**МГАПИ**

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
**МОСКОВСКАЯ ГОСУДАРСТВЕННАЯ АКАДЕМИЯ
ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ**

Кафедра ‘Персональные компьютеры и сети’

В.М.Баканов, Д.В.Осипов

# Введение в практику разработки параллельных программ в стандарте MPI

Учебно-методическое пособие по
выполнению лабораторных работ

Москва 2005

УДК 681.3

Введение в практику разработки параллельных программ в стандарте MPI: Учебно-методическое пособие по выполнению лабораторных работ / В.М.Баканов, Д.В.Осипов. -М.: МГАПИ, 2005. –63 с.: ил.

Предлагаемое учебное пособие предназначено для подготовки студентов III-V курсов различных форм обучения по специальности ‘Вычислительные машины, комплексы, системы и сети’, Пособие может использоваться студентами для подготовки к выполнению лабораторных и практических работ и курсовых/дипломных проектов.

В работе рассматриваются основы технологии создания параллельных программ в стандарте MPI (*Message Passing Interface*), занимающем ведущее место среди способов параллельного программирования при разработке прикладных задач с большим объемом вычислений.

Разобраны основные функции библиотеки MPI, стандартные приемы MPI-программирования, возможности загрузчика исполняемых программ. Знание основ программирования на языке C/C++ априори предполагается. При разработке стандартных MPI-программ дискуссируются стратегии распараллеливания. Создаваемые сетевые приложения работоспособны в многопроцессорной среде архитектуры MPP (*Massively Parallel Processes*); рассматривается, в частности виртуальный Linux-кластер на основе сети Windows-ПЭВМ.

Последняя версия данного методического пособия может быть получена в виде файла http://pilger.mgapi.edu/metods/1441/mpi_lab.zip.

Рецензент: профессор _____

## Содержание

Стр.

## Введение: общая информация о технологии программирования в стандарте MPI

MPI (*Message Passing Interface,* http://www.mpi-forum.org) является технологией создания параллельно выполняющихся программ, основанной на передаче сообщений между процессами (сами процессы могут выполняться как на одном, так и на различных вычислительных узлах). MPI-технология является типичным примером императивной (основанной на полном управлении программистом последовательности вычислений и распределения данных) технологии создания программ для параллельного выполнения [1÷3, 5].

Значительное число современных систем разработки параллельных программ основаны на императивности в распределении данных по процессорам; подобной MPI императивностью обладает система DVM (*Distributed Virtual Memory* или *Distributed Virtual Machine*, http://www.keldysh.ru/dvm) - в противоположность системе HPF (*Hight Performance Fortran,* http://www.erc.msstate.edu/hpff/home.html), где программист не предписывает компилятору конкретного распределения вычислений по процессорам, при этом существенная часть работ по распараллеливанию вычислений выполняется компилятором [3]. Свободно распространяемым инструментом автоматизации создания параллельных MPI-программ является система НОРМА (http://www.keldysh.ru/pages/norma), позволяющая (на основе специализированного декларативного языка) синтезировать исходный текст с вызовами MPI [1,2]; из иных (истинная эффективность неизвестна) можно назвать препроцессор пакета Forge (http://www.tc.cornell.edu/UserDoc/Software/PTools/forge). Полуавтоматическое распараллеливание достигается введением в состав универсального языка программирования (C или Fortran) дополнительных инструкций для записи параллельных конструкций кода и данных, препроцессор ‘переводит’ исходный текст в текст на стандартном языке с вызовами MPI; примеры: система mpC (*massively parallel C*, http://www.ispras.ru/~mpc) и HPF (*High Performance Fortran*, http://www.crpc.rice.edu/HPFF).

Формально MPI-подход основан на включении в программные модули в**ы**зовов функций специальной библиотеки (загол**о**вочные и библиотечные файлы для языков C/C++ и Fortran) и загрузчика параллельно исполняемого кода в вычислительные узлы (ВУ). Подобные библиотеки имеются практически для все платформ, поэтому написанные с использованием технологии MPI взаимодействия ветвей параллельного приложения программы независимы от машинной архитектуры (могут исполняться на однопроцессорных и многопроцессорных с общей или разделяемой памятью ЭВМ), от расположения ветвей (выполнение на одном или разных процессорах) и от API (*Application Program Interface*) конкретной операционной системы (ОС).

История MPI начинается в 1992 г. созданием стандарта эффективной и перенос**и**мой библиотеки передачи сообщений в *Oak Ridge National Laboratory (Rice University)*, окончательный вариант стандарта MPI 1.0 представлен в 1995 г., стандарт MPI-2 опубликован в 1997 г.

Из реализаций MPI известны MPICH (все UNIX-системы и Windows'NT, разработка *Argonne National Lab*., http://www.mcs.anl.gov), LAM (UNIX-подобные ОС, *Ohio Supercomputer Center*, http://www.osc.edu и *Notre-Damme University*, http://www.lsc.nd.edu), CHIMP/MPI (*Edinburgh Parallel Computing Centre*, http://www.epcc.ed.ac.uk), WMPI (все версии Windows, *разработка University of Coimbra*, *Portugal,* http://dsg.dei.uc.pt/wmpi; в 2002 г. прав**а** перешли к *Critical Software SA*, wmpi-sales@criticalsoftware.com) и др. (подробнее см. http://parallel.ru/mpi.html).

Основными отличиями стандарта MPI-2 являются: динамическое порождение процессов, параллельный ввод/вывод, интерфейс для C++, расширенные коллективные операции, возможность одностороннего взаимодействия процессов (см. [1] и http://www.mpi-forum.org). В настоящее время MPI-2 практически не используется; существующие реализации поддерживают только MPI 1.1.

MPI является (довольно) низкоуровневым инструментом программиста; с помощью MPI созданы рассчитанные на численные методы специализированные библиотеки (например, включающая решение систем линейных уравнений, обращение матриц, ортогональные преобразования, поиск собственных значений и т.п. библиотеки ScaLAPACK, http://www.netlib.org/scalapack и AZTEC, http://www.cs.sandia.gov/CRF/aztec1.html для плотнозаполненных и разреженных матриц соответственно). MPI не обеспечивает механизмов задания начального размещения процессов по вычислительным узлам (процессорам); это должен явно задать программист или воспользоваться неким стор**о**нним механизмом.

Разработчики MPI (справедливо) подвергаются жесткой критике за излишнюю громоздкость и сложность для реального (прикладного) программиста. Интерфейс оказался сложным и для реализации; в итоге в настоящее время практически не существует реализаций MPI, в полной мере обеспечивающие совмещение обменов с вычислениями. Проект MPI-2 выглядит еще более громоздким и тяжелоподъемным для реализации в заявленной степени.

В состав функций MPI входят специальные в**ы**зовы для обработки исключительных ситуаций и отладки. Цель обработчиков состоит в выдаче пользователю (определенное им же) сообщение об ошибке для принятия действий, не относящиеся к MPI (напр., очистка буферов ввода/вывода) перед выходом из программы. Реализация MPI допускает продолжение работы приложения после возникновения ошибки.

Полномасштабная отладка MPI-программ достаточно сложн**а** вследствие одновременного исполнения нескольких программных ветвей (при этом традиционная отладки методом включения в исходный текст операторов печати затруднена вследствие смешивания в файле выдачи информации от различных ветвей MPI-программы); одним из наиболее мощных отладчиков считается TotalView (http://www.dolphinics.com). В функции подобных отладчиков входит трассировка программы и профилирование – в**ы**дача (часто в графическом виде) информации об обм**е**нах в параллельной программе.

