Лабораторная работа 4: Параллельные методы сортировки данных

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений в порядке монотонного возрастания или убывания.

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T \sim n^2. \tag{4.1}$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$$T \sim n \log_2 n. \tag{4.2}$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из n значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких ($p > 1$) процессоров. Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются между собой. Результирующий (упорядоченный) набор, как правило, также разделен между процессорами; при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет сортировку данных.

- Упражнение 1 – Постановка задачи сортировки.
- Упражнение 2 – Реализация последовательного алгоритма сортировки.
- Упражнение 3 – Разработка параллельного алгоритма сортировки.
- Упражнение 4 – Реализация параллельного алгоритма сортировки.

Примерное время выполнения лабораторной работы: **90 минут**.

При выполнении лабораторной работы предполагается знание раздела 4 "Параллельное программирование на основе MPI", раздела 6 "Принципы разработки параллельных методов" и раздела 10 "Параллельные методы сортировки данных" учебных материалов курса. Кроме того, предполагается, что выполнены как минимум ознакомительная лабораторная работа "Параллельное программирование с использованием MPI" и лабораторная работа 1 "Параллельные алгоритмы матрично-векторного умножения".

Упражнение 1 – Постановка задачи сортировки данных

При выполнении этого, начального, упражнения необходимо изучить последовательный алгоритм пузырьковой сортировки. В ходе выполнения упражнения необходимо изучить основные принципы, используемые при сортировке данных, а также детально разобрать предлагаемый последовательный алгоритм.

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, ..., a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a_1', a_2', ..., a_n') : a_1' \le a_2' \le ... \le a_n'\}$$

При внимательном рассмотрении способов упорядочивания данных, применяемых в алгоритмах сортировки, можно обратить внимание, что многие методы основаны на применении одной и той же *базовой операции* "*сравнить и переставить*" (*compare-exchange*), состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановки этих значений, если их порядок не соответствует условиям сортировки.

```
// Basic compare-exchange operation
if(A[i] > A[j]) {
  temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения собственно и проявляется различие алгоритмов сортировки.

Выберем для реализации в данной лабораторной работе один из наиболее простых методов упорядочивания данных - *алгоритм пузырьковой сортировки* (см. раздел 10 "Параллельные методы сортировки данных" учебных материалов курса, а также Кнут (1981), Кормен, Лейзерсон и Ривест (1999)). Этот алгоритм сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности

$(a_1, a_2, \ldots, a_n)$

алгоритм сначала выполняет n-1 базовых операций "сравнения-обмена" для последовательных пар элементов

$(a_1, a_2), (a_2, a_3), ..., (a_{n-1},a_n)$.

В результате после первой итерации алгоритма самый большой элемент перемещается ("всплывает") в конец последовательности. Далее последний элемент в преобразованной последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности

$(a'_1, a'_2, ..., a'_{n-1})$.

Как можно увидеть, последовательность будет отсортирована после n-1 итерации. Эффективность пузырьковой сортировки может быть улучшена, если завершать алгоритм в случае отсутствия каких-либо изменений сортируемой последовательности данных в ходе какой-либо итерации сортировки.

```
// Sequential bubble sorting algorithm
BubbleSort(double A[], int n) {
  for(i = 1; i < n; i++)
    for(j = 0; j < n - i; j++)
      compare_exchange(A[j], A[j+1]);
}
```

Упражнение 2 – Реализация последовательного алгоритма сортировки

При выполнении этого упражнения необходимо реализовать последовательный алгоритм пузырьковой сортировки. Начальный вариант будущей программы представлен в проекте *SerialBubbleSort*, который содержит некоторую часть исходного кода. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода количества исходных данных, инициализации данных для сортировки, непосредственно сортировки данных и проверки правильности результатов работы программы.

Задание 1 – Открытие проекта SerialBubbleSort

Откройте проект **SerialBubbleSort**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню **File** выполните команду **Open→Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\ SerialBubbleSort**,
- Дважды щелкните на файле **SerialBubbleSort.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialBubbleSort.cpp**, как это показано на рис. 4.1. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в рабочей области Visual Studio.

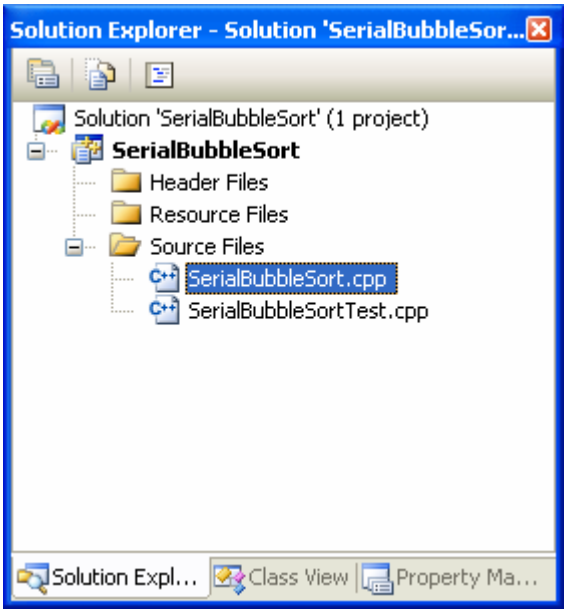

Рис. 4.1. Открытие файла SerialBubbleSort.cpp

В файле **SerialBubbleSort.cpp** подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции *main*. Подготовленный вариант программы

содержит объявление переменных и вывод на печать начального сообщения программы. В файле **SerialBubbleSortTest.cpp** содержатся подготовленные варианты тестовых функций, которые понадобятся для проверки правильности разрабатываемых функций.

Рассмотрим переменные, которые используются в основной функции (*main*) нашего приложения. Переменная *pData* есть указатель на данные для сортировки, после сортировки эта переменная будет указывать на упорядоченный набор данных. Переменная *DataSize* определяет количество данных для сортировки.

```
  double *pData;   // Data to be sorted
  int DataSize;    // Size of data to be sorted
```

Вывод начального сообщения обеспечивается при помощи следующих действий программы:

```
  printf("Serial bubble sort program\n");
  getch();
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет скомпилировать приложение. Если приложение скомпилировано успешно (в нижней части окна Visual Studio появилось сообщение "`Rebuild All: 1 succeeded, 0 failed, 0 skipped`"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

Сразу после запуска кода, в командной консоли появится сообщение: "`Serial bubble sort program`".

Задание 2 – Ввод количества сортируемых данных

Для задания исходных данных последовательного алгоритма сортировки реализуем функцию *ProcessInitialization*. Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества сортируемых данных, выделения памяти для сортируемых данных и для заполнения этой памяти начальными, неотсортированными, значениями. Таким образом, функция должна иметь следующий интерфейс:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize);
```

Добавим в программу код, позволяющий вводить количество сортируемых данных (задавать значение переменной *DataSize*) и проверять правильность произведенного ввода. Для этого добавьте на приведенный ниже код (выделенный полужирным шрифтом) в функцию *ProcessInitialization* и добавьте вызов этой функции в главную функцию (*main*) программы:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize) {
  do {
    printf("Enter the size of data to be sorted: ");
    scanf("%d", &DataSize);
    if(DataSize <= 0)
      printf("Data size should be greater than zero\n");
  }
  while(DataSize <= 0);

  printf("Sorting %d data items\n", DataSize);
}
```

Пользователю предоставляется возможность ввести размер сортируемых данных, который считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *DataSize*. Далее, значение переменной *DataSize* проверяется (оно должно быть больше нуля), в случае ошибочного ввода последний повторяется, и, наконец, введенное значение выводится на печать (рис. 4.2).

Скомпилируйте и запустите приложение. Убедитесь в том, что в случае ввода отрицательного значения переменной *DataSize*, программа выдает диагностическое сообщение: "`Data size should be greater than zero`".

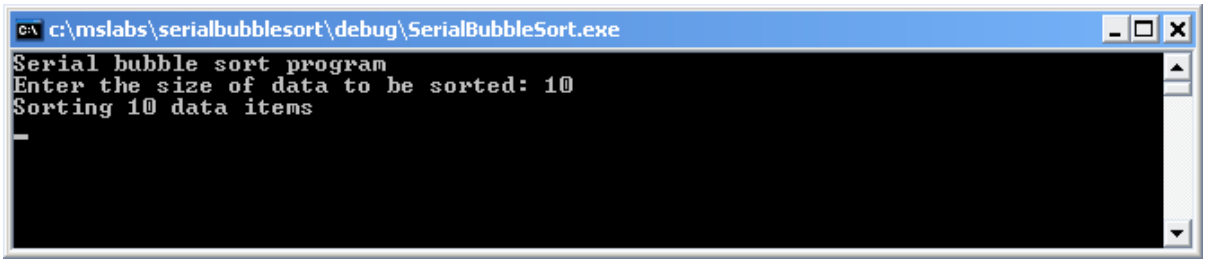

Рис. 4.2. Задание размера объектов

Далее, необходимо дополнить функцию *ProcessInitialization* выделением памяти под сортируемые данные и проинициализировать эти данные. Для этого добавьте выделенный код в функцию *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization (double *&pData, int& DataSize) {
  do {
    printf("Enter the size of data to be sorted: ");
    scanf("%d", &DataSize);
    if(DataSize <= 0)
      printf("Data size should be greater than zero\n");
  }
  while(DataSize <= 0);

  printf("Sorting %d data items\n", DataSize);

  pData = new double[DataSize];
  DummyDataInitialization(pData, DataSize);
}
```

Непосредственно инициализацию данных будет выполнять специальная функция. На первом этапе можно применить простой вариант реализации, который создает набор данных, правильность сортировки которого легко проверить. В тексте уже есть заготовка для такой функции (*DummyDataInitialization*), осталось дополнить её следующим кодом:

```
// Function for simple setting the initial data
void DummyDataInitialization(double*& pData, int& DataSize) {
  for(int i = 0; i < DataSize; i++)
    pData[i] = DataSize - i;
}
```

Как видно из представленного фрагмента кода, данная функция заполнят данные последовательными числами, начиная с количества данных и до единицы. Таким образом, в случае, когда пользователь ввел количество сортируемых данных, равное 10, данные будут определены следующим образом:

$$(10,9,8,7,6,5,4,3,2,1)$$

(задание данных при помощи датчика случайных чисел будет рассмотрено в задании 5).

Напомним, что в лабораторной работе рассматривается сортировка в порядке возрастания, таким образом, данная функция заполняет массив значениями, отсортированными в обратном порядке.

Скомпилируйте приложение (выполните команду **Rebuild Solution** пункта меню **Build**). Если обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в данном упражнении. После того, как все ошибки исправлены, запустите приложение.

Реализуем еще одну функцию, которая поможет в дальнейшем контролировать работу алгоритма. Это функция форматированного вывода данных *PrintData*. Заготовку этой функций можно найти в файле **SerialBubbleSortTest.cpp**. Перейти к редактированию этого файла можно аналогично выбору для редактирования файла **SerialBubbleSort.cpp** в задании 1. В качестве аргументов в функцию форматированной печати данных *PrintData* передается указатель на одномерный массив *pData*, где эти данных хранятся, а также количество элементов в этом массиве (*DataSize*):

```
// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
```

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

Добавим вызов функции *PrintData* в функцию *main* приложения, сразу же после вызова функции *ProcessInitialization* для проверки правильности задания исходных данных:

```
...
  // Process initialization
  ProcessInitialization(pData, DataSize);

  printf("Data before sorting\n");
  PrintData(pData, DataSize);
...
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным ранее правилам (рис. 4.3). Выполните несколько запусков приложения, задавайте различные размеры объектов.

