Programowanie współbieżne (Info II) 19/20

Zadanie zaliczeniowe 3: Współbieżność w C++

Zadanie zaliczeniowe 3: Prastarzy Szamani

Wprowadzenie

Klan Prastarych Szamanów jest u kresu swego życia. Jedynym ratunkiem jest przygotowanie eliksiru życia, który da im nieśmiertelność, jednak muszą to zrobić szybko, gdyż ich dni są policzone. Produkcja owego eliksiru to jednak długa przygoda, wymagająca zdobycia znacznej ilości smoczych jaj oraz magicznych kryształów.

Szamani zaczynają swoją przygodę w smoczej jamie pełnej smoczych jaj. Jaja różnią się rozmiarem oraz wagą. Worek bez dna jest magicznym przedmiotem pozwalającym na zabranie jaj z pieczary, jednak jak się okazało ma on swój limit pojemności. Aby przygotować eliksir życia szamani muszą wziąć tak dużo jaj (w sensie wagi) jak to tylko możliwe (tzn. ile tylko zmieści się do worka).

Następnie szamani muszą dotrzeć do alchemika, właściciela kryształowych grot. Alchemik chętnie podzieli się z szamanami jego kolekcją magicznych kryształów, jednak nie zrobi tego za darmo - w zamian oczekuje, że szamani uporządkują piasek na jego pustyni - układając go od ziarenek najmniejszych do największych.

Do stworzenia eliksiru życia wystarczy jeden magiczny kryształ, jednak musi on być najwyższej możliwej jakości. W przeciwnym razie mikstura się nie uda, a wszystkie smocze jaja będą do wyrzucenia. Dlatego szamani muszą dokładnie porównać wszystkie kryształy w celu wybrania tego najlepszego.

Zadanie

Celem zadania jest zaimplementowanie w C++ algorytmów w sposób sekwencyjny oraz współbieżnych trzech algorytmów, rozwiązujących następujące problemy:

- Dyskretny problem plecakowy (problem 1.)
- Sortowanie przez scalanie (Merge sort) lub sortowanie szybkie (Quicksort) (problem 2.)
- Znajdowanie elementu maksymalnego (problem 3.)

Implementacje algorytmów muszą znajdować prawidłowe rozwiązanie (a nie jego przybliżenie). Implementacje sekwencyjne powinny być zaimplementowanie wydajnie:

- Problem 1. przy użyciu programowania dynamicznego
- Problem 2. w czasie O(NlogN)
- Problem 3. w czasie O(N)

Implementacje współbieżne powinny osiągać praktyczne przyspieszenie (to znaczy działać szybciej niż ich wersje sekwencyjne dla odpowiednio dużych danych wejściowych), przy czym osiągnięte przyspieszenia będą różne dla poszczególnych algorytmów i różnych danych wejściowych. Celem zadania nie jest implementacja najszybszych współbieżnych algorytmów dla powyższych problemów opublikowanych w pracach naukowych, tylko wymyślenie własnych praktycznych modyfikacji oraz analiza ich wydajności.

Do rozwiązania powinien być dołączony raport w PDF, przedstawiający analizę czasu działania zaimplementowanych algorytmów w sposób sekwencyjny oraz współbieżny dla różnej liczby wątków i rdzeni. Raport powinien zawierać stosowne wykresy i komentarz, ale nie być dłuższy niż dwie strony. Każdy eksperyment powinien być przeprowadzony kilkukrotnie, w celu poznania wartości minimalnych, maksymalnych i średnich. Najlepiej, gdyby eksperymenty zostały przeprowadzone w dwóch różnych środowiskach np. laptop lub komputer w laboratorium i maszyna students (to zrozumiałe, że laptop może mieć niewystarczającą liczbę rdzeni do przeprowadzenia niektórych eksperymentów). Analiza powinna też uwzględnić dwa różne typy kompilacji: "-DCMAKE_BUILD_TYPE=Release" oraz "-DCMAKE_BUILD_TYPE=Debug". W raporcie można uwzględnić również wyniki własnych testów.