Строго говоря, технология MPI подразумевает подх**о**д MPMD (*Multiple Program – Multiple Data*: *множество программ – множество данных*), при этом одновременно (и относительно независимо друг от друга) на различных ВУ выполняются несколько (базирующихся на различных исходных текстах) программных ветвей, в определенные промежутки времени обменивающиеся данными. Однако подобные программы слишком гром**о**здки при написании (для каждого ВУ требуется отдельный исходной текст), поэтому на практике применяется SPMD-подход (*Single Program - Multiple Data: одна программа – множество данных*), предполагающий исполнение на различных ВУ логически выделенных условными операторами участков идентичного кода. Все ветви программы запускаются загрузчиком одновременно как процессы UNIX; количество ветвей фиксировано – согласно стандарту MPI 1.1 в ходе работы порождение новых ветвей невозможно.

Параллельно выполняющиеся программы обычно не используют привычного пользователям ОС Windows оконного интерфейса (вследствие как трудности связывания окон с конкретным экземпляром приложения, так и несущественности использования подобного интерфейса при решении серьезных счетных задач). Отладка параллельного приложения также существенно отличается от таков**о**й обычного последовательного [3].

В дальнейшем будем считать, что параллельное приложение состоит из нескольких ветвей (или процессов, или задач), выполняющихся одновременно на ВУ; при этом процессы обмениваются друг с другом данными в виде сообщений (рис.1). Каждое сообщение имеет идентификатор, который позволяет программе и библиотеке связи отличать их друг от друга. В MPI существует понятие области связи; области связи имеют независимую друг от друга нумерацию процессов (коммуникатор как раз и определяет область связи).

В общем случае создание параллельной программы включает в себя (укрупненно) две основные стадии:

- Исходно-последовательный алгоритм подвергается декомпозиции (распараллеливанию), т.е. разбивается на независимо работающие ветви; для взаимодействия в ветви вводятся два типа дополнительных нематематических операции: прием (Send) и передача (Receive) данных.
- Распараллеленный алгоритм оформляется в виде программы, в которой операции приема и передачи записываются в терминах конкретной системы связи между ветвями (в нашем случае MPI).

Несколько подробнее этот процесс описан в [4], для желающих серьезно заняться проблемой рекомендуется работа [1].

Т.к. время обмена данными между ветвями намного (на порядки) больше времени д**о**ступа к собственной (локальной) памяти, распределение работы между процессами должно быть ‘крупнозернистым’ [1]. Типичный размер этого зерна (*гр**а**нулы*) составляет десятки-сотни тысяч машинных операций

(что на порядки превышает типичный размер оператора языков Fortran или C/C++, [3]). В зависимости от размеров гранул говорят о мелкозернистом и крупнозернистом параллелизме (*fine-grained parallelism* и *coarse-grained parallelism*). На размер гр**а**нулы влияет и удобство программирования – в виде гранулы часто оформляют некий логически законченный фрагмент программы. Целесообразно стремиться к равномерной загрузке процессоров (как по количеству вычислительных операций, так и по загрузке оперативной памяти).

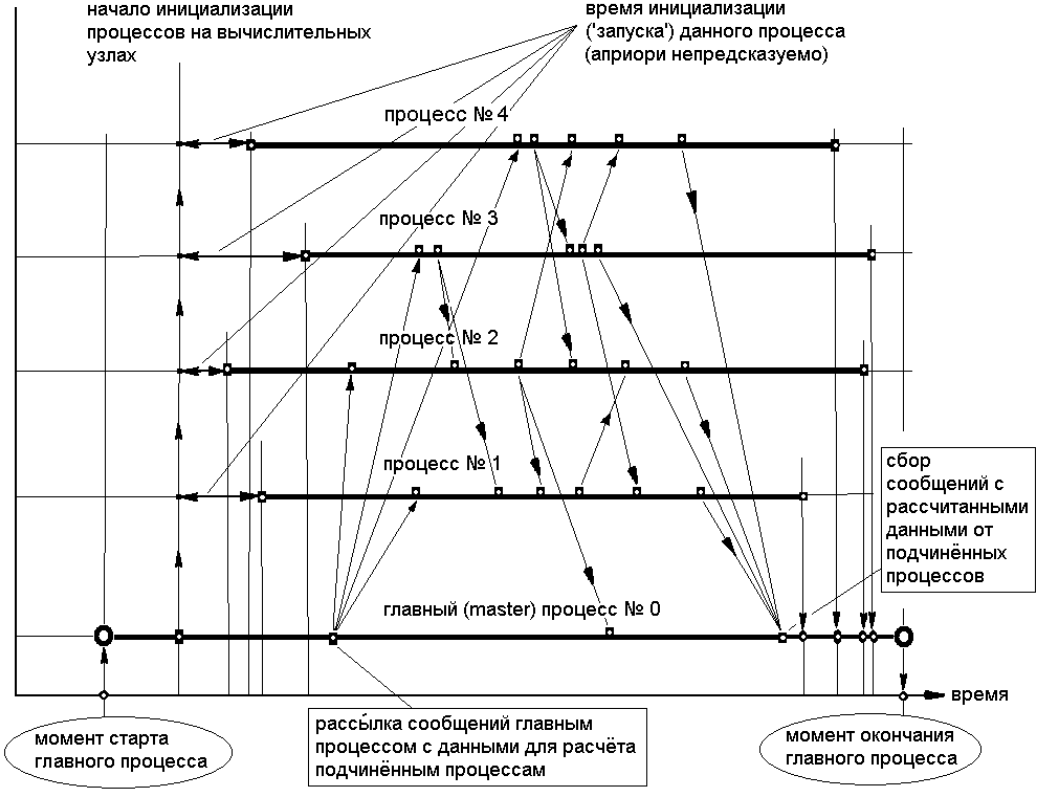



Рисунок 1.— Схема жизненного цикла процессов и обмена данными между ними

Система связи, в свою очередь, включает в себя программный и аппаратный компоненты; в рамках данной работы в основном рассматривается первый.

С точки же зрения программиста б**а**зовых методик работы (*парадигм*) существуют две - данные могут передаваться через разделяемую оперативную память (ОП, причем синхронизация д**о**ступа ветвей к такой памяти происходит с использованием механизма семафоров) и в виде сообщений. Первый метод является основным для ЭВМ с общей (физически или логически) для всех процессоров памятью, второй – для связи ВУ посредством сети; любая из этих дух парадигм может быть иммитирована другой.

В MPI определены три категории функций - блокирующие, локальные, коллективные:

- Блокирующие функции останавливают (блокируют) выполнение процесса до тех пор, пока производимая ими операция не будет выполнена. В про-

тивовес этому неблокирующие функции возвращают управление вызвавшей их программе немедленно, а выполнение операции продолжается в фоновом режиме; за завершением операции надо проследить особо. Неблокирующие функции возвращают квитанции (‘requests’), которые ‘погашаются’ при завершении; до погашения квитанции с переменными и массивами, которые были аргументами неблокирующей функции, ничего делать нельзя.

