Instytut Informatyki
Politechnika Śląska

Obliczenia równoległe 2

(Specjalność: Oprogramowanie systemowe)

Opracował: Zbigniew J. Czech

materiały dydaktyczne
Gliwice, kwiecień 2011

Spis treści

1 Programowanie równoległe przy użyciu biblioteki MPI

1.1 Prosty program — prog2.c

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  int zrodlo;          /* identyfikator procesu zrodlowego */
  int docel;           /* identyfikator procesu docelowego */
  int typ = 0;         /* typ (znacznik) wiadomosci */
  char wiadomosc[100]; /* pamiec na wiadomosc */
  MPI_Status status;   /* status powrotu dla funkcji receive */

  MPI_Init(&argc, &argv);              /* inicjacja MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &id); /* zwraca identyfikator procesu */
  MPI_Comm_size(MPI_COMM_WORLD, &p);  /* zwraca liczbe procesow */

  if (id == 0) {     /* proces 0 */
    printf("Proces %d: jestem procesem odbierajacym wiadomosci.\n", id);
    for (zrodlo = 1; zrodlo < p; zrodlo++) {
      MPI_Recv(wiadomosc, 100, MPI_CHAR, zrodlo, typ,
               MPI_COMM_WORLD, &status);
      printf("%s\n", wiadomosc);
    }
  }
  else { /* dla procesow o numerach != 0 */
    /* tworzenie wiadomosci */
    sprintf(wiadomosc, "Pozdrowienia od procesu %d!", id);
    docel = 0;
    /* uzycie strlen() + 1, tak aby '\0' zostalo przeslane */
    MPI_Send(wiadomosc, strlen(wiadomosc) + 1, MPI_CHAR, docel, typ,
             MPI_COMM_WORLD);
  }

  /* zamkniecie MPI. */
  MPI_Finalize();

  return 0;

}
```

```
Wykonanie komenda: mpirun -np 5 prog2
------------------------------------------------
Proces 0: jestem procesem odbierajacym wiadomosci.
Pozdrowienia od procesu 1!
Pozdrowienia od procesu 2!
Pozdrowienia od procesu 3!
Pozdrowienia od procesu 4!

Wykonanie komenda:  mpirun -np 15 prog2
------------------------------------------------
Proces 0: jestem procesem odbierajacym wiadomosci.
Pozdrowienia od procesu 1!
```

Tabela 1: Predefiniowane typy MPI, tzw. MPI_datatype's

Typ MPI	Typ w języku C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 cyfr binarnych (8 bitów)
MPI_PACKED	dla MPI_Pack i MPI_Unpack

```
Pozdrowienia od procesu 2!
Pozdrowienia od procesu 3!
Pozdrowienia od procesu 4!
Pozdrowienia od procesu 5!
Pozdrowienia od procesu 6!
Pozdrowienia od procesu 7!
Pozdrowienia od procesu 8!
Pozdrowienia od procesu 9!
Pozdrowienia od procesu 10!
Pozdrowienia od procesu 11!
Pozdrowienia od procesu 12!
Pozdrowienia od procesu 13!
Pozdrowienia od procesu 14!
```

```
int MPI_Send(void*        bufor       /* wej */,
             int          rozmiar     /* wej */,
             MPI_Datatype typ_danych  /* wej */,
             int          docel       /* wej */,
             int          typ         /* wej */,
             MPI_Comm     komunikator /* wej */);

int MPI_Recv(void*        bufor       /* wyj */,
             int          rozmiar     /* wej */,
             MPI_Datatype typ_danych  /* wej */,
             int          zrodlo      /* wej */,
             int          typ         /* wej */,
             MPI_Comm     komunikator /* wej */,
             MPI_Status*  status      /* wyj */);

int MPI_Get_count(
             MPI_Status*  status             /* wej */,
             MPI_Datatype typ_danych,        /* wej */,
             int*         Wskazn_do_dlugosci /* wyj */);
```

1.2 Modyfikacja prostego programu — prog3.c

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  int i;               /* zmienna pomocnicza */
  int docel;           /* identyfikator procesu docelowego */
  int typ = 0;         /* typ (znacznik) wiadomosci */
  char wiadomosc[100]; /* pamiec na wiadomosc */
  MPI_Status status;   /* status powrotu dla funkcji receive */
  int dlug;            /* dlugosc nazwy procesora */
  char nazwa_pr[100];  /* nazwa procesora (wezla) */

  MPI_Init(&argc, &argv);              /* inicjacja MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* zwraca identyfikator procesu */
  MPI_Comm_size(MPI_COMM_WORLD, &p);   /* zwraca liczbe procesow */
  MPI_Get_processor_name(nazwa_pr, &dlug); /* zwraca nazwe procesora */

  if (id == 0) {     /* proces 0 */
    printf("Proces %d w wezle %s: jestem procesem odbierajacym wiadomosci.\n",
           id, nazwa_pr);
    for (i = 1; i < p; i++) {
      MPI_Recv(wiadomosc, 100, MPI_CHAR, MPI_ANY_SOURCE, typ,
               MPI_COMM_WORLD, &status);
      printf("%s\n", wiadomosc);
    }
  }
  else { /* dla procesow o numerach != 0 */
    /* tworzenie wiadomosci */
    sprintf(wiadomosc, "Pozdrowienia od procesu %d w wezle %s!",
            id, nazwa_pr);
    docel = 0;
    /* uzycie strlen() + 1, tak aby '\0' zostalo przeslane */
    MPI_Send(wiadomosc, strlen(wiadomosc) + 1, MPI_CHAR, docel, typ,
             MPI_COMM_WORLD);
  }

  /* zamkniecie MPI. */
  MPI_Finalize();

  return 0;

}
```

```
Wykonanie komenda:  mpirun -np 5 prog3
-----------------------------------------------
Proces 0 w wezle n01: jestem procesem odbierajacym wiadomosci.
Pozdrowienia od procesu 2 w wezle n03!
Pozdrowienia od procesu 1 w wezle n02!
Pozdrowienia od procesu 3 w wezle n04!
Pozdrowienia od procesu 4 w wezle n05!

Wykonanie komenda:  mpirun -np 15 prog3
```

```
-------------------------------------------------
Proces 0 w wezle n01: jestem procesem odbierajacym wiadomosci.
Pozdrowienia od procesu 1 w wezle n02!
Pozdrowienia od procesu 4 w wezle n05!
Pozdrowienia od procesu 5 w wezle n06!
Pozdrowienia od procesu 8 w wezle n09!
Pozdrowienia od procesu 6 w wezle n07!
Pozdrowienia od procesu 10 w wezle n11!
Pozdrowienia od procesu 13 w wezle n14!
Pozdrowienia od procesu 2 w wezle n03!
Pozdrowienia od procesu 12 w wezle n13!
Pozdrowienia od procesu 3 w wezle n04!
Pozdrowienia od procesu 7 w wezle n08!
Pozdrowienia od procesu 11 w wezle n12!
Pozdrowienia od procesu 14 w wezle n15!
Pozdrowienia od procesu 9 w wezle n10!
```

1.3 Równoległy program całkowania — prog4.c

```
/* Rownolegly program calkowania metoda trapezow.
 * Wejscie: brak.
 * Wyjscie: Estymowana wartosc calki oznaczonej od a do b funkcji f(x)
 *          obliczonej z uzyciem n trapezow.
 *
 * Algorytm:
 *   1. Kazdy proces wyznacza "swoj" przedzial calkowania.
 *   2. Kazdy proces wyznacza przyblizenie calki funkcji f(x)
 *      w swoim przedziale stosujac metode trapezow.
 *   3. Kazdy proces != 0 wysyla swoja wartosc calki do procesu 0.
 *   4. Proces 0 sumuje wyniki obliczen otrzymane od poszczegolnych
 *      procesow i drukuje wynik
 * Uwaga: f(x), a, b i n sa ustalone w tekscie programu.
 */

#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  float a = 0.0;       /* lewy koniec przedzialu */
  float b = 0.5;       /* prawy koniec przedzialu */
  int n = 1024;        /* liczba przedzialow (trapezow) */
  float h;             /* dlugosc przedzialu */
  float lokal_a;       /* lokalny, lewy koniec przedzialu */
  float lokal_b;       /* lokalny, prawy koniec przedzialu */
  int lokal_n;         /* lokalna liczba przedzialow */
  float calka;         /* lokalna wartosc calki */
  float suma;          /* pelna wartosc calki */
  int zrodlo;          /* identyfikator procesu zrodlowego */
  int docel = 0;       /* identyfikator procesu docelowego, tj. procesu 0 */
  int typ = 0;         /* typ (znacznik) wiadomosci */
  MPI_Status status;   /* status powrotu dla funkcji receive */
  float obl_calke(float lokal_a, float lokal_b, /* funkcja lokalnego */
                  int lokal_n, float h);        /* obliczania calki  */
```

```
  MPI_Init(&argc, &argv);             /* inicjacja MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &id); /* zwraca identyfikator procesu */
  MPI_Comm_size(MPI_COMM_WORLD, &p);  /* zwraca liczbe procesow */

  h = (b - a) / (float)n; /* h jest to samo dla wszystkich procesow */
  lokal_n = n / p; /* ta sama liczba przedzialow w kazdym procesie  */

  lokal_a = a + (float)id *
            (float)lokal_n * h;  /* wyznaczenie lewego i prawego */
  lokal_b = lokal_a + (float)lokal_n * h; /* konca przedzialu */
  calka = obl_calke(lokal_a, lokal_b, lokal_n, h);

  if (id == 0) {     /* dla procesu 0 */
    suma = calka;
    for (zrodlo = 1; zrodlo < p; zrodlo++) {
      MPI_Recv(&calka, 1, MPI_FLOAT, zrodlo, typ,
               MPI_COMM_WORLD, &status);
      suma = suma + calka;
    }
  }
  else { /* dla procesow o numerach != 0 */
    /* wyslanie lokalnej wartosci calki */
    MPI_Send(&calka, 1, MPI_FLOAT, docel, typ, MPI_COMM_WORLD);
  }

  /* drukowanie wyniku */
  if (id == 0) {
    printf("Liczba procesow: %d\n", p);
    printf("Przy n = %d trapezach, przyblizenie calki\n", n);
    printf("w granicach od %f do %f wynosi %f\n", a, b, suma);
  }

  /* zamkniecie MPI */
  MPI_Finalize();

  return 0;

}

float obl_calke(float lokal_a, float lokal_b,
                int lokal_n, float h) {
  float calka; /* lokalna wartosc calki */
  float x;
  int i;
  float f(float x); /* funkcja ktora calkujemy */

  calka = (f(lokal_a) + f(lokal_b)) / 2.0;
  x = lokal_a;
  for (i = 1; i <= lokal_n - 1; i++) {
    x = x + h;
    calka = calka + f(x);
  }
  calka = calka * h;
  return calka;
} /* obl_calke */

float f(float x) {
```

```
  float wart;
  /* obliczenie f(x) */
  wart = x * x;
  return wart;
} /* f */
```

```
-------------------------------------------------
Liczba procesow: 8
Przy n = 32 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041687
-------------------------------------------------
Liczba procesow: 4
Przy n = 32 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041687
-------------------------------------------------
Liczba procesow: 2
Przy n = 32 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041687
-------------------------------------------------
Liczba procesow: 1
Przy n = 32 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041687
-------------------------------------------------
-------------------------------------------------
Liczba procesow: 8
Przy n = 1024 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Liczba procesow: 4
Przy n = 1024 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Liczba procesow: 2
Przy n = 1024 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Liczba procesow: 1
Przy n = 1024 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
-------------------------------------------------
Liczba procesow: 8
Przy n = 4096 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Liczba procesow: 4
Przy n = 4096 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Liczba procesow: 2
Przy n = 4096 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Liczba procesow: 1
Przy n = 4096 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
```