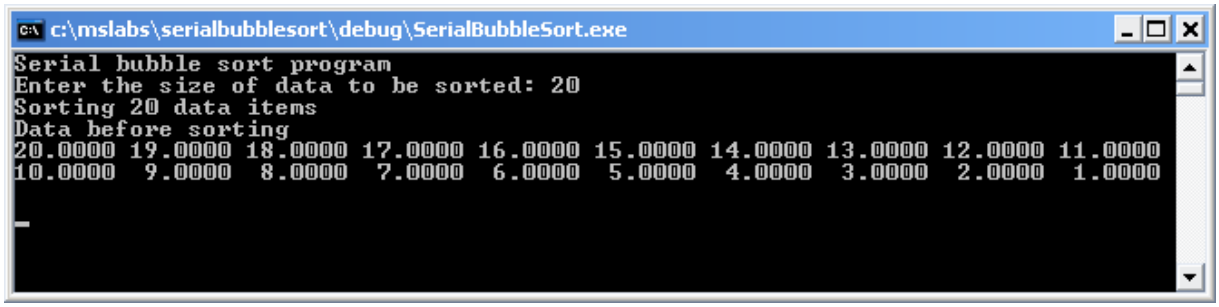

Рис. 4.3. Результат работы программы при завершении задания 2

Задание 3 – Завершение процесса вычислений

В данном задании перед выполнением сортировки данных сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*. Память выделялась для хранения сортируемых данных *pData*. Следовательно, указатель на эту память необходимо передать в функцию *ProcessTermination* в качестве аргумента:

```
// Function for computational process termination
void ProcessTermination(double *pData) {
  delete []pData;
}
```

Вызов функции *ProcessTermination* необходимо добавить в программу непосредственно перед завершением основной функции *main*, когда сортируемые данные уже больше не нужны:

```
...
  // Process termination
  ProcessTermination(pData);

  return 0;
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что оно выполняется корректно.

Задание 4 – Реализация алгоритма пузырьковой сортировки

Выполним теперь разработку основной вычислительной части программы. Для выполнения пузырьковой сортировки реализуем функцию *SerialBubble*, которая принимает на вход исходные данные *pData* и размер этих данных *DataSize*.

В соответствии с алгоритмом, изложенным в упражнении 1, код этой функции должен быть следующий:

```
// Serial bubble sort algorithm
void SerialBubble(double *pData, int DataSize) {
  double Tmp;

  for(int i = 1; i < DataSize; i++)
```

```
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}
```

Выполним вызов функции сортировки из основной программы. Для контроля правильности выполнения сортировки распечатаем получившиеся данные:

```
...
  // Process initialization
  ProcessInitialization(pData, DataSize);

  printf("Data before sorting\n");
  PrintData(pData, DataSize);

  // Serial bubble sort
  SerialBubble(pData, DataSize);

  printf("Data after sorting\n");
  PrintData(pData, DataSize);
...
```

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма сортировки. Если алгоритм реализован правильно, то результат должен представлять все числа от единицы и до количества введенных данных по порядку. Проведите несколько вычислительных экспериментов, изменяя размер сортируемого набора данных.

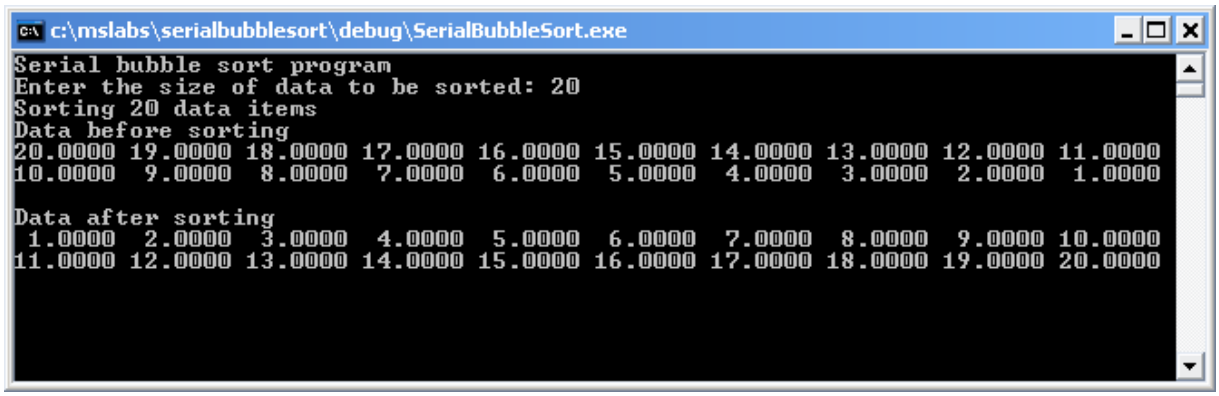

Рис. 4.4. Результат сортировки массива из 20 элементов

Задание 5 – Проведение вычислительных экспериментов

Для последующего тестирования ускорения работы параллельного алгоритма необходимо провести эксперименты по вычислению времени выполнения последовательного алгоритма. Анализ времени выполнения алгоритма целесообразно проводить для достаточно больших размеров данных. Задавать эти данные будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов *RandomDataInitialization* (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for initializing the data by the random generator
void RandomDataInitialization(double *&pData, int& DataSize) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}
```

Будем вызывать эту функцию вместо ранее разработанной функции *DummyDataInitialization* в функции *ProcessInitialization*, которая генерировала предсказуемые данные, при использовании которых было легко проверить правильность работы алгоритма:

```
// Function for allocating the memory and setting the initial values
```

```
void ProcessInitialization(double *&pData, int& DataSize) {
...
  pData = new double[DataSize];

  // Setting the data by the random generator
  RandomDataInitialization(pData, DataSize);
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Для определения времени добавьте в получившуюся программу вызовы функций, позволяющие узнать время работы программы или её части. Как и в других лабораторных работах, воспользуемся для этого функцией:

```
time_t clock(void);
```

Для вычисления времени работы нам понадобятся три дополнительные переменные, объявления которых нужно добавить в функцию *main*:

```
int main(int argc, char *argv[]) {
  double *pData = 0;
  int DataSize = 0;

  time_t start, finish;
  double duration = 0.0;
...
```

Добавим в программный код вычисление и вывод времени непосредственного выполнения сортировки, для этого поставим замеры времени до и после вызова функции *SerialBubble*:

```
...
  printf("Data before sorting\n");
  PrintData(pData, DataSize);

  // Serial bubble sort
  start = clock();
  SerialBubble(pData, DataSize);
  finish = clock();

  printf("Data after sorting\n");
  PrintData(pData, DataSize);

  duration = (finish - start) / double(CLOCKS_PER_SEC);
  printf("Time of execution: %f\n", duration);
...
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими объектами, отключите печать данных до и после сортировки (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты и заполните третий столбец таблицы:

Таблица 4.1. Результаты вычислительных экспериментов для метода пузырьковой сортировки

Номер теста	Количество сортируемых данных	Время работы пузырьковой сортировки (сек.)	Время работы сортировки из стандартной библиотеки (сек.)
1	10		
2	100		
3	10 000		
4	20 000		
5	30 000		
6	40 000		
7	50 000		

По результатам выполненных экспериментов оцените характер зависимости времени сортировки от количества упорядочиваемых данных – убедитесь, в том, что данная зависимость является квадратичной (при увеличении количества данных в два раза время сортировки увеличивается в четыре раза и т.д.).

Как отмечалось ранее, для сортировки существуют более эффективные методы (по сравнению с пузырьковым алгоритмом). Один из таких быстрых алгоритмов реализован в стандартной библиотеки алгоритмического языка C++. Выполним эксперименты для оценки эффективности этого стандартного алгоритма – для этого необходимо заменить вызов функции пузырьковой сортировки на вызов сортировки *sort* из стандартной библиотеки (для использования функции в программу надо добавить заголовочный файл <*algorithm*>). Воспользуемся функцией *SerialStdSort*, которая располагается в файле **SerialBubbleSortTest.cpp**:

```
// Sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}
```

Далее закомментируйте вызов *SerialBubble* и добавьте вызов рассматриваемой функции в функцию *main*:

```
  start = clock();
  // Serial buble sort
  // SerialBubble(pData, DataSize);
  // Sorting by the standard library algorithm
  SerialStdSort(pData, DataSize);
  finish = clock();
```

Проведите вычислительные эксперименты, аналогичные предыдущим, и заполните последний столбец таблицы.

Оценим аналитически трудоемкость алгоритма пузырьковой сортировки (см. раздел 10 "Параллельные методы сортировки данных" учебных материалов курса). Получение отсортированных данных предполагает повторение *DataSize* – 1 однотипных операций прохода по массиву. Каждая такая операция включает *DataSize* – *i* базовых операций "*сравнить и переставить*", где *i* – номер прохода по массиву. Таким образом, время выполнения алгоритма можно оценить как:

$$T_1 = (DataSize \cdot (DataSize - 1)/2) \cdot \tau, \qquad (4.3)$$

где τ есть время выполнения одной операции "*сравнить и переставить*".

Заполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (4.3). Для вычисления времени выполнения одной операции применим следующую методику: выберем один из экспериментов как образец (например, эксперимент по сортировке 30 000 элементов данных) и поделим время выполнения этого эксперимента на число выполненных операций. Таким образом, вычислим время выполнения одной операции τ. Далее, используя это значение, вычислим теоретическое время выполнения для всех оставшихся экспериментов.