- Локальные функции не инициируют пересылок данных между ветвями. Локальными являются большинство информационных функций (т.к. копии системных данных уже хранятся в каждой ветви). Функция передачи MPI_Send и функция синхронизации MPI_Barrier не являются локальными, поскольку производят пересылку. В то же время функция приема MPI_Recv (парная для MPI_Send) является локальной: она всего лишь пассивно ждет поступления данных, ничего не пытаясь сообщить другим ветвям.
- Коллективные функции должны быть вызваны всеми ветвями-абонентами коммуникатора (о понятии коммуникатора см. ниже), передаваемого им как аргумент. Несоблюдение этого правила приводит к ошибкам на стадии выполнения программы (обычно к ‘повисанию’ программы).

В MPI продумано объединение ветвей в коллективы (это помогает надежнее отличать сообщения друг от друга). Достигается это введением в функции MPI параметра типа ‘коммуникатор’, который можно рассматривать как дескриптор (номер) коллектива (он ограничивает область действия данной функции соответствующим коллективом). Коммуникатор коллектива, который включает в себя все ветви приложения, создается автоматически (при выполнении функции MPI_INIT) и называется MPI_COMM_WORLD; в дальнейшем имеется возможность (функция MPI_Comm_split) создавать новые коммуникаторы. Имеют место включающий только самого себя как процесс коммуникатор MPI_COMM_SELF и (заведомо неверный) коммуникатор MPI_COMM_NULL; в случае возможности (динамического) порождения новых процессов ситуация сложнее (подробнее см. [3÷5]).

Наиболее простыми функциями коммуникаций являются функции ‘точка-точка’, в таких взаимодействиях участвуют два процесса, причем один процесс является отправителем сообщения, а другой - получателем. Процесс-отправитель вызывает одну из процедур передачи данных и явно указывает номер в коммуникаторе процесса-получателя, а процесс-получатель должен вызвать одну из процедур приема с указанием того же коммуникатора (в некоторых случаях необязательно знать точный номер процесса-отправителя в заданном коммуникаторе).

Все процедуры этой группы делятся на два класса: *процедуры с блокировкой* (с синхронизацией) и *процедуры без блокировки* (асинхронные). Процедуры обмена с блокировкой приостанавливают работу процесса до выполнения определенного условия, а возврат из асинхронных процедур происходит немедленно после инициализации соответствующей коммуникационной опе-

рации. Неаккуратное использование процедур с блокировкой может привести к возникновению *тупиковой ситуации* (см. ниже), поэтому при этом требуется дополнительная осторожность; при использовании асинхронных операций тупиковых ситуаций не бывает, однако эти операции требуют более аккуратного использования массивов данных.

При использовании процедур ‘точка-точка’ программист имеет возможность выбрать способ взаимодействия процессов и способ взаимодействия коммуникационного модуля MPI с вызывающим процессом.

По способу взаимодействия процессов - в случае неодновременного вызова двумя процессами парных функций приема и передачи могут быть произведены следующим образом:

- Буферизация данных осуществляется на передающей стороне (при этом функция передачи захватывает временный буфер, копирует в него сообщение и возвращает управление вызвавшему процессу; содержимое буфера передается принимающему процессу в фоновом режиме).
- Режим ожидания на стороне принимающего процесса (при этом возможно завершение с в**ы**дачей кода ошибки на передающей стороне).
- Режим ожидания на передающей стороне (завершение с кодом ошибки на стороне приема).
- Автоматический выбор одного из трех вышеприведенных вариантов (именно так действуют простейшие блокирующие функции MPI_Recv и MPI_Send).

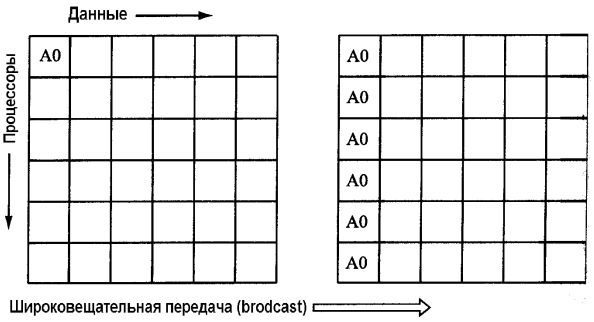
По способу взаимодействия MPI-модуля с вызывающим процессом:

- Блокирующий режим - управление вызывающему процессу возвращается только после того, как данные приняты или переданы (или скопированы во временный буфер).
- Неблокирующий режим - управление вызывающему процессу возвращается немедленно (т.е. процесс блокируется до завершения операции) и фактическая приемопередача происходит в фоновом режиме. Этот режим работает с ‘квитированием’ (функции неблокирующего приема имеют дополнительный параметр типа ‘квитанция’, причем процесс не имеет права производить какие-либо действия с буфером сообщения, пока квитанция не будет ‘погашена’).
- Персистентный режим (эффективен, например, при приемопередаче внутри цикла, когда создание/закрытие канала вынесены за его границы); в этом случае в отдельные функции выделены:
  - создание (постоянного) ‘канала’ для приема/передачи сообщений,
  - инициация процесса приема/передачи через канал,
  - закрытие канала.

Функций коллективных коммуникаций (при которых получателей и/или отправителей несколько) имеется несколько типов; в правом столбце нижерасположенной таблицы схематично показаны процессоры (вычислительные узлы) в виде строк и хран**и**мые в их ОП порции данных в виде столбцов, аббревиатурой БЛД обозначается блок данных (подробнее о применении описанных функций см. в [1,2] и http://parallel.ru):

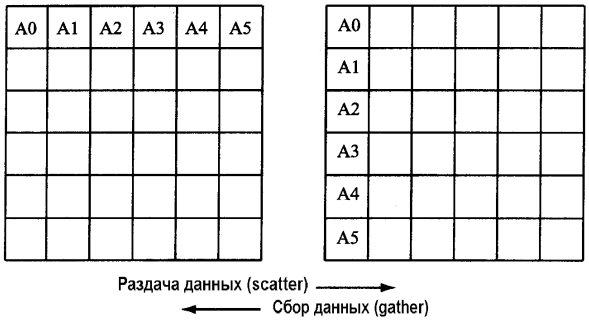
1. **Тип broadcast: *один-всем***

Широковещательная передача сообщений - один БЛД одного из процессов передается всем процессам группы MPI_Bcast(buffer, count, datatype, source, comm), где buffer – начальный адрес буфера данных, count – число элементов в буфере, datatype - тип элементов, source - номер передающего процесса, comm - коммуникатор




2. **Тип scatter: *один-каждому***

Раздача БЛД от одного процесса всем процессам группы MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, source, comm), где sendbuf, sendcount, sendtype – адрес начала, число и тип элементов буфера пос**ы**лки, recvbuf, recvcount, recvtype – то же для буфера сбора данных, source – но-



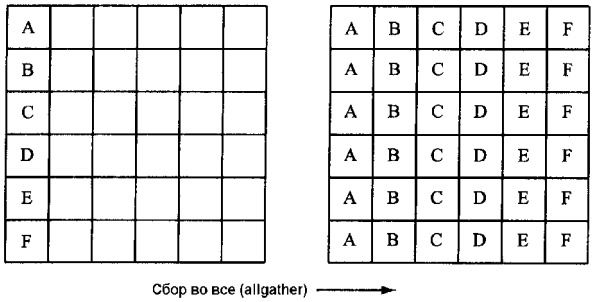
мер процесса сбора данных, comm - коммуникатор

**Тип gather:** ***каждый-одному***

Сбор БЛД от всех процессов в группе в один из процессов группы MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, dest, comm). Каждый процесс группы (включая корневой) посылает содержимого своего передающего буфера корневому процессу.

