Aplikacja równoległa MPI rozwiązująca układ równań liniowych z wykorzystaniem dekompozycji LU

1. Cel projektu

Celem projektu było wykonanie aplikacji umożliwiającej rozwiązywanie układu równań liniowych. Do znalezienia wektora rozwiązań układu równań program miał wykorzystywać rozkład LU, którą powinien sam generować na podstawie początkowych danych układu.

Podstawowym założeniem projektowym stało się zastosowanie w tym przypadku obliczeń równoległych.

2. Analiza problemu matematycznego

Rozpatrujemy układ równań:

$$Ax = b$$

zapisany w postaci macierzowej:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}$$

Macierz współczynników A oraz wektor wyrazów wolnych są dane, natomiast wartościami szukanymi są elementy wektora niewiadomych x.

2.1. Dekompozycja LU

Macierz A możemy zapisać jako iloczyn:

$$A = L \cdot U$$

Gdzie macierze L oraz U są zdefiniowane w sposób ogólny następująco:

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ l_{2,1} & 1 & 0 & \dots & 0 & 0 \\ l_{3,1} & l_{3,2} & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ l_{n-1,1} & l_{n-1,2} & l_{n-3,3} & \dots & 1 & 0 \\ l_{n,1} & l_{n,2} & l_{n,3} & \dots & l_{n,n-1} & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,1} & \dots & u_{1,n-1} & u_{1,n} \\ 0 & u_{2,2} & u_{2,3} & \dots & u_{2,n-1} & u_{2,n} \\ 0 & 0 & u_{3,3} & \dots & u_{3,n-1} & u_{3,n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & 0 & \dots & 0 & u_{n,n} \end{bmatrix}$$

Przedstawiony poniżej algorytm rozdzielania macierzy A na L oraz U działa „w miejscu” – po jego zakończeniu macierze L oraz U będą się znajdować w tym samym miejscu w pamięci, gdzie wcześniej była przechowywana macierz A. W macierzy L istotna jest tylko część, znajdująca się poniżej głównej przekątnej. Elementy, znajdujące się na niej wynoszą zawsze 1. Macierz U zawiera natomiast ważne elementy na głównej przekątnej i ponad nią. Macierz A’, powstała z A, wygląda następująco:

$$A' = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{3,n-1} & u_{1,n} \\ l_{2,1} & u_{2,2} & u_{2,3} & \dots & u_{3,n-1} & u_{2,n} \\ l_{3,1} & l_{3,2} & u_{3,3} & \dots & u_{3,n-1} & u_{3,n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ l_{n-1,1} & l_{n-1,2} & l_{n-1,3} & \dots & u_{n-1,n-1} & u_{n-1,n} \\ l_{n,1} & l_{n,2} & l_{n,3} & \dots & l_{n,n-1} & u_{n,n} \end{bmatrix}$$

Poniżej znajduje się wersja zorientowana kolumnowo algorytmu rozkładu LU macierzy w miejscu.

```
Dla każdej kolumny macierzy A k=1:n-1 wykonuj:
    # normalizacja kolumny k
    Dla każdego wiersza macierzy A s=k+1:n wykonuj:
        a[s][k] = a[s][k] / a[k][k]
    # modyfikacja elementów, które leżą równocześnie poniżej
    # kolumny j oraz poniżej wiersza i
    Dla każdej kolumny macierzy A j=k+1:n wykonuj:
        Dla każdego wiersza macierzy A i=k+1:n wykonuj:
            a[i][j] = a[i][j] - a[j][k]*a[k][j]
```

2.2. Układ równań liniowych

Po rozłożeniu macierzy A na macierze L oraz U, układ równań liniowych można zapisać jako:

$$L \cdot U \cdot x = b$$

Wektor niewiadomych może zostać obliczony w dwóch krokach:

$$\begin{cases} L \cdot y = b \\ U \cdot x = y \end{cases}$$

Dzięki tej metodzie rozwiązanie układu równań jest prostsze, gdyż postać macierzy L i U pozwala stosunkowo łatwo wyliczać wartości wektorów y oraz x. Algorytm składa się z dwóch części:

