przy liczeniu jednym wątkiem należy użyć tego samego kodu co dla wielu wątków, bez wyłączania mechanizmów synchronizacji.
- Jeśli czas wykonania dla danych parametrów przekroczy minutę, należy przerwać obliczenia i przyjąć współczynnik skalowalności -1 (lub inaczej czytelnie to zaznaczyć).
- Czasy należy mierzyć na maszynie na której referencyjne rozwiązanie zajmuje co najwyżej minutę dla wejścia `1 34 1 0 1` (np. maszyna students), z co najmniej 8 rdzeniami.
- Nie ma potrzeby załączać kodu do tworzenia wykresów; można użyć dowolnych bibliotek i języków programowania.

Rozwiązanie

- Rozwiązaniem powinno być archiwum postaci takiej jak załączony szablon `ab12345.zip`, czyli plik ZIP o nazwie `ab12345.zip` zawierający wyłącznie folder o nazwie `ab12345`, gdzie zamiast `ab12345` należy użyć własnych inicjałów i numeru indeksu (czyli nazwy użytkownika na students).
- Rozwiązanie nierekurencyjne powinno znajdować się w podfolderze `nonrecursive` i definiować wykonywalny cel o nazwie `nonrecursive` (jak w załączonym szablonie).
- Rozwiązanie równoległe powinno znajdować się w podfolderze `parallel` i definiować wykonywalny cel o nazwie `parallel` (jak w załączonym szablonie).
- Raport powinien być zawarty w pliku `ab12345/report.pdf`.
- Można w rozwiązaniu załączyć dodatkowe pliki i foldery, o ile będzie się dało skompilować i wykonać następująco na maszynie students:

```
unzip ab12345.zip
# plik ab12345/CMakeLists.txt oraz folder ab12345/common/ zostaną
skopiowane z oryginalnego archiwum, nadpisując wszelkie zmiany.
cmake -S ab12345/ -B build/ -DCMAKE_BUILD_TYPE=Release
cd build/
make
echo -n -e '1 3 1 0\n1\n\n' | ./nonrecursive/nonrecursive
echo -n -e '1 3 1 0\n1\n\n' | ./parallel/parallel
```

W szczególności:

?

    - Zmiany `/ab12345/CMakeLists.txt` zostaną cofnięte, ale można zmieniać `CMakeLists.txt` w podfolderach.
    - Do folderu `common` nie należy dodawać własnych plików (zostaną usunięte).
    - Foldery inne niż `nonrecursive` i `parallel` będą zignorowane przez `/ab12345/CMakeLists.txt`, elementy wspólne dla nierekurencyjnego i równoległego rozwiązania można zawrzeć w `nonrecursive`.
    - Można zmieniać `.clang-tidy` i `.clang-format` (użycie nie jest wymagane, jedynie zalecane).
- Nie należy zmieniać flag, opcji, itp. kompilatora, linkera i CMake. Własne pliki `CMakeLists.txt` powinny się ograniczać do `add_executable`, `add_library`, `target_link_libraries`.