Министерство образования и науки Российской федерации
Севастопольский Государственный университет

ПАРАЛЛЕЛЬНОЕ РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ MPI

Методические указания
к лабораторным работам
по дисциплине
«Теория распределенных систем и параллельных вычислений»
для студентов дневной и заочной форм обучения
направления 09.03.02 «Информационные системы и технологии»

Севастополь
2019

УДК 681.06+658.5

Параллельное распределенное программирование с использованием технологии MPI. Методические указания к лабораторным работам по дисциплине «Теория распределенных систем и параллельных вычислений» для студентов дневной и заочной форм обучения направления 09.03.02 «Информационные системы и технологии». / сост. А.Ю. Дрозин, К.В. Кротов.– Севастополь: Изд-во СГУ, 2019. – 116 с.

.

Методические указания предназначены для проведения лабораторных занятий по дисциплине «Теория распределенных систем и параллельных вычислений». Целью настоящих методических указаний является обучение студентов практическим навыкам разработки программ, использующих принципы параллельных распределенных вычислений.

Методические указания составлены в соответствии с требованиями программы дисциплины «Теория распределенных систем и параллельных вычислений» для студентов направления 09.03.02 «Информационные системы и технологии» дневной формы обучения.

Методические указания рассмотрены и утверждены на заседании кафедры Информационных систем. Протокол №1 от 25 августа 2014г.

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Рецензент: Фисун С.Н. канд. техн. наук, доц. каф. Информационных технологий и вычислительной техники.

СОДЕРЖАНИЕ

ОБЩИЕ ТРЕБОВАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

1. ЦЕЛЬ И ЗАДАЧИ ЛАБОРАТОРНЫХ РАБОТ

Цель настоящих лабораторных работ состоит в исследовании средств библиотеки MessagePassingInterface (MPI) для реализации распределенных параллельных вычислений и основных алгоритмов, обеспечивающих взаимодействие параллельных распределенных процессов. Задачами выполнения лабораторных работ являются: 1) углубленное изучение основных теоретических положений дисциплины; 2) получение практических навыков написания программ, реализующих параллельные распределенные приложения.

2. ОПИСАНИЕ ЛАБОРАТОРНОЙ УСТАНОВКИ

Объектом исследования в лабораторных работах являются методы и алгоритмы организации выполнения параллельных распределено выполняющихся приложений, реализации взаимодействия параллельных распределенных приложений. Инструментом исследования методов и алгоритмов реализации взаимодействия параллельных распределенных приложений является ЭВМ. Программным средством исследования является библиотека MPI, подключаемая к программным модулям, создаваемым в среде Visual studio, функции которой позволяют реализовывать выполнение параллельных распределенныъ приложений. Описание функций библиотеки для реализации алгоритмов организации выполнения параллельных распределено выполняющихся приложений и реализации взаимодействия между ними приведено ниже в лабораторных работах. Выполнение работы предусматривает создание проекта в среде Visual studio, разработку и отладку программы на языке C++ с использованием вызовов требуемых функций библиотеки MPI, запуск программы на выполнение в несколько потоков (требуемое количество копий исходного кода программы), получение результатов работы программы в виде протоколов сообщений (последовательности сообщений от выполняемых программ), комментирующих параллельное выполнение процессов и их взаимодействие в ходе выполнения, получение результатов вычислений.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчеты по лабораторной работе оформляются каждым студентом индивидуально. Отчет должен включать: название лабораторной работы; цель работы; краткие теоретические сведения; постановку задачи; текст программы, реализующей задание; распечатку результатов выполнения программы и протоколов взаимодействия параллельно выполняющихся процессов.

4.ЗАДАНИЕ НА РАБОТУ

Задание выбирается в соответствии с вариантом, назначаемым преподавателем.

1. ОБЩИЕ СВЕДЕНИЯ О БИБЛИОТЕКЕ MPI И ЕЕ ЦЕЛИ

Библиотека MessagePassingInterface (MPI) является средством, позволяющим выполнять создание параллельных распределенных приложений, и является стандартом при реализации распределено выполняющихся программ. Под MPI подразумевается модель реализации задач, взаимодействующих посредством параллельной передачи сообщений, в которой через совместные операции над каждым из процессов данные перемещаются из адресного пространства одного процесса в адресное пространство другого. MPI не является языком программирования, все операции MPI выражаются в виде функций, подпрограмм, или методов с соответствующими привязками к языку (C/C++, Fortran-77, 95).

Рисунок. 1.1 –Высокоуровневое представление MPI

Главное преимущество модели передачи сообщений, реализуемой в MPI, является его переносимость, легкость использования, а также возможность реализации как в вычислительных системах с общей памятью, так и в вычислительных системах с распределенной памятью. Необходимость использования библиотеки MPI и модели взаимодействия процессов посредством передачи сообщений является очевидной в вычислительной системе с распределенной памятью, в которой взаимодействие хостов (рабочих станций) реализуется посредством коммуникационной среды.

Реализация параллелизма в MPI

MPI-программа состоит из независимых процессов, выполняющих свой собственный код, т.е. реализует MIMD (**Multiple Instruction Multiple Data**) подход к организации параллельных программ. При реализации MPI-программ операционная система для каждого процесса размещает копию выполняемой программы в оперативной памяти отдельной компоненты (хоста, рабочей станции в сети) вычислительной системы. Таким образом, независимо от конфигурации системы (Рис. 1.2–1. 4) код программы располагается на всех процессах. Для запуска требуемого количества копий программы необходимо в исполняемой среде указать необходимое количество процессов (запускаемых копий программы) (Приложение А).

Каждый процесс начинает выполнять свою собственную копию кода. Различные процессы могут выполнять различные участки кода посредством ветвления внутри программы, т.е. коды, выполняемые процессом, не обязательно должны быть идентичными (как правило, они являются различными).

Для определения участков кода в программе, которые должны выполнять процессы (участка кода, который должен выполнять каждый процесс) используются ранги процессов, но могут быть созданы и более сложные структуры распределения заданий для каждого процесса.

Ветвление для процессов на основе рангов рассмотрено на Рис. 1.5.

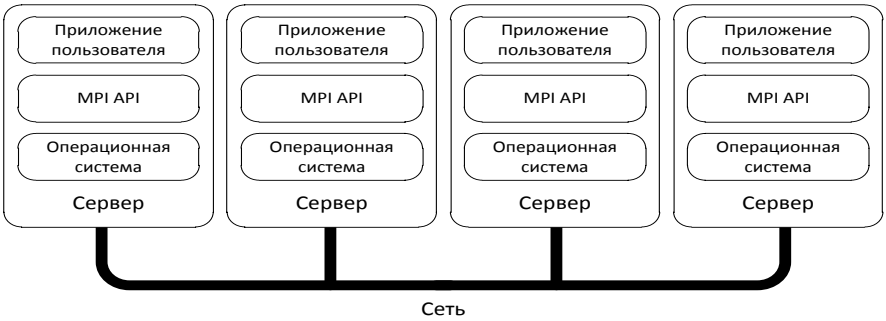


Рисунок 1.2 – Вычислительная система с четырьмя рабочими станциями

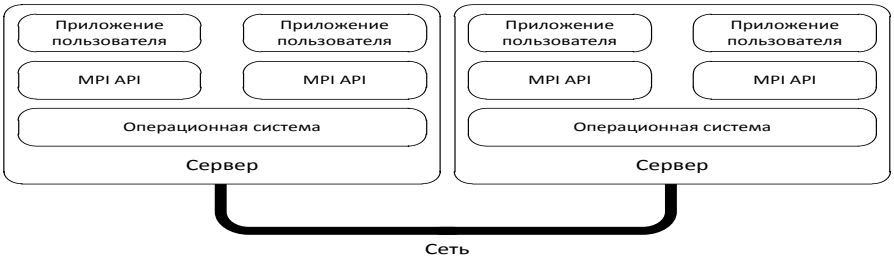


Рисунок 1.3 – Вычислительная система с двумя рабочими станциями

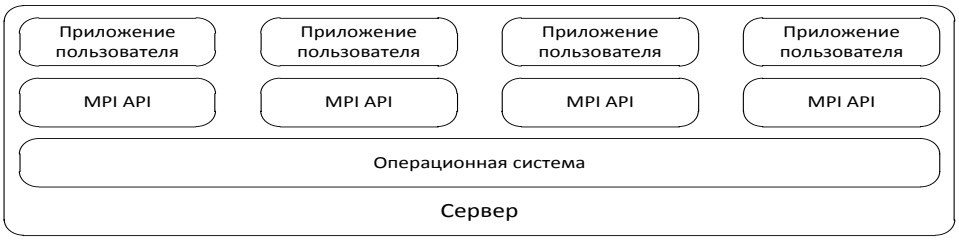


Рисунок 1.4 – Вычислительная система с одной рабочей станцией

Таким образом, каждый процесс исполняется в собственном адресном пространстве. Процессы взаимодействуют посредством вызовов функций библиотеки MPI, обеспечивающих передачу сообщений. Одно из преимуществ MPI –это достижение переносимости исходного кода. Это значит, что программа, использующая MPI и подчиняющаяся существующим стандартам языка, при написании уже является переносимой, т.е. не требуется вносить

какие-либо изменения в код при переносе программы с одной системы на другую.

Любая реализация MPI-программ требует некоторые начальные установки, перед тем как какие-либо другие MPI-подпрограммы могли бы быть вызванными. Для этого в программе на C++ должен быть выполнен вызов следующей MPI-функции (функции инициализации):
intMPI_Init(int * argc, char *argv);
in **argc** - указатель на счетчик аргументов командной строки; in **argv** - указатель на список аргументов;

Любая MPI программа должна содержать только один вызов инициализирующей подпрограммы: **MPI_Init** или **MPI_Init_thread** (для многопотоковых процессов).

Также каждый процесс должен вызывать функцию завершения после любого использования MPI-функций:
intMPI_Finalize(void)

Эта подпрограмма очищает всю MPI-систему.

Таким образом, структура любой MPI-программы следующая:

```
int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
   /* анализ аргументов */
   /* основная часть */
   MPI_Finalize();
}
```

Для того чтобы получить количество выполняемых копий процесса, используется следующая функция:
int MPI_Comm_size(MPI_Comm comm, int *size) **Атрибутами функции являются**:
in **comm** –коммуникатор; out **size** - количество процессов в группе comm

Если для аргумента **comm** указать константу **MPI_COMM_WORLD**, функция вернет количество всех доступных процессов программы. Аналогичным образом, при указании данной константы в качестве значения аргумента **comm** в рассматриваемых функциях действия выполняются со всеми процессами программы.

Ранг (идентификатор) копии программы (ранг процесса) в отдельной группе коммуникатора можно определить с помощью соответствующей функции: **int MPI_Comm_rank(MPI_Comm comm, int *rank)ее атрибутами функции** являются**:**
in **comm** –коммуникатор; out **rank** - ранг вызывающего процесса в группе **comm**

При получении от того же коммуникатора значение **size**, значение ранга будет лежать в диапазоне от **0** до **size-1**.

На рисунке 1.5 рассмотрен пример MPI-программы для трех процессов. В соответствии с рангом, каждый процесс выполняет определенный только для него код (серым цветом выделены участки кода, которые процесс не выполняет). Перед завершением каждый процесс совершает вызов функции **MPI_Barrier()**, которая приостанавливает выполнение процесса до тех пор, пока все процессы не совершат вызов данной функции (выполняется взаимная синхронизация выполняющихся процессов).

Процесс 0

```
#include "stdafx.h"
#include <iostream>
#include "mpi.h"

using namespace std;

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);

    if(rank == 0)
    {
        cout << "I am" << endl;
    }
    if(rank == 1)
    {
        cout << "MPI" << endl;
    }
    if(rank == 2)
    {
        cout << "program" << endl;
    }

    if(rank == 0)
    {
        cout << "Press any key..."
             << endl;
        getchar();
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();

    return 0;
}
```

Процесс 1

```
#include "stdafx.h"
#include <iostream>
#include "mpi.h"

using namespace std;

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);

    if(rank == 0)
    {
        cout << "I am" << endl;
    }
    if(rank == 1)
    {
        cout << "MPI" << endl;
    }
    if(rank == 2)
    {
        cout << "program" << endl;
    }

    if(rank == 0)
    {
        cout << "Press any key..."
             << endl;
        getchar();
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();

    return 0;
}
```

Процесс 2

```
#include "stdafx.h"
#include <iostream>
#include "mpi.h"

using namespace std;

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);

    if(rank == 0)
    {
        cout << "I am" << endl;
    }
    if(rank == 1)
    {
        cout << "MPI" << endl;
    }
    if(rank == 2)
    {
        cout << "program" << endl;
    }

    if(rank == 0)
    {
        cout << "Press any key..."
             << endl;
        getchar();
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();

    return 0;
}
```

Рисунок 1.5 – Пример ветвления при организации выполнения процессов

В приложениях А и Б приводится описание установки и настройки MPI, а также основные соглашения при программировании и дополнительные функции.

ЛАБОРАТОРНАЯ РАБОТА №1
ИСЛЕДОВАНИЕ СРЕДСТВ СОЗДАНИЯ РАСПРЕДЕЛЕННО ВЫПОЛНЯЮЩИХСЯ ПОГРАММ

ЦЕЛЬ РАБОТЫ: исследовать функции библиотеки MPI, необходимые для создания и взаимодействия распределено выполняемых программ.

1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Тип передачи данных «точка-точка»

Базовым механизмом связи между процессами в MPI является передача типа «точка-точка», в которой реализуются операции отправки (**send)** и приема (**receive**) сообщений (рис. 1.1).

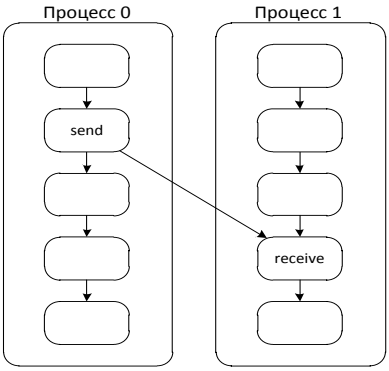


Рисунок 1.1 – Взаимодействие процкссов типа «точка-точка»

Одной из функций, реализующей отправку сообщений, является **MPI_Send()**. Она представляет собой операцию с блокировкой, ее завершение происходит после совершения передачи, синтаксис функции следующий:
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,int tag, MPI_Comm comm);

Атрибутами функции MPI_Send() **являются: in** buf**– указатель на буфер; in** Count**– количество элементов в буфере; in** Datatype **- тип данных буфера; in** Dest **- ранг пункта назначения (процесса приемника); in** Tag **- метка сообщения; in** Comm **– коммуникатор;**

Буфер отправки, определяемый операцией **MPI_Send**, состоит из некоторой последовательности элементов, количество которых указывается в аргументе **count**. Тип данных элементов передается в аргумент **datatype**. Указатель на последовательность данных содержится в переменной **buf**. Необходимо заметить, что длина сообщения указывается количеством элементов, а не количеством байтов, так как реализуемый код не зависит от машины, на которой он будет исполняться. Аргумент **count** может быть равен нулю, в таком случае информационная часть сообщения является пустой. Ос-

новные типы данных, которые могут быть переданы в сообщении, соответствуют основным типам данных языка С (табл. 1.1).

Таблица 1.1

Связь типов данных в MPI

Типы данных MPI	Типы данных С
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (синоним)	signed long long int
MPI_SIGNED_CHAR	signedchar
MPI_UNSIGNED_CHAR	unsignedchar
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex

Аргумент **comm** является коммуникатором, определяющим среду, в которой выполняется связь между процессами. Каждая такая среда обеспечивает некую отделенную «коммуникационную сферу» (**communication universe**), в которой осуществляется прием сообщений. Коммуникатор также определяет ряд процессов, которые делят между собой данную сферу. Эта **группа процессов** упорядочена, и процессы в ней идентифицируются по их рангу. Поэтому диапазон значений для аргумента **dest** может принимать значения в диапазоне от **0** до **n-1**, где **n** это число процессов в группе.

Для того чтобы принять отправленное сообщение необходимо использовать функцию приема MPI_Recv(). Синтаксис функции блокирующего приема следующий:
Int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,int tag, MPI_Comm comm, MPI_Status *status);
Атрибутами функции являются: out **Buf** - указатель на буфер; in **Count** - Количество элементов в буфере (неотрицательное значение); in **Datatype** - Тип данных буфера; in **Source** - Ранг пункта отправки; in **Tag** - Метка сообщения; in **Comm** – Коммуникатор; out **Status** - Статус передачи;

Длина принятого сообщения должна быть меньше или равна длине буфера приема (чтобы избежать связанных с этим ошибок, необходимо использовать функцию **MPI_Probe**, которая позволяет принимать сообщения с неизвестной длинной, ее синтаксис будет описан далее). В аргумент **source** можно передать так называемое групповое значение **MPI_ANY_SOURCE**, в этом случае функция примет сообщение от любого отправителя. Также можно в аргумент **tag** передать групповое значение **MPI_ANY_TAG**, указывающее, что функция примет сообщение с любой меткой. Для аргумента **comm** нет такого значения. Необходимо заметить асимметрию между операциями отправки и получения: операция получения может принимать сообщения от произвольного отправителя, с другой стороны, отправитель должен указать однозначного получателя. Это согласовано с механизмом “push” коммуникации, где передача данных выполняется отправителем (в отличие от механизма “pull”, где передача выполняется получателем).

Если в операции приема использовать групповые значения, то источник и ярлык (Tag) принятого сообщения в таком случае неизвестны. Также, если выполнять многократные запросы одной MPI-функцией, может понадобиться индивидуальный код ошибки для каждого запроса. Данная информация возвращается в аргументе **status**. Переменная статуса (**MPI_Status**) должна быть создана явно пользователем, так как это не системный объект. Данная переменная является структурой, которая содержит три поля (также могут быть и дополнительные поля). Таким образом, источник, ярлык и код ошибки соответствующие принятому сообщению содержаться в следующих полях: **status.MPI_SOURCE**, **status.MPI_TAG**, **status.MPI_ERROR**.

Аргумент **status** также возвращает информацию о длине принятого сообщения, но эта информация не является доступной напрямую как компонента структуры. Для «декодирования» данной информации необходим вызов следующей функции:
Int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count) **Атрибутами этой функции являются**:
in **status** - статус операции приема; in **Datatype** - тип данных в буфере; out **count** - количество элементов в буфере;

С помощью данной функции можно получать сообщения, количество элементов в которых первоначально неизвестно. Функцию необходимо ис-

пользовать после вызова **MPI_Probe**, получив информацией о типе данных, можно использовать вызов **MPI_Recv** с таким же типом для получения требуемого сообщения.

Во многих случаях приложения конструируются так, что для них нет необходимости проверять аргумент **status**. Тогда для пользователя не необходимости создавать переменную статуса, а для MPI заполнять поля этого объекта. Для этого существует константа **MPI_STATUS_IGNORE** (**MPI_STATUSES_IGNORE** для функций с массивами), которая при передаче аргумента **status** информирует систему о том, что данный аргумент не следует заполнять.

Все операции отправки и приема изложенные далее, используют аргументы **buf**, **count**, **datatype**, **source**, **dest**, **tag**, **comm** и **status** таким же образом, как и блокирующие операции **MPI_Send** и **MPI_Recv**. Использование описанных функций проиллюстрировано в следующем примере:

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include "mpi.h"
using namespace std;
int main(int argc, char* argv[])
{
   char message[20];
   int rank;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0)   /* code for process zero */
   {   strcpy(message, "Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1,99,
            MPI_COMM_WORLD);
   }
   else
   {  if (rank == 1)           /* code for process one */
      {   MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
&status);
         cout << "Received message: "'<< message << "'" << endl;
         cin.get();
      }
   }
   MPI_Finalize();
   return 0;
}
```

В этом примере нулевой процесс (**rank=0**) посылает сообщение первому процессу, используя операцию **MPI_Send**. Операция определяет буфер отправки в памяти процесса-отправителя, из которой берутся данные сообщения. Из примера видно, что буфер отправки – переменная **message** памяти нулевого процесса. Местоположение, размер и тип буфера отправки указаны первыми тремя параметрами **send** операции. Операция отправки ассоциирует «конверт» (**envelope**) с сообщением (формирует некоторую адресную информацию для сообщения). Этот конверт указывает пункт назначения сообщения и содержит характерную информацию, которая может быть использована операцией получения, чтобы выбрать именно это сообщение. Последние три параметра операции отправки, вместе с рангом отправителя определяют конверт для отправляемого сообщения. Первый процесс (**rank=1**) получает это сообщение с помощью **receive-**операции **MPI_Recv**. Получаемое сообщение выбирается согласно параметрам его «конверта», данные сообщения сохраняются в буфере приема. В примере буфер состоит из строковой переменной в памяти первого процесса. Первые три параметра операции приема указывают местоположение, размер и тип буфера приема. Следующие три параметра используются для выбора входящего сообщения. Последний параметр используется для того, чтобы получить недостающую информацию о только что принятом сообщении.

1.2. Режимы связи

Изложенная выше процедура отправки является блокирующей: выход из функции не происходит, пока сообщение не будет безопасно сохранено на приемной стороне, тогда отправитель свободно может изменять буфер отправки. Сообщение может быть напрямую скопировано в буфер приема или оно может быть скопировано во временную системную память. Отправка с блокировкой завершиться как только сообщение было буферизировано, даже если не была вызвана функция подходящего приема получателем.

С другой стороны буферизация сообщения может быть затратной, так как влечет за собой дополнительное копирование (**memory-to-memory**), что требует выделение памяти для буферизации. MPI предоставляет выбор нескольких коммуникационных режимов, которые позволяют контролировать способ передачи.

Вызов отправки, описанный выше, использует стандартный (**standard**) режим связи. В этом режиме MPI решает, будет ли исходящее сообщение буферизировано. Стандартный режим является **non-local**: успешное завершение операции отправки зависит от событий связанных с соответствующим получением.

Существует три дополнительных режима связи: буферизированный, синхронизированный, по готовности. Синтаксис первого из них следующий:
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)

Операция отправки в буферизированном (**buffered**) режиме может быть начата независимо от того была ли создана соответствующая функция приема на стороне получателя. Она может завершиться до того, как на стороне получателя процесс достиг точки выполнения функции приема. В отличие от стандартной отправки эта операция является локальной (**local)**, ее завершение не зависит от событий соответствующего приема. Поэтому если отправка была выполнена, и не была реализована функция приема, MPI должен буферизировать исходящее сообщение для того, чтобы завершить операцию отправки. Ошибка может произойти, только если недостаточно места для буферизации. Количество доступного места для буферизации контролируется пользователем.

Синтаксис функции, обеспечивающей передачу данных во втором режиме, следующий:

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

Отправка, которая использует синхронизированный (**synchronous**) режим может начаться в независимости от того был ли реализован вызов функция приема, но она завершиться успешно только если соответствующая функция приема создана и уже вызвана для получения сообщения синхронной отправки. Поэтому завершение синхронной отправки не только указывает, что буфер отправки может быть снова использован, но также указывает, что приемник достиг перекрестной точки выполнения, другими словами начал выполнять соответствующий прием. Если обе операции отправки и приема являются блокирующими операциями, тогда использование синхронного режима обеспечивает семантику синхронной коммуникации: связь не прекратится, пока оба процесса не «встретятся» во время коммуникации. Отправка, выполняемая в этом режиме, является **non-local** (зависит от взаимодействия процессов). Синтаксис функции, обеспечивающей передачу данных в третьем режиме, следующий:

Int MPI_Rsend(void* buf, intcount, MPI_Datatype datatype, intdest, int tag, MPI_Comm comm);

Отправка, которая использует режим связи по готовности(**ready**), может начаться, только если уже вызвана функция соответствующего приема. В противном случае операция является ошибочной, и ее результат является неопределенным. На некоторых системах это позволяет убрать подтверждение установления связи, что в свою очередь приводит к улучшению производительности. Завершение операции отправки не зависит от статуса соответствующего приема и только указывает, что буфер отправки может быть снова использован. Операция отправки, которая использует режим готовности, имеет туже семантику, что и стандартная отправка или отправка с синхронизацией, только отправитель при этом обеспечивает дополнительной информацией систему о том, что функция соответствующего приема уже создана. Рассмотренная операция приема подходит для любого режима отправки. Эта

операция приема является блокирующей, потому что завершение функции происходит только после того как буфер приема будет содержать новое принятое сообщение. Прием может быть завершен до завершения выполнения функции соответствующей отправки. Различные виды буферизации, представлены на Рис.1.2.

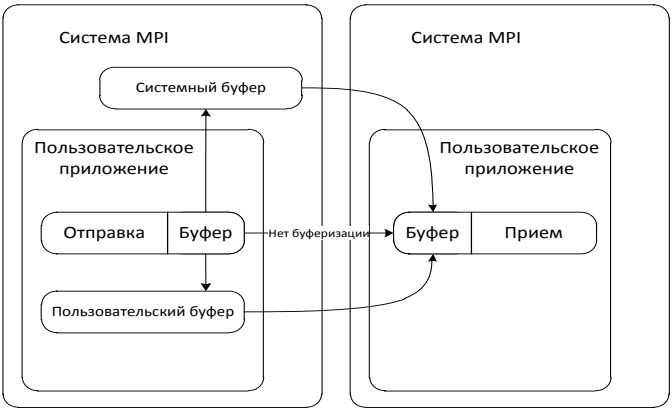


Рисунок 1.2 – Использование буферов при реализации обмена

Рассматриваемая реализация MPI обеспечивает следующие особенности обмена типа «точка-точка»:

1. **Порядок**. Сообщения «не обгоняют» друг друга(**non-overtaking**). Если отправитель отсылает два сообщения последовательно в один и тот же пункт назначения, оба сообщения имеют одинаковые параметры (могут подходить для одной и той же функции приема), то операция приема не может принять второе сообщение, пока не будет принято первое. Так как процесс имеет только один поток выполнения, любые две коммуникации, выполняющиеся этим процессом, упорядочены.

2. **Прогресс.** Если пара соответствующих операций отправки и приема были инициализированы на двух процессах, тогда хотя бы одна из этих двух операций должна завершиться:

- операция отправки будет окончена, если только соответствующая функция приема не получила другое подходящее сообщение и не завершилась;
- операция приема будет завершена, если только отправленное сообщение не было принято другой подходящей функцией приема, которая была создана в том же пункте назначения.

3. **Справедливость.** MPI не гарантирует справедливость в обработке коммуникаций между процессами. Если отправителем была вызвана функция отправки, существует возможность, что в пункте назначения процесс, неоднократно вызывающий функцию приема, подходящую под параметры сообщения, никогда не получит это сообщение, потому что каждый раз будет полу-

чать другое сообщение с такими же параметрами, но отправленное другим адресатом.

4. **Ограниченность ресурсов.** Любая незавершенная передача использует ресурсы системы, которые являются ограниченными. Из-за недостатка ресурсов выполнение MPI-функций может привести к ошибке. Программа считается «безопасной», если для завершения программы не требуется ни одна буферизация сообщения.

1.3. Размещение буфера

В буферизированном режиме пользователь может определить буфер, который будет использоваться для отправляемых сообщений. Чтобы обеспечить MPI видимость буфера, используемого для буферизации исходящих сообщений, реализуется вызов функции со следующим синтаксисом:

int MPI_Buffer_attach(void* buffer, int size). **Атрибутами этой функции являются: in** buffer **- указатель на буфер; in** size **- размер буфера в байтах (неотрицательное число);**

Буфер используется только для отправляемых сообщений в буферизированном режиме. Однократно только один буфер может быть выделен процессу. Для того чтобы отсоединить буфер, ассоциированный с MPI-программой для передачи данных в буферизированном режиме, необходимо использовать функцию, синтаксис которой следующий:

int MPI_Buffer_detach(void* buffer_addr, int* size)**. Атрибутами этой функции являются: out** buffer_addr **- указатель на буфер; out** size **- размер буфера в байтах (неотрицательное число).**

Операция будет заблокирована, пока все сообщения из буфера не будут переданы. В зависимости от возвращаемых значений пользователь может снова использовать или освободить место, занимаемое буфером.

1.4. Связь без блокировки

При реализации программ можно повысить производительность путем использования **неблокирующей коммуникации**. Вызов неблокирующего старта отправки (**sendstart**) инициирует операцию отправки, но не завершает ее. Возврат из данного вызова может осуществиться до того, как сообщение было скопировано из буфера отправки. Для завершения коммуникации необходим отдельный вызов операции завершения отправки (**sendcomplete**), т.е. подтверждение того, что данные были скопированы из буфера отправки. Таким образом, перенос данных из памяти отправки может продолжаться одновременно с совершаемыми вычислениями в отправителе, после того как была инициализирована отправка и до того как она окончилась.

Подобным образом возврат из вызова не блокирующего старта приема (**receivestartcall**) может осуществиться до того, как сообщение было сохранено в буфере приема. Для того чтобы завершить данную операцию и подтвердить, что данные были сохранены в буфере приема, необходим отдельный вызов завершения приема (**receivecomplete**). Перенос данных в память

приемника может продолжаться одновременно с совершаемыми вычислениями, после того как прием был инициирован, и перед тем как он завершился. Использование неблокирующих приемов может также позволить избежать системной буферизации и копирования из памяти в память. Не блокирующий вызов старта отправки использует те же четыре режима, как и отправка с блокировкой: **standard**, **buffered**, **synchronous** и **ready.** Не блокирующая коммуникация использует скрытый объект запроса (**request**) для идентификации коммуникационных операций и сопоставления операций, которые инициирует передачу с операциями, завершающими ее. Доступ к этому объекту можно получить через дескриптор. Объект запроса определяет различные свойства коммуникационных операций, такие как режим отправки, связанный с ним коммуникационный буфер, его контекст, ярлык и аргументы пункта назначения, используемые для отправки, или ярлык и аргументы пункта отправки, используемые для приема. Также этот объект хранит информацию о статусе незавершённых коммуникационных операций. Соглашения по именованию используются такие же, как и для блокирующих операций, но с добавлением префикса **I** (**immediate**), который говорит о том, что операция является не блокирующей. Для того чтобы начать не блокирующую отправку в стандартном режиме используется функция, синтаксис которой следующий:

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

Атрибутами этой функции являются: in buf **- указатель на буфер; in** count **- количество элементов в буфере; in** datatype **- тип данных буфера; in** dest **- ранг пункта назначения; in** tag **- метка сообщения; in** comm **– коммуникатор; out** request **- коммуникационный запрос.**

Название функций, использующие другие режимы коммуникации, следующие: **MPI_Ibsend**; **MPI_Issend**; **MPI_Irsend**.

Для того чтобы начать неблокирующий прием, используется функция, синтаксис которой приведен ниже:

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);

Атрибутами этой функции являются: out buf **- указатель на буфер; in** count **- количество элементов в буфере; in** datatype **- тип данных буфера; in** source **– ранг пункта отправки;in** tag **- метка сообщения; in** comm **– коммуникатор; out** request **- коммуникационный запрос.**

Вызовы данных функций создают коммуникационный объект запроса и ассоциируют его с дескриптором запроса (аргумент **request**). Этот аргумент может быть использован для получения статуса коммуникации или ожидания ее завершения. Вызов не блокирующей отправки говорит о том, что система может начать копировать данные из буфера отправки. Отправитель при этом не может модифицировать буфер отправки после того, как операция вызвана и до того, как она завершится. Вызов не блокирующего приема говорит о

том, что система может начать сохранять данные в буфере приема. Получатель при этом не может получить доступ к буферу приема после того как операция вызвана и до того как она завершиться. Для определения завершения неблокирующих коммуникаций используются функции **MPI_Wait** и **MPI_Test**. Завершение операции отправки говорит о том, что отравитель может свободно обновлять значения в буфере отправки (операция отправки оставляет содержимое буфера отправки неизмененным).

Завершение операции приема говорит о том, что буфер приема содержит принятое сообщение, приемник теперь свободно может получать доступ к нему и о том, что установлен статус объекта. Это не значит, что соответствующая операция отправки завершилась (но указывает на то, что отправка была инициирована). Для анализа атрибута **request** используются рассматриваемые ниже функции. Для ожидания завершения не блокирующих операций используется следующая функция:

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

Атрибутами этой функции являются: inout request **– запрос; out** status **- статус передачи.**

Возврат из функции **MPI_Wait** происходит, когда операция определенная аргументом **request** завершилась. Если коммуникационный объект, ассоциированный с этим запросом, был создан не блокирующим вызовом отправки или приема, тогда объект освобождается вызовом **MPI_Wait** и дескриптор запроса устанавливается в значение **MPI_REQUEST_NULL**.

Вызов возвращает в переменную status **информацию о завершённой операции. Также разрешен вызов** MPI_Wait **с нулевым или неактивным аргументом** request**. В таком случае операция завершается незамедлительно с пустым аргументом** status**. Также существует другая функция проверки завершения не блокирующей операции, синтаксис которой приведен ниже: Int** MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

Атрибутами этой функции являются:

Inout request – Запрос; Out flag - Флаг завершения операции; Out status - Статус передачи;

Вызов **MPI_Test** возвращает в переменную **flag** значение **true**, если операция, указанная в аргументе **request** завершена. В этом случае объект статуса установлен таким образом, что содержит информацию о завершенной операции; если объект коммуникации был создан неблокируемой отправкой или приемом, тогда он освобождается, и дескриптор запроса устанавливается в **MPI_REQUEST_NULL**. Вызов возвращает в переменную **flag** значение **false**, в том случае если значение объекта статуса неопределенно. Функция **MPI_Test** является локальной операцией. Коммуникационный объект запроса может быть освобожден без ожидания завершения ассоциированной с ним коммуникации, используя следующую операцию:

int MPI_Request_free(MPI_Request *request);

inout **Request** – запрос;

Данная функция освобождает память для коммуникационного объекта запроса, и присваивает аргументу **request** значение **MPI_REQUEST_NULL**. Продолжающаяся коммуникация, которая ассоциирована с запросом, будет завершена, но запрос будет освобожден только после ее завершения.

1.5. Зондирование и отмена

Операции **MPI_Probe** и **MPI_Iprobe** позволяют проверить входящие сообщения без непосредственного их приема. Пользователь может после решить, как принять их, основываясь на информации возвращенной зондированием (в основном информация возвращается в аргументе **status**). В частном случае пользователь может выделить память для буфера приема в соответствии с длиной сообщения зондирования.

Операция **MPI_Cancel** позволяет отменить незавершенные передачи. Эта функция необходима для освобождения ресурсов (памяти), используемых для обмена данными. Другими словами, реализация отправки или получения требует от пользователя выделения ресурсов системы (например, создать буфер), а отмена при этом необходима, чтобы освободить эти ресурсы в нужный момент. Синтаксис данных функций следующий:

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

Атрибутами этой функции являются:

in **source** - ранг пункта отправки; in **tag** - метка сообщения; in **comm** – коммуникатор; out **flag** - флаг успешности приема требуемого сообщения; out **status** - статус передачи;

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status); **Атрибутами этой функции являются:**

in **source** - ранг пункта отправки; in **tag** - метка сообщения; in **comm** – коммуникатор; out **status** - статус передачи;

int MPI_Cancel(MPI_Request *request);

Атрибутом этой функции является:

in **request** - коммуникационный запрос.

Если операция была отменена, тогда информация об этом будет возвращена в аргументе статуса операции, которая должна завершить коммуникацию. Для того что бы ее получить используется следующая функция:

Int MPI_Test_Cancelled (MPI_Status *status, int *flag);

in **status** - статус передачи; out **flag** - флаг состояния;

Функция вернет в переменную **flag** значение **true,**если коммуникация, ассоциированная с объектом статуса, была отменена успешно. В таком случае все другие поля аргумента **status** (**count** или **tag**) являются неопределенными. Для отмены операции приема сначала следует вызвать функцию **MPI_Test_cancelled**, чтобы проверить была ли операция отменена, для того, чтобы использовать остальные поля возвращенного объекта статуса.

2. ЗАДАНИЕ НА РАБОТУ

Задание выбирается в соответствии с вариантом, назначенным преподавателем. Реализовать при помощи посылки сообщений типа точка-точка следующие схемы коммуникации процессов:

Вариант№1. Программа осуществляет умножение двух матриц. Размеры матриц – 3*3 и 4*4. На каждом процессе, определяет произведение одной строки первой матрицы на все столбцы второй матрицы. Результаты возвращаются в родительскую задачу.

Вариант №2. Программа осуществляет вычисление определителя матрицы 4*4 методом треугольников. Каждый процесс подсчитывает только произведения, определение результата осуществляется в родительской задаче, куда передаются результаты работы процесса. По процессам распределяется вся матрица.

Вариант №3. Вычислительный кластер реализует конвейер по обработке введенных матриц (3*3). Конвейер состоит из трех сегментов (процессов). Первый сегмент конвейера реализует ввод данных. Второй сегмент конвейера определяет миноры матрицы на основе исходной матрицы, третий сегмент, используя матрицу миноров и, вычислив определитель, находит матрицу, обратную данной.

3. КОНТРОЛЬНЫЕ ВОПРОСЫ

3.1. Какими атрибутами обладает в MPI каждое посылаемое сообщение?
3.2. Что гарантирует блокировка при отправке/приеме сообщения?
3.3. Как принимающий процесс может определить длину полученного сообщения?
3.4. Сравнить эффективность реализации различных режимов пересылок данных с блокировкой между двумя выделенными процессами.
3.5. Сравнить эффективность реализации пересылок данных между двумя выделенными процессами с блокировкой и без блокировки.

ЛАБОРАТОРНАЯ РАБОТА №2
ИССЛЕДОВАНИЕ КОЛЛЕКТИВНОГО ТИПА ПЕРЕДАЧИ ДАННЫХ, ГРУПП И КОММУНИКАТОРОВ В MPI

Цель работы: исследовать способы обмена данными между процессами в режиме широковещания или группового обмена с использованием MPI-функций.

1.ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Любую задачу в MPI можно решить, используя только связь типа «точка-точка», например, если необходимо передать значение вектора **x** всем процессам параллельной программы, то можно реализовать следующий код:

```
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
for(int i = 1; i < ProcNum+1; i++)
   MPI_Send(&x, n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
```

где **x** – массив вещественных чисел, **n**– размер массива. Однако такое решение неэффективно, поскольку повторение операций передачи приведет к суммированию затрат на подготовку передаваемых сообщений. Кроме того данная операция может быть выполнена за (ProcNum-1) итераций передачи данных. Более целесообразно использовать специальную MPI-функцию широковещательной рассылки, которая будет описана далее.

рПод коллективными операциями в MPI понимаются операции над данными, в которых принимают участие все процессы используемого коммуникатора. Из этого следует, что одним из ключевых аргументов в вызове коллективной функции является коммуникатор, который определяет группу (или группы) участвующих процессов, между которыми выполняется широковещательная передача данных. Несколько коллективных функций, таких как широковещательная рассылка (**broadcast**), сбор данных (**gather**) имеют единственный инициирующий или принимающий процесс, этот процесс называется корнем (**root**). Некоторые аргументы в коллективных функциях определены как аргументы, которые используются только корневым процессом, всеми другими участниками обмена эти аргументы игнорируются.

Условия согласования типов для коллективных операций являются более строгими, чем соответствующие условия между отправителем и получателем в передаче типе «точка-точка». Для коллективных операций количество отправленных данных должно точно совпадать с количеством данных, определенным приемником. Типы данных для гетерогенных распределенных систем, соответствующих типам данных MPI, могут отличаться.

Завершение коллективной операции может происходить, как только участие процесса в данной операции окончено. Завершение в таком случае указывает на то, что вызвавшая программа может изменять буфер передачи,

но при этом это не значит, что другие процессы в группе завершили выполнение данной функции или даже начали ее. Поэтому коллективная операция имеет некоторый эффект синхронизации всех вызывающих процессов. Процессы полностью синхронизирует выполнение функции барьера (**MPI_barrier**). Коллективные операций могут использовать в качестве аргумента те же коммуникаторы, что и коммуникации типа «точка-точка». MPI гарантирует, что созданные сообщения от имени коллективной передачи не будут пересекаться с сообщениями, созданными для передачи типа «точка-точка».

1.1. Коммуникатор и виды обмена

Ключевой особенностью коллективных функций является использование группы или групп участвующих процессов. Функции при этом не имеют явного идентификатора группы в качестве аргумента, для этой цели используется коммуникатор. Существуют два типа коммуникаторов:

- коммуникаторы процессов внутри одной группы(**intra-communicators**),
- коммуникаторы процессов для нескольких групп (**inter-communicators**).

В упрощённом варианте интра-коммуникатор – это обычный коммуникатор для одной группы процессов, соединенных контекстом, в то время как интеркоммуникатор определяет две отдельные группы процессов соединенных контекстом (см. рис. 1.1). Интер-коммуникатор позволяет передавать данные между процессами из разных интра-коммуникаторов (групп).

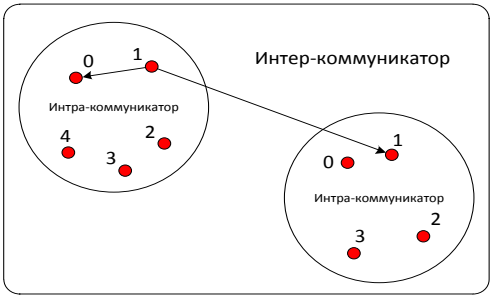


Рисунок 1.1 – Интра- и интер-коммуникаторы

Для выполнения коллективной операции (операции обмена) ее должны вызывать все процессы в группе, определенной интра-коммуникатором. В MPI используются коллективные операции интра-коммуникатора **All-To-All.** Каждый процесс в группе получает все сообщения от каждого процесса в этой же группе. Функции, используемые при обмене: **MPI_Allgather**, **MPI_Allgatherv**; **MPI_Alltoall**, **MPI_Alltoallv**, **MPI_Alltoallw**; **MPI_Allreduce**, **MPI_Reduce_scatter**; **MPI_Barrier**.

All-To-One. Все процессы группы формируют данные, но лишь один принимает их. Функции, используемые при обмене: **MPI_Gather**, **MPI_Gatherv**, **MPI_Reduce**.

One-To-All. Один процесс группы формирует данные, все процессы этой же группы принимает его. Функции, используемые при обмене: **MPI_Bcast**, **MPI_Scatter**, **MPI_Scatterv**.

Коллективные коммуникации для интер-коммуникаторов легче всего описать для двух групп. К примеру, операция «все ко всем» (**MPI_Allgather**) может быть описана как сбор данных от всех членов одной группы и получение результата всеми членами другой группы (рис.1.2).

Для интра-коммуникаторов группа обменивающихся процессов одна и та же. Для интер-коммуникаторов они различны. Для операций «все ко всем», каждой такой операции соответствуют две фазы, так что она имеет симметрию, полнодуплексное поведение. Следующие коллективные операции также применяются для интер-коммуникаций: **MPI_Barrier**, **MPI_Bcast**, **MPI_Gather**, **MPI_Gatherv**, **MPI_Scatter**, **MPI_Scatterv**, **MPI_Allgather**, **MPI_Allgatherv**, **MPI_Alltoall**, **MPI_Alltoallv**, **MPI_Alltoallw**, **MPI_AllReduce**, **MPI_Reduce**, **MPI_Reduce_scatter.**

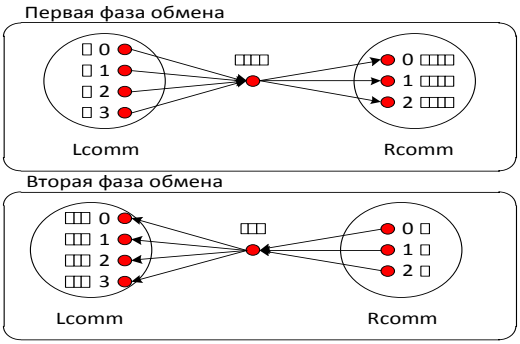


Рисунок 1.2 – Интер-коммуникатор **allgather**

Все процессы в обеих группах, определённых интер-коммуникатором, должны вызывать коллективные функции. Для коллективных операций интер-коммуникаторов если операция реализует обмен «все к одному» или «от одного ко всем», перемещение данных является однонаправленным. Направление передачи данных указывается в аргументе корня специальным значением. В этом случае для группы, содержащей корневой процесс, все процессы должны вызывать функцию, использующую для корня специальный аргумент. Для этого корневой процесс использует значение **MPI_ROOT**; все другие процессы в той же группе, что и корневой процесс, используют **MPI_PROC_NULL**. Если операция находиться в категории «все ко всем», то перенос является двунаправленным.

1.2. Барьерная синхронизация

Для синхронизации выполнения всех процессов группы используется следующая функция:

Int MPI_Barrier (MPI_Comm comm);

Атрибутом этой функции является: in comm **– коммуникатор;**

Если аргумент **comm** является интра-коммуникатором, функция **MPI_Barrier** блокирует процесс, который его вызвал до того момента, пока все члены группы не вызовут эту же операцию. В любом процессе функция может завершиться только после того, как все члены группы достигли выполнения этой функции. Если аргумент **comm –** интер-коммуникатор, функция **MPI_Barrier** включает в себя две группы и завершается в процессах первой группы только после того, как все члены другой не начали выполнять данную функцию (и наоборот). При этом процесс может завершить выполнение данной функции до того, как все процессы в его собственной группе не начали ее выполнять.

1.3. Основные функции, реализующие широковещание

Чтобы передать значение от одного процесса всем процессам его группы (совершить **широковещательную рассылку**) используется функция, синтаксис которой имеет вид:

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

Атрибутами этой функции являются:

in out **buffer** - указатель на буфер; in **count** - количество элементов в буфере; in **datatype** - тип данных буфера; in **Root** - ранг корня широковещательной рассылки; in **Comm** – коммуникатор;

Если аргумент **comm** является интра-коммуникатор, функция **MPI_Bcast** распространяет сообщение от процесса с рангом **root** ко всем процессам группы, включая и себя. Это реализуется всеми членами группы, используя одни и те же аргументы **comm** и **root**. После завершения содержимое корневого буфера является скопированным во всех других процессах.

Рисунок 1.3 – Реализация функции широковещательной рассылки

Если аргумент **comm** является предварительно созданными интер-коммуникатором, тогда вызов включает в себя все процессы в интер-коммуникаторе, но при этом процесс-корень находиться в одной группе (группе А), а всем процессам другой группы (группы В) должен быть указан в качестве аргумента идентификатор корня (**root**) из группы, где находиться

источник рассылки (группа A). Корневой процесс передает значение **MPI_ROOT** в аргумент **root**. Все другие процессы в группе A передают значение **MPI_PROC_NULL** в аргумент **root**. Данные распространяются от корня ко всем процессам в группе B. Для того чтобы собрать (**редукция**) значения со всех процессов группы в одном процессе, используется функция:
Int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatypesendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
Атрибутами этой функции являются:
in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема (используется только корневым процессом); in **recvcount** - количество элементов одиночного приема (неотрицательное число, используется только корневым процессом); in **recvtype** - тип данных буфера приема (используется только корневым процессом); in **root** - ранг принимающего процесса; in **comm** - коммуникатор;

Если аргумент **comm** интра-коммуникатор, тогда каждый процесс (включая корневой процесс) отправляют содержимое собственного буфера отправки корневому процессу. Корневой процесс принимает сообщения и сохраняет их в ранжированном порядке. Вызов данной функции является идентичным тому, что каждый из **n** процессов в группе (включая коневой процесс) выполнили вызов **MPI_Send(sendbuf, sendcount, sendtype, root, …)**, а корень выполнил n вызовов **MPI_Recv().**

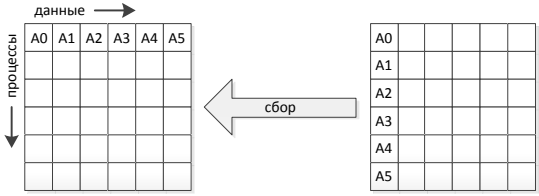


Рисунок 1.4 – Иллюстрация функции сбора

Если аргумент **comm** интер-коммуникатор, тогда вызов включает в себя все процессы интер-коммуникатора, но с одной группой (группа A), определяющей корневой процесс. Все процессы в другой группе (группа B) передают то же значение в аргумент **root**, который является корнем в группе A. В корневом процессе в качестве аргумента **root** указывается значение **MPI_ROOT**. Все другие процессы в группе A передают значение **MPI_PROC_NULL** в аргумент **root**. Данные собираются ото всех процессов в группе B к корню. Для сбора данных используется следующая функция:
Int MPI_Gatherv(void* sendbuf, intsendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, int root, MPI_Comm comm);
Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcounts** - массив неотрицательных чисел (длина равна размеру группы), содержащий количество элементов принимаемых от каждого процесса (используется только корневым процессом); in **displs** - целочисленный массив (длина равна размеру группы, элемент **i** указывает смещение относительно начала массива **recvbuf**, в котором необходимо заменить входные данные от процесса **i** (используется только корневым процессом)); in **recvtype** - Тип данных буфера приема (используется только корневым процессом); in **root** - Ранг принимающего процесса;

Так как аргумент **recvcounts** является массивом, функция **MPI_Gatherv** расширяет функциональность операции **MPI_Gather**, допуская переменное количество данных в каждом процессе. Дополнительную гибкость дает аргумент **displs** для данных, размещаемых в корне.

Пример использования функции **MPI_Gather().**

Необходимо произвести сбор 100 целочисленных значений от каждого процесса в группе к корню (рис. 1.5):

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf;
// ...
MPI_Comm_size(comm, &gsize);
rbuf = (int*)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

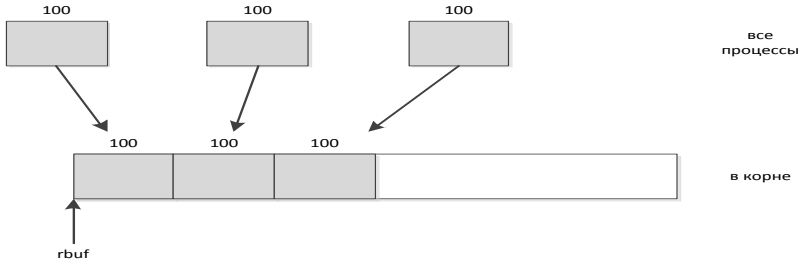


Рисунок 1.5 – Схема реализации сбора значений

Обратная функция для операции **MPI_Gather** следующая:

Int MPI_Scatter(void* sendbuf, intsendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема (используется только корневым процессом); in **recvcount** - количество элементов одиночного приема (используется только

коневым процессом); in **recvtype** - тип данных буфера приема (используется только корневым процессом); in **root** - ранг принимающего процесса; in comm - Коммуникатор;

Если аргумент **comm** является интра-коммуникатор, результат можно проинтерпретировать так, как будто корень выполнил **n** операций отправки:
MPI_Send(sendbuf +i * sendcount * extent(sendtype), sendcount, sendtype, i, …);
и каждый процесс выполнил прием:
MPI_Recv(recvbuf, recvcount, recvtype, i, …);

Альтернативное описание является следующим: корень отсылает сообщение с помощью
MPI_Send(sendbuf, sendcount * n, sendtype, …);

Это сообщение расщепляется на **n** равных сегментов, **i**-ый сегмент отправляется **i**-ому процессу в группе, и каждый процесс принимает сообщение так, как описано выше. Буфер отправки игнорируется для всех некорневых процессов.

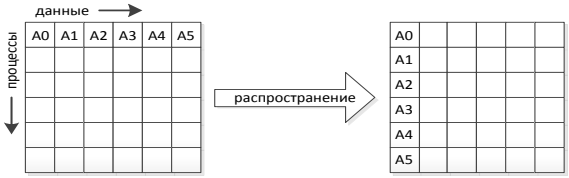


Рисунок 1.6 – Иллюстрация функции распространения MPI_Scatter ()

Если аргумент **comm** является интер-коммуникатором, тогда вызов включает в себя все процессы интер-коммуникатора, но только в одной группе (группа A) определяется корневой процесс. Все процессы в другой группе (группа B) передают одно и то же значение в аргумент **root**, которое является рангом корневого процесса в группе A. Корень передает значение **MPI_ROOT** в аргумент **root**. Данные распространяются от корня ко всем процессам в группе B. Чтобы совершить действия, обратные операции **MPI_Gatherv**, используется функция, синтаксис которой следующий:
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

Атрибутами этой функции являются:
in **sendbuf** - указатель на буфера отправки; in **sendcount** - количество элементов буфера отправки; in **displs** - целочисленный массив (длина–размер группы, элемент **i** указывает смещение относительно **sendbuf**, у которого необходимо взять данные, передаваемые процессу **I);** in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcount** - количество элементов буфера приема; in **recvtype** - тип данных буфера приема; in **root** - ранг принимающего процесса; in **comm** – коммуникатор;

Так как **sendcounts** является массивом, функция **MPI_Scatterv** расширяет функциональность **MPI_Scatter**, допуская переменное количество данных для отправки каждому процессу. Также дает дополнительную гибкость аргумент **displs**, указываемый для данных, которые берутся в корне.

Для того чтобы передать данные всем процессам, а не одному корню (как в функции **MPI_Gather**), используется следующая функция:
Int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
Атрибутами этой функции являются:
in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcount** - количество элементов буфера приема; in **recvtype** - тип данных буфера приема; in **comm** – коммуникатор.

Блок данных, отправленных от **j**-ого процесса, принимается каждым процессом и размещается в **j**-ом блоке буфера **recvbuf**. Если аргумент **comm** является интра-коммуникатором, результат вызова функции **MPI_Allgather** можно считать таким, что все процессы выполнили **n** вызовов: **MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);** где аргумент root принимает значения от 0 до n-1. Правила правильного использования функции **MPI_Allgather** аналогичны соответствующим правилам использования функции **MPI_Gather**.

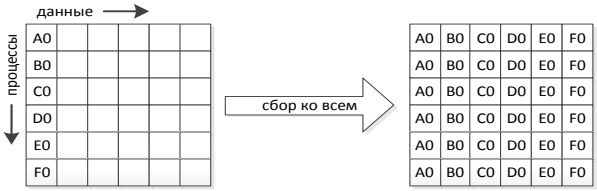


Рисунок 1.7 – Иллюстрация функции сбора ко всем

Если необходимо передать различное количество данных всем процессам, то используется функция, синтаксис которой следующий:
Int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)
Атрибутами этой функции являются:
in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out recvbuf - указатель на буфер приема; in **recvcounts** - количество элементов буфера приема; in **displs** -цЦелочисленный массив (длина равна размеру группы). Элемент **I** указывает смещение относительно **recvbuf**, где необходимо поместить входные данные от процесса **I;** in recvtype - Тип данных буфера приема; in comm – Коммуникатор;

Блок данных, отправленных от **j**-ого процесса, принимается всеми процессами и размещается в **j**-ом блоке буфера **recvbuf**. Для того чтобы каждый процессом отправил различные данные каждому процессу-приемнику, используется следующая функция:

Int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфераотправки; out **recvbuf** - указатель на буфер приема; in **recvcount**- количество элементов буфера приема; in **recvtype** - тип данных буфера приема; in **comm** – коммуникатор;

Функция **MPI_Alltoall** является расширением функции **MPI_Allgather**, где **j**-ый блок, отправленный от **i**-ого процесса, принимается **j**-ым процессом и размещается в **i**-ом блоке **recvbuf**. Все аргументы на всех процесса являются значимыми. Аргумент **comm** должен иметь идентичное значение на всех процессах.

Рисунок 1.8 – Иллюстрация функции полного обмена

Если аргумент **comm** является интер-коммуникатором, тогда результат таков, что каждый процесс группы A отправляет сообщение каждому процессу группы B и наоборот.

Для того чтобы каждый процессом отправил данные различного размера каждому процессу-приемнику, используется следующая функция:

Int MPI_Alltoallv(void*sendbuf, int*sendcounts,int *sdispls,MPI_Datatypesendtype, void*recvbuf, int *recvcounts,int*rdispls,MPI_Datatype recvtype,MPI_Comm comm);

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcounts** - массив неотрицательных целочисленных значений (длина равна размеру группы) указывающий количество элементов, которые необходимо отправить каждому процессу; in **sdispls** - целочисленный массив (длина – размер группы, элемент **j** указывает смещение относительно аргумента **sendbuf**, у которого необходимо взять данные передаваемые процессу **j);** in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcounts** - массив неотрицательных чисел (длина равна размеру группы), указывающий количество элементов, которое может быть принято от каждого процесса; in **rdispls** - целочисленный массив (длина равна размеру группы, элемент **i** указывает смещение относительно аргумента **recvbuf**, где необходимо поместить входные

данные от процесса **I);** in **recvtype** - тип данных буфера приема; in **comm** – коммуникатор;

1.4. Глобальные операции предварительной обработки

Функции, описываемые в этом разделе, выполняют глобальные операции предварительной обработки (к примеру, сумма, максимум и т.д.) для всех членов группы. Операция предварительной обработки может быть как одной из списка, так и определенной пользователем. Глобальные функции предварительной обработки имеют несколько разновидностей:

– обработка, которая возвращает результат одному процессу группы,
– обработка, которая возвращает результат всем членам группы,
– две операции сканирования,
– операция одновременной обработки и распространения.

Синтаксис первой из них следующий:

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

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; out **recvbuf** - указатель на буфер приема (используется только корневым процессом); in **count** - количество элементов буфера отправки; in **datatype** - тип данных буфера отправки; in **op** - операция предварительной обработки; in **root** - ранг корневого процесса; in **comm** – коммуникатор;

Если аргумент **comm** является интра-коммуникатором, функция **MPI_Reduce** собирает все элементы, переданные во входной буфер каждого процесса в группе, выполняет операцию **op** и возвращает значение в выходной буфер процесса с рангом **root**. Входной буфер определяется аргументами **sendbuf**, **count** и **datatype**; выходной буфер – аргументами **recvbuf**, **count** и **datatype**; оба буфера имеют одинаковое количество элементов с одним и тем же типом. Функция вызывается всеми членами группы, используя одни и те же аргументы **count**, **datatype**, **op**, **root** и **comm**. Поэтому все процессы обеспечивают функцию входными (выходным для корня **root**) буферами одной и той же длины, с элементами одинакового типа. Каждый процесс может предоставлять один элемент или последовательность элементов, в таком случае комбинированная операция выполняется над каждым элементом последовательности:

$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, \qquad 0 \le j < n;$$

где $\otimes$ – операция, задаваемая при вызове функции **MPI_Reduce**. Пример выполнения операции редукции при суммировании пересылаемых данных трех процессов. В каждом сообщении 4 элемента, сообщения собираются на процессе с рангом 2 (рис. 1.9):

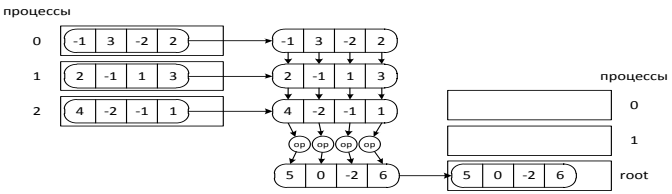


Рисунок 1.9 – Пример выполнения операции редукции

Список предопределенных функций рассмотрен в табл. 1.1.Каждая функция может выполняться только с определёнными типами данных. В дополнении к этому пользователь может определить собственную операцию, которая может быть перегружена с несколькими типами.

Операция **op** всегда предполагается как ассоциативная. Все предопределенные операции также предполагаются, что являются и коммутативными. «Каноническое» определение порядка предварительной обработки обуславливается рангом процессов в группе. Как бы то ни было, реализация может получать преимущество ассоциативности или ассоциативности и коммутативности при изменении порядка обработки, нов таком случае может измениться результат предварительной обработки для операций, которые не строго ассоциативные или коммутативные, такие например как сложение чисел с плавающей точкой.

Аргумент **datatype** функции **MPI_Reduce** должен быть совместимым с операцией **op**. Более того аргумент **datatype** и операция **op** для предопределённых операторов должны быть одинаковы во всех процессах.

Необходимо заметить, что для пользователя возможно определение различных пользовательских операций, но тогда MPI не ведет контроль над тем, какая операция используется над каким операндом.

Таблица 1.1

Предопределенные операций в Reduce

MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое И
MPI_BAND	побитовое И
MPI_LOR	логическое ИЛИ
MPI_BOR	побитовое ИЛИ
MPI_LXOR	логическое исключающее ИЛИ (xor)
MPI_BXOR	побитовое исключающее ИЛИ (xor)
MPI_MAXLOC	максимальное значение и местоположение
MPI_MINLOC	минимальное значение и положение

Оператор **MPI_MINLOC** используется для вычисления глобального минимума, а также индекса прикрепленного к минимальному значению. Операция **MPI_MAXLOC** подобным образом вычисляет глобальный максимум и индекс. Операция, которая определяет **MPI_MAXLOC** следующая:

$$\binom{u}{i} \circ \binom{v}{j} = \binom{w}{k}$$

где

$$w = max(u, v)$$

и

$$k = \begin{cases} i & if\ u > v \\ min(i, j) & if\ u = v \\ j & if\ u < v \end{cases}$$

Операция **MPI_MINLOC** определена следующим образом:

$$\binom{u}{i} \circ \binom{v}{j} = \binom{w}{k}$$

где

$$w = min(u, v)$$

и

$$k = \begin{cases} i & if\ u < v \\ min(i, j) & if\ u = v \\ j & if\ u > v \end{cases}$$

Пример использования функции Reduce:

```
// каждый процесс имеет массив 30 double: ain[30]
double ain[30], aout[30];
int ind[30];
struct
{
   double val;
   int rank;
} in[30], out[30];
int i, myrank, root;
MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i)
{
   in[i].val = ain[i];
   in[i].rank = myrank;
}
MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
// В этой точке ответ находится в корневом процессе
if (myrank == root)
{
   // read ranks out
```

```
   for (i = 0; i < 30; ++i)
   {
      aout[i] = out[i].val;
      ind[i] = out[i].rank;
   }
}
```

Каждый процесс содержит массив 30 вещественных чисел. Для каждого из 30 местоположений вычисляется значение и ранг процесса содержащего наибольшее значение.

Для того, чтобы соединить определенные пользователем функции предварительной обработки с дескриптором операции op, используется следующая функция: int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op);

Атрибутами этой функции являются:

in **Function** - определенная пользователем функция; in **Commute** - True если функция коммутативная, false в любом другом случае; out **Op** – операция;

Пользовательские определенные операции предполагаются ассоциативными. Если аргумент **commute** = **true**, тогда операция должна быть как коммутативной, так и ассоциативная. Если аргумент **commute** = **false**, тогда порядок операндов фиксирован и определяется восходящим порядком рангов процессов, начиная с нулевого процесса. Если аргумент **commute** = **true**, тогда порядок вычисления может быть изменен. Аргумент **function** является пользовательской функцией, которая должна иметь следующие четыре аргумента: **invec**, **inoutvec**, **len** и **datatype**.

Прототип выглядит следующим образом:

```
void MPI_User_function(void *invec, void *inoutvec, int *len, MPI_Datatype
*datatype);
```

Аргументы **invec** и **inoutve –** массивы с **len** элементами, которые функция комбинирует. Результат предварительной обработки сохраняет значения в аргументе **inoutvec**. Каждый вызов функции ведет к поточечному вычислению оператора предварительной обработки **len** элементов.

Пример использования определенных пользователем операций с использованием интра-коммуникатора:

```
typedef struct {
   double real, imag;
} Complex;
// определенная пользователем функция
void myProd(Complex *in, Complex *inout, int *len, MPI_Datatype *dptr)
{
   int i;
   Complex c;
   for (i=0; i< *len; ++i)
   {
```

```
        c.real = inout->real*in->real - inout->imag*in->imag;
        c.imag = inout->real*in->imag + inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}
// каждый процесс имеет массив 100 Complex-ов
    Complex a[100], answer[100];
    MPI_Op myOp;
    MPI_Datatype ctype;
    // объяснение MPI как тип Complex определен
    MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
    MPI_Type_commit(&ctype);
    // создание комплексного произведения пользовательской операцией
    MPI_Op_create(myProd, 1, &myOp);
    MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
    /* В этой точке ответ, который состоит из 100 Complex-ов, находящий-
ся в корневом процессе */
```

Функция, которая относится ко второй разновидности редукции, имеет следующий синтаксис:

```
Int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Opop, MPI_Commcomm);
```

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; out **recvbuf** - указатель на буфер приема; in **count** - количество элементов буфера отправки (неотрицательное целочисленное значение); in **datatype** - тип данных буфера отправки; in **op** - операция предварительной обработки; in **comm** – коммуникатор;

Если аргумент **comm–**интра-коммуникатор, функциия **MPI_Allreduce** ведет себя также как и функция **MPI_Reduce** за исключением того, что после выполнения результат содержится в буфере приема всех членов группы.

Если аргумент **comm** является интер-коммуникатором, тогда результат предварительной обработки данных, обеспеченных процессами в группе A,сохраняются в каждом процессе группы B, и наоборот. Обе группы должны обеспечить аргументы **count** и **datatype**.

MPI включает в себя варианты операций предварительной обработки, где после выполнения результат распространяется ко всем процессами в группе. Один вариант распространяет блоки одинокого размера всем процессам, в то время как другой вариант распространяет блоки, которые могут варьироваться по размеру для каждого процесса.

Вариант второй из них следующий:

```
Int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int
*recvcounts,MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Атрибутами этой функции являются:

in **sendbuf** - указатель набуферотправки; out **recvbuf** - указатель на буфер приема; in **recvcounts** - массив неотрицательных целочисленных значений (длина равна размеру группы), указывающий количество элементов распределенного результата для каждого процесса; in **datatype** - тип данных буфера отправки и приема; in **op** – операция; in **comm** – коммуникатор;

Функция **MPI_Reduce_scatter** является расширением **MPI_Reduce_scatter_block**, так как в отличие от нее распространяет блоки, которые могут отличаться в размере. Размеры блоков определяются массивом **recvcounts**, **i**-ый блок содержит **recvcounts[i]** элементов.

Если аргумент **comm** является интра-коммуникатором, функция **MPI_Reduce_scatter** сначала выполняет глобальную поэлементную предварительную обработку над вектором:

$$count = \sum_{i=0}^{n-1} recvcount[i]$$

где **n** является количеством процессов в группе аргумента **comm**. Функция вызывается всеми членами группы, используя одинаковые аргументы **recvcount**, **datatype**, **op** и **comm**. Результирующий вектор принимается, как **n** последовательных блоков, где количество элементов **i**-ого блока является значением **recvcount[i]**. Таким образом, **i**-ый блок отправляется процессу **I** и сохраняется в буфере приема, определенном с помощью аргументов **recvbuf**, **recvcount[i]** и **datatype**.

Если аргумент **comm–** интер-коммуникатор, тогда результат предварительной обработки данных предоставленных процессами группы A распространяется по процессам в группе B и наоборот. Внутри каждой группы все процессы имеют одинаковый аргумент **recvcounts**, входное количество (значение **count**) элементов сохраненных в буфере отправки. Результирующий вектор от другой группы распространяется в блоки **recvcount[i]** элементов по процессам в группе. Количество элементов **count** должно быть одинаково для двух групп.

1.5. Группы, контексты, коммуникаторы

Группы определяют упорядоченную совокупность процессов, каждый из которых имеет ранг, т.е. группа определяет низкоуровневые имена для коммуникаций между процессами. Поэтому группы определяют границы для имен процессов в коммуникации типа точка-точка. В дополнение, группы определяют границы для коллективных операций. Группы управляются отдельно от коммуникаторов в MPI, но только коммуникаторы могут быть использованы в коммуникационных операциях внутри групп.

Наиболее часто используемые средства для передачи сообщений в группах являются интра-коммуникаторами. Они содержат экземпляр группы, контексты коммуникаций (информацию для коммуникации) как для связи типа точка-точка, так и для коллективного типа связи, возможность включать в себя топологию и другие атрибуты. Интер-коммуникаторы реализуют

взаимодействие между двумя не накрадывающимися друг на друга группами. Когда приложение построено образованием нескольких параллельных модулей, удобно разрешать одному модулю взаимодействовать с другим, используя локальные ранги для адресации внутри второго модуля. Это особенно удобно в клиент-серверной обрабатывающей парадигме, где как клиент, так и сервер являются параллельными. Интер-коммуникаторы связывают две группы вместе с коммуникационным контекстом, разделяемым обеими группами. Для интер-коммуникаторов использование контекстов обеспечивают возможность иметь обособленные защищенные «среды» передачи сообщений между двумя группами. Отправка в локальной группе всегда принимается в удалённой группе, и наоборот. Процесс установления различия организует система. Локальная и удалённая группы определяют пункты отправки и назначения для интер-коммуникатора.

1.6. Основная концепция работы с несколькими процессами

Группа является упорядоченным набором идентификаторов процессных (далее процессы). Каждый процесс в группе ассоциируется с целочисленным рангом. Ранги являются последовательными и начинаются с нуля. Группа используется внутри коммуникатора для описания участников коммуникационной «среды» и для ранжирования этих участников (т.е. происходит раздача им уникального имени внутри «среды» коммуникации).

Существует специальная предопределенная группа: **MPI_GROUP_EMPTY**, которая является группой без участников. Предопределённая константа **MPI_GROUP_NULL** является значением, используемым для недопустимого идентификатора группы.

MPI-коммуникационные операции ссылаются на коммуникаторы для определения границ и «коммуникационной среды», в которой операции типа точка-точка или коллективного типа являются возможными. Каждый коммуникатор содержит группу допустимых участников; эта группа всегда включает в себя локальный процесс. Источник и место назначения сообщения определяются рангом процесса внутри группы. Для коллективных коммуникаций интра-коммуникатор определяет ряд процессов, которые участвуют в коллективной операции (и их порядок, когда это необходимо). Поэтому коммуникатор ограничивает «пространственные» границы коммуникации, и обеспечивает адресацию процессов независящую от машины посредством ранга.

После инициализации локальный процесс может общаться со всеми процессами интра-коммуникатора **MPI_COMM_WORLD** (включая и себя), определённого один раз **MPI_INIT** или **MPI_INIT_THREAD** вызовами. Предопределенная константа **MPI_COMM_NULL** является значением, использующимся для недопустимого дескриптора коммуникатора.

В реализации статической модели процессов MPI, все процессы, которые участвуют в общении, являются доступными после инициализации. Для этого случая **MPI_COMM_WORLD** является коммуникатором всех процес-

сов доступных для коммуникации. В реализации MPI, процессы начинают вычисления без доступа ко всем другим процессам. В таком случае **MPI_COMM_WORLD** является коммуникатором, объединяющим все процессы, с которыми присоединяющийся процесс может незамедлительно общаться. По этой причине **MPI_COMM_WORLD** может одновременно представлять отделенные группы в разных процессах.

Коммуникатор **MPI_COMM_WORLD** не может быть освобождён в течение функционирования процесса. Группа соответствующая этому коммуникатору нигде не проявляется, так как является предопределенной константой, но она может быть доступна через использование функции **MPI_Comm_Group**.

1.7. Управление группами

Чтобы получить информацию о группе, используются следующие функции:

int MPI_Group_size(MPI_Group group, int *size).

Атрибутами этой функции являются:

in **group** – группа; out **size** - количество процессов в группе.

int MPI_Group_rank(MPI_Group group, int *rank).

Атрибутами этой функции являются:

in **group** – группа; out **rank** - ранг вызывающего процесса в группе, или значение **MPI_UNDEFINED**,если процесс не является членом группы.

Следующая функция используется для определения относительной нумерации одних и тех же процессов в двух разных группах:

int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
MPI_Group group2, int *ranks2)

Атрибутами этой функции являются:

in **group1** - первая группа; in **n** - количество рангов в массивах **ranks1**и **ranks2;** in **ranks1** - массив нулевого или более допустимых рангов в **group1;** in **group2** - вторая группа; out **ranks2** - массив соответствующих рангов в **group2**, значение **MPI_UNDEFINED,** если нет соответствий;

К примеру, если известны ранги текущих процессов в группе **MPI_COMM_WORLD**, может понадобиться узнать их ранги в подмножестве этой группы. Значение**MPI_PROC_NULL**является допустимым рангом для ввода в **MPI_Group_translate_ranks**, который возвращает значение **MPI_PROC_NULL** как переданный ранг.

Конструкторы группы используются для создания подмножества и расширения существующих групп. Эти конструкторы создают новые группы из существующих групп. Существуют локальные операции и отдельные группы, которые могут быть определены на разных процессах; процесс может также определять группу, которая не включает себя. Устойчивое определение необходимо, когда группы используются как аргументы в функциях создания коммуникаторов. MPI не обеспечивает механизм для создания

групп с нуля, их можно создать только из других до этого определенных групп. Основная группа, с помощью которой строятся все остальные группы, является группа, ассоциированная с начальным коммуникатором **MPI_COMM_WORLD**, доступная через функцию **MPI_Comm_group**:
Int MPI_Comm_group(MPI_Commicomm, MPI_Group *group);
Атрибутами этой функции являются:
in **comm** – коммуникатор; out **group** –группа, соответствующая **comm;**
Следующие функции позволяют создавать новые группы:
Int MPI_Group_union(MPI_Groupgroup1, MPI_Groupgroup2,
MPI_Group *newgroup);
Атрибутами этой функции являются:
in group1 - первая группа; in group2 - вторая группа; out newgroup - группа объединения.
Int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
Атрибутами этой функции являются:
in **group1** - первая группа; in **group2** - вторая группа; out **newgroup** - группа пересечения.
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup) **Атрибутами этой функции являются:**
in **group1** - первая группа; in **group2** - вторая группа; out **newgroup** - группа разницы;
Операции определены следующим образом:
1) в **MPI_Group_Union()** все элементы первой группы (**group1**), затем все элементы второй группы (**group2**), которых нет в первой группе.
2) в **MPI_Group_Intersection()** все элементы первой группы, которые также есть и во второй группе, упорядоченные как в первой группе.
3) в **MPI_Group_Difference()** все элементы первой группы, которых нет во второй группе, упорядоченные как в первой группе.
Для этих операций порядок процессов выходной группы определяется главным образом порядком в первой группе (если возможно) и , если необходимо, порядком во второй группе. Ни объединение, ни пересечение не являются коммутативными, но обе являются ассоциативными. Чтобы включить процессы в новую группу с определенными рангами используется следующая функция:
int MPI_Group_incl(MPI_Group group, int n, int *ranks,
MPI_Group *newgroup);
Атрибутами этой функции являются:
in **group** – группа; in **n** - количество элементов в массиве рангов (и размер **newgroup**); in **ranks** - ранги процессов в **group**, которые должны появиться в **newgroup;**out **newgroup** - новая группа производная от вышеупомянутой, в порядке определения **ranks;**

Функция **MPI_Group_incl()** создает группу **newgroup**, которая состоит из **n** процессов в **group** с рангами **ranks[0]**, …, **ranks[n-1]**; процесс с рангом **i** в **newgroup** является процессом с рангом **ranks[i]** в **group**. Каждый из **n** элементов ранга **ranks** должен быть допустимым рангом в **group**, и все элементы должны быть индивидуальными, иначе программа является ошибочной. Если **n = 0**, тогда группа **newgroup** является **MPI_GROUP_EMPTY**. Эти функции могут, к примеру, использоваться для переупорядочивания элементов группы, или для сравнения. Для того, чтобы исключить процессы из группы с определенными рангами используется следующая функция:

int MPI_Group_excl(MPI_Group group, int n, int *ranks,
MPI_Group *newgroup);

Атрибутами этой функции являются:

in **group** – группа; in **n** - количество элементов в массиве рангов (и размер **newgroup**); in **ranks** - массив целочисленных рангов в **group**, которые не должны появиться в **newgroup;** out **newgroup** - новая группа производная от вышеупомянутой, сохраняющая порядок определенный **group;**

Функция **MPI_Group_excl** создает группу процессов **newgroup** путем удаления из **group** этих процессов с рангами **ranks[0]**, …, **ranks[n-1]**. Упорядочивание процессов в **newgroup** является идентичным упорядочиванию в **group**. Каждый из **n** элементов рангов **ranks** должен быть допустимым в **group.** Если **n = 0**, тогда группа **newgroup** является идентичной **group**.

1.8. Управление коммуникаторами

Операции, которые получают доступ к коммуникаторам, являются локальными и их выполнение не требует взаимодействия процессов. Операции, которые создают коммуникаторы, являются коллективными и могут требовать взаимодействия между процессами. Для того чтобы сравнить два коммуникатора используется следующая функция:

int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result);

Атрибутами этой функции являются:

in **comm1** - первый коммуникатор; in **comm2** - второй коммуникатор; out **result** – результат.

Результат будет иметь значение **MPI_IDENT**, если аргументы **comm1** и **comm2** являются дескрипторами одного и того же объекта (идентичные группы и одинаковый контекст). Значение **MPI_CONGRUENT** будет результатом, если группы являются идентичными в компонентах и порядке рангов; эти коммуникаторы отличаются только контекстом. Значение **MPI_SIMILAR** является результатом, если члены группы обоих коммуникаторов одинаковы, но порядок рангов отличается. Результат будет равен значению **MPI_UNEQUAL** в любом другом случае. Следующие операции являются коллективными функциями, которые вызываются всеми процессами в группе или группах ассоциированных с аргументом **comm.**

MPI обеспечивает четыре функции конструирования коммуникатора, которые применяются к интра-коммуникаторам и интер-коммуникаторам. Функция конструирования **MPI_Intercomm_create** применяется только к интер-коммуникаторам. В интер-коммуникаторе группы называются левой и правой. Процесс в интер-коммуникаторе является членом либо левой, либо правой группы. С точки зрения этого группа процесса, членом которой он является, называется локальной; другая группа (относительно этого процесса) является удаленной (дистанционной). Метки левой и правой групп дают возможность указывать две группы в интер-коммуникаторе, которые не относятся к какому-либо конкретному процессу.

Для создания коммуникатора используется следующая функция:
Int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm) **;**

Атрибутами этой функции являются:
in **comm** – коммуникатор; in **group** - группа, которая является подмножеством группы; коммуникатора **comm;** out **newcomm** - новый коммуникатор**.**

Если аргумент **comm** является интра-коммуникатором, эта функция возвращает новый коммуникатор **newcomm** с коммуникационной группой, определённой аргументом **group**. Никакая кэшированная информации не распространяется от коммуникатора **comm** к новому коммуникатору **newcomm**. Каждый процесс должен выполнять вызов функции с аргументом **group**, который является подгруппой группы ассоциированной с аргументом **comm**. Процессы могут указывать различные значения для аргумента **group**.
Если аргумент **comm** является интер-коммуникатором, тогда выходной коммуникатор также является интер-коммуникатором, где локальная группа состоит только из тех процессов, которые содержатся в группе **group**. Аргумент **group** должен иметь те процессы в локальной группы входного интер-коммуникатора, который является частью коммуникатора **newcomm**. Все процессы в одной и той же локальной группе коммуникатора **comm** должны указывать одинаковое значение для аргумента **group**. Если группа **group** не указывает хотя бы один процесс в локальной группе интер-коммуникатора, или если вызывающий процесс не является включенным в аргумент **group**, возвращается **MPI_COM_NULL.**

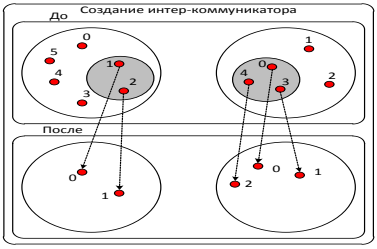


Рисунок 1.10 – Создание интер-коммуникатора

Для разделения коммуникатора используется следующая функция:
Int MPI_Comm_split(MPI_Commcomm, int color, int key, MPI_Comm *newcomm);

Атрибутами этой функции являются:

in **comm** – коммуникатор; in **color** - контроль распределения подмножества; in **key** - контроль распределения рангов; out **newcomm** - новый коммуникатор.

Функция разделяет группу, ассоциированную с аргументом **comm** на отдельные подгруппы, одна для каждого значения аргумента **color**. Каждая подгруппа содержит все процессы одинакового цвета. Внутри каждой группы процессы ранжируются в порядке определенном значением аргумента **key**, со связями, соответствующими их рангу в старой группе. Новый коммуникатор создается для каждой подгруппы и возвращается в аргументе **newcomm**. Процесс может обеспечивать значение аргумента color как **MPI_UNDEFINED**, в таком случае коммуникатор **newcomm** будет иметь значение **MPI_COMM_NULL**. Это коллективный вызов, но каждому процессу разрешается задавать различное значения **color** и **key**.

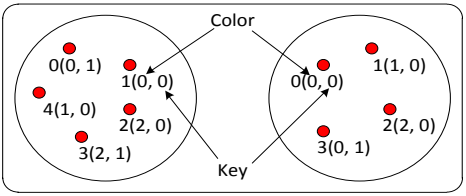


Рисунок 1.11 – Входной коммуникатор (**comm**)

С интра-коммуникатором **comm** вызов **MPI_Comm_create(comm, group, newcomm)** является эквивалентом вызова **MPI_Comm_split(comm, color, key, newcomm)**, где процессы, которое являются членами их аргумента **group** обеспечивают **color**= числу групп (основывается на уникальной нумерации всех отдельных групп) и **key** = рангу в группе, все процессы которые не являются членами их аргумента **group** обеспечивают **color**= **MPI_UNDEFINED**.

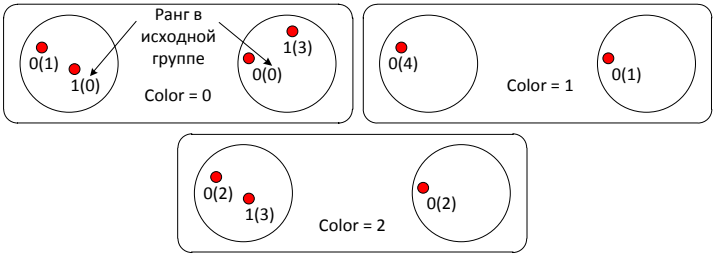


Рисунок 1.12 – Отделенные выходные коммуникаторы (**newcomm**)

Результат функции **MPI_Comm_split** на интер-коммуникаторе является таким, что процессы слева с одинаковым значением аргумента **color**, как у процессов справа, объединяются, чтобы создать новый интер-коммуникатор. Аргумент **key** описывает относительный ранг процессов на каждой стороне интер-коммуникатора. Для тех цветов, которые указываются только на одной стороне интер-коммуникатора, возвращается значение **MPI_COMM_NULL**. Для освобождения коммуникатора используется функция, синтаксис которой следующий:

Int MPI_Comm_free(MPI_Comm *comm)**;**

Атрибутом этой функции является: in comm - **коммуникатор, который необходимо освободить.**

Дескриптор устанавливается в значение **MPI_COMM_NULL**. Любые неоконченные операции, которые используют этот коммуникатор, будут завершены в нормальном режиме; объект освобожден фактически, только если не существует активных ссылок на него. Этот вызов применяется для интра- и интер-коммуникаторов. Функция **MPI_Intercomm_create** используется для того, чтобы связать два интра-коммуникатора в интер-коммуникатор. Функция **MPI_Intercomm_merge** создает интра-коммуникатор, соединяя локальную и удаленную группы интер-коммуникатора. Частичное совпадение локальной и удаленной групп, которые связаны в интер-коммуникаторе, запрещено. Если существует частичное совпадение, то тогда программа является ошибочной и вероятней всего ведет к взаимной блокировке.

Функция **MPI_Intercomm_create** может быть использована для создания интер-коммуникатора из двух существующих интра-коммуникаторов в следующей ситуации: хотя бы один выбранный член из каждой группы («лидер группы») имеет возможность взаимодействовать с выбранным членом из другой группы; другими словами существует так называемый равноправный коммуникатор (“peer” communicator), которому принадлежат оба лидера, и каждый лидер знает ранг другого лидера в этом равноправном коммуникаторе. Кроме того члены каждой группы знают ранг их лидера.

Построение интер-коммуникатора из двух интра-коммуникаторов требует вызовов отдельных коллективных операций в локальной группе и в удаленной группе, также как передача типа «точка-точка» между процессом в локальной группе и процессом в удалённой группе.

Алгоритм создания интер-коммуникатора (для локальных групп с последующим обменом между ними): 1) формирование групп процессов для коммуникатора по умолчанию (**MPI_COMM_WORLD**); 2) исключение процессов из групп (формирование ограниченных групп процессов); 3) создание коммуникатора для обмена внутри группы (связывание интра-коммуникатора с каждой из групп); 4) создание интер-коммуникатора.

Синтаксис функции создания интер-коммуникатора следующий:

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
MPI_Comm peer_comm, int remote_leader, int tag,

MPI_Comm *newintercomm);

Атрибутами этой функции являются:

in **local_comm** - локальный интра-коммуникатор; in **local_leader** - ранг лидера локальной группы в коммуникаторе **local_comm;**in **peer_comm** - Равноправный коммуникатор; используется только процессом **local_leader; i**n **remote_leader** - ранг лидера удаленной группы в коммуникаторе **peer_comm**; используется только процессом **local_leader;** In **tag** - метка «безопасности»; out **newintercomm** - новый интер-коммуникатор.

Вызов данной функции является коллективным через объединение локальной и удаленной групп. Процессы должны обеспечивать одинаковые аргументы **local_comm** и **local_leader** внутри каждой группы. Групповое значение не разрешено для аргументов **remote_leader**, **local_leader** и **tag**.

Этот вызов использует коммуникацию типа «точка-точка» с коммуникатором **peer_comm** и с меткой **tag**.

Для создания интра-коммуникатора используется следующая функция:

int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
MPI_Comm *newintracomm);

Атрибутами этой функции являются:

in **intercom** - Интер-коммуникатор; in **high** – флаг; out **newintracomm** - новый интра-коммуникатор.

Эта функция создает интра-коммуникатор из объединения двух групп, которые ассоциированы с коммуникатором **intercomm**. Все процессы должны иметь одинаковое значение **high** внутри каждой группы. Если процессы в одной группе указывают значение **false** для аргумента **high**, и процессы в другой указывают значение **true** для аргумента **high**, тогда объединение упорядочивается так, что «нижняя» группа располагает до «верхней» группы. Если все процессы указывают одинаковое значение для аргумента **high**, тогда порядок в объединении является произвольным. Вызов данной функции является блокирующимся и коллективным внутри объединения двух групп.

Рассмотрим пример конвейера трех групп. Группы 0 и 1 являются связанными. Группы 1 и 2 также являются связанными. По этой причине группа 0 требует один интер-коммуникатор, группа 1 требует два интер-коммуникатора, и группа 2 требует один интер-коммуникатор. Код программы следующий:

```
int main(int argc, char **argv)
{
   MPI_Comm myComm;       // интра-коммуникатор локальной подгруппы
   MPI_Comm myFirstComm;   // интер-коммуникатор
   MPI_Comm mySecondComm;  // второй интер-
коммуникатор (группа 1 только)
   int membershipKey;
   int rank;

   MPI_Init(&argc, &argv);
```

```
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // Пользовательский код должен генерировать membershipKey
   // в диапозоне [0, 1, 2]
   membershipKey = rank % 3;
   // Построение интракоммуникатора для локальной подгруппы
MPI_Comm_split(MPI_COMM_WORLD,membershipKey, rank, &myComm);
   // Построение интер-коммуникатора. Метки жестко закодированные
   if(membershipKey == 0)
   {
      // Группа 0 связывается с группой 1
      MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1, 1, &myFirst
Comm);
   }
   else
      if (membershipKey == 1)
      {
         // Группа 1 связывается с группами 0 и 2
     MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0, 1,
            &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2, 12,
            &mySecondComm);
      }
      else
         if (membershipKey == 2)
         {
            // Группа 2 связывается с группа 1
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1, 12,
               &myFirstComm);
         }
   switch(membershipKey)
   {
   case 1:
      MPI_Comm_free(&mySecondComm);
   case 0:
   case 2:
      MPI_Comm_free(&myFirstComm);
      break;
   }
   MPI_Finalize(); }
```

2. ЗАДАНИЕ НА РАБОТУ

Задание выбирается в соответствии с вариантом, назначенным преподавателем.

2.1. Вариант №1

Реализовать блочный алгоритм распределенного параллельного перемножения матриц *A*и *B*с размерами (8*5) и (5*3) соответственно. Вид распределяемых между процессами блоков представлен на рисунке 2.13:

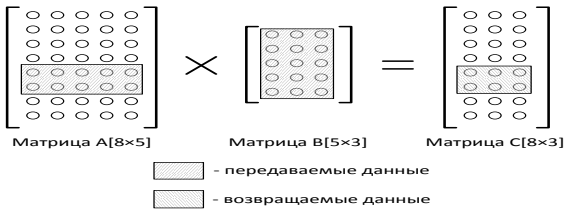


Рисунок 2.1 – Перемножение матриц

Корневой процесс реализует рассылку:

1) блоков матрицы *A* по две строки;
2) широковещательную рассылку элементов матрицы *B* между обрабатывающими процессами внутри своей группы (по умолчанию) – режим **One-To-All**.

