Нижегородский государственный университет им. Н.И Лобачевского

Факультет вычислительной математики кибернетики ННГУ

Учебно-исследовательская лаборатория «Математические и программные технологии для современных компьютерных систем (Информационные технологии)»

Библиотека параллельных алгоритмов ParaLib. Руководство пользователя

Нижний Новгород
2004

Содержание

1. Общая характеристика библиотеки

Библиотека ParaLib включает в свой состав процедуры для параллельного решения ряда типовых задач вычислительной математики на многопроцессорных вычислительных системах с распределенной памятью (кластерах). В библиотеке реализованы:

- Для задач **упорядочения (сортировки) данных** - параллельные алгоритмы пузырьковой и быстрой сортировки и метод сортировки Шелла:
- Для задачи **умножения матриц** - блочные алгоритмы Фокса и Кеннона и метод умножения матриц при ленточной схеме разделения данных;
- Для задач **обработки графов** - алгоритм Прима для нахождения минимально охватывающего дерева графа и метод Дейкстры для поиска кратчайших путей.

Для демонстрации работоспособности алгоритмов в комплекте поставки имеется исполняемый **демо-вариант библиотеки** (ParaLib Viewer), используя который можно выполнить все необходимые эксперименты для оценки эффективности осуществляемых параллельных вычислений. В рамках демо-варианта библиотеки предоставляется возможность:

- *осуществить постановку вычислительной задачи*, для которой имеются реализованные параллельные алгоритмы решения,
- *выполнить задание параметров* задачи;
- *выбрать параллельный метод* для решения выбранной задачи;
- *определить* число используемых процессоров;
- *выполнить эксперимент* для параллельного решения выбранной задачи;
- *накапливать и анализировать результаты выполненных экспериментов*; по запомненным результатам в системе имеется возможность построения графиков, характеризующих параллельные вычисления зависимости в*ремени решения* от параметров задачи.

Выполнение экспериментов может осуществляться или *локально* - на одном компьютере, где имеется библиотека передачи сообщений MPI (многопоточное выполнение эксперимента в режиме разделения времени), или на реальной многопроцессорной кластерной вычислительной системе в режиме *доступа к вычислительному кластеру*.

Назначение библиотеки ParaLib носит учебно-исследовательский характер и, тем самым, библиотека может быть использована при обучении специалистов по тем или иным дисциплинам в области параллельного программирования. Программы библиотеки могут быть применяться также и в качестве образцов написания параллельных программ с использованием библиотеки MPI. Использование библиотеки ParaLib может предварять работы по освоению профессиональных библиотек параллельных вычислений (таких, например, как библиотека PLAPACK – см. [10]).

2. Описание методов библиотеки

2.1. Алгоритмы сортировки

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

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

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

$$T_1 \sim n^2 .$$

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

$$T_1 \sim n \log_2 n .$$

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*, *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.

2.1.1. Алгоритм пузырьковой сортировки

Описание метода

Алгоритм пузырьковой сортировки основан на применении 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:

```
// операция "сравнить и переставить"
        if ( a[i] > a[j] ) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
                }
```

На первой итерации алгоритма осуществляется последовательное сравнение всех соседних элементов; в результате прохода по упорядочиваемому набору данных в последнем (верхнем) элементе оказывается максимальное значение ("всплывание пузырька"); далее для продолжения сортировки этот уже упорядоченный элемент не рассматривается и действия алгоритма повторяются:

```
// пузырьковая сортировка
  for ( i=1; i<n; i++ ){
 for ( j=0; j<n-i; j++ )
<сравнить и переставить элементы (a[j],a[j+1])>
                }
```

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания: сравнение пар соседних элементов происходит строго последовательно. Для организации параллельных вычислений обычно используется модификация алгоритма пузырьковой сортировки – *метод чет-нечетной перестановки* [6]. 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. Т.е., на всех нечетных итерациях сравниваются пары

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

на четных итерациях обрабатываются элементы

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

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

Параллельное обобщение этого алгоритма не вызывает затруднений, так как сравнение элементов в парах происходит независимо и может выполняться одновременно. Сначала рассмотрим схему вычислений, когда на каждый процессор приходится один элемент исходного массива. Предположим, что процессоры соединены в кольцо и элементы a_i расположены на процессорах p_i (i=1, 2,..., n). Тогда сравнение пары значений a_i и a_{i+1}, $1 \le i < n$, располагаемых на процессорах P_i и P_{i+1} соответственно, можно организовать следующим образом:

- выполнить взаимообмен имеющихся на процессорах P_i и P_{i+1} значений (с сохранением на этих процессорах исходных элементов);
- сравнить на каждом процессоре P_i и P_{i+1} получившиеся одинаковые пары значений (a_i, a_{i+1});

результаты сравнения используются для разделения данных между процессорами – на одном процессоре (например, P_i) остается меньший элемент, другой процессор (т.е. P_{i+1}) запоминает для дальней обработки большее значение пары

$$a_i' = \min(a_i, a_{i+1}),\ a_{i+1}' = \max(a_i, a_{i+1}).$$

Рассмотренная параллельная схема может быть надлежащим образом адаптирована и для случая $p < n$, когда количество процессоров является меньшим числа упорядочиваемых значений. В данной ситуации каждый процессор будет содержать уже не единственное значение, а часть (*блок* размера n/p) сортируемого набора данных. Эти блоки обычно упорядочиваются в самом начале сортировки на каждом процессоре в отдельности при помощи какого-либо быстрого алгоритма (предварительная стадия параллельной сортировки). Далее, 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}

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

Следует отметить, что сформированные в результате такой процедуры блоки на процессорах P_i и P_{i+1} совпадают по размеру с исходными блоками A_i и A_{i+1} и все значения, расположенные на процессоре P_i, являются меньшими значений на процессоре P_{i+1}.

Рассмотренная процедура обычно именуется в литературе как операция "*сравнить и разделить*" (*compare-split*).

Структура метода

Схема вызовов функций в программе, реализующей метод пузырьковой сортировки, представлена на рис. 1.

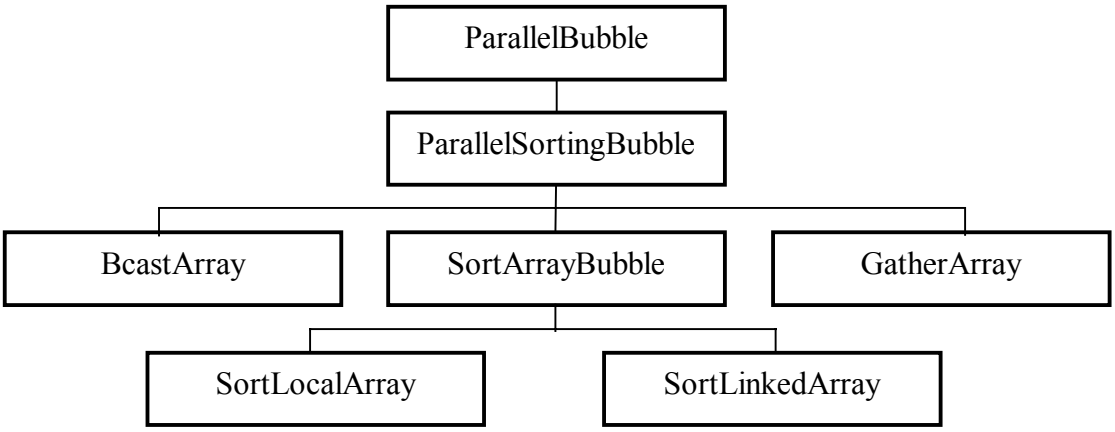

Рис. 1. Схема вызовов функций. Пузырьковая сортировка

Управляющий процессор (процессор 0) выполняет:

- разбиение исходного массива на блоки и их пересылку на соответствующие процессоры (функция *BcastArray*);
- выполнение итераций параллельной сортировки (эта часть действий является общей для всех процессоров и детально описана в алгоритме работы функциональных процессоров - см. ниже);
- сбор отсортированных частей от всех процессоров (функция *GatherArray*).

Функциональные процессоры выполняют:

- прием от процессора 0 соответствующих блоков (функция *BcastAraay*);
- внутреннюю сортировку при помощи алгоритма пузырьковой сортировки на каждом процессоре (функция *SortLocalArray*);
- реализацию параллельной чет-нечетной перестановки (функции *SortBubble* и *SortLinkedArray*):
  - обмен данными между соседними процессорами;
  - объединение слиянием упорядоченных блоков;
  - разделение объединенного блока (процессор с меньшим номером сохраняет блок с меньшими элементами, процессор с большим номером – блок с большими элементами);
- передача результата (отсортированного блока данных) на управляющий процессор.

Формат вызова

Вызов параллельного алгоритма пузырьковой сортировки имеет вид:

```
void ParallelBubble ( int *pData, int DataSize)
```

где:

- `int *pData` – сортируемый массив,
- `int DataSize` – количество элементов сортируемого массива.

