14.3. Большое домашнее задание. Реализовать программу нахождения простых чисел в заданном диапазоне с использованием технологии MPI [1] и провести вычислительный эксперимент: определить время в секундах и в условных единицах и ускорение при использовании 2,3, 4, 5, 6 потоков, приняв значение переменной porog=6001.

5.1. Задача нахождения простых чисел в заданном диапазоне

Для разработки алгоритмов шифрования с открытым ключом используется целый ряд понятий теории чисел [3]. Одним из главных объектов теории чисел являются простые числа. В данном подразделе мы рассмотрим задачу по нахождению простых чисел в заданном диапазоне. Для начала давайте вспомним, какое число является простым.

Простым называют целое число, больше единицы, единственными множителями которого является 1 и оно само. Простое число не делится ни на одно другое число [2]. Самым простым способом проверки, простое ли данное число, является перебор делителей. Необходимо попытаться разделить проверяемое число на все возможные делители. Если оно не делится ни на один из них, то проверяемое число является простым. Однако данный подход является достаточно трудоемким. Поэтому для нахождения всех возможных простых чисел в заданном диапазоне предлагается использовать распределенные вычисления.

Как правило, параллельная программа, написанная с использованием функции MPI, состоит из главного процесса, управляющего вычислениями, и всех остальных процессов, участвующих в вычислениях. Таким образом, при рассмотрении задачи определения простых чисел главный процесс должен распределить заданный диапазон поиска равномерно между всеми процессами. Пусть в наших вычислениях участвует **size** процессов и диапазон поиска задается от 2 (так как 0 и 1 не являются простыми числами) до переменной величины **porog**. Тогда каждому из процессов необходимо будет проанализировать **diapazon** чисел, равный целому значению от деления общего числа анализируемых чисел **porog** на количество

Таблица 5.1

Пример распределения данных между процессами

№ процесса	porog	size	diapazon	ost	(номер процесса - 1)*diapason + 2	номер процесса*diapason + 1	(size − 1)* *diapason + + 2	size*diapason + ost
0	110	4	25	10	–	–	$(4-1)\cdot 25+$ $+ 2 = 77$	$4\cdot 25+$ $+ 10 = 110$
1	110	4	25	10	$(1-1)\cdot 25+2=2$	$1\cdot 25+1=26$	–	–
2	110	4	25	10	$(2-1)\cdot 25+2=27$	$2\cdot 25+1=51$	–	–
3	110	4	25	10	$(3-1)\cdot 25+2=52$	$3\cdot 25+1=76$	–	–

процессов **size**, участвующих в вычислениях. Таким образом, получается, что всем процессам, за исключением главного процесса, необходимо проанализировать числа от значения ((номер процесса − 1) * diapason + 2) до значения (номер процесса * diapason + 1). Так как диапазон поиска не всегда делится нацело на число процессов вычисления, то для главного процесса диапазон анализа будет больше, чем у остальных процессов на остаток (**ost**) от деления общего числа анализируемых чисел на количество процессов. И, таким образом, анализ будет проходить от значения (**(size - 1) * diapason + 2**) до значения **(size * diapazon) + ost**.

Поясним сказанное на примере. Пусть нам необходимо провести поиск простых чисел в диапазоне от 2 до 110 (т. е. **porog = 110**). И пусть в вычислениях принимает участие 4 процесса (**size = 4**). Тогда данные между процессами распределятся так, как показано в табл. 5.1.

После того как каждым из процессов будет найдены все возможные простые числа в указанном диапазоне, им необходимо передать полученные данные главному процессу, который объединит их в один выходной файл с именем «простые числа.txt». Для передачи данных от главного процесса всем остальным мы будем использовать функцию **MPI_BCAST**, которая посылает сообщение из корневого процесса всем процессам группы (см. Приложение Б). Ниже приводится листинг программы с подробными комментариями.