Проведите указанные вычисления, результаты занесите в таблицу:

Таблица 4.2. Сравнение экспериментального и теоретического времени метода пузырьковой сортировки

Время выполнения одной операции τ (сек):			
Номер теста	Количество сортируемых данных	Время работы (сек)	Теоретическое время (сек)
1	10		
2	100		
3	10 000		
4	20 000		
5	30 000		
6	40 000		
7	50 000		

Заметим, что время выполнения одной операции "*сравнить и переставить*", вообще говоря, зависит от количества сортируемых данных. Такая зависимость объясняется особенностями архитектуры компьютера. Если данных немного, то они полностью могут быть помещены в кэш-память процессора, доступ к этой памяти осуществляется очень быстро. Если алгоритм со средним количеством данных, такими, которые полностью могут быть помещены в оперативную память, но не могут быть помещены в

кэш, то время выполнения одной операции в этом случае будет несколько больше, так как для обращения к элементу оперативной памяти требуется больше времени, чем для обращения к кэш. Если же данных настолько много, что они не могут быть помещены в оперативную память, то включается механизм работы с файлами подкачки (swap file), данные сохраняются на жестком диске компьютера, время чтения и записи на жесткий диск существенно превышает время записи в ячейку оперативной памяти. Таким образом, при выборе эксперимента для образца (такого эксперимента, для которого будет вычисляться время выполнения одной операции), следует ориентироваться на некоторую среднюю ситуацию. Именно поэтому нами в качестве образца был выбран эксперимент по сортировке 30 000 элементов данных.

Упражнение 3 – Разработка параллельного алгоритма сортировки

При выполнении этого упражнения необходимо изучить принципы распараллеливания алгоритма пузырьковой сортировки. Для этого необходимо провести декомпозицию задачи, выделить информационные взаимодействия между подзадачами, определить вычислительную схему и выполнить распределение набора подзадач по вычислительным устройствам.

Принципы распараллеливания

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как метод *чет-нечетной перестановки* (*odd-even transposition*) – см. раздел 10 "Параллельные методы сортировки данных" учебных материалов курса, а также Kumar et al. (1994). Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары

(a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n) (при четном n),

а на четных итерациях обрабатываются элементы

(a_2, a_3), (a_4, a_5), ..., (a_{n-2},a_{n-1}).

После n- кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

```
// Sequential odd-even transposition algorithm
OddEvenSort ( double A[], int n ) {
  for ( i=1; i<n; i++ ) {
    if ( i%2==1 ) { // odd iteration
      for ( j=0; j<n/2-2; j++ )
        compare_exchange(A[2j+1],A[2j+2]);
      if ( n%2==1 ) // the comparison of the last pair, if n is odd
        compare_exchange(A[n-2],A[n-1]);
     }
     if ( i%2==0 )    // even iteration
       for ( j=1; j<n/2-1; j++ )
         compare_exchange(A[2j],A[2j+1]);
  }
}
```

Более подробное изложение алгоритма чет-нечетной сортировки содержится в разделе 10 "Параллельные методы сортировки данных" учебных материалов курса.

Определение подзадач и выделение информационных зависимостей

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. В случае $p < n$, когда количество процессоров является меньшим числа упорядочиваемых значений, процессоры содержат блоки данных размера n/p. Алгоритм сортировки в этом случае может быть получен как обобщение процедуры чет-нечетной сортировки (см. также подраздел 10.2 раздела 10 "Параллельные методы сортировки данных" учебных материалов курса).

Следуя схеме одноэлементного сравнения, взаимодействие пары процессоров P_i и P_{i+1} для совместного упорядочения содержимого блоков A_i и A_{i+1} может быть осуществлено следующим образом:

- выполнить взаимообмен блоков между процессорами P_i и P_{i+1},
- объединить блоки A_i и A_{i+1} на каждом процессоре в один отсортированный блок двойного размера (при исходной упорядоченности блоков A_i и A_{i+1} процедура их объединения сводится к быстрой операции слияния упорядоченных наборов данных),
- разделить полученный двойной блок на две равные части и оставить одну из этих частей (например, с меньшими значениями данных) на процессоре P_i, а другую часть (с большими значениями соответственно) – на процессоре P_{i+1}

$$[A_i \cup A_{i+1}]_{copm} = A_i^{'} \cup A_{i+1}^{'} : \forall a_i^{'} \in A_i^{'}, \forall a_j^{'} \in A_{i+1}^{'} \Rightarrow a_i^{'} \le a_j^{'} .$$

Определенная выше операция "сравнить и разделить" может быть использована в качестве *базовой подзадачи* для организации параллельных вычислений, Получаемый в результате параллельный алгоритм может быть представлен следующим образом:

```
// Parallel odd-even transposition algorithm
ParallelOddEvenSort(double A[], int n) {
  int id = GetProcId();  // Process number
  int np = GetProcNum(); // Number of processes
  for ( int i=0; i<np; i++ ) {
    if ( i%2 == 1 ) { // Odd iteration
      if ( id%2 == 1 ) { // Odd process number
        // Compare-exchange with the right neighbor process
        if ( id < np -1 ) compare_split_min(id+1);
      }
      else
        // Compare-exchange with the left neighbor process
        if ( id > 0 ) compare_ split_max(id-1);
    }
    if ( i%2 == 0 ) { // Even iteration
      if( id%2 == 0 ) { // Even process number
        // Compare-exchange with the right neighbor process
        if ( id < np -1 ) compare_ split_min(id+1);
      }
      else
        // Compare-exchange with the left neighbor process
        compare_ split_max(id-1);
    }
  }
}
```

Для пояснения такого параллельного способа сортировки в табл. 4.1 приведен пример упорядочения данных при $n=16$, $p=4$ (т.е. блок значений на каждом процессоре содержит $n/p=4$ элемента). В первом столбце таблицы приводится номер и тип итерации метода, перечисляются пары процессов, для которых параллельно выполняются операции "сравнить и разделить". Взаимодействующие пары процессов выделены в таблице двойной рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

Таблица 4.3. Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итерации	Процессоры			
	1	2	3	4
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
1 нечет (1,2),(3.4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
2 чет (2,3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
3 нечет (1,2),(3.4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
4 чет (2,3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

Масштабирование и распределение подзадач по процессорам

Как отмечалось ранее, количество подзадач соответствует числу имеющихся процессоров и, как результат, необходимости в проведении масштабирования вычислений не возникает. Исходное распределение блоков упорядочиваемого набора данных по процессорам может быть выбрано произвольным образом. Для эффективного выполнения рассмотренного параллельного алгоритма сортировки необходимым является, чтобы процессоры с соседними номерами имели прямые линии связи.

Упражнение 4 – Реализация параллельного алгоритма сортировки

При выполнении этого упражнения необходимо разработать параллельный алгоритм сортировки. Это упражнение направлено на:

- Расширение практических знаний по разработке параллельных программ,
- Индивидуальную разработку параллельной программы для сортировки данных.

Как и ранее, разрабатываемая параллельная программа будет состоять из следующих основных структурных частей:

- Инициализация среды выполнения MPI-программ,
- Основная часть программы, в которой реализуется необходимый алгоритм решения поставленной задачи и в которой осуществляется обмен сообщениями между параллельно выполняемыми частями программы,
- Завершение MPI программы.

При выполнении упражнения предполагается знание раздела 4 "Параллельное программирование на основе MPI".

Задание 1 – Открытие проекта ParallelBubbleSort

Откройте проект **ParallelBubbleSort**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню **File** выполните команду **Open→Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\ ParallelBubbleSort**,
- Дважды щелкните на файле **ParallelBubbleSort.sln** или подсветите его выполните команду **Open**.

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **ParallelBubbleSort.cpp**, как это показано на рис. 4.5. После этих действий код, который предстоит модифицировать, будет открыт в рабочей области Visual Studio.

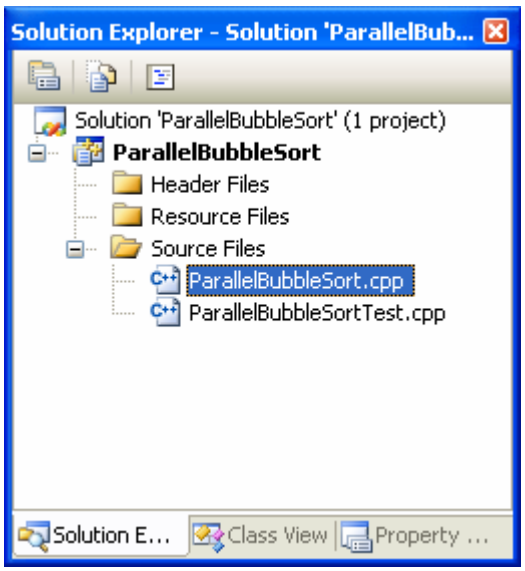

Рис. 4.5. Открытие файла ParallelBubbleSort.cpp

В файле **ParallelBubbleSort.cpp** расположена главная функция (*main*) будущего параллельного приложения, которая содержит объявления необходимых переменных. Также в файле **ParallelBubbleSort.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм сортировки: *DummyDataInitialization*, *RandomDataInitialization*. Кроме того, в файле **ParallelBubbleSortTest.cpp** можно видеть функцию *PrintData*, отвечающую за тестовый вывод данных (подробно о назначении этой функции рассказывается в упражнении 2 данной лабораторной работы). Эти функции можно будет использовать и в параллельной программе. Кроме того, в исходный код помещены заготовки для функций инициализации процесса вычислений (*ProcessInitialization*) и завершения процесса (*ProcessTermination*).

Скомпилируйте и запустите приложение стандартными средствами Visual Studio. Убедитесь в том, что в командную консоль выводится приветствие: "`Parallel bubble sort program`".

Задание 2 – Инициализация и завершение параллельной программы

Как уже отмечалось ранее, в параллельной программе должен присутствовать заголовочный файл "mpi.h".