1. podstawienie w przód:

$$y_1 = b_1$$

$$y_i = b_i - \sum_{j=1}^{i-1} l_{i,j} y_j \ , \ \ dla\ i = 2,3,\dots,n$$

2. podstawienie wstecz:

$$x_n = \frac{y_n}{u_{n,n}}$$

$$x_i = \frac{1}{u_{i,i}}\left(y_i - \sum_{j=i+1}^{n} u_{i,j}x_j\right), \quad dla\ i = n-1, n-2, \ldots, 1$$

Aby zaoszczędzić miejsca w pamięci, wektor pomocniczy y można przechowywać w tym samym miejscu, co wektor x., gdyż podczas ich obliczania żadne potrzebne dane nie zostaną nadpisane przed ich użyciem.

3. Podejście równoległe

Algorytmy równoległe pozwalają wykonywać wiele operacji w jednej chwili. Dzięki zrównolegleniu czynności mogą wymagać krótszego czasu na realizację zadanego problemu niż algorytmy sekwencyjne.

Aby skorzystać z tej zalety, można spróbować przekształcić algorytm sekwencyjny w równoległy. Jest to możliwe, jeżeli uda się wyodrębnić w nim czynności, które nie zależą od siebie. Wówczas mogą zostać one wykonane równocześnie bez problemu wzajemnego oczekiwania na siebie. Większość algorytmów sekwencyjnych nie da się całkowicie zrównoleglić ze względu na zależności, występujące między kolejnymi krokami, dlatego ulepszone przepisy będą posiadać zarówno części równoległe, jak i szeregowe.

Czas wykonywania algorytmu równoległego może zostać zdefiniowany jako:

$$T(n,p) = T_s(n) + \frac{T_p(n)}{p} + k(n,p)$$

gdzie:

$T_S(n)$ – czas wykonywania części sekwencyjnej;
$T_p(n)$ – czas wykonywania części równoległej w sposób sekwencyjny;
$k(n, p)$ – czas zużyty na synchronizację i komunikację;
n – rozmiar problemu;
p – ilość procesorów.

Maksymalne przyspieszenie, jakie może zostać osiągnięte dzięki równoległości, określa prawo Amdahla:

$$S(p) = \frac{1}{T + \frac{1-T}{p}}$$

gdzie:

T – czas wykonywania części nie dającej się zrównoleglić;
$1-T$ – czas wykonywania części dającej się zrównoleglić;
p – liczba procesorów.

Wynika z niego, że nawet przy zastosowaniu bardzo dużej ilości procesorów zysk z obliczeń równoległych może nadal być niewielki, jeśli część szeregowa będzie wystarczająco duża. Jednak każdy wystarczająco duży problem może być efektywnie zrównoleglony, o czym mówi prawo Gustafsona.

Przyspieszenie absolutne algorytmu równoległego względem sekwencyjnego można opisać jako:

$$S_p = \frac{T_S}{T_p}$$

gdzie:

T_S – czas wykonywania się algorytmu sekwencyjnego;

T_P – czas wykonywania się algorytmu równoległego przy zastosowaniu p procesorów.

Inną miarą może być wydajność, definiowana dla algorytmu realizowanego na p procesorach jako:

$$E_p = \frac{S_p}{p} = \frac{T_S}{p \cdot T_p}$$

Wartość ta przyjmuje zazwyczaj wartości z przedziału od 0 do 1. Obrazuje ona, jak dobrze jest wykorzystana moc obliczeniowa użytych procesorów w stosunku do czasu, zmarnowanego na komunikację i synchronizację.

Przedstawiony w poprzednim rozdziale algorytm rozkładu LU może zostać zrównoleglony przy pomocy dekompozycji danych – podziału obliczanych obszarów na podobszary, dla których zostaną przydzielone osobne procesory. Podczas realizacji obliczeń jest konieczna wymiana danych. W przypadku uruchamiania programu w systemie wielokomputerowym komunikacja zajmuje niepomijalny czas z punktu widzenia czasu jego wykonania, jednak zależy to m.in. od relacji między szybkością obliczeń w jednostkach obliczeniowych a szybkością przesyłu danych między nimi.

