Patryk Jędrzejczak

Problem Collatza – Raport

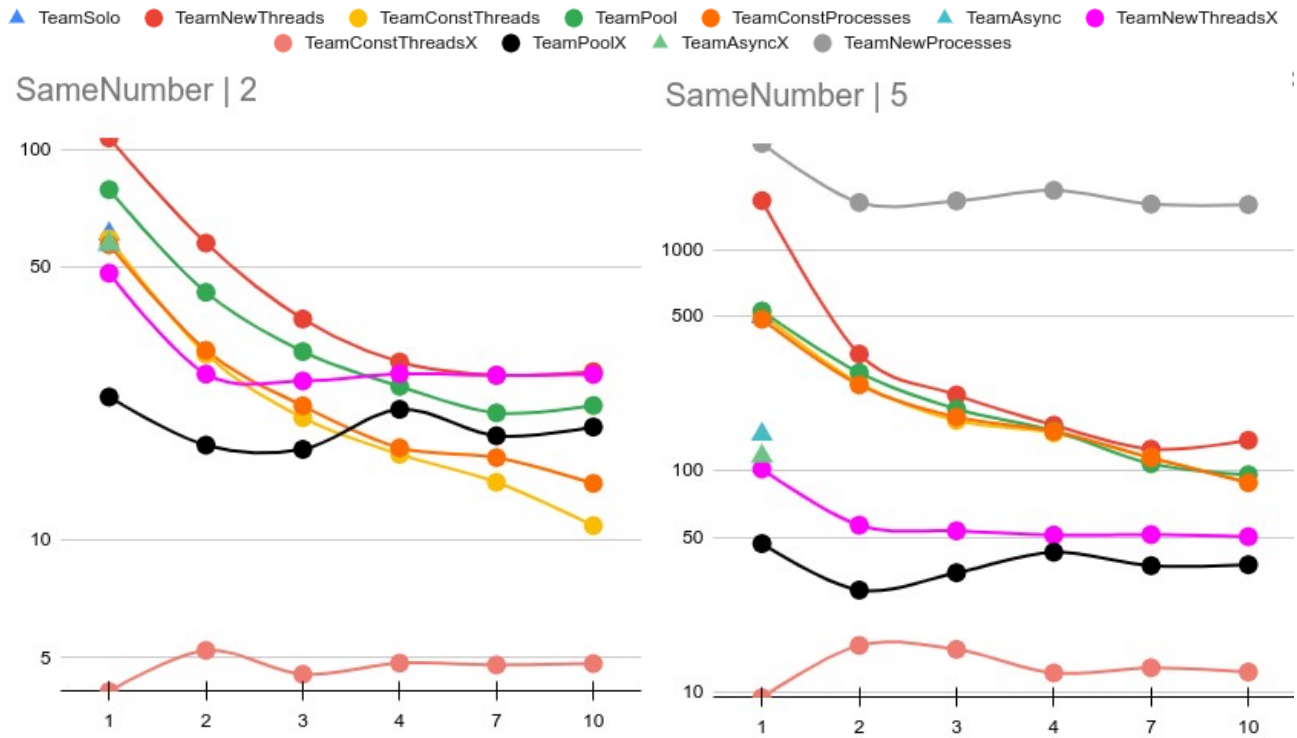
Zaimplementowałem grupy z wyjątkiem TeamNewProcessesX i TeamConstProcessesX, a dokładnie działają one tak samo jak TeamNewProcesses i TeamConstProcesses, korzystając z funkcji calcCollatz. Nie zakomentowałem testów dla tych dwóch grup, bo nie było potrzeby, skoro testy przechodzą.

Oto opis środowiska (fragment wyniku komendy lscpu), na którym testowałem rozwiązanie:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	12
On-line CPU(s) list:	0-11
Thread(s) per core:	2
Core(s) per socket:	6
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	158
Model name:	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

Zwracam uwagę, że podczas testowania na maszynie students TeamAsync i TeamAsyncX mogą zwracać bad_alloc.