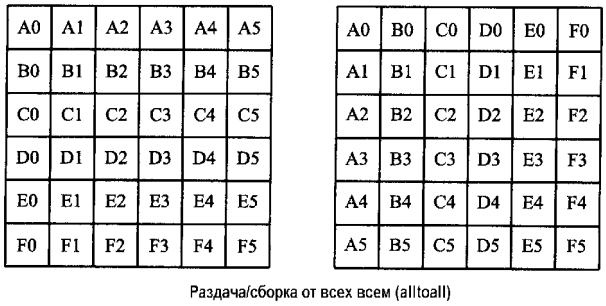
3. **Тип allgather:** ***все-каждому***

Сбор БЛД из всех процессов группы с получением результата сбора всеми процессами группы MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, source_dest, comm).



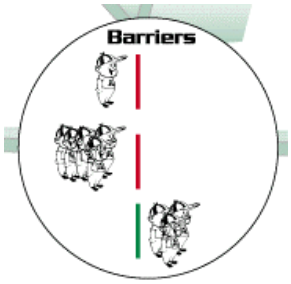

4. **Тип alltoall:** ***каждый-каждому***

Раздача/сборка БЛД от всех процессов во все процессы MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm).

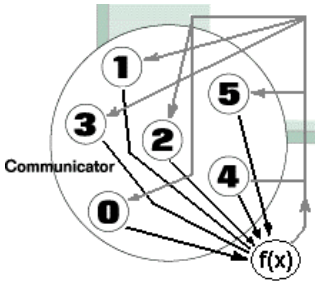



5. **Барьерная синхронизация группы процессов** MPI_Barrier

При вызове этой процедуры управление в вызывающую программу не возвращается до тех пор, пока *все процессы группы* не произведут **вы**зовов MPI_Barrier(comm), где comm - коммуникатор




6. **Глобальные (собирающие данные со всех процессов) операции редукции** (сложение, вычисление максимума, минимума и определенные пользователем), результат передается одному или всем процессам (MPI_Reduce, MPI_Allreduce, MPI_Reduce_Scatter, MPI_Scan).




Вообще говоря, для написания подавляющего большинства программ достаточно 6 функций интерфейса MPI:

MPI_Init - инициализация MPI-библиотеки

| | |
|---|---|
| MPI_Comm_size | - определение числа процессов |
| MPI_Comm_rank | - определение процессом собственного номера |
| MPI_Send | - посылка сообщения |
| MPI_Recv | - получение сообщения |
| MPI_Finalize | - завершение программы MPI |

среди которых основными являются функции MPI_Send/MPI_Recv обмена сообщениями типа ‘точка-точка’.

Например, функция широковещательной передачи MPI_Bcast посылает сообщение buffer от процесса source всем процессам группы comm (включая себя). Она *вызывается всеми процессами группы* с одинаковыми аргументами count, datatype, source, comm; в момент возврата управления содержимое buffer гарантированно скопировано в buffer всех процессов.

Коллективные функции MPI_Bcast и MPI_Reduce можно выразить через п**а**рные операции MPI_Send и MPI_Recv. Например, операция MPI_Bcast для расс**ы**лки целого числа N всем процессам (в т.ч. самому себе) эквивалентна следующему циклу (однако MPI_Bcast выполняется быстрее):

```
for (i=0; i<i_procs; i++) // всем процессам от 0 до i_procs
  MPI_Send(&N, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
```

Области передачи сообщений коллективных функций не пересекаются с таковыми для функций ‘точка-точка’ (например, с помощью MPI_Recv принять сообщение от MPI_Bcast невозможно).

При программировании на C список header-файлов обязательно должен быть дополнен строкой

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

Общая структура простейшей параллельной C-программы приблизительно такова (типов**о**й пример MPI-программы в стиле SIMD); процесс с нулевым номером обычно считают главным (MASTER), другие процессы – подчиненными (SLAVE):

```
#include "mpi.h"
// иные необходимые header’ы для C-программы

int
main(int argc, char **argv)
{
 int me, size;

 MPI_Init (&argc, &argv); // инициализировать MPI
 MPI_Comm_rank (MPI_COMM_WORLD, &me); // номер данной ветви
 MPI_Comm_size (MPI_COMM_WORLD, &size); // общее число ветвей

// …совместных код для всех ветвей (включая MASTER-процесс)
```

```
 if (me == 0) // это главная ветвь (MASTER-процесс)
 {
 //… выполнить в главной ветви
 }
 else
 if (me != 0) // это все ветви отличные от главной (MASTER'а-процесса)
 {
 //… выполнить во всех рабочих (SLAVE) ветвях
 }
// дальнейшие варианты не так часто встречаются,
// однако формально должны быть рассмотрены
 else
 if (me == 1) // это ветвь номер 1
 {
  //… код для ветви 1
 }
 else
 if (me == 2) // это ветвь номер 2
 {
 //… код для ветви 2
 }

 MPI_Finalize(); // закрыть MPI

} // конец функции main
```

Для корректного завершения параллельной программы необходимо вызывать функцию MPI_Finalize перед возвращением из программы в следующих случаях:

- перед вызовом стандартной C-функции exit,
- перед каждым после MPI_Init оператором return в функции main,
- если функция main возвращает тип void (стандартно int) и она не заканчивается оператором return, то MPI_Finalize следует вызвать перед концом main.

В качестве приведенных ниже примеров MPI-программ частично использованы тексты из http://www.parallel.ru, http://www.keldysh.ru, данные работ $[1 \div 6]$ и некоторые другие. Единственным путем профессионального освоения MPI является постоянная практика и изучение литературных источников по проблеме.

## 1. *Лабораторная работа 1.* Ознакомление с архитектурой вычислительного виртуального LINUX-кластера и основами его администрирования

*Общие сведения.* Специализированные вычислительные кластеры в большинстве случаев работают под управлением Linux как наиболее мобильной (машинно-независимой) ОС. Для реализации параллельных технологий обычно не требуются многие возможности ОС, столь чрезмерно р**а**звитые в Windows (например, графический пользовательский интерфейс).

Специализированные вычислительные кластерные системы достаточно д**о**роги; в то же время в эксплуатации находится большое количество компьютерных классов, оснащенных работающих под управлением Windows ПЭВМ. Сотрудником Института Прикладной Математики (ИПМ) имени М.В.Келдыша РАН (http://www.kiam.ru) А.О.Лацисом [2] предложена реализация вычислительной системы на основе работающих под Windows ПЭВМ, делающая доступным изучение и применение кластерных систем в большинстве учреждений.

Возможны несколько путей организации подобных систем. Один из них заключается в совместной установке ОС Windows и Linux (с выбором загрузки необходимой ОС при включении ПЭВМ – т.н. *вариантная загрузка*). Недостатком этого является невозможность использования ПЭВМ для проведения обычных работ (обычно требующих Windows) во время Linux-сеанса.

Второй способ основан на применении технологии ***виртуальных машин***, конкретно - на мониторе виртуальных машин фирмы VMware (http://www.vmware.com). Преимуществом этого способа является одновременная работа Windows и Linux на одной ЭВМ, а также возможность конфигурировать по своему усмотрению виртуальную машину. Например, на управляющей машине кластера желательно иметь три разных сетевых интерфейса (из них два - для целей управления и межузлового обмена - строго обязательны, [2]). Виртуальная машина легко может быть сконфигурирована с тремя *виртуальными сетевыми картами*, в то время как по первому способу пришлось бы действительно монтировать в компьютер две или три сетевых карты.