Организуется четыре процесса, обрабатывающих данные и формирующих фрагменты (2*3) матрицы результата C. После подготовки блоков матрицы *C*всеми обрабатывающими процессами (взаимная синхронизация функцией **MPI_Barrier**) выполняется совместная передача результатов (блоков матрицы C) корневому процессу – режим **All-To-One.**

2.2. Вариант №2

Требуется выполнить вычисление максимального и минимального значения функции *f(x,y)* внутри некоторой области. Функция *f(x,y)* задана в виде: $f(x,y)=sin(x)+e^y$, а интервалы изменения параметров функции (ее аргументов) определены следующим образом: $x, y \in [0,1]$. Интервал дискретизации для каждого из аргументов –0,1. Тогда по каждому из аргументов получено 10 по значений *f(x,y)*. Полученные значения функции *f(x,y)* сведены в корневом процессе (**root**) в матрицу *A* (количество элементов в матрице A – 100). В программе должны быть реализованы две группы процессов: первая группа процессов выполняет определение минимального значения, вторая группа – максимального, корень первой и второй группы является один и тот же процесс (**root**), в котором выполняется расчет значений функции *f(x,y)* внутри области – формирование матрицы *A*. Для каждой из групп создается свой коммуникатор. Матрица *A* разбита на блоки по 4 элемента (2 строки, 2 столбца) в виде, указанном на рисунке 2.2 (здесь $A_{i,j}$ соответствующая подматрица (блок), передаваемая корневым процессом (*i,j)*-ому процессу в одной и другой группах).

Каждый из процессов в группе получает (в результате обмена с корневым процессом) свою подматрицу (блок) и ожидает взаимной синхронизации

с другими процессами (функция **MPI_Barrier**). После чего каждым из процессов в группе вызывается функция, определяющая минимум (максимум) среди элементов подматрицы (блока) – выполнение вычислений с каждым из блоков реализуется параллельно с вычислениями для других блоков. Далее процессом из группы (для соответствующего блока матрицы $A_{i,j}$) выполняется определение глобального минимума (максимума) внутри каждой их групп (вызов функции **MPI_Reduce**), после чего глобальные значения **min** и **max** записываются в соответствующие переменные корневого процесса с последующим выводом результатов.

2.3. Вариант №3

Требуется выполнить численное определение значения интеграла функции *f(x,y)* на интервале $x[a,b]$. Вид функции следующий: $f(x)=e^x$. Формула для вычисления значения интеграла имеет вид:

$$I = \int_a^b f(x)dx = \sum_{i=1}^{N} a_i,$$

где a_i – частичная сумма, полученная по формуле прямоугольников следующим образом:

$$a_i = \frac{f(x_i) + f(x_{i+1})}{2}.$$

Количество интервалов, на которые разбит интервал интегрирования, соответствует числу параллельно вычисляющих частичные суммы процессов. Порядок реализации поставленной задачи следующий:

1) ввод для процесса корня границ интегрирования *a*, *b*, определение в корневом процессе числа копий задачи (функция **MPI_Comm_size**);

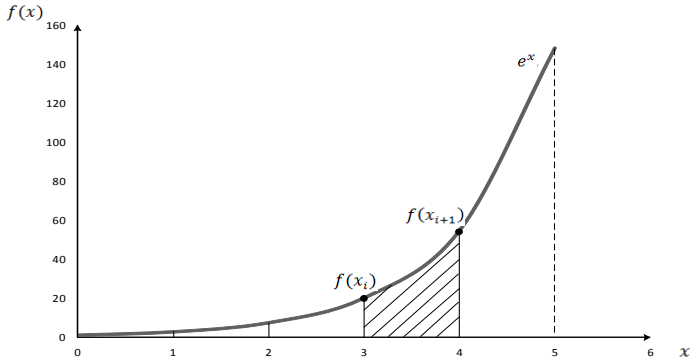


Рисунок 2.15 – Частичная сумма интеграла

2) выполнение широковещательной рассылки из корневого процесса границ интервала интегрирования и числа процессов, выполняющих вычисления частичных сумм (число вычислительных процессов равно общему числу процессов, возвращаемых функцией **MPI_Comm_size - 1** (сам процесс-корень));
3) для каждого процесса определение длины подинтервала нахождения частичной суммы, определение левой и правой границ подинтервала, возможный синтаксис имеет следующий вид:
len = (b – a) / numproc;
local_a = a + my_rank * len;
local_b = local_a + len;
Здесь **my_rank**– ранг соответствующего процесса, возвращаемый функцией **MPI_Comm_rank**;
4) вызов каждого из **numproc**-процессов функции интегрирования, выполняющий определение локального значения частичной суммы по методу средних прямоугольников;
5) полученные локальные результаты каждого из процессов обобщаются (суммируются) функцией **MPI_Reduce** с указанием вида операции (суммирование) и размещением конечного результата в переменной корня;
6) взаимная синхронизация всех вычислительных процессов (функция **MPI_Barrier**).

3. Контрольные вопросы

3.1. Чем коллективные операции отличаются от взаимодействия типа точка-точка?
3.2. Верно ли, что в коллективных взаимодействиях участвуют все процессы приложения?
3.3. Могут ли возникать конфликты между обычными сообщениями, посылаемыми процессами друг другу, и сообщениями коллективных операций? Если да, как они разрешаются?
3.4. Можно ли при помощи процедуры **MPI_Recv** принять сообщение, посланное процедурой **MPI_Bcast**?
3.5. В чем различие в функциональности процедур **MPI_Cast** и **MPI_Scatter**?

ЛАБОРАТОРНАЯ РАБОТА №3
ИСЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ ФОРМИРОВАНИЯ ВИРТУАЛЬНЫХ ТОПОЛОГИЙ ВЫЧИСЛИТЕЛЬНЫХ КЛАСТЕРОВ

ЦЕЛЬ РАБОТЫ: исследовать возможности, предоставляемые MPI по формированию виртуальных топологий.

1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Топология является дополнительным необязательным атрибутом, который может соответствовать интра-коммуникатору; топологии не могут быть добавлены к интер-коммуникатору. Виртуальная топология вычислительного кластера – это удобный механизм именования для процессов группы внутри коммуникатора. Как было сказано ранее, группа в MPI является коллекцией n процессов. Каждый процесс в группе имеет ранг от **0** до $n-1$. Во многих параллельных приложениях линейное ранжирование процессов недостаточно отображает логическую коммуникационную модель взаимодействия процессов (которая обычно определяется проблемой геометрии топологии и используемым алгоритмом нумерации). Часто процессы организуются в топологические модели, такие как двух- или трех-мерные сетки. В общем виде логическая организация процессов описывается графом. Такая логическая организация процессов называется «виртуальной топологией». Существует четкое различие между виртуально топологией процессов и топологией физической аппаратуры. Виртуальная топология может быть использована системой для распределения процессов на физических процессорах, если это помогает улучшить коммуникационную производительность на данной машине. Описание виртуальной топологии, с другой стороны, зависит только от приложения и является машинно-независимой.

1.1. Виртуальные топологии

Коммуникационная модель взаимодействия процессов может быть представлена в виде графа. Узлы представляют собой процессы, ребра соединяют процессы, которые взаимодействуют друг с другом. MPI обеспечивает передачу сообщений между любой парой процессов в группе. После создания связей между процессами (в виртуальной топологии) отсутствуют специальные условия для открытия канала, поэтому недостающее ребро в определённом пользователем графе не запрещает соответствующую передачу между процессами. Данное состояние говорит только о том, что связь является игнорируемой в виртуальной топологии. Такая стратегия предполагает, что топология определяет способ именования путей коммуникации. Указание виртуальной топологии в виде графа является достаточным для всех приложений. Большая часть всех параллельных приложений использует топологию процессов, таких как кольцо, сетку с двумя или более измерениями, или торы. Эти структуры являются совершенно различными по количеству измерений и количеству процессов в каждом координатном направлении. Координаты процесса в координатной структуре начинают свое исчисление с **0**. Из-

мерение по строкам является всегда более важным в координатной структуре. Это значит что отношение между рангами групп и координатами для четырех процессов в сетке (2*2) является следующим:

координата (0, 0): ранг 0
координата (0, 1): ранг 1
координата (1, 0): ранг 2
координата (1, 1): ранг 3

1.2. Конструкторы топологий

Для создания декартовой топологии используется функция:

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods,int reorder, MPI_Comm *comm_cart);

Атрибутами этой функции являются:

in **comm_old** - входной коммуникатор; in **ndims** - количество измерений декартовой сетки; in **dims** - целочисленный массив размера **ndims**, указывающий количество процессов в каждом измерении; in **periods** - логический массив размера **ndims**, указывающий является ли сетка периодической (**true**) или нет (**false**) в каждом измерении; in **reorder** - ранжирование может быть переупорядочено (**true**) или нет (**false**); out **comm_cart** - коммуникатор с новой декартовой топологией.

Функция возвращает дескриптор нового коммуникатора, к которому прикреплена информация о декартовой топологии. Если аргумент **reorder** равен значению **false**, тогда ранг каждого процесса в новой группе является идентичным его рангу в старой группе. В противном случае функция может переупорядочить процессы. Если общий размер декартовой сетки является меньше, чем размер группы коммуникатора **comm**, тогда некоторые процессы возвращаются как **MPI_COMM_NULL**, по аналогии с функцией **MPI_Comm_split**. Если аргумент **ndims** равен нулю, тогда создается декартовая нуль-мерная топология. Вызов является ошибочным, если он определяет сетку больше чем размер группы, или если аргумент **ndims** является отрицательным. Для декартовых топологий функция **MPI_Dims_create** помогает пользователю выбрать сбалансированное распределение процессов по каждому координатному направлению в зависимости от количества процессов в группе и от дополнительных ограничений, которые могут быть указаны пользователем. Одним из использований является разбитие всех процессов на **n**-мерную топологию. Вид вызова функции следующий:

int MPI_Dims_create(int nnodes, int ndims, int *dims)

Атрибутами этой функции являются:

in **nnodes** - количество узлов в сетке; in **ndims** - количество декартовых измерений; inout **dims** - целочисленный массив размера **ndims** указывающий количество узлов в каждом измерении;

Элементы в массиве **dims** определяются для описания декартовой сетки с **ndims** измерениями и суммой **nnodes** узлов. Измерения устанавливаются так, чтобы быть как можно ближе друг к другу, используя подходящий алго-

ритм делимости. Если элемент **dims[i]**является положительным числом, функция не будет модифицировать количество узлов в измерении **i**; только те элементы, где значение **dims[i] = 0**, будут модифицированы вызывающей стороной.

Отрицательные входные значения **dims[i]** являются ошибочными. Ошибка будет происходить, если аргумент **nnodes** не является произведением:

$$\prod_{i,dims\,[i]\neq 0} dims[i].$$

Функция **MPI_Dims_create** является локальной.

Таблица 1.1

Пример использования функции **MPI_Dims_create**

Dims до вызова	вызов функции	**Dims** на выходе
(0,0)	**MPI_Dims_create(6,2,dims)**	(3, 2)
(0, 0)	**MPI_Dims_create(7,2,dims)**	(7, 1)
(0, 3, 0)	**MPI_Dims_Create(6,3,dims)**	(2, 3, 1)
(0, 3, 0)	**MPI_Dims_create(7,3,dims)**	Ошиб. вызов

Для создания топологии графа используется следующая функция:
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph);

Атрибутами этой функции являются:
in **comm_old** - входной коммуникатор; in **nnodes** - количество узлов в графе; in **index** - целочисленный массив, описывающий степени узлов; in **edges** - целочисленный массив, описывающий ребра графа; in **reorder** - ранжирование может быть переупорядочено (**true**) или нет (**false**); out **comm_graph** - коммуникатор с добавленной топологией графа.

Функция возвращает дескриптор нового коммуникатора, к которому прикреплена информация о топологии графа. Если аргумент **reorder** равен значению **false**, тогда ранг каждого процесса в новой группе является идентичным его рангу в старой группе. Если размер (аргумент **nnodes)** графа меньше чем размер группы коммуникатора **comm**, тогда некоторые процессы будут возвращены как **MPI_COMM_NULL**, по аналогии с **MPI_Cart_create**. Если граф пустой, т.е. аргумент **nnodes = 0**, тогда значение **MPI_COMM_NULL** возвращается во всех процессах. Вызов является ошибочным, если он указывает граф, который является больше, чем размер группы входного коммуникатора. Три параметра **nnodes**, **index** и **edges** определяют структуру графа. Аргумент **nnodes** является количеством узлов графа. Узлы нумеруются от **0** до **nnodes-1**. При этом ***i***-ый элемент массива **index** сохраняет общее число соседей первых ***i*** узлов графа. Список соседей узлов **0**, **1**, …, **nnodes-1** сохраняется в последовательности массива **edges**. Массив **edges** является развернутым представление списка ребер. Общее количество

элементов в аргументе **index** является равным **nnodes,**общее количество элементов в аргументе **edges** является равным количеству ребер графа.

Пример: Предположим, что существует четыре процесса 0, 1, 2, 3 со следующей матрицей смежности:

процессы	соседи
0	1, 3
1	0
2	3
3	0, 2

Тогда, входные аргументы следующие:

nnodes = 4

index = 2, 3, 4, 6

edges = 1, 3, 0, 3, 0, 2

Поэтому в С элемент **index[0]** является степенью нулевого узла, а **index[i] – index[i-1]** является степенью узла **i**, где **i=1**, …, **nnodes-1**; список соседей нулевого узла сохраняется в элементе **edges[j]**, для **0≤ j ≤ index[0]-1** исписок соседей узлов **i (**где**i>0)**, сохраняются в элементе **edges[j]**, для**index[i-1] ≤ j≤ index[i]-1**.

1.3. Реализация топологических запросов

Если топология была определена одной из предыдущих функции, тогда информация о топологии может быть просмотрена, используя функции запросов, которые являются локальными вызовами:

intMPI_Topo_test(MPI_Commcomm, int *status) **Атрибутами этой функции являются:**

in **comm** – коммуникатор; out **status** - тип топологии коммуникатора **comm;**

Функция возвращает тип топологии, которая назначена коммуникатору. Выходное значение аргумента **status** является одним из следующих:

MPI_GRAPH	топология графа
MPI_CART	декартова топология
MPI_DIST_GRAPH	распределенная топология графа
MPI_UNDEFINED	топологии нет

Функции **MPI_Graphdims_get** и **MPI_Graph_get** получают информацию о топологии графа, которая была ассоциирована с коммуникатором через **MPI_Cart_create**:

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges);

Атрибутами этой функции являются:

in **comm** - коммуникатор с топологией графа; out **nnodes** - количество узлов в графе; out **nedges** -количество ребер в графе;

Функции **MPI_Cartdim_get** и **MPI_Cart_get** возвращают информацию о декартовой топологии, которая была ассоциирована с коммуникатором, используя функцию **MPI_Cart_create**:

int MPI_Cartdim_get(MPI_Comm comm, int *ndims);

Атрибутами этой функции являются:
in **comm** - коммуникатор с декартовой структурой; out **ndims** - количество измерений декартовой структуры.

Если аргумент **comm** ассоциирован с нуль-мерной декартовой топологией, функция **MPI_Cartdim_get** возвратит аргумент **ndims=0**, функция **MPI_Cart_get** при этом оставит все выходные аргументы неизмененными:
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,int *coords). **Атрибутами этой функции являются:**
in **comm** - коммуникатор с декартовой структурой; in **maxdims** - длина вектора **dims**, **period** и **cords** в вызывающей программе; out **dims** - количество процессов для каждого декартового измерения; out **periods** - периодичность для каждого декартового измерения; out **cords** - координаты вызывающего процесса в декартовой структуре;

Для того, чтобы на основе координат получить ранг процесса, используется следующая функция:
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
Атрибутами этой функции являются:
in **comm** - коммуникатор с декартовой структурой; in **cords** - целочисленный массив (размера **ndims**) определяющий декартовы координаты процесса; out **rank** - ранг указанного процесса;

Для группы процессов с декартовой структурой функция **MPI_Cart_rank** переведет логические координаты процесса в ранг процесса, как они использовались бы процедурами точка-точка.
Для обратного отображения, перевод ранга в координаты, используется функция, синтаксис которой следующий:
Int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int *coords);
Атрибутами этой функции являются:
in **comm** - коммуникатор с декартовой структурой; in **rank** - ранг процесса внутри группы **comm; i**n **maxdims** - длина вектора **cords** в вызывающей программе; out **cords** - целочисленный массив (размера **ndims**) содержащий декартовы координаты указанного процесса.

Если аргумент **comm** ассоциирован снуль мерной декартовой топологией, аргумент **cords** будет неизменен.

Функции **MPI_Graph_neighbors_count** и **MPI_Graph_neighbors** обеспечивают информацию о смежности для общей топологии графа:
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors);
Атрибутами этой функции являются:
in **comm** - коммуникатор с топологией графа; in **rank** - ранг процесса в группе **comm;** out **nneighbors** - количество соседей указанного процесса.
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors);
Атрибутами этой функции являются:

in comm - коммуникатор с топологией графа; in rank - ранг процесса в группе **comm;** in maxneighbors -размер массива соседей; out neighbors - ранги процессов, которые являются соседями указанного процесса;

Возвращаемое количество и массив соседей для запрашиваемого ранга будет как включать всех соседей, так и отражать такой же порядок ребер, как было определено оригинальным вызовом функции **MPI_Graph_create.** Точнее данные функции будут возвращать значения, соответствующие аргументам **index** и **edges**, переданным в функцию **MPI_Graph_create.** Количество соседей возвращенное функцией **MPI_Graph_neighbors_count** будет равно **index[rank]–index[rank–1]**. Массив **neighbors**, возвращенный функцией **MPI_Graph_neighbors** будет от **edges[index[rank – 1]]** до **edges[index[rank – 1]]**.

2. Задание на работу

Задание выбирается в соответствии с вариантом, назначенным преподавателем. Алгоритмы перемножения матриц, которые необходимо реализовать в вариантах, находятся в приложении В.

2.1. Вариант №1

Необходимо реализовать алгоритм перемножения матриц ленточным способом с распределением столбцов.

2.2. Вариант №2

Необходимо реализовать алгоритм перемножения матриц ленточным способом с распределением строк.

2.3. Вариант №3

Необходимо реализовать алгоритм перемножения матриц по методу Фокса.

3. Контрольные вопросы

3.1.Обязана ли виртуальная топология повторять физическую топологию целевого компьютера?

3.2.Любой ли коммуникатор может обладать виртуальной топологией?

3.3.Может ли процесс входить одновременно в декартову топологию и в топологию графа?

3.4.Как определить, с какими процессами в топологии графа связан данный процесс?

ЛАБОРАТОРНАЯ РАБОТА №4
ИССЛЕДОВАНИЕ ВЗАИМОДЕЙСТВИЙ РАСПРЕДЕЛЕННЫХ ПРОЦЕССОВ ТИПА «КЛИЕНТ-СЕРВЕР»

Цель работы: исследовать механизм взаимодействия распределено выполняющихся параллельных процессов типа «клиент-сервер».

1. ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ.

Одной из типичных схем взаимодействия между процессами в параллельных (распределенных) программах является взаимосвязь «клиент-сервер». Процесс-клиент при этом запрашивает некоторый сервис, затем ожидает обработку запроса. Процесс-сервер многократно ожидает запрос (на использование ресурса), обрабатывает его и посылает ответ. При этом существует двунаправленный поток информации от клиента к серверу и обратно. Таким образом, сервер – это процесс, постоянно обрабатывающий запросы от клиентских процессов, используя при этом асинхронную передачу сообщений (рис 1.1).

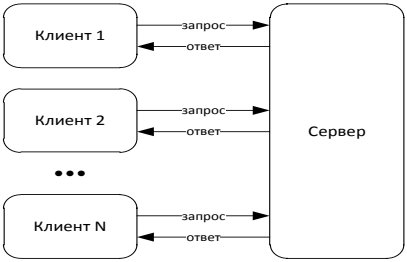


Рисунок 1.1 – Схема взаимодействия типа «клиент-сервер»

В распределенных системах (системах с распределенной памятью) сервер реализуется набором подпрограмм, обеспечивающих доступ к ресурсу или структурам данных, представляющих собой ресурс.

Взаимодействие между клиентом и сервером реализуется вызовом соответствующих процедур доступа к ресурсу.

Серверы могут быть реализованы в соответствии со следующими схемами управления:

1) с помощью рассылки сообщений;
2) с использованием рандеву.

1.1. Процедура обмена сообщениями при взаимодействии клиента и сервера и алгоритмы функционирования сервера

Взаимодействие клиента и сервера с помощью рассылки сообщений при выделении ресурсов инициируется следующим образом: клиентский процесс отправляет сообщение в канал запроса, а затем получает ответ (ре-

зультат) из канала ответа. При этом каждому клиенту выделяется собственный канал ответа.

В структуре сообщения должно быть определено, какую опцию вызывает клиент (запрос или освобождение ресурса). Необходимо также передавать аргументы сообщения (номер выделяемого ресурса и его запрашиваемое количество, если ресурс выделяется не целиком, а по частям). Когда нет доступных элементов ресурса, сервер не может ожидать, обслуживая запрос с выделением ресурса. Он запоминает запрос и откладывает посылку ответа (тем самым блокируя клиента). Когда ресурс освобождается, сервер возвращается к сохраненному запросу и передает освободившийся элемент запрашивающему процессу. После отправки сообщения с запросом на выделение ресурса клиент ждет получения элемента ресурса, освобождая ресурс, клиент не ждет подтверждения его освобождения. Серверу приходится сохранять ожидающий запрос, а также впоследствии проверять очередь ожидающих процессов. Обобщенные алгоритмы функционирования клиента и сервера представлены на рис 1.2 и 1.3.

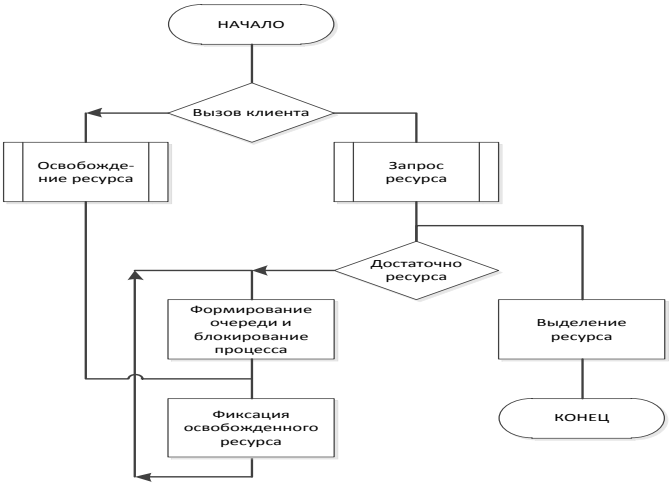


Рисунок 2.2 – Алгоритм функционирования сервера

При взаимодействии клиента и сервера при разделении ресурсов осуществляется обмен между ними тремя сообщениями (запрос, подтверждение, освобождение) и выделение ресурса клиенту, если он свободен.

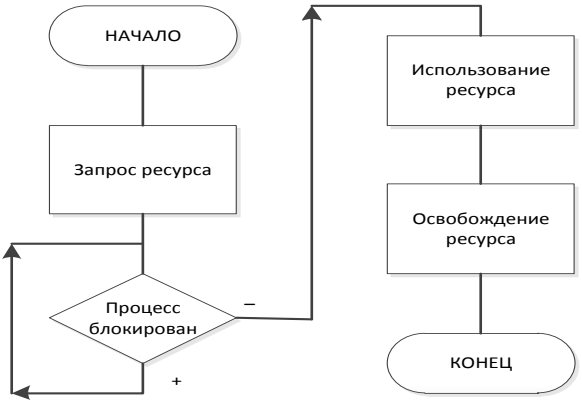


Рисунок 1.3 – Алгоритм функционирования клиента

1.2. Реализация концепции рандеву при функционировании сервера

Реализация рандеву предполагает инициализацию выполнения некоторой процедуры, доступ к которой регулируется сервером, в ответ на запрос клиента. Действия этой процедуры и клиентов необходимо синхронизировать (клиенты-поставщики информации для обработки этой процедурой).

Процедуре и клиенту необходимо рандеву (взаимодействие), которое синхронизируется сервером. Возможно отличие от первой схемы, когда клиент непосредственно не взаимодействует с процедурой, запрашивает ресурс (обращаясь к серверу), ждет от сервера результатов обработки, сообщение о завершении этой обработки отсылает серверу сама вычислительная процедура (совместно с результатами). Как и в описанном выше случае, сервер обслуживает любой процесс, запросивший ресурс, клиент должен ждать освобождения ресурса. Таким образом, существует два способа решения задачи диспетчеризации при организации серверов:

1) отделение сервера от ресурса; в этом случае клиент осуществляет вызов сервера для получения доступа к ресурсу (запрос ресурса) и в случае получения доступа непосредственно обращается к ресурсу; В этом случае используется пять сообщений (рис 1.4).

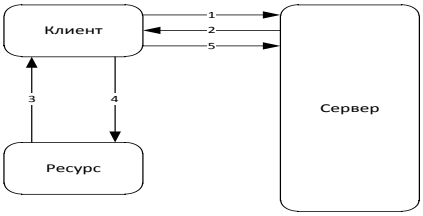


Рисунок 1.4 – Схема взаимодействия сервера, клиента и ресурса

2) сервер является посредником между клиентом и ресурсом (процессом, являющимся ресурсом); клиенты вызывают сервер, указывая в нем требуемый вид обработки ресурса, далее сервер сам обращается к ресурсу, разграничивая доступ клиентов; в данном случае инициируется передача 4 сообщений (рис 1.5).

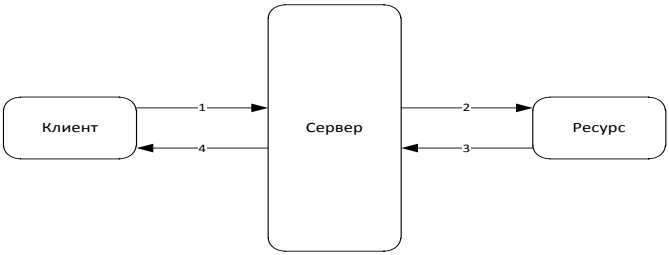


Рисунок 1.5 – Схема сервера-посредника между клиентом и ресурсом

1.3. Информационные структуры сервера

В случае, если ресурс представляет собой совокупность дискретных элементов и выделяется клиентом не целиком, а по частям, сервером должны быть использованы следующие информационные структуры:

1) таблица ресурсов, каждый элемент которой должен содержать поля (рис 1.6), где $V_{св}$– количество свободного ресурса, которое сравнивается с количеством требуемого клиенту ресурса ($V_{тр}$), передаваемом в запросе; в случае превышения свободного количества над требуемым ресурс выделяется в использовании клиентом; в противном случае клиент блокируется; при выделении ресурса клиенту вносятся изменения в количество свободного ресурса.

№ ресурса	Информация	
	Количество свободного ресурса $V_{св}$	Ссылка на очередь клиентов

Рисунок 1.6. – Таблица ресурсов сервера

2) очередь ожидающих данный ресурс клиентов; она может представлять собой массив структур, каждая строка которого соответствует номеру ресурса, а каждый элемент, которого имеет вид, представленный на рис 1.7.

Идентификатор процесса	Требуемое количество ресурса

Рисунок 1.7 – Элемент очереди доступа к ресурсу

В этом случае действия сервера при выполнении запроса и освобождения ресурсов могут быть определены в виде фрагмента алгоритма, представленного на рис 1.8.

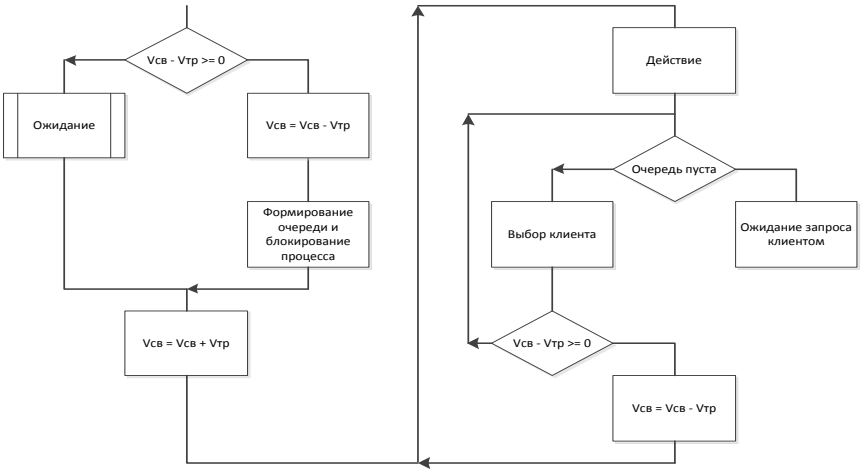


Рисунок 1.8 –Фрагмент алгоритма контроля информационных структур сервера при запросе и освобождении ресурса

2. Задание на работу

Задание выбирается в соответствии с вариантом, назначенным преподавателем

2.1. Вариант №1

В состав вычислительного кластера входит три хоста, один из которых реализует функции сервера, два остальных – клиентов.

Сервер разграничивает доступ к трем общим ресурсам – переменным, хранящим общую вырученную сумму от продажи товаров, общее количество товаров и остальных товаров. Доступ к ресурсам осуществляется в произвольном порядке, все ресурсы разделяются между клиентами по отдельности. Реализована процедура выделяющая ресурсы (путем передачи сообщения) в использование клиентам. Реализовать серверный процесс, который разграничивает доступ клиентов к этой процедуре (процедурам) и к ресурсам. Реализацию сервера выполнять в соответствии со схемой управления, использующую рассылку сообщений.

2.2. Вариант №2

Сервер разграничивает доступ двух клиентов к общей процедуре. Клиент запрашивает у сервера доступ к процедуре и передает ей данные для расчета. Процедура выполняет простейшие операции суммирования введенных элементов.

При выполнении задания в основу построения сервера положить концепцию рандеву с оповещением процедурой сервера об ее освобождении при обработке данных. Реализовать две схемы построения сервера с отделением

сервера от ресурса и сервера посредника между клиентом и ресурсом. Результаты работы процедуры возвращаются в вызванные клиентские процессы.