Wymagania techniczne

Rozwiązania w innej formie lub niespełniające tych wymagań nie będą przyjmowane:

- należy sklonować gitowe repozytorium https://www.mimuw.edu.pl/~aj334557/shamans/.git/
- rozwiązanie proszę dostarczyć jako wynik `git diff` wobec czubka gałęzi master w tym repozytorium
- `cmake . && make` powinno skompilować jego zawartość; tam jest przygotowane środowisko do kompilacji oraz podstawowe testy
- domyślnie cmake skonfiguruje kompilację z flagami do debugowania (bez optymalizacji i z dodatkowymi asercjami); żeby skompilować z flagami zorientowanymi na wydajność należy odpalić `cmake -DCMAKE_BUILD_TYPE=Release`
- dodając własne pliki, należy również uaktualnić plik `CMakeLists.txt`, aby `cmake . && make` dalej budowało rozwiązanie
- w `CMakeLists.txt` nie należy zmieniać flag kompilatora, ani struktury tego pliku;
- można (i należy) modyfikować plik "adventure.h"

- rozwiązanie powinno się kompilować na maszynie students bez użycia żadnych bibliotek oprócz pthreads i załączonej biblioteki threadpool
- kompilacja ma nie generować żadnych ostrzeżeń
- kod ma być sformatowany za pomocą skryptu `scripts/format.sh`
- skrypty `scripts/tidy.sh` oraz `scripts/lint.sh` nie powinny wyświetlać żadnych ostrzeżeń (do ich działania potrzebne są `clang-tidy` i `clang-format`). Ponieważ clang-tidy nie działa na students dopuszczalne jest pominięcie clang-tidy, o ile zadbasz o jakość kodu we własnym zakresie
- testy obecne w repozytorium mają przechodzić bez modyfikacji
- nazwy zmiennych, klas, komentarze, etc. proszę pisać w języku angielskim

Ocena

Oceniana będzie przede wszystkim poprawność. Rozwiązania ograniczające współbieżność ponad miarę będą uznane za niepoprawne. Oprócz poprawności będzie oceniana również wydajność oraz jakość kodu (w najmniejszym stopniu).

Za zadanie można uzyskać 10 pkt, które są podzielone w następujący sposób: 1. Rozwiązanie problemu pakowania smoczych jaj (3 pkt) 2. Rozwiązanie problemu odnalezienia drogi do wyroczni (3 pkt) 3. Rozwiązanie problemu poszukiwania najlepszego kryształu (2pkt) 4. Raport PDF z analizą czasu działania poszczególnych etapów (2 pkt).

Osiągnięcie dużego przyspieszenia zrównoleglając algorytmy jest zdaniem trudnym. Wymaga nie tylko ograniczenia synchronizacji między wątkami do minimum, ale również efektywnego wykorzystania cache procesora. Warto pamiętać, aby rozwiązanie:

- grupowało zadania przydzielane wątkom o ile to możliwe
- dbało o to, by różne wątki (na ile to możliwe) korzystały z odległych obszarów pamięci, aby ograniczyć false sharing
- dbało o to, by pojedynczy jak najwięcej sekwencyjnie czytał dane które są blisko siebie w pamięci, aby umożliwić efektywny prefetch

Dobrym wprowadzeniem na temat działania cache procesora jest ten artykuł: https://medium.com/software-design/why-software-developers-should-care-about-cpu-caches-8da04355bb8a

Mimo zastosowania się do powyższych wskazówek skalowalność algorytmów może nie być duża, np. jeśli algorytm działa tak efektywnie, że już przy małej liczbie wątków osiąga wąskie gardło na dostępach do pamięci. Dla ułatwienia, metody klas Egg, Crystal i GrainOfSand używają funkcji burden(), która wymusza dodatkowe obliczenia dla operacji dominujących w poszczególnych algorytmach. Ciekawym ćwiczeniem jest uruchomienie algorytmów z wykomentowaną zawartością funkcji burden i zrozumienie jak działają zrównoleglone algorytmy w takim scenariuszu, ale nie jest to wymaganą częścią raportu.

FAQ

Tutaj będą pojawiać się ciekawsze z zadanych pytań. Proszę je kierować pod adres aj334557@mimuw.edu.pl.~

Status przesłanego zadania

Status przesłanego zadania	Nie próbowano
Stan oceniania	Nie ocenione
Termin oddania	Monday, 3 February 2020, 00:00 AM
Pozostały czas	17 dni 7 godz.
Ostatnio modyfikowane	-
Komentarz do przesłanego zadania	▶ Komentarze (0)

Dodaj zadanie

Nie złożyłeś (-łaś) jeszcze zadania.

◄ Przykłady - synchronizacja w c++ Przejdź do...