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include <cmath>
#include <algorithm>
#include <mpi.h>
```

Далее, в главной функции программы необходимо проинициализировать среду выполнения MPI-программы, определить число процессов, доступных для MPI-программы, определить ранг процесса в рамках коммуникатора MPI_COMM_WORLD, а также завести глобальные переменные для хранения этих значений (*ProcNum* и *ProcRank* соответственно). Добавьте выделенный код:

```
int ProcNum = 0;      // Number of available processes
int ProcRank = -1;     // Rank of current process

void main(int argc, char* argv[]) {
  double *pData = 0;
  double *pProcData = 0;
  int DataSize = 0;
  int BlockSize = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  printf("Parallel bubble sort program\n");
```

```
  MPI_Finalize();
}
```

Скомпилируйте параллельное приложение средствами **Visual Studio** (выполните команду **Rebuild Solution** пункта меню **Build**). Для того чтобы запустить параллельную программу, запустите программу "Command prompt", выполняя следующие действия:

1. Нажмите кнопку **Пуск**, а затем **Выполнить**,
2. В появившемся диалоговом окне наберите название программы "**cmd**" (рис. 4.6).

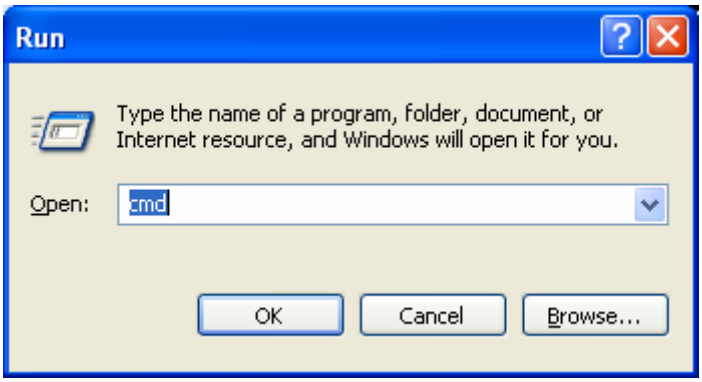

Рис. 4.6. Запуск Command Prompt

В командной строке перейдите в папку, где содержится исполняемый модуль разработанной программы (рис. 4.7):

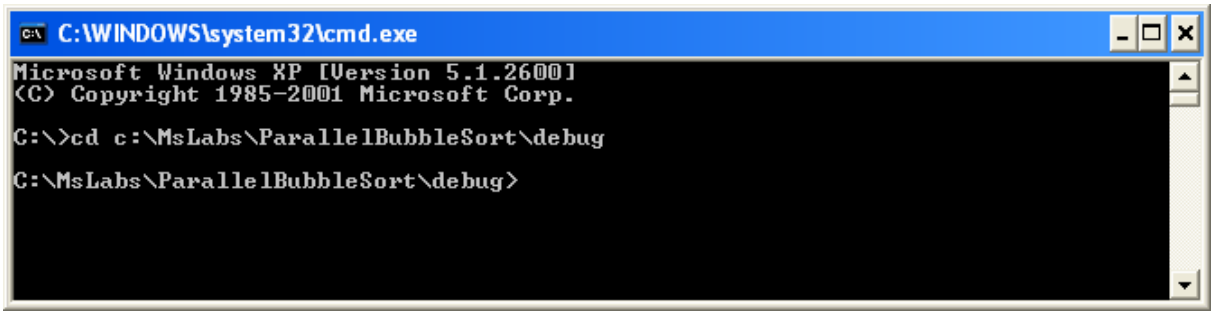

Рис. 4.7. Задание папки, в которой содержится исполняемый модуль параллельной программы

Аналогично запуску программ из предыдущих лабораторных работ, для запуска программы из 4 процессов наберите команду (рис. 4.8):

```
mpiexec –n 4 ParallelBubbleSort.exe
```

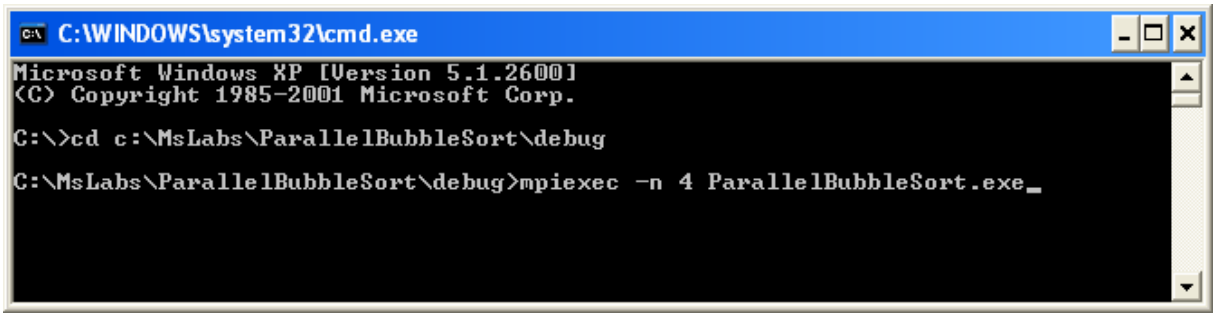

Рис. 4.8. Запуск параллельной программы

Убедитесь в том, что в командную консоль выводится приветствие: "`Parallel bubble sort program`".

Задание 3 – Ввод исходных данных

Перейдем к организации ввода данных. Необходимо дополнить нашу программу кодом, обеспечивающим задание количества сортируемых данных и выделяющим память под сортируемые элементы. Как и в других работах, определение начальных данных будет осуществляться одним из процессов (пусть этим процессом будет процесс с рангом 0). Далее, согласно схеме параллельных

вычислений, изложенной в упражнении 3, сортируемые данные распределяется между всеми процессами таким образом, что каждый процесс обрабатывает непрерывную последовательность (блок) данных. Отметим, что первая версия разрабатываемой программы ориентирована на случай, когда размер объектов делится нацело на число процессов, то есть блоки данных на всех процессах содержат одно и то же число элементов. Это количество будем хранить в переменной *BlockSize*. Адрес буфера памяти, где содержится блок данных на каждом из процессов, будем хранить в переменной *pProcData*. В результате сортировки, каждый процесс получает *BlockSize* отсортированных элементов результирующих данных. Затем эти данные необходимо будет вновь собрать на *ведущем* процессе (процессе с рангом 0).

Для инициализации вычислительных процессов, как и ранее, служит функция *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize,
  double *&pProcData, int& BlockSize);
```

Итак, сначала нужно определить количество сортируемых данных, то есть задать значение переменной *DataSize*. Для определения размеров объектов необходимо реализовать диалог с пользователем. Как и в предыдущих лабораторных работах, реализуем проверку правильности вводимого значения. Добавьте выделенный фрагмент кода в тело функции *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize,
  double *&pProcData, int& BlockSize) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    do {
      printf("Enter the size of data to be sorted: ");
      scanf("%d", &DataSize);
      if(DataSize < ProcNum) {
        printf("Data size should be greater than number of processes\n");
      }

      if(DataSize % ProcNum != 0) {
        printf("Data size should be divisible by number of processes\n");
      }
    } while((DataSize < ProcNum) || (DataSize % ProcNum != 0));

    printf("Sorting %d data items\n", DataSize);
  }
}
```

Теперь необходимо разослать количество сортируемых данных остальным процессам. Для этого используем знакомую по предыдущим лабораторным работам функцию широковещательной рассылки *MPI_Bcast*. Добавьте следующий код в вашу программу (обратите внимание, что вызов *MPI_Bcast* должен выполняться всеми процессами):

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize,
  double *&pProcData, int& BlockSize) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    ...
    printf("Sorting %d data items\n", DataSize);
  }
  // Broadcasting the data size
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
  BlockSize = DataSize / ProcNum;
}
```

Добавьте вызов функции инициализации вычислительных процессов:

```
int main(int argc, char *argv[]) {
  double *pData = 0;
  double *pProcData = 0;
  int DataSize = 0;
  int BlockSize = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  if(ProcRank == 0)
    printf("Parallel bubble sort program\n");

  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  MPI_Finalize();

  return 0;
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что все ошибочные ситуации обрабатываются корректно. Для этого выполните несколько запусков приложения, задавая различное количество параллельных процессов (при помощи параметра запуска утилиты **mpiexec**) и разные размеры сортируемого набора данных.

После того, как количество сортируемых данных определено, можно перейти к выделению памяти под эти данные и блоки, принадлежащие процессам. Добавьте выделенный код в тело функции *ProcessInitialization*:

```
  // Broadcasting the data size
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
  BlockSize = DataSize / ProcNum;

  pProcData = new double[BlockSize];

  if(ProcRank == 0) {
    pData = new double[DataSize];

    // Data initalization
    DummyDataInitialization(pData, DataSize);
  }
}
```

Отметим, что для задания сортируемых элементов на ведущем процессе была использована функция генерации данных *DummyDataInitialization*, которая была разработана при реализации последовательного приложения сортировки. Напомним, что эта функция заполняет массив значениями, отсортированными по убыванию.

Задание 4 – Завершение процесса вычислений

Для корректного завершения параллельной программы разработаем функцию остановки процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*.

На ведущем процессе выделялась память для всех сортируемых данных *pData*, кроме того, на всех процессах выделялась память для хранения блока данных *pProcData*. Оба этих указателя необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination(double *pData, double *pProcData) {
  if(ProcRank == 0) delete []pData;

  delete []pProcData;
}
```

Вызов функции остановки процесса вычислений необходимо добавить в функцию *main* непосредственно перед вызовом функции *MPI_Finalize*:

```
...
  // Process termination
  ProcessTermination(pData, pProcData);

  MPI_Finalize();
}
```

Добавьте в код основной функции команды для печати сортируемых данных на ведущем процессе (используйте функцию *PrintData*, реализованную в ходе разработки последовательного приложения). Скомпилируйте и запустите приложение. Убедитесь в том, что исходные данные задаются верно.

Задание 5 – Распределение данных между процессами

В соответствии со схемой параллельных вычислений, изложенной в предыдущем упражнении, сортируемые данные должны быть распределены между процессами равными блоками.

За распределение данных отвечает функция *DataDistribution*. На вход этой функции в качестве аргументов необходимо передать указатель на сортируемые данные *pData*, количество этих данных и адрес буфера для хранения части данных, принадлежащих процессу *pProcData*, а также размер блока *BlockSize*:

```
// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData,
  int BlockSize);
```

Далее необходимо, используя обобщенную операцию передачи данных от одного процесса всем процессам (распределение данных), распределить сортируемые данные между процессами:

```
// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData,
  int BlockSize) {

  MPI_Scatter(pData, BlockSize, MPI_DOUBLE, pProcData, BlockSize,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Соответственно, вызывать эту функцию из основной программы следует после вызова функции инициализации вычислительного процесса *ProcessInitialization*:

```
  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  // Distributing the initial data among the processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);
```

Теперь выполним проверку правильности разделения данных между процессами. Для этого после выполнения функции *DataDistribution* распечатаем сортируемые данные, а затем блоки данных, содержащиеся на каждом из процессов. Добавим в код приложения еще одну функцию, которая служит для проверки правильности выполнения этапа распределения данных, и назовем ее *TestDistribution*.

Для того чтобы организовать форматированный вывод данных, воспользуемся разработанной ранее функцией *PrintData*:

```
// Function for testing the data distribution
void TestDistribution(double *pData, int DataSize, double *pProcData,
  int BlockSize) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial data:\n");
    PrintData(pData, DataSize);
  }

  MPI_Barrier(MPI_COMM_WORLD);

  for (int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Block:\n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

Функция *TestDistribution* похожа на разработанные в других лабораторных работах функции аналогичного предназначения: ведущий процесс распечатывает все данные, затем процессы

параллельной программы печатают свои блоки данных по порядку (сначала свои данные печатает процесс с рангом 0, далее процесс с рангом 1 и т.д.).

