	Politechnika Warszawska	***Programowanie równoległe i rozproszone***
Data: ***27.01.2018***	***Wykonawca:*** Królikowski Krzysztof (244739)	***MPI - Zadanie JB5.*** ***Rozwiązywanie układów równań liniowych metodą Jacobiego***

1. Zrównoleglenie programu (OpenMPI).

Zrównoleglenie za pomocą interfejsu MPI polegało na zbudowaniu w programu w oparciu o komunikaty przesyłane pomiędzy procesami. Technologia ta umożliwia uruchomienie tego samego programu jako osobne procesy, a następnie za pomocą wspomnianych komunikatów, rozdzielenie pracy pomiędzy nimi. Zrównolegleniu podlega wyznaczenie elementów wektora $\boldsymbol{x_k}$, obliczanych dla *k*-tej iteracji algorytmu. Innymi słowy macierz $\boldsymbol{A}$ jest podzielona na poziome pasy, a każdym z pasów zajmują się osobne procesy. Rozdziałem pracy zajmuje się wątek główny (dalej określany jako *master)*, który nie wykonuje obliczeń związanych z obliczeniem poszczególnych elementów wektora $\boldsymbol{x_k}$. Kluczowy listing funkcjonalności zaprezen-

Tabela 1 Fragment funkcji assign_worker

```
MPI_Recv(&b_tmp[0], nrows_chunk, MPI_DOUBLE, MASTER_ID, FROM_MASTER_TAG, MPI_COMM_WORLD,
&status);

MPI_Recv(&A_tmp[0][0], nrows_chunk*ncols, MPI_DOUBLE, MASTER_ID, FROM_MASTER_TAG,
MPI_COMM_WORLD, &status);

      while (1)

      {

         MPI_Recv(&k, 1, MPI_INT, MASTER_ID, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

         if (status.MPI_TAG == STOP_TAG)

            break;

         MPI_Recv(x_0_tmp, ncols, MPI_DOUBLE, MASTER_ID, FROM_MASTER_TAG, MPI_COMM_WORLD,
         &status);

         calculate_x(x__tmp, A_tmp, b_tmp, x_0_tmp, ncols, offset, nrows_chunk);

         MPI_Send(&offset, 1, MPI_INT, MASTER_ID, FROM_WORKER_TAG, MPI_COMM_WORLD);

         MPI_Send(&nrows_chunk, 1, MPI_INT, MASTER_ID, FROM_WORKER_TAG, MPI_COMM_WORLD);

         MPI_Send(&x__tmp[0], nrows_chunk, MPI_DOUBLE, MASTER_ID, FROM_WORKER_TAG,
         MPI_COMM_WORLD);

      }
```

towano poniżej.

Jest to fragment funkcji, odpowiadającej za przydzielenie pracy poszczególnym workerom. Algorytm działania programu jest następujący:

1. Wątek główny (*master*) wczytuje macierz **A** oraz wektor **b** do programu.
2. *Master* wyznacza liczbę elementów wektora $\boldsymbol{x_k}$, która ma zostać obliczony przez dostępne workery.
3. Master wysyła do każdego workera odpowiednie wiersze macierzy **A**, odpowiednie wiersze wektora b oraz cały wektor **x_0**
4. Po otrzymaniu paczki danych, każdy worker wyznacza przydzielone mu elementy wektora x. Każdy worker po skończonej pracy, odsyła wektor x. Proces ten odbywa się iteracyjnie do momentu, aż master nie wyśle workerowi flagi STOP, która sprawia, że worker wychodzi z nieskończonej pętli while.
5. Master po otrzymaniu od każdego workera płaczki danych zawierającej elementy nowego wektora **x** buduje cały wektor oraz wyznacza normę kwadratową miedzy nowym przybliżeniem **x** oraz jego starym przybliżeniem. Jeżeli norma jest mniejsza od *epsilonu*, bądź tez uzyskano maksymalną liczbę iteracji, master kończy pracę i wysyła do wszystkich workerów flagę, która powoduje, że również i one kończą swoją pracę.