1.4 Wejście/wyjście w systemach równoległych — prog5.c

```
/* Rownolegly program calkowania metoda trapezow.
 * Wejscie: Dane wejsciowe czytane za pomoca funkcji wez_dane.
 * Wyjscie: Estymowana wartosc calki oznaczonej od a do b funkcji f(x)
 *          obliczonej z uzyciem n trapezow.
 *
 * Algorytm:
 *   1. Kazdy proces wyznacza "swoj" przedzial calkowania.
 *   2. Kazdy proces wyznacza przyblizenie calki funkcji f(x)
 *      w swoim przedziale stosujac metode trapezow.
 *   3. Kazdy proces != 0 wysyla swoja wartosc calki do procesu 0.
 *   4. Proces 0 sumuje wyniki obliczen otrzymane od poszczegolnych
 *      procesow i drukuje wynik
 * Uwaga: f(x) jest ustalona w tekscie programu.
 */

#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  float a;             /* lewy koniec przedzialu */
  float b;             /* prawy koniec przedzialu */
  int n;               /* liczba przedzialow (trapezow) */
  float h;             /* dlugosc przedzialu */
  float lokal_a;       /* lokalny, lewy koniec przedzialu */
  float lokal_b;       /* lokalny, prawy koniec przedzialu */
  int lokal_n;         /* lokalna liczba przedzialow */
  float calka;         /* lokalna wartosc calki */
  float suma;          /* pelna wartosc calki */
  int zrodlo;          /* identyfikator procesu zrodlowego */
  int docel = 0;       /* identyfikator procesu docelowego, tj. procesu 0 */
  int typ = 0;         /* typ (znacznik) wiadomosci */
  MPI_Status status;   /* status powrotu dla funkcji receive */
  float obl_calke(float lokal_a, float lokal_b, /* funkcja lokalnego */
                  int lokal_n, float h);        /* obliczania calki  */
  void wez_dane(float* ods_a, float* ods_b, int* ods_n, int id, int p);

  MPI_Init(&argc, &argv);             /* inicjacja MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &id); /* zwraca identyfikator procesu */
  MPI_Comm_size(MPI_COMM_WORLD, &p);  /* zwraca liczbe procesow */

  wez_dane(&a, &b, &n, id, p);

  h = (b - a) / (float)n; /* h jest to samo dla wszystkich procesow */
  lokal_n = n / p; /* ta sama liczba przedzialow w kazdym procesie  */

  lokal_a = a + (float)id *
            (float)lokal_n * h;  /* wyznaczenie lewego i prawego */
  lokal_b = lokal_a + (float)lokal_n * h; /* konca przedzialu */
  calka = obl_calke(lokal_a, lokal_b, lokal_n, h);

  if (id == 0) {     /* dla procesu 0 */
    suma = calka;
    for (zrodlo = 1; zrodlo < p; zrodlo++) {
```

```
      MPI_Recv(&calka, 1, MPI_FLOAT, zrodlo, typ,
               MPI_COMM_WORLD, &status);
      suma = suma + calka;
    }
  }
  else { /* dla procesow o numerach != 0 */
    /* wyslanie lokalnej wartosci calki */
    MPI_Send(&calka, 1, MPI_FLOAT, docel, typ, MPI_COMM_WORLD);
  }

  /* drukowanie wyniku */
  if (id == 0) {
    printf("Liczba procesow: %d\n", p);
    printf("Przy n = %d trapezach, przyblizenie calki\n", n);
    printf("w granicach od %f do %f wynosi %f\n", a, b, suma);
  }

  /* zamkniecie MPI */
  MPI_Finalize();

  return 0;

}

float obl_calke(float lokal_a, float lokal_b,
                int lokal_n, float h) {
  float calka; /* lokalna wartosc calki */
  float x;
  int i;
  float f(float x); /* funkcja ktora calkujemy */

  calka = (f(lokal_a) + f(lokal_b)) / 2.0;
  x = lokal_a;
  for (i = 1; i <= lokal_n - 1; i++) {
    x = x + h;
    calka = calka + f(x);
  }
  calka = calka * h;
  return calka;
} /* obl_calke */

float f(float x) {
  float wart;
  /* obliczenie f(x) */
  wart = x * x;
  return wart;
} /* f */

/* Funkcja wez_dane sluzaca do czytania wartosci a, b oraz n.
 * Parametry wejsciowe:
 *   1. int id: identyfikator procesu.
 *   2. int p:  liczba procesow.
 * Parametry wyjsciowe:
 *   1. float* ods_a: odsylacz do lewego konca przedzialu.
 *   2. float* ods_b: odsylacz do prawego konca przedzialu.
 *   3. int*   ods_n: odsylacz do liczby przedzialow (trapezow).
 * Dzialanie:
```

```
 *   1. Proces 0 prosi o dane a, b oraz n, oraz odczytuje ich wartosci.
 *   2. Proces 0 przesyla te wartosci do pozostalych procesow.
 */
void wez_dane(float* ods_a /* wyj */,
              float* ods_b /* wyj */,
              int* ods_n   /* wyj */,
              int id /* wej */,
              int p  /* wej */) {

  int zrodlo = 0;  /* dane uzywane w MPI_Send i MPI_Recv */
  int docel;
  int typ;
  MPI_Status status;

  if (id == 0) { /* dla procesu 0 */
    printf("Wprowadz a, b oraz n\n");
    scanf("%f %f %d", ods_a, ods_b, ods_n);
    for (docel = 1; docel < p; docel++) {
      typ = 0;
      MPI_Send(ods_a, 1, MPI_FLOAT, docel, typ, MPI_COMM_WORLD);
      typ = 1;
      MPI_Send(ods_b, 1, MPI_FLOAT, docel, typ, MPI_COMM_WORLD);
      typ = 2;
      MPI_Send(ods_n, 1, MPI_INT,   docel, typ, MPI_COMM_WORLD);
    }
  }
  else {
    typ = 0;
    MPI_Recv(ods_a, 1, MPI_FLOAT, zrodlo, typ,
             MPI_COMM_WORLD, &status);
    typ = 1;
    MPI_Recv(ods_b, 1, MPI_FLOAT, zrodlo, typ,
             MPI_COMM_WORLD, &status);
    typ = 2;
    MPI_Recv(ods_n, 1, MPI_INT, zrodlo, typ,
             MPI_COMM_WORLD, &status);
  }
} /* wez_dane */
```

```
-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 32
Liczba procesow: 8
Przy n = 32 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041687
-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 1024
Liczba procesow: 8
Przy n = 1024 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 4096
Liczba procesow: 8
Przy n = 4096 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
```

1.5 Rozgłaszanie — prog7.c

```
/* Rownolegly program calkowania metoda trapezow.
 * Wejscie: Dane wejsciowe czytane za pomoca funkcji wez_dane2,
 *          w ktorej stosuje sie funkcje MPI_Bcast.
 * Wyjscie: Estymowana wartosc calki oznaczonej od a do b funkcji f(x)
 *          obliczonej z uzyciem n trapezow.
 *
 * Algorytm:
 *   1. Kazdy proces wyznacza "swoj" przedzial calkowania.
 *   2. Kazdy proces wyznacza przyblizenie calki funkcji f(x)
 *      w swoim przedziale stosujac metode trapezow.
 *   3. Kazdy proces != 0 wysyla swoja wartosc calki do procesu 0.
 *   4. Proces 0 sumuje wyniki obliczen otrzymane od poszczegolnych
 *      procesow i drukuje wynik
 * Uwaga: f(x) jest ustalona w tekscie programu.
 */

#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  float a;             /* lewy koniec przedzialu */
  float b;             /* prawy koniec przedzialu */
  int n;               /* liczba przedzialow (trapezow) */
  float h;             /* dlugosc przedzialu */
  float lokal_a;       /* lokalny, lewy koniec przedzialu */
  float lokal_b;       /* lokalny, prawy koniec przedzialu */
  int lokal_n;         /* lokalna liczba przedzialow */
  float calka;         /* lokalna wartosc calki */
  float suma;          /* pelna wartosc calki */
  int zrodlo;          /* identyfikator procesu zrodlowego */
  int docel = 0;       /* identyfikator procesu docelowego, tj. procesu 0 */
  int typ = 0;         /* typ (znacznik) wiadomosci */
  MPI_Status status;   /* status powrotu dla funkcji receive */
  float obl_calke(float lokal_a, float lokal_b, /* funkcja lokalnego */
                  int lokal_n, float h);        /* obliczania calki  */
  void wez_dane2(float* ods_a, float* ods_b, int* ods_n, int id);

  MPI_Init(&argc, &argv);             /* inicjacja MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &id); /* zwraca identyfikator procesu */
  MPI_Comm_size(MPI_COMM_WORLD, &p);  /* zwraca liczbe procesow */

  wez_dane2(&a, &b, &n, id);

  h = (b - a) / (float)n; /* h jest to samo dla wszystkich procesow */
  lokal_n = n / p; /* ta sama liczba przedzialow w kazdym procesie  */

  lokal_a = a + (float)id *
            (float)lokal_n * h;  /* wyznaczenie lewego i prawego */
  lokal_b = lokal_a + (float)lokal_n * h; /* konca przedzialu */
  calka = obl_calke(lokal_a, lokal_b, lokal_n, h);

  if (id == 0) {     /* dla procesu 0 */
    suma = calka;
```

```
    for (zrodlo = 1; zrodlo < p; zrodlo++) {
      MPI_Recv(&calka, 1, MPI_FLOAT, zrodlo, typ,
               MPI_COMM_WORLD, &status);
      suma = suma + calka;
    }
  }
  else { /* dla procesow o numerach != 0 */
    /* wyslanie lokalnej wartosci calki */
    MPI_Send(&calka, 1, MPI_FLOAT, docel, typ, MPI_COMM_WORLD);
  }

  /* drukowanie wyniku */
  if (id == 0) {
    printf("Liczba procesow: %d\n", p);
    printf("Przy n = %d trapezach, przyblizenie calki\n", n);
    printf("w granicach od %f do %f wynosi %f\n", a, b, suma);
  }

  /* zamkniecie MPI */
  MPI_Finalize();

  return 0;

}

float obl_calke(float lokal_a, float lokal_b,
                int lokal_n, float h) {
  float calka; /* lokalna wartosc calki */
  float x;
  int i;
  float f(float x); /* funkcja ktora calkujemy */

  calka = (f(lokal_a) + f(lokal_b)) / 2.0;
  x = lokal_a;
  for (i = 1; i <= lokal_n - 1; i++) {
    x = x + h;
    calka = calka + f(x);
  }
  calka = calka * h;
  return calka;
} /* obl_calke */

float f(float x) {
  float wart;
  /* obliczenie f(x) */
  wart = x * x;
  return wart;
} /* f */

void wez_dane2(float* ods_a /* wyj */,
               float* ods_b /* wyj */,
               int* ods_n   /* wyj */,
               int id       /* wej */) {

  if (id == 0) { /* dla procesu 0 */
    printf("Wprowadz a, b oraz n\n");
    scanf("%f %f %d", ods_a, ods_b, ods_n);
```

```
  }
  MPI_Bcast(ods_a, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(ods_b, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(ods_n, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* wez_dane2 */

-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 32
Liczba procesow: 8
Przy n = 32 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041687
-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 1024
Liczba procesow: 8
Przy n = 1024 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 4096
Liczba procesow: 8
Przy n = 4096 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------


int MPI_Bcast(void*        wiadomosc   /* wej/wyj */,
              int          rozmiar     /* wej */,
              MPI_Datatype typ_danej   /* wej */,
              int          korzen      /* wej */,
              MPI_Comm     komunikator /* wej */);
```

1.6 Redukcja danych — operacje globalne na danych lokalnych — prog8.c

```
/* Rownolegly program calkowania metoda trapezow.
 * Wejscie: Dane wejsciowe czytane za pomoca funkcji wez_dane3,
 *          w ktorej stosuje sie funkcje MPI_Bcast.
 * Wyjscie: Estymowana wartosc calki oznaczonej od a do b funkcji f(x)
 *          obliczonej z uzyciem n trapezow.
 *
 * Algorytm:
 *   1. Kazdy proces wyznacza "swoj" przedzial calkowania.
 *   2. Kazdy proces wyznacza przyblizenie calki funkcji f(x)
 *      w swoim przedziale stosujac metode trapezow.
 *   3. Wyniki obliczen procesorow sa sumowane za pomoca funkcji MPI_Reduce.
 * Uwaga: f(x) jest ustalona w tekscie programu.
 */

#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  float a;             /* lewy koniec przedzialu */
```

```
  float b;              /* prawy koniec przedzialu */
  int n;                /* liczba przedzialow (trapezow) */
  float h;              /* dlugosc przedzialu */
  float lokal_a;        /* lokalny, lewy koniec przedzialu */
  float lokal_b;        /* lokalny, prawy koniec przedzialu */
  int lokal_n;          /* lokalna liczba przedzialow */
  float calka;          /* lokalna wartosc calki */
  float suma;           /* pelna wartosc calki */
  float obl_calke(float lokal_a, float lokal_b, /* funkcja lokalnego */
                  int lokal_n, float h);        /* obliczania calki  */
  void wez_dane3(float* ods_a, float* ods_b, int* ods_n, int id);

  MPI_Init(&argc, &argv);             /* inicjacja MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &id); /* zwraca identyfikator procesu */
  MPI_Comm_size(MPI_COMM_WORLD, &p);  /* zwraca liczbe procesow */

  wez_dane3(&a, &b, &n, id);

  h = (b - a) / (float)n; /* h jest to samo dla wszystkich procesow */
  lokal_n = n / p; /* ta sama liczba przedzialow w kazdym procesie  */

  lokal_a = a + (float)id *
            (float)lokal_n * h;  /* wyznaczenie lewego i prawego */
  lokal_b = lokal_a + (float)lokal_n * h; /* konca przedzialu */
  calka = obl_calke(lokal_a, lokal_b, lokal_n, h);

  MPI_Reduce(&calka, &suma, 1, MPI_FLOAT,
             MPI_SUM, 0, MPI_COMM_WORLD);

  /* drukowanie wyniku */
  if (id == 0) {
    printf("Liczba procesow: %d\n", p);
    printf("Przy n = %d trapezach, przyblizenie calki\n", n);
    printf("w granicach od %f do %f wynosi %f\n", a, b, suma);
  }

  /* zamkniecie MPI */
  MPI_Finalize();

  return 0;

}

float obl_calke(float lokal_a, float lokal_b,
                int lokal_n, float h) {
  float calka; /* lokalna wartosc calki */
  float x;
  int i;
  float f(float x); /* funkcja ktora calkujemy */

  calka = (f(lokal_a) + f(lokal_b)) / 2.0;
  x = lokal_a;
  for (i = 1; i <= lokal_n - 1; i++) {
    x = x + h;
    calka = calka + f(x);
  }
  calka = calka * h;
```

```
  return calka;
} /* obl_calke */

float f(float x) {
  float wart;
  /* obliczenie f(x) */
  wart = x * x;
  return wart;
} /* f */

void wez_dane3(float* ods_a /* wyj */,
               float* ods_b /* wyj */,
               int* ods_n   /* wyj */,
               int id       /* wej */) {

  if (id == 0) { /* dla procesu 0 */
    printf("Wprowadz a, b oraz n\n");
    scanf("%f %f %d", ods_a, ods_b, ods_n);
  }
  MPI_Bcast(ods_a, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(ods_b, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(ods_n, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* wez_dane3 */

-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 32
Liczba procesow: 8
Przy n = 32 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041687
-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 1024
Liczba procesow: 8
Przy n = 1024 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------
Wprowadz a, b oraz n
0 0.5 4096
Liczba procesow: 8
Przy n = 4096 trapezach, przyblizenie calki
w granicach od 0.000000 do 0.500000 wynosi 0.041667
-------------------------------------------------

int MPI_Reduce(void*        dana        /* wej */,
               void*        wynik       /* wyj */,
               int          liczba_wart /* wej */,
               MPI_Datatype typ_danej   /* wej */,
               MPI_Op       operacja    /* wej */,
               int          korzen      /* wej */,
               MPI_Comm     komunikator /* wej */);
```