Добавьте вызов функции проверки распределения непосредственно после функции *DataDistribution*:

```
...
  // Distributing the initial data between processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);
  // Testing the distribution
  TestDistribution(pData, DataSize, pProcData, BlockSize);
...
```

Напомним, функция генерации исходных данных *DummyDataInitialization* устроена таким образом, что она формирует исходные данные, упорядоченные по убыванию. Значит, после разделения данных на процессе с рангом *i* должны оказаться данные в интервале от *DataSize–i*BlockSize* до *DataSize–*(*i*+1)***BlockSize*+1.

Скомпилируйте приложение. Если в приложении обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в данной работе. Запустите приложение из четырех процессов и установите размер данных 24. Убедитесь в том, что распределение данных выполняется правильно (рис. 4.9).

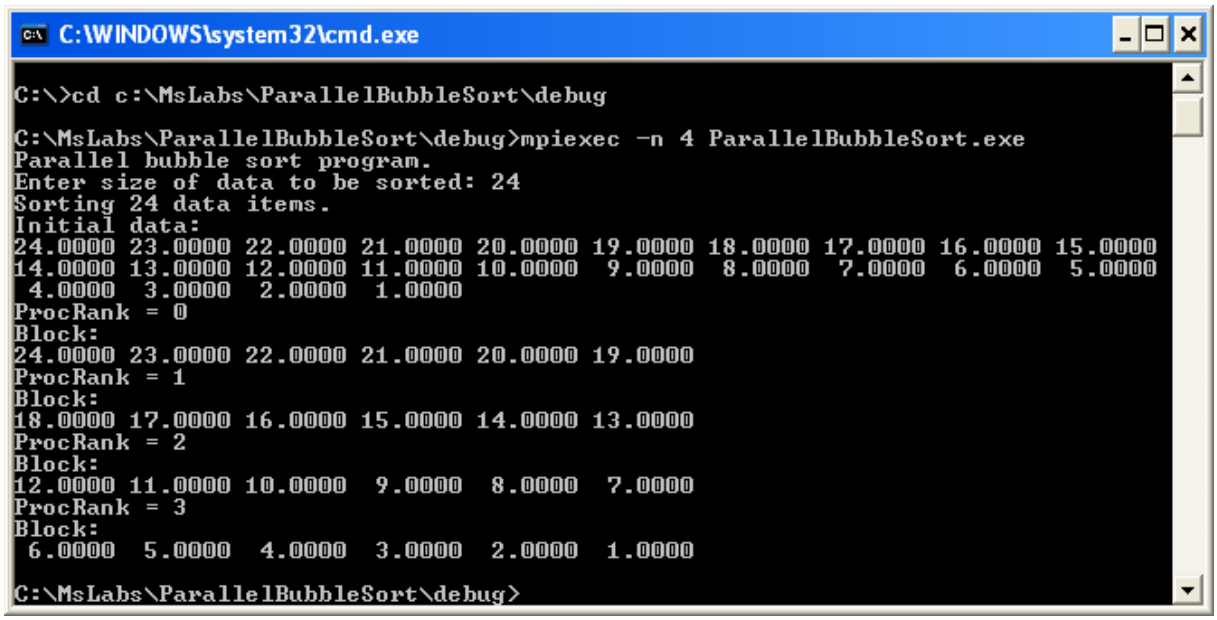

Рис. 4.9. Распределение данных для программы из четырех процессов.

Задание 6 – Локальная сортировка данных

Выполним реализацию алгоритма параллельной чет-нечетной сортировки в ходе нескольких последовательных этапов, каждый из которых будет достаточно простым и правильность каждого из которых можно будет легко проверить.

Определим заголовок функции параллельной сортировки данных. Для сортировки блока данных, принадлежащих процессу необходимо иметь указатель на участок памяти *pProcData*, где эти данные расположены и размер этого участка *BlockSize*. Как результат, разрабатываемая функция сортировки будет иметь следующий заголовок:

```
void ParallelBubble(double *pProcData, int BlockSize);
```

В соответствии с общей схемой параллельной чет-нечетной сортировки, прежде всего необходимо отсортировать локальную копию данных, для чего можно воспользоваться разработанной в упражнении 2 функцией *SerialBubbleSort* (которую можно найти в файле **ParallelBubbleSortTest.cpp**). Для выполнения локальной сортировки выполните первую реализацию функции *ParallelBubble*:

```
// Parallel bubble sort algorithm
void ParallelBubble(double *pProcData, int BlockSize) {

  // Local sorting the process data
  SerialBubbleSort(pProcData, BlockSize);
```

```
  // Print the sorted data
  ParallelPrintData(pProcData, BlockSize);
}
```

Как можно заметить, в текст функции *ParallelBubble* добавлено обращение к функции тестовой печати блоков всех процессов для проверки правильности выполнения локальной сортировки. Возможный вариант функции отладочной печати *ParallelPrintData* состоит в следующем:

```
// Function for parallel data output
void ParallelPrintData(double *pProcData, int BlockSize) {
  // Print the sorted data
  for(int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Proc sorted data:\n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

Добавьте вызов функций *ParallelBubble* функцию *main*. Кроме того, для уменьшения объема отладочного вывода, закомментируйте вызов функции *TestDistribution*. Откомпилируйте и запустите приложение. Убедитесь, что локально данные сортируются верно.

Задание 7 – Обмен отсортированными данными

Следующим этапом в разработке параллельного алгоритма пузырьковой сортировки будет реализация обмена процессами копиями своих отсортированных локальных данных. Разработаем для этого функцию *ExchangeData*, заголовок которой приведен ниже:

```
// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData);
```

Эта функция обменивается данными, на которые указывает параметр *pProcData*, размером *BlockSize*, с процессом с рангом *DualRank*. Принимаемые данные записываются в массив, адрес которого передается в функцию через параметр *pDualData*. Обмен блоками данных между процессами будем производить с помощью уже знакомой по прошлым лабораторным работам функцией *MPI_Sendrecv*. Добавьте функцию *ExchnageData* в исходный код разрабатываемого параллельного приложения:

```
// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData) {

  MPI_Status status;
  MPI_Sendrecv(pProcData, BlockSize, MPI_DOUBLE, DualRank, 0,
    pDualData, BlockSize, MPI_DOUBLE, DualRank, 0,
    MPI_COMM_WORLD, &status);
}
```

Теперь необходимо вызвать разработанную функцию из функции *ParallelBubble*. Для того чтобы этот вызов был осуществим, необходимо выделить память для принимаемых данных. В соответствии с общей схемой параллельной чет-нечетной сортировки обмен будет производиться с соседним по номеру процессом справа или слева от текущего, в зависимости от номера итерации и ранга процесса. Для индикации соседнего процесса, с которым производится обмен, введем переменную *Offset*, которая будет принимать значения либо +1 (при необходимости обмена со следующим процессом), либо –1 в противном случае.

Добавьте следующий код в функцию *ParallelBubble* после вызова *SerialBubbleSort*:

```
...
  double *pDualData = new double[BlockSize];

  int Offset;

  if(ProcRank != 0) {
    Offset = -1;
    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData);
```

```
  }

  delete []pDualData;
...
```

Как можно видеть из приведенного фрагмента, в данном упрощенном варианте реализации обмен проводится с процессом, находящимся слева от текущего (если таковой существует). Следует отметить, что функция *MPI_Sendrecv* в случае указания несуществующего ранга процесса не выполняет каких-либо действий и просто завершает свою работу.

Для проверки правильности выполнения процедуры обмена переместите вызов функции *ParallelPrintData* на следующую после вызова функции *ExchangeData* строку, указав для печати полученные процессом данные. Скомпилируйте и запустите приложение. Убедитесь, что процессы обмениваются локальными данными верно. Протестируйте обмен с правыми процессами.

Задание 8 – Слияние и разделение данных

Далее, процессы в соответствии с алгоритмом чет-нечетной сортировки должны объединить свои имеющиеся данные (исходный блок *pProcData* и новый полученный от соседнего процесса набор данных *pDualData*). Операция объединения может быть сведена к процедуре слияния, поскольку оба набора объединяемых данных являются упорядоченными. Для слияния данных можно воспользоваться функцией *merge* стандартной библиотеки алгоритмического языка C++. При этом, для хранения результирующего, объединенного массива (назовем указатель на него *pMergedData*) следует выделить память, и вернуть эту память в систему, когда она станет больше не нужна. Добавьте следующие строки в функцию *ParallelBubble*:

```
...
  double *pDualData   = new double[BlockSize];
  double *pMergedData = new double[2 * BlockSize];
...
    // Data merging
    merge(pProcData, pProcData + BlockSize, pDualData,
      pDualData + BlockSize, pMergedData);
...
  delete []pDualData;
  delete []pMergedData;
...
```

В соответствии с правилами использования функции *merge*, ей передаются указатели на начала объединяемых массивов, на первые элементы за последними в них, а также указатель на массив, в который будет занесен результат.

Для проверки правильности процедуры слияния переместите вызов функции *ParallelPrintData* на следующую после вызова функции *merge* строку, указав для печати объединенный набор данных. Скомпилируйте и запустите приложение. Убедитесь, что слияние данных выполняется верно.

Далее, каждый процесс должен оставить у себя только часть объединенного набора данных. Объявим для этого переменную *SplitMode*, показывающую, какую часть объединенного массива процесс должен оставить. Определим перечислимый тип для задания возможных значений переменной *SplitMode*:

```
enum split_mode { KeepFirstHalf, KeepSecondHalf };
```

При *SplitMode* равном *KeepFirstHalf*, процесс должен оставить у себя первую половину данных, а при значении *KeepSecondHalf* – вторую. Копирование оставляемых данных должно осуществляться в память по указателю *pProcData*.

Внесите следующие изменения в текст функции *ParallelBubble*:

```
...
  int Offset;
  split_mode SplitMode = KeepFirstHalf;

  if(ProcRank != 0) {
    Offset = -1;
    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData);

    // Data merging
    merge(pProcData, pProcData + BlockSize, pDualData,
```

```
      pDualData + BlockSize, pMergedData);