1.1 Uwagi dodatkowe

Zdecydowano się na strukturę programu, w której jeden wątek jest odpowiedzialny za rozdzielanie pracy, a reszta wątków odpowiada za jej wykonanie z takiego powodu, iż obawiano się że dodanie wątkowi głównemu dodatkowych zadań mogłoby sprawić, iż wątek główny zajęty byłby wykonywaniem obliczeń, a pozostałe wątki mogłyby już swoje zadania wykonać i oczekiwać na to, aż wątek główny przydzieli im nowe zadania. Sprawiłoby to, że komunikacja między procesami byłaby nieefektywna. Dodatkowo zdecydowano się na przesyłanie fragmentów macierzy **A** pomiędzy masterem a workerami aby nie doszło do sytuacji, że każdy worker wczytywałby dużą macierz **A** do swojej pamięci, a następnie wykonywał obliczenia tylko na jej niewielkim fragmencie. Byłoby to rozwiązanie zdecydowanie prostsze, jednakże bardzo kosztowne pod względem pamięci. W tej implementacji każdy worker alokuje tylko tyle pamięci ile jest mu potrzebne na odbiór danych od mastera. Warto zauważyć, że macierz **A** oraz wektor **b**, nie jest przesyłany przy każdej iteracji k, tylko jest to jednokrotna operacja. Dlatego narzut na czas obliczeń jest względnie niewielki.

2. Wynik zrównoleglenia

Zrównoleglanie wykonano na klastrze składającym się z kilkunastu procesorów Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz z 130GB pamięci RAM. Z racji tego, iż obliczenia zostały przeprowadzone na innej maszynie niż wcześniejsze etapy projektu, należało przeliczyć część sekwencyjną oraz OpenMP jeszcze raz.

Implementacja MPI

- - MPICH/3.2

Sposób kompilacji:

Wszystkie etapy skompilowano z użyciem flag (kompilator gcc):

- *-march=native*
- *-O2*

Pomiar czasu:

- OpenMP - *omp_get_wtime( )*
- MPI - *MPI_Wtime()*
- Sekwencyjny – *clock()*

Tabela 2. Porównanie czasów [s] wykonania programu w wersji sekwencyjnej oraz przy zastosowaniu OpenMP ze zmienną liczbą wykorzystanych wątków

Rozmiar problemu	Typ	1	2	3	4	8
1000	Sekwencyjny	1,97	-	-	-	-
	OpenMP	1,87	1,11	0,74	0,56	0,31
	MPI	-	1,67	1,04	0,65	0,28
5000	Sekwencyjny	21,40	-	-	-	-
	OpenMP	23,97	12,14	9,59	9,14	4,94
	MPI	-	23,72	14,00	7,75	3,61

Współczynnik przyspieszenia jest definiowany jako:

$$S(n,p) = \frac{T(n,1)}{T(n,p)}$$

Gdzie:

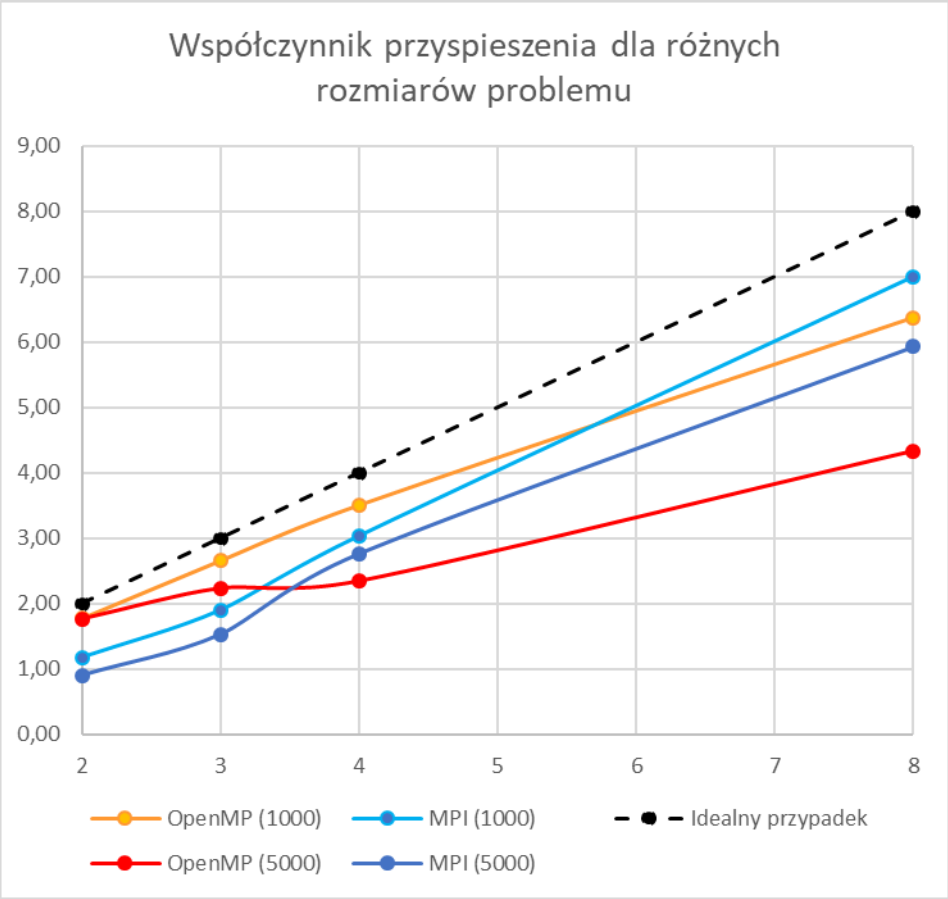
$S(n,p)$ – współczynnik przyspieszenia wykonania programu realizującego algorytm dla zadania o wielkości n na maszynie równoległej z p procesorami,