Листинг программы

```
#include "stdafx.h"
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#include <time.h>
int main(int argc, char* argv[])
```

```
{
  int *chisla, *chisla_buf;
  int myrank, size; // Номер текущего процесса и общее число процессов
  int namelen, num_proc;
  int TAG=0;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  double startwtime, endwtime;
  FILE *prchisla;
  int fl;
  char ch[50];
  int porog, diapason, ost, start, end;
  int i, j, flag;
  int buf, kol;
  MPI_Init(&argc, &argv); // Инициализируем работу программы MPI
  // Инициализируем счетчик отсчета времени работы программы
  // Определяем номер процесса
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  // Определяем общее число процессов
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  // Определяем имя компьютера, на котором запущен процесс
  MPI_Get_processor_name (processor_name, &namelen);
  // Выводим полученную информацию на экран
  fprintf(stderr, "\nПроцесс %d начал работу на компьютере %s\n», myrank,
  processor_name);
  fflush (stderr);
  if (myrank==0) // Если это главный процесс, то
  {
    // Открываем на запись файл, в который будем записывать
    // найденные простые числа
    prchisla=fopen("простые числа.txt", "w");
    // Вводим число, в пределах которого необходимо будет найти
    // все простые числа
    printf("\n \Введите диапазон для определения простых чисел:");
    fflush (stdout);
    do
    {
      gets(ch); // Считываем вводимое число посимвольно
      fl=1;
      for (i=0; i< strlen(ch); i++)
      {
        if ((ch[i]<'0') ||(ch[i]>'9'))// и проверяем каждый символ
        fl=0;
      }
      if (fl==0) // Если хотя бы один из символов введен не верно,
      // то выводим соответствующее сообщение об ошибке
```

```
    {
      printf("\n Ошибка. Попробуйте еще раз.");
      fflush(stdout);
      // Повторяем попытку ввода
      printf("\n Введите диапазон для определения простых чисел:");
      fflush(stdout);
    }
  }
  while (fl==0);
  porog = atoi(ch); // Преобразуем введеное число
  startwtime = MPI_Wtime(); // Инициализируем счетчик отсчета
  // времени расчетов
  diapazon = porog/size; // Количество чисел для анализа каждым
  // процессом
  ost = porog - diapazon*size; // Остаток от деления
  // Главный процесс посылает всем остальным процессам диапазон
  // анализа
  MPI_Bcast(&diapazon, 1, MPI_INT,0,MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);
  // Ожидаем пока все процессы получат данные
  start = (size-1)*diapazon + 2; // Определяем начальное число анализа
  end = (size*diapazon) + ost;// Определяем конечное число анализа
  chisla=(int*)malloc((diapazon+ost)*sizeof(int));
  // Инициализируем массивы для хранения результатов анализа
  chisla_buf=(int*)malloc(diapazon*sizeof(int));
  for (i=0; i< (diapazon+ost); i++)
    chisla[i]=0; // Обнуляем инициализированные массивы
  for (i=0; i< diapazon; i++)
    chisla_buf[i]=0;
  kol = 0;
  if (start == 2)// Если начальное число анализа равно 2,
  {
    chisla[kol]=2; // то заносим его в массив результата
    kol++;
    start= start+1;
  } // и переходим к дальнейшему анализу
  for (i=start; i<=end; i++) // Поочередно берем каждый из
  // элементов
  {
    j=1;
    flag=1; // Устанавливаем флаг в единицу
    do
    {
      j++;
      // Начинаем делить на все возможные делители