Пример использования

```
#include "paralib.h"
#include "mpi.h"

int main(int argc, char *argv[])
{
MPI_Init ( &argc, &argv );

    int DataSize=100;
int *pData = new int [DataSize];

// генерация массива случайным образом
for(int i=0; i<DataSize; i++)
    pData[i] = rand();

//запуск параллельного алгоритма пузырьковой сортировки
```

```
ParallelBubble (pData, DataSize);

MPI_Finalize();
   return 0;
      }
```

2.1.2. Быстрая сортировка

Описание метода

Алгоритм быстрой сортировки [4] основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности. На первой итерации метода осуществляется деление исходного набора данных на первые две части – для организации такого деления выбирается некоторый ведущий элемент и все значения набора, меньшие ведущего элемента, переносятся в первый формируемый блок, все остальные значения образуют второй блок набора. На второй итерации сортировки описанные правила применяются последовательно для обоих сформированных блоков и т.д. После выполнения *log(n)* итераций исходный массив данных оказывается упорядоченным (при оптимальном выборе ведущих элементов).

Параллельное обобщение алгоритма быстрой сортировки наиболее простым способом может быть получено для вычислительной системы с топологией в виде *N*-мерного гиперкуба (т.е. $p=2^N$). Пусть, как и ранее, исходный набор данных распределен между процессорами блоками одинакового размера *n/p*; результирующее расположение блоков должно соответствовать нумерации процессоров гиперкуба. Возможный способ выполнения первой итерации параллельного метода при таких условиях может состоять в следующем:

- выбрать каким-либо образом ведущий элемент и разослать его по всем процессорам системы;
- разделить на каждом процессоре имеющийся блок данных на две части с использованием полученного ведущего элемента;
- образовать пары процессоров, для которых битовое представление номеров отличается только в позиции N, и осуществить взаимообмен данными между этими процессорами; в результате таких пересылок данных на процессорах, для которых в битовом представлении номера бит позиции N равен 0, должны оказаться части блоков со значениями, меньшими ведущего элемента; процессоры с номерами, в которых бит N равен 1, должны собрать, соответственно, все значения данных, превышающие значение ведущего элемента.

В результате выполнения такой итерации сортировки исходный набор оказывается разделенным на две части, одна из которых (со значениями меньшими, чем значение ведущего элемента) располагается на процессорах, в битовом представлении номеров которых бит *N* равен 0. Таких процессоров всего *p/2* и, таким образом, исходный *N*-мерный гиперкуб оказывается разделенным на два гиперкуба размерности *(N-1)*. К этим подкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После *N*-кратного повторения подобных итераций для завершения сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре вычислительной системы. Эффективность параллельного метода быстрой сортировки, как и в последовательном варианте, во многом зависит от успешности выбора значений ведущих элементов. Определение общего правила для выбора этих значений является достаточно трудной задачей; сложность такого выбора может быть снижена, если выполнить упорядочение локальных блоков процессоров перед началом сортировки и обеспечить однородное распределение сортируемых данных между процессорами вычислительной системы.

Структура метода

Схема вызовов функций в программе, реализующей метод быстрой сортировки, представлена на рис. 2.

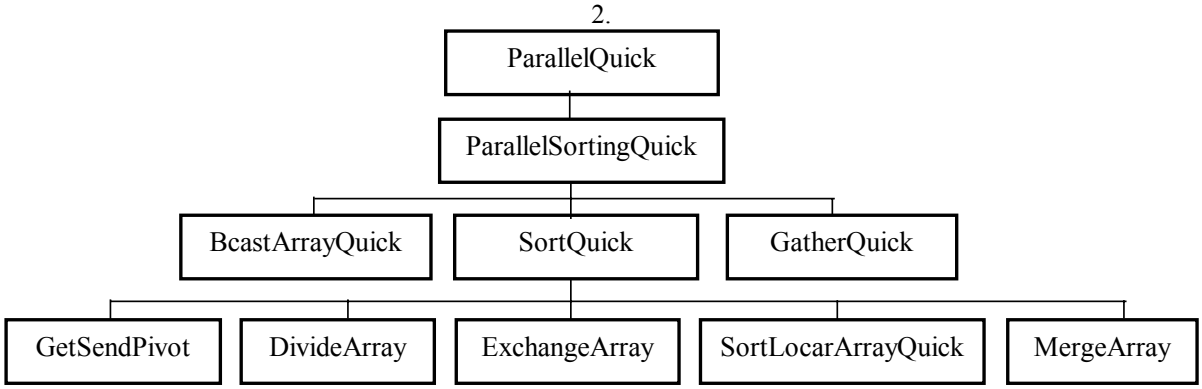

Рис. 2. Схема вызовов функций. Быстрая сортировка

Управляющий процессор (процессор 0) осуществляет:

- разбиение исходного массива на блоки и их пересылку на соответствующие процессоры (функция *BcastArrayQuick*);
- выбор ведущего элемента и рассылку его всем процессорам подгиперкуба (функция *GetSendPivot*);
- выполнение итераций параллельной сортировки (эта часть является общей для всех процессоров и детально описана в алгоритме работы функциональных процессоров - см. ниже);
- сбор отсортированных частей от всех процессоров (функция *GatherQuick*).

Функциональные процессоры осуществляют:

- прием от процессора 0 соответствующих блоков (функция *BcastArrayQuick*);
- прием ведущего элемента (функция *GetSendPivot*);
- разделение данных на две части относительно ведущего элемента (функция *DivideArraya*);
- обмен данными с соответствующим процессором из другого подгиперкуба (функция *ExchangeArray*);
- внутреннюю сортировку при помощи алгоритма быстрой сортировки (функция *SortLocalArrayQuick*) и алгоритма слияния (функция *MergeArray*)
- передачу результата (отсортированного блока данных) на управляющий процессор.

Формат вызова

Вызов параллельного алгоритма быстрой сортировки имеет вид:

```
void ParallelQuick ( int *pData, int DataSize )
```

где:

- `int *pData` – сортируемый массив,
- `int DataSize` – количество элементов сортируемого массива.

Пример использования

```
#include "paralib.h"
#include "mpi.h"

int main(int argc, char *argv[])
{
   MPI_Init ( &argc, &argv );

   int DataSize=100;
int *pData = new int [DataSize];

// генерация массива случайным образом
   for(int i=0; i<DataSize; i++)
      pData[i] = rand();

//запуск параллельного алгоритма быстрой сортировки
   ParallelQuick (pData, DataSize);

   MPI_Finalize();
   return 0;
}
```

2.1.3. Сортировка Шелла

Описание метода

Параллельный алгоритм сортировки Шелла [4] может быть получен как обобщение метода параллельной пузырьковой сортировки. Основное различие состоит в том, что на первых итерациях алгоритма Шелла происходит сравнение пар элементов, которые в исходном наборе данных находятся далеко друг от друга (для упорядочивания таких пар в пузырьковой сортировке может понадобиться достаточно большое количество итераций).

Для алгоритма Шелла может быть предложен параллельный аналог метода, если топология коммуникационный сети имеет структуру N-мерного гиперкуба (т.е. количество процессоров равно $p=2^N$). Выполнение сортировки в таком случае может быть разделено на два последовательных этапа. На первом этапе (N итераций) осуществляется взаимодействие процессоров, являющихся соседними в структуре гиперкуба (но эти процессоры могут оказаться далекими при линейной нумерации; для установления соответствия двух систем нумерации процессоров обычно используется код Грея). Второй этап состоит в реализации обычных итераций параллельного алгоритма чет-нечетной перестановки. Итерации данного этапа выполняются до прекращения фактического изменения сортируемого набора и, тем самым, общее количество L таких итераций может быть различным - от 2 до p.

Структура метода

Схема вызовов функций в программе, реализующей метод пузырьковой сортировки, представлена на рис. 3.

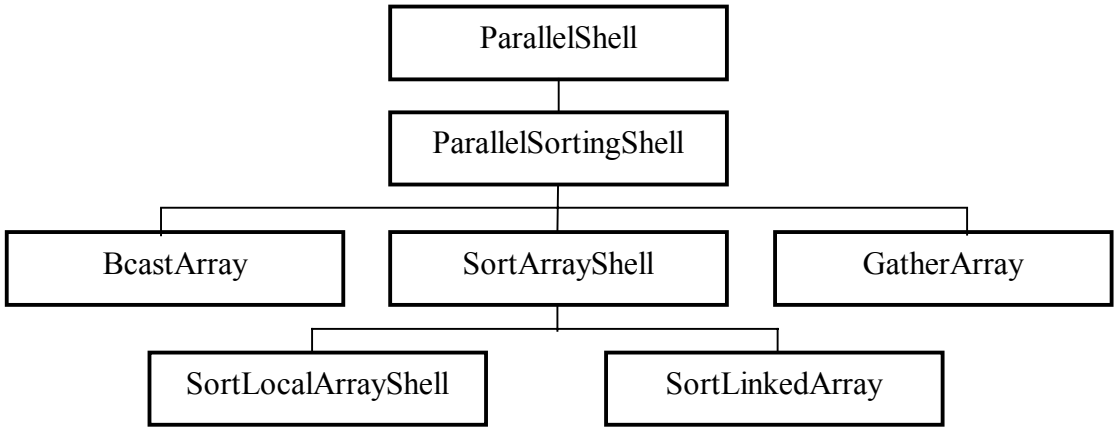

Рис. 3. Схема вызовов функций. Сортировка Шелла

Управляющий процессор (процессор 0) осуществляет:

- разбиение исходного массива на блоки и их пересылку на соответствующие процессоры (функция *BcastArray*);
- выполнение итераций параллельной сортировки (эта часть является общей для всех процессоров и детально описана в алгоритме работы функциональных процессоров - см. ниже);
- сбор отсортированных частей от всех процессоров (функция *GatherArray*).