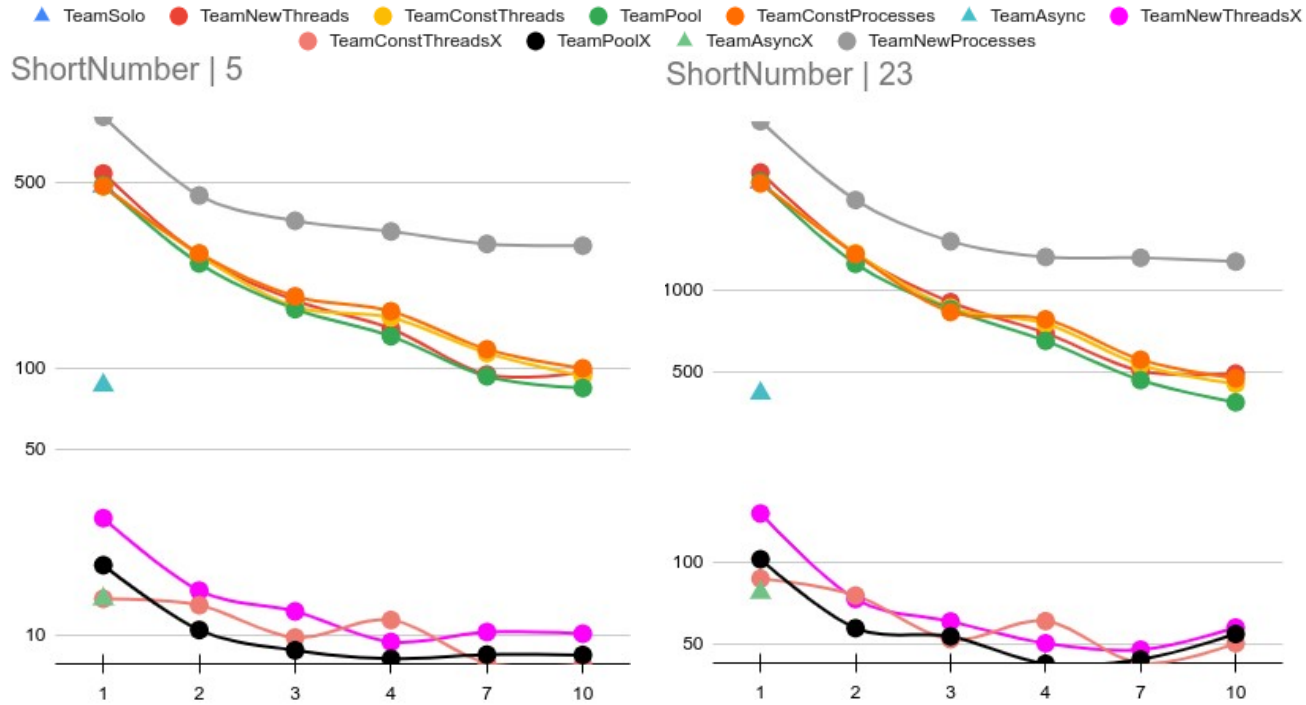
Poniżej przedstawiam wykresy pokazujące jak radziły sobie poszczególne zespoły w niektórych testach na moim komputerze.