2.3. Вариант №3

Сервер разграничивает доступ со стороны двух клиентов к общей базе данных (массиву структур). При этом указанный ресурс рассматривается как дискретный, т.е. по запросу клиента ему может быть выделено столько записей, сколько необходимо. Информация из БД считывается обрабатывающей программой, доступ которой к БД также разграничивается сервером. Общее количество записей в БД равно N. Для синхронизации доступа к БД используются целые переменные: P1 и P2 – определяющие первую занятую ячейку и первую пустую ячейку, P3 – число заполненных ячеек. Возможно использование следующих сигналов: запрос требуемого количества ресурса, освобождение ресурса после считывания, "буфер пуст" для потребителя и "буфер полон" для производителя. При построении сервера реализовать концепцию передачи сообщений. При реализации отделить сервер от ресурса, а также реализовать сервер как посредник между клиентами и ресурсом.

3. Контрольные вопросы.

3.1. Определить назначение серверного процесса при взаимодействии с клиентами.
3.2. Охарактеризовать схемы управления при реализации сервера, содержание и особенности этих схем управления.
3.3. Сформулировать обобщенный алгоритм функционирования сервера.
3.4. Определить структуру сообщений при обмене между клиентом и сервером.
3.6. Охарактеризвать информационные структуры сервера при разделении доступа к дискретным ресурсам.

ЛАБОРАТОРНАЯ РАБОТА № 5
ИССЛЕДОВАНИЕ МОДЕЛЕЙ ВЗАИМОДЕЙСТВИЯ РАСПРЕДЕЛЕННО ВЫПОЛНЯЮЩИХСЯ ПРОЦЕССОВ

Цель работы: исследовать алгоритмическое построение методов взаимодействия распределено выполняющихся процессов**.**

1. Теоретическое введение.

Наряду с известными моделями взаимодействия распределенных процессов (взаимодействие «производитель-потребитель», «взаимодействие равных», «клиент-сервер») существует целый ряд важных моделей, к которым могут быть отнесены следующие:

1) модель «зонд-эхо»;
2) модель «распределенны семафоров»;

3) модель передачи маркера.

1.1. **Модель «зонд-эхо»**

Топология вычислительного кластера (топология определяет взаимодействие вычислительных процессов), построенного для распределенных вычислений, может быть представлена в виде графа с узлами – процессами, и ребрами – каналами передачи данных. «Зонд» - это сообщение, передаваемое узлами графа своему преемнику, «эхо» - это последующий ответ преемника родительскому узлу. Модель «зонд-эхо» используется как для построения топологии кластера, так и для широковещательной рассылки сообщений по всем узлам (процессам) кластера (процесс, выполняющийся на узле S должен разослать сообщения процессам, выполняющимся на всех остальных узлах).

Основным требованием при широковещательной рассылке является необходимость того, чтобы сообщение попадало на каждый узел (в каждый процесс) только один раз. Для обеспечения этого требования узел, выполняющий рассылку, должен иметь два специальных средства, обеспечивающих эту рассылку – это информация о топологии кластера (взаимодействии распределено выполняющихся процессов) и об его (кластера) **остовном** дереве. Топология может быть представлена в виде симметричной матрицы, где элемент i-ой строки и j-го столбца равен 1, если i-ый и j-ый узлы связаны между собой каналами передачи данных, в противном случае – 0.

Топология строится на основе выполнения широковещательных рассылок между узлами кластера эхо-запросов и получения эхо-ответов (построение топологии рассматривается далее). Для эффективной рассылки узел должен построить и использовать остовное дерево. Корнем остовного дерева является узел, инициирующий рассылку. Остовное дерево отличается от топологии сети тем, что в нем исключены ребра топологии, дублирующие связи между узлами. Отличие остовного дерева от топологии кластера комментирует рис. 1.1.

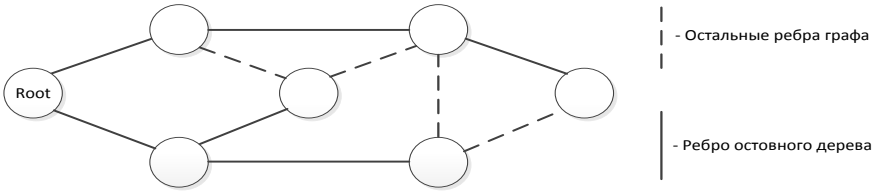


Рисунок 1.1 – Остовное дерево кластера

Остовное дерево интерпретируется соответствующей матрицей, построенной на основе матрицы топологии, из которой исключены единичные элементы, **дублирующие связи между узлами**.

Процесс рассылки широковещательных сообщений выполняется корневым узлом остовного дерева (т.е. процессом, реализующим широковеща-

тельную рассылку). При этом между узлами (процессами) кластера первоначально выполняется рассылка остовного дерева. Рассылка остовного дерева (предваряющая рассылку широковещательных сообщений) необходима для того, чтобы другие узлы топологии знали, куда им необходимо далее пересылать сообщение.

Пересылка сообщения между узлами остовного дерева осуществляется с использованием уже существующих каналов передачи данных, созданных предварительно для обмена сообщениями между процессами. Построению остовного дерева предшествует процедура построения топологии кластера, которая заключается в следующем. Каждому узлу известна лишь его локальная топология (т.е. связи с его соседями). Задача состоит в том, чтобы собрать воедино все локальные топологии и построить общую топологию кластера. Агоритм построения топологии кластера состоят в следующем:

1) каждый узел (начиная с инициатора рассылки) посылает сообщение-зонд (запрос на топологию) своим соседям. В зонде указывается источник этого зонда (Рис. 1.2);

Тип сообщения (зонд)	Идентификатор источника

Рисунок 1.2 – Формат зонда

2) узлы-преемники, получив зонд, ретранслируют его далее, но не посылают эхо-ответ до тех пор, пока не придёт эхо-ответ от их преемников (при ретрансляции указывается уже другой источник зонда);

3) так как источник зонда (начальный или последующий) знает, сколько у него каналов рассылки зондов, из них он и будет ожидать возврат эхо-ответов;

4) процесс рассылки зонда продолжается до тех пор, пока он не достигнет узлов, не имеющих соседей, либо, если топология циклическая, на узел не придет пустой эхо-ответ, что определяет отсутствие необходимости дальнейшей ретрансляции зонда;

5) в топологии с циклами, если узел получает последующие зонды во время ожидания эхо-ответа, он сразу отсылает пустой эхо-ответ на тот родительский узел, который послал этот зонд;

6) получив эхо от соседних узлов, узел объединяет эти ответы со своим множеством соседей и отсылает это сообщение родительскому узлу; таким образом, суммарное сообщение должно содержать некоторую часть топологии сети, идентифицируемую массивом топологии (его часть заполняется узлами по мере движения снизу вверх по сети)(Рис. 1.3);

Тип сообщения (эхо)	Источник ответа	Массив топологии

Рисунок 1.3 – Формат эхо-ответов

7) инициировавший рассылку узел обобщает эхо-ответы, поступившие от соседей, строит дерево смежности, остовное дерево и пересылает широковещательно сообщение по сети.

1.2. Модель «распределенные семафоры»

Данная модель позволяет осуществлять разделение общего ресурса между распределенными процессами при их выполнении. При реализации механизма распределенных семафоров можно не только разграничивать доступ к ресурсу, но и формировать очередь процессов, ожидающих этого доступа. Возможность организации очереди связана с понятием логических часов. Логические часы – это целочисленный счетчик, увеличивающийся при реализации события. У каждого процесса есть свой логический счетчик, имеющий нулевое начальное значение. В каждом передаваемом сообщении, используемом в алгоритме распределенных семафоров, выделено поле для временной метки. Изменение значения счетчика осуществляется в соответствии со следующими правилами:

1) передавая сообщение, процесс присваивает его метке текущее значение логических часов (переменная lc (localclock)) и увеличивает lc на 1;

2) получая сообщение с меткой времени ts, процесс присваивает своей переменной lc значение, максимальное из lc и $ts + 1$, затем увеличивает свою переменную lc на 1.

По аналогии со стандартными семафорами распределенные семафоры оперируют с P- и V- операциями. При синхронизации доступа к ресурсу процессы обмениваются сообщениями требуемого формата (Рис.1.4).

Идентификатор процесса	Дескриптор типа операции (P-или V-)	Метка времени

Рисунок 1.4 – Формат сообщения при синхронизации доступа к ресурсу

При рассылке сообщений, процесс их генерирующий, сохраняет их временные метки. Так как очереди P-операций всеми процессами обрабатываются одновременно, каждый процесс знает, когда он может занять общий ресурс (т.е. когда в очереди обрабатывается именно его операция P).

Алгоритм синхронизации процессов с помощью распределенных семафоров состоит из следующих шагов:

1) при необходимости доступа к ресурсу процесс рассылает всем остальным процессам сообщение с дескриптором P-операции и меткой времени; при этом процесс источник сообщения с P-операцией размещает в очереди процессов к ресурсу свой идентификатор, с которым связана соответствующая временная метка; отсылка сообщения приводит к изменению значения логических часов источника.

2) принятие широковещательного сообщения (с P-операцией) приводит к изменению логических часов на всех хостах (за исключением источника); одновременно идентификатор процесса, запросившего ресурс (с соответствующей временной меткой) устанавливается в общую очередь процессов на

обработку этим ресурсом; очередь сообщений должна быть упорядочена в соответствии с временной меткой; процессы, принявшие широковещательные сообщения с P-операцией, подтверждают его принятие, отсылая источнику сообщение ask, которое не изменяет временных меток ни на одном из хостов;

3) приняв от каждого из процессов кластера ask-сообщения, процесс-инициатер захвата ресурса обращается к своей локальной очереди идентификаторов процессов на обработку ресурсом, аналогично выполнив передачу сообщения ask, хосты кластера также обрабатывают очередь идентификаторов процессов на доступ к ресурсу; из этой очереди каждым процессом извлекается первый идентификатор (из головы очереди); тот процесс, идентификатор которого извлечен из очереди (первый процесс в очереди) выполняет обращение к ресурсу, все остальные процессы удаляют этот идентификатор из своих очередей; для синхронизации доступа на каждом хосте выделяется локальная переменная S (переменная семафор), определяющая доступность ресурса; при захвате ресурса операции с семафором S аналогичны модели с общей памятью, при этом обращение к очереди процессов выполняется только в случае S=1 (первоначальная не занятость ресурса и освобождение ресурса при выполнении V-операции);

4) если переменная $S = 1$, то всеми хостами выполняется P-операция (соответствующая первому в очереди идентификатору процессов), которая переводит значение S в 0; идентификатор процесса, соответствующий этой операции, удаляется из очереди; при $S = 0$, процесс доступа к ресурсу не получает; он может получить доступ к ресурсу только в случае выполнения V-операции процессом, освобождающим ресурс;

5) если процесс освобождает ресурс, он формирует сообщение с V-операцией, которое подтверждается всеми принявшими его процессами (сообщения ask), после чего каждый процесс изменяет свое локальное значение « распределенного» семафора *S= S+1*; после этого каждый из распределенных процессов обращается к своей локальной очереди и если она не пуста, то выполняет ее обработку с выделением ресурса первому в этой очереди процессу.

Реализация алгоритма взаимодействия процессов с использованием распределенных семафоров рассмотрена на Рис.1.5-1.13.

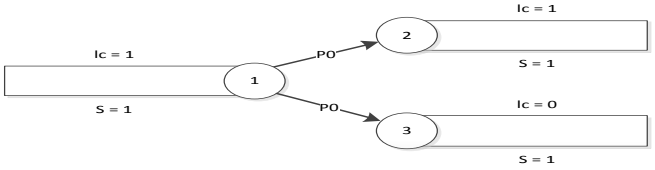


Рисунок 1.5 – Запрос захвата ресурса первым процессом

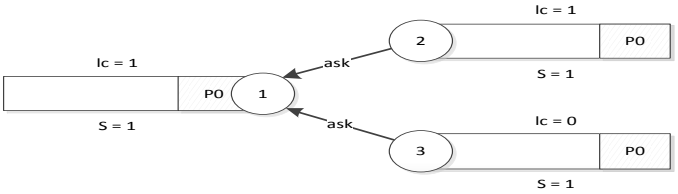


Рисунок 1.6 – Размещение сообщения P0 в очереди и отсылка подтверждающих сообщений ask(логические часы при принятии P-сообщения увеличиваются на 1)

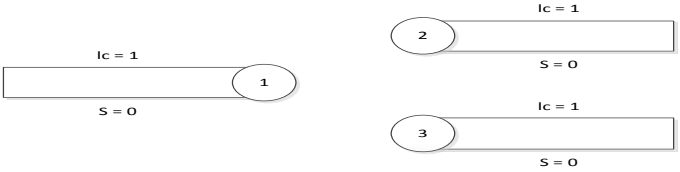


Рисунок 1.7 – Обработка очереди (P-операция), удаление P-операции из очереди, изменение значения переменной S

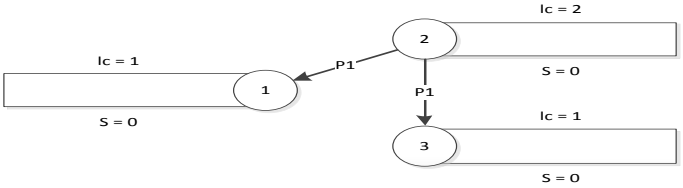


Рисунок 1.8 – Запрос вторым процессом ресурса. Отправка сообщений с идентификатором P-операции.

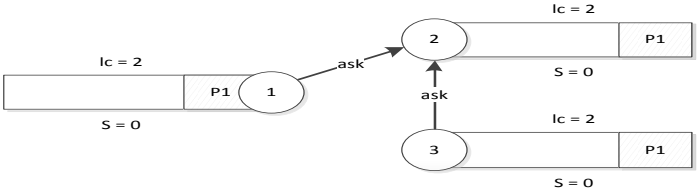


Рисунок 1.9 – Размещение полученного сообщения в очереди, подтверждение приемниками получения сообщения сигналом ask.

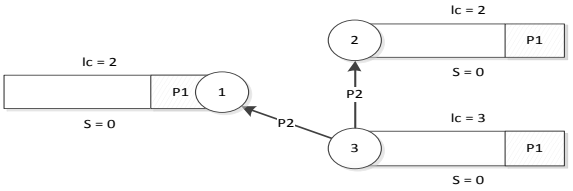


Рисунок 1.10 – Запрос третьим процессом ресурса. Отправка сообщения с идентификатором P-операции.

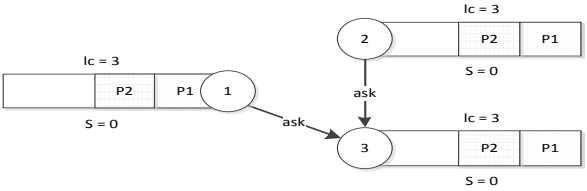


Рисунок 1.11 – Размещение P-сообщения в очереди. Подтверждение принятия P-сообщения

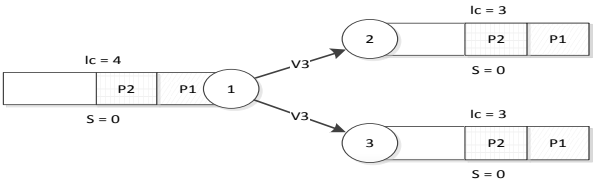


Рисунок 1.12 – Обработка третьим процессом значения переменной S. Посылка первым процессом сообщения V об освобождении ресурса.

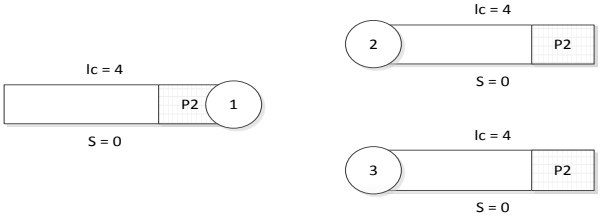


Рисунок 1.13 – Обработка очереди

Происходит изменение значения переменной S на 1, извлечениеоперации P1 (соответствующей второму процессу) из очереди, изменение переменной S на значение 0. Разрешение доступа процессу, сгенерировавшему

операцию P (второй процесс) доступа к ресурсу. Удаление операции P1, соответствующей второму процессу, использующему ресурс из очереди.

1.3. Модель передачи маркера

Алгоритм передачи маркера предназначен для синхронизации доступа распределенно выполняющихся процессов к общему ресурсу. Маркер – это сообщение специального вида, которое предназначено для передачи разрешения на доступ к ресурсу (маркер– сообщение, передача которого соответствует возможности доступа к общему ресурсу определенного процесса).

С каждым из распределено выполняющихся процессов, реализующих вычисления (процесс-*user)*, сопоставлен специальный вспомогательный процесс, реализующий передачу маркера и синхронизирующий доступ к ресурсу (процесс-*helper*). Вспомогательные процессы образуют кольцо, т.е. первый процесс передает маркер второму процессу, он – третьему и т.д., N-ый процесс передает первому процессу. Алгоритм распределения ресурса с использованием передачи маркера комментирует рис. 1.14.

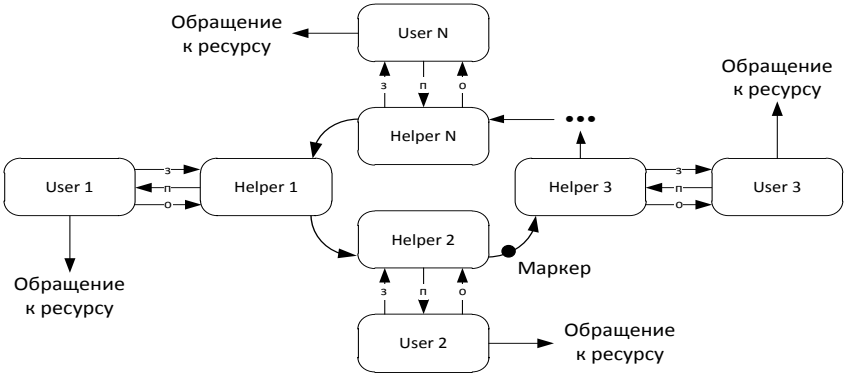


Рисунок 1.14 – Реализации алгоритма передачи маркера

На рис 1.14 использованы следующие обозначения:

- User – вычислительный процесс,
- Helper – вспомогательный процесс,
- З – запрос ресурса,
- П – подтверждение возможности использования ресурса,
- О – сигнал освобождения ресурса.

Алгоритм управления доступом процессов к ресурсу с использованием передачи маркера состоит из следующих шагов:

1) приняв маркер, процесс **`Helper[i]`** проверяет наличие запроса от вычислительного процесса на доступа к ресурсу; если процесс **`User[i]`** выполнил запрос, то процесс **`Helper[i]`** далее маркер не передает, выдает процес-

су **User[i]** разрешение на использование ресурса и ждет от него сигнала об освобождении;
2) в том случае, если процесс **User[i]** не осуществил запрос на использование ресурса, процесс **Helper[i]** передает маркер (т.е сообщение без аргумента) процессу **Helper[i+1]**.

Таким образом, реализуется механизм исключительного доступа к ресурсу лишь одного из группы процессов с использованием единственного сообщения маркера.

2. Задание на работу.

2.1. Вариант №1

Осуществить построение топологии кластера требуемого вида (рис. 2.1); выполнить широковещательную рассылку вводимого с клавиатуры сообщения от узла S на все остальные узлы. На узле, инициирующем рассылку, выводить (в виде матрицы) топологию и остовное дерево, на остальных хостах кластера после получения сообщения выводить номер хоста и сам текст сообщения.

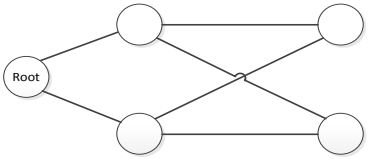


Рисунок 2.1 – Схема каналов взаимодействия процессов в кластере

2.2. Вариант №2

Распределено выполняются три процесса, каждый из которых вводит данные (целые числа) из локального файла по 5 элементов, выполняет суммирование 5 элементов и помещает полученную сумму в выходной файл. При этом все три процесса используют общую процедуру суммирования (процедура является общим разделяемым ресурсом), в которую передают массив элементов, обратно получают сумму. Выполнить синхронизацию доступа трех процессов к общему ресурсу (процедуре суммирования), используя модель распределенных семафоров.

2.3. Вариант №3

Реализовать задание второго варианта, используя модель передачи маркера. Выводить сообщения, комментирующие синхронизацию каждого вычислительного процесса (запрос и получение ресурса, освобождение ресурса), а также получение маркера в кольце вспомогательных процессов.

3. Контрольные вопросы.

3.1. Назовите особенности реализации и использования рассматриваемых моделей взаимодействия распределенных процессов.

3.2. Сформулируйте понятие топологии кластера и остовного дерева, определите форматы рассылаемых сообщений, алгоритмы построения топологии кластера и остовного дерева, алгоритм рассылки.
3.3. Сформулируйте алгоритм реализации механизма «Распределенных семафоров», назначение логических часов и очереди сообщений, определите форматы передаваемых сообщений.
3.4. Сформулируйте алгоритм реализации модели «передачи маркера».

ЛАБОРАТОРНАЯ РАБОТА №6
ИССЛЕДОВАНИЕ АЛГОРИТМОВ СОРТИРОВКИ ДАННЫХ МЕТОДАМИ ПУЗЫРЬКА И ШЕЛЛА, ИСПОЛЬЗУЕМЫХ ПРИ ПРОЕКТИРОВАНИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ ПРОГРАММНЫХ СИСТЕМ

Цель работы: программно реализовать и исследовать эффективность алгоритмов параллельной сортировки с использованием функций библиотеки MPI в сравнении с последовательными версиями тех же алгоритмов.

1. Теоретическое введение

1.1 Общие понятия о параллельных вычислительных системах

Параллельные вычислительные системы (ВС) являются одними из самых перспективных направлений увеличения производительности вычислительных средств. При решении задач распараллеливания существует два подхода:

1) имеется параллельная система, для которой необходимо подготовить план и схему решения поставленной задачи, т.е. ответить на следующие вопросы о том, в какой последовательности будут выполняться программные модули, на каких процессорах, как происходит обмен данными между процессорами, каким образом минимизировать время выполнения поставленной задачи;

2) имеется класс задач, для решения которых необходимо спроектировать параллельную вычислительную систему, минимизирующую время решения поставленной задачи, при минимальных затратах на ее проектирование.

При создании параллельных вычислительных систем учитываются различные аспекты их эксплуатации, такие как требования к времени решения и т.д., что приводит к различным структурным схемам построения таких систем. Перечислим их в порядке возрастания сложности:

1) однородные многомашинные вычислительные комплексы (ОМВК), которые представляют собой сеть однотипных ЭВМ;

2) неоднородные многомашинные вычислительные комплексы (НМВК), которые представляют собой сеть разнотипных ЭВМ;

3) однородные многопроцессорные вычислительные системы (ОМВС), которые представляют собой ЭВМ с однотипными процессорами и общим полем оперативной памяти или без него;

4) неоднородные многопроцессорные вычислительные системы (НМВС), которые представляют собой системы с разнотипными процессорами и общим полем оперативной памяти или без него.

1.2 Общие понятия сортировки данных

Задана исходная последовательность вида $S = (a_1, a_2, \dots, a_n)$ либо $S = (a_i | i = \overline{1,n})$. На основе последовательности S должна быть сформирова-

на последовательность S’ вида: $S' = (a'_1, a'_2, \dots, a'_n)$, где $a'_i \le a'_{i+1} (i = \overline{1, n-1})$ (здесь и далее все пояснения для краткости будут даваться только на примере упорядочивания данных по неубыванию).

Базовая операция при реализации методов сортировки – операция «Сравнить и переставить» (compare-exchange). Операция предполагает сравнение одной пары значений из сортируемой последовательности и перестановку этих значений в том случае, если их порядок не соответствует условиям сортировки. Методы (алгоритмы) реализации сортировки различаются способами выбора пар значений для сравнения.

Реализация базовой операции «Сравнить и переставить» (при $i<j$)

```
if (a[i]>a[j]) {
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
```

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

$$T(n^2)$$

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

$$T(n \log_2 n)$$

Ускорение сортировки может быть обеспечено при использовании нескольких ($p>1$) *вычислительных элементов* (процессоров или ядер). Исходный упорядочиваемый набор в этом случае разделяется на блоки, которые могут обрабатываться вычислительными элементами параллельно.

1.3 Принципы распараллеливания сортировки

В основу реализации подхода к распараллеливанию процесса сортировки данных положены следующие принципы:

1) исходный набор данных (значений) разделяется между устройствами, т. е. набор данных разделяется на блоки, каждый из которых закрепляется за конкретным вычислительным устройством (номер блока соответствует номеру процессорного элемента);

2) в ходе сортировки данные пересылаются между устройствами и сравниваются между собой (выполняется сравнение данных, входящих в разные блоки);

3) результирующий набор данных также разделен между устройствами, при этом значения, расположенные на процессоре с меньшими номерами, не превышает значения, расположенные на процессорах с большими номерами. Т.е. если блоки данных (идентификаторы блоков данных) являют-

ся закрепленными за соответствующими процессорными элементами, тогда в процессе сортировки изменяется состав этих блоков.

Обозначим через l идентификатор блока данных, соответствующий l-му процессорному элементу P_l , n_l –количество элементов в l-ом блоке данных. Тогда значение a_{l,n_l} (n_l–ый элемент в l-ом блоке данных) на процессоре P_l не больше значения $a_{l+1,1}$ (первый элемент) на процессоре P_{l+1}.

Внутри l-ого блока данные упорядочиваются по рассматриваемому признаку.

1.3.1 *Реализация операции «Сравнить и переставить» для (P = n)*

Здесь через P обозначено количество процессорных элементов, через n–количество данных в последовательности. При $i<j$ имеем: a_i–>P_i, a_j–>P_j (данные a_i соответствуют процессору P_i, данные a_j соответствуют процессору P_j). Параллельная реализация операции «Сравнить и переставить» предполагает:

1) обмен имеющимися на процессорах P_i и P_j значениями a_i и a_j; в результате на каждом процессоре рассматриваются одинаковые пары значений (a_i, a_j) (т.к. (a_i, a_j)–>P_i, (a_i, a_j)–>P_j);

2) сравнение на каждом процессоре P_i и P_j пар (a_i, a_j) т.о., чтобы при $i<j$ на P_i сохранялся минимальный элемент в паре, на P_j – максимальный элемент в паре (a_i, a_j). Т.о. на основе пары (a_i, a_j) формируется новый элемент a'_i, закрепленный за процессором P_i, следующим образом:

$$a'_i = \min(a_i, a_j)$$

На основе пары (a_i, a_j) определяется новый элемент a'_j , закрепленный за процессором P_j, следующим образом:

$$a'_j = \max(a_i, a_j)$$

Итоговая запись результата выполнения операций «Сравнить и переставить»:

$$a'_i \to P_i, \text{ где } a'_i = \min(a_i, a_j)$$
$$a'_i \to P_j, \text{ где } a'_i = \max(a_i, a_j)$$

1.3.2 Распространение базовой операции «Сравнить и переставить» для случая p<n. Операция «Сравнить и разделить»

При $p<n$ должно быть определено P блоков данных, каждый из блоков имеет размер n/p. Тогда при $p<n$ каждому процессору ставится в соответствие не единственное значение a_i и совокупность значений (блок) из сортируемого набора данных.

Реализация параллельной (распределенной) сортировки должна предусматривать:

1) упорядочивание элементов (значений) внутри блоков;

2) упорядочивание элементов (значений) между блоками (т.е. $a_{l,n_l} < a_{l+1,1}$), где элементы a_{l,n_l} и $a_{l+1,1}$ – последний и первый элементы в упорядоченных блоках l и (l+1) процессорных элементах.

Т.о. на основе исходных составов блоков, формируемых на базе исходный последовательности $S = (a_1, a_2, \dots, a_n)$, определяются модифицированные составы блоков в соответствии с введенными принципами сортировки. Для формирования модифицированных составов блоков используется операция «сравнить и разделить».

Обозначим через b_l блок значение, соответствующих процессорному элементу P_l, вид блока b_l следующий: $b_l = (a_{l1}, a_{l2}, \dots, a_{ln_l})$.

Тогда в результате обмена блоками b_l и b_h между ПЭ P_l и P_h формируется фрагмент (блок) данных длиной $2n/p$ (в результате слияния блоков b_l и b_h формируется блок $b_l \cup b_h$, где «∪» — операция конкатенации (соединения) блоков).

$(b_l \cup b_h)'_2 \to P_h$ Результирующий блок $b_l \cup b_h$ должен быть упорядочен в соответствии с введенным правилом. После чего формируется левая половина блока $(b_l \cup b_h)'$ в виде $(b_l \cup b_h)'_1$ и правая половина блока в виде $(b_l \cup b_h)'_2$. Левая половина $(b_l \cup b_h)'_1$ блока закрепляется за ПЭ P_l, правая за P_h. Таким образом $(b_l \cup b_h)'_1 \to P_l$,

1.3.3 Вид последовательности действий при реализации операции «сравнить и разделить»

В результате реализации процедуры «сравнить и разделить» блоки на процессорах P_l и P_h совпадают по размеру с блоками b_l и b_h (исходными блоками). Все значения, расположенные на процессоре P_l, не превышают значений на процессоре P_h

Формализация введенного утверждения:

$$[b_l \cup b_h]' = b'_l \cup b'_h; \forall a_i \in \forall b'_l, \forall a_j \in \forall b'_h : a_i \le a_j .$$

Операция «сравнить и разделить» является базовой подзадачей при организации параллельной сортировки.

Схема реализации процедуры «сравнить и разделить» представлена на Рис. 1.1.