Функциональные процессоры осуществляют:

- прием от процессора 0 соответствующих блоков (функция *BcastArray*);
- внутреннюю сортировку при помощи алгоритма быстрой сортировки на каждом процессоре (функция *SortLocalArrayShell*);
- реализацию итераций алгоритма:

1. первая фаза алгоритма - операция “сравнения-разбиения” вдоль d измерения в гиперкубе (функции *SortArrayShell* и *SortLinkedArray*)
2. вторая фаза алгоритма - параллельная чет-нечетная перестановка (функции *SortArrayShell* и *SortLinkedArray*):

- обмен данными между соседними процессорами;
- объединение упорядоченных блоков (слиянием);

- разделение объединенного блока (процессор с меньшим номером сохраняет блок с меньшими элементами, процессор с большим номером – блок с большими элементами);

- передача результата (отсортированного блока данных) на управляющий процессор.

Формат вызова

Вызов параллельного алгоритма сортировки Шелла имеет вид:

```
void ParallelShell ( int *pData, int DataSize)
```

где:

- `int *pData` – сортируемый массив,
- `int DataSize` – количество элементов сортируемого массива.

Пример использования

```
#include "paralib.h"
  #include "mpi.h"

int main(int argc, char *argv[])
                {
   MPI_Init ( &argc, &argv );

          int DataSize=10;
int *pData = new int [DataSize];

// генерация массива случайным образом
    for(int i=0; i<DataSize; i++)
          pData[i] = rand();

//запуск параллельного алгоритма сортировки Шелла
       ParallelShell (pData, DataSize);

          MPI_Finalize();
             return 0;
                }
```

2.2. Матричное умножение

Задача умножения матрицы на матрицу определяется соотношениями:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, 1 \le i, j \le n$$

(для простоты изложения материала будем предполагать, что перемножаемые матрицы A и B являются квадратными и имеют порядок $n \times n$). Как следует из приведенных соотношений, вычислительная сложность задачи является достаточно высокой (оценка количества выполняемых операций имеет порядок n^3).

Основу возможности параллельных вычислений для матричного умножения составляет независимость расчетов для получения элементов c_{ij} результирующей матрицы C. Тем самым все элементы матрицы C могут быть вычислены параллельно при наличии n^2 процессоров, при этом на каждом процессоре будет располагаться по одной строке матрицы A и одному столбцу матрицы B. При меньшем количестве процессоров подобный подход приводит к *ленточной схеме* разбиения данных, когда на процессорах располагаются по несколько строк и столбцов (*полос*) исходных матриц.

Другой широко используемый подход для построения параллельных способов выполнения матричного умножения состоит в использовании *блочного представления* матриц, при котором исходные матрицы A, B и результирующая матрица C рассматривается в виде наборов блоков (как правило, квадратного вида некоторого размера $m \times m$). Тогда операцию матричного умножения матриц A и B в блочном виде можно представить следующим образом:

$$\begin{pmatrix} A_{11}A_{12}...A_{1k} \\ \cdots \\ A_{k1}A_{k2}...A_{kk} \end{pmatrix} \times \begin{pmatrix} B_{11}B_{12}...B_{1k} \\ \cdots \\ B_{k1}B_{k2}...B_{kk} \end{pmatrix} = \begin{pmatrix} C_{11}C_{12}...C_{1k} \\ \cdots \\ c_{k1}C_{k2}...C_{kk} \end{pmatrix},$$

где каждый блок C_{ij} матрицы C определяется в соответствии с выражением:

$$C_{ij} = \sum_{l=1}^{k} A_{il} B_{lj} .$$

Полученные блоки C_{ij} также являются независимыми и, как результат, возможный подход для параллельного выполнения вычислений может состоять в выделении для расчетов, связанных с получением отдельных блоков C_{ij}, на разных процессорах. Применение подобного подхода позволяет получить многие *эффективные параллельные методы умножения блочно-представленных матриц*.

2.2.1. Алгоритм Фокса

Описание метода

Для организации параллельных вычислений при блочном представлении матриц предположим, что процессоры образуют логическую прямоугольную решетку размером $k \times k$ (обозначим через p_{ij} процессор, располагаемый на пересечении i строки и j столбца решетки). Основные положения параллельного метода, известного как алгоритм Фокса (*Fox*) [6], состоят в следующем:

- каждый из процессоров решетки отвечает за вычисление одного блока матрицы C;
- в ходе вычислений на каждом из процессоров p_{ij} располагается четыре матричных блока:
  - блок C_{ij} матрицы C, вычисляемый процессором;
  - блок A_{ij} матрицы A, размещенный в процессоре перед началом вычислений;
  - блоки $A_{ij}^{'}$, $B_{ij}^{'}$ матриц A и B, получаемые процессором в ходе выполнения вычислений.

Выполнение параллельного метода включает:

- этап инициализации, на котором на каждый процессор p_{ij} передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех процессорах;
- этап вычислений, на каждой итерации l, $1 \le l \le k$, которого выполняется:
  - для каждой строки i, $1 \le i \le k$, процессорной решетки блок A_{ij} процессора p_{ij} пересылается на все процессоры той же строки i; индекс j, определяющий положение процессора p_{ij} в строке, вычисляется по соотношению
$$j = (i + l - 1) \bmod k + 1,$$
($\bmod$ есть операция получения остатка от целого деления);
  - полученные в результате пересылок блоки A'_{ij}, B'_{ij} каждого процессора p_{ij} перемножаются и прибавляются к блоку C_{ij}:
$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij};$$
  - блоки B'_{ij} каждого процессора p_{ij} пересылаются процессорам p_{ij}, являющимися соседями сверху в столбцах процессорной решетки (блоки процессоров из первой строки решетки пересылаются процессорам последней строки решетки).

Структура метода

Схема вызовов функций в программе, реализующей метод Фокса, представлена на рис. 4.

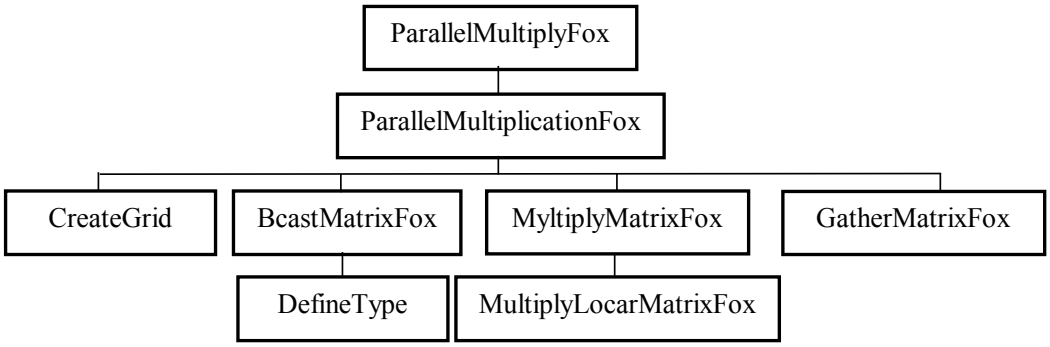

Рис. 4. Схема вызовов функций. Алгоритм Фокса

Управляющий процессор (процессор 0) выполняет:

- создание топологии "решетка" (функция *CreateGrid*);
- создание производного типа – "блок матрицы" (функция *DefineType*) и пересылку блоков на соответствующие процессоры в топологии "решетка" (функция *BcastMatrixFox*);
- выполнение итераций параллельного перемножения матриц (функция *MultiplyMatrixFox*);
- сбор результатов (перемноженных блоков) от всех процессоров (функция *GatherMatrixFox*).

Функциональные процессоры выполняют:

- создание топологии "решетка" (функция *CreateGrid*);
- создание производного типа блок матрицы (функция *DefineType*);
- прием от процессора 0 соответствующих блоков матриц (функция *BcastMatrixFox*);
- выполнение итераций параллельного перемножения матриц (*функция MultiplyMatrixFox*);
- передача результата (перемноженного блока) на управляющий процессор.

Формат вызова

Вызов параллельного алгоритма Фокса имеет вид:

```
void ParallelMultiplyFox ( int *pMatrixA, int *pMatrixB, int **pMatrixC,
                           int DataSize )
```

где:

- `int *pMatrixA` и `int *pMatrixB` – перемножаемые матрицы,
- `int **pMatrixC` – матрица результата,
- `int DataSize` – порядок матриц.

