Gliwice, 14 stycznia 2014

Obliczenia Równoległe II

Temat: Komunikacja między procesami oraz redukcja danych (MPI)

Prowadzący: prof. dr hab. inż. Zbigniew Czech

Marius Rejdak
Informatyka mgr, Grupa OS1

Wstęp

Standard MPI (ang. Message Passing Interface) sięga wczesnych lat dziewięćdziesiątych ubiegłego wieku. W ramach warsztatów dotyczących standardów przesyłania wiadomości w środowiskach z pamięcią rozproszoną (kwiecień 1992 r.) powołano grupę roboczą nazwaną później Message Passing Interface Forum, której zadaniem było opracowanie jednolitego standardu biblioteki. Pierwszą opublikowaną wersją tego standardu było MPI-1 (maj 1994r.), kolejne ważniejsze wersje to MPI-2 (lipiec 1997r.), MPI-2.2 (wrzesień 2009r.), oraz MPI-3 (wrzesień 2012r.). Implementacje bibliotek MPI wchodzą w skład oprogramowania praktycznie wszystkich komercyjnych komputerów równoległych. W ramach tej pracy postanowiłem opisać najpopularniejszą bibliotekę OpenMPI, której aktualna wersja jest zgodna ze standardem MPI-2.2, a wersja rozwojowa ma być zgodna MPI-3.

Ogólnym założeniem standardu MPI jest ujednolicenie interfejsu bibliotek pomocnych przy implementacji programów równoległych, w których zadania współpracują ze sobą korzystając z przesyłanych wiadomości. Zastosowanie znajduje w sieciach komputerowych stanowiących klastry obliczeniowe oraz superkomputerach zbudowanych nierzadko w oparciu o hybrydowe architektury. OpenMPI w wersji rozwojowej obsługuje także bezpośrednie przesyłanie komunikatów między jednostkami obliczeniowymi wykorzystującymi procesory graficzne oparte o architekturę CUDA, dzięki czemu znalazł zastosowanie w wielu superkomputerach znajdujących się na liście TOP500.

Komunikacja między procesami

Biblioteka OpenMPI służy do implementacji obliczeń rozproszonych z przesyłaniem wiadomości. Zakłada się że system który wykonuje obliczenia, składa się z pewnej liczby procesorów wyposażonych w pamięci lokalne, oraz że nie występuje pamięć wspólna. Umożliwia przesyłanie wiadomości między dowolną parą lub zdefiniowaną grupą procesów (zdefiniowanej przez użytkownika w ramach komunikatora). Procesy w OpenMPI ponumerowane są od 0 do p-1, gdzie 0 to proces zarządcy, a p to ilość wszystkich procesów.

Funkcje *MPI_Send* oraz *MPI_Isend*

Funkcje ta służą do przesyłania wiadomości między parą procesów (ang. point-to-point-communication), wywoływana jest po stronie procesu wysyłającego, po stronie procesu odbierającego należy wywołać analogicznie *MPI_Recv* lub *MPI_Irecv* z odpowiednimi parametrami. Dla *MPI_Send* proces wywołujący tę funkcję wstrzymuje swoje obliczenia aż wysyłana wiadomość zostanie odebrana przez proces odbierający (operacja blokująca). Dla *MPI_Isend* proces wywołujący tę funkcję wstrzymuje swoje obliczenia aż wysyłana wiadomość zostanie odebrana przez proces odbierający (operacja blokująca).

Składnia poleceń:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Parametry wejściowe:

Nazwa	Typ	Opis
`buf`	`void*`	Adres bufora danych wysyłanych
`count`	`int`	Ilość wysyłanych elementów
`datatype`	`MPI_Datatype`	Typ wysyłanych elementów
`dest`	`int`	Numer procesu docelowego
`tag`	`int`	Znacznik wiadomości
`comm`	`MPI_Comm`	Komunikator

Parametry wyjściowe:

Nazwa	Typ	Opis
`request`	`MPI_Request`	Wskaźnik na strukturę służącą do sprawdzenia stanu przesyłu

Funkcje *MPI_Recv* i *MPI_Irecv*

Funkcje analogiczne do *MPI_Send* i *MPI_Isend*, lecz wywoływane po stronie procesu odbierającego. W przypadku *MPI_Recv* jest to operacja blokująca, czyli wstrzymuje wykonywanie obliczeń procesu odbierającego aż do otrzymania wiadomości, oraz dla *MPI_Irecv* operacja nieblokująca.