1.7 Redukcja danych — obliczanie iloczynu skalarnego

```
float Sekw_il(float x[] /* wej */,
```

Tabela 2: Predefiniowane operacje redukcji w MPI

Operacja	Znaczenie
MPI_MAX	maksimum
MPI_MIN	minimum
MPI_SUM	suma
MPI_PROD	iloczyn
MPI_LAND	logiczne and
MPI_BAND	bitowe and (ciągi bitów)
MPI_LOR	logiczne or
MPI_BOR	bitowe or
MPI_LXOR	logiczne xor
MPI_BXOR	bitowe xor
MPI_MAXLOC	maksimum i jego miejsce (location)
MPI_MINLOC	minimum i jego miejsce

```
              float y[] /* wej */,
              int   n   /* wej */) {
  int i;
  float sum = 0.0;

  for (i = 0; i < n; i++)
    sum = sum + x[i] * y[i];

  return sum;
} /* Sekw_il */


float Rown_il(float lokal_x[] /* wej */,
              float lokal_y[] /* wej */,
              int   n_z_kres  /* wej */) {
  float lokal_il;
  float il;
  float Sekw_il(lokal_x, lokal_y, n_z_kres);

  lokal_il = Sekw_il(lokal_x, lokal_y, n_z_kres);
  MPI_Reduce(&lokal_il, &il, 1, MPI_FLOAT,
             MPI_SUM, 0, MPI_COMM_WORLD);

  return il;

} /* Rown_il */


int MPI_Allreduce(void*        dana        /* wej */,
                  void*        wynik       /* wyj */,
                  int          liczba_wart /* wej */,
                  MPI_Datatype typ_danej   /* wej */,
                  MPI_Op       operacja    /* wej */,
                  MPI_Comm     komunikator /* wej */)
```

1.8 Grupowanie danych w celu przyspieszenia komunikacji

1.8.1 Parametr liczba danych

```
float wektor[100];
MPI_Status status;
int moj_nr;
...
...
/* wyznacz wartosci wektora i wyslij */
if (moj_nr == 0) {
  ...
  MPI_Send(wektor+50, 50, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
}
else { /* moj_nr == 1 */
  MPI_Recv(wektor+50, 50, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
}
```

1.8.2 Typy pochodne — funkcja MPI_Type_struct

```
void Budowanie_typu_pochodnego(
        float*        ods_a        /* wej */,
        float*        ods_b        /* wej */,
        int*          ods_n        /* wej */,
        MPI_Datatype* ods_typ_poch /* wyj */) {
                      /* odsylacz do budowanego typu pochodnego */
  /* Liczba elementow w kazdym bloku typu pochodnego.
     U nas - 1 element w kazdym bloku. */
  int dl_blokow[3];

  /* Przesuniecie kazdego bloku od poczatku typu pochodnego.
     MPI_Aint - predefiniowany typ MPI (zwykle int lub long int). */
  MPI_Aint przesuniecia[3];

  /* Typy elementow (a dokladniej - blokow). */
  MPI_Datatype lista_typow[3];

  /* Tablica pomocnicza dla obliczania przesuniec. */
  MPI_Aint adresy[3];

  /*************************************************************/

  /* okreslenie liczby elementow w blokach */
  dl_blokow[0] = dl_blokow[1] = dl_blokow[2] = 1;

  /* okreslenie typow elementow */
  lista_typow[0] = MPI_FLOAT;
  lista_typow[1] = MPI_FLOAT;
  lista_typow[2] = MPI_INT;

  /* obliczanie przesuniec */
  MPI_Address(ods_a, &adresy[0]);
  MPI_Address(ods_b, &adresy[1]);
  MPI_Address(ods_n, &adresy[2]);
  przesuniecia[0] = 0; /* przesuniecie 1-go elementu = 0 */
  przesuniecia[1] = adresy[1] - adresy[0];
  przesuniecia[2] = adresy[2] - adresy[0];
```

```
  /* zbudowanie typu pochodnego */
  MPI_Type_struct(3, dl_blokow, przesuniecia, lista_typow,
                  ods_typ_poch);
  /* zapamietanie typu w systemie MPI, tak aby mogl byc pozniej uzywany */
  MPI_Type_commit(ods_typ_poch);

} /* Budowanie_typu_pochodnego */



void wez_dane4(float* ods_a  /* wej */,
               float* ods_b  /* wej */,
               int*   ods_n  /* wej */,
               int    moj_nr /* wej */) {
  MPI_Datatype trojka; /* typ pochodny zbudowany z a, b oraz n */

  if (moj_nr == 0) {
    printf("Wprowadz a, b oraz n\n");
    scanf("%f %f %d", ods_a, ods_b, ods_n);
  }
  Budowanie_typu_pochodnego(ods_a, ods_b, ods_n, &trojka);
  MPI_Bcast(ods_a, 1, trojka, 0, MPI_COMM_WORLD);

} /* wez_dane4 */


int MPI_Type_struct(int           liczba_blokow  /* wej */,
                    int           dl_blokow[]    /* wej */,
                    MPI_Aint      przesuniecia[] /* wej */,
                    MPI_Datatype  lista_typow[]  /* wej */,
                    MPI_Datatype* ods_typ_poch   /* wyj */);

MPI_Address(void*     zmienna /* wej */,
            MPI_Aint* adres   /* wyj */);
```

1.8.3 Inne konstruktory typów pochodnych

```
if (moj_nr == 0)
  MPI_Send(&(A[2][0]), 10, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
else /* moj_nr == 1 */
  MPI_Recv(&(A[2][0]), 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
```

Konstruktor MPI_Type_vector

```
int MPI_Type_vector(int           liczba_blokow /* wej */,
                    int           dl_bloku      /* wej */,
                    int           skok          /* wej */,
                    MPI_Datatype  typ_elementu  /* wej */,
                    MPI_Datatype* ods_typ_poch  /* wyj */);

MPI_Datatype kolumny;
...
MPI_Type_vector(10, 1, 10, MPI_FLOAT, &kolumny);
MPI_Type_commit(&kolumny);
```

```
if (moj_nr == 0)
  MPI_Send(&(A[0][2]), 1, kolumny, 1, 0, MPI_COMM_WORLD);
else /* moj_nr == 1 */
  MPI_Recv(&(A[0][2]), 1, kolumny, 0, 0, MPI_COMM_WORLD, &status);
```

Konstruktor MPI_Type_indexed

```
int MPI_Type_indexed(int          liczba_blokow  /* wej */,
                     int          dl_blokow[]    /* wej */,
                     int          przesuniecia[] /* wej */,
                     MPI_Datatype  typ_elementu   /* wej */,
                     MPI_Datatype* ods_typ_poch   /* wyj */);
```

```
float        A[n][n]; /* cala tablica */
float        T[n][n]; /* gorny trojkat */
int          dl_blokow[n];
int          przesuniecia[n];
MPI_Datatype trojkat;
...
for (i = 0; i < n; i++) {
  dl_blokow[i] = n - i;
  przesuniecia[i] = (n + 1) * i; /* (n * i + i) */
}
MPI_Type_indexed(n, dl_blokow, przesuniecia, MPI_FLOAT, &trojkat);
MPI_type_commit(&trojkat);

if (moj_nr == 0)
  MPI_Send(A, 1, trojkat, 1, 0, MPI_COMM_WORLD);
else /* moj_nr == 1 */
  MPI_Recv(T, 1, trojkat, 0, 0, MPI_COMM_WORLD, &status);
```

1.9 Zgodność typów

Rozważmy instrukcję:

```
  if (moj_nr == 0)
    MPI_Send(wiadomosc, rozmiar_wysyl, typ_wysyl, 1, 0, MPI_COMM_WORLD);
  else  /* moj_nr == 1 */
    MPI_Recv(wiadomosc, rozmiar_odbier, typ_odbier, 0, 0, MPI_COMM_WORLD, &status);
```

Pytanie 1: Czy `typ_wysyl` i `typ_odbier` muszą być identyczne?

Pytanie 2: Czy `rozmiar_wysyl` i `rozmiar_odbier` muszą być równe?

Definicja typu pochodnego — jest to sekwencja par:

$$\{(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})\}$$

gdzie t_i to podstawowy typ MPI, a d_i przesunięcie w bajtach.

Sygnatura typu pochodnego to sekwencja typów: $\{t_0, t_1, \ldots, t_{n-1}\}$.

Podstawowa zasada zgodności typów w MPI mówi, że sygnatury typów nadawcy i odbiorcy muszą być kompatybilne. Załóżmy, że typ nadawcy jest:

$$\{t_0, t_1, \ldots, t_{n-1}\}$$

a odbiorcy jest:

$$\{u_0, u_1, \ldots, u_{m-1}\}.$$

Typy są kompatybilne jeśli:

a) $n \leqslant m$, oraz

b) $t_i = u_i$, dla $i = 0, 1, \ldots, n$.

Przykład

W podrozdz. 1.8.3 utworzyliśmy typ pochodny `kolumny` o definicji:

```
{(MPI_FLOAT, 0), (MPI_FLOAT, 10 x sizeof(float)),
 (MPI_FLOAT, 20 x sizeof(float)), ..., (MPI_FLOAT, 90 x sizeof(float))}
```

oraz sygnaturze:

$$\{\texttt{MPI\_FLOAT}, ..., \texttt{MPI\_FLOAT}\} \leftarrow \text{powtórzone 10 razy}$$

Korzystając z tego typu można przesłać kolumnę tablicy A i zapamiętać ją w wierszu po stronie odbiorcy:

```
float A[10][10];
...
if (moj_nr == 0)
  MPI_Send(&(A[0][0]), 1, kolumny, 1, 0, MPI_COMM_WORLD);
else  /* moj_nr == 1 */
  MPI_Recv(&(A[0][0]), 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
```