Пример использования

```
#include "paralib.h"
#include "mpi.h"

int main(int argc, char *argv[])
{
  MPI_Init ( &argc, &argv );

  int DataSize = 100;
int *pMatrixA = new int[DataSize*DataSize];
int *pMatrixB = new int[DataSize*DataSize];
  int *pMatrixC;

  for( int i=0; i<DataSize*DataSize; i++)
  {
    pMatrixA[i] = (int) rand()/1000;
    pMatrixB[i] = (int) rand()/1000;
  }
```

```
        //запуск параллельного алгоритма Фокса
ParallelMultiplyFox (pMatrixA, pMatrixB, &pMatrixC, DataSize );

                MPI_Finalize();
                   return 0;
                       }
```

2.2.2. Алгоритм Кэннона

Описание метода

Отличие алгоритма Кэннона [6] от метода Фокса состоит в изменении схемы начального распределения блоков перемножаемых матриц между процессорами вычислительной системы. Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы располагаемые блоки на процессорах могли бы быть перемножены без каких-либо дополнительных передач данных между процессорами. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между процессорами в ходе вычислений может осуществляться с использованием более простых коммуникационных операций.

С учетом высказанных замечаний этап инициализации алгоритма Кэннона включает выполнение следующих операций передач данных:

- на каждый процессор p_{ij} передаются блоки A_{ij}, B_{ij};
- для каждой строки i процессорной решетки блоки матрицы A сдвигаются на $(i-1)$ позиций влево;
- для каждого столбца j процессорной решетки блоки матрицы B сдвигаются на $(j-1)$ позиций вверх.

В ходе вычислений на каждой итерации алгоритма Кэннона каждый блок матрицы A сдвигается на один процессор влево по решетке, а каждый блок матрицы B - на один процессор вверх.

Структура метода

Схема вызовов функций в программе, реализующей метод Кэннона, представлена на рис. 5.

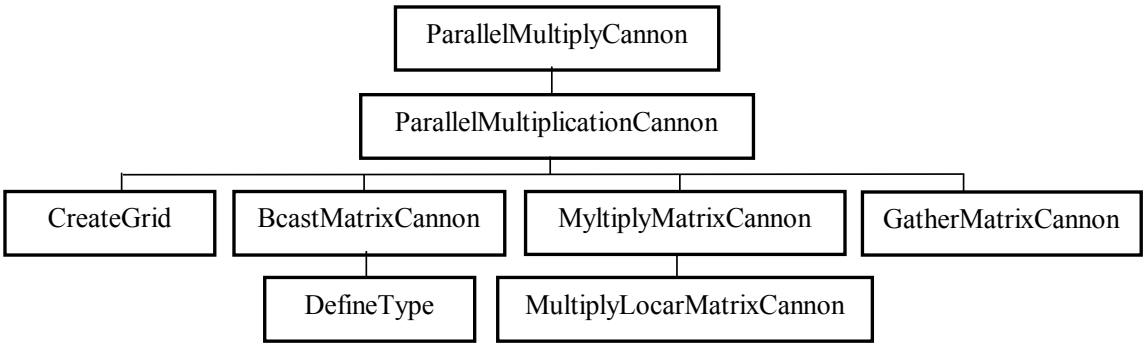

Рис. 5. Схема вызовов функций. Алгоритм Кэннона

Управляющий процессор (процессор 0) выполняет:

- создание топологии "решетка" (функция *CreateGrid*);
- создание производного типа – "блок матрицы" (функция *DefineType*) и пересылку блоков на соответствующие процессоры в топологии "решетка" (функция *BcastMatrixCannon*);
- выполнение итераций параллельного перемножения матриц (эта часть действий является общей для всех процессоров и детально описана в алгоритме работы функциональных процессоров. См. ниже);
- сбор результатов (перемноженных блоков) от всех процессоров (функция *GatherMatrixCannon*).

Функциональные процессоры выполняют:

- создание топологии "решетка" (функция *CreateGrid*);
- создание производного типа "блок матрицы" (функция *DefineType*);
- прием от процессора 0 соответствующих блоков матриц (функция *BcastMatrixCannon*);

- выполнение итераций параллельного перемножения матриц (функция *MultiplyMatrixCannon*):
  - локальное перемножение блоков матриц на каждом процессоре (функция *MultiplyLocalMatrixCannon*);
  - циклический сдвиг блоков Ai,j влево на один шаг;
  - циклический сдвиг блоков Bi,j вверх на один шаг;
- передача результата (перемноженного блока) на управляющий процессор.

Формат вызова

Вызов параллельного алгоритма Кэннона имеет вид:

```
void ParallelMultiplyCannon ( int *pMatrixA, int *pMatrixB,
            int **pMatrixC, int DataSize )
```

где:

`int *pMatrixA` и `int *pMatrixB` – перемножаемые матрицы,
`int **pMatrixC` – матрица результата,
`int DataSize` – порядок матриц.

Пример использования

```
#include "paralib.h"
  #include "mpi.h"

int main(int argc, char *argv[])
                    {
   MPI_Init ( &argc, &argv );

             int DataSize = 100;
int *pMatrixA = new int[DataSize*DataSize];
int *pMatrixB = new int[DataSize*DataSize];
              int *pMatrixC;

    // генерация матриц случайным образом
  for( int i=0; i<DataSize*DataSize; i++)
                    {
     pMatrixA[i] = (int) rand()/1000;
     pMatrixB[i] = (int) rand()/1000;
                    }

//запуск параллельного алгоритма Кэннона
ParallelMultiplyCannon (pMatrixA, pMatrixB, &pMatrixC, DataSize);

                MPI_Finalize();
                   return 0;
                       }
```

2.2.3. Ленточный алгоритм

Описание метода

При ленточной схеме разделения данных исходные матрицы разбиваются на горизонтальные (для матрицы *A*) и вертикальные (для матрицы *B*) полосы (см. рис. 6). Получаемые полосы распределяются по процессорам, при этом на каждом из имеющегося набора процессоров располагается только по одной полосе матриц *A* и *B*. Перемножение полос (а выполнение процессорами этой операции может быть выполнено параллельно) приводит к получению части блоков результирующей матрицы *C*. Для вычисления оставшихся блоков матрицы *C* сочетания полос матриц *A* и *B* на процессорах должны быть изменены. В наиболее простом виде это может быть обеспечено, например, при кольцевой топологии вычислительной сети (при числе процессоров равном количеству полос) – в этом случае необходимое для матричного умножения изменение положения данных может быть обеспечено циклическим сдвигом полос матрицы *B* по кольцу. После многократного выполнения описанных действий (количество необходимых повторений является равным числу процессоров) на каждом процессоре получается набор блоков, образующий горизонтальную полосу матрицы *C*.

Рассмотренная схема вычислений позволяет определить параллельный алгоритм матричного умножения при ленточной схеме разделения данных как итерационную процедуру, на каждом шаге которой происходит параллельное выполнение операции перемножения полос и последующего циклического сдвига полос одной из матриц по кольцу.

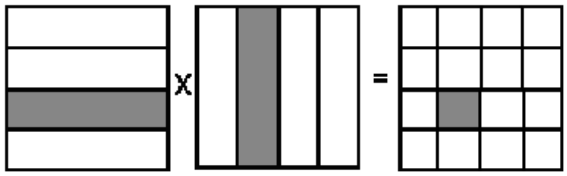

Рис. 6. Разбиение данных при выполнении ленточного алгоритма

Структура метода

Схема вызовов функций в программе, реализующей ленточный алгоритм умножения матриц, представлена на рис. 7.

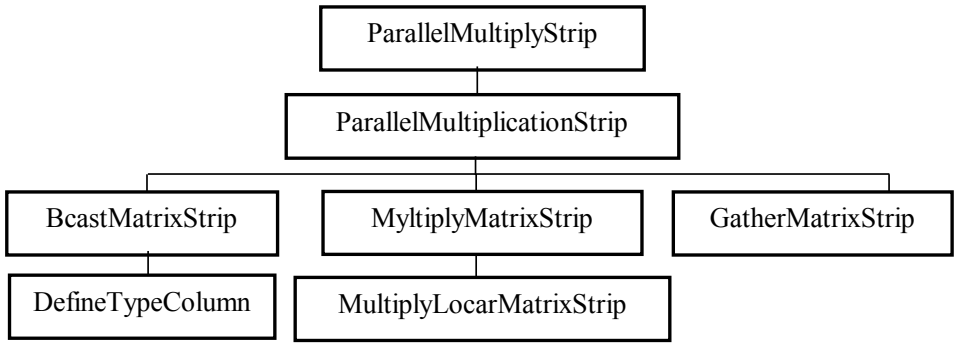

Рис. 7. Схема вызовов функций. Ленточный алгоритм

Управляющий процессор (процессор 0) выполняет:

- создание производного типа – "полоса матрицы" (функция *DefineTypeColumn*) и рассылку полос матриц на соответствующие процессоры (функция *BcastMatrixStrip*);
- выполнение итераций параллельного перемножения матриц (эта часть действий является общей для всех процессоров и детально описана в алгоритме работы функциональных процессоров - см. ниже);
- сбор результатов (перемноженных блоков) от всех процессоров (функция *GatherMatrixStrip*).