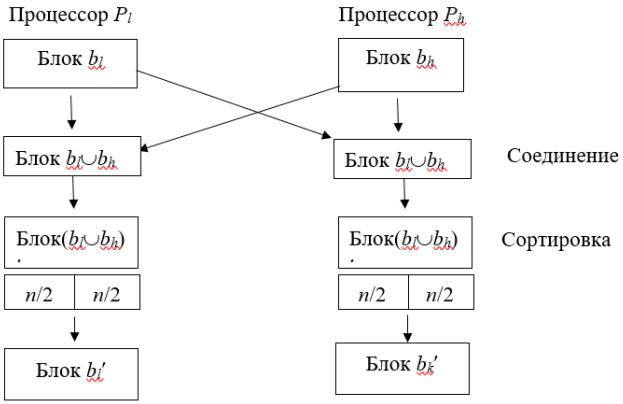


Рисунок 1.1. – Схема реализации процедуры «сравнить и разделить»

Особенности использования операции «сравнить и разделить» при реализации алгоритмов сортировки:

1) составы блоков данных, относящиеся к процессорным элементам, изменяются в ходе выполнения сортировки;

2) размер блоков данных может быть постоянным и одинаковым, либо может различаться в ходе реализации сортировки.

1.4 Метод пузырьковой сортировки. Параллельная реализация

1.4.1 Реализация чет-нечетной перестановки при P=*n*

Для параллельной реализации метода пузырьковой сортировки используется его модификация, называемая чет-нечетной перестановкой.

Этапы реализации метода чет-нечетной перестановки:

1) Разбиение массива (последовательности) S на пары вида (a_0, a_1), (a_2, a_3), (a_4, a_5), …, (a_{n-2}, a_{n-1}) – четная сортировка.

Для каждой пары элементов выполняется операция «сравнить и переставить». В каждой паре слева помещается наименьший элемент, справа – наибольший.

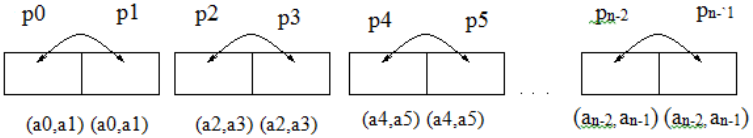
2) Массив значений (последовательности) S разбивается на пары вида (a_1, a_2), (a_3, a_4), …, (a_{n-3}, a_{n-2})– нечетная сортировка.

Для каждой пары элементов выполняется операция «сравнить и переставить». Таким образом при четной сортировке пары начинаются с четных индексов, при нечетной сортировке с нечетных индексов.

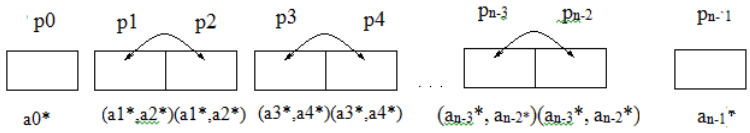
Рассмотренный алгоритм непосредственно может быть использован при реализации параллельной сортировки в случае, если число ПЭ равно количеству n элементов в последовательности S.

Порядок реализации обмена при формировании пар значений:

1. Первый этап (четные пары)



2. Второй этап (нечетные пары процессов)



Пример реализации чет-нечетной перестановки элементов при пузырьковой сортировке

Исходная последовательность: 18752

1 этап: 18 *75* 2 –> 18 *57* 2

2 этап: 1 *85 72* –> 1 *58 27*

1 этап: 15 *82* 7 –> 15 *28* 7

2 этап: 1 *52 87* –> 1 *25 78*

Вывод по параллельной реализации пузырьковой сортировки в случае P= n:

1) Процессоры с номерами, соответствующими элементам в парах, обмениваются друг с другом значениями и формируют их пары.

2) Каждый ПЭ, сформировавший на данном этапе пару элементов, реализует операцию «сравнить и переставить» параллельно с другими ПЭ.

3) Если номер ПЭ соответствует меньшему номеру элемента в паре, то он сохраняет минимальный элемент, если номер процессорного элемента соответствует большему номеру, то он сохраняет максимальный элемент.

1.4.2 Реализация пузырьковой сортировки при P<n (упрощенная интерпретация)

Последовательность из n элементов разделяется на части одинакового размера n/p. Каждая из частей назначается соответствующему устройству. Элементы, входящие в блок предварительно, сортируются (блок закреплен b_i за устройством P_i). После начальной инициализации и сортировки блоков алгоритм пузырьковой сортировки предполагает реализацию следующих этапов (предполагает реализацию обмена между вычислительными устройствами следующих образом):

1) Обмен внутри четных пар процессоров с номерами (0, 1), (2, 3), (4,5),...;
2) Обмен внутри нечетных пар процессоров с номерами (1, 2), (3, 4), (5, 6);

После обмена каждое устройство, реализовывавшее обмен, выполняет операцию «сравнить и разделить» параллельно с другими устройствами.

Пример реализации обмена при пузырьковой сортировке, при условии *P<n*, представлен на Рис.1.2.

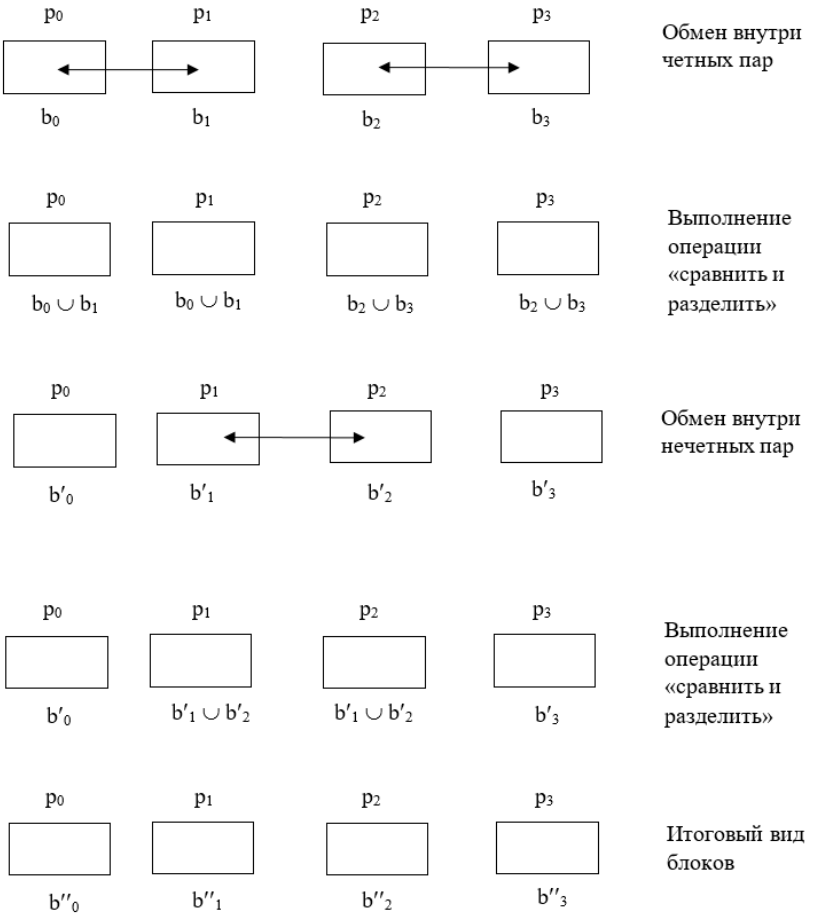


Рисунок 1.2 – Схема реализации четно-нечетной перестановки при $P<n$

Пример реализации обмена при пузырьковой сортировке, (n = 6, p = 3)
Исходная последовательность S имеет вид:

S = (2, 8, 4, 5, 6, 7)

0 этап	2 8	4 5	6 7
	p_0	p_1	p_2
1-ый этап	p_0	p_1	p_2

	2 8 4 5	2 8 4 5	6 7
Операция «сравнить и	p_0	p_1	p_2
разделить»	2 4 5 8	2 4 5 8	6 7
	2 4	5 8	6 7
2-ой этап	p_0	p_1	p_2
	2 4	5 8 7 6	5 8 6 7
Операция «сравнить и	p_0	p_1	p_2
разделить»	2 4	5 6 7 8	5 6 7 8
Конечный вид последо-	2 4	5 6	7 8
вательности			

1.4.3 Формализация метода чет-нечетной перестановки для случая P<n (блочный аналог четно-нечетной перестановки)

Алгоритм «сортировки слиянием» двух упорядоченных массивов.

Заданы исходные массивы, элементы которых должны быть упорядочены. Массивы являются отсортированными.

Вид исходных массивов:

$$A = \{a_{j_a} \mid j_a = \overline{0, n_A - 1}\}, a_{j_a+1} \geq a_{j_a}$$
$$B = \{b_{j_b} \mid j_b = \overline{0, n_B - 1}\}, b_{j_b+1} \geq b_{j_b}$$

Вид результирующего массива:

$$C = \{c_j \mid j = \overline{0, n_C - 1}\},$$

где $n_C = n_A + n_B$, в итоге элементы такие, что $c_{j+1} \geq c_j$;

Начальными условиями для реализации алгоритма является инициализация индексов массивов A, B, C значением 0 (j=0)

Три возможных варианта реализации вычислительного процесса в соответствии с алгоритмом:

I. Если $a_{n_A-1} \leq b_0$, тогда

1. $c_j = a_{j_a}$, где $j_a = \overline{0, n_A - 1}; j = \overline{0, n_A - 1}$
2. $c_j = b_{j_b}$, где $j_b = \overline{0, n_B - 1}; j = \overline{n_A, n_A + n_B - 1}$

II. если $b_{n_b-1} \leq a_0$, тогда:

1. $c_j = b_{j_b}$, где $j_b = \overline{0, n_B - 1}; j = \overline{0, n_B - 1}$
2. $c_j = a_{j_a}$, где $j_a = \overline{0, n_A - 1}; j = \overline{n_B, n_B + n_B - 1}$

т.е. сначала в С записывается один массив, потом другой.

III. Сравнение a_{ja} и b_{jb}, если:

1. $a_{j_a} < b_{j_b}$, тогда $c_j = a_{j_a}$;
   $j = j + 1$; $j_a = j_a + 1$;(индекс j_b не изменяется)
2. $b_{j_b} \leq a_{j_a}$, тогда $c_j = b_{j_b}$;
   $j = j + 1$; $j_b = j_b + 1$;(индекс j_a не изменяется)
3. сравнение a_{j_a} и b_{j_b} продолжается до тех пор, пока
   $j_a \leq n_A - 1$ либо $j_b \leq n_B - 1$;
   Если $j_a > n_A - 1$, тогда шаг 4;
   Если $j_b > n_B - 1$, тогда шаг 5;
4. если $j_a > n_A - 1$,тогда
- $c_j = b_{j_b}, i = i + 1; j_b = j_b + 1;$
- Если $j_b \leq n_B - 1$, тогда шаг 4;
5. если $j_b > n_B - 1$,тогда
- $c_j = a_{j_a}, j = j + 1; j_a = j_a + 1;$
- если $j_a \leq n_A - 1$, тогда шаг 5

Т.е. как только исчерпан один из входных массивов (A или B), но не исчерпан другой, тогда оставшаяся часть не законченного массива переписывается в массив C.

1.5 Сортировка Шелла

1.5.1 Последовательная реализация сортировки Шелла

Особенность реализации перестановки – обмен при выполнении условия сортировки выполняется между элементами массива, расположенными друг от друга на большом расстоянии.

При этом на 1-м этапе рассматриваются группы по 2 элемента, на 2-м этапе рассматриваются группы по 4 элемента, на 3-м этапе - по 8 элементов и на заключительном этапе рассматривается весь массив.

Таким образом на 1-м шаге происходит упорядочивание элементов в $^{n}/_{2}$ парах следующего вида: $(a_i; a_{^{n}/_{2}+i})$, где $i = \overline{1, ^{n}/_{2}}$.

На 2-м этапе упорядочиваются элементы в $^{n}/_{4}$ группах из 4-х элементов вида: $(a_i; a_{^{n}/_{4}+i}; a_{^{n}/_{2}+i}; a_{^{3n}/_{4}+i})$, где $i = \overline{1, ^{n}/_{4}}$.

На 3-м этапе упорядочиваются элементы в $^{n}/_{8}$ группах по 8 элементов вида: $(a_i; a_{^{n}/_{8}+i}; a_{^{2n}/_{8}+i}; a_{^{3n}/_{8}+i}; a_{^{4n}/_{8}+i}; a_{^{5n}/_{8}+i}; a_{^{6n}/_{8}+i}; a_{^{7n}/_{8}+i})$, где $i = \overline{1, ^{n}/_{8}}$

На заключительном этапе упорядочиваются элементы во всем массиве $(a_1, a_2, \dots, a_n)$.

Таким образом на каждом следующем этапе расстояния между элементами в группе уменьшается в 2 раза, а число элементов в группе увеличивается в 2 раза. На последнем этапе сортируется весь массив как одна группа.

При определении элементов, входящих в соответствующие группы, внутри этих групп выполняется сортировка элементов.

Таким образом на каждом этапе выполняется сортировка элементов внутри выделенных групп.

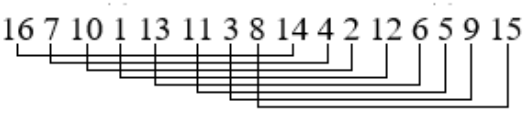
1.5.2 Пример реализации последовательной сортировки Шелла

Исходный массив имеет вид:

16 10 1 13 11 3 8 14 4 2 12 6 5 9 15

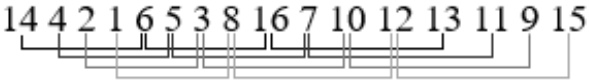
1 этап (1; $n/2$ +1), (2; $n/2$ +2),…,($n/2$; n)

8 групп по 2 элемента



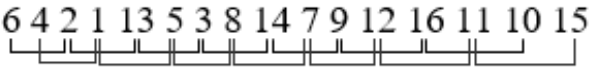
2 этап 4 группы по 4 элемента, группы элементов с индексами:

$$(1; n/4 + 1; n/2 + 1; 3n/4 + 1), \dots, (n/4; 2n/4; 3n/4; n)$$

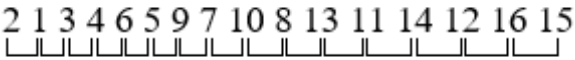


3 этап 2 группы по 8 элементов, группы элементов с индексами:

$$\left(1; n/8 + 1; n/4 + 1; 3n/8 + 1; n/4 + 1; 5n/8 + 1; 6n/8 + 1; 7n/8 + 1\right),$$
$$\dots, (2; n/8 + 2; 2n/8 + 2; 3n/8 + 2; 4n/8 + 2; 5n/8 + 2; 6n/8 + 2; 7n/8 + 2)$$



4 этап 1 группа по 16 элементов:



1 2 3 4 5 6 7 9 8 10 11 13 12 14 15 16

Необходим повторный проход по массиву.

1.5.3 Параллельная реализация сортировки Шелла

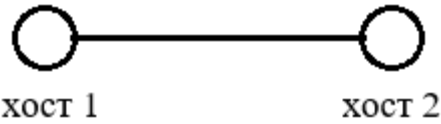
Реализация параллельной сортировки предполагает наличие параллельно действующих узлов. Здесь должно быть определено различие между взаимодействием блоков массива и взаимодействием ПЭ.

Гиперкуб — топология соединения отдельных хостов кластера либо гиперкуб – топология взаимодействия процессов, выполняющихся на хостах кластера.

Реализацию параллельного алгоритма сортировки Шелла рассмотрим на примере гиперкуба заданной размерности.

Имеем: N=1, тогда число ПЭ, входящих в гиперкуб $p=2^N$, тогда p = 2.

В этом случае топология кластера имеет вид:



Количество элементов в массиве равно 16.

Для реализации параллельной сортировки Шелла исходный массив должен быть разбит на 2*p блоков данных (2*p блоков элементов массива).

При N=1 p=2 n=16 должен быть сформировано 4 блока по 4 элемента (количество блоков данных $q=2^{N+1}$).

Из блоков данных должен быть образован гиперкуб размерности N+1. Таким образом каждому из блоков данных ставится в соответствие процесс. Нумерация процессов реализуется в двоичной системе, соответственно 00, 01, 10, 11 (номер процесса соответствует номеру блока данных). Вид номера: первый нулевой, нумерация справа-налево, начиная с нулевого разряда.

Реализация параллельной сортировки Шелла выполняется в 2 этапа:

1 этап: (N+1 итерация) предполагает выполнение операции «сравнить и разделить» для соответствующих пар процессов в кубе.

Правило формирования номеров взаимодействующих процессов (номеров блоков данных, для которых выполняется операция "сравнить и разделить").

Если i-номер итерации ($i=\overline{0,\mathrm{N}}$ всего N+1 итерация), тогда пары образуют те процессы (блоки), у которых различие в битовом представлении их номеров имеются в позиции (разряде) N-i.

При N=1, i=0 для первых пар блоков должно быть различие в первом разряде. На первой итерации обмен блоками и реализация операции «сравнить и разделить» должен быть выполнены для номеров 00 и 10, 01 и 11.

При N=1, i=1 для вторых пар блоков должно быть различие в нулевом разряде. Таким образом на второй итерации обмен блоками и реализация операции "сравнить и разделить" выполняется для номеров 00 и 01, 10 и 11.

Таким образом при N=1 должно быть выполнено 2 итерации, на которых реализуется обмен блоками между процессами с соответствующими но-

мерами и выполнение процессами операции «сравнить и разделить».

2 этап предполагает реализацию итераций алгоритма четной-нечетной перестановок (т.е. последовательный обмен блоками между процессами, номера которых определяются алгоритмом четной-нечетной перестановки, и реализация операции «сравнить и разделить».

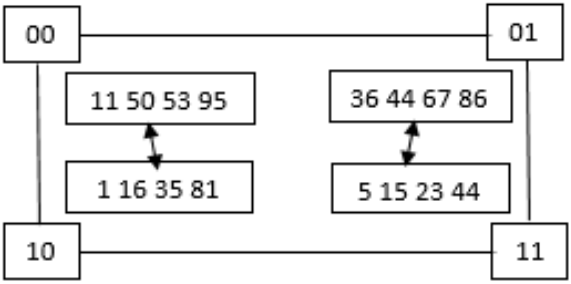
Итерации продолжаются до прекращения изменения сортируемого набора.

В рассматриваемом случае четная перестановка – обмен данными между процессами с номерами (00, 01) и (10, 11) (обмен блоками в указанных парах процессов). Нечетная перестановка – (01, 10).

Пример реализации итераций алгоритма параллельной сортировки Шелла

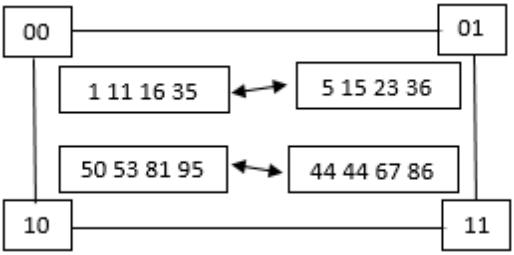
Некоторый массив (с разбиением на блоки) имеет вид:
11 50 53 95 | 36 44 67 86 | 1 16 35 81 | 5 15 23 44

1-я итерация 1-го этапа алгоритма:
Процессы/блоки

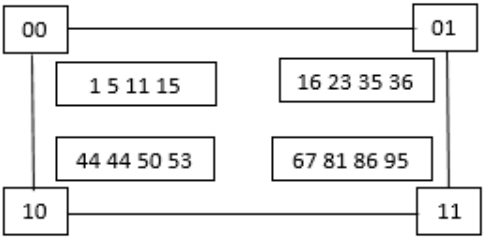


Гиперкуб с N=1 для хостов
Гиперкуб с N=2 для процессов

2-я итерация 1-го этапа



Результат второй итерации 1-го этапа



Исходя из результатов 1-го этапа четно-нечетные перестановки не требуется.

Пример организации обмена между процессами на первом этапе при N=2, p=4, q=2*p=8

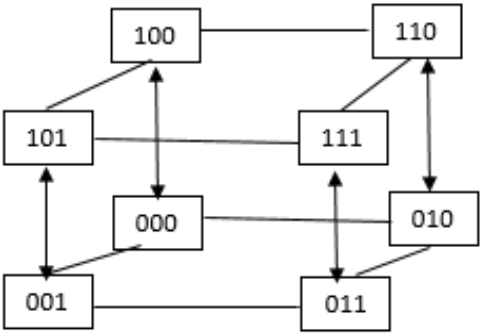
Нумерация блоков (процессов): 000, … ,111 (0, … ,7)

Формат номера блока: 2-я позиция, 1-я позиция, 0-я позиция.

Определение номеров процессов, реализующих обмен блоками данных на каждой итерации 1-го этапа.

Итерация 0 (i=0) =>$N_{\text{позиции}} = N - i = 2$.

Вид обмена между процессами в гиперкубе



Обмен между парами блоков:

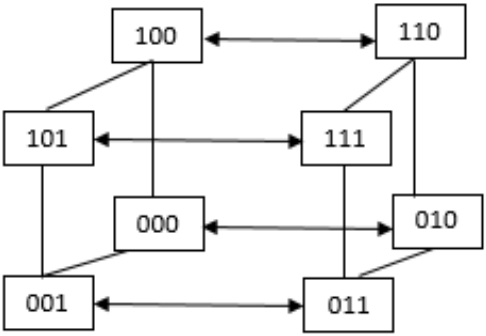
0(0̲00)⇔4(1̲00)

1(0̲01)⇔5(1̲01)

2(0̲10)⇔6(1̲10)

3(0̲11)⇔7(1̲11)

Итерация 1 (i=1) $N_{\text{позиции}} = N - i = 1$.



Обмен между парами блоков:
0(0<u>0</u>0)⇔2(0<u>1</u>0)
1(0<u>0</u>1)⇔3(0<u>1</u>1)
4(1<u>0</u>0)⇔6(1<u>1</u>0)
5(1<u>0</u>1)⇔7(1<u>1</u>1)

Итерация 2 (i=2) $N_{\text{позиции}} = N - i = 0$.
Обмен между парами блоков:
0(00<u>0</u>)⇔1(00<u>1</u>)
2(01<u>0</u>)⇔3(01<u>1</u>)
4(10<u>0</u>)⇔5(10<u>1</u>)
6(11<u>0</u>)⇔7(11<u>1</u>)

2. Задание на работу.

Выполнить разработку и отладку программы параллельной сортировки данных с использованием вызовов требуемых функций библиотеки MPI в соответствии с вариантом, указанным преподавателем. Дополнительно реализовать последовательный вариант того же метода сортировки. Получить результаты работы программы в виде протоколов сообщений, комментирующих параллельное выполнение процессов и их взаимодействие в ходе выполнения. Оценить эффективность параллельного процесса сортировки в сравнении с последовательным на том же наборе исходных данных.

Таблица 6.1 — Варианты заданий

Номер варианта	Вид сортировки
1	Чет-нечетная
2	Шелла N=2
3	Шелла N=3

3. Контрольные вопросы

3.1. Назовите особенности реализации и использования рассматриваемых моделей взаимодействия распределенных процессов.
3.2. Сформулируйте понятие топологии кластера и остовного дерева, определите форматы рассылаемых сообщений, алгоритмы построения топологии кластера и остовного дерева, алгоритм рассылки.
3.3. Сформулируйте алгоритм реализации механизма «Распределенных семафоров», назначение логических часов и очереди сообщений, определите форматы передаваемых сообщений.
3.4. Сформулируйте алгоритм реализации модели «передачи маркера».

ЛАБОРАТОРНАЯ РАБОТА №7
ИССЛЕДОВАНИЕ АЛГОРИТМОВ ПАРАЛЛЕЛЬНОЙ БЫСТРОЙ СОРТИРОВКИ ДАННЫХ, ИСПОЛЬЗУЕМЫХ ПРИ ПРОЕКТИРОВАНИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ ПРОГРАММНЫХ СИСТЕМ

Цель работы: реализовать и исследовать эффективность алгоритмов параллельной быстрой сортировки с использованием функций библиотеки MPI в сравнении с последовательными версиями тех же алгоритмов.

1 Теоретическое введение

1.1 Алгоритм последовательной быстрой сортировки

Алгоритм быстрой сортировки, предложенный Хоаром (*Hoare C.A.R.*), относится к числу эффективных методов упорядочивания данных и широко используется в практических приложениях.

Схема быстрой сортировки представлена на Рис.1.1

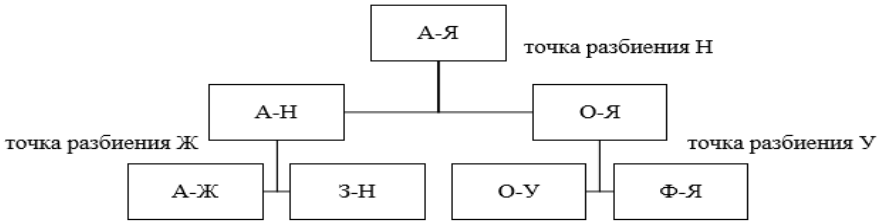


Рисунок 1.1 – Схема реализации последовательной сортировки

Метод основывается на последовательном разделении сортируемого набора на блоки меньшего размера таким образом, чтобы между значениями разных блоков обеспечивалось отношение упорядоченности (для пары блоков все значения одного из них не превышают значения другого).

Особенности реализации алгоритма:

1. В левую часть массива помещаются все элементы, значения которого не больше опорного элемента (меньше или равны), в правую часть массива – те элемента которые не меньше опорного.

2. Выбор опорного элемента должен быть выполнен таким образом, чтобы массив был разделен на примерно равные части.

Один из вариантов реализации алгоритма.

1. В рассмотрение вводятся 2 указателя для обозначения начального и конечного элемента последовательности элементов A (начальный – left, конечный – right), вводится опорный элемент mid.

2. Индекс опорного элемента вычисляется как (left+right)/2. Значение этого элемента присваивается переменной mid.

3. Указатель left смещается с шагом 1 вправо к концу массива до тех пор, пока не выполнится условие фиксации A[left]>mid. Указатель right сме-

щается с шагом -1 влево к началу массива до тех пор, пока не выполнится условие фиксации A[right]<mid.

4. Найденные таким образом элементы меняются местами.
5. Шаги 3 и 4 повторяются пока left<right.
6. Если полученные подмассивы до и после опорного элемента имеют более чем 1 элемент, выполняется быстрая сортировка каждого из них тем же методом.

Пример реализации алгоритма:

Исходный массив A[1..8] имеет вид:

```
6 7 2 5 9 1 3 8
```

1. left – индекс первого элемента, right – индекс последнего элемента. left=1, right=8.
2. Определение mid: (left+right)/2=4.5. Ближайшее значение индекса равно 4. Значение опорного элемента mid=A[4] => mid=5.
3. Элемент A[left]=6 сравнивается с mid=5. Условие фиксации left уже выполнено (A[1]>mid), указатель left не изменяется.
4. Элемент A[right]=8 сравнивается с mid=5. Так как условие A[right]<mid не выполняется, указатель изменяется на -1 и проверка повторяется. Условие фиксации выполняется при right=7.
5. Имеем left=1, right=7. Элемент A[1] и A[7] меняются местами.

Вид массива после реализации обмена:

```
3 7 2 5 9 1 6 8
```

После реализации обмена указатели изменяют свое значение на 1 (left=2, right=6).

6. Выполняется проверка и изменение указателей до условий их фиксации A[left]>mid, A[right]<mid.

Так как условия снова выполняются, то элементы A[2] и A[6] меняются местами, указатели изменяют значения.

Вид массива:

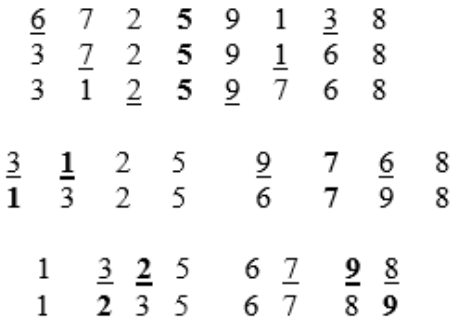
```
3 1 2 5 9 7 6 8
```

Новые значения указателей: left=3, right=5.

7. Снова выполняется проверка условий A[left]>mid и A[right]<mid. Так как A[3]<mid, left увеличивается на 1.

– Так как A[5]>mid (то есть условие не выполняется), то указатель right уменьшается на 1. Тогда left=right и первый этап упорядочивания завершается. Вид массива не изменен.

8. Так как в каждом из подмассивов больше чем 1 элемент, каждый из подмассивов нуждается в дополнительном упорядочивании. Массив разделяется на два с границами A[1..left] и A[right+1..8]. Каждый из них упорядочивается тем же методом. Пример полной сортировки приведен ниже:



Варианты реализации алгоритма

Эффективность алгоритма зависит от выбора границ подмассивов.

Возможны различные методы определения опорного элемента:

- Среднее арифметическое всего массива
- Среднее арифметическое крайних элементов массива
- Центральный элемент массива.

Таким образом, опорный элемент в общем случае может быть и одним из крайних элементов.

После упорядочивания элементов в левой и правой частях новые рассматриваемые массивы должны включать опорный элемент (опорный элемент включается либо в левую, либо в правую части).

1.2 Алгоритм параллельной быстрой сортировки

Параллельная реализация быстрой сортировки выполняется при условии, что топология связи узлов (ПЭ) представлена в виде *N*-мерного гиперкуба. Тогда $p=2N$ – количество ПЭ в гиперкубе.

Если n – количество элементов в массиве тогда количество элементов в блоках данных, закрепляемых за ПЭ, определяется как n/p. Нумерация блоков соответствует нумерации ПЭ.

Среди p процессорных элементов один ПЭ является ведущим, он определяет первоначальное значение опорного элемента.

Способ реализации 1-й стадии алгоритма

1. На ведущем ПЭ выполняется выбор опорного элемента (способ формирования опорного элемента – среднее арифметическое элементов блока, размещенных на ведущем ПЭ).

2. Широковещательная рассылка ведущим ПЭ опорного элемента между остальными (p-1) ПЭ.