Rozkład LU można podzielić na dwa kroki, przeprowadzane w każdej iteracji dla kolumn:

- ✓ normalizacja kolumny k-tej;
- ✓ aktualizacja wartości kolumn, leżących na prawo od k-tej kolumny i równocześnie poniżej k-tego wiersza.

Pierwszy krok jest wymagany w celu możliwości obliczania nowych wartości kolumn w kroku drugim, które z kolei mogą być przetwarzane niezależnie od siebie. Dlatego w przypadku zastosowania p procesorów po zrealizowaniu pierwszego kroku przez procesor P_1, drugi krok może być realizowany na procesorach P_2 do P_p, po otrzymaniu przez nie kolumny od procesowa P_1. Zakładając rozmiar macierzy A jako n x n, każdy z p-1 procesorów będzie miał do przetworzenia $\frac{n}{p-1}$ kolumn.

Możliwe do zrealizowania byłoby równoległe przetwarzanie normalizowanej k-tej kolumny. Jednak przyniosłoby to wymierny zysk dopiero w przypadku bardzo dużych macierzy, z powodu liniowej złożoności tego problemu dla k-tego kroku. Dlatego dużo bardziej atrakcyjne okazało się skupienie na zrównolegleniu kroku drugiego, który ze względu na iterowanie zarówno po wierszach, jak i kolumnach posiada złożoność kwadratową.

Drugi etap zadania, czyli liczenie wektorów y oraz x jest trudny do zrównoleglenia. Do obliczenia wektora x jest potrzebny cały znany wektor y ze względu na używanie zarówno podstawień wprzód jak i wstecz. Same natomiast kolejne wartości obydwu wektorów zależą od wartości poprzednich, co również nie pozwala na oddelegowanie ich obliczania do pracujących równolegle jednostek.

4. Opis wykorzystanych technologii

Projekt został wykonany w języku C. Równoległość w aplikacji została zrealizowana przy pomocy biblioteki MPICH-3.2, która implementuje standard Message Passing Interface(MPI). W projekcie

została wykorzystana także pakiet MPE2-2.4.9b, który pozwolił nam wykonać wizualizację tego, jak pracuje nasza aplikacja w trybie równoległym. W celu ułatwienia współpracy grupowej nad projektem został użyty system kontroli wersji Git.

5. Struktura projektu

Projekt składa się z dwóch części: sekwencyjnej oraz zrównoleglonej. Są one jednakże umieszczone w jednolitej strukturze ze względu, iż spora część kodu jest dla nich wspólna, zaś one posiadają jednolity interfejs z punktu widzenia kodu, który je wykorzystuje.

W skład projektu wchodzą następujące pliki:

- /include/
  - *equations.h* – definicja struktury układu równań oraz deklaracje metod, służących do jego rozwiązania;
  - *LU.h* – definicje struktur oraz deklaracje metod, służących do obsługi dekompozycji LU; znajduje się tam jednolity interfejs dla obydwu strategii: sekwencyjnej oraz równoległej;
  - *matrix.h* – definicja struktury macierzy oraz deklaracje metod, służących do pracy z nią;
  - *mpe_utils.h* – stałe identyfikatorów, wykorzystywane do generowania raportów graficznych przy pomocy MPE;
  - *vector.h* – definicja struktury wektora oraz deklaracje metod, służących do operacji na niej;
- */resources/*
  - *A.txt* – macierz współczynników A_1 o wymiarach 400 x 400;
  - *A2.txt* – macierz współczynników A_2 o wymiarach 6 x 6;
  - *b.txt* – wektor b_1 wyrazów wolnych równania dla macierzy A_1, o wymiarach 400 x 1;
  - *b2.txt* – wektor b_2 wyrazów wolnych równania dla macierzy A_2, o wymiarach 400 x 1;
- */src/*
  - *equations.c* – funkcje służące do rozwiązywania układów równań;
  - *LU.c* – funkcje służące do sekwencyjnego rozkładu LU;
  - *LUmpi.c* – funkcje służące do równoległego rozkładu LU;
  - *main.c* – główny plik programu;
  - *matrix.c* – funkcje służące do operowania na macierzach;
  - *vector.c* – funkcje służące do operowania na wektorach;
- *.gitignore* – lista lokalizacji, któ®e mają być ignorowane przez system kontroli wersji;
- *makefile* – plik programu MAKE, który pozwala budować aplikację, w zależności od podanych ustawień;
- *README.md* – podstawowe informacje o projekcie;