Składnia poleceń:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
```

Parametry wejściowe:

Nazwa	Typ	Opis
`buf`	`void*`	Adres bufora dla danych odbieranych
`count`	`int`	Ilość odbieranych elementów
`datatype`	`MPI_Datatype`	Typ odbieranych elementów
`source`	`int`	Numer procesu źródłowego
`tag`	`int`	Znacznik wiadomości
`comm`	`MPI_Comm`	Komunikator

Parametry wyjściowe:

Nazwa	Typ	Opis
`status`	`MPI_Status`	Struktura przechowująca dane dotyczące odebranej wiadomości
`request`	`MPI_Request`	Wskaźnik na strukturę służącą do sprawdzenia stanu przesyłu

Funkcja *MPI_Test*

Wykorzystywana jest dla struktury *MPI_Request* zwracanej przez *MPI_Isend* and *MPI_Irecv*. Służy do sprawdzenia czy operacja nieblokującego przesyłu została zakończona.

Składnia polecenia:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Parametry wejściowe:

Nazwa	Typ	Opis
`request`	`MPI_Request`	Wskaźnik na strukturę służącą do sprawdzenia stanu przesyłu

Parametry wyjściowe:

Nazwa	Typ	Opis
`flag`	`int*`	Wpisuje *true* jeżeli operacja została zakończona
`status`	`MPI_Status`	Struktura przechowująca dane dotyczące przesyłanej wiadomości

Funkcja *MPI_Wait*

Wykorzystywana jest dla struktury *MPI_Request* zwracanej przez *MPI_Isend* and *MPI_Irecv*. Służy do zatrzymania procesu aż do zakończenia zadanej operacji przesyłu.

Składnia polecenia:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Parametry wejściowe:

Nazwa	Typ	Opis
`request`	`MPI_Request`	Wskaźnik na strukturę służącą do sprawdzenia stanu przesyłu

Parametry wyjściowe:

Nazwa	Typ	Opis
`status`	`MPI_Status`	Struktura przechowująca dane dotyczące przesyłanej wiadomości

Funkcje *MPI_Probe* i *MPI_Iprobe*

Wykorzystywane są do sprawdzenia czy na proces oczekuje wiadomość do odebrania. *MPI_Probe* jest operacją blokującą – zatrzymuje proces aż do nadejścia wiadomości, *MPI_Iprobe* jest operacją nieblokującą – zwraca tylko informację czy określona wiadomość oczekuje na odbiór.

Składnia poleceń:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status
*status)
```

Parametry wejściowe:

Nazwa	**Typ**	**Opis**
`source`	`int`	Numer procesu źródłowego
`tag`	`int`	Znacznik wiadomości
`comm`	`MPI_Comm`	Komunikator

Parametry wyjściowe:

Nazwa	**Typ**	**Opis**
`flag`	`int*`	Wpisuje *true* jeżeli istnieje oczekująca wiadomość
`status`	`MPI_Status`	Struktura przechowująca dane dotyczące oczekującej wiadomości

Funkcja *MPI_Bcast*

Funkcja ta służy do przesyłania wiadomości między wieloma procesami poprzez operację rozgłaszania (ang. one-to-all), należy ją wywołać we wszystkich procesach które uczestniczą w tej komunikacji (proces wysyłający oraz wszystkie procesy odbierające w ramach wybranego komunikatora). Jest to operacja blokująca, zatem proces wysyłający czeka aż wszystkie procesy odbierające otrzymają wiadomość.

Składnia polecenia:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm)
```

Parametry wejściowe:

Nazwa	Typ	Opis
buffer	void*	Adres bufora danych wysyłanych/odbieranych
count	int	Ilość przesyłanych elementów
datatype	MPI_Datatype	Typ przesyłanych elementów
root	int	Numer procesu wysyłającego
comm	MPI_Comm	Komunikator

Przykładowa implementacja przy pomocy *MPI_Send* i *MPI_Recv*:

```
void MPI_Bcast_custom(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm) {
	int me, numprocs;
	MPI_Status status;
	MPI_Comm_rank(MPI_COMM_WORLD, &me);
	MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
	if(root == me) {
		for(int i = 0; i < numprocs; ++i) {
			if(i != me) {
				MPI_Send(buffer, count, datatype, i, 0, comm);
			}
		}
	}
	else {
		MPI_Recv(buffer, count, datatype, root, 0, comm, &status);
	}
}
```

Operacja redukcji – funkcja *MPI_Reduce*

Funkcja ta służy do przesyłania wiadomości między wieloma procesami typu "wszystkie do jednego" (ang. All-to-one), oraz jednocześnie wykonuje operację redukcji – czyli wykonanie na wszystkich otrzymanych danych określonej operacji aby otrzymać dokładnie jedną danę wynikową. Należy ją wywołać we wszystkich procesach które uczestniczą w tej komunikacji (proces odbierający oraz wszystkie procesy wysyłające w ramach wybranego komunikatora). Jest to operacja blokująca, zatem proces odbierający czeka aż wszystkie procesy wysyłające nadadzą swoje wiadomości.

Składnia polecenia:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm)
```

Parametry wejściowe:

Nazwa	Typ	Opis
`sendbuf`	`void*`	Adres bufora dla danych wysyłanych (istotne tylko dla procesów wysyłających)
`recvbuf`	`void*`	Adres bufora dla danych odbieranych (istotne tylko dla procesów odbierających)
`count`	`int`	Ilość przesyłanych elementów
`datatype`	`MPI_Datatype`	Typ przesyłanych elementów
`op`	`MPI_Op`	Typ wykonywanej operacji
`root`	`int`	Numer procesu do którego dane zostaną zgromadzone
`comm`	`MPI_Comm`	Komunikator

Obsługiwane operacje (typ *MPI_Op*):

Nazwa	Typ danych w języku C
MPI_MAX	Maximum ze wszystkich wartości
MPI_MIN	Minimum ze wszystkich wartości
MPI_SUM	Suma
MPI_PROD	Iloczyn
MPI_LAND	Iloczyn logiczny
MPI_BAND	Iloczyn logiczny na bitach
MPI_LOR	Suma logiczna
MPI_BOR	Suma logiczna na bitach
MPI_LXOR	Suma modulo 2
MPI_BXOR	Suma modulo 2 na bitach

Przykładowa implementacja redukcji liczb całkowitych z operacją sumy przy pomocy *MPI_Send* i *MPI_Recv*:

```
void MPI_Reduce_sum_ints(int *send, int *recv, int root, MPI_Comm comm) {
	int me, numprocs, buffer;
	MPI_Status status;
	MPI_Comm_rank(MPI_COMM_WORLD, &me);
	MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
	if(root == me) {
		*recv = *send;
		for(int i = 0; i < numprocs; ++i) {
			if(i != me) {
				MPI_Recv(&buffer, 1, MPI_INT, i, 0, comm, &status);
				*recv += buffer;
			}
		}
	}
	else {
		MPI_Send(send, 1, MPI_INT, root, 0, comm);
	}
}
```

Podstawowe typy danych zdefiniowane w bibliotece OpenMPI (typ *MPI_Datatype*)

Nazwa	**Typ danych w języku C**
`MPI_CHAR`	`signed char`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_BYTE`	dane o rozmiarze 1 bajta (dla przesyłania własnych struktur)
`MPI_SHORT`	`signed short int`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_INT`	`signed int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_PACKED`	dane spakowane za pomocą funkcji *MPI_Pack()*

Symulacja firmy „Macierz" – komunikacja niesynchroniczna w MPI

Firma „Macierz" posiada trzy zakłady: Zakład A, Zakład B i Zakład C. Każdy z zakładów zajmuję się mnożeniem macierzy dostarczanych przez klientów (w symulacji dla uproszczenia należy przyjąć losowe wymiary macierzy, nie większe niż 100 oraz losowe wartości elementów macierzy). Ponadto każdy z zakładów specjalizuje się w obsłudze całej firmy w zakresie jednej funkcji wsparcia:

- Zakład A zajmuje się obsługą kadrową, głównie rekrutacją
- Zakład B zapewnia obsługę prawną w przypadku wystąpienia sporów
- Zakład C dostarcza serwisu informatycznego w przypadku awarii.