1.10 Funkcje MPI_Pack i MPI_Unpack

```
void wez_dane5(float* ods_a  /* wyj */,
               float* ods_b  /* wyj */,
               int*   ods_n  /* wyj */,
               int    moj_nr /* wej */) {
  char bufor[100]; /* bufor na dane */
  int pozycja;     /* zmienna wskazujaca pozycje danej w buforze */

  if (moj_nr == 0) {
    printf("Wprowadz a, b i n\n");
    scanf("%f %f %d", ods_a, ods_b, ods_n);

    /* pakowanie danych; pozycja = 0 oznacza, ze nalezy rozpoczac
       od poczatku bufora */
    pozycja = 0;
    /* parametr o wartosci &pozycja ma charakter wej/wyj */
    MPI_Pack(ods_a, 1, MPI_FLOAT, bufor, 100,
             &pozycja, MPI_COMM_WORLD);
    /* zmienna pozycja zostala zinkrementowana; teraz wskazuje
       na pierwsze wolne miejsce w buforze */
    MPI_Pack(ods_b, 1, MPI_FLOAT, bufor, 100,
             &pozycja, MPI_COMM_WORLD);
    /* zmienna pozycja zostala ponownie zinkrementowana */
    MPI_Pack(ods_n, 1, MPI_INT, bufor, 100,
             &pozycja, MPI_COMM_WORLD);

    /* rozgloszenie zawartosci bufora */
    MPI_Bcast(bufor, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
  }
  else {
    MPI_Bcast(bufor, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
```

```
    /* rozpakowanie  bufora */
    pozycja = 0;
    MPI_Unpack(bufor, 100, &pozycja, ods_a, 1,
               MPI_FLOAT, MPI_COMM_WORLD);
    /* zmienna pozycja zostala zinkrementowana, teraz wskazuje
       na poczatek b */
    MPI_Unpack(bufor, 100, &pozycja, ods_b, 1,
               MPI_FLOAT, MPI_COMM_WORLD);
    MPI_Unpack(bufor, 100, &pozycja, ods_n, 1,
               MPI_INT, MPI_COMM_WORLD);
  }

} /* wez_dane5 */



int MPI_Pack(void*        pakowana_dana  /* wej */,
             int          rozmiar_danej  /* wej */,
             MPI_Datatype typ_danej      /* wej */,
             void*        bufor          /* wyj */,
             int          rozmiar_bufora /* wej */,
             int*         pozycja        /* wej/wyj */,
             MPI_Comm     komunikator    /* wej */);

int MPI_Unpack(void*        bufor             /* wej */,
               int          rozmiar_bufora    /* wej */,
               int*         pozycja           /* wej/wyj */,
               void*        rozpakowana_dana  /* wyj */,
               int          rozmiar_danej     /* wej */,
               MPI_Datatype typ_danej         /* wej */,
               MPI_Comm     komunikator       /* wej */);


float* niezerowe;      /* elementy niezerowe */
int*   ind_kolumn;     /* indeksy kolumn */
int    liczba_niezer;  /* liczba elementow niezerowych w wierszu */
int    pozycja;        /* zmienna uzywana przy pakowaniu */
int    nr_wiersza;     /* numer wiersza */
char   bufor[ROZMIAR];

MPI_Status status;

if (moj_nr == 0) {
  /* pakowanie i wyslanie */
  pozycja = 0;
  MPI_Pack(&liczba_niezer, 1, MPI_INT, bufor, ROZMIAR,
           &pozycja, MPI_COMM_WORLD);
  MPI_Pack(&nr_wiersza, 1, MPI_INT, bufor, ROZMIAR,
           &pozycja, MPI_COMM_WORLD);
  MPI_Pack(niezerowe, liczba_niezer, MPI_FLOAT, bufor, ROZMIAR,
           &pozycja, MPI_COMM_WORLD);
  MPI_Pack(ind_kolumn, liczba_niezer, MPI_INT, bufor, ROZMIAR,
           &pozycja, MPI_COMM_WORLD);
  MPI_Send(bufor, pozycja, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else { /* moj_nr == 1 */
```

```
  MPI_Recv(bufor, ROZMIAR, MPI_PACKED, 0, 0, MPI_COMM_WORLD,
           &status);
  pozycja = 0;
  MPI_Unpack(bufor, ROZMIAR, &pozycja, &liczba_niezer, 1,
             MPI_INT, MPI_COMM_WORLD);
  MPI_Unpack(bufor, ROZMIAR, &pozycja, &nr_wiersza, 1,
             MPI_INT, MPI_COMM_WORLD);

  /* alokowanie miejsca na elementy i indeksy kolumn */
  niezerowe  = (float *)malloc(liczba_niezer * sizeof(float));
  ind_kolumn = (int *)malloc(liczba_niezer * sizeof(int));

  MPI_Unpack(bufor, ROZMIAR, &pozycja, niezerowe, liczba_niezer,
             MPI_FLOAT, MPI_COMM_WORLD);

  MPI_Unpack(bufor, ROZMIAR, &pozycja, ind_kolumn, liczba_niezer,
             MPI_INT, MPI_COMM_WORLD);
}
```

1.11 Komunikatory

```
MPI_Group grupa_swiat;
MPI_Group grupa_pierwszego_wiersza;
MPI_Comm  komunikator_pierwszego_wiersza;
int*      nry_procesow;

/* utworzenie listy procesow nowego komunikatora */
nry_procesow = (int *)malloc(q * sizeof(int));
for (i = 0; i < q; i++)
  nry_procesow[i] = i;

/* uzyskanie grupy podstawowej MPI_COMM_WORLD */
MPI_Comm_group(MPI_COMM_WORLD, &grupa_swiat);

/* utworzenie nowej grupy */
MPI_Group_incl(grupa_swiat, q, nry_procesow,
               &grupa_pierwszego_wiersza);

/* utworzenie nowego komunikatora */
MPI_Comm_create(MPI_COMM_WORLD, &grupa_pierwszego_wiersza,
                &komunikator_pierwszego_wiersza);



int moj_nr_wpw; /* numer w grupie pierwszego wiersza */
...
MPI_Comm_rank(komunikator_pierwszego_wiersza, &moj_nr_wpw);
MPI_Bcast(&cos, 1, MPI_FLOAT, 0, komunikator_pierwszego_wiersza);


int MPI_Comm_group(MPI_Comm   komunikator /* wej */,
                   MPI_Group* grupa       /* wyj */);

int MPI_Group_incl(MPI_Group  stara_grupa           /* wej */,
                   int        rozmiar_nowej_grupy   /* wej */,
                   int*       nry_w_starej_grupie[] /* wej */,
```

```
                    MPI_Group* nowa_grupa           /* wyj */);

int MPI_Comm_create(MPI_Comm  stary_komunikator /* wej */,
                    MPI_Group nowa_grupa        /* wej */,
                    MPI_Comm* nowy_komunikator  /* wyj */);



MPI_Comm komunikator_mojego_wiersza;
int      moj_wiersz;
...
/* moj_nr jest numerem w MPI_COMM_WORLD. Zachodzi p = q^2. */

moj_wiersz = moj_nr / q;
MPI_Comm_split(MPI_COMM_WORLD, moj_wiersz, moj_nr,
               &komunikator_mojego_wiersza);

int MPI_Comm_split(MPI_Comm  stary_komunikator /* wej */,
                   int       klucz_podzialu    /* wej */,
                   int       klucz_numeru      /* wej */,
                   MPI_Comm* nowy_komunikator  /* wyj */);
```

1.12 Przykład 1 — Sortowanie w czasie $O(\log n)$

Przedstawimy algorytm sortowania o złożoności $O(\log n)$. Użyjemy w nim n^2 procesorów ułożonych w tablicę o rozmiarach $n \times n$. Zadanie polega na posortowaniu tablicy $a[1..n]$ w porządku niemalejącym. Sortowanie zostanie zrealizowane przez wpisanie elementów tablicy a w odpowiednie miejsca tablicy b tak, aby tablica b była posortowana. Zadanie sprowadza się do wyznaczenia indeksu, pod który należy wpisać każdy element $a[i]$ do tablicy b. W celu wyznaczenia indeksów zostaje obliczona tablica $w[1..n, 1..n]$ według następującej reguły:

$$w[i,j] = \begin{cases} 1, & \text{gdy } a[i] > a[j], \text{ lub gdy } a[i] = a[j] \text{ oraz } i > j, \\ 0, & \text{w przeciwnym razie.} \end{cases} \tag{1}$$

Suma jedynek w każdym wierszu i tablicy w określa indeks elementu $a[i]$, pod który należy go wpisać do tablicy b.

Algorytm Sortowanie w czasie $O(\log n)$

Dane wejściowe: Tablica $a[1..n]$ umieszczona w pamięci wspólnej modelu CREW PRAM o n^2 procesorach. Zmienne lokalne przechowujące rozmiar n oraz numer procesora w postaci pary zmiennych i oraz j.

Dane pomocnicze: Tablica $w[1..n, 1..n]$ umieszczona w pamięci wspólnej. Zmienne lokalne k oraz r.

Dane wyjściowe: Tablica $b[1..n]$ umieszczona w pamięci wspólnej zawierająca wartości elementów tablicy $a[1..n]$ w porządku niemalejącym.

1. **begin**
2. **parfor** $P_{i,j}$, $1 \leqslant i, j \leqslant n$ **do**
3. **if** $(a[i] > a[j])$ **or** $((a[i] = a[j])$ **and** $(i > j))$ **then**
4. $w[i,j] := 1$;
5. **else**
6. $w[i,j] := 0$;
7. **end if**;
8. **end for**;
9. $k := n$;
10. **for** $r := 1$ **to** $\lceil \log n \rceil$ **do** {zliczanie jedynek w wierszach tablicy w}
11. **parfor** $P_{i,j}$, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant \lfloor k/2 \rfloor$ **do**
12. $w[i,j] := w[i,j] + w[i, k+1-j]$;
13. **end for**;
14. $k := \lceil k/2 \rceil$;
15. **end for**; {wartości $w[i,1]$ wyznaczają pozycje elementów $a[i]$ w tablicy b}

16. **parfor** $P_{i,j}$, $1 \leqslant i \leqslant n$, $j = 1$ **do**
17. $\quad b[w[i,1]+1] := a[i]$;
18. **end for**;
19. **end**;

Rozważmy przypadek sortowania tablicy $a[1..4] = [5, -1, 2, -1]$. Oznaczmy przez „$\succ$" relację „$(a[i] > a[j])$ **or** $((a[i] = a[j])$ **and** $(i > j))$". W lewej części rys. 1 pokazano relacje pomiędzy elementami tablicy a badane przez poszczególne procesory w celu obliczenia tablicy w, zaś w prawej części — uzyskaną tablicę w dla sortowanej tablicy a. Kolumna $\sum$ zawiera liczby jedynek w poszczególnych wierszach tablicy w, które określają miejsce umieszczenia

$P_{i,j}$	1	2	3	4
1	$a[1] \succ a[1]$	$a[1] \succ a[2]$	$a[1] \succ a[3]$	$a[1] \succ a[4]$
2	$a[2] \succ a[1]$	$a[2] \succ a[2]$	$a[2] \succ a[3]$	$a[2] \succ a[4]$
3	$a[3] \succ a[1]$	$a[3] \succ a[2]$	$a[3] \succ a[3]$	$a[3] \succ a[4]$
4	$a[4] \succ a[1]$	$a[4] \succ a[2]$	$a[4] \succ a[3]$	$a[4] \succ a[4]$

w	1	2	3	4	$\sum$
1	0	1	1	1	3
2	0	0	0	0	0
3	0	1	0	1	2
4	0	1	0	0	1

Rysunek 1: Relacje sprawdzane przez procesory oraz postać tablicy w

elementów tablicy a w tablicy b zgodnie z krokiem 17. algorytmu.

Implementacja w MPI:

```
#include <stdio.h>
#include <mpi.h>

#define N 5 /* rozmiar sortowanej tablicy */

int main(int argc, char** argv) {
    float a[N]={1.4, 1.2, 1.1, 1.1, 0.471}; /* tablica przed sortowaniem */
    float b[N];                             /* tablica po sortowaniu */
    int k, w;             /* zmienne pomocnicze  */
    int i, j;             /* "współrzędne" procesu */
    int id;               /* identyfikator procesu */
    int p;                /* liczba procesów */
    int ind;              /* indeks w tablicy b */
    MPI_Comm kom_wiersza; /* komunikator wiersza */
    MPI_Comm kom_kolumny; /* komunikator kolumny */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    /* czy zgadza się liczba procesów? */
    if((p != N * N)) && (id == 0)) {
      printf("Wymagane %d procesów!\n", N * N); return 1;
    }

    /* obliczenie "współrzędnych" procesu */
    i = id / N; j = id % N;

    if ((a[i] > a[j]) || ((a[i] == a[j]) && (i > j)))  w = 1;
    else w = 0;

    MPI_Comm_split(MPI_COMM_WORLD, i, j, &kom_wiersza);
    MPI_Reduce(&w, &ind, 1, MPI_INT, MPI_SUM, 0, kom_wiersza);

    MPI_Comm_split(MPI_COMM_WORLD, j, ind, &kom_kolumny);
    if (j == 0)
```

```
      MPI_Gather(&a[i], 1, MPI_FLOAT, b, 1, MPI_FLOAT, 0, kom_kolumny);

    if ((j == 0) && (ind == 0)) { /* proces x o minimalnej wartości a[i] */

      printf("a:\n"); /* przed sortowaniem */
      for (k = 0; k < N; k++) printf("%f ", a[k]);
      printf("\nb:\n"); /* po sortowaniu */
      for (k = 0; k < N; k++) printf("%f ", b[k]);
      printf("\n");
    }

    MPI_Comm_free(&kom_wiersza); /* usunięcie komunikatorów */
    MPI_Comm_free(&kom_kolumny);

    MPI_Finalize();
    return 0;
}
```

```
[wcss] zjc@nova ~/paral mpirun -np 25 sort
a:
1.400000 1.200000 1.100000 1.100000 0.471000
b:
0.471000 1.100000 1.100000 1.200000 1.400000
```

```
int MPI_Gather(void*        wysylane_dane     /* wej */,
               int          liczba_danych_wys /* wej */,
               MPI_Datatype typ_danych_wys    /* wej */,
               void*        odbierane_dane    /* wyj */,
               int          liczba_danych_odb /* wej */,
               MPI_Datatype typ_danych_odb    /* wej */,
               int          korzen            /* wej */,
               MPI_Comm     komunikator       /* wej */);
```

1.13 Przykład 2 — Minimalne połowienie grafu

Problem minimalnego połowienia (bisekcji) grafu formułuje się następująco. Dany jest graf $G = (V, E)$, gdzie V jest zbiorem wierzchołków, a E zbiorem krawędzi. Zakładamy, że liczba wierzchołków $|V| = n$ jest parzysta. Należy podzielić zbiór wierzchołków grafu na dwa równe podzbiory Y_1 i Y_2 tak, aby liczba krawędzi łączących wierzchołki podzbiorów Y_1 i Y_2 była minimalna (rys. 2).