6. Działanie aplikacji

Kod programu dzieli się na dwie części: wykonywaną na procesorze P_1, pełniącym funkcję centralnego serwera dla aplikacji oraz drugą, realizowaną na procesorach od P_2 do P_p. Jednostką P_1 jest ta, na której program został fizycznie uruchomiony przez użytkownika.

Na początku, po uruchomieniu na procesorze P_1 następuje wczytanie macierzy A oraz wektora b, Można te zmienne przekazać jako parametry programu. W innym przypadku zostaną ustawione wartości domyślne. Następnie procesor P_1 wysyła do pozostałych informacje o wymiarach macierzy A, determinujących również wymiar wektora b. Kolejna instrukcja to uruchomienie procedury *solve()*,

której zadaniem jest rozwiązanie układu równań. Po dokonaniu tego na procesorze P_1 następuje zapisanie wyników do plików oraz wyświetlenie informacji o działaniu programu.

Pierwszą częścią metody *solve()* jest wywołanie metody *decompose()*, odpowiadającej za dokompozycję LU na każdym procesorze. Następnie procesor P_1 oblicza wektory y oraz x.

Metoda *decompose()* iteruje po k-tej kolumnie, począwszy od pierwszej, a skończywszy na ostatniej. Dla każdej wykonywane są dwa kroki. Pierwszy ma miejsce na procesorze P_1. Następuje tam normalizacja kolumny k-tej, która jest następnie wysyłana do wszystkich pozostałych procesorów.

Krok drugi to aktualizacja kolumn, wykorzystująca wartości kolumny k-tej. Najpierw procesor P_1 przekazuje wartości kolumn do obliczenia procesorom od P_2 do P_p. Następnie czeka na odebranie od nich obliczonych wyników. Nie jest ważna kolejność, w jakiej będzie on odbierał dane, można jednak przypuszczać, iż w przypadku sprzętu o zbliżonych parametrach krótsze kolumny zostaną przetworzone szybciej. Kolumny, znajdujące się bliżej k-tej pozycji są dłuższe, a więc będą liczone szybciej. Podział kolumn między procesory został więc dokonany w sposób równomierny tak, aby procesor P_z otrzymał do obliczenia co z-tą kolumnę. Procesory od P_2 do P_p odbierają kolumnę, aktualizują jej wartość i zwracają ją do procesora P_0.

W przypadku rozwiązania sekwencyjnego nie występuje podział na procesory, a cały algorytm jest wykonywany na jednej jednostce, zgodnie z algorytmem opisanym w rozdziale 2.

7. Testowanie

Program został przetestowany w środowisku pracowni komputerowej dla dwóch układów równań.

7.1. Mały układ równań

Jednym z przygotowanych przez nas układów był zawierający sześć równań. Poniżej są przedstawione macierz A oraz wektor b.

$$A = \begin{bmatrix} 25 & 94 & 10 & 54 & 64 \\ 60 & 32 & 74 & 12 & 39 \\ 62 & 5 & 97 & 46 & 38 \\ 51 & 20 & 84 & 67 & 15 \\ 6 & 41 & 78 & 94 & 38 \end{bmatrix}$$

$$b = \begin{bmatrix} 14 \\ 56 \\ 91 \\ 41 \\ 31 \end{bmatrix}$$

Program dokonał rozkładu macierzy A na dwie macierze: L oraz U.