Na podstawie wieloletnich obserwacji wiadomo, że w każdym zakładzie potrzeba rekrutacji pojawia się średnio co 10 wymnożonych macierzy, spór prawny – średnio co 20 wymnożonych macierzy, a problem informatyczny – średnio co 5 wymnożonych macierzy. Obsługa sytuacji nadzwyczajnych ma zróżnicowany czas:

- Rekrutacja trwa tyle co podniesienie do kwadratu macierzy o wymiarze 10
- Rozwiązanie problemu prawnego trwa tyle co podniesienie do kwadratu macierzy o wymiarze 1000
- Rozwiązanie problemu informatycznego trwa tyle co podniesienie do kwadratu macierzy o wymiarze 100.

W czasie realizacji funkcji wsparcia zakład nie zajmuje się mnożeniem macierzy dla klientów. Zakład w którym wystąpił problem prowadzi w dalszym ciągu działalność usługową i jednocześnie oczekuje na rozwiązanie zgłoszonego problemu.

Stworzyć program współbieżny symulujący działanie firmy „Macierz" przy pomocy trzech procesów odpowiadających trzem zakładom. Przyjąć, że sprawdzanie czy został zgłoszony problem ma miejsce po każdym wymnożeniu macierzy dla klienta. Symulację należy prowadzić do czasu, aż którykolwiek z zakładów wykona 100 wymnożeń macierzy dla klientów. Po zakończeniu symulacji każdy z procesów wyświetla na ekranie statystykę: ile czasu zajmował się mnożeniem macierzy dla klientów, ile czasu zajmował się obsługą funkcji wsparcia oraz ile czasu czekał na rozwiązanie problemu przez inny zakład.

W drugim etapie zadanie można uogólnić na n zakładów, które zajmują się obsługą w zakresie trzech funkcji wsparcia. Należy także wprowadzić 1 proces klienta, generującego zamówienia do losowo wybranego zakładu.
Do zaimplementowania opisanej symulacji należy wykorzystać funkcje:

- MPI_Probe i/lub MPI_Iprobe – do cyklicznego „przepytywania", czy w buforze odbiornika pojawiła się wiadomość gotowa do odebrania;
- MPI_Send i MPI_Recv – do wysyłania i odbierania (do zrealizowania zadania nie jest konieczne wprowadzanie innych trybów komunikacji);
- MPI_Wtime, MPI_Barrier – mogą się przydać;

UWAGA: Można w razie potrzeby manipulować wartościami podanych w treści zadania parametrów. Operacje mnożenia, rozmiary mnożonych macierzy i liczba mnożeń są najmniej istotnym elementem zadania. Proszę zająć się tym na samym końcu. Zadanie może być zaliczone również wtedy, jeżeli zamiast mnożenia przedsiębiorstwa/procesy będą wykonywały cokolwiek innego (stopień „skomplikowania" czynności wpłynie wtedy na ocenę końcową).

Opis rozwiązania

Proces zerowy pełni funkcję nadzorcy, oraz generuje macierze od klientów. Pozostałe procesy są równo dzielone pomiędzy kolejne zakłady.

Wyniki uruchomienia programu

```
marius@marius-pc: ~/p/p/O/l/s/lab3|master⚡* $ make
mpicc -O -std=c99 -c systest.c
mpicc -O -o systest.out systest.o -lm
marius@marius-pc: ~/p/p/O/l/s/lab3|master⚡* $ make run
mpirun -n 7 systest.out
Klient: wysłano 68655 macierzy
Zakład 0: rozwiązywano macierze 0.048774s, rozwiązywano problemy 0.005283s
Zakład 0: rozwiązywano macierze 0.050338s, rozwiązywano problemy 0.004817s
Zakład 2: rozwiązywano macierze 0.001561s, rozwiązywano problemy 0.057079s
Zakład 2: rozwiązywano macierze 0.001859s, rozwiązywano problemy 0.060081s
Zakład 1: rozwiązywano macierze 0.000198s, rozwiązywano problemy 0.245448s
Zakład 1: rozwiązywano macierze 0.000306s, rozwiązywano problemy 0.282974s
```