Функциональные процессоры выполняют:

- создание производного типа – "полоса матрицы" (функция *DefineType*);
- прием от процессора 0 соответствующих полос матриц (функция *BcastMatrixStrip*);
- выполнение итераций параллельного перемножения матриц (функция *MultiplyMatrixStrip*):

– локальное перемножение элементов на каждом процессоре (функция *MultiplyLocalMatrixStrip*);

– циклический сдвиг полос матрицы B вдоль матрицы A (функция *MultiplyMatrixStrip*);

- передача результата (перемноженного блока) на управляющий процессор.

Формат вызова

Вызов параллельного алгоритма ленточного умножения матриц имеет вид:

```
void ParallelMultiplyStrip ( int *pMatrixA, int *pMatrixB,
          int **pMatrixC, int DataSize )
```

где:

– `int *pMatrixA` и `int *pMatrixB` – перемножаемые матрицы,

– `int **pMatrixC` – матрица результата,

– `int DataSize` – порядок матриц.

Пример использования

```
#include "paralib.h"
  #include "mpi.h"

int main(int argc, char *argv[])
                {
   MPI_Init ( &argc, &argv );

        int DataSize = 100;
int *pMatrixA = new int[DataSize*DataSize];
int *pMatrixB = new int[DataSize*DataSize];
         int *pMatrixC;

   // генерация матриц случайным образом
  for( int i=0; i<DataSize*DataSize; i++)
                {
    pMatrixA[i] = (int) rand()/1000;
    pMatrixB[i] = (int) rand()/1000;
                }

//запуск параллельного ленточного алгоритма
ParallelMultiplyStrip (pMatrixA, pMatrixB, &pMatrixC, DataSize);

                MPI_Finalize();
                return 0;
                }
```

2.3. Обработка графов

Для описания графов известны различные способы задания. Пусть G есть граф

$$G = (V, R),$$

для которого набор вершин v_i, $1 \le i \le n$, задается множеством V, а список дуг графа

$$r_j = (v_{s_j}, v_{t_j}),\ 1 \le j \le k,$$

определяется множеством R. В общем случае дугам графа могут приписываться некоторые числовые характеристики w_j, $1 \le j \le k$ (взвешенный граф).

При малом количестве дуг в графе (т.е. $k << n^2$) целесообразно использовать для определения графов списки, перечисляющие имеющиеся в графах дуги. Представление достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т.е. $k \sim n^2$), может быть эффективно обеспечено при помощи матрицы инцидентности $A=(a_{ij})$, $1 \le i, j \le n$, ненулевые значения элементов которой соответствуют дугам графа

$$a_{ij} = \begin{cases} w(v_i, v_j), & \text{если } (v_i, v_j) \in R, \\ 0, & \text{если } i = j, \\ \infty, & \text{иначе} \end{cases}$$

Использование матрицы инцидентности позволяет применять также при реализации вычислительных процедур для графов матричные алгоритмы обработки данных. Более широко способы представления графов рассмотрены, например, в [5].

2.3.1. Алгоритм Прима

Описание метода

Охватывающим деревом (или *остовом*) неориентированного графа G называется подграф T графа G, который является деревом и содержит все вершины из G. Определив вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, тогда под *минимально охватывающим деревом* (*МОД*) T будем понимать охватывающее дерево минимального веса. Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием наименьшего количества соединительных линий связи.

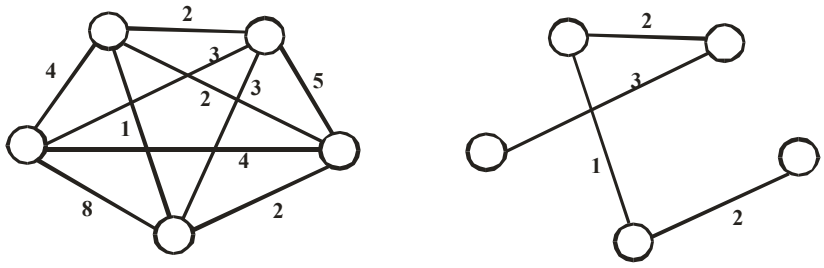

Рис. 8. Ненаправленный взвешенный граф и его минимальное охватывающее дерево

Дадим краткое описание алгоритма решения поставленной задачи, известного под названием *метода Прима* (*Prim*) [4]. Алгоритм начинает работу с произвольной вершины графа, выбираемого в качестве корня дерева, и в ходе последовательно выполняемых итераций расширяет конструируемое дерево до МОД. Пусть V_T есть множество вершин, уже включенных алгоритмом в МОД, а величины d_i, $1 \le i \le n$, характеризуют дуги минимальной длины от вершин, еще не включенных в дерево, до множества V_T, т.е.

$$\forall i \notin V_T \Rightarrow d_i = \min\{w(i,u) : u \in V_T, (i,u) \in R\}$$

(если для какой либо вершины $i \notin V_T$ не существует ни одной дуги в V_T, значение d_i устанавливается в ∞). При начале работы алгоритма выбирается корневая вершина МОД s и полагается

$$V_T = \{s\},\ d_s = 0.$$

Действия, выполняемые на каждой итерации алгоритма Прима, состоят в следующем:

- определяются значения величин d_i для всех вершин, еще не включенные в состав МОД;
- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T
$$t : d_t = \min d_i, i \notin V_T;$$
- включение выбранной вершины t в V_T.

После выполнения $n-1$ итераций метода МОД будет сформировано; вес этого дерева может быть получен при помощи выражения

$$W_T = \sum_{i=1}^{n} d_i .$$

Оценим возможности для параллельного выполнения рассмотренного алгоритма нахождения минимально охватывающего дерева. Итерации метода должны выполняться последовательно и, тем самым, не могут быть распараллелены. С другой стороны, выполняемые на каждой итерации алгоритма действия являются независимыми и могут реализовываться одновременно. Так, например, определение величин d_i может осуществляться для каждой вершины графа в отдельности, нахождение дуги минимального веса может быть реализовано по каскадной схеме и т.д.

Распределение данных между процессорами вычислительной системы должно обеспечивать независимость перечисленных операций алгоритма Прима. В частности, это может быть обеспечено, если каждая вершина графа располагается на процессоре вместе со всей связанной с вершиной информацией. Соблюдение данного принципа приводит к тому, что при равномерной загрузке каждый процессор P_j, $1 \le j \le p$, будет содержать набор вершин

$$V_j = \{v_{i_j+1}, v_{i_j+2}, ..., v_{i_j+k}\},\ i_j = k(j-1),\ k = n/p,$$

соответствующий этому набору блок из k величин d_i, $1 \le i \le n$, и вертикальную полосу матрицы инцидентности графа G из k соседних столбцов, а также общую часть набора V_j и формируемого в процессе вычислений множества вершин V_T.

С учетом такого разделения данных итерация параллельного варианта алгоритма Прима состоит в следующем:

- определяются значения величин d_i для всех вершин, еще не включенных в состав МОД; данные вычисления выполняются независимо на каждом процессоре в отдельности
- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T; для выбора такой вершины необходимо осуществить поиск минимума в наборах величин d_i, имеющихся на каждом из процессоров (количество параллельных операций n/p), и выполнить сборку полученных значений.
- рассылка всем процессорам номера выбранной вершины для включения в охватывающее дерево.

Получение МОД обеспечивается при выполнении $(n-1)$ итерации алгоритма Прима.

Структура метода

Схема вызовов функций в программе, реализующей алгоритм Прима, представлена на рис. 9.

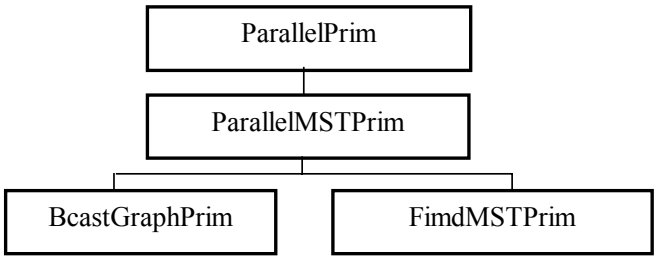

Рис. 9. Схема вызовов функций. Алгоритм Прима

Управляющий процессор (процессор 0) выполняет:

- разбиение матрицы графа на части и их пересылку на соответствующие процессоры (функция *BcastGraphPrim*);
- выбор стартовой вершины случайным образом и рассылка ее всем процессорам (функция *FindMSTPrim*);
- формирование массива *d* - весов ребер с минимальным значением от любой вершины в V_T к вершине *v* и рассылка частей массива по соответствующим процессорам (функция *FindMSTPrim*);
- выполнение итераций параллельного нахождения минимального остовного дерева (функция *FindMSTPrim*):
  - определение значения величин d_i для всех вершин, которые не включены в остовное дерево;
  - выбор вершины графа, которая имеет дугу минимального веса до множества V_T;
  - рассылка выбранной вершины всем процессорам и включение ее во множество V_T;
- формирование минимального остовного дерева (функция *FindMSTPrim*).