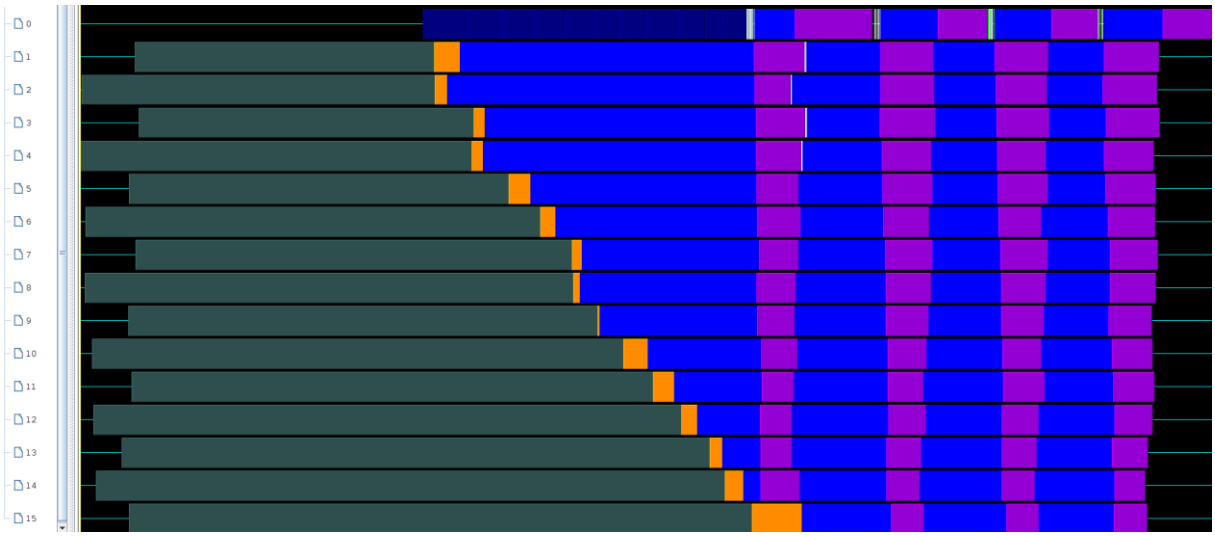
$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2.4 & 1 & 0 & 0 & 0 \\ 2.48 & 1.178306 & 1 & 0 & 0 \\ 2.04 & 0.88719 & 1.448319 & 1 & 0 \\ 0.24 & -0.095248 & 6.049239 & 19.417806 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 25 & 94 & 10 & 54 & 64 \\ 0 & -193.6 & 50 & -117.6 & -114.6 \\ 0 & 0 & 13.284711 & 50.64876 & 14.313843 \\ 0 & 0 & 0 & -12.181996 & -34.619024 \\ 0 & 0 & 0 & 0 & 597.362202 \end{bmatrix}$$

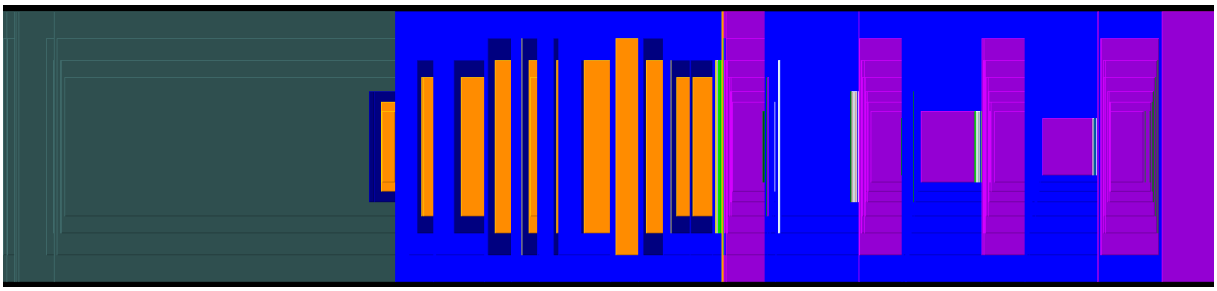
Ostatecznym efektem jego działania było obliczenie szukanego wektora x:

$$x = \begin{bmatrix} 0.558364 \\ -1.062448 \\ -0.004057 \\ 0.196655 \\ 1.395816 \end{bmatrix}$$

Poniższe rysunki przedstawiają wykresy widoku procesów, komunikacji oraz ich legendę, zwizualizowane przy pomocy programu *jumpshot*.