      // Первый раз j = 2.
```

```
            buf = i/j;
            if ((buf*j)==i) // Если число поделилось нацело,
            // то оно не является простым
            flag = 0; // и поэтому флаг устанавливается в 0
        }
        // Анализ продолжается, пока не будут проверены все делители
        // и флаг будет равен единице
        while((j< (i-1))&&(flag==1));
            if (flag==1) // Если флаг остался равен единице,
            {
                chisla[kol] = j+1; // то это простое число
                // и мы заносим его в массив
                kol++;
            }
    }
    if (size> 1) // Если в вычислениях участвует более одного процесса,
    {
        for (j = 1; j< size; j++)
        {
            // то принимаем данные от каждого из них
            num_proc = j;
            MPI_Recv(chisla_buf, diapazon, MPI_INT, num_proc,
            TAG, MPI_COMM_WORLD, &status);
            i = 0;
            while ((i< diapazon)&&(chisla_buf[i]!=0))
            {
                if (chisla_buf[i]!=0) // Заносим принятые данные в выходной файл
                {
                    fprintf(prchisla, "%d \n", chisla_buf[i]);
                    fflush (prchisla);
                }
                i++;
            }
        }
    }
    i = 0; // После этого заносим в выходной файл данные,
    // полученные главным процессом
    while ((i< (diapazon+ost))&&(chisla[i]!=0))
    {
        if (chisla[i]!=0)
        {
            fprintf(prchisla, "%d \n", chisla[i]);
            fflush (prchisla);
        }
        i++;
    }
```

```
   MPI_Barrier(MPI_COMM_WORLD)
   // Ожидаем пока все процессы вызовут эту функцию
   endwtime = MPI_Wtime(); // Выводим на экран
   printf("\n wall clock time =  %f\n", endwtime-startwtime);
   // Время работы программы
   fflush(stdout);
   printf("\nАнализ завершен!\n");
   fflush(stdout);
   free(chisla); // Освобождаем динамическую память
   free(chisla_buf);
   fclose(prchisla); // Закрываем выходной файл
}
else // Если это не главный процесс,
{
   // то принимаем данные от главного процесса
   MPI_Bcast(&diapazon, 1, MPI_INT, 0, MPI_COMM_WORLD);
   printf("\n Процесс  %d получил данные от главного процеса,
   diapazon= %d\n", myrank, diapazon);
   fflush(stdout);
   MPI_Barrier(MPI_COMM_WORLD); // Ожидаем пока все процессы
   // вызовут эту функцию
   start = (myrank-1)*diapazon + 2; // Определяем начальное число анализа
   end = (myrank*diapazon) + 1; // Определяем конечное число анализа
   // Инициализируем массив для хранения результатов анализа
   chisla=(int*)malloc(diapazon*sizeof(int));
   for (i=0; i< (diapazon); i++)
        chisla[i]=0; // Обнуляем инициализированные массивы
   kol = 0;
   if (start == 2)// Если начальное число анализа равно 2,
   {
      chisla[kol]=2; // то заносим его в массив результата
      kol++; // и переходим к дальнейшему анализу
      start++;
   }
   for (i=start; i<=end; i++)// Поочередно берем каждый из элементов
   {
      j=1;
      flag=1; // Устанавливаем флаг в единицу
      do
      {
         j++; // Начинаем делить на все возможные делители
         // Первый раз j = 2.
         buf = i/j;
         if ((buf*j)==i) // Если число поделилось нацело,
         // то оно не является простым
         flag = 0; // и поэтому флаг устанавливается в 0
```