Функциональные процессоры выполняют:

- прием от процессора 0 соответствующего блока матрицы (функция *BcastGraphPrim*);
- прием стартовой вершины от процессора 0 (функция *FindMSTPrim*);
- прием части массива *d* от процессора 0 (функция *FindMSTPrim*);
- выполнение итераций параллельного нахождения минимального остовного дерева (функция *FindMSTPrim*):
- определение значения величин d_i для всех вершин, которые не включены в остовное дерево;
- прием новой вершины и включение ее в множество V_T.

Формат вызова

Вызов параллельного алгоритма Прима имеет вид:

```
void ParallelPrim( int *pGraphMatrix, int **pSpanTree, int DataSize)
```

где:

- `int *pGraphMatrix` – взвешенная матрица инцидентности графа,
- `int **pSpanTree` – остовное дерево,
- `int DataSize` – число вершин в графе.

Пример использования

```
#include "paralib.h"
  #include "mpi.h"

int main(int argc, char *argv[])
                {
   MPI_Init ( &argc, &argv );
       int DataSize = 100;
       int *pGraphMatrix;
        int *pSpanTree;

// генерация матрицы графа случайным образом
   InitGraph(&pGraphMatrix, &DataSize);

   //запуск параллельного алгоритма Прима
ParallelPrim(pGraphMatrix, &pSpanTree, DataSize);

              MPI_Finalize();
                 return 0;
                     }
```

2.3.2. Алгоритм Дейкстры

Описание метода

Задача поиска кратчайших путей на графе состоит в нахождении путей минимального веса от некоторой заданной вершины s до всех имеющихся вершин графа. Постановка подобной проблемы имеет важное практическое значение в различных приложениях, когда веса дуг означают время, стоимость, расстояние, затраты и т.п.

Возможный способ решения поставленной задачи, известный как *алгоритм Дейкстры* [4], практически совпадает с методом Прима. Различие состоит лишь в интерпретации и в правиле оценки вспомогательных величин d_i, $1 \le i \le n$. В алгоритме Дейкстры эти величины означают суммарный вес пути от начальной вершины до всех остальных вершин графа. Как результат, после выбора очередной вершины t графа для включения в множество выбранных вершин V_T, значения величин d_i, $1 \le i \le n$, пересчитываются в соответствии с новым правилом:

$$\forall i \notin V_T \Rightarrow d_i = \min\{d_i, d_t + w(t,i)\}.$$

С учетом измененного правила пересчета величин d_i, $1 \le i \le n$, схема распределения данных по процессорам при выполнении алгоритма Дейкстры может быть сформирована по аналогии с параллельным вариантом метода Прима.

С учетом такого разделения данных итерация параллельного варианта алгоритма Дейкстры состоит в следующем:

- определяются значения величин d_i для всех вершин, еще не включенных в состав дерева кратчайших путей; данные вычисления выполняются независимо на каждом процессоре в отдельности;
- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T; для выбора такой вершины необходимо осуществить поиск минимума в наборах величин d_i, имеющихся на каждом из процессоров (количество параллельных операций n/p), и выполнить сборку полученных значений.

- рассылка номера выбранной вершины для включения в дерево кратчайших путей всем процессорам.

Получение дерева кратчайших путей обеспечивается при выполнении $(n-1)$ итерации алгоритма Дейкстры.

Структура метода

Схема вызовов функций в программе, реализующей алгоритм Дейкстры, представлена на рис. 10.

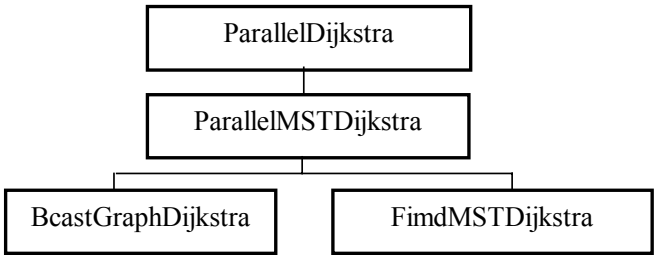

Рис. 10. Схема вызовов функций. Алгоритм Дейкстры

Управляющий процессор (процессор 0) выполняет:

- разбиение матрицы графа на части и их пересылку на соответствующие процессоры (функция *BcastGraphGraphDijkstra*);
- выбор стартовой вершины случайным образом и рассылка ее всем процессорам (функция *FindMSTGraphDijkstra*);
- формирование массива *d* - весов ребер с минимальным значением от любой вершины в V_T к вершине *v* и рассылка частей массива по соответствующим процессорам (функция *FindMSTGraphDijkstra*);
- выполнение итераций параллельного нахождения минимального остовного дерева (функция *FindMSTGraphDijkstra*):
  - определение значения величин d_i для всех вершин, которые не включены в остовное дерево;
  - выбор вершины графа, которая имеет дугу минимального веса до множества V_T;
  - рассылка выбранной вершины всем процессорам и включение ее во множество V_T;
- формирование минимального остовного дерева (функция *FindMSTGraphDijkstra*).

Функциональные процессоры выполняют:

- прием от процессора 0 соответствующего блока матрицы (функция *BcastGraphDijkstra*);
- прием стартовой вершины от процессора 0 (функция *FindMSTGraphDijkstra*);
- прием части массива *d* от процессора 0 (функция *FindMSTGraphDijkstra*);
- выполнение итераций параллельного нахождения минимального остовного дерева (функция *FindMSTGraphDijkstra*):
  - определение значения величин d_i для всех вершин, которые не включены в остовное дерево;
  - прием новой вершины и включение ее во множество V_T.

Формат вызова

Вызов параллельного алгоритма Дейкстры имеет вид:

```
void ParallelDijkstra ( int *pGraphMatrix, int **pSpanTree, int DataSize)
```

где:

- `int *pGraphMatrix` – взвешенная матрица инцидентности графа,
- `int **pSpanTree` – остовное дерево,
- `int DataSize` – число вершин в графе.

Пример использования

```
#include "paralib.h"
  #include "mpi.h"

int main(int argc, char *argv[])
                {
   MPI_Init ( &argc, &argv );
       int DataSize = 100;
       int *pGraphMatrix;
         int *pSpanTree;

// генерация матрицы графа случайным образом
   InitGraph(&pGraphMatrix, &DataSize);

  //запуск параллельного алгоритма Дейкстры
ParallelDijkstra (pGraphMatrix, &pSpanTree, DataSize);

                MPI_Finalize();
                   return 0;
                       }
```

3. Программа демонстрации работоспособности методов

3.1. Постановка задачи

Для выбора задачи необходимо выбрать пункт меню **Задача** и выделить левой клавишей мыши одну из задач: **Сортировка**, **Матричное умножение**, **Обработка графов** (рис. 11). Выбранная задача станет текущей в активном окне.

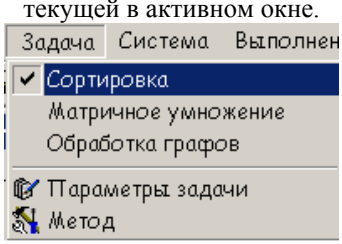

Рис. 11. Выбор задачи

Для выбранной задачи необходимо задать объем исходных данных. Для задачи сортировки – это размер упорядочиваемого массива, для задачи матричного умножения – порядок исходных матриц, для задачи обработки графов – число вершин в графе. Для выбора параметров задачи необходимо выполнить команду **Параметры задачи** пункта меню **Задача**. В появившемся диалоговом окне (рис.12) следует при помощи бегунка задать необходимый объем исходных данных. Нажмите **ОК** (Enter) для подтверждения задания параметра. Для возврата в основное меню без сохранения изменений нажмите **Отмена** (Escape).

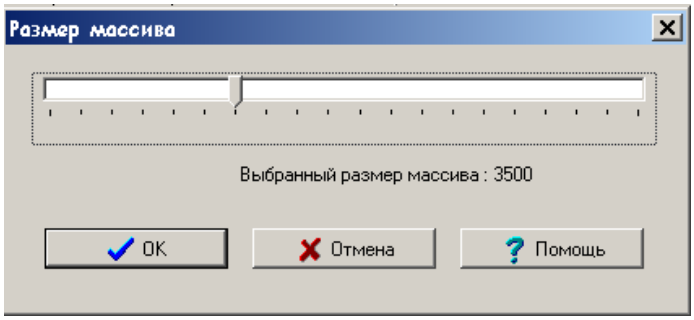

Рис. 12. Диалоговое окно задания параметров задачи в случае решения задачи сортировки

3.2. Выбор метода решения

Для выбора метода решения задачи необходимо выполнить команду **Метод** пункта меню **Задача**. В появившемся диалоговом окне (рис. 13) выделите мышью нужный метод, нажмите **ОК** для подтверждения выбора или нажмите **Отмена** для возврата в основное меню.