3. На каждом ПЭ выполняется разделение имеющегося блока данных на две части (2 подмассива) с использованием опорного элемента - выполняется быстрая сортировка на каждом ПЭ. В результате будут сформированы подмассивы элементов, не больших, чем опорный (левый подмассив), и не меньших, чем опорный (правый подмассив).

4. Процессы, для которых битовое представление номеров отличается

в N-ой позиции реализуют обмен данными.

В результате обмена на ПЭ, номер которых в позиции N в битовом представлении содержит 0, окажутся значения меньше опорного элемента, на ПЭ номер которых в позиции N в битовом представлении содержит 1, окажутся значения данных (элементов массива) больше опорного элемента.

Пример реализации 1-й стадии процесса быстрой сортировки (N=2, n=16, p=4, n/p=4):

1. Распределение исходной последовательности по процессам имеет вид:

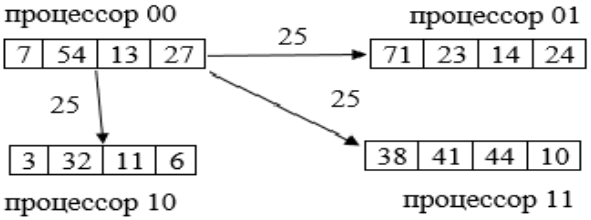


Рисунок 1.2 — Распределение данных и опорного элемента по ПЭ.

–

2. На процессоре 00 в качестве опорного элемента выбирается значение, соответствующее среднему арифметическому значению данных, хранящихся на этом ПЭ (среднее значение = 25).

3. В результате быстрой сортировки с учетом опорного элемента, равного 25, получены левые и правые подпоследовательности (подмассивы) в следующем виде:

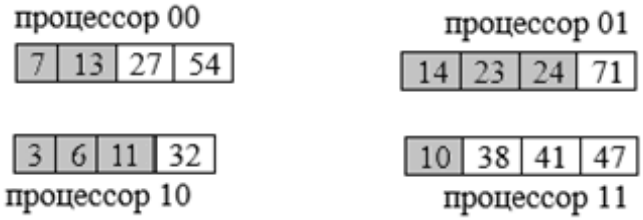


Рисунок 1.3 - Результат разделения на подмассивы относительно опорного элемента

Реализация обмена между процессорами 00-10 и 01-11.

От процессора 00 к процессору 10 передается правая часть, от процессора 10 к процессору 00 - левая часть.

От процессора 01 к процессору 11 передается правая часть, от процессора 11 к процессору 01 передается левая часть.

Полученный вид последовательности представлен на Рис 1.4.

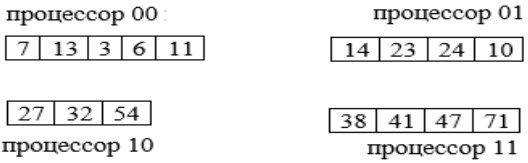


Рисунок 1.4 — Результат обмена подмассивами

Способ реализации 2-й стадии алгоритма

1. Рассматриваются гиперкубы, размерностью на 1 меньше, чем на предыдущей стадии.

В данном случае размерность гиперкубов равна 1 и процессоры, входящие в гиперкубы, следующие:

а) гиперкуб 1: процессоры 00 и 01;

б) гиперкуб 2: процессоры 10 и 11.

2. На процессоре с меньшим номером определяется значение опорного элемента, который рассматривается между ПЭ, входящими в гиперкуб.

После выполнения широковещательной рассылки опорного элемента на каждый ПЭ, входящий в гиперкуб меньшей размерности, реализуется алгоритм быстрой сортировки (с учетом опорного элемента).

На каждом ПЭ гиперкубов:

В результате сформированы подпоследовательности элементов, не больших, чем опорный, и больших опорного.

ПЭ, входящие в гиперкубы меньшей размерности, реализуют обмен подпоследовательностями больших и меньших, чем опорный, элементов. В результате на каждом ПЭ будут сформированы новые последовательности.

Пример реализации 2-й стадии алгоритма параллельной быстрой сортировки.

1. На процессорах 00 и 10 определение опорных элементов (соответственно, значение 10 и 37), их широковещательная рассылка между ПЭ, входящими в гиперкуб (Рис. 1.7).
2. На ПЭ реализация алгоритма быстрой сортировки с учетом опорных элементов, формирование «левых» и «правых» подпоследовательностей (подмассивов) (Рис. 1.8).
3. Реализация обмена подпоследовательностями (Рис. 1.9).

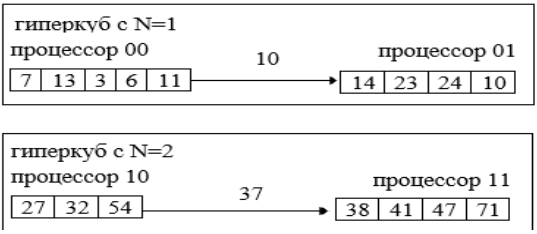


Рисунок 1.5 — Широковещательная рассылка опрного элемента

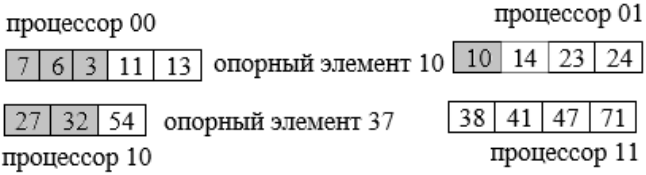


Рисунок 1.6 — Результат разделения на подмассивы относительно опорного элемента

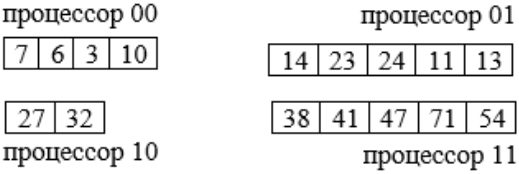


Рисунок 1.7 — Результат обмена подмассивами

Способ реализации 3-й стадии алгоритма

Рассматривается гиперкуб размерности на 1 меньше, чем размерность гиперкуба на предыдущей стадии (размерность гиперкуба $N = 0$).

Опорный элемент на каждом ПЭ определяется как среднее значение из элементов, хранящихся на этом ПЭ (Рис. 1.10). Реализация быстрой сортировки выполняется до тех пор, пока элементы в последовательностях не будут упорядочены.

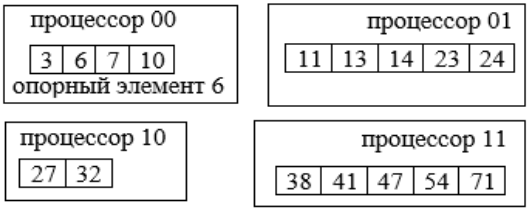


Рисунок 1.8 — Гиперкубы с размерностью N=0

1.3 Модифицированный метод параллельной быстрой сортировки

Первая стадия базового методом параллельной быстрой сортировки и модифицированного метода являются одинаковыми. Методы отличаются последующими стадиями сортировки. Рассмотрим на примере реализацию модифицированного метода быстрой сортировки. (N=2, p=4, n/p=4 при n=16). Исходная последовательность имеет вид:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	54	13	27	71	23	14	24	3	32	11	6	38	41	44	10
Процессор 1				Процессор 2				Процессор 3				Процессор 4			

В качестве опорного элемента выбираем последний элемент последовательности равный 10.

На каждом ПЭ выполняется (локально) алгоритм быстрой сортировки. В результате на каждом ПЭ сформированы «левые» и «правые» последовательности элементов.

Для рассматриваемой последовательности результаты реализации первой итерации алгоритма представлены на Рис 1.11.

Процессор 1			
7	54	13	27
Процессор 3			
3	6	11	32

Процессор 2			
71	23	14	24
Процессор 4			
10	41	44	38

Рисунок 1.9 — Результат первой итерации

Способ реализации 2-й стадии алгоритма

1. Каждый из процессоров определяет сколько элементов оказалась в левой и правой частях. В результате на ведущем ПЭ формируется последовательность значений вида:

а) количество элементов в блоке, меньших или разных заданному опорному элементу.

1	0	2	1

б) Количество элементов в блоке больше заданного.

3	4	2	3

2. Для полученных последовательностей вычисляются префиксные суммы, дополняемые слева 0. Префиксной суммой (префиксом) последовательности чисел $x_1, x_2, \dots, x_n$ называется другая последовательность $\pi_1, \pi_2, \dots, \pi_N$, элементы которой определяется следующим образом:

$$\pi_1 = x_1,\ \pi_2 = x_1 + x_2, \dots, \pi_N = x_1 + x_2 + \cdots + x_N$$

Так как получаемые префиксные суммы для сформированных последовательностей значений дополняются слева нулем, тогда вид префиксных сумм следующий:

0	1	1	3	4
0	3	7	9	12

3. Последнее значение префиксной суммы для левых частей исключается и прибавляется к каждому элементу префиксной суммы для правых час-

тей.

Модифицированные префиксные суммы имеет следующий вид:

а)

0	1	1	3
(1-й ПЭ)	(2-й ПЭ)	(3-й ПЭ)	(4-й ПЭ)

б) 4+

0	3	7	9	12

4	7	11	13	16
(1-й ПЭ)	(2-й ПЭ)	(3-й ПЭ)	(4-й ПЭ)	

Разряды префиксных сумм соответствуют номерам позиций, начиная с которых в результирующей последовательности размещаются соответствующее «левые» и «правые» последовательности для соответствующих ПЭ.

Таким образом префиксная сумма, полученная в пункте а), соответствует номерам позиций левых последовательностей в формируемой результирующей последовательности.

Разряды префиксной суммы, полученной в пункте б), соответствуют номерам позиций правых последовательностей в формируемой результирующей последовательности.

Вид сформированной результирующей последовательности представлен на Рис 1.12.

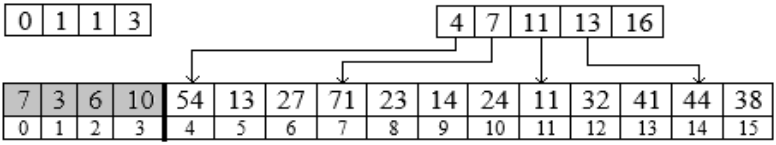


Рисунок 1.10 — Результирующая последовательность (первый этап)

Таким образом элементы левой последовательности с ПЭ3 располагается в позициях, начиная с 1, элементы правой последовательности этого ПЭ размещаются, начиная с 11 позиции.

После формирования результирующей последовательности она распределяется по ПЭ. При этом элементы, входящие в левые части последовательностей на каждом ПЭ, закрепляются за одним ПЭ и сортируются независимо от других элементов.

То есть элементы в позициях с 4 по 15 распределяется по 3 ПЭ и их сортировка выполняется аналогичным образом, независимо от элементов, входящих в левые последовательности на рассматриваемой стадии.

Результат распределения последовательностей по ПЭ представлен на Рис. 2.13.

Процессор 1			
7	3	6	10
Процессор 3			
23	14	24	11

Процессор 2			
54	13	27	71
Процессор 4			
32	41	44	38

Рисунок 1.11 — Распределение данных по процессорным элементам

Вывод: на последующей итерации сортировка элементов на процессоре 1 и процессоре 2, 3, 4 выполняется независимо.

Реализация следующей итерации алгоритма (опорный элемент 38)

Процессор 1			
3	6	7	10
Процессор 3			
23	14	24	11

Процессор 2			
13	27	54	71
Процессор 4			
32	38	41	44

Рисунок 1.12 — Результат второй итерации

Итоговая последовательность после второй итерации алгоритма представлена на Рис. 2.13.

3	6	7	10	13	27	23	14	24	11	32	38	54	71	41	44

часть не рассматривается

Рисунок 2.13 — Результирующая последовательность (второй этап)

На последующих шагах сортируется последовательность, полученная из левых частей на текущей итерации, и последовательность элементов правых частей (на одном из процессоров).

1.4 Сортировка с использованием регулярного набора образцов

Алгоритм сортировки с использованием регулярного набора образцов (the parallel sorting by regular sampling) является обобщением метода быстрой сортировки. Упорядочивание данных в соответствии с данным вариантом алгоритма быстрой сортировки осуществляется в ходе выполнения следующих четырех этапов:

1) на первом этапе сортировки производится упорядочивание имеющихся на процессорах блоков. Данная операция может быть выполнена каждым процессором независимо друг от друга при помощи обычного алгоритма быстрой сортировки; далее каждый процессор формирует набор из элементов своих блоков с индексами 0, *m, 2m, ..., (p-1)m*, где *m=n/p2* ;

2) на втором этапе выполнения алгоритма все сформированные на процессорах наборы данных собираются на одном из процессоров системы и объединяются в ходе последовательного слияния в одно упорядоченное множество. Далее из полученного множества значений из элементов с индексами

$$p + [p/2] – 1, 2p + [p/2] – 1, \ldots, (p – 1)p + [p/2]$$

формируется новый набор ведущих элементов, который передается всем используемым процессорам. В завершение этапа каждый процессор выполняет разделение своего блока на *p* частей с использованием полученного набора ведущих значений;

3) на третьем этапе сортировки каждый процессор осуществляет рассылку выделенных ранее частей своего блока всем остальным процессорам системы; рассылка выполняется в соответствии с порядком нумерации – часть *j*, $0<=j<p$, каждого блока пересылается процессору с номером *j*;

4) на четвертом этапе выполнения алгоритма каждый процессор выполняет слияние *p* полученных частей в один отсортированный блок.

По завершении четвертого этапа исходный набор данных становится отсортированным.

Ниже приведен пример сортировки массива данных с помощью алгоритма, описанного выше. Следует отметить, что число процессоров для данного алгоритма может быть произвольным, в данном примере оно равно 3.

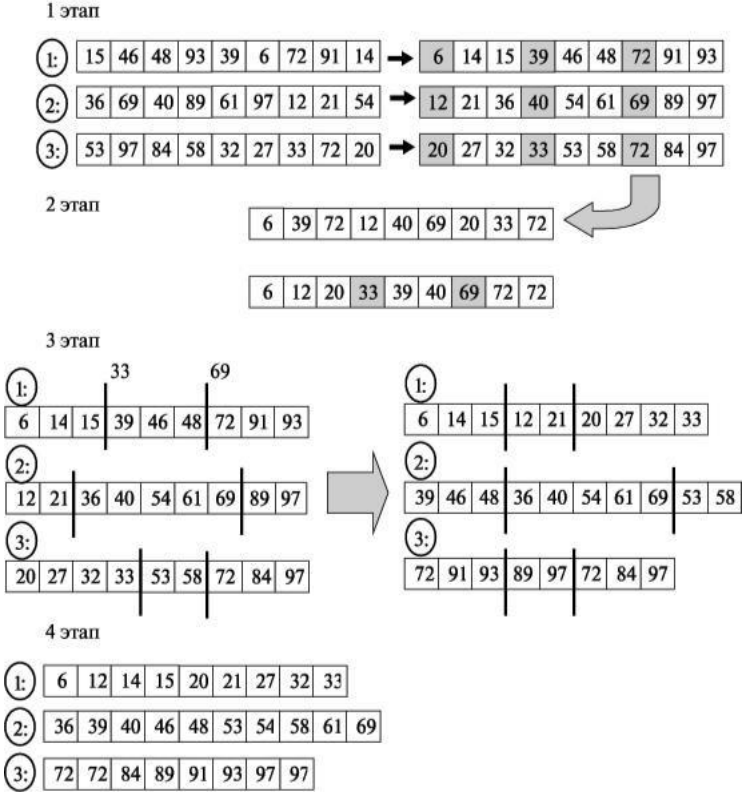


Рисунок 1.4 — Пример работы алгоритма сортировки с использованием регулярного набора образцов

2. Задание на работу.

Выполнить разработку и отладку программы быстрой сортировки данных с использованием вызовов требуемых функций библиотеки MPI для реализации варианта сортировки в соответствии с вариантом, указанным преподавателем. В качестве базового варианта реализовать также сортировку последовательным методом. Получить результаты работы программы в виде протоколов сообщений, комментирующих параллельное выполнение процессов и их взаимодействие в ходе выполнения. Оценить эффективность параллельного процесса сортировки в сравнении с последовательным на том же наборе исходных данных.

Таблица 7.1 — Варианты заданий

№ варианта	Вид сортировки
1	Быстрая сортировка
2	Модифицированная быстрая сортировка
3	Сортировка с использованиемрегулярного набора образцов

3. Контрольные вопросы

3.1. Назовите особенности реализации и использования рассматриваемых моделей взаимодействия распределенных процессов.
3.2. Сформулируйте понятие топологии кластера и остовного дерева, определите форматы рассылаемых сообщений, алгоритмы построения топологии кластера и остовного дерева, алгоритм рассылки.
3.3. Сформулируйте алгоритм реализации механизма «Распределенных семафоров», назначение логических часов и очереди сообщений, определите форматы передаваемых сообщений.
3.4. Сформулируйте алгоритм реализации модели «передачи маркера».

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Немнюгин С.А. Параллельное программирование для многопроцессорных вычислительных систем/ С.А. Немнюгин, О.А. Стесик - СПБ.: Издательство ” BHV”,2002.-400 с.
1. Эндрюс Г.Р. Основы многопоточного параллельного и распределенного программирования/ Г.Р. Эндрюс – М.: Издательство “Вильямс”, 2003.- 505 с.
3. Воеводин В.В. Параллельные вычисления/ В.В. Воеводин– СПБ.: Издательство ” BHV”,2002.-599 с.
4. Топорков В.В. Модели распределенных вычислений/ В.В. Топорков.– Изд-во ФИЗМАТЛИТ, 2004.– 320 с.
5. ГергельВ.П. Лекции по параллельным вычислениям/ В.П. Гергель, В.А.Фурсов.– Самара, Изд-во СГАЭУ, 2009. – 164 с.
6. Антонов А.С. Параллельное программирование с использованием технологии MPI/ А.С. Антонов.– М.:Изд-во МГУ, 2004. – 71 с.
7. Шпаковский Г.И. Применение технологии MPI в ГРИД/ Г.И. Шпаковский, В.И.Стецюренко, Н.В.Серикова.– Минск: Изд-во БГУ, 2008.– 140 с.
8. Хьюз К. Параллельное и распределенное программирование с использованием С++/ К. Хьюз, Т.Хьюз.– М.:Изд-во «Вильямс», 2004.– 672 с.
9. Руденко Ю.М., Волкова Е.А. “Вычислительные системы”.-М.: НИИ РЛ МГТУ им. Н.Э.Баумана, 2010.-212 с.
10. Демьянович Ю.К. Технология программирования для распределенных параллельных систем./ Ю.К. Демьянович, О.Н. Иванцова. – СПБ.: Изд-во СПГУ, 2005.– 90 с.
11. Головашкин Д. Л. Современные методы и алгоритмы решения сложных задач на суперкомпьютерах. / Д. Л. Головашкин.– Самара: Изд-во Самарского государственного аэрокосмического университета, 2010. – 104 с.

ПРИЛОЖЕНИЕ А
СОЗДАНИЕ СРЕДЫ ДЛЯ РАБОТЫ

1. Установка

Для разработки MPI-программ можно использовать различные реализации стандарта MPI. В данном приложении изложено описание работы с HPC PACK 2008 SDK. Заметим, что необязательно иметь Windows кластер или даже многоядерную/многопроцессорную рабочую станцию для разработки MPI-программ: любой настольный компьютер, который может работать с Windows XP может быть использован для разработки MPI-программ. Для корректной работы создаваемая среда для исполняемых программ требует, чтобы имя пользователя компьютера было на английском языке (проще всего создать нового) и путь к проекту не должен содержаться символы кириллицы.

1.1. VisualStudio

Первым шагом в подготовке к написанию MPI-программы является установка Microsoft Visual Studio. Стоит заметить, что от варианта издания Visual Studio будет зависеть дальнейшая работа с разрабатываемой MPI-программой, т.е. версии Express и Standard не поддерживают MPI-кластерную отладку, которая намного упрощает отслеживание ошибок. Но даже при её отсутствии существует возможность отлаживать MPI-программы, используя метод, который описан в данном приложении.

1.2. HPCPack

Для реализации кластерных MPI-программ Microsoft выпустил High Performance Computing Pack 2008 SDK. Также были разработаны Windows HPC Server 2008 или Microsoft Computer Cluster Server 2003, обеспечивающие поддержку развертывания MPI-программ. Для целей разработки MPI-программ достаточно использовать SDK.

2. Настройка

2.1. Создание проекта

Теперь, когда все установлено, необходимо создать проект. Для этого запустим VisualStudio, перейдем в меню **File** и выберем **New→Project**. Появится диалоговое окно, выберем консольное приложение, назовем его **HelloWorld** (желательно чтобы название не содержало пробелы), выберем папку для проекта и нажмем OK. В следующем окне нажмем **NEXT**, так как следует убрать опцию **Precompiled Header**, это необходимо для того чтобы в последующем можно было проще скомпилировать исходный код на другой платформе. Нажмем **FINISH**.

2.2. Изменение кода

Изменим код по умолчанию и скомпилируем программу, убедившись, что все работает правильно:

```
#include "stdafx.h"
```

```
#include <iostream>
using namespace std;
//int _tmain(int argc, _TCHAR* argv[])
int main(int argc, char* argv[])
{
    cout << "Hello World" << endl;
    return 0;
}
```

Есть несколько причин, по которым необходимо изменить **_tmain** на стандартный **main**. Во-первых, это было сделано для последующей совместимости на другой платформе. Во-вторых, нам позже понадобится **argv** как переменная типа **char***, а не **TCHAR***.

2.3. Конфигурация проекта

Теперь, когда программа скомпилирована и работает, нам необходимо её сконфигурировать, чтобы она включала в себя библиотеки и заголовочные файлы MPI. Выберем свойства (**Properties**), нажав на проект правой кнопкой в обозревателе решения (**SolutionExplorer**). В раскрывшемся окне выберем конфигурацию (**Configuration**) все конфигурации (**All Configurations**) и перейдем к пункту **VC++ Directories**категории**Configuration Properties**. Далее в ниспадающем меню **Include Directories** выберем **Edit**. Добавим новую строку и укажем путь к директории C:\Program Files\Microsoft HPC Pack 2008 SDK\Include. Нажмём кнопку OK. Аналогичные действия проделаем для **Library Directories**, указав в независимости от разрядности путь C:\Program Files\Microsoft HPC Pack 2008 SDK\Lib\i386. Далее необходимо указать от чего зависит проект. Для этого перейдем в категорию **Linker** и выберем пункт **Input**. Добавим через **Edit** к **Additional Dependencies** **msmpi.lib**. Нажмем OK и для проверки скомпилируем программу.

3. Запуск

3.1. Изменение кода

Необходимо выполнить изменение кода, чтобы можно было проверить результат работы библиотеки. Функций, которые используются, описаны в следующих разделах:

```
#include "stdafx.h"
#include <iostream>
#include "mpi.h"
using namespace std;
int main(int argc, char* argv[])
{
    int nTasks, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nTasks);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    cout << "Number of threads = " << nTasks << endl
         << "Hello World from rank " << rank << endl;
    MPI_Finalize();
    return 0;
}
```

3.2. Запуск программы в несколько потоков

Если теперь откомпилировать программу, она всегда будет показывать количество потоков равное 1, а ранг – 0. Для использования MPI необходимо запустить программу через приложение **mpiexec.exe**. Для этого запустим командную строку и перейдем к папке проекта. Введя **mpiexecHello-World.exe** мы скорее всего получим другой результат, который зависит от того какой процессор исполняет программу. Чтобы точно указать какое количество потоков мы хотим использовать, необходимо добавить спецификатор **-n**, т.е. **mpiexec -n 8 HelloWorld.exe**. Теперь программа будет выполняться в восемь потоков.

4. Отладка

Конечно же, запуск через командную строку не удобен и не позволяет отлаживать программы. Чтобы это исправить необходимо перейти к пункту **Debugging** категории **ConfigurationProperties** в свойствах проекта. Далее существует два способа начать отлаживать программу.

4.1. Прикрепление к процессам

В поле **Command** необходимо указать путь к программе C:\Program Files\Microsoft HPC Pack 2008 SDK\Bin\mpiexec.exe, в **CommandArguments** необходимо указать количество потоков и исполняемый файл, например вот так **-n 8 "$(TargetPath)"**. Далее необходимо добавить код, который позволит остановить на время все потоки программы, и прикрепить отладчик к процессам. Добавим следующий код после **MPI_Comm_rank (MPI_COMM_WORLD, &rank);**:

```
    if (rank == 0)
    {
        cout << "Press any key" << endl;
        cin.get();
    }
    MPI_Barrier(MPI_COMM_WORLD);
```

Теперь, когда программа ожидает ввода пользователя можно прикрепить отладчик к запущенным процессам. Для этого запустим отладку, необходимо подтвердить намерение отлаживать запускаемую программу, далее перейдем в меню **Debug** и выберем пункт **AttachtoProcess…**, где можно указать, к какому процессу необходимо прикрепить отладчик. Далее указав точ-

ку останова в программе можно её отлаживать. Еще очень полезно бывает видеть, какой процесс отлаживается, для этого необходимо открыть окно **Debug→Windows→Processes**.

4.2. Использование MPI-кластерного отладчика

Намного удобнее и проще использовать специально разработанный кластерный отладчик. Для его включения необходимо перейти к пункту **Debugging** и в ниспадающем меню **Debuggertolaunch:** выбрать **MPIClusterDebugger**. Далее в **RunEnvironment** можно указать узел и количество процессов. При запуске каждый процесс будет иметь свою собственную консоль.

ПРИЛОЖЕНИЕ Б
ТЕРМИНЫ И СОГЛАШЕНИЯ В MPI

1.1. Спецификация аргументов

Для каждой функции аргументы помечаются как **IN**, **OUT** или **INOUT**. Значение этого следующее:

- **IN**: вызов функции может использовать входное значение, но не обновляет аргумент;
- **OUT**: вызов функции может обновить аргумент, но не использует его входное значение;
- **INOUT**: вызов функции может как использовать, так обновлять аргумент.

К примеру:

```
void copyIntBuffer(int *pin, int *pout, int len)
{
    for (int i = 0; i < len; ++i)
        *pout++ = *pin++;
}
```

1.2.Семантические термины

При рассмотрении MPIфункций используются следующие семантические термины:

- **`nonblocking`**: функция является неблокируемой, если она может вернуть результат до завершения операции, и прежде чем пользователь может повторно использовать ресурсы (например такие как буфер) указанные в вызове. Если необходим данный ресурс, то обязательно следует проверить окончена ли операция над ним.
- **`blocking`**: функция является блокируемой, если возврат из процедуры указывает, что пользователь может повторно использовать ресурсы указанные в вызове.
- **`local`**: функция является локальной, если завершение процедуры зависит только от локально исполняемого процесса.
- **`non-local`**: функция является не локальной, если завершение операции может требовать выполнения некоторой MPIпроцедуры другим процессом. Такая операция может требовать связи с другим процессом.
- **`collective`**: функция является коллективной, если все процессы в группе требуют вызова данной процедуры. Коллективный вызов может быть как синхронным, так и асинхронным. Коллективные вызовы через один и тот же коммуникатор должны выполняться в том же самом порядке всеми членами группы.

- **predefined**: тип является предопределенным, если он имеет предопределенное (константное) имя (например **MPI_INT**) или тип, который конструируется при помощи функций.
- **derived**: тип является производным, если он не относится к какому-либо предопределенному типу.
- **portable**: тип является портативным, если он относится к предопределенному типу, или является производным от портативного типа, использовавшего конструктор типа MPI(например **MPI_TYPE_CREATE_DARRAY**). Такие типы являются портативными, потому что могут использоваться на вычислительных машинах с различной архитектурой процессоров или памяти.
- **equivalent**: два типа являются эквивалентными, если они созданы с одинаковым порядком вызовов (и аргументов) и поэтому имеют одинаковую разметку типа (typemap). Они не обязательно должны иметь одинаковые кэшированные атрибуты или одинаковые имена.

1.3.Типы данных

MPI управляет **системной памятью**, которая используется для буферизации сообщений и сохранения внутренних представлений различных MPI-объектов (групп, коммуникаторов, типов данных, и т.д.). Эта память не доступна напрямую пользователю, и объекты, сохраненные в ней, являются скрытыми (**opaque**): их размер и форма не видны пользователю. Скрытые объекты доступны через дескрипторы (**handles**), которые существуют в пользовательском пространстве. MPI-функции, которые оперируют скрытыми объектами, получают доступ к ним через аргументы дескриптора. Вдобавок к этому дескрипторы могут участвовать в присвоении и сравнении.

MPI-вызов может потребовать аргумент, который является массивом скрытых объектов, или массивом дескрипторов. Массив дескрипторов является обычным массивом с элементами, которые являются дескрипторами объектов того же типа, что и при последовательном расположении в массиве. Для массива необходим дополнительный аргумент **len**, чтобы указать количество действительных элементов (за исключением, когда это значение можно получить каким-либо другим образом). Действительные элементы находятся в начале массива, а **len** указывает, сколько их, но это не обязательно размер всего массива. В некоторых случаях **NULL-**дескриптор является обоснованно действительным элементом. Когда **NULL-**аргумент требуется для массива статусов, следует использовать **MPI_STATUS_IGNORE.**

MPI-функции используют в различных местах аргументы с типом состояния (**state**). Значения таких типов данных все определены по имени, и нет операций, определённых над ними. MPI-функции иногда используют специальные значения основных типов аргументов. Например, **tag** имеет целочисленное значение аргумента операций коммуникации точка-точка. Данный аргумент может принимать специальное значение **MPI_ANY_TAG.**

ПРИЛОЖЕНИЕ В
ПАРАЛЛЕЛЬНЫЕ МЕТОДЫ МАТРИЧНОГО УМНОЖЕНИЯ

1.1. Постановка задачи

Умножение матрицы *A* размера *m*n* и матрицы *B* размера *n*l* приводит к получению матрицы *C* размера *m*l*, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj},\ 0 \le i < m,\ 0 \le j < l$$

Каждый элемент результирующей матрицы *C* есть скалярное произведение соответствующих строки матрицы *A* и столбца матрицы *B*:

$$c_{ij} = (a_i, b_j^T),\ a_i = (a_{i0}, a_{i1} \ldots, a_{in-1}),\ b_j^T = (b_{0j}, b_{1j}, \ldots, b_{n-1j})^T.$$

Этот алгоритм предполагает выполнение *m*n*l* операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера *n*n* количество выполненных операций имеет порядок $O(n^3)$. Предполагается, что все матрицы являются квадратными и имеют размер *n*n*.

