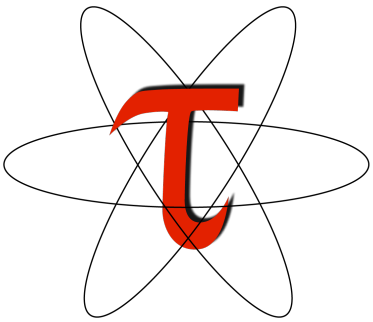
Algorytmy Równoległe

Oscar Teeninga
Projekt analizy wydajności laboratoriów

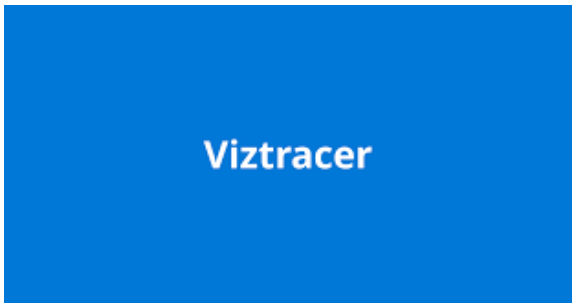
1. Narzędzia i sprzęt

Wykorzystany sprzęt to prywatny laptop z procesorem **Intel Core i9**, który dysponuje 8 rdzeniami i 16 wątkami. Do dyspozycji jest 16GB RAM. System operacyjny to MacOS 11.

Początkowo planowałem skorzystać z narzędzia **TAU** (Tuning and Analysis Utilities), który wymagał stosunkowo zaawansowanej konfiguracji. Udało mi się skompilować oraz uruchomić jedno laboratorium, natomiast była to analiza bardzo uboga, bardzo nieprzyjemnie graficznie zaprezentowana i wymagająca dużej konfiguracji podczas uruchomienia.

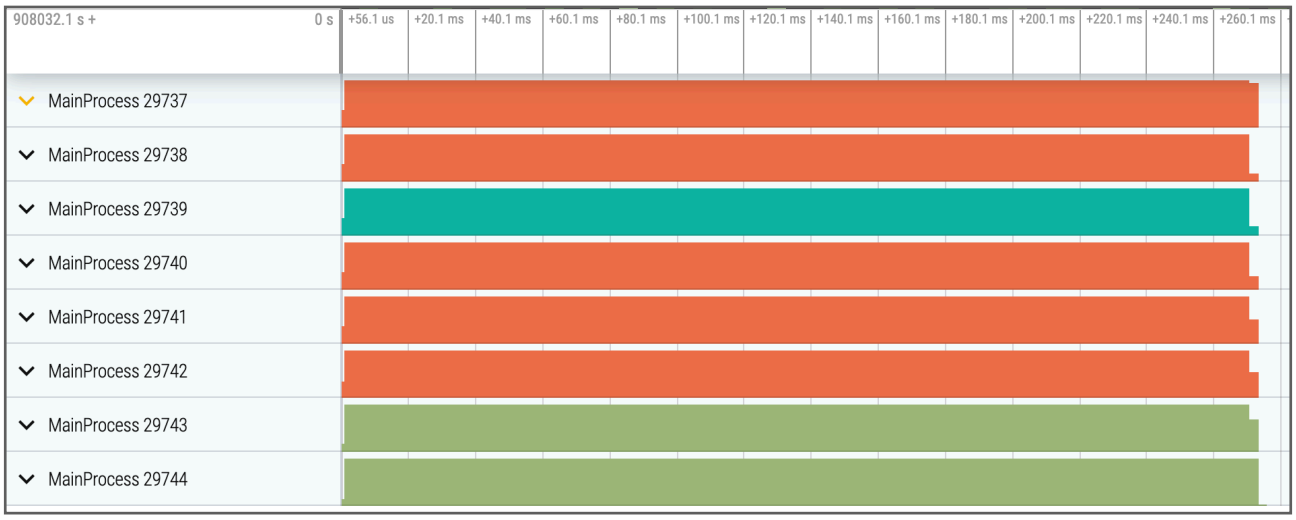


Ze względu na te okoliczności chciałem skorzystać z narzędzie wygodniejszego i łatwiejszego w obsłudze. Natknąłem się na bibliotekę **VizTracer**. Jest to narzędzie do rejestrowania/ debudowania/profilowania o niskim nakładzie pracy, które może śledzić i wizualizować kod wykonywany w interpreterze Pythona, umożliwiając graficzną reprezentację analizy wydajnościowej.