При использовании виртуальной машины в качестве узла кластера возникает вопрос об эффективности (прежде всего – о быстродействии процессора). Виртуальная машина выполняется под управлением MS Windows на физической машине в *качестве процесса* и, как и в любом другом процессе, непривилегированные команды (арифметические, переходов и им подобные) *выполняются физическим процессором напрямую*. При этом *эмулируются лишь привилегированные команды*, в основном связанные с вводом – выводом. В конечном итоге центральным процессором выполняется поток команд (перемежаемый, естественно, предписаниями б**а**зовой ОС), полностью идентичный таков**о**му в настоящих ‘железных’ Linux-кластерах, при этом чистые потери быстродействия процессора на накладные расходы составляют около

7% (по данным разработчиков МВС-900). Именно вследствие этого вычислительно**е**мкие приложения могут выполняться на виртуальной машине практически так же быстро, как на физической, если бы на нее непосредственно была установлена та же операционная система.

Набор виртуального оборудования, которым оснащена та или иная виртуальная машина, задается при ее конфигурировании средствами *диспетчера виртуальных машин*. Возможно оснащать виртуальную машину эмулируемыми дисками (в действительности им соответствуют просто файлы), эмулируемыми сетевыми картами и пр.

Из версий Windows необходимо использовать 2000 или XP, версия VMware Workstation for Windows (виртуальная машина) загружается с http://www.vmware.com (рекомендуемый объем ОП для виртуальной машины – не менее 128 kБайт), в качестве ОС на виртуальной машине используется Slackware Linux (рис.2).

В целом созданный по технологии МВС-900 виртуальный кластер принадлежит к вычислительным системам архитектуры MPP (*Massively Parallel Processing*), системы подобной архитектуры допускают почти неограниченную масштабируемость (известны комплексы, включающие многие тысячи вычислительных узлов). В MPP-системах каждый вычислительный узел имеет собственную (быстродоступную) оперативную память (ОП) - в отличие от SMP (*Symmetric Multi Processing*)-систем, где все процессорные узлы работают с общей памятью. Программирование SMP-систем проще (между процессорами разделяются только вычисления), в случае MPP-систем приходится заботиться и о распределении больших массивов данных между ОП вычислительных узлов. Нечто среднее между MPP и SMP представляет архитектура NUMA (*Non-uniform Memory Access*), содержащая (логически, однако необязательно физически) общую (но существенно менее быстродействующую по сравнению с локальной памятью процессоров) для всех процессоров память; в случае физически распределенной памяти встает проблема *когерентности кэша*.

Как видно из рис.2, сеть разделяется на *сеть управления* и *сеть обмена данными* (зах**о**д извн**е** пользователя на управляющий узел требует еще одной сети). Однако на каждой ПЭВМ используется всего лишь один сетевой адаптер (карта); *логическое разделение* сети реализуется программно путем использования технологии виртуализации адаптеров в виртуальных Linux-машинах. Пл**а**той за это является значительная латентность сетевого соединения - до 300 мксек для Fast Ethernet (против $5 \div 20$ мксек у SCI, Myrinet; по данным работы [4]).

При включении МВС-900 управляющая машина должна включаться первой, а выключаться последней (т.к. на ней располагаются принадлежащие разным узлам файлы NFS – *Network File System*). Вычислительные Linux-узлы (NODE's) запускаются автоматически как Windows-с**е**рвисы и при работе не администрируются вообще; управляется МВС-900 исключительно с управляющей (HOST)-машины.

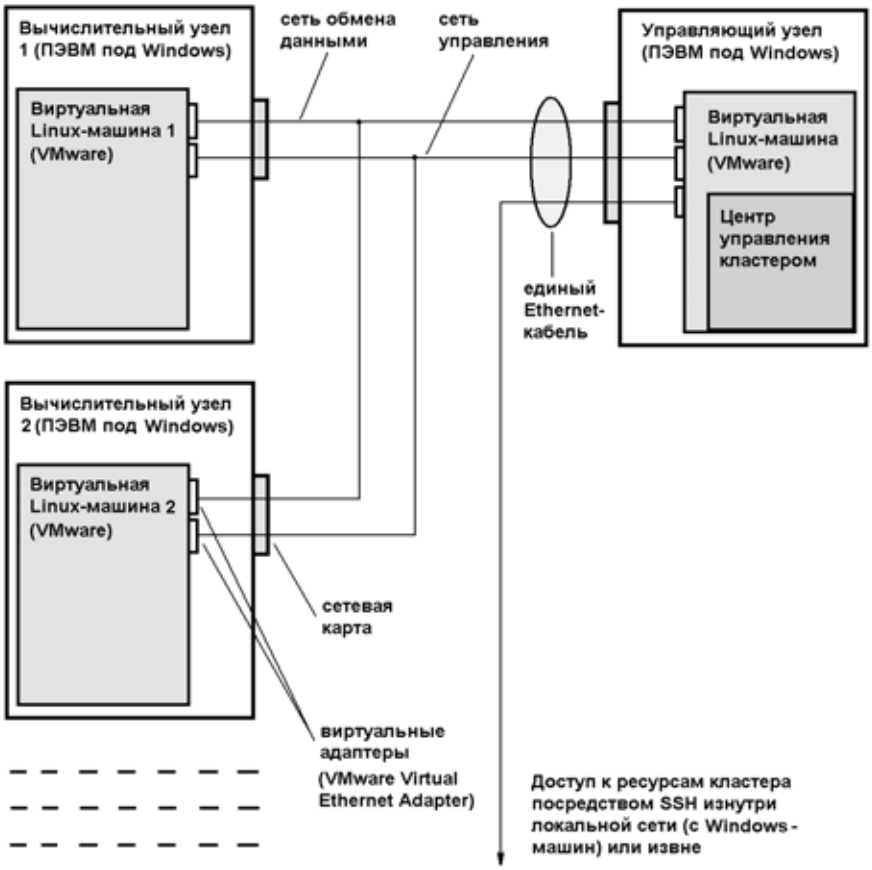



Рисунок 2.— Укрупненная схема виртуального Linux-кластера на основе локальной сети Windows-машин (технология МВС-900)

Вычислительная система МВС-900 обладает очень высокой *степенью замкнутости* в смысле администрирования и сопровождения. Она не разделяет с локальной сетью, на которой она реализована, ни сетевых адресов, ни дисциплины и прав доступа. Все сложные трудоемкие действия по обеспечению безопасности, авторизации, настройке сетевых маршрутов, правил доступа и т.п. выполняются для физических и виртуальных машин независимо.

Режим работы пользователя, программиста и администратора системы МВС-900 практически полностью соответствуют таковым (могущего считаться классическим в России) суперкластера МВС-1000 Межведомственного Суперкомпьютерного Центра (http://www.jscc.ru), для дальнейшего совершенствования навыков полезно использовать соответствующую документацию (http://pilger.mgapi.edu/metods/1441/mvs_gui.zip).

При взаимодействии процессы системы управления прохождением задач передают друг другу файл специального формата - *паспорт параллельной задачи*. В процессе прохождения заданий в системах МВС-1000/МВС-900 важную роль играет программа qserver - *сервер очередей*, которая ведет очередь параллельных задач согласно заданной *политике планирования очередей*. В момент запуска задачи сервер очередей вызывает программу mrun, которая осуществляет (рис.3):

- определение свободных вычислительных модулей,
- выделение требуемого параллельного ресурса для задачи,
- порождение (с помощью системного вызова помощью fork) специального процесса-менеджера manager, который запускает задачу и контролирует ее.