$T(n,1)$ - czas wykonania programu realizującego algorytm dla zadania o wielkości n na maszynie równoległej z 1 procesorem,

$T(n,p)$ – czas wykonania programu realizującego ten sam algorytm dla zadania o wielkości n na maszynie równoległej z p procesorami.

Tabela 3. Współczynnik przyspieszenia względem czasu wykonania części sekwncyjnej [-]

Rozmiar problemu	Typ	2	3	4	8
1000	OpenMP	1,78	2,66	3,51	6,37
	MPI	1,18	1,90	3,03	7,00
5000	OpenMP	1,76	2,23	2,34	4,33
	MPI	0,90	1,53	2,76	5,93



3. Wnioski

Na wykresie można zaobserwować, że lepsze rezultaty uzyskano przy użyciu technologii MPI. Warto zwrócić uwagę na to, że program wykonany w MPI został tak skonstruowany, że używając 2 wątków tak naprawdę praca nie jest zrównoleglania (master + 1 worker). Stąd dla małej liczby wątków, lepsze rezultaty uzyskuje się używając technologii OpenMP. Z drugiej strony, wraz ze wzrostem liczby możliwych do wykorzystania wątków, przewaga MPI się powiększa. Uzyskano 6 krotne przyśpieszenie dla liczby wątków równej 8, zaś w przypadku OpenMP współczynnik ten wyniósł już tylko 4,3.

Warto zwrócić uwagę, iż skompilowany program sekwencyjny dla w przypadku 1000 wierszy macierzy **A** wykonuje się wolniej, niż skompilowany program z wykorzystaniem dyrektyw OpenMP, czy z wykorzystaniem interfejsu MPI dla takiego samego rozmiaru problemu. Może mieć to związek z niedokładnością pomiaru czasu, co ma kluczowe znaczenie w przypadku tak niewielkich czasów wykonania, gdyż w trzech przypadkach, korzystano z różnych sposobów pomiaru czasu (dedykowanych poszczególnych rozwiązaniom). Dla rozmiaru problemu równego 5000, programy zachowują się tak, jak się oczekuje – to znaczy część sekwencyjna programu jest szybsza niż część OpenMP wywołana na 1 wątków oraz część MPI wykonana na 2 wątkach – efekt narzutu komunikacji między procesami/wątkami.

Pod względem trudności implementacji, technologia OpenMP jest prostsza i mniej czasochłonna. Z drugiej strony MPI oferuje bardzo dużą kontrolę nad praktycznie wszystkimi aspektami programu. Uzyskane rezultaty pozwalają twierdzić, iż OpenMP powoduje większy narzut obliczeniowy niż dobrze napisany program w technologii MPI. Z drugiej strony możliwości jakie daje MPI pociągają za sobą duże ryzyko wystąpienia błędów implementacyjnych, których efektem może być nawet sytuacja odwrotna – to znaczy spowolnienie działania programu. W przypadku MPI należy dążyć więc do ograniczenia do minimum ilości przekazywanych między procesami komunikatów.