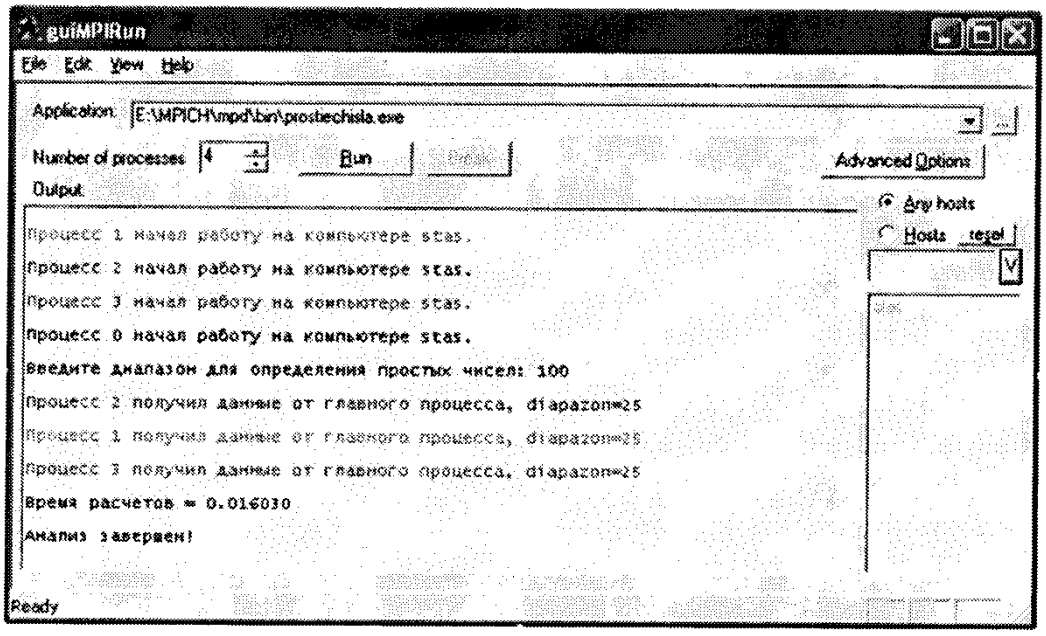

Рис. 5.1. Пример работы программы

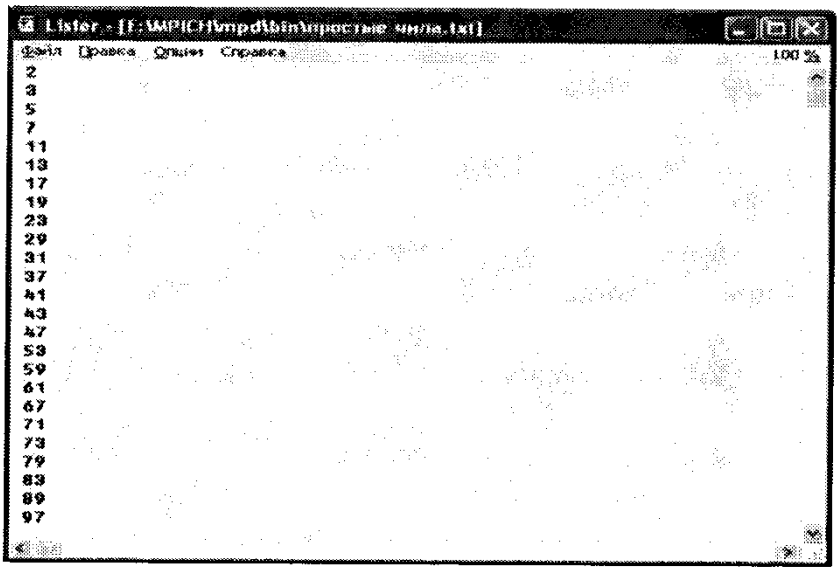

Рис. 5.2. Результат работы программы

```
}
// Анализ продолжается, пока не будут проверены
// все делители и флаг будет равен единице
while((j< (i-1))&&(flag==1));
   if (flag==1) // Если флаг остался равен единице,
   { // то это простое число и мы заносим его в массив
      chisla[kol] = j+1;
      kol++;
   }
}
// Передаем массив с результатами анализа главному процессу
MPI_Send(chisla,diapazon,MPI_INT,0,TAG,MPI_COMM_WORLD);
```

```
    MPI_Barrier(MPI_COMM_WORLD);
    // Ожидаем пока все процессы вызовут эту функцию
    free(chisla_buf); // Освобождаем динамическую память
  }
  MPI_Finalize(); // Завершаем работу программы MPI
  return 0;
}
```

Пример работы программы, запущенной с использованием guiMPIRun.exe, приведен на рис. 5.1. Результат работы программы заносится в файл и отражен на рис. 5.2.

5.2. Задача разложения произведения на простые сомножители

Известно, что одна и та же задача может быть решена различными способами даже с использованием одних и теж же инструментов. Рассмотрим два способа решения одной задачи разложения на простые сомножители с использованием биьблиотеки MPI.

5.2.1. Первый вариант решения

Задача разложения числа на множители является одной из древнейшей в теории чисел. Этот процесс несложен, но занимает много времени. Существует достаточно много различных алгоритмов решения этой задачи. Среди них такие, как решето числового поля, квадратичное решето, метод эллиптических кривых, алгоритм Монте-Карло Полларда и др. (более подробно см. [19]).

Мы рассмортим самый старый и самый простой алгоритм, который заключается в пробном делении. Предложенное к анализу число мы будем пробовать делить на простые числа. Для того чтобы не осуществлять каждый раз поиск простых чисел, воспользуемся файлом с простыми числами, полученным с помощью программы, описанной в предыдущем подразделе.

Алгоритм распределения данных к анализу будет схож с рассмотренным алгоритмом для предыдущей программы. Однако есть несколько отличий. Во-первых, каждый из процессов должен получить анализируемое число **proizv** и количество **kol** простых чисел в файле для выделения динамической памяти для них. После того как все процессы выделят память для массива простых чисел, главный процесс может передавать сам массив всем остальным процессам. Во-вторых, распределение вычислений будет проходить по массиву простых чисел. Если в программе, описанной в подразделе 5.1, мы не анализировали первые два значения и отсчет начинался с числа 2, то теперь нам необходимо провести анализ с использованием всех

ЛИТЕРАТУРА

1. Бабенко Л.К., Ищукова Е.А., Сидоров И.Д. Параллельные алгоритмы для решения задач защиты информации. – М.: Горячая линия – Телеком, 2014. – . с. 118 .. 125.