Listing programu

```
/*
Symulacja firmy Macierz - komunikacja niesynchroniczna w MPI
Data wykonania ćwiczenia: 2013-11-27
*/

#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>

//Numeracja zakładów (!= numeracja procesów)
#define ZAKLAD_A 0
#define ZAKLAD_B 1
#define ZAKLAD_C 2

//Tagi
#define END_PROGRAM 0
#define MESSAGE 1
#define PROBLEM_SOL 2

//Limit wykonanych działań
#define ILOSC_MACIERZY 100

//Średnie występowanie problemów
#define PORTZ_REKRUTACJA 10   //Zakład A
#define PORTZ_SPOR_PRAWN 20   //Zakład B
#define PORTZ_INFOR 5     //Zakład C

//Trudności problemów
#define ROZW_REKRUTACJA 10
#define ROZW_SPOR_PRAWN 1000
#define ROZW_PROB_INFOR 100

//Oznaczenie problemu
enum Problem {NONE = 0, REKRUTACJA, SPOR_PRAWNY, INFORM};

//Struktura do przesyłania informacji o każdym zakładzie
```

```
struct ZakladInfo
{
 int typ;
 bool problemRekrutacja;
 bool problemSporPrawny;
 bool problemInform;
};

//Strunktura do przesyłania opisu problemu
struct Message
{
 int problemSource;
 int matrixSize;
 enum Problem problemType;
};

//Generacji macierzy
void getMatrix(int size, double **values)
{
 *values = malloc(size * size * sizeof(double));

 for(int i = 0; i < size*size; ++i)
 {
  (*values)[i] = rand()%1000;
 }
}

//Zwraca wielkość macierzy do rozwiązania
int getProblemSize(enum Problem p)
{
 switch(p)
 {
  case NONE:
  default:
  return rand()%100 + 1;

  case REKRUTACJA:
  return ROZW_REKRUTACJA;

  case SPOR_PRAWNY:
  return ROZW_SPOR_PRAWN;

  case INFORM:
  return ROZW_PROB_INFOR;
 }
}

//Generacja problemu
enum Problem getProblem()
{
 unsigned int s = rand();
 if(s % PORTZ_SPOR_PRAWN == 0)
 {
  return SPOR_PRAWNY;
 }
 else if((s+1) % PORTZ_REKRUTACJA == 0)
 {
  return REKRUTACJA;
 }
 else if((s+2) % PORTZ_INFOR == 0)
 {
  return INFORM;
```

```
 }
 else
 {
  return NONE;
 }
}

//Rozwiązanie zadanego problemu
double solveProblem(int size, double *values)
{
 double start_time = MPI_Wtime();
 size_t num_bytes = sizeof(double)*size*size;
 double *macierz_src = malloc(num_bytes);
 memcpy(macierz_src, values, num_bytes);

 for(int i = 0; i < size*size; ++i)
 {
  values[i] = macierz_src[i]*macierz_src[i];
 }

 free(macierz_src);
 return MPI_Wtime()-start_time;
}

//Przekształca numer procesu na numer zakładu
inline int zakladNr(int me)
{
 return (me-1) % 3;
}

//Dla numeru zakładu zwraca numer rozwiązywanego problemu
enum Problem rozwiazuje(int zaklad)
{
 switch(zaklad)
 {
  case ZAKLAD_A:
  return REKRUTACJA;

  case ZAKLAD_B:
  return SPOR_PRAWNY;

  case ZAKLAD_C:
  return INFORM;

  default:
  return NONE;
 }
}

//Zwraca numer zakładu który może rozwiązać zadany problem
int hasProblem(enum Problem p, struct ZakladInfo *zaklady, int mpi_size)
{
 for(int i = 1; i < mpi_size; ++i)
 {
  if(p == REKRUTACJA)
  {
   if(zaklady[i].problemRekrutacja)
    return i;
  }
  else if(p == SPOR_PRAWNY)
  {
   if(zaklady[i].problemSporPrawny)
```

```
    return i;
   }
   else if(p == INFORM)
   {
    if(zaklady[i].problemInform)
     return i;
   }
  }

  return 0;
}

//Główny proces zakładu
void proces_zaklad(int me, int typ_zakladu, double *times)
{
 //Zapisywane łączne czasy wykonywanych operacji
 double *timeWork = times+0, *timeProblems = times+1, *timeMyProblems = times+2, *timeWait =
times+3;
 //Czy ten zakład ma nierozwiązany problem
 bool problemRekrutacja = false, problemSporPrawny = false, problemInform = false;
 //Licznik rozwiązanych macierzy od klientów
 int matrixes_solved = 0;
 //Rodzaj rozwiązywanych problemów przez zakład
 enum Problem rozwiazujeP = rozwiazuje(typ_zakladu);

 //Maksymalna liczba rozwiązanych macierzy
 while(matrixes_solved < 10000)
