HPC - MPI

Tomasz Kępa
tk359746@students.mimuw.edu.pl

2 czerwca 2019

1 Struktura plików

Rozwiązanie podzielone jest na pliki w następujący sposób:

- Struktury danych:
  - *config.h* – struktura przechowująca sparsowane flagi przekazane do programu
  - *mpigroup.h* – struktura definiująca grupę obliczeniową (wszystkie procesy, grupy replikacyjne, warstwy)
  - *matrix.h matrix.cpp* – struktury przechowujące informacje o macierzy oraz macierze rzadkie i gęste
- *densematgen.h densematgen.cpp* – pliki dostarczone razem z treścią zadania służacego do generowania macierzy gęstej
- *utils.h* – mniej istotne pomocnicze funkcje
- *matrixmul.h matrixmul.cpp* – definicje głównych etapów obliczeń i główny program, implementacje w następujących plikach:
  - *initialization.cpp* – wczytywanie macierzy A, generowanie macierzy B, początkowa dystrybucja danych (jak dla $c = 1$)
  - *replication.cpp* – rozsyłanie danych między procesami z tych samych grup replikacyjnych, w przypadku algorytmu InnerABC dodatkowo wykonanie początkowego przesunięcia fragmentów macierzy A
  - *communication.cpp* – operacja shift-and-compute (jeden krok obliczeń)
  - *multiplication.cpp* – mnożenie macierzy i komunikacja z tym związana
  - *gathering.cpp* – zbieranie wyników

2 Struktury danych

Przed rozpoczęciem obliczeń wszystkie macierze są wyrównywane zerami, tak aby ilość wierszy i kolumn była podzielna przez ilość procesów.

2.1 Metadane macierzy

Najważniejsze informacje o macierzy przechowywane są w strukturze *MatrixInfo*. Zawiera ona wszystkie informacje potrzebne do zaalokowania odpowiedniej ilości pamięci przed odbiorem nieznanej macierzy i z tego powodu wysyłana jest zawsze przez wysłaniem macierzy.

2.2 Macierze rzadkie

Macierze rzadkie przechowywane są w strukturze *SparseMatrix*. Jest to dokładnie taki sam format jak format na wejściu, czyli CSR z trzema tablicami. Taki format został wybrany z kilku powodów:

- Nie trzeba tracić czasu na zmianę formatu podczas inicjalizacji
- Biblioteka MKL potrafi używać ten format (w wersji z czterema tablicami, ale przejście do tego fomatu nic nie kosztuje)
- Jest to format row-major, co jest korzystne dla ręcznej implementacji mnożenia macierzy

2.3 Macierze gęste

Macierze gęste przechowywane są w pojedynczej tablicy rozmiaru $rows \times cols$ w kolejności column-major. Taki formata został wybrany z następujących powodów:

- Jest to korzystna kolejność dla implemetacji lokalnego mnożenia macierz
- Procesy trzymają w swoich fragmentach całe kolumny oryginalnych macierzy, więc rozsyłanie i zbieranie macierzy jest dużo prostsze w takim formacie i nie wymaga wykonywania dodatkowych kopii (lokalnych) macierzy

3 Intesywność numeryczna

Oznaczenia:

- d – oznaczmy średnią ilość niezerowych elementów w jednym wierszu macierzy A
- e – ilość mnożeń macierzy A przez macierz B
- n – wymiary macierzy A, B i C

3.1 Operacje zmiennoprzecinkowe

Do policzenia jednej komórki macierzy C trzeba wykonać średnio d mnożeń i $d-1$ dodawań w jednym kroku. Zatem do policzenia macierzy C potrzebne jest $(2d-1) \cdot n^2 \cdot e$ operacji zmiennoprzecinkowych.