CalcCollatzSoloTimer: 18,1228 us, n = 2998
18,1228us * 2998 = 54,332 ms

CalcCollatzSoloTimer: 82,4469 us, n = 5995
82,4469 us * 5995 = 494,269 ms

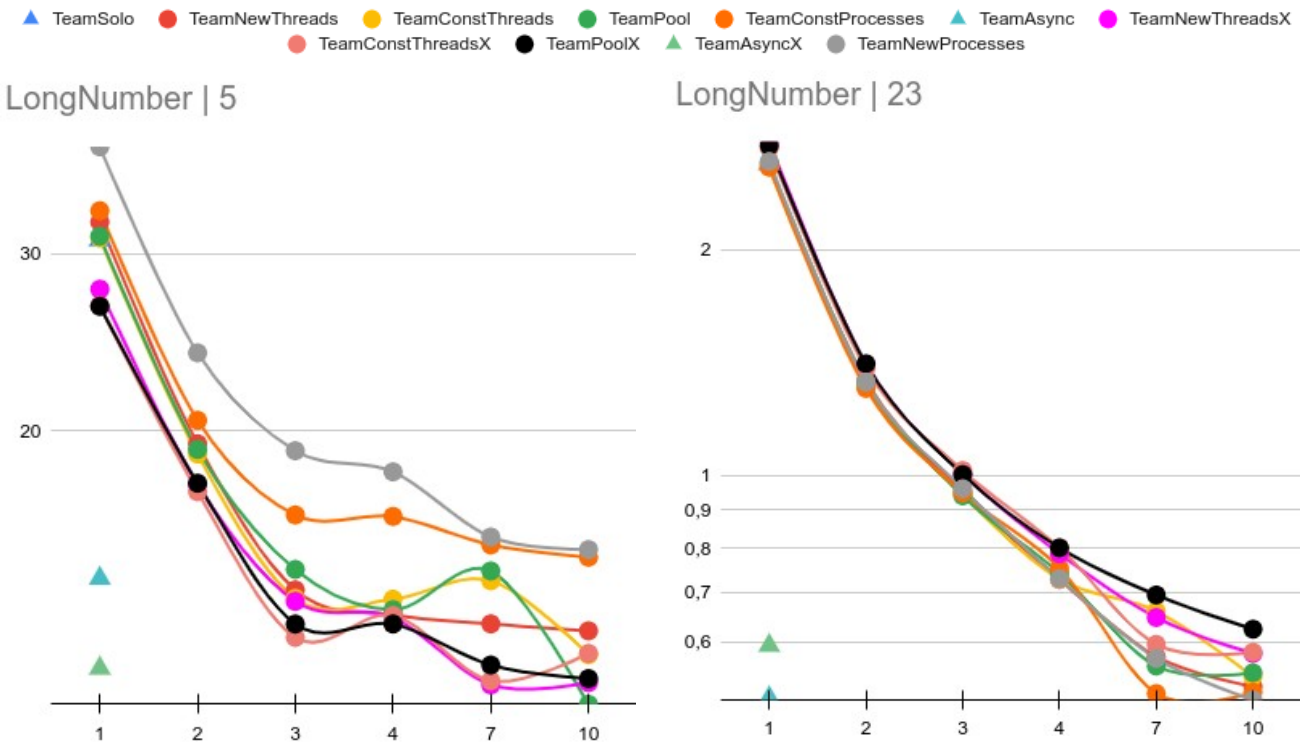
Oś pionowa reprezentuje czas (w ms), a oś pozioma liczbę wątków/procesów. Należy zwrócić uwagę na to, że czas jest w skali logarytmicznej, żeby mniej krzywych się na siebie nakładało. Na wykresie „SameNumber | 2” nie ma zespołu TeamNewProcesses, bo jego czasy były zbyt duże (dokładnie 0.7-1.2 s) i obniżyłyby jakość wykresu. Przedyskutujmy krótko wyniki. Drużyny TeamNewThreads, TeamConstThreads, TeamPool, TeamConstProcesses zachowują się zgodnie z oczekiwaniami. Wzrastająca liczba wątków prawie liniowo przyspiesza działanie programu, a przynajmniej dzieje się tak do 4 wątków/procesów. TeamNewThreads jest z tych drużyn najgorsza, co wynika z tego, że koszt tworzenia nowych wątków jest znaczący. TeamNewProcesses jest zdecydowanie najgorszą drużyną. Koszt tworzenia nowych procesów jest bardzo duży. Drużyny X zachowują się zauważalnie lepiej od swoich odpowiedników. Zdecydowanie najlepszą drużyną jest TeamConstThreadsX. Wynika to prawdopodobnie z implementacji. Z góry zadana praca oznacza brak potrzeby dodatkowych i często kosztownych operacji synchronizacyjnych na semaforach, muteksach i zmiennych warunkowych. Z implementacji wynika zapewne również zaskakująco szybkie wypłaszczenie (a nawet miejscowe wzrastanie) wykresów drużyn X. Zapamiętywanie wyników powoduje, że koszt liczenia wartości calcCollatz zostaje zdominowany przez koszt operacji synchronizacyjnych. Dość słabo radzą sobie drużyny Async, ale żeby stwierdzić dlaczego, trzeba by się zagłębić w implementację std::async. Warto zwrócić również uwagę, że wartości timerów CalcCollatzSoloTimer wymnożone przez ilości ich uruchomień są istotnie większe niż czasy działania zespołów współbieżnych. Przypatrzmy się teraz wynikom innych testów.