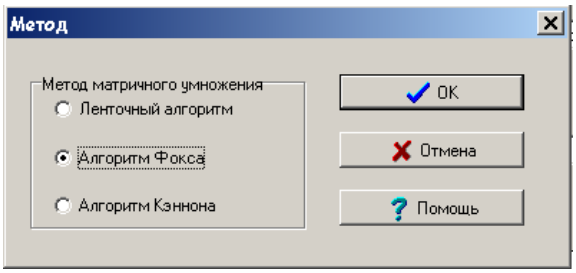

Рис. 13. Диалоговое окно выбора метода в случае решения задачи матричного умножения

3.3. Определение количества процессоров

Для выбора числа процессоров выполните команду **Количество процессоров** пункта меню **Система**. В появившемся диалоговом окне (рис. 14) при помощи бегунка задайте нужное число процессоров. Нажмите **ОК** (Enter) для подтверждения выбора или **Отмена** (Escape) для возврата в основное меню без изменений.

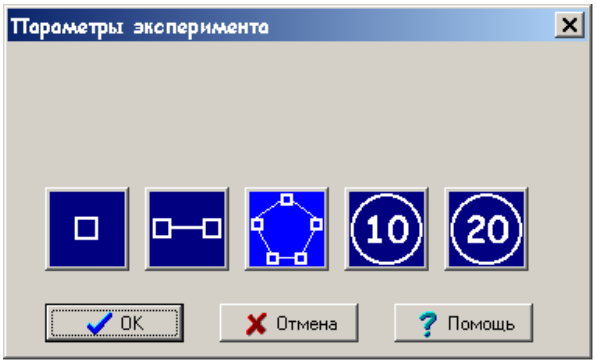

Рис. 14. Окно выбора количества вычислительных узлов для случая пузырьковой сортировки

3.4. Выполнение эксперимента

Для выполнения вычислительного эксперимента в пункте меню **Выполнение** выберите один из режимов выполнения эксперимента: **Локально**, **Кластер** (рис. 15).

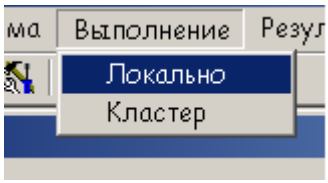

Рис. 15. Выбор режима выполнения

В случае выбора локального режима запуска происходит выполнение эксперимента на компьютере, на котором осуществляется работа (вычисления происходят в режиме разделения времени процессора между всеми процессами параллельной программы). Для оценки объема выполненных вычислений в окне эксперимента отображается ленточный индикатор времени. По окончании в окне эксперимента отображаются общие затраты времени и затраты на передачу данных.

В случае выбора режима **Кластер** появляется диалоговое окно открытия файла (рис. 16), в котором необходимо открыть подготовленный конфигурационный файл для запуска параллельной программы на нескольких компьютерах (правила подготовки конфигурационного файла зависят от типа применяемой на кластере версии библиотеки MPI).

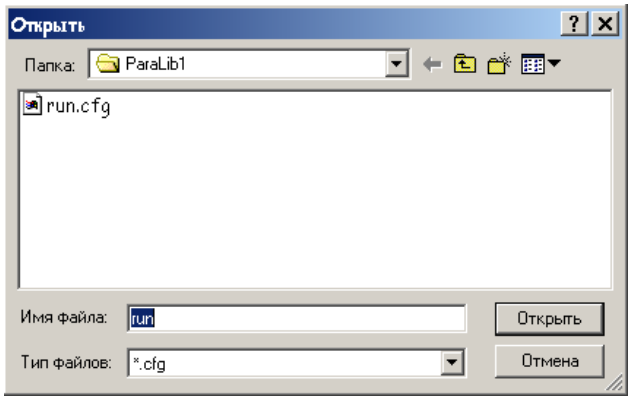

Рис. 16. Диалоговое окно открытия конфигурационного файла

Конфигурационный файл определяет процессоры, используемые при решении задачи, и для применяемой в ННГУ библиотеке MPICH 1.2.5 имеет следующий формат:

```
exe c:\somepath\myapp.exe
[args arg1 arg2 arg3 ...]
[env VAR1=VAL1|...|VARn=VALn]
[dir drive:\some\path]
[map drive:\\host\share]
hosts
hostA #procs [path\myapp.exe]
hostB #procs [\\host\share\somepath\myapp2.exe]
hostC #procs
. . .
```

Поля строк, которые должны приводиться в конфигурационном файле без изменений, выделены в тексте жирным шрифтом; строки, наличие которых необязательно, указаны в квадратных скобках. Назначение приводимых параметров является следующим:

– имя и местоположение запускаемой программы:
exe c:\somepath\myapp.exe

– аргументы командной строки:
[**args** arg1 arg2 arg3 ...]

– переменные окружения:
[**env** VAR1=VAL1|...|VARn=VALn]

– рабочий каталог (если опция не определена, используется текущий каталог):
[**dir** drive:\some\path]

– подключаемый сетевой диск:
[**map** drive:\\host\share]

– узлы (компьютеры) для запуска (если указано имя программы path\myapp.exe, то будет запущена указанная программа, если не указано, то будет запущена программа, заданная в строке **exe**):
hosts
hostA #procs [path\myapp.exe]

. . .

В качестве примера можно привести конфигурационный файл, который использовался для выполнения вычислительных экспериментов с библиотекой ParaLib на кластере ННГУ.

```
exe c:\parallel\bubble.exe
        args 50000
           hosts
        srv114-2-1 1
        srv114-2-2 1
        srv114-2-3 1
        srv114-2-4 1
```

3.5. Анализ результатов

Для отображения итогов экспериментов используется окно, содержащее **Таблицу итогов** и **Лист графиков**.

Каждая строка таблицы итогов представляет один выполненный эксперимент.

Для экспериментов строятся зависимости времени выполнения алгоритма от числа процессоров и объема исходных данных. Для построения требуемой зависимости в таблице итогов выбираются все эксперименты, в которых совпадают неварьируемые параметры со значениями из выделенной строки таблицы, т. е. в выбираемых экспериментах. Для графика зависимости времени от объема исходных данных должны совпадать метод решения, тип запуска и число процессоров; для зависимости времени от числа процессоров совпадающими параметрами должны быть метод решения, тип запуска и объем исходных данных.

Для того, чтобы изменить вид отображаемой зависимости, нужно выбрать соответствующие пункты в списках, расположенных в левом верхнем и нижнем правом углу листа графиков.

Для более детального изучения зависимостей временных характеристик выполнения алгоритма от различных параметров на листе графиков предусмотрена возможность изменения масштаба.

Для демонстрации накопленных результатов экспериментов следует выбрать пункт меню **Результаты**, выделить команду **Итоги** и выполнить одну из двух команд: **Из активного окна** или **Из всех окон**. При выполнении первой команды будут отображены результаты, накопленные в активном окне вычислительного эксперимента. При выполнении второй команды – результаты из всех открытых окон экспериментов.

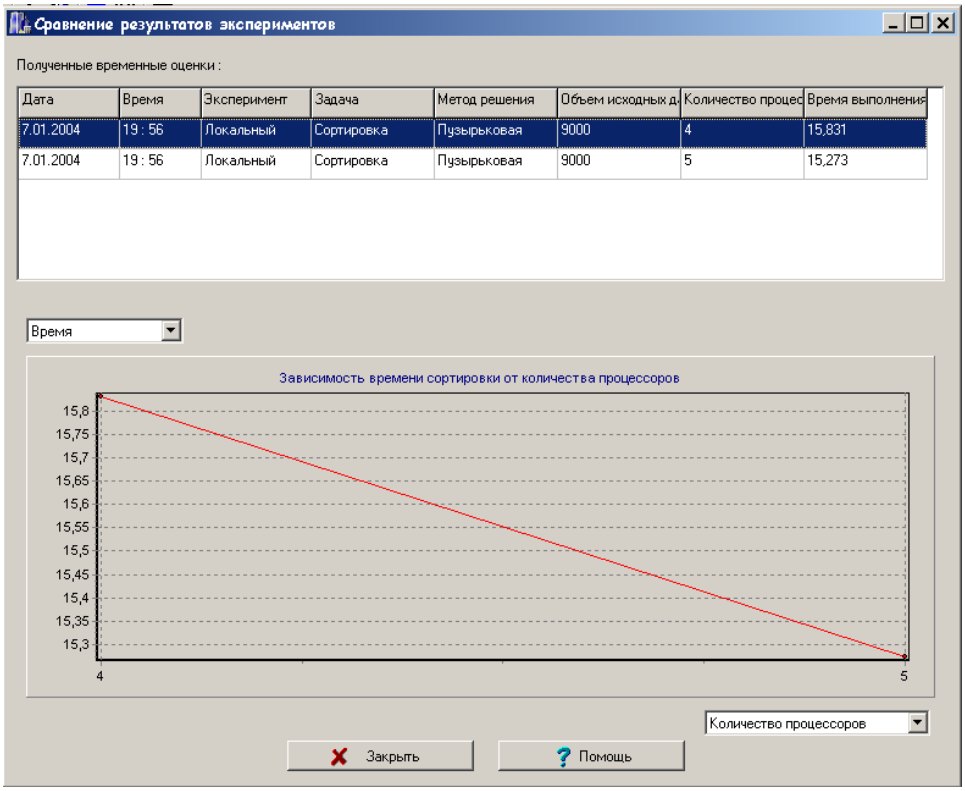

Рис. 17. Окно сравнения результатов экспериментов

4. Результаты экспериментов

Для оценки эффективности параллельного выполнения программ библиотеки приведены результаты вычислительных экспериментов. Выполнение экспериментов проводилось на кластере Нижегородского университета на вычислительных узлах с процессорами Intel Pentium III 1000 МГц, оперативная память 256 Мб, объединенных сетью Gigabit Ethernet 100 Мбит/сек.