3.2 Transfery pamięci

Rozpatrzmy to podobnie jak poprzednio. W tym przypadku istotne staje się jak trzymana jest macierz rzadka A. Założę teraz, że jest to format CRS. W jednym kroku, do policzenia jednej komórki potrzebujemy odczytać jeden wiersz macierzy A i jedną kolumnę macierzy B. Jeśli chodzi o zapisy to potrzebujemy to zrobić to tylko raz w macierzy C.

Wczytanie wiersza macierzy A wymaga odczytania średnio $(4 \cdot 2 + 4 \cdot d + 8 \cdot d$ bajtów (2 el. tablicy trzymającej długości wierszy, d indeksów kolumn, d elementów macierzy A). W macierzy B czytamy średnio $8 \cdot d$ bajtów (d elementów). Na koniec jeszcze musimy wykonać jeden zapis w macierzy C

Razem daje to $16 + 20d$ bajtów na element macierzy C w jednym kroku.

W sumie policzenie całej macierzy C wymaga transferu $(16 + 20d) \cdot n^2 \cdot e$ bajtów.

4 Komunikacja

W obu algorytmach (o ile $c \neq 1$) procesy podzielone są w dwuwymiarową siatkę. Najpierw, wszystkie procesy podzielone są w grupy replikacyjne rozmiaru c, a pomiędzy tymi grupami, procesy o tych samych rangach wyznaczają warstwy. Warstw musi być oczywiście c.

W trakcie inicjalizacji koordynator wczytuje macierz A i wysyła odpowiednie fragmenty do wszystkich procesów. Następnie, procesy w tych samych grupach replikacji wymieniają między sobą swoje fragmenty

macierzy A (i C dla InnerABC). W trakcie obliczeń procesy wysyłają komunikaty tylko wewnątrz swojej warstwy. W trakcie końcowego zbierania danych komunikacja wygląda inaczej w obu algorytmach i szerzej opisana jest w części poświęconej optymalizacjom.

5 Optymalizacje

5.1 MPI

Przy implementacji największy nacisk położyłem na efektywne wykorzystanie możliwości danych przez MPI:

- Większość komunikacji jest asynchroniczna i przepleciona z obliczeniami.
- Początkowe rozsyłanie macierzy przez koordynatora odbywa się tabela po tabeli. Po wysłaniu jednej tabeli mechanizmem *scatter* (asynchronicznie), koorydator od razu zabiera się za przygotowanie kolejnej.
- Przesyłanie macierzy w trakcie replikacji w grupie replikacyjnej odbywa się przez customowe komunikatory za pomocą mechanizmu *broadcast*.
- Przesyłanie macierzy wewnątrz warstw (*shift*) w trakcie obliczeń przeplecione jest z lokalnie wykonywanym mnożeniem.
- Zbieranie macierzy C wewnątrz grupy replikacyjnej odbywa się za pomocą mechanizmu *reduce* (tylko w InnerABC), operacja sumy. Zbieranie macierzy C w warstwie zerowe to zwykłe *gather*.
- Zliczanie liczby elementów większych od danego – w przypadku ColumnA jest to zwykłe *reduce* po wszystkich procesach, w przypadku InnerABC macierz C najpierw zbierana jest do warstwy 0 i dopiero w tej warstwie jest wykonywana redukcja

Macierz A dzielona jest między procesy po równej ilości kolumn/wierszy. Innych opcji nie rozważałem ponieważ dokładnie tak to było opisane w artykule opisującym algorytmy.

5.2 Flagi kompilacji

Najlepsze wyniki uzyskałem na kompilatorze **Cray**. Włączyłem następujące opcje:

- *-O3* – domyślne optymalizacje
- *-hfp3* – optymalizacje operacji zmiennoprzecinkowych
- *-h vector3* – wektoryzacja operacji wykonywanych w pętlach
- *-hipa4 -hwp -h pl=...* – automatyczne inline'owanie funkcji

5.3 Lokalne mnożenie

Formaty macierzy zostały dobrane tak, aby było jak najmniej skoków w tablicach reprezentujących macierze. Zatem macierz A jest row-major, macierze B i C są column-major.