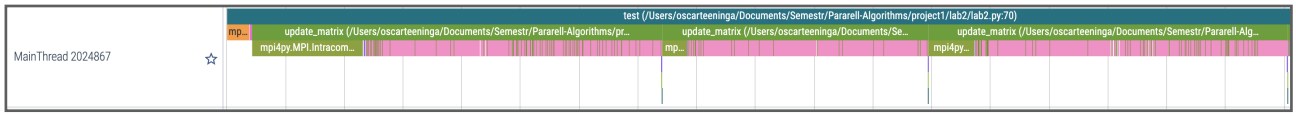
2. Analiza Laboratorium 2

Zadanie to polegało na wymyśleniu konwersji problemu w taki sposób, żeby dało się problem zrównoleglić. Mój pomysł polegał na obliczeniach z 4 elementów wokół, co wymagało dużej liczby komunikacji. Program uruchamiałem w ośmio-wątkowej wersji z komunikacją MPI.



Czasy działania każdego wątku

Zgodnie z wykresem, zauważyć można, że wątki są dobrze synchronizowane pod względem czasu wykonania. Nie ma sytuacji, że jeden opóźnia pozostałe i wszystkie wykonują się podobną ilość czasu. Oznacza to, że pod tym względem mój algorytm nie ma sobie nic do zarzucenia.



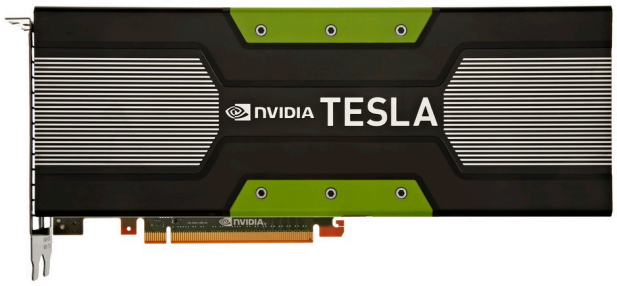
Czasy ewaluacji funkcji w jednym wątku brzegowym

Czasy ewaluacji funkcji w jednym wątku wewnątrz

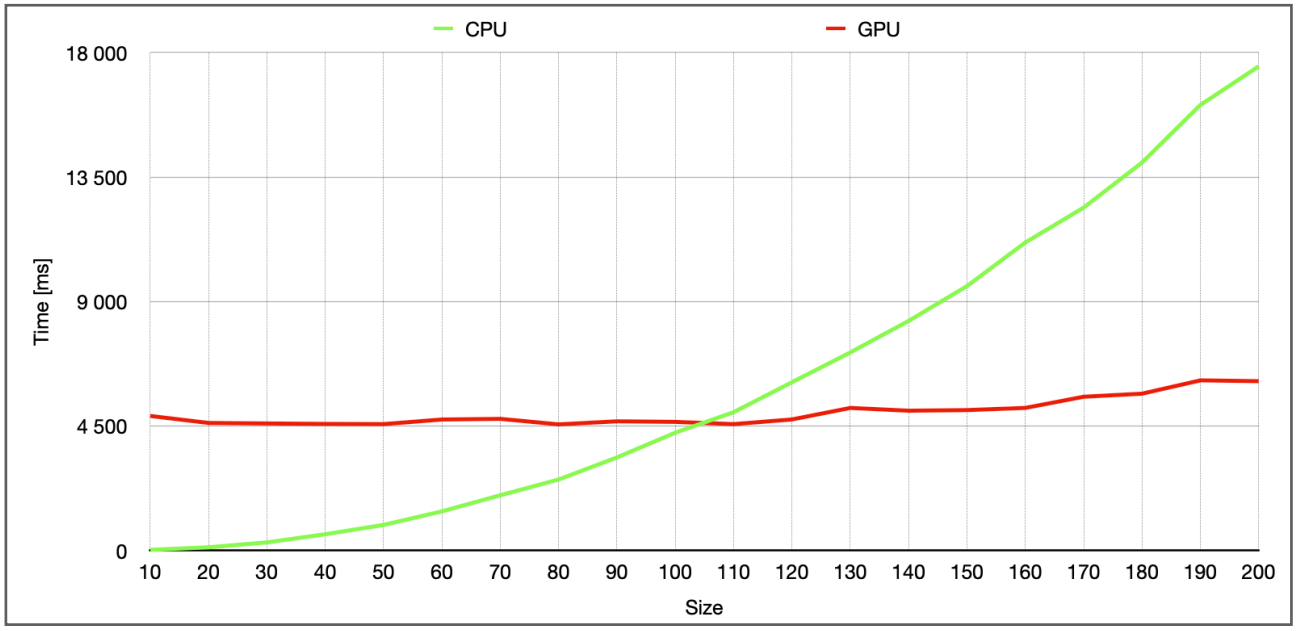
Komunikacja jest wymagana przed każdym rozpoczęciem aktualizacji macierzy potencjału. Z tego wynikają początkowe czasy (oznaczone kolorem jasno-zielonym) komunikacji podczas wysyłania. Jest to mankament, który jest objawem pomysłu algorytmu oraz fakt, że MPI nie przewiduje pamięci współdzielonej, aby poprawić ten aspekt należałoby zmodyfikować algorytmu od jego podstaw (matematyczne modyfikacje) lub technologię komunikacji (np. wątki i semafory gwarantujące synchronizację). Dodatkowo fajnym pomysłem jest wykonywanie tych obliczeń na GPU, które ze względu na przytłaczającą liczbę procesorów - są w stanie wykonywać operacje per-punkt, a więc można oprzeć się na operacjach atomowych, które zagwarantują synchronizację.