    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData + BlockSize, pMergedData + 2*BlockSize, pProcData);
  }
...
```

Как можно видеть из приведенного фрагмента, в тестовом режиме показатель того, какую часть результирующего массива процесс должен оставить себе, определяется непосредственно в исходном коде. Правильный же способ задания значений переменной *SplitMode* будет добавлен далее при подготовке окончательного варианта функции *ParallelBubble*.

Переместите вызов функции *ParallelPrintData* в конец функции *ParallelBubble*, скомпилируйте и запустите приложение. Убедитесь, что процессы производят слияние и разделение данных правильно. Протестируйте случай, когда процессы оставляют себе вторую половину объединенного массива.

Задание 9 – Выполнение итераций параллельной чет-нечетной сортировки

В соответствии с общей схемой чет-нечетной сортировки, реализованные в заданиях 7–8 действия по обмену блоков, слиянию и разделению данных должны быть повторены p раз. Напомним, что на каждой такой итерации обмен должен проводиться с соседним по номеру процессом справа или слева от текущего (для индикации нужного соседнего процесса используется переменная *Offset*). На нечетных итерациях процессы с нечетными номерами должны обмениваться с соседями справа, а процессы с четными номерами – с соседями слева, на четных итерациях – наоборот (процессы с нечетными номерами – с соседями слева, процессы с четными номерами – с соседями справа). Аналогично определяется и часть объединенных данных, оставляемых на процессах.

Рассмотренные выше правила могут быть реализованы следующим образом (добавьте необходимые строки в функцию *ParallelBubble*):

```
...
  for(int i = 0; i < 2 * ProcNum; i++) {
    if((i % 2) == 1) {
      if((ProcRank % 2) == 1) {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
      else {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
    }
    else {
      if((ProcRank % 2) == 1) {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
      else {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
    }
    // Check the first and last processes
    if((ProcRank == ProcNum - 1) && (Offset ==  1)) continue;
    if((ProcRank == 0          ) && (Offset == -1)) continue;

    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData);

    // Data merging
    merge(pProcData, pProcData + BlockSize, pDualData,
      pDualData + BlockSize, pMergedData);
```

```
    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData + BlockSize, pMergedData + 2*BlockSize, pProcData);
  }
...
```

Отметим, что при отсутствии необходимых соседей (слева для процесса с рангом 0 и справа для процесса с рангом *ProcNum*–1) все действия, выполняемые на итерации, для процесса пропускаются (две операции проверки перед вызовом функции *ExchangeData*).

Этот этап, как и все предыдущие, необходимо проверить. Снова используем отладочную печать с помощью функции *ParallelPrintData*. Закомментируйте все предыдущие вызовы этой функции и добавьте новый вызов этой функции непосредственно после вызова функции сортировки *ParallelBubble*:

```
// Distributing the initial data between processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);
// TestDistribution(pData, DataSize, pProcData, BlockSize);

  // Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);
  ParallelPrintData(pProcData, BlockSize);
```

Для данных, задаваемых при помощи функции *DummyDataInitialization*, результат сортировки заранее известен. На процессе с рангом *i* получается блок результирующего вектора, содержащий элементы в диапазоне от *i***BlockSize* + 1 до (*i* + 1)**BlockSize*. Так, например, если параллельная программа запускается на четырех процессах, а количество сортируемых данных равно двадцати четырем, то на первом процессе должен получиться блок, содержащий числа от 1 до 6, а на втором процессе – блок, содержащий числа от 7 до 12 и т. д. (рис. 4.10).

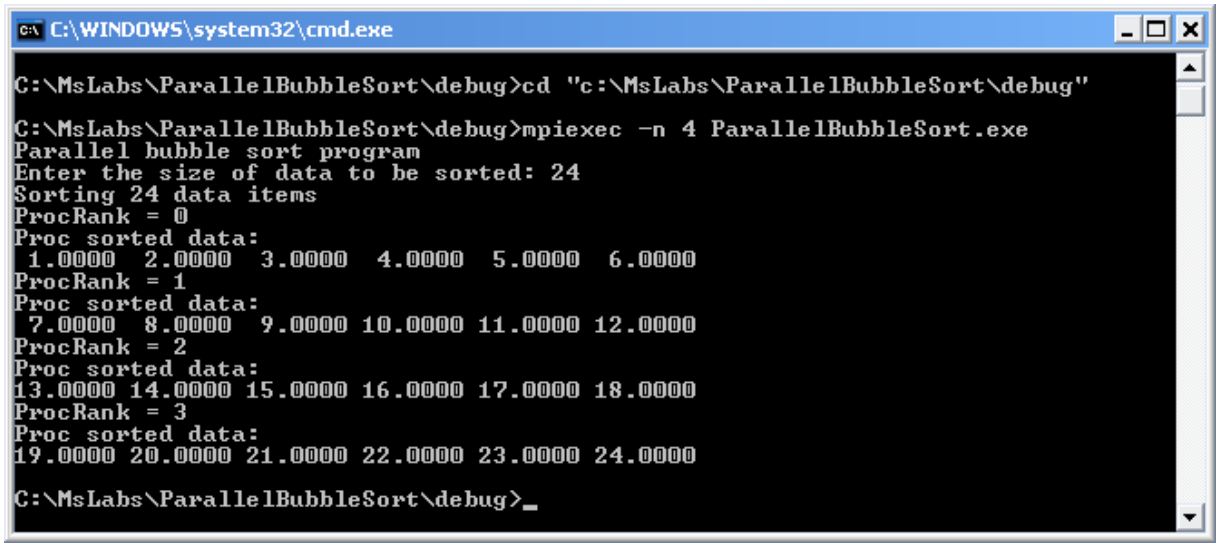

Рис. 4.10. Результат проверки отсортированных блоков в случае, когда программа запускается на четырех процессах, и количество сортируемых данных равно двадцати четырем

Скомпилируйте и запустите приложение. Проверьте правильность получения частичных результатов по приведенным выше соотношениям, задавая разное количество процессов и разное количество сортируемых данных.

Задание 10 – Сбор отсортированных данных

В завершение всех выполненных действий, организуем сбор отсортированных данных на ведущем процессе (на процессе с рангом 0). Следует отметить, что этот этап не является обязательным при выполнении параллельной сортировки, поскольку количество сортируемых данных может оказаться столь значительным, что может и не поместиться в оперативной памяти одного компьютера. В данной лабораторной работе этот этап приведен как пример еще одного учебного задания, а также для итогового сравнения результатов параллельной и последовательной сортировок.

Для сбора данных введем функцию *DataCollection*, которая состоит практически только из вызова ранее уже применявшейся функции *MPI_Gather*:

```
// Function for data collection
void DataCollection(double *pData, int DataSize, double *pProcData,
  int BlockSize) {

  MPI_Gather(pProcData, BlockSize, MPI_DOUBLE, pData,
    BlockSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Вызов функции из основной программы:

```
// Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);
  ParallelPrintData(pProcData, BlockSize);

  // Execution of data collection
  DataCollection(pData, DataSize, pProcData, BlockSize);
```

После выполнения сбора, добавьте в код основной функции приложения печать отсортированных данных при помощи функции *PrintData* на ведущем процессе параллельного приложения. Скомпилируйте и запустите приложение. Проверьте правильность работы программы.

Задание 11 – Проверка правильности работы программы

Теперь, после выполнения функции сбора, необходимо проверить правильность выполнения алгоритма. Для этого разработаем функцию *TestResult*, которая сравнит результаты последовательного и параллельного алгоритмов. Для выполнения последовательного алгоритма можно использовать ранее разработанную функцию *SerialBubbleSort*, находящуюся в файле **ParallelBubbleSortTest.cpp**.

Для того чтобы сортировка *SerialBubbleSort* оперировала теми же данными, что и разработанная нами функция *ParallelBubble*, необходимо создать копию этих данных, используя функцию *CopyData* (располагающуюся также в файле **ParallelBubbleSortTest.cpp**):

```
// Function for copying the sorted data
void CopyData(double *pData, int DataSize, double *pDataCopy) {
  copy(pData, pData + DataSize, pDataCopy);
}
```

Для проверки правильности сравним поэлементно результат последовательной сортировки с результатом, полученным разработанным параллельным алгоритмом, при помощи функции *CompareData*, также располагающейся в файле **ParallelBubbleSortTest.cpp**:

```
// Function for comparing the data
bool CompareData(double *pData1, double *pData2, int DataSize) {
  return equal(pData1, pData1 + DataSize, pData2);
}
```

Добавим вызовы этих функций в исходный код. В функции *main* необходимо объявить переменную, предназначенную для хранения копии данных, участвующей в последовательной сортировке, а также создать саму эту копию:

```
...
  int DataSize = 0;
  int BlockSize = 0;

  double *pSerialData = 0;

  // Process Initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  if (ProcRank == 0) {
    // Data copying
    pSerialData = new double[DataSize];
    CopyData(pData, DataSize, pSerialData);
  }
...
```

Кроме того, необходимо удалить занимаемую память, когда она становится больше не нужной:

```
...
  ProcessTermination(pData, pProcData);
  if (ProcRank == 0)
```

```
    delete []pSerialData;

  MPI_Finalize();

  return 0;
}
```

Далее, добавьте функцию *TestResult* в исходный текст программы:

```
// Function for testing the result of parallel bubble sort
void TestResult(double *pData, double *pSerialData, int DataSize) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialBubbleSort(pSerialData, DataSize);
    if(!CompareData(pData, pSerialData, DataSize)) {
      printf("The results of serial and parallel algorithms are "
        "NOT identical. Check your code\n");
    }
    else {
      printf("The results of serial and parallel algorithms are "
        "identical\n");
    }
  }
}
```

Результатом работы этой функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельного алгоритма независимо от того, насколько велик объем сортируемого массива при любых значениях исходных данных.

Закомментируйте вызовы функций, использующих отладочную печать, которые ранее использовались для контроля правильности выполнения этапов параллельного приложения (функции *TestDistribution*, *TestPartialResult*). Вместо функции *DummyDataInitialization*, которая генерирует исходные данные простого вида, вызовите функцию *RandomDataInitialization*, которая генерирует исходные данные при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Выполните эксперименты для различных объемов исходных данных. Убедитесь в том, что программа работает правильно.

Задание 12 – Реализация сортировки для любого количества сортируемых данных

Параллельная программа, которая разрабатывалась в ходе выполнения предыдущих заданий, была ориентирована на случай, когда количество сортируемых данных *DataSize* нацело делится на число процессоров *ProcNum*. В этом случае данные делятся между процессами поровну, и размеры блоков, обрабатываемых каждым из процессов, были одинаковыми.

Теперь рассмотрим случай, когда количество сортируемых данных *DataSize* не кратно числу процессов *ProcNum*. В этом случае значение *BlockSize* числа обрабатываемых данных на каждом процессе может различаться: некоторые процессы получат $\lfloor DataSize/ProcNum \rfloor$, а остальные - $\lceil Data\,Size/ProcNum \rceil$ элементов сортируемого набора (операция $\lfloor\ \rfloor$ означает округление значения до ближайшего меньшего целого числа, операция $\lceil\ \rceil$ – округление до ближайшего большего целого числа).

