Współbieżne ujecie problemu Collatza, raport

Kacper Kramarz-Fernandez (indeks: 429629)

11.02.2022

Celem raportu jest ukazanie różnic czasu działania programu wykonujacego współbieżne obliczenia przy zastosowaniu różnych metod synchronizacyjnych.

Opis zmian w istniejacych plikach:

- ./lib/infint/InfInt.h: - dodanie metody/gettera ‘InfInt::getVal‘ zwracajacej atrybut ‘val‘. Posłużyła ona w celu napisania funkcji hashujacej dla mapy wykorzystywanej w celach memoizacyjnych.
- ./sharedresults.hpp: - wyposażenie klasy w metody atomowego zapisu i odczytu (odpowiednio ‘SharedResults::atomicRead‘ i ‘SharedResults::atomicWrite‘) spamietanych wyników oraz zmiennych, z które one korzystaja, w tym bufora.
- ./main.cpp: - testowanie drużyn ‘TeamNewProcesses‘ i ‘TeamConstProcesses‘ zostało zakomentowane, z uwagi na niepełna implementacje kodu obu drużyn.
- ./collatz.hpp: - nagłówek ‘cassert‘ zamiast ‘assert.h‘ (deprecated)

Opis dodanych plików:

- ./my_collatz.hpp: - plik zawierajacy moja własna implementacje funkcji Collatza ‘t_myCalcCollatz‘, która przyjmuje jeden dodatkowy argument typu ‘SharedResults‘ i korzysta z niego w celu przyspieszenia obliczeń

Dodatkowe uwagi:

- ./new_process.hpp: - plik ten pozostał niewykorzystany, ponieważ w pełni wystarczyło mi zapisanie wywołań ‘fork‘ i ‘wait‘ w kodzie drużyn ”procesowych”,
- użyto standardu C++20 ze wzgledu na konieczność wykorzystania ‘std::shared_lock‘ i ‘std::unique_lock‘
- niestety nie było możliwe zrealizowanie testów na maszynie students, mimo ustawienia w wywołaniach ‘std::async‘ flagi ‘std::launch::deferred‘ i obniżenia rozmiaru stosu poprzez ”ulimit -s 128”.