Первый эксперимент проводился для задачи сортировки данных при помощи параллельного варианта пузырьковой сортировки (см. раздел 2.1.1) В табл. 1 приведены числовые данные по результатам экспериментов, на рис. 18 полученные данные представлены в графическом виде.

Таблица 1. Результаты работы пузырьковой сортировки

Размер массива	Время последовательного выполнения	2 процессора		4 процессора	
		время, с	ускорение	время, с	ускорение
20000	8,49973	2,11568	4,01747	0,65060	13,06445
30000	19,09485	4,79707	3,98052	1,34420	14,20530
50000	53,12223	13,31622	3,98928	3,34894	15,86237

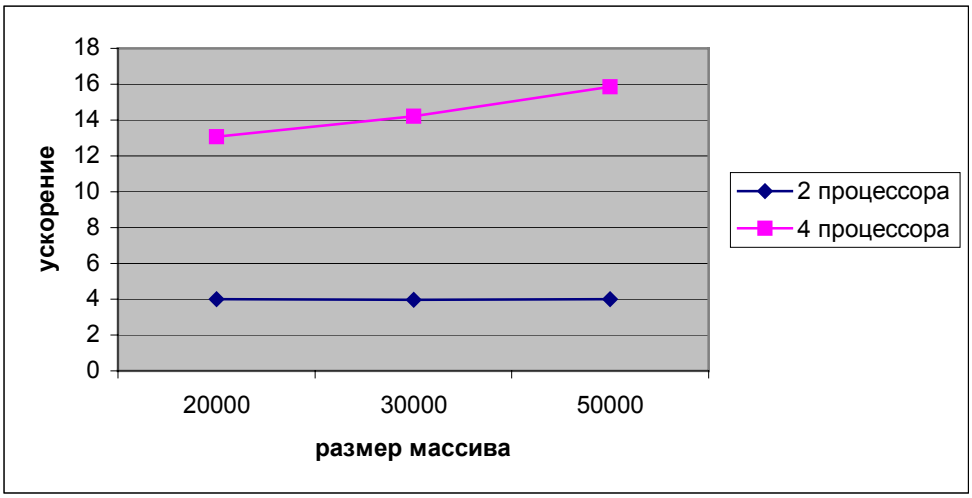

Рис. 18. Результаты работы пузырьковой сортировки

Второй эксперимент проводился для задачи умножения матриц при помощи параллельного варианта ленточного умножения матриц (см. раздел 2.2.3) В табл. 2 приведены числовые данные по результатам экспериментов, на рис. 19 полученные данные представлены в графическом виде.

Таблица 2. Результаты работы ленточного умножения матриц

Порядок матрицы	Время последовательного выполнения	2 процессора		4 процессора	
		время, с	ускорение	время, с	ускорение
100	0,03265	0,01808	1,80555	0,03117	1,04757
200	0,26086	0,15094	1,72826	0,11041	2,36258
500	6,38843	3,23809	1,97290	1,49766	4,26560

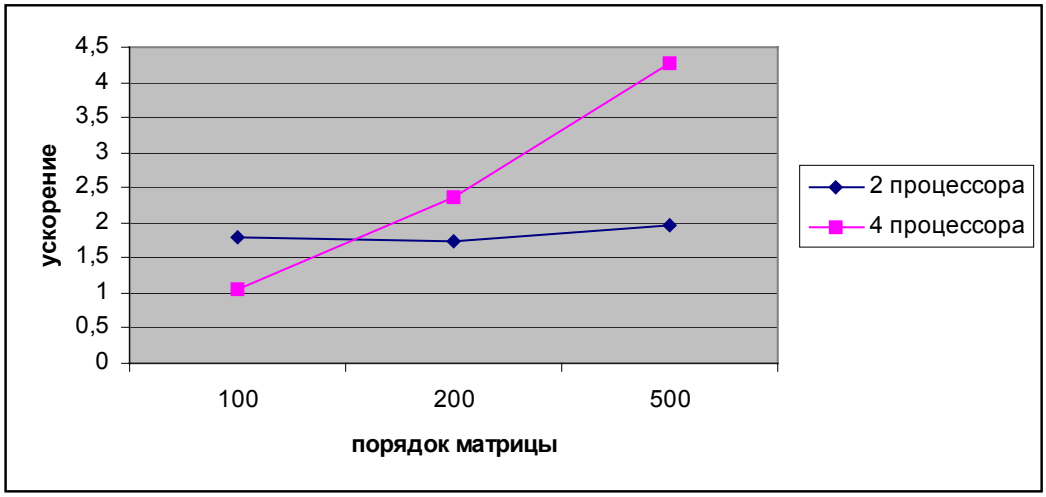

Рис. 19. Результаты работы ленточного умножения матриц

Третий эксперимент проводился для задачи поиска кратчайших путей при помощи параллельного варианта алгоритма Дейкстры (см. раздел 2.3.2) В табл. 3 приведены числовые данные по результатам экспериментов, на рис. 20 полученные данные представлены в графическом виде.

Таблица 3. Результаты работы алгоритма Дейкстры

Порядок матрицы	Время последовательного выполнения	2 процессора		4 процессора	
		время, с	ускорение	время, с	ускорение
50	0,11019	0,02218	4,96619	0,13563	0,81240
100	0,07513	0,07239	1,03779	0,29262	0,25676
500	9,86200	5,20530	1,89460	3,76575	2,61886

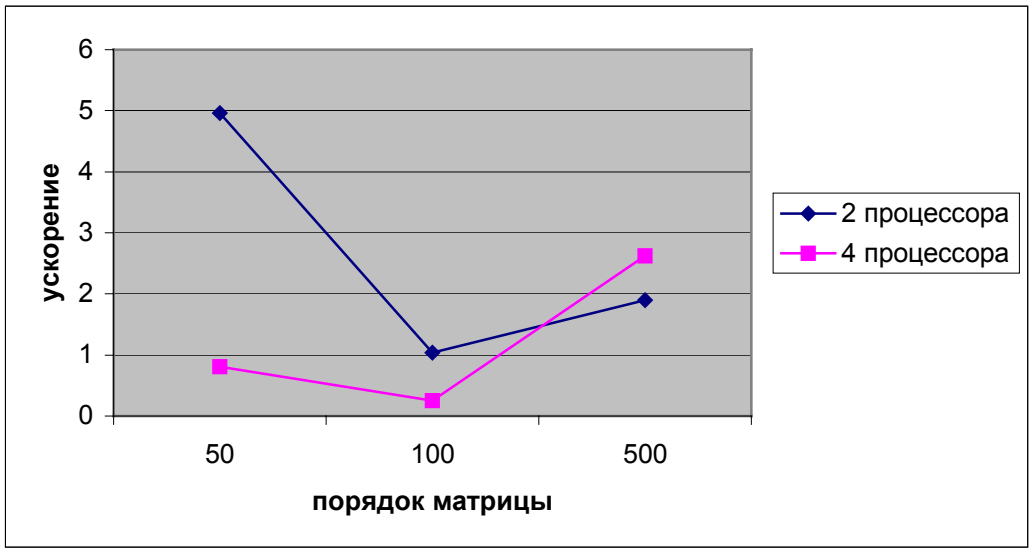

Рис. 20. Результаты работы алгоритма Дейкстры

Литература

1. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. - Н.Новгород, ННГУ, 2001.
2. Гергель В.П., Лабутина А.А. Паралаб. Программная система для изучения и исследования методов параллельных вычислений. Учебное пособие – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003
3. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем — СПб.: БХВ-Петербург, 2002.
4. Кнут Д. Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. - М.: Мир, 1981.
5. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. - М.: МЦНТО, 1999.
6. Kumar V., Grama A., Gupta A., Karypis G. Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc., 1994

Информационные ресурсы сети Интернет

7. Информационно-аналитические материалы по параллельным вычислениям (http://www.parallel.ru)
8. Информационные материалы Центра компьютерного моделирования Нижегородского университета (http://www.software.unn.ac.ru/ccam)
9. Introduction to Parallel Computing (Teaching Course) (http://www.ece.nwu.edu/~choudhar/C58/)
10. PLAPACK (http://www.cs.utexas.edu/users/plapack/)