Rysunek 1 Wykres procesów dla rozwiązywania układu 5 równań



Rysunek 2 Wykres komunikacji dla rozwiązywania układu 5 równań (proces "serwera")

Topo	Name	V	S	count	incl	excl
	Preview_State	☑	☑	0	0	0
	RECV_COLUMN_ID	☑	☑	80	0,007	0,007
	RECV_COLUMN_SIZE	☑	☑	80	5,268	5,268
	RECV_COLUMN_VALUES	☑	☑	80	2,42	2,42
	RECV_DIMENSIONS_COLS_SIZE	☑	☑	15	0,226	0,226
	RECV_DIMENSIONS_ROWS_SI...	☑	☑	15	6,263	6,263
	SEND_COLUMN_ID	☑	☑	80	0,004	0,004
	SEND_COLUMN_SIZE	☑	☑	80	0,018	0,018
	SEND_COLUMN_VALUES	☑	☑	80	0,008	0,008
	SEND_DIMENSIONS_COLS_SIZE	☑	☑	15	0,002	0,002
	SEND_DIMENSIONS_ROWS_SI...	☑	☑	15	0,283	0,283

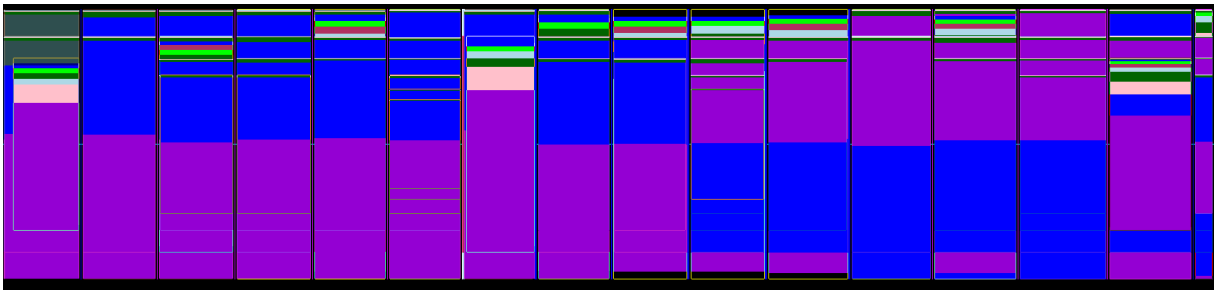
Rysunek 3 Legenda do wyników dla układu 5 równań

7.2. Duży układ równań

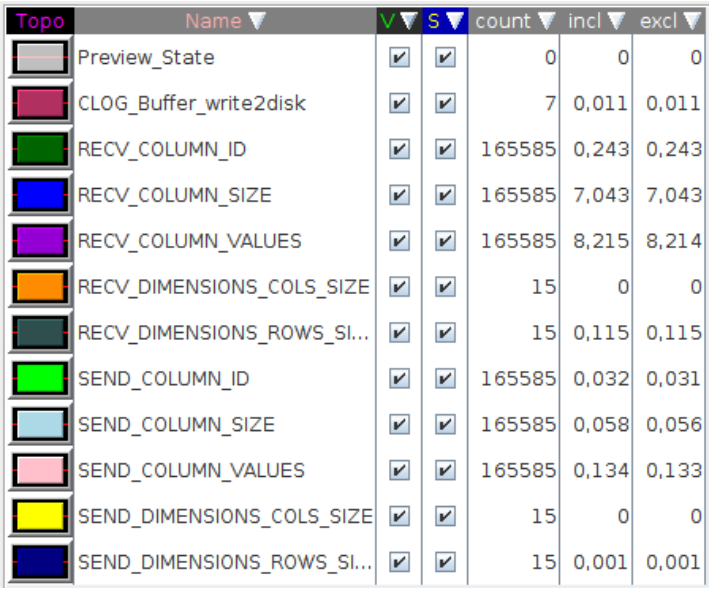
Drugim układem równań, na którym testowaliśmy program, był układ 400 równań liniowych. Rozmiar macierzy A w tym przypadku wynosił 400 x 400, a więc był 6400 razy większy niż w poprzednim podpunkcie.