Algorytm należy poprawić w obecnej formie. Stwierdziłem, że najlepszą optymalizacją będzie przepisanie całego programu na program *CUDA* oraz uruchomienia go na GPU.

VS



Do wykorzystania miałem jednostki, które są zamontowane w Prometeuszu - NVIDIA Tesla K40d. Jest to bardzo mocny sprzęt, natomiast wydajność jednowątkowa mojego procesora w trybie boost również jest porównywalna do najmocniejszych jednostek dostępnych na rynku.

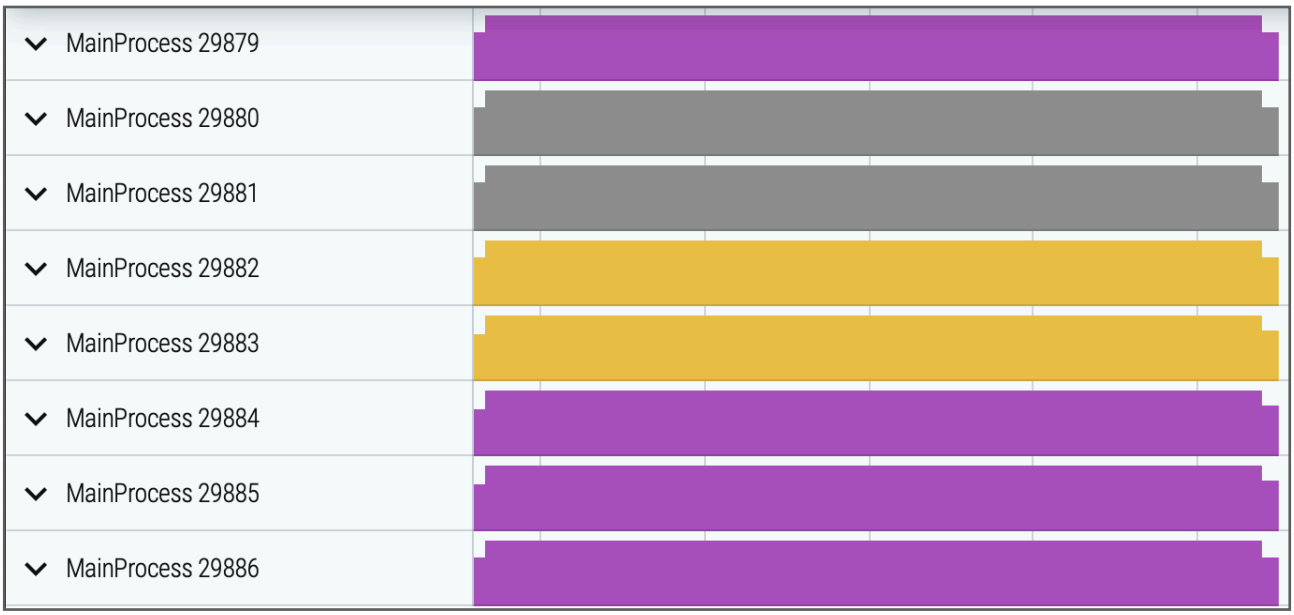


Porównanie czasów dla sekwencyjnego CPU oraz GPU

Cieżko dokonać analizy programu wykonywającego się na GPU, natomiast tutaj mamy porównanie dla tego samego problemu. Program napisałem w ramach przedmiotu Metody Programowania Równoległego. Jest to porównanie do rozwiązania sekwencyjnego, a więc teoretyczne różnica może okazać się zmniejsza, natomiast widać, że skalowalność rozwiązania GPU jest niesamowita względem GPU i MPI. Głównym narzędziem synchronizacyjnym jest operacja atomowa atomicAdd.

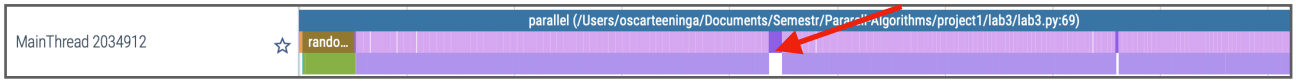
3. Analiza Laboratorium 3