CalcCollatzSoloTimer: 0,8103 ms, n = 595
0,8103 ms * 595 = 482,1285 ms

CalcCollatzSoloTimer: 1,04421 ms, n = 2377
1,04421 ms * 2377 = 2482,087 ms

Co się tutaj zmieniło? Po pierwsze, TeamNewProcesses nie jest już dużo gorszy od innych. Koszt tworzenia nowych procesów nie jest już tak znaczący, bo koszt liczenia calcCollatz wzrósł. Drużyny Async również prezentują się przyzwoicie. Patrząc na drużyny „nie X”, można zauważyć, że rzeczywiście testowanie odbyło się w środowisku z procesorem z dwunastoma rdzeniami. Czasy dla 10 wątków są zauważalnie lepsze od czasów dla 7 wątków. TeamNewThreads nie jest już gorszy od innych zespołów „nie X”. Koszt tworzenia nowych wątków przestał być znaczący. TeamConstThreadsX stracił przewagę nad innymi zespołami X. Jest tak dlatego, że koszt operacji synchronizacyjnych nie dominuje już tak bardzo kosztu obliczania calcCollatz. Widać również ogromną różnicę w czasie pracy zespołów X i ich odpowiedników. Został nam jeszcze jeden test.



CalcCollatzSoloTimer: 4,41829 ms, n = 7
4m41829 ms * 7 = 30m928 ms

CalcCollatzSoloTimer: 104,163 ms, n = 25
104,163 ms * 25 = 2604.075 ms

Na wykresie „LongNumber | 23” zminiła się skala osi pionowej z na sekundy. Z wykresu „LongNumber | 5” trudno dużo wywnioskować, bo różnice w czasie są minimalne. Na pewno X zachowują się minimalnie lepiej dla małych ilości wątków/procesów. Dla większych ilości wygląda to dość losowo. Jednak wyraźnie widać nieznaczną przewagę zespołów wątkowych od procesowych. Ponadto, przestaje mieć znaczenie, czy zespół jest X czy „nie X”. Wynika to oczywiście ze specyfiki testu, w którym mamy tylko 7 liczb na wejściu. Dla tak małej ilości liczb, bycie zespołem współbieżnym oraz bycie zespołem X niewiele daje. Na wykresie „LongNumber | 23” doszło do jeszcze bardziej zaskakującej sytuacji – zespoły X są wolniejsze od swoich odpowiedników. Trzeba się dobrze przyjrzeć, żeby to zauważyć. Jest jasne, że zespoły „nie X” działają dla ilości wątków/procesów nie większej od 4 zgodnie (prawie tak samo szybko) i zespoły X działają zgodnie, jednak procesy X działają nieco wolniej. Test został tak dobrany, że koszt operacji przy używaniu klasy SharedResults jest większy niż zysk z jej używania. To na co można jeszcze zwrócić uwagę, to że przy tak małej ilości liczb koszt tworzenia nowych wątków/procesów jest nieistotny. Interesujące jest też zachowanie zespołów dla 7 i 10 wątków/procesów. Trudno jednak powiedzieć, czy te różnice w czasie są losowe (raz się lepiej ułożą przeploty, a raz gorzej), czy jednak z jakiegoś powodu niektóre zespoły lepiej się zachowują przy dużej ilości wątków/procesów. Trzeba by wiele razy powtórzyć test, aby to stwierdzić.

Podsumowując, wyniki są zadowalające i wiele można z nich wywnioskować. Widać, że zespoły X mają dużą przewagę (choć nie zawsze) oraz można zaobserwować wpływ tworzenia wielu procesów/wątków na czas działania programu.