Процесс-менеджер осуществляет конфигурацию выделенных вычислительных модулей для параллельной задачи, после чего на модуле, стоящим первым в списке выделенных для задачи с помощью команды rsh, запускает на выполнение специальный командный файл, инициирующий параллельную задачу. Далее процесс-менеджер с помощью fork порождает процесс sleeper, контролирующий время выполнения задачи. Процесс mrun, получив определенный сигнал от менеджера, формирует код возврата и диагностическое сообщение, передает их серверу очередей через стандартный вывод и завершается.

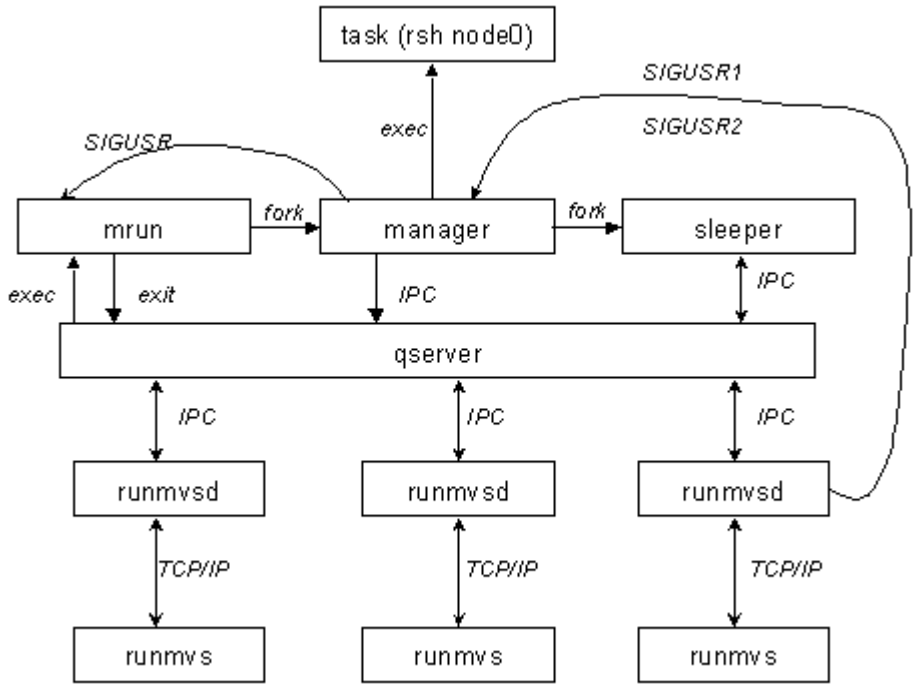



Рисунок 3.— Схема взаимодействия процессов при прохождении задания в системе МВС-1000М

Успешно запущенная задача может быть завершена четырьмя различными способами:

- ‘естественным’ путем (т.е. закончив счет),
- по истечении заказанного времени;
- принудительно пользователем;
- принудительно администратором системы.

В любом случае, при завершении задачи менеджер осуществляет следую-

щие действия:

- производит очистку всех выделенных задаче вычислительных модулей,
- освобождает вычислительные модули в системе учета занятых модулей и вносит изменения в специальные файлы, содержащие информацию о запущенных и завершенных задачах,
- соединяется со своим сервером очередей и сообщает ему (пользуясь средствами IPC – *Inter Process Communications*) о завершении задачи и освобождении ресурсов, после чего самозавершается.

При истечении заказанного времени счета sleeper завершается (в случае завершения задачи ‘естественным’ путем менеджер завершает sleeper и выполняет вышеперечисленные действия). В случае истечения заданного времени выполнения задания менеджер, получив сигнал о завершении sleeper, завершает процесс rsh и также выполняет перечисленные действия.

Процесс управления фоновой задачей (могущей прерываться системой и выполняться по квантам) несколько отличается от описанного (подробнее см. в http://pilger.mgapi.edu/metods/1441/mvs_gui.zip)

Создание выполняемого файла (компиляция и редактирование связей модуля program.c) осуществляется пользователем командой (исполняемый файл получит название program без расширения):

```
mpicc -o program program.c
```

З**а**пуск программы на выполнение осуществляется с помощью пакетного файла (*скрипт**а***) mpirun следующим образом:

```
mpirun –np N [mpi_args] program [command_line_args …]
```

где –np N – обязательный ключ, причем параллельное приложение будет образовано N задачами-копиями, загруженного из программного файла program_file (экземпляры задачи получают номера от 0 до N-1),

mpi_args – (необязательные) аргументы MPI-системы (например,
-q Q, где Q – очередь, в которую будет поставлена задача;
-p P, где P – приоритет задачи в очереди;
-maxtime T, где T – максимальное время выполнения задачи, минут),

command_line_args … - (необязательные) аргументы командной строки, передающиеся каждому экземпляру задачи.

Пример (на 7 вычислительных узлах запускается программа из файла my_program с параметрами сред**ы** MPI по умолчанию):

```
mpirun –np 7 my_program
```

Заметим, что при з**а**пуске задачи на N вычислительных узлах львиную доля вычислительной работы обычно выполняют N-1 рабочих SLAVE-узлов; один управляющий (MASTER)-узел координирует работу остальных (подготавливает и рассылает SLAVE-узлам данные для расчета, собирает данные и осуществляет их дополнительную обработку).

На каждом узле в данный момент исполняется единственная задача (процесс); в случае недоступности требуемого количества узлов (в**ы**ключены соответствующие ПЭВМ, не загружен Linux или узлы в данный момент заняты стор**о**нними задачами) задание ставится в очередь до момента освобождения нужного количества узлов.

При регистрации в системе МВС-900 нового пользователя user_name создается его личный (‘домашний’) каталог /home/user_name, в котором и происходит вся работа пользователя. В**ы**дача программой my_program данных в поток stdout (например, при использовании функции printf) перенаправляется в файл /home/user_name/current_dir/my_program.N/output (где current_dir – текущий подкаталог в домашнем каталоге; N обычно =1); дополнительно в этом же каталоге создаются файлы .hosts (список участвующих в решении вычислительных узлов), errors (информация о происшедших при выполнении ошибках), manager.log (история прохождения задания в системе МВС-900) и runmvs.bat (полный текст пакетного файла запуска задания).

Просмотр очереди заданий выполняется командой:

```
mtask –n <идентификатор_задачи> [-q <очередь>]
```

Задача с идентификатором ID из очереди удаляется командами (от имени пользователя и администратора соответственно):

```
mqdel имя_задачи
```

```
mqdelete /имя_user’a/имя_задачи
```

Также можно пользоваться командами mkill (снятие задачи со счета, в случае mkill ‘*’ снимается исполняющаяся задача пользователя), mfree (узнать число свободных число свободных процессоров), mqinfo (просмотр очереди).

Администратор кластера имеет возможность работать непосредственно с консоли HOST-машины или в режиме удаленного доступа, используя Telnet или SSH-клиентские программы (см. ниже). Пользователю в режиме удаленного доступа доступны следующие действия:

- Обмен (двусторонний) файлами между своей (клиентской) машиной и HOST-машиной кластера
- Управление файловой системой (создание/изменение/удаление файлов и каталогов и т.п.)