.

1.2. Последовательный алгоритм

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

```
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
  for (j=0; j<Size; j++){
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++){
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы *C*. Действительно, при выполнении одной итерации внешнего цикла (цикла по переменной *i*) вычисляется одна строка результирующей матрицы (см. рис. В.1).

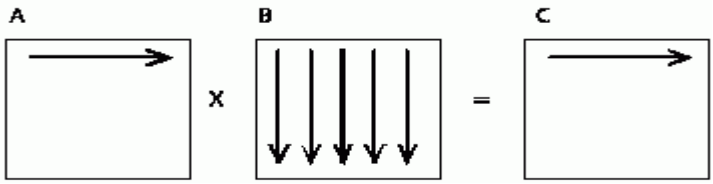


Рисунок В.1 – Схема получения результата на первой итерации цикла по переменной *i* (используется первая строка матрицы *A* и все столбцы матрицы *B* для того, чтобы вычислить элементы первой строки результирующей матрицы *C*)

Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы *C* размером *n*n* необходимо выполнить $n^2*(2n-1)$ скалярных операций и затратить время: $T = n^2 * (2n-1) * \tau$, где τ – время выполнения одной элементарной скалярной операции.

1.3. Умножение матриц при ленточной схеме разделения данных

Рассмотрим два параллельных алгоритма умножения матриц, в которых матрицы *A* и *B* разбиваются на непрерывные последовательности строк или столбцов (полосы).

1.3.1. Определение подзадач

Из определения операции матричного умножения следует, что вычисление всех элементов матрицы *C* может быть выполнено независимо друг от друга. Организация параллельных вычислений состоит в использовании в качестве базовой подзадачи процедуры определения одного элемента результирующей матрицы *C*. Для проведения всех необходимых вычислений каждая подзадача должна содержать по одной строке матрицы *A* и одному столбцу матрицы *B*. Общее количество получаемых при таком подходе подзадач равно n^2 (по числу элементов матрицы *C*). Достигаемый при этом уровень параллелизма является избыточным т.к. при проведении практических расчетов такое количество сформированных подзадач превышает число имеющихся процессоров и делает неизбежным этап укрупнения базовых задач. В этом этом случае выполняется агрегация вычислений на шаге выделения базовых подзадач. Возможное решение связано с объединением в рамках одной подзадачи всех вычислений, связанных не с одним, а с несколькими элементами результирующей матрицы *C*. Определим базовую задачу как процедуру вычисления всех элементов одной из строк матрицы С. Такой подход приводит к снижению общего количества подзадач до величины *n*.

Для выполнения всех необходимых вычислений базовой подзадаче должны быть доступны одна из строк матрицы *A* и все столбцы матрицы *B*. Простое решение – дублирование матрицы *B* во всех подзадачах – является неприемлемым в силу больших затрат памяти для хранения данных. Поэтому организация вычислений должна быть построена таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть данных, необходимых для проведения расчетов, а доступ к остальной части данных обеспечивался бы при помощи передачи данных между процессорами. Два возможных способа выполнения параллельных вычислений подобного типа рассмотрены в пункте 1.3.2.

1.3.2. Выделение информационных зависимостей

Для вычисления одной строки матрицы *C* необходимо, чтобы в каждой подзадаче содержалась строка матрицы *A* и был обеспечен доступ ко всем столбцам матрицы *B*. Возможные способы организации параллельных вычислений представлены следующими алгоритмами.

1. **Алгоритм 1.** Алгоритм представляет собой итерационную процедуру, количество итераций которой совпадает с числом подзадач. На каждой итерации алгоритма каждая подзадача содержит по одной строке матрицы *A* и одному столбцу матрицы *B*. При выполнении итерации проводится скалярное умножение содержащихся в подзадачах строк и столбцов, что приводит к получению соответствующих элементов результирующей матрицы *C*. По завершении вычислений в конце каждой итерации столбцы матрицы *B* должны быть переданы между подзадачами с тем, чтобы в каждой подзадаче оказались новые столбцы матрицы *B*, могли быть вычислены новые элементы матрицы *C*. При этом данная передача столбцов между подзадачами должна быть организована таким образом, чтобы после завершения итераций алгоритма в каждой подзадаче последовательно оказались все столбцы матрицы *B*. Простая схема организации последовательности передач столбцов матрицы *B* между подзадачами состоит в представлении топологии информационных связей подзадач в виде кольцевой структуры. В этом случае на каждой итерации подзадача *i*, $0<=i<n$, будет передавать свой столбец матрицы *B* подзадаче с номером $i+1$ (в соответствии с кольцевой структурой подзадача $n-1$ передает свои данные подзадаче с номером 0 – Рис. В.2). После выполнения всех итераций алгоритма необходимое условие будет обеспечено – в каждой подзадаче поочередно окажутся все столбцы матрицы *B*.

На Рис. В.2 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов ($n=4$). В начале вычислений в каждой подзадаче *i* (0<=i<n), располагаются *i* -я строка матрицы *A* и *i* -й столбец матрицы *B*. В результате их перемножения подзадача получает элемент c_{ii} результирующей матрицы *C*. Далее подзадачи осуществляют обмен столбцами, в ходе которого каждая подзадача передает свой столбец матрицы *B* следующей подзадаче в соответствии с кольцевой

структурой информационных взаимодействий. Выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

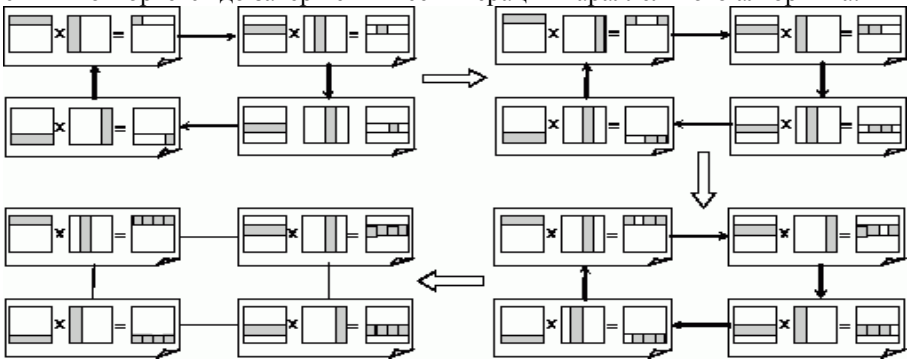


Рисунок В.2– Схема передачи данных для первого параллельного алгоритма матричного умножения (ленточная схема разделения данных)

2. **Алгоритм 2.** Алгоритм предполагает, что в подзадачах располагаются не столбцы, а строки матрицы *B*. Как результат, перемножение данных каждой подзадачи сводится не к скалярному умножению имеющихся векторов, а к их поэлементному умножению. В результате подобного умножения в каждой подзадаче получается строка частичных результатов для матрицы *C*. При рассмотренном способе разделения данных для выполнения операции матричного умножения нужно обеспечить последовательное получение в подзадачах всех строк матрицы *B*, поэлементное умножение данных (элементов строк матриц *A* и *B)* и суммирование вновь получаемых значений с ранее вычисленными результатами. Организация необходимой последовательности передач строк матрицы *B* между подзадачами также может быть выполнена с использованием кольцевой структуры информационных связей (рис. В.3).

На рис. В.3 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов ($n=4$). В начале вычислений в каждой подзадаче i ($0<=i<n$), располагаются i-е строки матриц *A* и *B*. В результате их перемножения подзадача определяет i -ю строку частичных результатов искомой матрицы *C*. Далее подзадачи осуществляют обмен строками, в ходе которого каждая подзадача передает свою строку матрицы *B* следующей подзадаче в соответствии с кольцевой структурой информационных связей (вновь определяется i -я строка частичных результатов искомой матрицы *C*, элементы которой складываются с частичными результатами с предыдущей итерации) . Далее выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

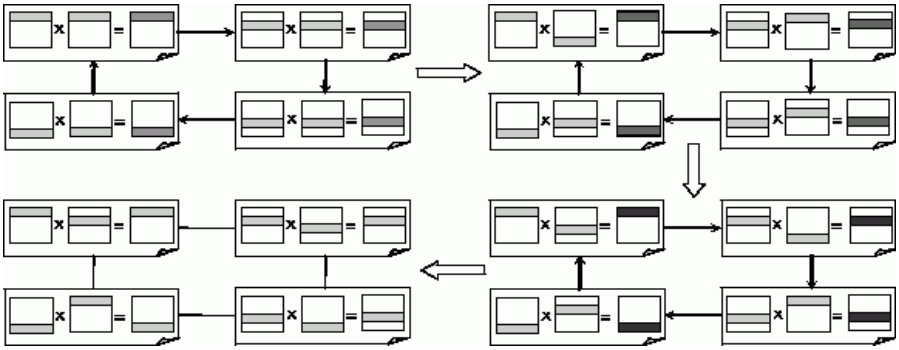


Рисунок В.3– Схема передачи данных для второго параллельного алгоритма матричного умножения (ленточная схема разделения данных)

1.4. Алгоритмы умножения матриц при блочном разделении данных

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Рассмотрим более подробно данный способ организации вычислений. При таком способе разделения данных исходные матрицы *A*, *B* и результирующая матрица *C* представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера *n*n*, количество блоков по горизонтали и вертикали одинаково и равно *q* (размер каждого блока равен *k*k*, *k=n/q*). При таком представлении данных операция матричного умножения матриц *A* и *B* в блочном виде может быть представлена так:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0\,q-1} \\ \dots & \dots & \dots & \dots \\ A_{q-1\,0} & A_{q-1\,1} & \dots & A_{q-1\,q-1} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0\,q-1} \\ \dots & \dots & \dots & \dots \\ B_{q-1\,0} & B_{q-1\,1} & \dots & B_{q-1\,q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0\,q-1} \\ \dots & \dots & \dots & \dots \\ C_{q-1\,0} & C_{q-1\,1} & \dots & C_{q-1\,q-1} \end{pmatrix},$$

каждый блок Cij матрицы C определяется в соответствии с выражением

вида $C_{ij} = \sum_{s=0}^{q-1} A_{is} \cdot B_{sj}$, где A_{is} и B_{sj} – соответствующие блоки матриц *A* и *B*, которые перемножаются между собой, а затем получены результаты складываются для получения блока C_{ij} матрицы C. При блочном разбиении данных базовым подзадачам требуется реализовывать вычисления, выполняемые над матричными блоками. С учетом сказанного базовая подзадача– это процедура вычисления всех элементов одного из блоков матрицы *C*.

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы *A* и столбцов матрицы *B*. Размещение всех требуемых данных в каждой подзадаче неизбежно приведет к дублированию и к значительному росту объема используемой памяти. Вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы при помощи передачи данных между процессорами. Возможным подходом к решению задачи распределения данных (обмена данными) является алгоритм Фокса.

1.4.1. Алгоритм Фокса. Выделение информационных зависимостей

За основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы *C* и при этом в подзадачах на каждой итерации расчетов располагается только по одному блоку исходных матриц *A* и *B*. Для нумерации подзадач использованы индексы размещаемых в подзадачах блоков матрицы *C*, т.е. подзадача *(i,j)* реализует вычисление блока C_{ij} . Тогда набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче *(i,j)* располагается четыре матричных блока:

- блок C_{ij} матрицы *C*, вычисляемый подзадачей;
- блок A_{ij} матрицы *A*, размещаемый в подзадаче перед началом вычислений;
- блоки A'_{ij} , B'_{ij} матриц *A* и *B*, получаемые подзадачей в ходе выполнения вычислений.

Выполнение параллельного метода включает:

1) этап инициализации, на котором каждой подзадаче *(i,j)* передаются блоки A_{ij} , B_{ij} и обнуляются блоки C_{ij} на всех подзадачах;

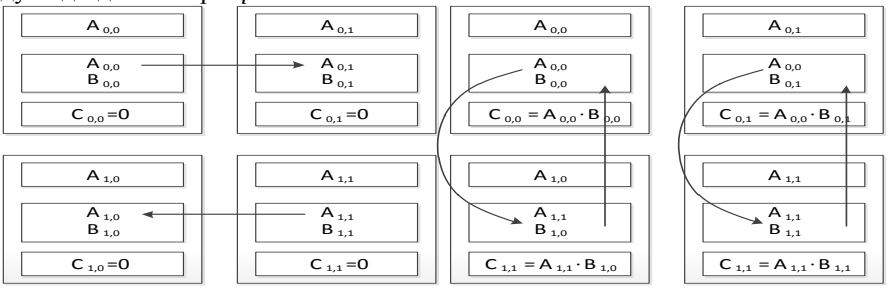
2) этап вычислений, в рамках которого на каждой итерации *l* (*0<=l<q,* т.е. всего *q* итераций*)*, осуществляются следующие операции:

- для каждой строки i ($0<=i<q$) блок A_{ij} подзадачи (i,j) пересылается подзадаче этой же строки i решетки; индекс j, определяющий положение подзадачи в строке (которой пересылается блок A_{ij}), вычисляется в соответствии с выражением $j = (i + l) \bmod q$, где *mod* есть операция получения остатка от целочисленного деления;

- полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждой подзадачи (i,j) перемножаются и прибавляются к блоку C_{ij}.

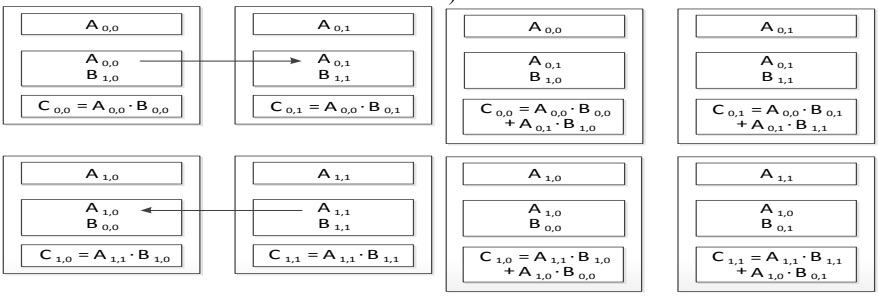
- блоки B'_{ij} каждой подзадачи (i,j) пересылаются подзадачам, являющимся соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Напомним, что число итераций алгоритма поблочного матричного перемножения равно количеству блоков q, на которое разбиваются исходные матрицы A и B. На Рис.В.4 представлена схема обмена блоками матриц A и B между подзадачами при $q=2$.



Обмен в строке Перемножение и обмен в столбце

а)



Обмен в строке Перемножение и суммирование

б)

Рисунок В.4– Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса.

а) первая итерация алгоритма; б) вторая итерация алгоритма

На Рис.В.5 представлена схема обмена блоками матриц *A* и *B* между подзадачами при *q=4.*

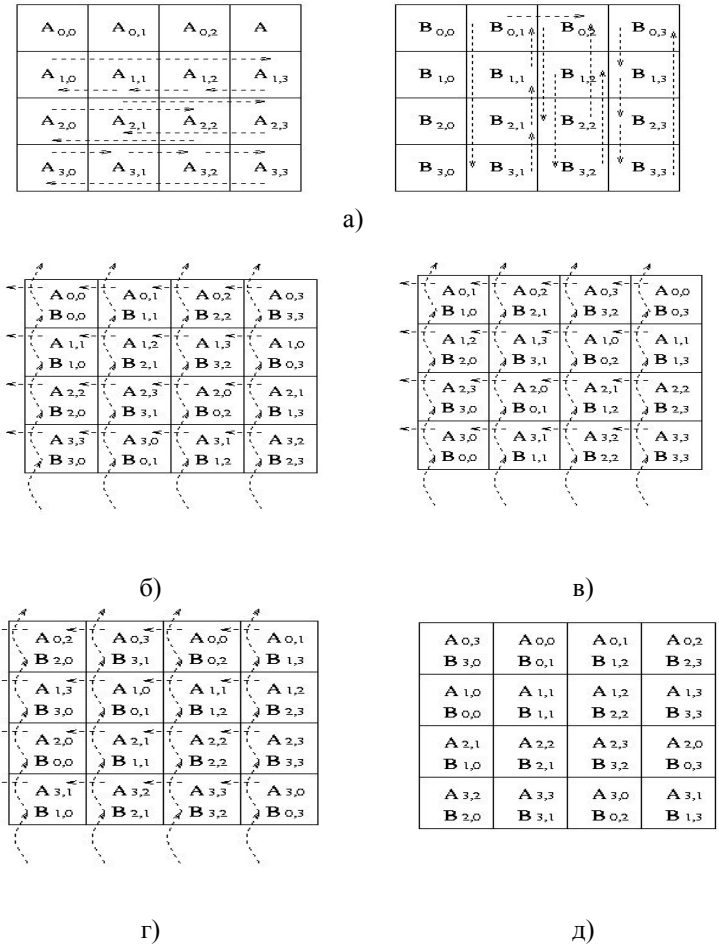


Рисунок В.5– Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса при *q=4*.

а) первоначальный обмен блоками матриц; б) умножение блоков и обмен (1 шаг); в) умножение блоков и обмен (2 шаг); г) умножение блоков и обмен (3 шаг);д) умножение блоков (4 шаг);

1.4.2. Программная реализация алгоритма Фокса

Представим возможный вариант *программной реализации* алгоритма Фокса для *умножения матриц* при блочном представлении данных. Приво-

димый программный код содержит основные модули параллельной программы, отсутствие отдельных вспомогательных функций не сказывается на общем понимании реализуемой схемы параллельных вычислений.

Главная функция программы. Определяет основную логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Условия выполнения программы: все матрицы квадратные,
// размер блоков и их количество по горизонтали и вертикали
// одинаково, процессы образуют квадратную решетку
int ProcNum = 0;     // Количество доступных процессов
int ProcRank = 0;    // Ранг текущего процесса
int GridSize;        // Размер виртуальной решетки процессов
int GridCoords[2];   // Координаты текущего процесса в процес
сной
                     // решетке
MPI_Comm GridComm;   // Коммуникатор в виде квадратной решетк
и
MPI_Comm ColComm;    // коммуникатор – столбец решетки
MPI_Comm RowComm;    // коммуникатор – строка решетки
void main ( int argc, char * argv[] )
{
    double* pAMatrix;   // Первый аргумент матричного умноже
ния
    double* pBMatrix;   // Второй аргумент матричного умноже
ния
    double* pCMatrix;   // Результирующая матрица
    int Size;           // Размер матриц
    int BlockSize;      // Размер матричных блоков, располож
енных
                        // на процессах
    double *pAblock;    // Блок матрицы A на процессе
    double *pBblock;    // Блок матрицы B на процессе
    double *pCblock;    // Блок результирующей матрицы C на
процессе
    double *pMatrixAblock;
    double Start, Finish, Duration;
    setvbuf(stdout, 0, _IONBF, 0);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    GridSize = sqrt((double)ProcNum);
    if (ProcNum != GridSize*GridSize)
```

```
    {
        if (ProcRank == 0)
        {
            printf ("Number of processes must be a perfect s
quare \n");
        }
    }
    else
    {
        if (ProcRank == 0)
            printf("Parallel matrix multiplication program\n
");

        // Создание виртуальной решетки процессов и коммуник
аторов строк и столбцов
        CreateGridCommunicators();
        // Выделение памяти и инициализация элементов матриц
       ProcessInitializa-
tion ( pAMatrix, pBMatrix, pCMatrix, pAblock,pBblock, pCbloc
k, pMatrixAblock, Size, BlockSize );
        // Блочное распределение матриц между процессами
        DataDistribu-
tion(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
            BlockSize);
        // Выполнение параллельного метода Фокса
        ParallelResultCalcula-
tion(pAblock, pMatrixAblock, pBblock,
            pCblock, BlockSize);
        // Сбор результирующей матрицы на ведущем процессе
       ResultCollection(pCMatrix, pCblock, Size, BlockSize);
        // Завершение процесса вычислений
        ProcessTermina-
tion (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock, pCbloc
k, pMatrixAblock);
    }
    MPI_Finalize();
}
```

Функция `CreateGridCommunicators`. Данная функция создает коммуникатор в виде двумерной квадратной решетки, определяет координаты каждого процесса в этой решетке, а также создает коммуникаторы отдельно для каждой строки и каждого столбца.

Создание решетки производится при помощи функции **MPI_Cart_create**(вектор **Periodic** определяет возможность передачи сообщений между граничными процессами строк и столбцов создаваемой решетки). После создания решетки каждый процесс параллельной программы будет иметь координаты своего положения в решетке; получение этих координат обеспечивается при помощи функции **MPI_Cart_coords**.

Формирование топологий завершается созданием множества коммуникаторов для каждой строки и каждого столбца решетки в отдельности (функция **MPI_Cart_sub**).

```
// Создание коммуникатора в виде двумерной квадратной решетки
// и коммуникаторов для каждой строки и каждого столбца решетки
void CreateGridCommunicators()
{
    int DimSize[2];     // Количество процессов в каждом измерении
                        // решетки
    int Periodic[2];    // =1 для каждого измерения, являющегося
                        // периодическим
    int Subdims[2];     // =1 для каждого измерения, оставляемого
                        // в подрешетке
    DimSize[0] = GridSize;
    DimSize[1] = GridSize;
    Periodic[0] = 0;
    Periodic[1] = 0;
    // Создание коммуникатора в виде квадратной решетки
    MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1,
 &GridComm);
    // Определение координат процесса в решетке
    MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);
    // Создание коммуникаторов для строк процессной решетки
    Subdims[0] = 0; // Фиксация измерения
    Subdims[1] = 1; //Наличие данного измерения в подрешетке
    MPI_Cart_sub(GridComm, Subdims, &RowComm);
    // Создание коммуникаторов для столбцов процессной решетки
    Subdims[0] = 1;
    Subdims[1] = 0;
    MPI_Cart_sub(GridComm, Subdims, &ColComm);
```

```
}
```

Функция `ProcessInitialization`. Данная функция определяет параметры решаемой задачи (размеры матриц и их блоков), выделяет память для хранения данных и осуществляет ввод исходных матриц (или формирует их при помощи какого-либо датчика случайных чисел). Всего в каждом процессе должна быть выделена память для хранения четырех блоков – для указателей на выделенную память используются переменные **`pAblock`**, **`pBblock`**, **`pCblock`**, **`pMatrixAblock`**. Первые три указателя определяют блоки матриц A, B и C соответственно. Следует отметить, что содержимое блоков **`pAblock`** и **`pBblock`**постоянно меняется в соответствии с пересылкой данных между процессами, в то время как блок **`pMatrixAblock`** матрицы A остается неизменным и применяется при рассылках блоков по строкам решетки процессов (см. функцию **`AblockCommunication`**).

Для определения элементов исходных матриц будем использовать функцию **`RandomDataInitialization`**, реализацию которой читателю предстоит выполнить самостоятельно.

```
// Функция для выделения памяти и инициализации исходных дан
ных
void ProcessInitialization (double* &pAMatrix, double* &pBMa
trix,
    double* &pCMatrix, double* &pAblock, double* &pBblock,
    double* &pCblock, double* &pTemporaryAblock, int &Size,
    int &BlockSize )
{
    if (ProcRank == 0)
    {
        do
        {
            printf("\nВведите размер матриц: ");
            scanf("%d", &Size);

            if (Size%GridSize != 0)
            {
                printf ("Размер матриц должен быть кратен ра
змеру сетки! \n");
            }
        }
        while (Size%GridSize != 0);
    }
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    BlockSize = Size/GridSize;
    pAblock = new double [BlockSize*BlockSize];
```

```
    pBblock = new double [BlockSize*BlockSize];
    pCblock = new double [BlockSize*BlockSize];
    pTemporaryAblock = new double [BlockSize*BlockSize];
    for (int i=0; i<BlockSize*BlockSize; i++)
    {
        pCblock[i] = 0;
    }
    if (ProcRank == 0)
    {
        pAMatrix = new double [Size*Size];
        pBMatrix = new double [Size*Size];
        pCMatrix = new double [Size*Size];
        RandomDataInitialization(pAMatrix, pBMatrix, Size);
    }
}
```

Функции `DataDistribution` и `ResultCollection`. После задания исходных матриц на нулевом процессе необходимо осуществить распределение исходных данных. Для этого предназначена функция **`DataDistribution`**. Может быть предложено два способа выполнения блочного разделения матриц между процессорами, организованными в двумерную квадратную решетку. В первом из них для организации передачи блоков в рамках одной и той же коммуникационной операции можно сформировать средствами MPI производный тип данных. Во втором способе можно организовать двухэтапную процедуру. На первом этапе матрица разделяется на горизонтальные полосы. Эти полосы распределяются на процессы, составляющие нулевой столбец процессорной решетки. Далее каждая полоса разделяется на блоки между процессами, составляющими строки процессорной решетки. Для выполнения сбора результирующей матрицы из блоков предназначена функция **`ResultCollection`**. Сбор данных также можно выполнить либо с использованием производного типа данных, либо при помощи двухэтапной процедуры, зеркально отображающей процедуру распределения матрицы. Реализация функций **`DataDistribution`** и **`ResultCollection`** представляет собой задание для самостоятельной работы. **Функция `AblockCommunication`.** Функция выполняет рассылку блоков матрицы *A* по строкам процессорной решетки. Для этого в каждой строке решетки определяется ведущий процесс **`Pivot`**, осуществляющий рассылку. Для рассылки используется блок **`pMatrixAblock`**, переданный в процесс в момент начального распределения данных. Выполнение операции рассылки блоков осуществляется при помощи функции **`MPI_Bcast`**. Следует отметить, что данная операция является коллективной и ее локализация пределами отдельных строк решетки

обеспечивается за счет использования коммуникаторов **RowComm**, определенных для набора процессов каждой строки решетки в отдельности.

```
// Рассылка блоков матрицы A по строкам решетки процессов
void ABlockCommunication (int iter, double *pAblock,
    double* pMatrixAblock, int BlockSize)
{
    // Определение ведущего процесса в строке процессной реш
етки
    int Pivot = (GridCoords[0] + iter) % GridSize;

    // Копирование передаваемого блока в отдельный буфер пам
яти
    if (GridCoords[1] == Pivot)
    {
        for (int i=0; i<BlockSize*BlockSize; i++)
            pAblock[i] = pMatrixAblock[i];
    }
    // Рассылка блока
    MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivo
t, RowComm);
}
```

Функция `BlockMultiplication`. Функция обеспечивает перемножение блоков матриц *A* и *B*. Следует отметить, что для более легкого понимания рассматриваемой программы приводится простой вариант реализации функции – выполнение операции блочного умножения может быть существенным образом оптимизировано для сокращения времени вычислений. Данная оптимизация может быть направлена, например, на повышение эффективности использования кэша процессоров, векторизации выполняемых операций и т.п.

```
// Умножение матричных блоков
void BlockMultiplication (double *pAblock, double *pBblock,
    double *pCblock, int BlockSize)
{
    // Вычисление произведения матричных блоков
    for (int i=0; i<BlockSize; i++)
    {
        for (int j=0; j<BlockSize; j++)
        {
            double temp = 0;
            for (int k=0; k<BlockSize; k++ )
                temp += pAblock [i*BlockSize + k] * pBblock
[k*BlockSize + j];
```

```
            pCblock [i*BlockSize + j] += temp;
        }
    }
}
```

Функция `BblockCommunication`. Функция выполняет циклический сдвиг блоков матрицы *B* по столбцам процессорной решетки. Каждый процесс передает свой блок следующему процессу **`NextProc`** в столбце процессов и получает блок, переданный из предыдущего процесса **`PrevProc`** в столбце решетки. Выполнение *операций передачи данных* осуществляется при помощи функции **`MPI_SendRecv_replace`**, которая обеспечивает все необходимые пересылки блоков, используя при этом один и тот же буфер памяти **`pBblock`**. Кроме того, эта функция гарантирует отсутствие возможных тупиков, когда *операции передачи данных* начинают одновременно выполняться несколькими процессами при кольцевой топологии сети.

```
// Циклический сдвиг блоков матрицы B вдоль столбца процессн
ой решетки
void BblockCommunication (double *pBblock, int BlockSize)
{
    MPI_Status Status;
    int NextProc = GridCoords[0] + 1;
    if ( GridCoords[0] == GridSize-1 )
        NextProc = 0;
    int PrevProc = GridCoords[0] - 1;
    if ( GridCoords[0] == 0 )
        PrevProc = GridSize-1;
    MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_
DOUBLE,
        NextProc, 0, PrevProc, 0, ColComm, &Status);
}
```

Функция `ParallelResultCalculation`. Для непосредственного выполнения параллельного алгоритма Фокса *умножения матриц* предназначена функция **`ParallelResultCalculation`**, которая реализует логику работы алгоритма.

```
// Функция для параллельного умножения матриц
void ParallelResultCalculation(double* pAblock, double* pMat
rixAblock,
    double* pBblock, double* pCblock, int BlockSize)
{
    for (int iter = 0; iter < GridSize; iter ++)
    {
        // Рассылка блоков матрицы A по строкам процессной р
ешетки
```

```
        ABlockCommunica-
tion (iter, pAblock, pMatrixAblock, BlockSize);
        // Умножение блоков
        BlockMultiplica-
tion(pAblock, pBblock, pCblock, BlockSize);
        // Циклический сдвиг блоков матрицы B в столбцах про
цессной решетки
        BblockCommunication(pBblock, BlockSize);
    }
}
```