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define LPROB 4 /* liczba prob */

int g[37][37];     /* graf - macierz incydencji */
int podzial[37];   /* pozycje 1-18  lewy podgraf
                      pozycje 19-36 prawy podgraf */
int podzial_r[37]; /* roboczy stan tablicy podzial */

int main(int argc, char** argv) {

  int i, j;
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  int zrodlo;          /* identyfikator procesu zrodlowego */
```

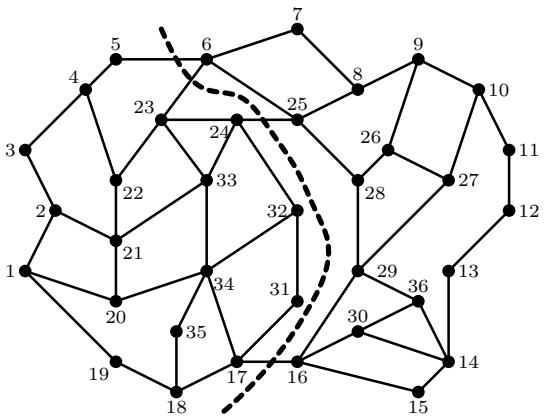
```
int docel;           /* identyfikator procesu docelowego */
MPI_Status status;   /* status powrotu dla funkcji receive */

double czas; /* zmienna do mierzenia czasu */
int min = 32767, t;
int zarodek; /* zarodek generatora */

MPI_Init(&argc, &argv);              /* inicjacja MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &id); /* zwraca identyfikator procesu */
MPI_Comm_size(MPI_COMM_WORLD, &p);  /* zwraca liczbe procesow */

czas = MPI_Wtime();

if (id == 0) { /* proces 0 */

  /* zerowanie grafu */
  for (i = 1; i <= 36; i++)
    for (j = 1; j <= 36; j++)
      g[i][j] = 0;

  /* wpisywanie grafu */
  g[1][2] = 1; g[1][19] = 1; g[1][20] = 1;
  g[2][3] = 1; g[2][21] = 1;
  g[3][4] = 1;
  g[4][5] = 1; g[4][22] = 1;
  g[5][6] = 1;
  g[6][7] = 1; g[6][23] = 1; g[6][25] = 1;
  g[7][8] = 1;
  g[8][9] = 1; g[8][25] = 1;
  g[9][10] = 1; g[9][26] = 1;
  g[10][11] = 1; g[10][27] = 1;
  g[11][12] = 1;
  g[12][13] = 1;
  g[13][14] = 1;
  g[14][15] = 1; g[14][30] = 1; g[14][36] = 1;
  g[15][16] = 1;
  g[16][17] = 1; g[16][29] = 1; g[16][30] = 1;
  g[17][18] = 1; g[17][31] = 1; g[17][34] = 1;
  g[18][19] = 1; g[18][35] = 1;
  g[20][21] = 1; g[20][34] = 1;
  g[21][22] = 1; g[21][33] = 1;
  g[22][23] = 1;
  g[23][24] = 1; g[23][33] = 1;
  g[24][25] = 1; g[24][32] = 1; g[24][33] = 1;
  g[25][28] = 1;
  g[26][27] = 1; g[26][28] = 1;
  g[27][29] = 1;
  g[28][29] = 1;
  g[29][36] = 1;
  g[30][36] = 1;
  g[31][32] = 1;
  g[32][34] = 1;
  g[33][34] = 1;
  g[34][35] = 1;

  /* uczynienie grafu symetrycznym */
  for (i = 1; i <= 36; i++)
```

```
      for (j = 1; j <= 36; j++)
        if (g[i][j] == 1) g[j][i] = 1;

    /* rozeslanie grafu */
    for (docel = 1; docel < p; docel++)
      MPI_Send(g, 37*37, MPI_INT, docel, 0, MPI_COMM_WORLD);

    /* zebranie wynikow */
    for (zrodlo = 1; zrodlo < p; zrodlo++) {
      MPI_Recv(&t, 1, MPI_INT, zrodlo, 1, MPI_COMM_WORLD, &status);
      MPI_Recv(podzial_r, 37, MPI_INT, zrodlo, 2, MPI_COMM_WORLD, &status);
      if (t < min) {
        min = t;
        for (i = 1; i <= 36; i++) podzial[i] = podzial_r[i];
      }
    } /* end for */
    printf("Komenda kompilacji: mpicc -o podzial podzial.c\n");
    printf("Komenda wykonania:  mpirun -np %d podzial %s\n", p, argv[1]);
    printf("Liczba prob (LPROB*(p-1)): %d*%d\n", LPROB, p-1);
    printf("Minimalna liczba krawedzi podzialu: %d\n", min);
    /* wydruk podzialu */
    printf("Skladowe: \n");
    for (i = 1; i <= 18; i++)
      printf("%3d|", podzial[i]);
    printf("\n");
    for (i = 19; i <= 36; i++)
      printf("%3d|", podzial[i]);
    printf("\n");
  } /* koniec procesu 0 */
  else { /* dla procesow o numerach != 0 */

    zarodek = (int)(atoi(argv[1]) * (id+1));  /* wyznacz zarodek generatora */
    srand(zarodek); /* inicjalizacja generatora*/

    /* odebranie grafu */
    MPI_Recv(g, 37*37, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    min = szukaj(); /* szukanie najlepszego podzialu */
    MPI_Send(&min, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Send(podzial, 37, MPI_INT, 0, 2, MPI_COMM_WORLD);
  } /* koniec procesów o nr != 0 */

  if (id == 0) printf("czas: %f sek\n", MPI_Wtime()-czas);

  MPI_Finalize();
  return 0;

} /* end main */


/* szukanie najlepszego podzialu */
int szukaj() {

int i, j, liczba_krawedzi, liczba_prob, liczba_zamian,
    min = 32767, nowa_liczba_krawedzi, t, w, zamiana;
double r; /* liczba losowa z przedzialu [0,1) */

int oblicz();
```

```
  for (liczba_prob = 1; liczba_prob <= LPROB; liczba_prob++) {

    /* inicjalizacja przynaleznosci wierzcholkow do polowek grafu */
    for (i = 1; i <= 36; i++) podzial_r[i] = i;

    /* generowanie losowego podzialu */
    for (i = 36;  i >= 2; i--) {
      r = ((double)rand() / ((double)(RAND_MAX)+(double)(1))); /* r \in [0,1) */
      w = (int) (r * (double) i) + 1; /* losowa pozycja w tablicy */
      t = podzial_r[i]; podzial_r[i] = podzial_r[w]; podzial_r[w] = t;
    }

    /* obliczanie poczatkowej liczby krawedzi miedzy skladowymi */
    liczba_krawedzi = oblicz();

    /* proba zamian wierzcholkow pomiedzy skladowymi */
    liczba_zamian = 0;
    do {
      zamiana = 0;
      for (i = 1; i <= 18; i++) {
        for (j = 19; j <= 36; j++) {

          /* zamiana podzial_r[i] <-> podzial_r[j] */
          t = podzial_r[i]; podzial_r[i] = podzial_r[j]; podzial_r[j] = t;

          /* obliczanie nowej liczby krawedzi po zamianie */
          nowa_liczba_krawedzi = oblicz();

          if (nowa_liczba_krawedzi < liczba_krawedzi) {
            liczba_krawedzi = nowa_liczba_krawedzi;
            zamiana = 1; liczba_zamian = liczba_zamian + 1;
          }
          else { /* odwrocenie zamiany */
            t = podzial_r[i]; podzial_r[i] = podzial_r[j]; podzial_r[j] = t;
          }
        } /* end for j */
      } /* end for i */
    } while (zamiana == 1);

    if (liczba_krawedzi < min) {
      min = liczba_krawedzi;
      for (i = 1; i <= 36; i++) podzial[i] = podzial_r[i];
    } /* end if */
  }   /* end for liczba prob */

  return min;

} /* end szukaj */


/* obliczanie liczby krawedzi miedzy polowkami podzialu */
int oblicz() {
  int i, j, lk;

  lk = 0;
  for (i = 1; i <= 18; i++)
```

```
    for (j = 19; j <= 36; j++)
      if (g[podzial_r[i]][podzial_r[j]] == 1) lk = lk + 1;
  return(lk);
}
```

```
------------------------------------------------------------------------
Komenda kompilacji: mpicc -o podzial podzial.c
Komenda wykonania:  mpirun -np 5 podzial 2345
Liczba prob (LPROB*(p-1)): 4*4
Minimalna liczba krawedzi podzialu: 4
Skladowe:
  6| 15| 13| 30| 36| 29|  8| 25| 14| 28| 27| 10| 12| 11|  7| 26| 16|  9|
 21|  1| 35| 34| 24| 32|  2| 17| 20|  4| 19|  5| 23| 31| 18| 22|  3| 33|
czas: 0.035156 sek
------------------------------------------------------------------------
Komenda kompilacji: mpicc -o podzial podzial.c
Komenda wykonania:  mpirun -np 8 podzial 3782738
Liczba prob (LPROB*(p-1)): 4*7
Minimalna liczba krawedzi podzialu: 4
Skladowe:
 36| 15|  8| 26| 28| 27| 13| 30| 11| 10|  7| 25|  9| 16| 14|  6| 12| 29|
 34|  2| 33|  3| 22| 21| 32| 17| 23| 20| 18|  1| 35| 24|  4|  5| 19| 31|
czas: 0.070312 sek
------------------------------------------------------------------------
Komenda kompilacji: mpicc -o podzial podzial.c
Komenda wykonania:  mpirun -np 11 podzial 3782738
Liczba prob (LPROB*(p-1)): 4*10
Minimalna liczba krawedzi podzialu: 4
Skladowe:
 36| 15|  8| 26| 28| 27| 13| 30| 11| 10|  7| 25|  9| 16| 14|  6| 12| 29|
 34|  2| 33|  3| 22| 21| 32| 17| 23| 20| 18|  1| 35| 24|  4|  5| 19| 31|
czas: 0.121094 sek
------------------------------------------------------------------------
```



Rysunek 2: Minimalne połowienie grafu

Ulepszenie programu z przykładu 2

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define N 37
```

```
#define LPROB 1 /* liczba prob */

int g[N][N];      /* graf - macierz incydencji */
int podzial[N];   /* pozycje 1..(N-1)/2  lewy podgraf
                     pozycje (N+1)/2..(N-1) prawy podgraf */
int podzial_r[N]; /* roboczy stan tablicy podzial */

int main(int argc, char** argv) {

  int i, j;
  int id;              /* identyfikator (ranga) procesu */
  int p;               /* liczba procesow */
  MPI_Status status;   /* status powrotu dla funkcji receive */

  double czas; /* zmienna do mierzenia czasu */
  int zarodek; /* zarodek generatora */

  struct {
    int min; /* minimalna liczba krawedzi */
    int nr;  /* nr procesu */
  } dane, wyniki;

  MPI_Init(&argc, &argv);              /* inicjacja MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &id); /* zwraca identyfikator procesu */
  MPI_Comm_size(MPI_COMM_WORLD, &p);  /* zwraca liczbe procesow */

  czas = MPI_Wtime();
  zarodek = (int)(atoi(argv[1]) * (id+1));  /* wyznacz zarodek generatora */
  srand(zarodek); /* inicjalizacja generatora*/

  if (id == 0) { /* proces 0 */

    /* zerowanie grafu */
    for (i = 1; i < N; i++)
      for (j = 1; j < N; j++)
        g[i][j] = 0;

    /* wpisywanie grafu */
    g[1][2] = 1; g[1][19] = 1; g[1][20] = 1;
    g[2][3] = 1; g[2][21] = 1;
    g[3][4] = 1;
    g[4][5] = 1; g[4][22] = 1;
    g[5][6] = 1;
    g[6][7] = 1; g[6][23] = 1; g[6][25] = 1;
    g[7][8] = 1;
    g[8][9] = 1; g[8][25] = 1;
    g[9][10] = 1; g[9][26] = 1;
    g[10][11] = 1; g[10][27] = 1;
    g[11][12] = 1;
    g[12][13] = 1;
    g[13][14] = 1;
    g[14][15] = 1; g[14][30] = 1; g[14][36] = 1;
    g[15][16] = 1;
    g[16][17] = 1; g[16][29] = 1; g[16][30] = 1;
    g[17][18] = 1; g[17][31] = 1; g[17][34] = 1;
    g[18][19] = 1; g[18][35] = 1;
```

```
  g[20][21] = 1; g[20][34] = 1;
  g[21][22] = 1; g[21][33] = 1;
  g[22][23] = 1;
  g[23][24] = 1; g[23][33] = 1;
  g[24][25] = 1; g[24][32] = 1; g[24][33] = 1;
  g[25][28] = 1;
  g[26][27] = 1; g[26][28] = 1;
  g[27][29] = 1;
  g[28][29] = 1;
  g[29][36] = 1;
  g[30][36] = 1;
  g[31][32] = 1;
  g[32][34] = 1;
  g[33][34] = 1;
  g[34][35] = 1;

  /* uczynienie grafu symetrycznym */
  for (i = 1; i < N; i++)
    for (j = 1; j < N; j++)
      if (g[i][j] == 1) g[j][i] = 1;

  /* rozgloszenie grafu */
  MPI_Bcast(g, N*N, MPI_INT, 0, MPI_COMM_WORLD);

  dane.min = szukaj(); /* szukanie najlepszego podzialu grafu */
  dane.nr = 0; /* proces korzen */

  /* redukcja wynikow */
  MPI_Allreduce(&dane, &wyniki, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
  if (wyniki.nr != 0) {
    MPI_Recv(podzial, N, MPI_INT, wyniki.nr, 1, MPI_COMM_WORLD, &status);
  }

  printf("Komenda kompilacji: mpicc -o podzial podzial.c\n");
  printf("Komenda wykonania:  mpirun -np %d podzial %s\n", p, argv[1]);
  printf("Liczba prob (LPROB*p): %d*%d\n", LPROB, p);
  printf("Minimalna liczba krawedzi podzialu: %d\n", wyniki.min);
  /* wydruk podzialu */
  printf("Skladowe: \n");
  for (i = 1; i <= (N-1)/2; i++)
    printf("%3d|", podzial[i]);
  printf("\n");
  for (i = (N+1)/2; i < N; i++)
    printf("%3d|", podzial[i]);
  printf("\n");
} /* koniec procesu 0 */

else { /* dla procesow o numerach != 0 */

  /* rozgloszenie grafu */
  MPI_Bcast(g, N*N, MPI_INT, 0, MPI_COMM_WORLD);

  dane.min = szukaj(); /* szukanie najlepszego podzialu */
  dane.nr = id;

  MPI_Allreduce(&dane, &wyniki, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
  if (wyniki.nr = id) {
```

```
      MPI_Send(podzial, N, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
  } /* koniec procesów o nr != 0 */

  if (id == 0) printf("czas: %f sek\n", MPI_Wtime()-czas);

  MPI_Finalize();
  return 0;

} /* end main */


/* szukanie najlepszego podzialu */
int szukaj() {
  /* Tekst funkcji nie ulega zmianie. */
}
```

```
------------------------------------------------------------------------
Komenda kompilacji: mpicc -o podzial podzial.c
Komenda wykonania:  mpirun -np 2 podzial 32783
Liczba prob (LPROB*p): 1*2
Minimalna liczba krawedzi podzialu: 8
Skladowe:
  2| 20| 19|  1| 35| 33| 31| 22| 32|  3| 34| 18|  4| 17|  5| 21| 15| 16|
 28|  6| 10| 14| 12| 36| 26| 11| 29| 30|  8| 27| 25| 24| 23|  7| 13|  9|
czas: 0.003906 sek
------------------------------------------------------------------------
Komenda kompilacji: mpicc -o podzial podzial.c
Komenda wykonania:  mpirun -np 3 podzial 38933
Liczba prob (LPROB*p): 1*3
Minimalna liczba krawedzi podzialu: 7
Skladowe:
 32| 14| 30| 17| 16| 15| 19| 20| 11| 18| 13| 29| 31| 35| 12| 34|  1| 36|
 24|  6|  8|  4|  9| 25|  3| 26|  5| 23| 33|  7| 22|  2| 28| 27| 10| 21|
czas: 0.027344 sek
------------------------------------------------------------------------
Komenda kompilacji: mpicc -o podzial podzial.c
Komenda wykonania:  mpirun -np 5 podzial 383894
Liczba prob (LPROB*p): 1*5
Minimalna liczba krawedzi podzialu: 4
Skladowe:
 10| 13|  7| 26| 36| 25|  8| 30| 28| 14| 11| 27| 29| 15| 12|  9| 16|  6|
  1| 34|  5| 35|  2| 23|  4| 21| 33| 20| 17| 22| 24| 18|  3| 32| 31| 19|
czas: 0.226562 sek
------------------------------------------------------------------------
```

