NJP3 **Zadanie zaliczeniowe** *Cezary Bartoszuk*
#277617

Algorytm

W rozwiązaniu został użyty algorytm Cannona opisany tak, jak w załączonych do treści zadania materiałach.

Implementacja

Rozwiązaniem są dwie implementacje.

1. Oficjalne rozwiązanie (niestabilna implementacja) napisana w C++.
2. Testowe rozwiązanie (stabilna implementacja) jest napisana w C.

Jako, że oba wymienione wyżej algorytmy mnożą losowe macierze zakładam, że każdy węzeł losuje macierz dla węzła przesuniętego (tj. pomijam pierwszą fazę algorytmu Cannona ze względów wydajnościowych - dane losowane na wszystkich węzłach mają ten sam rozkład).

Implementacja testowa w C

Rozwiązywanie zadania postanowiłem rozpocząć od prostej implementacji w C zrównoleglonej za pomocą MPI. Do rozwiązania podproblemu mnożenia macierzy użyłem tracycyjnego algorytmu złożoności czasowej $\Theta(n^3)$. Algorytm działa poprawnie. Rozważałem też opcję zrównoleglenia tradycyjnego algorytmu za pomocą `OMP`, lecz po krótkim namyśle stwierdziłem, że najpewniej tak, czy inaczej nie uda mi się pobić szybkośći implementacji bibliotecznej. Postanowiłem więc jej użyć, co doprowadziło mnie do implementacji całości w C++.

Implementacja w C++

Implementacja w C++ oparta jest o zrównoleglony za pomocą `MPI` algorytm Cannona. W uzyskaniu

przyzwoitej implementacji pomogło mi użycie bibliotek `boost::mpi` oraz `boost::numeric`. Użycie biblioteki `boost` umożliwiło mi bardzo sprytne zagnieżdżenie algorytmów mnożenia macierzy oparte o model kompozytowy. Algorytm Cannona, jest parametryzowany podprocedurą mnożenia macierzy o tej samej strukturze. Dzięki temu można implementując zupełnie osobno mnożenie macierzy zrównoleglone za pomocą `OMP` bądź tradycyjnych wątków otrzymać algorytm zrównoleglony na dwu (lub większej ilości) warstwach.

Dodatkową generyczność algorytmu otrzymujemy dzięki dowolności użycia kontenera dla przechowywania macierzy. Do wyboru mamy dwa kontenery z biblioteki `boost::numeric` (konkretnie `bounded_array` oraz `unbounded_array`) oraz standardową implementację z bibioteki STL (`std::vector`).

Biblioteka `boost` umożliwia również parametryzowanie struktur danych dla macierzy funktorami określającymi kolejność elementów w kontenerze. Dwa, z których skorzystałem to `row_major` oraz `column_major` (które przechowują dane odpowiednio kolumnami - lub wierszami). Dzięki temu przechowując lewy wyraz produktu w macierzy uporządkowanej wierszami oraz prawy w macierzy uporządkowaje kolumnami jesteśmy w stanie skorzystać z cache'owania pamięci przez procesor podczas mnożenia.

Moim pomysłem na rozwijanie tej implementacji jest przeprowadzenie eksperymentu, który mógłby dowieść czy faktycznie przy użyciu 4 rdzeni, `OMP`/`boost::threads` oraz wsparcia ze strony `SIMD` nie jestem w stanie prześcignąć procedury bibliotecznej (dla odpowiednio dużej macierzy). Ostatnio czytałem o bardzo ciekawej strukturze danych z biblioteki standardowej szablownów w C++ jak `valarray`, która ma hintować kompilator o użuciu `SIMD` (klasa ta ma dodatkowo zaimplementowane wszystkie potrzebne operatory arytmetyczne i większość standardowych funkcji matematycznych dla przeprowadzania operacji wektorowych).

Wyniki eksperymentów

Niestety ze względu na potworny korek na nautilusie byłem w stanie jak dotąd odpalić zadanie jedynie dla 4 węzłów (16 rdzeni), które dla żądanej macierzy boku 2^{16} przekroczyło czas działania 8 godzin pracy każdego z procesorów. Jak tylko uzyskam jakieś ciekawe wyniki, podzielę się nimi z Państwem.

Testy

Obie implementacje doczekały się jak dotąd jedynie ręcznych testów. Planuję jednak implementację testu probabilistycznego polegającego na losowaniu z rozkładem jednostajnym α wektorów, którymi później przybliżać będę równość $C = A * B$ układem $\forall_x C * x = A * (B * x)$.

Przy dostatecznie dużej stałej α będę w stanie sprawdzić z wysokim prawdopodobieństwem czy mój algorytm działa poprawie w czasie $\Theta(\alpha n^2)$.