- Управление заданиями (з**а**пуск на выполнение и останов, получение справок о состоянии)

Команды на выполнение двух последних функций могут быть введены непосредственно в режиме командной строки или с применением файл-менеджеров; удобно пользоваться вызываемым консольной командой mc -ac программой Midnight Commander (далее MC), весьма близкой по пользовательскому интерфейсу и принципам управления известной оболочке Norton Commander.

Для удаленного доступа с рабочей консоли пользователя используются клиентские программы (незащищенный протокол Telnet для д**о**ступа изнутри локальной сети и протокол с повышенной защищенностью SSH для д**о**ступа извн**е** на портах 23 и 22 соответственно):

- Telnet (штатная Win-версия с командной строкой)
- HyperTerminal (штатная оконная Win-версия Telnet, использование MC не очень удобно)
- Telneat (консольный ан**а**лог Telnet, использование MC не вызывает затруднений)
- PSCP (PuTTY Secure CoPy client, управляется командной строкой) служит для защищенного обмена данными (каталогами и файлами) между рабочей машиной пользователя и сервером (при этом может производиться преобразование текстовых файлов из формата Windows в формат Linux и обратно)
- WinSCP (Windows Secure CoPy, оконный вариант PSCP), позволяет выбрать Norton Commander или MS Explorer-подобный режим файл-менеджера
- PuTTY (поддерживает протоколы Telnet и SSH, работа с MC удобна)
- SSH-клиенты оконного режима (напр., SSH Secury Shell 3.2.3 с входящим в комплект модулем SSH Secure File Transfer Client)

Автор данной работы рекомендует пользоваться последней программой (именно она установлена на Win-машинах компьютерных классов кафедры). Все клиентские программы требуют указания адреса (IP или дом**е**нного) HOST-машины, далее вводятся login и password для зах**о**да собственно на Linux-управляющий узел.

Дополнительно доступны баз**и**рованная на WEB-технологии консоль оператора и реализующая пульт пользователя программа PULT; обе системы разработаны в испытательной лаборатория проекта МВС ИПМ им. М.В.Келдыша РАН.

При работе с классическим Linux-кластером не используют**с**я ср**е**ды (системы) визуального программирования (типа Borland Delphi/C++Builder, Visual C++ и др.), стандартные языки программирования – C/C++ и Fortran.

Исходные тексты программ могут быть подготовлены на стор**о**нних ЭВМ (в т.ч. работающих под Windows) и перенесены для исполнения на кластер

(особенно удобно использовать для этого входящий в комплект SSH Secury Shell модуль SSH Secure File Transfer Client).

Пакетная обработка заданий предполагает, что программа не должна быть интерактивной (следует избегать ввода с клавиатуры). Данные в программу вводятся из командной строки или дискового файла (имя файла может быть фиксированным или указываться в командной строке); иногда рационально использовать механизм перенаправления ввода из файла (в этом случае команда запуска программы может иметь вид: mpirun –np 7 my_program < datafile). Вывод рассчитанных данных также происходит в дисковый файл (содержание которого затем может быть проанализировано и в Windows); при этом рационально в комментариях к программе использовать исключительно латиницу. В целях избежания непредсказуемого смешения порций данных ввод и вывод следует осуществлять исключительно главным процессом.

Пользователей регистрирует (задавая password и login) администратор системы. На администратора вычислительного кластера ложится большая нагрузка (б**о**льшая, чем для администратора учебного класса Windows-машин). Работа администратора заключается в корректном включении и выключении виртуального кластера, регистрации пользователей, настройке очередей, выявлении и устранении тупиковых ситуаций и др.

‘Железные’ (не виртуальные) вычислительные кластеры широко описаны в литературе (например, кластеры НИВЦ МГУ, [4]).

*Вопросы для самопроверки:*

1. Чем отличается виртуальный вычислительный Linux-кластер от реального ‘железного’ кластера?
2. Чем отличаются архитектуры MPP и SMP? Каковы их преимущества и недостатки? Что такое NUMA-архитектура и каковы ее достоинства и недостатки?
3. Что такое ‘виртуальная машина’ и в чем особенность ее функционирования? Почему потери производительности при функционировании виртуального Linux’а в рассматриваемом случае минимальны?
4. Какие сети необходимы при создании виртуального кластера? Какие из них являются виртуальными? Какие из них желательно превратить (в первую очередь) в реальные для повышения производительности кластера?
5. В чем отличия администрирования виртуального вычислительного кластера от таков**о**го вычислительной сети?
6. Каким образом реализуется ограничение времени выполнения задания по заданному максимального времени счета?

## 2. *Лабораторная работа 2.* Жизненный цикл процессов и простейший обмен данными между ними, тупиковые ситуации

*Общие сведения.* При использовании компактно расположенных кластеров исполняемый код приложения компилируется главной машиной, рассылается по ВУ и стартуется на каждом средствами ОС. Момент старта каждой ветви программы может отличаться от такового на других ВУ, также нельзя априори точно определить момент выполнения любого оператора внутри конкретной ветви, см. рис.2 (именно поэтому применяются различного типа приемы синхронизации при обмене сообщениями);

Для практического осознания необходимости синхронизации процессов при параллельном программировании служит первая часть данной работы. Во второй части практически рассматривается формальный обмен данными между процессами (использование простейших функций передачи MPI_Send и приема данных MPI_Recv).

Для этих (б**а**зовых) функций полезно привести прот**о**типы (многие из приведенных формальных параметров используются и в других MPI-функциях):

```
int
MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag,
         MPI_Comm comm);
```

- buf - адрес начала буфера посылки сообщения
- count - число передаваемых в сообщении элементов (*не байт* – см. ниже)
- datatype - тип передаваемых элементов
- dest - номер процесса-получателя
- msgtag - идентификатор сообщения (0÷32767, выбирается пользователем)
- comm - идентификатор группы (MPI_COMM_WORLD для создаваемого по умолчанию коммуникатора)

Функция MPI_Send осуществляет *блокирующую* пос**ы**лку сообщения с идентификатором msgtag процессу с номером dest; причем сообщение состоит из count элементов типа datatype (все элементы сообщения расположены подряд в буфере buf, значение count может быть и нулем). Тип передаваемых элементов datatype должен указываться с помощью предопределенных констант типа (см. ниже), разрешается передавать сообщение самому себе. При пересылке сообщений можно использовать специальное значение MPI_PROC_NULL для несуществующего процесса; операции с таким процессом немедленно успешно завершаются (код завершения MPI_SUCCESS).

*Блокировка* гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Это достигается копированием в промежуточный буфер или непосредственной передачей процессу dest (определяется MPI). Важно отметить, что возврат из MPI_Send не означает ни того, что сообщение уже передано процессу dest, ни того, что сообщение покинуло

процессорный элемент, на котором выполняется процесс, выполнивший MPI_Send.

Имеются следующие модификации процедуры передачи данных с блокировкой MPI_Send:

- MPI_Bsend - передача сообщения *с буферизацией.* Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в специальный буфер, и произойдет немедленный возврат из процедуры. Выполнение этой процедуры никоим образом не зависит от соответствующего вызова процедуры приема сообщения. Тем не менее процедура может вернуть код ошибки, если места под буфер недостаточно (о выделении массива для буферизации должен озаботиться пользователь).
- MPI_Ssend - передача сообщения *с синхронизацией.* Выход из этой процедуры произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера посылки, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Использование передачи сообщений с синхронизацией может замедлить выполнение программы, но позволяет избежать наличия в системе большого количества не принятых буферизованных сообщений.
- MPI_Rsend - передача сообщения *по готовности.* Этой процедурой можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов процедуры является ошибочным и результат ее выполнения не определен. Гарантировать инициализацию приема сообщения перед вызовом процедуры MPI_Rsend можно с помощью операций, осуществляющих явную или неявную синхронизацию процессов (например, MPI_Barrier или MPI_Ssend). Во многих реализациях процедура MPI_Rsend сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи данных.