Tabela 3: Typy danych biblioteki MPI związane z operacjami `MPI_MAXLOC` i `MPI_MINLOC`

Nazwa	Znaczenie
`MPI_2INT`	Para wielkości całkowitych (`int`)
`MPI_SHORT_INT`	Para wielkości `short` oraz `int`
`MPI_LONG_INT`	Para wielkości `long` oraz `int`
`MPI_LONG_DOUBLE_INT`	Para wielkości `long double` oraz `int`
`MPI_FLOAT_INT`	Para wielkości `float` oraz `int`
`MPI_DOUBLE_INT`	Para wielkości `double` oraz `int`

1.14 Przykład 3 — Wyznaczanie liczb pierwszych

Przedstawimy program wyznaczania liczb pierwszych z przedziału $2\,..\,n$, dla pewnej liczby naturalnej n. Jest on zbudowany na podstawie dekompozycji danych. Korzysta się w nim z obserwacji, iż do wyznaczania liczb pierwszych z przedziału $B = \lfloor\sqrt{n}\rfloor + 1\,..\,n$ wystarczy znajomość liczb pierwszych z przedziału $A = 2\,..\,\lfloor\sqrt{n}\rfloor$. Każda liczba złożona należąca do przedziału B dzieli się bowiem przez jedną lub więcej liczb z przedziału A. Nazwijmy liczby pierwsze z przedziału A podzielnikami. Wówczas, aby sprawdzić czy dowolna liczba $j \in B$ jest złożona, wystarczy sprawdzić, że dzieli się ona bez reszty przez któryś z podzielników. Przykładowo, jeśli chcemy znaleźć liczby pierwsze z przedziału $2\,..\,10000$, to najpierw znajdujemy liczby pierwsze z przedziału $2\,..\,100$ stosując np. sito Eratostenesa. Podzielników takich jest 25, są nimi liczby: 2, 3, 5, ..., 97. Następnie dla dowolnej liczby $j \in B$ należy sprawdzić, czy dzieli sie ona przez któryś z podzielników. Aby zminimalizować koszty komunikacji, w pierwszej fazie programu wszystkie procesy wyznaczają podzielniki z przedziału A. W fazie drugiej następuje dekompozycja danych, tj. podział zakresu liczbowego $B = \lfloor\sqrt{n}\rfloor + 1\,..\,n$ na p podprzedziałów o długości $d = \lceil(n - \lfloor\sqrt{n}\rfloor)/p\rceil$, gdzie p jest liczbą procesów realizujących obliczenia. Procesom o numerach 0, 1, ..., $p-1$ przydzielane są odpowiednio podprzedziały $S+1\,..\,S+d$, $S+1+d\,..\,S+2d$, ..., $S+1+(p-1)d\,..\,n$, gdzie $S = \lfloor\sqrt{n}\rfloor$. W przydzielonych podprzedziałach procesy pracując równolegle wyznaczają liczby pierwsze przez eliminację liczb złożonych będących wielokrotnościami podzielników z przedziału A.

```
/* program sito.c */
/* kompilacja: mpicc -o sito sito.c */
/* wykonanie: mpirun -np 2 sito */


#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <stdlib.h>

#define N 10000000 /* definiuje zakres 2..N */
#define S (int)sqrt(N)
#define M N/10

int main(int argc, char** argv) {
  long int a[S + 1];           /* tablica pomocnicza */
  long int podzielniki[S + 1]; /* podzielniki z zakresu 2..S */
  long int pierwsze[M];        /* liczby pierwsze w podprzedzialach */ \\
  long int liczba, reszta;
  int dl_podprz; /* dlugosc podprzedzialu liczb */
  int llpier;    /* liczba liczb pierwszych w podprzedziale */
  int lpodz;     /* liczba podzielnikow w tablicy podzielniki */
  int liczba_p;  /* sumaryczna liczba znalezionych liczb pierwszych */
  int id;        /* identyfikator procesu */
  int p;         /* liczba procesów */
  int *liczby_grom; /* tablica liczb gromadzonych liczb pierwszych */
  int *przem;       /* tablica przemieszczen */
  int i, llpierx, k;
  double czas;      /* zmienna do mierzenia czasu */
  MPI_Status status;

  MPI_Init(&argc, &argv);
  czas = MPI_Wtime(); /* pomiar czasu */
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  /* wyznaczanie podzielnikow z zakresu 2..S */
  for (i = 2; i <= S; i++) a[i] = 1; /* inicjalizacja */
  i = 1; llpier = lpodz = 0;
  while (i <= S) {
```

```
    i++;
    if (a[i] == 1) {
      podzielniki[lpodz++] = pierwsze[llpier++] = i; /* zapamietanie liczby pierwszej */
      /* wykreslanie liczb zlozonych bedacych wielokrotnosciami i */
      for (k = i + i; k <= S; k += i) a[k] = 0;
    }
  }

  /* rownolegle wyznaczanie liczb pierwszych w podprzedzialach */
  dl_podprz = (N - S) / p; /* obliczanie dlugosci podprzedzialu */
  if ((N - S) % p != 0) dl_podprz++;
  if (id > 0) llpier = 0;
  for (liczba = S + 1 + dl_podprz * id;
       liczba < S + 1 + dl_podprz * (id + 1); liczba++) {
    if (liczba <= N) {
      for (k = 0; k < lpodz; k++) {
        reszta = (liczba % podzielniki[k]);
        if (reszta == 0) break; /* liczba zlozona */
      }
      if (reszta != 0) pierwsze[llpier++] = liczba; /* zapamietanie liczby pierwszej */
    }
  }

  liczby_grom = (int *)malloc(p * sizeof(int));
  MPI_Gather(&llpier, 1, MPI_INT, liczby_grom, 1, MPI_INT, 0,
             MPI_COMM_WORLD);

  przem = (int *)malloc(p * sizeof(int)); /* przemieszczenia */
  przem[0] = 0;
  liczba_p = liczby_grom[0]; /* sumaryczna liczba wyznaczonych liczb pierwszych */
  for (i = 1; i < p; i++) {
    przem[i] = przem[i - 1] + liczby_grom[i - 1];
    liczba_p += liczby_grom[i];
  }

  /* gromadzenie liczb pierwszych w procesie 0 */
  MPI_Gatherv(pierwsze, llpier, MPI_LONG, pierwsze, liczby_grom, przem, MPI_LONG, 0,
             MPI_COMM_WORLD);

  if (id == 0) {
    czas = MPI_Wtime() - czas; /* obliczenie czasu dzialania */
    printf("czas: %f sek\n", czas);
    printf("Liczba procesow: %d N: %d dlugosc_podprzedzialow: %ld\n", p, N, dl_podprz);
  }

  MPI_Finalize();
  return 0;
}


int MPI_Gatherv (
  void*         wysyłane_dane,
  int           liczba_wysyłanych_danych,
  MPI_Datatype  typ_wysyłanych_danych,
  void*         odbierane_dane,
  int           liczby_odbieranych_danych[],
  int           przemieszczenia[],
```

```
MPI_Datatype  typ_odbieranych_danych,
int           korzeń,
MPI_Comm      komunikator);
```

Tabela 4: Liczby liczb pierwszych π_i wyznaczone przez procesy w swoich podprzedziałach, $i = 0, 1, \ldots, p-1$; p oznacza liczbę procesów; $n = 10^7$

p	π_0	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9
1	664579									
2	348620	315959								
3	239248	216201	209130							
4	183235	165385	159686	156273						
5	149103	134177	129651	126891	124757					
6	125998	113250	109372	106829	105278	103852				
7	109295	98111	94614	92595	91165	89886	88913			
8	96655	86580	83621	81764	80278	79408	78573	77700		
9	86742	77597	74909	73165	72002	71034	70364	69739	69027	
10	78717	70386	67857	66320	65340	64311	63771	63120	62686	62071

```
double MPI_Wtime(void);

double start, stop;
...
start = MPI_Wtime();
/* fragment procesu, którego czas
   wykonania jest mierzony */
stop = MPI_Wtime();
printf("Czas wykonania fragmentu wynosi %f sekund\n", stop - start);

double MPI_Tick(void);

double start, stop;
...
MPI_Barrier(komunikator);
if (id == 0) start = MPI_Wtime();
/* program, którego czas
   działania jest mierzony */
MPI_Barrier(komunikator);
if (id == 0) {
   stop = MPI_Wtime();
   printf("Czas wykonania programu wynosi %f sekund\n", stop - start);
}
```

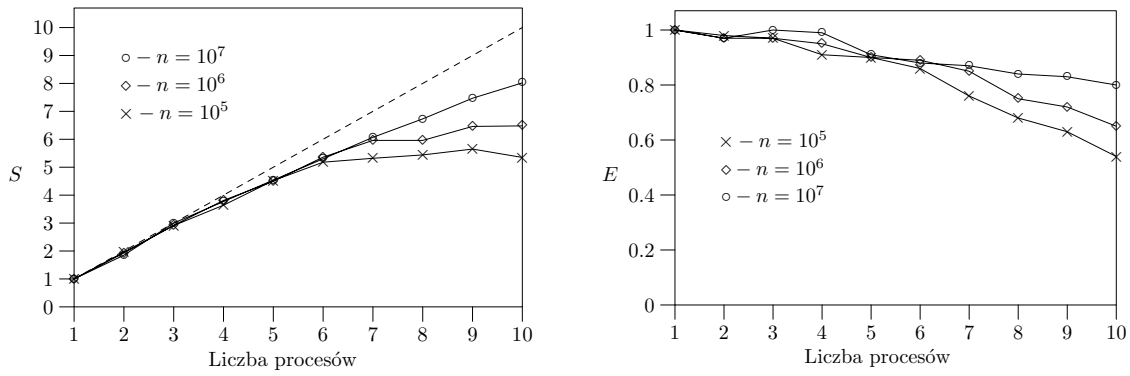
2 Programowanie równoległe przy użyciu interfejsu OpenMP

2.1 Model obliczeń OpenMP

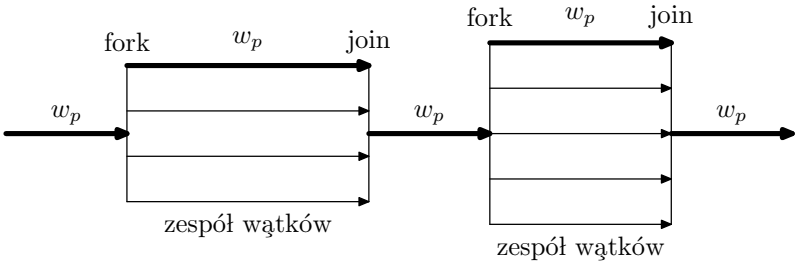
Patrz rys. 4.