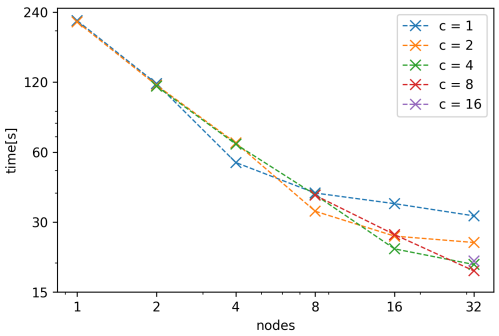
5.4 OpenMP

Wykonałem próbę użycia równoległej implementacji MKL (używającej OpenMP). Wyniki były znacznie gorsze niż przy wykorzystaniu wersji sekwencyjnej i poleganiu na MPI, dlatego nie próbowałem używać OpenMP w mojej implementacji.

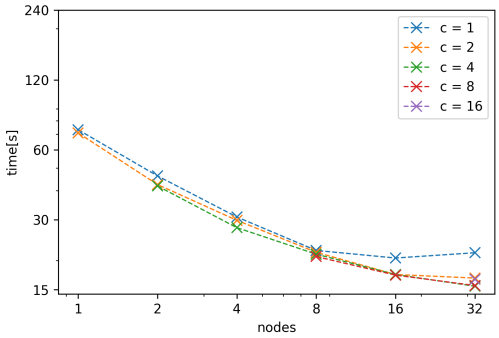
6 Skalowalność

6.1 Silna skalowalność

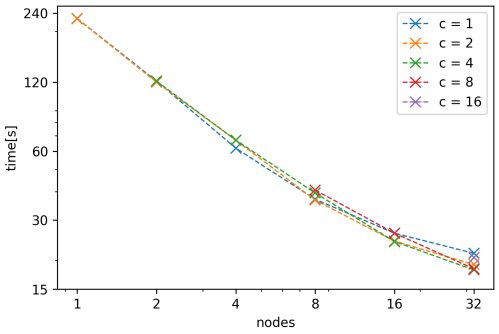
Testy silnej skalowalności zostały przeprowadzone na dwóch macierzach A. Pierwsza miała $n = 50000$ wierszy i $d = 1000$ wartości w każdym wierszu. Druga – $n = 100000, d = 100$. W pierwszy przypadku mnożenie było wykonywane $e = 5$ razy, w drugim $e = 2$. W obu przypadkach czas wczytywania macierzy wynosił ok. $9.5s$ Wyniki zostały przedstawione na rys. 1 i rys. 2.