To zadanie zostało przeze mnie zaimplementowane jedynie w najbardziej podstawowej wersji, która wystarczała na ocenę 3.0. Ze względu na to, analizy nie będzie zbyt wiele, a sam algorytm jest bardzo prosty. Korzystam z biblioteki MPI.



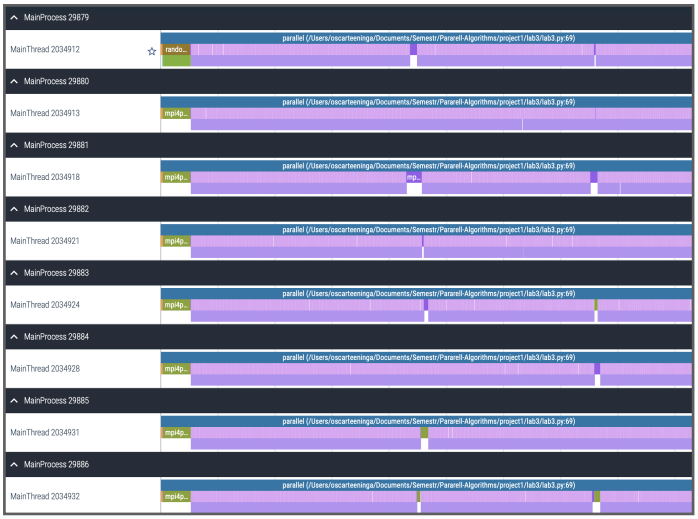
Czasy działania każdego wątku

Tutaj sytuacja powtarza się z laboratorium 2. Wątki wykonują się w tym samym czasie, nie widać również, żeby któryś zaczynał, bądź kończył się wcześniej lub później niż pozostałe. Oznacza to, że pod tym względem algorytm jest zsynchronizowany i nie jest wymaga tutaj żadna interwencja ani poprawka.



Czasy ewaluacji funkcji w jednym wątku

Czerwoną strzałką oznaczyłem fragment programu który jest wysyłaniem danych w MPI. Pozostałe (inne odcienie fioletowego) są obliczeniem wykonywanym na rzecz pojedynczej gwiazdy. Oznacza to, że algorytm jest bardzo dobrze zorganizowany, nakłady na komunikacje są niewielkie, a i każdy wątek poświęca niewielki odsetek swojego czasu na odebranie/wysłanie danych. Jedynym rozwiązaniem polepszającym działanie byłoby według mnie zmiana biblioteki na taką, która dysponuje pamięcią współdzieloną. Wówczas komunikacja zostałaby zredukowana do zerowych nakładów (ponieważ wysyłanych danych jest bardzo mało).