Poniższe rysunki przedstawiają wykresy widoku procesów, komunikacji oraz ich legendę.

Rysunek 4 Wykres procesów dla rozwiązywania układu 400 równań



Rysunek 5 Wykres komunikacji dla rozwiązywania układu 400 równań (proces "serwera")



Topo	Name	V	S	count	incl	excl
	Preview_State	☑	☑	0	0	0
	CLOG_Buffer_write2disk	☑	☑	7	0,011	0,011
	RECV_COLUMN_ID	☑	☑	165585	0,243	0,243
	RECV_COLUMN_SIZE	☑	☑	165585	7,043	7,043
	RECV_COLUMN_VALUES	☑	☑	165585	8,215	8,214
	RECV_DIMENSIONS_COLS_SIZE	☑	☑	15	0	0
	RECV_DIMENSIONS_ROWS_SI...	☑	☑	15	0,115	0,115
	SEND_COLUMN_ID	☑	☑	165585	0,032	0,031
	SEND_COLUMN_SIZE	☑	☑	165585	0,058	0,056
	SEND_COLUMN_VALUES	☑	☑	165585	0,134	0,133
	SEND_DIMENSIONS_COLS_SIZE	☑	☑	15	0	0
	SEND_DIMENSIONS_ROWS_SI...	☑	☑	15	0,001	0,001

Rysunek 6 Legenda do wyników dla układu 400 równań

8. Wnioski

Podsumowując wyniki możemy stwierdzić, iż program wykonywał się równolegle na 16 procesorach. Pierwszy z nich - serwer - przeprowadzał główną część obliczeń, stąd jego wykres różni się od pozostałych 15. One z kolei posiadają wykresy zbliżone wyglądem, gdyż realizowane przez nie operacje były podobne. Drobne różnice mogą wynikać z faktu, iż w różnych chwilach czasowych procesory te mogły wykonywać zadania o odmiennym stopniu złożoności ze względu na różne rozmiary obliczanych kolumn - im wyższy numer kolumny, tym krótsza była obliczana jej część.

W przypadku małego układu równań dominującą czasowo operacją dla serwera było wysyłanie rozmiaru pionowego macierzy oraz odbieranie wartości kolumn, a dla pozostałych procesorów - odbieranie rozmiaru macierzy, a także rozmiaru i wartości kolumn. Podczas rozwiązywania dużego układu równań dla serwera taką dominującą czasowo operacją było odbieranie wartości kolumn, zaś dla pozostałych procesów odbieranie rozmiarów oraz wartości kolumn. Pozostałe operacje wydają się być pomijalne w porównaniu z wymienionymi powyżej. Duża różnica między czasem wysyłania oraz odbierania, występująca mimo że złożoność tych dwóch czynności wydaje się być podobna, może wynikać z faktu, iż druga operacja jest blokująca. Po jej wywołaniu proces czeka więc na otrzymanie wartości, wstrzymując wykonywanie innych instrukcji na ten czas, zanim dostanie żądaną wartość. Z punktu widzenia maksymalizacji wydajności przetwarzania równoległego celem jest więc możliwie największe skrócenie czasu pomiędzy wywołaniem funkcji MPI_Recv(), a otrzymaniem przez nią oczekiwanej wartości. W przypadku rosnącej liczby zależności między wymienianymi danymi ten problem może stać się niebanalny.