В функции *ProcessInitialization* уберем обработку ошибочной ситуации, которая возникает в случае, когда количество сортируемых данных не делится нацело на число процессов. Для определения количества данных, которые будут обрабатываться в процессах, применим следующий алгоритм распределения. Будем последовательно выделять данные процессам: в первую очередь определим, сколько значений будет обрабатывать процесс с рангом 0, затем – процесс с рангом 1, и так далее. Процессу с рангом 0 выделим $\lfloor Data\,Size/ProcNum \rfloor$ значений (результат операции $\lfloor\ \rfloor$ совпадает с результатом целочисленного деления переменной *DataSize* на переменную *ProcNum*). После выполнения этой операции остается распределить $DataSize - \lfloor DataSize/ProcNum \rfloor$ элементов данных между *ProcNum*-1 процессами. Следующему процессу назначим количество значений, равное результату целочисленного деления оставшегося количества значений на оставшееся число процессов и т.д. В общем случае, процессу с рангом *i* следует назначить $\lfloor RestData/(ProcNum - i) \rfloor$ значений (*RestData* есть оставшееся – не распределенное - количество значений).

Включите необходимые изменения в текст функции *ProcessInitilization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize,
  double *&pProcData, int& BlockSize) {
...
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  int RestData = DataSize;
  for(int i = 0; i < ProcRank; i++)
    RestData -= RestData / (ProcNum - i);
  BlockSize = RestData / (ProcNum - ProcRank);
...
}
```

Необходимые изменения необходимо выполнить и для функции *DataDistribution*, поскольку в случае блоков разного размера для распределения данных должна использоваться общая функция *MPI_Scatterv*. Как уже показывалось в предыдущих лабораторных работах, для вызова функции *MPI_Scatterv*, необходимо определить два вспомогательных массива, размер этих массивов совпадает с числом доступных процессов. Внесем необходимые изменения в код функции *DataDistribution*:

```
// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData, int
BlockSize) {

  // Alloc memory for temporary objects
  int *pSendInd = new int[ProcNum];
  int *pSendNum = new int[ProcNum];

  int RestData = DataSize;

  int CurrentSize = DataSize / ProcNum;
  pSendNum[0] = CurrentSize;
  pSendInd[0] = 0;
  for(int i = 1; i < ProcNum; i++) {
    RestData    -= CurrentSize;
    CurrentSize  = RestData / (ProcNum - i);
    pSendNum[i]  = CurrentSize;
    pSendInd[i]  = pSendInd[i - 1] + pSendNum[i - 1];
  }

  MPI_Scatterv(pData, pSendNum, pSendInd, MPI_DOUBLE, pProcData,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  //Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}
```

Аналогично для сбора данных, вместо функции *MPI_Gather*, ориентированной на сбор данных одинакового объема со всех процессов коммуникатора, необходимо использовать более общую функцию *MPI_Gatherv*. Как и в случае *MPI_Scatterv*, при использовании *MPI_Gatherv* требуются два дополнительных массивов:

```
// Function for data collection
void DataCollection(double *pData, int DataSize, double *pProcData,
  int BlockSize) {
  // Alloc memory for temporary objects
  int *pReceiveNum = new int[ProcNum];
  int *pReceiveInd = new int[ProcNum];

  int RestData = DataSize;

  pReceiveInd[0] = 0;
  pReceiveNum[0] = DataSize / ProcNum;
  for(int i = 1; i < ProcNum; i++) {
    RestData -= pReceiveNum[i - 1];
    pReceiveNum[i] = RestData / (ProcNum - i);
```

```
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  MPI_Gatherv(pProcData, BlockSize, MPI_DOUBLE, pData,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete []pReceiveNum;
  delete []pReceiveInd;
}
```

Кроме того, придется несколько изменить функцию *ExchangeData*, поскольку размеры блоков могут теперь различаться у соседних процессов. Добавьте еще один параметр *DualBlockSize* в заголовок этой функции и измените реализацию функции для учета этого факта:

```
// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData, int DualBlockSize) {

  MPI_Status status;

  MPI_Sendrecv(pProcData, BlockSize, MPI_DOUBLE, DualRank, 0,
    pDualData, DualBlockSize, MPI_DOUBLE, DualRank, 0,
    MPI_COMM_WORLD, &status);
}
```

Далее, необходимо изменить и функцию *ParallelBubble*. Перед выполнением обмена сортируемыми блоками необходимо выяснить размер блока соседа, что можно сделать, используя уже знакомую функцию *MPI_Sendrecv*:

```
...
    MPI_Status status;

    int DualBlockSize;

    MPI_Sendrecv(&BlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      &DualBlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      MPI_COMM_WORLD, &status);

    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData,
      DualBlockSize);
...
```

Далее, необходимо учесть тот факт, что количество сортируемых данных у соседнего процесса хранится теперь в переменной *DualBlockSize*, и соответствующим образом остальную часть функции. Прежде всего, изменения затронут выделение/освобождение памяти под массивы *pDualData* и *pMergedData*. Изменятся также и вызовы функций *ExchangeData*, *merge* и *copy*.

Внесите необходимые изменения в соответствии с приведенным ниже текстом:

```
  for(int i = 0; i < ProcNum; i++) {
...
    MPI_Sendrecv(&BlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      &DualBlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      MPI_COMM_WORLD, &status);

    double *pDualData   = new double[DualBlockSize];
    double *pMergedData = new double[BlockSize + DualBlockSize];

    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData,
      DualBlockSize);

    //Data merging
    merge(pProcData, pProcData + BlockSize,
      pDualData, pDualData + DualBlockSize, pMergedData);
```

```
    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData + DualBlockSize, pMergedData + BlockSize +
        DualBlockSize, pProcData);

    delete []pDualData;
    delete []pMergedData;
  }
...
```

Скомпилируйте и запустите приложение. Проверьте правильность сортировки при помощи функции *TestResult*.

При завершении данного задания можно подвести итог, что программа для параллельного алгоритма пузырьковой разработки подготовлена. В случае затруднений при выполнении тех или иных заданий можно свериться с полным текстом программы, приведенном в приложении 2 к данной лабораторной работе.

Задание 13 – Проведение вычислительных экспериментов

Основная задача при реализации параллельных алгоритмов решения сложных вычислительных задач – обеспечить ускорение вычислений (по сравнению с последовательным алгоритмом) за счет использования нескольких процессоров. Процессы параллельной программы могут быть запущены на разных процессорах вычислительной системы. При этом время выполнения параллельного алгоритма должно быть меньше, чем при выполнении последовательного алгоритма.

Определим время выполнения параллельного алгоритма. Для этого добавим в программный код замеры времени. Поскольку параллельный алгоритм включает этап распределения данных, сортировки данных на каждом процессе и сбора отсортированных значений, то отсчет времени должен начинаться непосредственно перед вызовом функции *DataDistribution*, и останавливаться сразу после выполнения функции *DataCollection*:

```
...
  double start, finish;

  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  start = MPI_Wtime();
  // Distributing the initial data between the processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);

  // Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);

  // Process data collection
  DataCollection(pData, DataSize, pProcData, BlockSize);
  finish = MPI_Wtime();;

  duration = finish - start;
  if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);

  // Process termination
  ProcessTermination(pData, pProcData);

  MPI_Finalize();
...
```

Очевидно, что таким образом будет распечатано то время, которое было затрачено на выполнение вычислений нулевым процессом. Возможно, что время выполнения алгоритма другими процессами может оказаться несколько отличным. Но эти отличия не должны быть существенными, поскольку на этапе разработки параллельного алгоритма было уделено особое внимание равномерной загрузке (*балансировке*) процессов.

Добавьте выделенный фрагмент кода в тело основной функции приложения. Скомпилируйте и запустите приложение. Выполните вычислительные эксперименты и заполните таблицу результатов:

Таблица 4.4. Результаты вычислительных экспериментов для параллельного метода пузырьковой сортировки

Номер теста	Количество данных	Последовательная пузырьковая сортировка	Последовательная стандартная сортировка	Параллельный алгоритм для количества процессов		
				2	4	8
1	10					
2	100					
3	10 000					
4	20 000					
5	30 000					
6	40 000					
7	50 000					

В графы "Последовательная пузырьковая сортировка" и "Последовательная стандартная сортировка" внесите время выполнения последовательных алгоритмов пузырьковой сортировки и сортировки, предоставляемой стандартной библиотекой, замеренное при проведении тестирования последовательной программы в упражнении 2. Далее, рассчитайте получившиеся ускорение вычислений как отношение времени последовательного алгоритма ко времени параллельного алгоритма и заполните таблицу (в графу "Ускорение 1" занесите ускорение параллельного алгоритма по отношению к последовательной пузырьковой сортировке, а в графу "Ускорение 2"- по отношению к последовательной сортировке из стандартной библиотеки).

Таблица 4.5. Ускорение вычислений, получаемое для параллельного метода пузырьковой сортировки

Номер теста	Параллельный алгоритм					
	2 процесса		4 процесса		8 процессов	
	Ускорение 1	Ускорение 2	Ускорение 1	Ускорение 2	Ускорение 1	Ускорение 2
1						
2						
3						
4						
5						
6						
7						

Для того чтобы оценить теоретическое время выполнения параллельного алгоритма, реализованного согласно вычислительной схеме, приведенной в упражнении 3, можно воспользоваться следующим соотношением:

$$T_p = ((n/p)\log_2(n/p) + 2n)\tau + p\cdot(\alpha + w\cdot(n/p)/\beta) \qquad (4.4)$$

(подробный вывод данного выражения приведен в подразделе 10.3.5 раздела 10 "Параллельные методы сортировки данных" учебных материалов курса). Здесь n – количество сортируемых данных, p – количество процессов, τ – время выполнения базовой операции сортировки (значение было нами вычислено при тестировании последовательного алгоритма), α – латентность и β – пропускная способность сети передачи данных. В качестве значений латентности и пропускной способности следует использовать величины, полученные при выполнении лабораторной работы "Выполнение заданий под управлением Microsoft Compute Cluster Server 2003".

Вычислите теоретическое время выполнения параллельного алгоритма по формуле (4.4). Результаты занесите в таблицу:

Таблица 4.6. Сравнение экспериментального и теоретического времени параллельного метода пузырьковой сортировки

Номер теста	Количество данных	Время выполнения параллельного алгоритма					
		2 процесса		4 процесса		8 процессов	
		Модель	Эксперимент	Модель	Эксперимент	Модель	Эксперимент
1	10						
2	100						
3	10 000						
4	20 000						
5	30 000						
6	40 000						
7	50 000						

Контрольные вопросы

- В качестве времени выполнения параллельного алгоритма было выбрано время выполнения первым процессом. Как нужно модифицировать код для того, чтобы выбрать максимальное среди времен, полученных на всех процессах?
- Насколько сильно отличаются времена, затраченные на выполнение последовательного алгоритма пузырьковой сортировки и параллельного алгоритма? Почему?
- Сравните времена работы параллельного и последовательного алгоритмов и проанализируйте результаты. Какие можно сделать выводы о перспективности улучшений полученного параллельного алгоритма? Можно ли было предсказать такие результаты заранее?
- Получилось ли ускорение при сортировке 10 элементов данных? Почему?
- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

- Модифицируйте разработанное приложение таким образом, чтобы в качестве локальной сортировки использовалась сортировка из стандартной библиотеки (для этого используйте функцию *SerialStdSort*, находящуюся в файле **ParallelBubbleSortTest.cpp**). Проведите вычислительные эксперименты по рассмотренной схеме и сравните результаты нового варианта программы с полученными ранее.
- Изучите другие параллельные алгоритмы сортировки (сортировка Шелла, различные виды быстрой сортировки – см. раздел 10 "Параллельные методы сортировки данных" учебных материалов курса). Разработайте программы, реализующие эти алгоритмы.

Приложение 1. Программный код последовательного приложения пузырьковой сортировки

Файл SerialBubbleSort.cpp

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include "SerialBubbleSort.h"
#include "SerialBubbleSortTest.h"

using namespace std;

const double RandomDataMultiplier = 1000.0;

int main(int argc, char *argv[]) {
  double *pData = 0;
  int DataSize = 0;
```

```
  time_t start, finish;
  double duration = 0.0;

  printf("Serial bubble sort program\n");

  // Process initialization
  ProcessInitialization(pData, DataSize);

  printf("Data before sorting\n");
  PrintData(pData, DataSize);

  // Serial bubble sort
  start = clock();
  SerialBubble(pData, DataSize);
  finish = clock();

  printf("Data after sorting\n");
  PrintData(pData, DataSize);

  duration = (finish - start) / double(CLOCKS_PER_SEC);
  printf("Time of execution: %f\n", duration);

  // Process termination
  ProcessTermination(pData);

  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize) {
  do {
    printf("Enter the size of data to be sorted: ");
    scanf("%d", &DataSize);
    if(DataSize <= 0)
      printf("Data size should be greater than zero\n");
  }
  while(DataSize <= 0);

  printf("Sorting %d data items\n", DataSize);

  pData = new double[DataSize];

  // Simple setting the data
  DummyDataInitialization(pData, DataSize);

  // Setting the data by the random generator
  //RandomDataInitialization(pData, DataSize);
}

// Function for computational process termination
void ProcessTermination(double *pData) {
  delete []pData;
}

// Function for simple setting the initial data
void DummyDataInitialization(double*& pData, int& DataSize) {
  for(int i = 0; i < DataSize; i++)
    pData[i] = DataSize - i;
}

// Function for initializing the data by the random generator
void RandomDataInitialization(double *&pData, int& DataSize) {
```

```
  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}

// Serial bubble sort algorithm
void SerialBubble(double *pData, int DataSize) {
  double Tmp;

  for(int i = 1; i < DataSize; i++)
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}
```