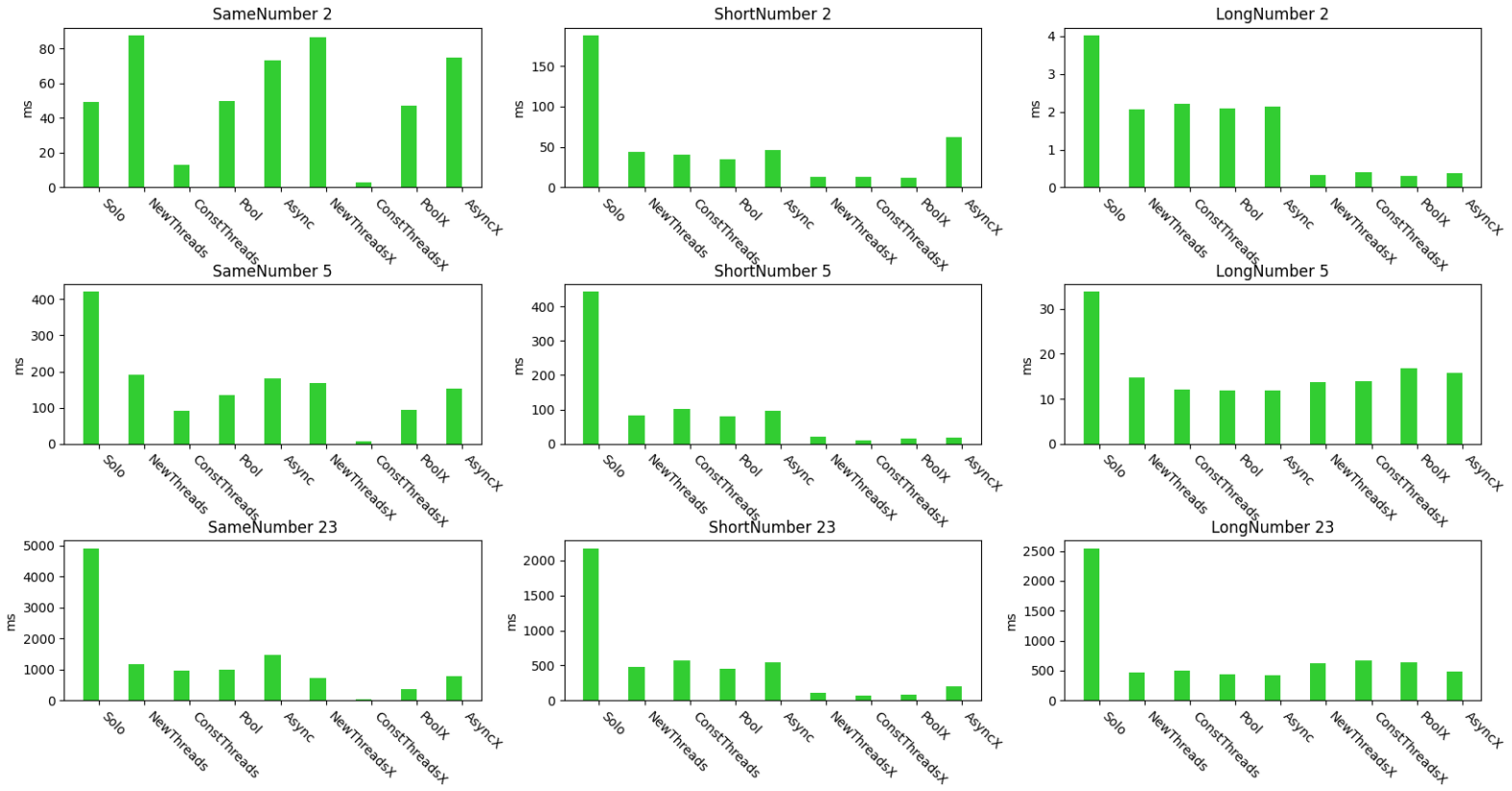

Figure 1: Graficzne porównanie czasów na moim laptopie

Struktura ‘SharedResults‘ jest wykorzystywana przez drużyny operujace na watkach (‘TeamNewThreads‘, ‘TeamConstThreads‘ i ‘TeamAsync‘) w celu przyspieszenia obliczeń. Przyspieszenie polega na spamietywaniu cześci wyników (memoizacji), tj. wpisywaniu obliczeń do mapy (‘std::unordered_map‘) w nadziei na to, że jeśli w obrebie danej drużyny to samo obliczenie bedzie zlecone w przyszłości, to jego wynik zostanie zczytany z mapy, zamiast bycia ponownie liczonym.

Synchronizacja struktury odbywa sie poprzez wykorzystanie mechanizmów "rygli" (ang. lock) (‘std::shared_lock‘ oraz ‘std::unique_lock‘) podlegajacych zasadom RAII. Zapewniaja one niezwykle wygodne rozplanowanie algorytmu synchronizacyjnego, jednak standard jezyka nie opisuje żywotności tych mechanizmów.

Drużyny operujace na procesach korzystać z SharedResults nie moga ze wzgledu na trudności, jakie sprawia umieszczenie realokowalnego kontenera z STL (w tym przypadku - mapy) w pamieci współdzielonej - nie można określić ograniczenia górnego wartości, do których dane wejściowe moga urosnać (ponieważ problem Collatza w dalszym ciagu pozostaje otwartym), wiec nie sposób jest przewidzieć pewna dana z góry pojemność dla mapy.

Spamietywanie wyników w obrebie drużyn procesowych (‘TeamNewProcesses‘ i ‘TeamConstProcesses‘) odbywa sie zatem inaczej niż dla watków, bardziej prymitywnie:

- na poczatku drużny wyliczaja $n := min(c, max(contestInput))$ wartości (gdzie $c > 0$ to pewna stała), dla których wyniki funkcji Collatza zapisuja w statycznym buforze rozmiaru $n + 1$,

- nastepnie, jeśli zlecone im zostana obliczenie z zakresu $[1, n]$, to zczytuja jego wyniki z bufora, a jeśli nie, to licza je w sposób normalny. Warto zaznaczyć, że jednak nawet przy liczeniu wartości wiekszych od n poczatkowa memoizacja pomogła, ponieważ mimo wszystko redukuje liczbe kroków potrzebnych w obliczeniach, choć nie całkowicie.

1. Opis środowiska - mój prywatny laptop:
- procesor Intel Core i7-10875H CPU @ 2.30GHz,
- 8 rdzeni,
- 16GB RAM.

2. Opis środowiska - serwer wydziałowy:
- Intel Xeon Processor (Skylake, IBRS) @ 2.09GHz
- 64 rdzenie,
- 344.22GB RAM.