2.2 Konstrukcja równoległa

`#pragma omp parallel` [*klauzula*[[,]*klauzula*]...]
ustrukturowany-blok



Rysunek 3: Przyspieszenia S oraz efektywności E w funkcji liczby procesów dla programu MPI wyznaczania liczb pierwszych; linią kreskowaną przedstawiono maksymalne przyspieszenie równe liczbie procesów



Rysunek 4: Tworzenie zespołu wątków w celu wykonania regionów równoległych, w_p oznacza wątek początkowy

```
 1. #include <stdio.h>
 2. #include <omp.h>
 3. int main(int argc, char *argv[]) {
 4.    #pragma omp parallel
 5.    {
 6.       printf("Wykonanie regionu przez watek %d.\n", omp_get_thread_num());
 7.       if (omp_get_thread_num() == 1)
 8.          printf("--- Watek 1 wykonuje dodatkowe obliczenia.\n");
 9.    } /* koniec regionu */
10.    return 0;
11. }
```

```
Wykonanie regionu przez watek 0.
Wykonanie regionu przez watek 3.
Wykonanie regionu przez watek 1.
--- Watek 1 wykonuje dodatkowe obliczenia.
Wykonanie regionu przez watek 2.
```

Kompilacja i wykonanie programu

```
pgcc -mp -o <nazwa-programu-wynikowego> <nazwa-programu-źródłowego>.c
```

```
setenv OMP_NUM_THREADS 4
./<nazwa-programu-wynikowego> <parametry>
```

2.3 Konstrukcja iteracji

`#pragma omp for` [*klauzula*[[,]*klauzula*]...]
instrukcja-for

```
1. #pragma omp parallel default(none) shared(a, b, n) private(i)
2. {
3.    #pragma omp for
4.    for (i = 0; i < n; i++) {
5.       printf("Przebieg %d wykonuje watek %d\n", i,
6.                omp_get_thread_num());
7.       b[i] = a[i] * a[i];
8.    }
9. }
```

```
Przebieg 0 wykonuje watek 0
Przebieg 4 wykonuje watek 0
Przebieg 8 wykonuje watek 0
Przebieg 2 wykonuje watek 2
Przebieg 6 wykonuje watek 2
Przebieg 1 wykonuje watek 1
Przebieg 5 wykonuje watek 1
Przebieg 9 wykonuje watek 1
Przebieg 3 wykonuje watek 3
Przebieg 7 wykonuje watek 3
```

Pełny zapis:

```
#pragma omp parallel [klauzula[[,]klauzula]...]
{
   #pragma omp for [klauzula[[,]klauzula]...]
   instrukcja-for
}
```

Skrót:

```
#pragma omp parallel for [klauzula[[,]klauzula]...]
instrukcja-for
```

```
1. #pragma omp parallel for shared(a, b, n) private(i)
2. for (i = 0; i < n; i++) {
3.    printf("Przebieg %d wykonuje watek %d\n", i,
4.             omp_get_thread_num());
5.    b[i] = a[i] * a[i];
6. }
```

2.4 Konstrukcja sekcji

```
#pragma omp sections [klauzula[[,]klauzula]...]
{
   [#pragma omp section]
   ustrukturowany-blok
   [#pragma omp section
   ustrukturowany-blok]
   ...
}
```

Wersja pełna:

```
 1.  void zadanie_1();
 2.  void zadanie_2();
 3.  void zadanie_3();
     ...
 4.  #pragma omp parallel
 5.  {
 6.     #pragma omp sections
 7.     {
 8.        #pragma omp section
 9.        zadanie_1();
10.        #pragma omp section
11.        zadanie_2();
12.        #pragma omp section
13.        zadanie_3();
14.     }
15.  }
```

Skrót:

```
 4.  #pragma omp parallel sections
 5.  {
 6.     #pragma omp section
 7.     zadanie_1();
 8.     #pragma omp section
 9.     zadanie_2();
10.     #pragma omp section
11.     zadanie_3();
12.  }
```

2.5 Konstrukcja pojedynczego wątku

`#pragma omp single [`*klauzula*`[[,]`*klauzula*`]...]`
ustrukturowany-blok

`#pragma omp master [`*klauzula*`[[,]`*klauzula*`]...]`
ustrukturowany-blok

```
 1.  #pragma omp parallel
 2.  {
 3.     #pragma omp for
 4.     {
 5.        wykonanie pierwszej części obliczeń
 6.     }
 7.     #pragma omp master
 8.     drukowanie wyników pośrednich
 9.     ...wykonanie dalszych części obliczeń
10.  }
```

2.6 Przykład 1 — Sortowanie

```
#include <stdio.h>
#define N 10 /* rozmiar sortowanej tablicy */

int main(int argc, char *argv) {
  float a[N]={1.4, 1.2, 1.1, 1.1, 0.47, /* tablica przed sortowaniem */
              0.99, 7.86, 2.3, 9.0, 6.7};
  float b[N];   /* tablica po sortowaniu */
  int   ind[N]; /* indeks w tablicy b */
```

```
  int   i, j;

  #pragma omp parallel for shared(ind, a, b) private(i, j)
  for (i = 0; i < N; i++) {
    ind[i] = 0;
    for (j = 0; j < N; j++) /* wyznaczanie indeksu dla elementu a[i] */
      if ((a[i] > a[j]) || ((a[i] == a[j]) && (i > j))) ind[i]++;
    b[ind[i]] = a[i]; /* krok sortowania */
  }

  printf("a:\n"); /* przed sortowaniem */
  for (i = 0; i < N; i++) printf("%6.2f", a[i]);
  printf("\nb:\n"); /* po sortowaniu */
  for (i = 0; i < N; i++) printf("%6.2f", b[i]);
  printf("\n");
  return 0;
}

a:
  1.40  1.20  1.10  1.10  0.47  0.99  7.86  2.30  9.00  6.70
b:
  0.47  0.99  1.10  1.10  1.20  1.40  2.30  6.70  7.86  9.00
```

Zagnieżdżona równoległość

```
#include <stdio.h>
#define N 10 /* rozmiar sortowanej tablicy */

int main(int argc, char *argv) {
  float a[N]={1.4, 1.2, 1.1, 1.1, 0.47, /* tablica przed sortowaniem */
              0.99, 7.86, 2.3, 9.0, 6.7};
  float b[N];   /* tablica po sortowaniu */
  int   ind[N]; /* indeks w tablicy b */
  int   i, j;

  #pragma omp parallel for shared(ind, a, b) private(i)
  for (i = 0; i < N; i++) {
    ind[i] = 0;
    #pragma omp parallel for private(j)
    for (j = 0; j < N; j++) /* wyznaczanie indeksu dla elementu a[i] */
      if ((a[i] > a[j]) || ((a[i] == a[j]) && (i > j)))
        #pragma omp critical
        ind[i]++;
    b[ind[i]] = a[i]; /* krok sortowania */
  }

  printf("a:\n"); /* przed sortowaniem */
  for (i = 0; i < N; i++) printf("%6.2f", a[i]);
  printf("\nb:\n"); /* po sortowaniu */
  for (i = 0; i < N; i++) printf("%6.2f", b[i]);
  printf("\n");
  return 0;
}
```

```
    int omp_set_nested(int r);

    int omp_get_nested(void);
```

```
    printf("Zagnieżdżona równoległość jest %s\n",
            omp_get_nested() ? "aktywna" : "niektywana");

    setenv OMP_NESTED true
```

2.7 Przykład 2 — Minimalne połowienie grafu

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <omp.h>
#define N 37    /* dla grafu o 36 wierzcholkach */
#define LWYN 10 /* liczba wynikow */

int g[N][N];      /* graf - macierz incydencji */
struct {
  int min;        /* minimalna liczba krawedzi */
  int podzial[N]; /* pozycje 1..(N-1)/2  lewy podgraf
                     pozycje (N+1)/2..(N-1) prawy podgraf */
} wyniki[LWYN];

int main(int argc, char** argv) {

  int i, j, min, min_p, w, w_min, w_min_p;
  int zarodek;       /* zarodek generatora */
  int szukaj(int b); /* szukanie najlepszego podzialu */

  /* zerowanie grafu */
  for (i = 1; i < N; i++)
    for (j = 1; j < N; j++)
      g[i][j] = 0;

  /* wpisywanie grafu */
  g[1][2] = 1; g[1][19] = 1; g[1][20] = 1;
  g[2][3] = 1; g[2][21] = 1;
  g[3][4] = 1;
  g[4][5] = 1; g[4][22] = 1;
  g[5][6] = 1;
  g[6][7] = 1; g[6][23] = 1; g[6][25] = 1;
  g[7][8] = 1;
  g[8][9] = 1; g[8][25] = 1;
  g[9][10] = 1; g[9][26] = 1;
  g[10][11] = 1; g[10][27] = 1;
  g[11][12] = 1;
  g[12][13] = 1;
  g[13][14] = 1;
  g[14][15] = 1; g[14][30] = 1; g[14][36] = 1;
  g[15][16] = 1;
  g[16][17] = 1; g[16][29] = 1; g[16][30] = 1;
  g[17][18] = 1; g[17][31] = 1; g[17][34] = 1;
  g[18][19] = 1; g[18][35] = 1;
  g[20][21] = 1; g[20][34] = 1;
  g[21][22] = 1; g[21][33] = 1;
  g[22][23] = 1;
  g[23][24] = 1; g[23][33] = 1;
  g[24][25] = 1; g[24][32] = 1; g[24][33] = 1;
  g[25][28] = 1;
```

```
g[26][27] = 1; g[26][28] = 1;
g[27][29] = 1;
g[28][29] = 1;
g[29][36] = 1;
g[30][36] = 1;
g[31][32] = 1;
g[32][34] = 1;
g[33][34] = 1;
g[34][35] = 1;

/* uczynienie grafu symetrycznym */
for (i = 1; i < N; i++)
  for (j = 1; j < N; j++)
    if (g[i][j] == 1) g[j][i] = 1;

min = min_p = INT_MAX;
#pragma omp parallel shared(min, wyniki) private(zarodek, w_min_p) \
                     firstprivate(min_p)
{
  zarodek = (int)(atoi(argv[1]) *
            (omp_get_thread_num()+1)); /* wyznaczanie zarodka generatora */
  srand(zarodek); /* inicjalizacja generatora */

  #pragma omp for /* wyznaczenie wynikow */
  for (w = 0; w < LWYN; w++) {

   /* wydruk kontrolny */
    printf("Watek %d liczy wyniki[%d]\n", omp_get_thread_num(), w);

    wyniki[w].min = szukaj(w); /* szukanie najlepszego podzialu grafu */
  } /* end omp for */

  #pragma omp for nowait /* redukcja wynikow */
  for (w = 0; w < LWYN; w++)
    if (wyniki[w].min < min_p) {
      min_p = wyniki[w].min; w_min_p = w;
    } /* end omp for */

  #pragma omp critical
  if (min_p < min) {
    min = min_p; w_min = w_min_p;
  } /* end omp critical */

} /* end omp parallel */

printf("Przykladowa komenda wykonania: ./podzial 3904893 >podzial-wynik\n");
printf("min: %d w_min: %d\n", min, w_min);
printf("Liczba wynikow: %d\n", LWYN);
for (j = 0; j < LWYN; j++) {
  printf("wyniki[%d]---------Minimalna liczba krawedzi podzialu: %d\n",
         j, wyniki[j].min);
  printf("Skladowe: \n"); /* wydruk podzialu */
  for (i = 1; i <= (N-1)/2; i++)
    printf("%3d|", wyniki[j].podzial[i]);
  printf("\n");
  for (i = (N+1)/2; i < N; i++)
    printf("%3d|", wyniki[j].podzial[i]);
```

```
    printf("\n");
  }

  return 0;

} /* end main */


/* szukanie najlepszego podzialu */
int szukaj(int b) { /* b jest indeksem w tablicy wyniki */

  int i, j, liczba_krawedzi, nowa_liczba_krawedzi, t, w, zamiana;
  double r; /* liczba losowa z przedzialu [0,1) */
  int oblicz(int b); /* obliczanie liczby krawedzi miedzy polowkami grafu */

  /* inicjalizacja przynaleznosci wierzcholkow do polowek grafu */
  for (i = 1; i < N; i++) wyniki[b].podzial[i] = i;

  /* generowanie losowego podzialu */
  for (i = N - 1;  i >= 2; i--) {
    r = ((double)rand() / ((double)(RAND_MAX)+(double)(1))); /* r \in [0,1) */
    w = (int) (r * (double) i) + 1; /* losowa pozycja w tablicy */
    t = wyniki[b].podzial[i]; wyniki[b].podzial[i] = wyniki[b].podzial[w];
    wyniki[b].podzial[w] = t;
  }

  /* obliczanie poczatkowej liczby krawedzi miedzy skladowymi */
  liczba_krawedzi = oblicz(b);

  /* proba zamian wierzcholkow pomiedzy skladowymi */
  do {
    zamiana = 0;
    for (i = 1; i <= (N-1)/2; i++) {
      for (j = (N+1)/2; j < N; j++) {