Файл SerialBubbleSortTest.cpp

```
#include <algorithm>
#include <cstdio>
using namespace std;

// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
  printf("\n");
}

// Sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}
```

Приложение 2. Программный код параллельного приложения пузырьковой сортировки

Файл ParallelBubbleSort.cpp

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include <cmath>
#include <algorithm>
#include <mpi.h>

#include "ParallelBubbleSort.h"
#include "ParallelBubbleSortTest.h"

using namespace std;

const double RandomDataMultiplier = 1000.0;

int ProcNum = 0;     // Number of available processes
int ProcRank = -1;   // Rank of current process

int main(int argc, char *argv[]) {
  double *pData = 0;
  double *pProcData = 0;
```

```
  int DataSize = 0;
  int BlockSize = 0;

  double *pSerialData = 0;

  double start, finish;
  double duration = 0.0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  if(ProcRank == 0)
    printf("Parallel bubble sort program\n");

  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  if (ProcRank == 0) {
    pSerialData = new double[DataSize];
    CopyData(pData, DataSize, pSerialData);
  }

  start = MPI_Wtime();
  // Distributing the initial data between processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);

  // Testing the distribution
  TestDistribution(pData, DataSize, pProcData, BlockSize);

  // Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);
  // Print the sorted data
  ParallelPrintData(pProcData, BlockSize);

  // Execution of data collection
  DataCollection(pData, DataSize, pProcData, BlockSize);
  TestResult(pData, pSerialData, DataSize);
  finish = MPI_Wtime();

  duration = finish - start;
  if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);

  if (ProcRank == 0)
    delete []pSerialData;

  // Process termination
  ProcessTermination(pData, pProcData);

  MPI_Finalize();

  return 0;
}

// Function for allocating the memory and setting the initial values
void   ProcessInitialization(double   *&pData,   int&   DataSize,   double
*&pProcData, int& BlockSize) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    do {
      printf("Enter the size of data to be sorted: ");
      scanf("%d", &DataSize);
```

```
      if(DataSize < ProcNum)
        printf("Data size should be greater than number of processes\n");

    } while(DataSize < ProcNum);

    printf("Sorting %d data items\n", DataSize);
  }

  // Broadcasting the data size
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  int RestData = DataSize;
  for(int i = 0; i < ProcRank; i++)
    RestData -= RestData / (ProcNum - i);
  BlockSize = RestData / (ProcNum - ProcRank);

  pProcData = new double[BlockSize];

  if(ProcRank == 0) {
    pData = new double[DataSize];

    // Data initalization
    //RandomDataInitialization(pData, DataSize);
    DummyDataInitialization(pData, DataSize);
  }
}

// Function for computational process termination
void ProcessTermination(double *pData, double *pProcData) {
  if(ProcRank == 0)
    delete []pData;

  delete []pProcData;
}

// Function for simple setting the data to be sorted
void DummyDataInitialization(double*& pData, int& DataSize) {
  for(int i = 0; i < DataSize; i++)
    pData[i] = DataSize - i;
}

// Function for initializing the data by the random generator
void RandomDataInitialization(double *&pData, int& DataSize) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}

// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData, int
BlockSize) {

  // Allocate memory for temporary objects
  int *pSendInd = new int[ProcNum];
  int *pSendNum = new int[ProcNum];

  int RestData = DataSize;

  int CurrentSize  = DataSize / ProcNum;
  pSendNum[0] = CurrentSize ;
  pSendInd[0] = 0;
  for(int i = 1; i < ProcNum; i++) {
```

```
    RestData    -= CurrentSize;
    CurrentSize  = RestData / (ProcNum - i);
    pSendNum[i]  = CurrentSize;
    pSendInd[i]  = pSendInd[i - 1] + pSendNum[i - 1];
  }

  MPI_Scatterv(pData, pSendNum, pSendInd, MPI_DOUBLE, pProcData,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}

// Function for data collection
void DataCollection(double *pData, int DataSize, double *pProcData, int
BlockSize) {

  // Allocate memory for temporary objects
  int *pReceiveNum = new int[ProcNum];
  int *pReceiveInd = new int[ProcNum];

  int RestData = DataSize;

  pReceiveInd[0] = 0;
  pReceiveNum[0] = DataSize / ProcNum;
  for(int i = 1; i < ProcNum; i++) {
    RestData -= pReceiveNum[i - 1];
    pReceiveNum[i] = RestData / (ProcNum - i);
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  MPI_Gatherv(pProcData, BlockSize, MPI_DOUBLE, pData,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete []pReceiveNum;
  delete []pReceiveInd;
}

// Parallel bubble sort algorithm
void ParallelBubble(double *pProcData, int BlockSize) {

  // Local sorting the process data
  SerialBubbleSort(pProcData, BlockSize);

  int Offset;
  split_mode SplitMode;

  for(int i = 0; i < ProcNum; i++) {
    if((i % 2) == 1) {
      if((ProcRank % 2) == 1) {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
      else {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
    }
    else {
      if((ProcRank % 2) == 1) {
        Offset    = -1;
```

```
        SplitMode = KeepSecondHalf;
      }
      else {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
    }

    // Check the first and last processes
    if((ProcRank == ProcNum - 1) && (Offset ==  1)) continue;
    if((ProcRank == 0          ) && (Offset == -1)) continue;

    MPI_Status status;

      int DualBlockSize;

    MPI_Sendrecv(&BlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      &DualBlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      MPI_COMM_WORLD, &status);

    double *pDualData   = new double[DualBlockSize];
    double *pMergedData = new double[BlockSize + DualBlockSize];

    // Data exchange
    ExchangeData(pProcData,   BlockSize,   ProcRank   +   Offset,   pDualData,
DualBlockSize);

    // Data merging
    merge(pProcData,   pProcData   +   BlockSize,   pDualData,   pDualData   +
DualBlockSize, pMergedData);

    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData   +   BlockSize,   pMergedData   +   BlockSize   +
DualBlockSize, pProcData);

    delete []pDualData;
    delete []pMergedData;
  }
}

// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData, int DualBlockSize) {

  MPI_Status status;
  MPI_Sendrecv(pProcData, BlockSize, MPI_DOUBLE, DualRank, 0,
    pDualData, DualBlockSize, MPI_DOUBLE, DualRank, 0,
    MPI_COMM_WORLD, &status);
}

// Function for testing the data distribution
void TestDistribution(double *pData, int DataSize, double *pProcData, int
BlockSize) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial data:\n");
    PrintData(pData, DataSize);
  }

  MPI_Barrier(MPI_COMM_WORLD);
```

```
  for (int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Block:\n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for parallel data output
void ParallelPrintData(double *pProcData, int BlockSize) {
  // Print the sorted data
  for(int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Proc sorted data: \n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for testing the result of parallel bubble sort
void TestResult(double *pData, double *pSerialData, int DataSize) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialBubbleSort(pSerialData, DataSize);
    //SerialStdSort(pSerialData, DataSize);
    if(!CompareData(pData, pSerialData, DataSize))
      printf("The results of serial and parallel algorithms are "
        "NOT identical. Check your code\n");
    else
      printf("The results of serial and parallel algorithms are "
        "identical\n");
  }
}
```

Файл ParallelBubbleSortTest.cpp

```
#include <cstdio>
#include <cstdlib>
#include <algorithm>

#include "ParallelBubbleSortTest.h"

using namespace std;

// Function for copying the sorted data
void CopyData(double *pData, int DataSize, double *pDataCopy) {
  copy(pData, pData + DataSize, pDataCopy);
}

// Function for comparing the data
bool CompareData(double *pData1, double *pData2, int DataSize) {
  return equal(pData1, pData1 + DataSize, pData2);
}

// Serial bubble sort algorithm
void SerialBubbleSort(double *pData, int DataSize) {
  double Tmp;
```

```
  for(int i = 1; i < DataSize; i++)
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}

// Sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}

// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
  printf("\n");
}
```