Czasy dla wszystkich wątków

4. Narzędzie do analizy wydajności dla języka Scala

Wykorzystywane wcześniej narzędzie VizTracer nie jest w stanie analizować programów uruchamianych w JVM, także musiałem zdecydować się na inne. Dlatego skorzystałem z alternatywnego rozwiązania VisualVM, który jest bardzo prosty w użytkowaniu.

5. Analiza Laboratorium 5

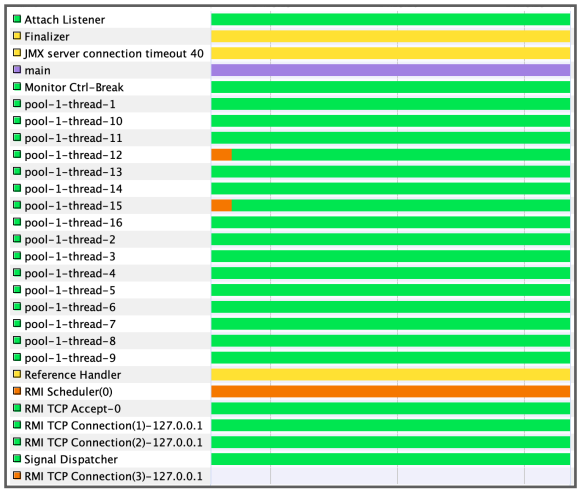
W przypadku tego zadania musiałem skorzystać z innego narzędzia, które opisałem podpunkt wyżej. Tutaj komunikacja była synchroniczna podczas zakończenia obliczeń - oznacza to, że spodziewany nakład na komunikację będzie niewielki.

Aktywność wątków

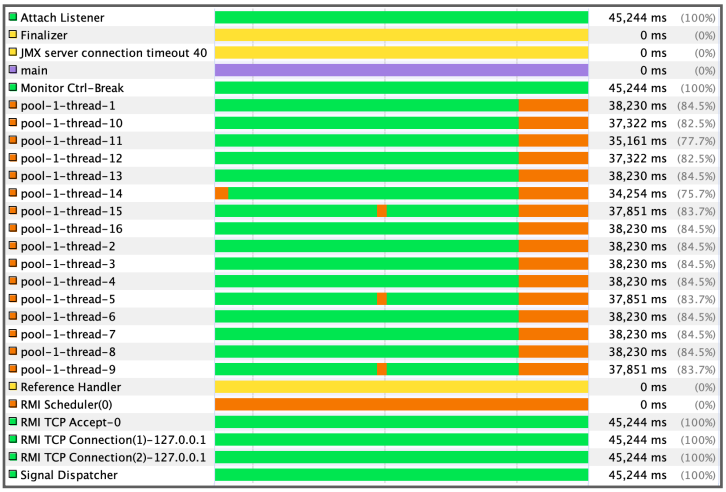
Jak widać, moje rozwiązanie nie działa równolegle, ale działa współbieżnie. Kolor zielony oznacza pracę, pomarańczowy oznacza oczekiwanie. Oznacza to, że należy zupełnie zmienić algorytm, ponieważ ten nie spełnia swojej funkcji. Dlatego napisałem od nowa cały program.

Początkowo usunąłem semafory, które synchronizują działanie. Nie dało to efektów, ponieważ każdy wątek oczekiwał na zakończenie poprzedniego. Można usprawnić to poprzez zaniechanie tego mechanizmu, natomiast problemem jest wówczas czas zakończenia działania algorytmu. Można to nadzorować poprzez VisualVM (czy wątki pracują), a jeżeli nie to zapisana odpowiedź jest tą aktualną.

Bardzo duże usprawnienie, które niestety okupione jest “manualną” potrzebą zamknięcia programu, który “nie wie” kiedy się skończy. Zapewne da się to zautomatyzować, ale jeżeli chodzi o nasza analizę nie ma to znaczenia.



Poprawiona wersja asynchroniczna



Poprawiona wersja z zakończeniem

6. Podsumowanie

Jak widać, udało się bardzo sprawnie odnaleźć bolączki moich implementacji oraz poprawienie nich. Szczególnie dobrze widać na laboratorium 5, które właściwie w momencie oddawania było nierównoległe, a jedynie współbieżne. Analiza pozwoliła znaleźć mankamenty oraz słabości rozwiązań, które pomogły usprawnić rozwiązanie postawionych w laboratoriach problemów.