 {
  MPI_Status status, status_tmp;
  int tag, source, flag = 0;

  //Sprawdzenie czy istnieje oczekująca wiadomość
  MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
  if(flag) //Oczekująca wiadomość na odebranie
  {
   source = status.MPI_SOURCE; //Nadawca wiadomości
   tag = status.MPI_TAG; //Tag wiadomości
   if(tag == END_PROGRAM) //Oczekująca wiadomość to informacja o zakończeniu programu
   {
    //Opcjonalne, odebranie wiadomości o zakończeniu programu
    MPI_Recv(0, 0, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status_tmp);
    return;
   }
   else if(tag == MESSAGE) //Oczekująca wiadomość to problem do rozwiązania
   {
    struct Message m;

    //Odebranie wiadomości
    MPI_Recv(&m, sizeof(struct Message), MPI_CHAR, source, tag, MPI_COMM_WORLD, &status_tmp);

    if(m.problemType == NONE) //Wiadomość to macierz do rozwiązania od klient
    {
     //Rozwiązanie macierzy od klienta
     double *values = NULL;
     int size = m.matrixSize;
     getMatrix(size, &values);
     *timeWork += solveProblem(size, values);
     free(values);

     matrixes_solved++;
```

```
     //"Generacja" problemu
     enum Problem p = getProblem();
     switch(p)
     {
      case REKRUTACJA:
      problemRekrutacja = true;
      break;

      case SPOR_PRAWNY:
      problemSporPrawny = true;
      break;

      case INFORM:
      problemInform = true;
      break;
     }

     if(p != NONE) //Wystąpił problem, powiadomienie procesu nadzorcy o oczekującym na
rozwiązanie problemie
     {
      struct Message mx;
      mx.matrixSize = getProblemSize(p);
      mx.problemType = p;
      mx.problemSource = me;
      MPI_Send(&mx, sizeof(struct Message), MPI_CHAR, 0, MESSAGE, MPI_COMM_WORLD);
     }

    }
    else if(m.problemType == rozwiazujeP) //Wiadomość to problem który może rozwiązać ten
zakład
    {
     //Rozwiązanie problemu
     double *values = NULL;
     int size = m.matrixSize;
     getMatrix(size, &values);
     *timeProblems += solveProblem(size, values);
     free(values);

     //Powiadomienie procesu nadzorcy o rozwiązaniu problemu
     MPI_Send(&m, sizeof(struct Message), MPI_CHAR, 0, PROBLEM_SOL, MPI_COMM_WORLD);
     //Powiadomienie "autora" problemu o rozwiązaniu problemu
     MPI_Send(&m, sizeof(struct Message), MPI_CHAR, m.problemSource, PROBLEM_SOL,
MPI_COMM_WORLD);
    }
   }
   else if(tag == PROBLEM_SOL) //Oczekująca wiadomość to informacja o rozwiązaniu problemu
   {
    //Odebranie wiadomości
    struct Message m;
    MPI_Recv(&m, sizeof(struct Message), MPI_CHAR, source, tag, MPI_COMM_WORLD, &status_tmp);

    switch(m.problemType)
    {
     case REKRUTACJA:
     problemRekrutacja = false;
     break;

     case SPOR_PRAWNY:
     problemSporPrawny = false;
     break;
```

```
     case INFORM:
     problemInform = false;
     break;
    }
   }
  }
  else //Brak oczekującej wiadomości, żądanie otrzymania zadania do wykonania
  {
   struct Message m;
   m.problemType = NONE;
   MPI_Send(&m, sizeof(struct Message), MPI_CHAR, 0, MESSAGE, MPI_COMM_WORLD);
  }
 }

 //Powiadomienie procesu nadzorcy o zakończeniu programu
 MPI_Send(0, 0, MPI_CHAR, 0, END_PROGRAM, MPI_COMM_WORLD);
}

//Główny proces klienta (nadzorcy)
void proces_klient(int me, int mpi_size, int *send_matrixes)
{
 //Tablica przechowująca informacje o zakładach
 struct ZakladInfo zaklady[mpi_size];
 //Inicjalizacja w/w tablicy
 for(int i = 1; i < mpi_size; ++i)
 {
  zaklady[i].typ = zakladNr(i);
  zaklady[i].problemRekrutacja = false;
  zaklady[i].problemSporPrawny = false;
  zaklady[i].problemInform = false;
 }

 while(true)
 {
  MPI_Status status, status_tmp;