Porównanie długości wykonywania się programów w wersji równoległej i sekwencyjnej, przedstawione w poniższej tabeli, pokazuje, iż dla testowanych przez nas przypadków algorytm zrównoleglony wykonuje się wolniej.

n	5	400
T_S	0.000315	0.431274
T_P	0.007854	3.886212
$\frac{T_P}{T_S}$	24.93333	9.011005
S_P	0.040107	0.110975
E_P	0.002507	0.006936

Tabela 1 Zestawienie osiągów czasowo - wydajnościowych dla analizowanych przypadków z danymi wejściowymi

Jak można zauważyć wartości te są korzystniejsze dla większej macierzy. Wnioskujemy z tego, iż dobrane przez nas wielkości rozwiązywanego problemu były zbyt małe, aby dla tak skonstruowanego algorytmu równoległego korzyści ze zrównoleglenia przerosły koszty komunikacji i synchronizacji. W przypadku odpowiednio dużej macierzy A wydajniejszą wersją okaże się rozwiązanie równoległe.

9. Instrukcja działania programu

Po rozpakowaniu projektu wchodzimy do katalogu głównego projektu, w którym znajduje się plik makefile. Poniżej przedstawiamy listę kilku użytecznych podstawowych komend, które umożliwią wyczyszczenie projektu, a także jego zbudowanie i uruchomienie.

a) wyczyszczenie projektu; zbudowanie go w wersji równoległej w trybie MPE_LOGS, uruchomienie programu bin/main z trzema procesami, zmienna MPEPATH jest ustawiana z poziomu konsoli, wygenerowanie plików bin/mpe_logs.clog2 oraz bin/mpe_logs.slog2, uruchomienie programu jumpshot:

```
make clean; make jumpshot MPEPATH=/opt/nfs/mpe2-2.4.9b/
MPE_LOGS=true NUMOFPROCS=3
```

b) wyczyszczenie projektu, zbudowanie projektu w wersji równoległej w trybie MPE_LOGS, uruchomienie programu bin/main z trzema procesami, zmienna MPEPATH jest ustawiana z poziomu konsoli, wygenerowanie plików bin/mpe_logs.clog2 oraz bin/mpe_logs.slog2, uruchomienie programu jumpshot:

```
make clean; make METHOD_TYPE=METHOD_SEQ NUMOFPROCS=1
```

Wyczyszczenie projektu. Budowanie projektu w wersji sekwencyjnej bez trybu MPE_LOGS.
Uruchomienie programu bin/main z jednym procesem.

Poniżej znajduje się opis tasków makefile naszego projektu, a także porady oraz przykłady użycia.

all	Wykonuje task run. Może wyzwolić task log.
build	Generuje pliki obj/*.o oraz bin/main. Tworzy katalogi obj/ oraz bin/ . Jeśli MPE_LOGS=true, wtedy bin/main jest w trybie MPE_LOGS. Wartości domyślne: MPE_LOGS=false. METHOD_TYPE=METHOD_MPI, MPEPATH=/home/lukasz/mpe-install/mpe2/ Jeśli projekt jest budowany z flagą METHOD_TYPE=METHOD_MPI, wtedy program bin/main jest w wersji równoległej. Jeśli projekt jest budowany z flagą METHOD_TYPE=METHOD_SEQ, wtedy program bin/main jest w wersji sekwencyjnej.
clean	Usuwa pliki obj/*.o i bin/* oraz foldery obj/ i bin/ .
help	Pokazuje pomoc
jumpshot	Uruchamia program jumpshot. Projekt musi być zbudowany tu w trybie MPE_LOGS. Może wyzwolić task log.
log	Generuje plik bin/mpe_logs.slog2.Może wyzwolic task run Projekt musi być zbudowany tu w trybie MPE_LOGS i METHOD_TYPE=METHOD_MPI.
mem	Uruchamia valgrind --leak-check=full bin/main Może wyzwolić task build.
run	Uruchamia bin/main. Może wyzwolić task build. Jeśli projekt był budowany w trybie MPE_LOGS

	i METHOD_TYPE=METHOD_MPI, wtedy ten task generuje plik bin/mpe_logs.clog2. Jeśli projekt był budowany z flagą METHOD_TYPE=METHOD_MPI, wtedy ten task uruchamia program w wersji równoległej. Jeśli projekt był budowany z flagą METHOD_TYPE=METHOD_SEQ, wtedy ten task uruchamia program w wersji sekwencyjnej.