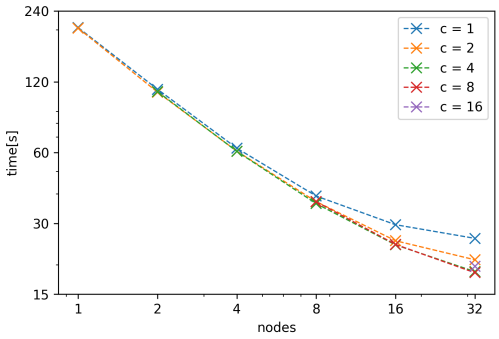
(a) ColumnA, no MKL



(b) ColumnA, MKL



(c) InnerABC, no MKL

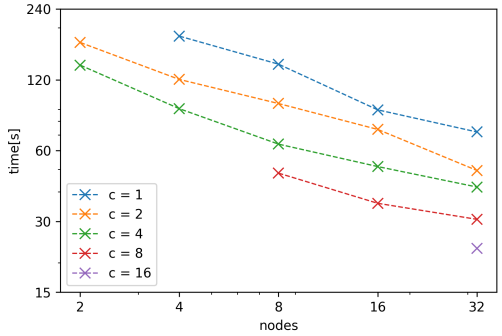


(d) InnerABC, MKL

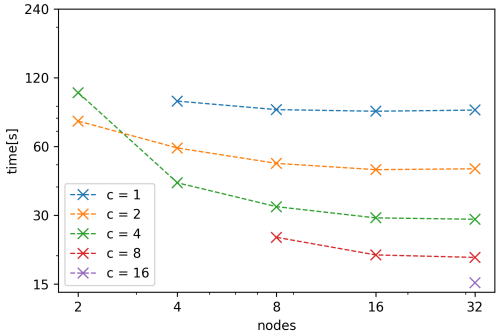
Rysunek 1: Silna skalowalność. Czasy dla $n = 50000, d = 1000, e = 5$, tasks-per-node $= 24$

6.2 Słaba skalowalność

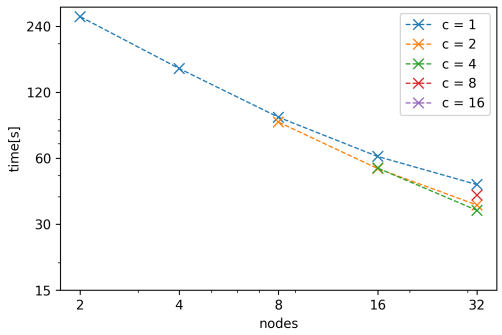
Testy silnej skalowalności zostały przeprowadzone na macierzy $n = 100000, d = 100$. Zmienna była liczba wykonywanych mnożesz i wynosiła $e = nodes \times 5$. Parametr c został ustalony na $c = 2$ Czas wczytywania macierzy wynosił ok. $9.5s$ Wyniki zostały przedstawione na rys. 3.



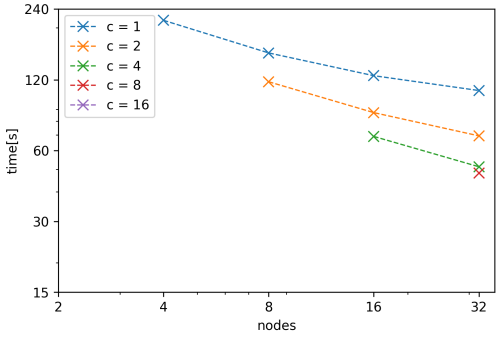
(a) ColumnA, no MKL



(b) ColumnA, MKL

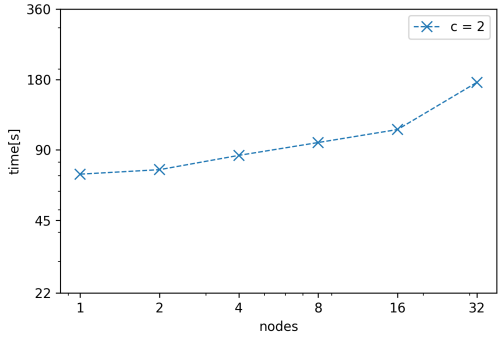


(c) InnerABC, no MKL

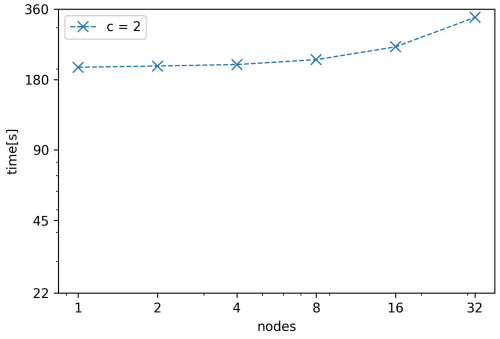


(d) InnerABC, MKL

Rysunek 2: Silna skalowalność. Czasy dla $n = 100000, d = 100, e = 2$, tasks-per-node $= 24$



(a) ColumnA, MKL



(b) InnerABC, MKL

Rysunek 3: Słaba skalowalność. Czasy dla $n = 50000, d = 1000, e = nodes \times 5$, tasks-per-node $= 24$