  int source, tag;

  //Czekanie na nową oczekującą wiadomość
  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  source = status.MPI_SOURCE; //Nadawca wiadomości
  tag = status.MPI_TAG; //Tag wiadomości
  if(tag == END_PROGRAM) //Zakończenie programu
  {
   //Odebranie wiadomości
   MPI_Recv(0, 0, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status_tmp);
   //Powiadomienie pozostałych procesów o zakończeniu programu
   for(int i = 1; i < mpi_size; ++i)
   {
    if(i != source)
    {
     MPI_Send(0, 0, MPI_CHAR, i, END_PROGRAM, MPI_COMM_WORLD);
    }
   }

   //Informacja o zakończeniu programu może dojść do zakładów w momencie kiedy będą
zablokowane
   //na operacji wysyłania, należy ich wiadomości odebrać aby mogły się normalnie zakończyć
   int flag = 0;
   do
   {
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status_tmp);
```

```
    if(flag)
    {
     struct Message m;
     MPI_Recv(&m, sizeof(struct Message), MPI_CHAR, status_tmp.MPI_SOURCE, status_tmp.MPI_TAG,
MPI_COMM_WORLD, &status_tmp);
    }
   } while(flag);

   return;
  }
  else if(tag == MESSAGE)
  {
   //Odebranie wiadomości
   struct Message m;
   MPI_Recv(&m, sizeof(struct Message), MPI_CHAR, source, tag, MPI_COMM_WORLD, &status_tmp);

   if(m.problemType == NONE) //Zakład wysłał rządanie problemu do rozwiązania
   {
    //Sprawdzenie czy oczekuje jakiś problem oczekujący na rozwiązanie dla tego zakładu
    enum Problem p = rozwiazuje(zakladNr(source));
    int todoProblem = hasProblem(p, zaklady, mpi_size);

    if(todoProblem)
    {
     m.problemSource = todoProblem;
     m.problemType = p;
     m.matrixSize = getProblemSize(p);
    }
    else //Wygenerowanie macierzy "od klienta"
    {
     m.problemSource = 0;
     m.problemType = NONE;
     m.matrixSize = getProblemSize(NONE);
    }

    //Wysłanie wiadomości
    MPI_Send(&m, sizeof(struct Message), MPI_CHAR, source, MESSAGE, MPI_COMM_WORLD);

    //Zliczanie wysłanych macierzy
    (*send_matrixes)++;
   }
   else //Zakład zgłosił wystąpienie problemu
   {
    switch(m.problemType)
    {
     case REKRUTACJA:
     zaklady[source].problemRekrutacja = true;
     break;

     case SPOR_PRAWNY:
     zaklady[source].problemSporPrawny = true;
     break;

     case INFORM:
     zaklady[source].problemInform = true;
     break;
    }
   }
  }
  else if(tag == PROBLEM_SOL) //Zakład zgłosił rozwiązanie problemu
  {
```

```
   struct Message m;
   MPI_Recv(&m, sizeof(struct Message), MPI_CHAR, source, tag, MPI_COMM_WORLD, &status_tmp);

   switch(m.problemType)
   {
    case REKRUTACJA:
    zaklady[m.problemSource].problemRekrutacja = false;
    break;

    case SPOR_PRAWNY:
    zaklady[m.problemSource].problemSporPrawny = false;
    break;

    case INFORM:
    zaklady[m.problemSource].problemInform = false;
    break;
   }
  }
 }
}

int main(int argc, char *argv[])
{
 int me, size;
 double times[4] = {0.};
 int send_matrixes = 0;

 //Inicjalizacja OpenMPI
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &me);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 //Na jednym komputerze wszystkie procesy dochodzą do tego momentu jednocześnie
 //+me gwarantuje różne ziarna generatora liczb pseudolowych we wszystkich procesach
 srand(time(0)+me);

 //Wypisanie danych wynikowych
 if(me != 0)
 {
  proces_zaklad(me, zakladNr(me), times);
  printf("Zakład %d: rozwiązywano macierze %lfs, rozwiązywano problemy %lfs\n", zakladNr(me),
times[0], times[1]);
 }
 else
 {
  proces_klient(me, size, &send_matrixes);
  printf("Klient: wysłano %d macierzy\n", send_matrixes);
 }

 //Zwolnienie zasobów OpenMPI
 MPI_Finalize();
}
```