        /* zamiana wyniki[b].podzial[i] <-> wyniki[b].podzial[j] */
        t = wyniki[b].podzial[i]; wyniki[b].podzial[i] = wyniki[b].podzial[j];
        wyniki[b].podzial[j] = t;

        /* obliczanie nowej liczby krawedzi po zamianie */
        nowa_liczba_krawedzi = oblicz(b);

        if (nowa_liczba_krawedzi < liczba_krawedzi) {
          liczba_krawedzi = nowa_liczba_krawedzi; zamiana = 1;
        }
        else { /* odwrocenie zamiany */
          t = wyniki[b].podzial[i]; wyniki[b].podzial[i] = wyniki[b].podzial[j];
          wyniki[b].podzial[j] = t;
        }
      } /* end for j */
    } /* end for i */
  } while (zamiana == 1);

  return liczba_krawedzi;

} /* end szukaj */
```

```
/* obliczanie liczby krawedzi miedzy polowkami podzialu */
int oblicz(int b) { /* b jest indeksem w tablicy wyniki */
  int i, j, lk = 0;

  for (i = 1; i <= (N-1)/2; i++)
    for (j = (N+1)/2; j < N; j++)
      if (g[wyniki[b].podzial[i]][wyniki[b].podzial[j]] == 1) lk++;
  return(lk);
}
```

Wynik wykonania programu:

```
Watek 0 liczy wyniki[0]
Watek 2 liczy wyniki[4]
Watek 1 liczy wyniki[2]
Watek 2 liczy wyniki[5]
Watek 1 liczy wyniki[3]
Watek 0 liczy wyniki[1]
Watek 3 liczy wyniki[6]
Watek 3 liczy wyniki[7]
Watek 3 liczy wyniki[8]
Watek 3 liczy wyniki[9]
Przykladowa komenda wykonania: ./podzial 3904893 >podzial-wynik
min: 4 w_min: 1
Liczba wynikow: 10
wyniki[0]---------Minimalna liczba krawedzi podzialu: 6
Skladowe:
 13| 17| 35| 11| 10| 14| 36|  9| 27| 30| 15| 16| 29| 18| 12| 28| 26| 19|
 21| 23| 33|  4|  1| 20| 24|  8| 32|  2|  7| 31|  5| 22|  6|  3| 34| 25|
wyniki[1]---------Minimalna liczba krawedzi podzialu: 4
Skladowe:
 34| 35| 24|  1| 17| 31| 33| 18| 32|  4| 23|  5| 22| 21| 20| 19|  3|  2|
 14| 16| 10| 13|  8| 25| 28|  7| 26| 27| 30| 29| 36|  6|  9| 15| 12| 11|
wyniki[2]---------Minimalna liczba krawedzi podzialu: 8
Skladowe:
 13| 29| 17| 18| 32| 27| 30| 14| 16| 15| 31| 19| 12| 28| 34| 35| 26| 36|
  4|  2| 33| 25| 24|  7| 20| 22| 11|  6|  3|  1|  5|  8| 21| 10| 23|  9|
wyniki[3]---------Minimalna liczba krawedzi podzialu: 10
Skladowe:
 22| 18| 34| 19| 21| 31| 33| 10| 23| 26|  1| 27| 20| 24|  9| 35| 32| 17|
 15| 13|  2|  8|  4|  5| 11| 16| 12| 30| 29|  3|  6| 25| 14|  7| 28| 36|
wyniki[4]---------Minimalna liczba krawedzi podzialu: 12
Skladowe:
 18|  1| 29| 16| 15| 28| 14|  8| 36| 25| 35| 19| 17| 34| 20|  7|  6| 30|
  5| 32| 27|  4| 26| 33|  3| 10|  2| 24| 11| 23| 31| 21|  9| 22| 13| 12|
wyniki[5]---------Minimalna liczba krawedzi podzialu: 7
Skladowe:
 13| 31|  1| 29| 14| 35| 20| 36| 34| 12| 16| 11| 30| 18| 19| 15| 17| 32|
 22|  5| 26|  8|  7| 23|  6|  4| 24| 25| 33| 10| 28|  9| 21| 27|  3|  2|
wyniki[6]---------Minimalna liczba krawedzi podzialu: 11
Skladowe:
 27| 25| 19| 10| 16|  8|  7| 17| 18| 35|  9|  6| 26| 11|  5| 31| 28| 29|
 33| 30| 14|  2| 22| 12| 34|  1| 23| 36| 32|  4| 21| 20| 13|  3| 24| 15|
wyniki[7]---------Minimalna liczba krawedzi podzialu: 10
Skladowe:
  1| 29| 19| 26| 28|  9|  8| 12| 18| 27| 10|  7| 11| 13|  5| 25|  6| 35|
```

```
 16| 33| 20|  3| 22|  2| 31| 32| 24| 34| 36| 15| 14| 21| 30| 17|  4| 23|
wyniki[8]---------Minimalna liczba krawedzi podzialu: 8
Skladowe:
 26| 23| 11| 21| 29| 27|  6| 33| 22| 24| 25| 28| 10|  8|  5|  7|  4|  9|
 14| 18| 35| 19| 30| 36| 17| 16|  2| 32| 15| 34| 20| 31| 13|  1|  3| 12|
wyniki[9]---------Minimalna liczba krawedzi podzialu: 8
Skladowe:
 21| 36| 18| 19| 30| 35| 33|  2| 16| 14| 20| 34| 31|  1| 32| 15| 24| 17|
  8|  5| 28| 27|  9|  7| 10| 29| 13|  4| 25|  3| 23|  6| 26| 12| 22| 11|
```

2.8 Przykład 3 — Wyznaczanie liczb pierwszych

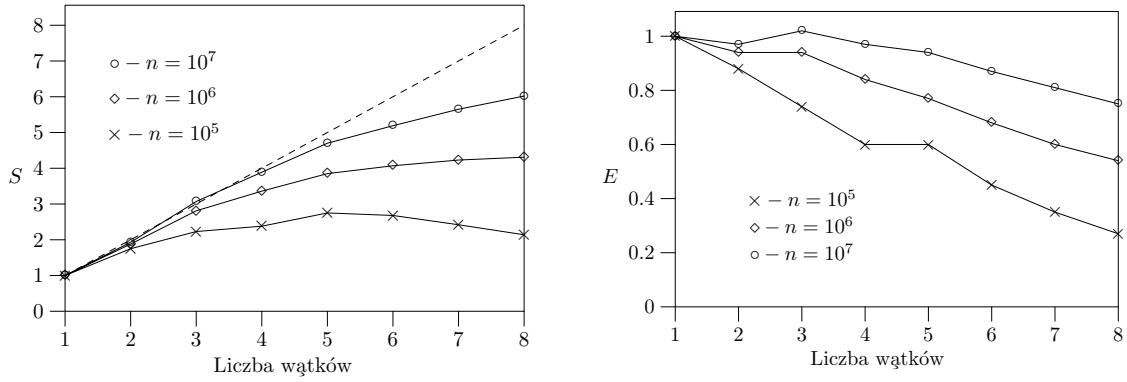
Program wyznaczania liczb pierwszych z przedziału $2..n$ skonstruowany przy użyciu biblioteki MPI był omawiany na str. 34. Przypomnijmy, że program składa się z dwóch faz. W pierwszej z nich wyznacza się liczby pierwsze z przedziału $A = 2\,..\,\lfloor\sqrt{n}\rfloor$, zwane podzielnikami, stosując sito Eratostenesa. W drugiej fazie poszukuje się liczb pierwszych w przedziale $B = \lfloor\sqrt{n}\rfloor+1\,..\,n$. Dowolna liczba $j \in B$ jest liczbą pierwszą, jeśli nie dzieli się bez reszty przez żaden z podzielników z przedziału A. Program wielowątkowy oparty na tej idei został przedstawiony poniżej. Najpierw wyznacza się podzielniki z przedziału A, tj. 2..`S`. Tylko inicjalizacja pomocniczej tablicy `a` przebiega równolegle. Dalszy fragment będący implementacją sita Eratostenesa wykonywany jest sekwencyjnie. Wyznaczanie liczb pierwszych z przedziału B, tj. `S`+1..N, odbywa się przy użyciu zespołu wątków. Znalezione liczby pierwsze gromadzone są w tablicy `pierwsze`. Zapis liczby do tablicy oraz aktualizacja indeksu `llpier` w tej tablicy realizowane są wewnątrz sekcji krytycznej. Zauważmy, że w programie MPI liczby pierwsze z przedziału A były gromadzone w tablicy `podzielniki`. W przedstawionym programie liczby te przechowywane są w tablicy `pierwsze` na pozycjach od 0 do `lpodz` − 1.

```
/* Wyznaczanie liczb pierwszych - sito Eratostenesa */
/* program sito.c */
/* kompilacja:
use_pgi
pgcc -mp -o sito -Minfo sito-2.c    */
/* wykonanie:
#PBS -N sito-4
#PBS -S /bin/csh
#PBS -l nodes=1:ppn=4
#PBS -l mem=100mb
#PBS -l walltime=00:10:00
cd ~zjc/omp
setenv OMP_NUM_THREADS 4
./sito >wynik-sito-4      */

#include <stdio.h>
#include <math.h>
#include <omp.h>
#include <stdlib.h>

#define N 10000000 /* definiuje zakres 2..N */
#define S (int)sqrt(N)
#define M N/10

int main(int argc, char** argv) {
  long int a[S + 1];           /* tablica pomocnicza */
  long int podzielniki[S + 1]; /* podzielniki z zakresu 2..S */
  long int pierwsze[M];        /* liczby pierwsze w przedziale 2 .. N */
  long int i, k, liczba, reszta;
  long int lpodz = 0;   /* liczba podzielnikow w tablicy podzielniki */
  long int llpier = 0;  /* liczba liczb pierwszych w tablicy pierwsze */
  double czas;     /* zmienna do mierzenia czasu */
```

```
  omp_set_num_threads(2);
  printf("Liczba watkow: %d\n", omp_get_max_threads());
  czas = omp_get_wtime(); /* pomiar czasu */

  /* wyznaczanie podzielnikow z zakresu 2..S */
  #pragma omp parallel for default(none) shared(a)
  for (i = 2; i <= S; i++) a[i] = 1; /* inicjalizacja */

  for (i = 2; i <= S; i++)
    if (a[i] == 1) {
      pierwsze[llpier++] = podzielniki[lpodz++] = i; /* zapamietanie liczby pierwszej */
      /* wykreslanie liczb zlozonych bedacych wielokrotnosciami i */
      for (k = i + i; k <= S; k += i) a[k] = 0;
    }

  /* rownolegle wyznaczanie liczb pierwszych */
  #pragma omp parallel for default(none) private(k, reszta) \
                           shared(llpier, lpodz, pierwsze, podzielniki)
  for (liczba = S + 1; liczba <= N; liczba++) {
    for (k = 0; k < lpodz; k++) {
      reszta = (liczba % podzielniki[k]);
      if (reszta == 0) break; /* liczba zlozona */
    }

    if (reszta != 0) {
      #pragma omp critical
      pierwsze[llpier++] = liczba; /* zapamietanie liczby pierwszej */
    }
  }

  czas = omp_get_wtime() - czas; /* obliczenie czasu dzialania */
  printf("czas: %f sek wtick: %f sek\n", czas, omp_get_wtick());
  printf("Liczba liczb pierwszych: %d\n", llpier);
  printf("------------ lpodz: %d \n", lpodz);

  /* printf("Wszystkie liczby pierwsze ***********\n");
  for (i = 0; i < llpier; i++) {
    printf(" %ld ", pierwsze[i]);
    if (i % 10 == 0) printf("\n");
  }
  printf("\n*************\n"); */

  return 0;
}
```

Literatura

1. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R., Parallel programming in OpenMP, Morgan Kaufmann, Academic Press, 2001.
2. Chapman, B., Jost, G., van der Pas, R., Using OpenMP. Portable shared memory parallel programming, MIT Press, Cambridge, MA, 2008.
3. Czech, Z.J., Wprowadzenie do obliczeń równoległych, PWN, Warszawa 2010.
4. Herlichy, M., Shavit, N., Sztuka programowania wieloprocesorowego, PWN, Warszawa, 2010.



Rysunek 5: Przyspieszenia S oraz efektywności E w funkcji liczby procesów dla programu OpenMP wyznaczania liczb pierwszych; linią kreskowaną przedstawiono maksymalne przyspieszenie równe liczbie wątków

5. Message Passing Interface Forum. MPI2: A Message Passing Interface standard, International Journal of High Performance Computing Applications 12, (1-2) (1998), 1-299.
6. Pacheco, P. S., Parallel programming with MPI, Morgan Kaufmann Pubs., 1997.
7. Quinn, M. J., Parallel programming in C with MPI and OpenMP, McGraw-Hill, 2004.
8. Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., Dongarra, J., MPI - The Complete Reference: Volume 1. The MPI Core, 2nd edition, MIT Press, Cambridge, MA, 1998.
9. Witryna: www.compunity.org (OpenMP).
10. Witryna: www.lam-mpi.org.
11. Witryna: www.mpi-forum.org.
12. Witryna: www.netlib.org/mpi.
13. Witryna: www.openmp.org.
14. Witryna: www-unix.mcs.anl.gov/mpi.