В MPI не используются привычные для C типы данных (int, char и др.), вместо них удобно применять определенные для данной платформы константы MPI_INT, MPI_CHAR и т.д. (см. табл.1).

Таблица 1.— Предопределенные в MPI константы типов данных.

| *Константы MPI* | *Соответствующий тип в C* |
|---|---|
| MPI_INT | signed int |
| MPI_UNSIGNED | unsigned int |
| MPI_SHORT | signed int |
| MPI_LONG | signed long int |

| | |
|---|---|
| MPI_UNSIGNED_SHORT | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_CHAR | signed char |

Пользователь может зарегистрировать в MPI свои собственные типы данных (например, структуры), после чего MPI сможет обрабатывать их наравне с базовыми.

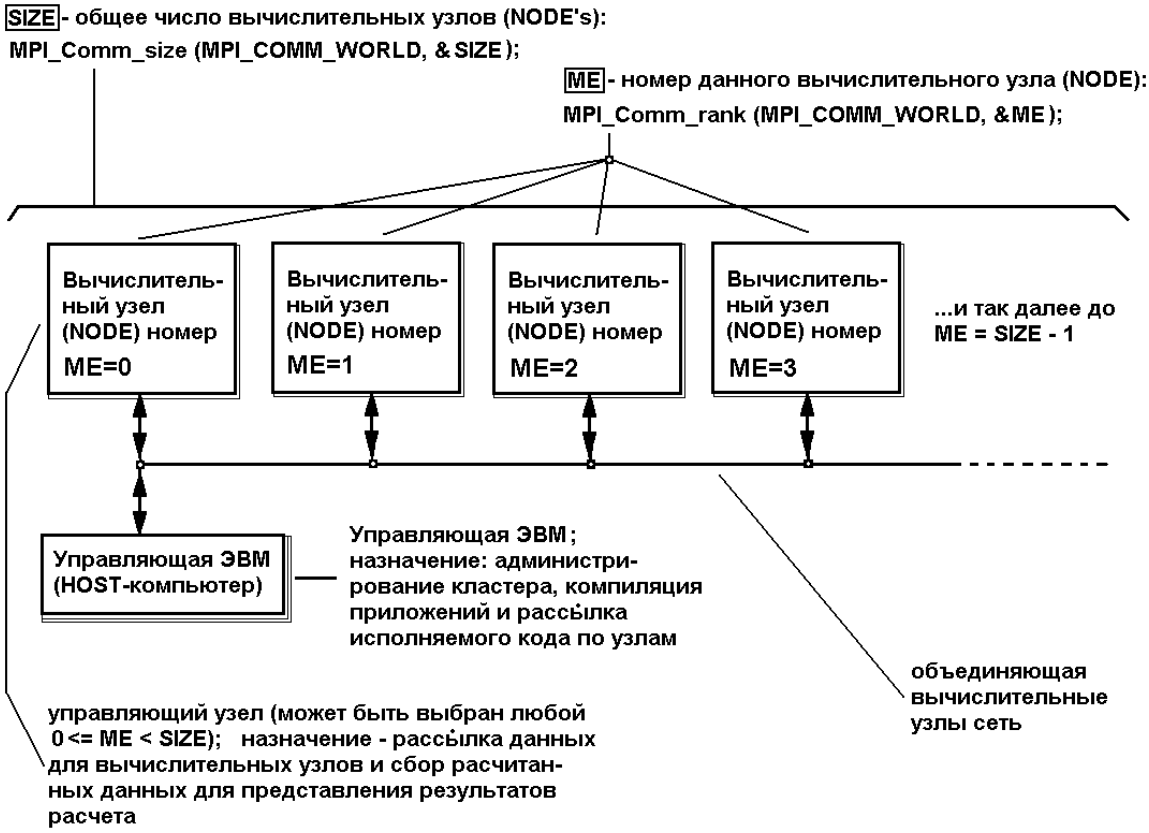



Рисунок 4.— Нумерация входящих в кластер вычислительных узлов (по умолчанию) и их стандартные назначения

Практически в каждой MPI-функции одним из параметров является коммуникатор (идентификатор группы процессов); в момент инициализации библиотеки MPI создается коммуникатор MPI_COMM_WORLD и в его пределах процессы нумеруются линейно от 0 до size (рис.4). Однако с помощью коммуникатора для процессов можно определить и другие системы нумерации (*пользовательские топологии*). Дополнительных систем в MPI имеются две: декартова N-мерная решетка (с цикличностью и без) и ориентированный граф. Существуют функции для создания и тестирования нумераций (MPI_Cart_xxx, MPI_Graph_xxx, MPI_Topo_test) и для преобразования номеров из одной системы в другую. Этот механизм *чисто логический и не связан с аппаратной топологией*; при его применении автоматизируется пересчет ад-

рес**о**в ветвей (например, при вычислении матриц иногда выгодно использовать картезианскую систему координат, где координаты вычислительной в**е**тви совпадают с координатами вычисляемой ею подматрицы).

```
int
MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag,
        MPI_Comm comm, MPI_Status *status);
```

- buf - адрес начала буфера приема сообщения (возвращаемое значение)
- count - максимальное число элементов в принимаемом сообщении
- datatype - тип элементов принимаемого сообщения
- source - номер процесса-отправителя
- msgtag - идентификатор принимаемого сообщения
- comm - идентификатор группы
- status - параметры принятого сообщения (возвращаемое значение)

Функция MPI_Recv осуществляет прием сообщения с идентификатором msgtag от процесса source с блокировкой (блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере buf). Число элементов в принимаемом сообщении не должно превосходить значения count (если число принятых элементов меньше count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения). Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову MPI_Recv, первым будет принято то сообщение, которое было отправлено раньше.

Т.о. с помощью пары функций MPI_Send/MPI_Recv осуществляется надежная (но не слишком эффективная) передача данных между процессами. Однако в некоторых случаях (например, когда принимающая сторона ожидает приема сообщений, но априори не знает длины и типа их) удобно использовать блокирующую функцию MPI_Probe, позволяющую определить характеристики сообщения *до того*, как оно будет помещено в приемный пользовательский буфер (гарантируется, что следующая вызванная функция MPI_Recv прочитает именно протестированное MPI_Probe сообщение):

```
int
MPI_Probe( int source, int msgtag, MPI_Comm comm, MPI_Status *status);
```

- source - номер процесса-отправителя (или MPI_ANY_SOURCE)
- msgtag - идентификатор ожидаемого сообщения (или MPI_ANY_TAG)
- comm - идентификатор группы
- status - параметры обнаруженного сообщения (возвращаемое значение)

В структуре status (тип MPI_Status) содержится информация о сообщении – его идентификатор (поле MPI_TAG), идентификатор процесса-отправителя

(поле MPI_SOURCE), фактическую длину сообщения можно узнать посредством вызовов

```
MPI_Status status;
int count;
MPI_Recv ( ... ,  MPI_INT, ... , &status );
MPI_Get_count (&status, MPI_INT, &count); /* тип элементов тот же, что у MPI_Recv
                теперь в  count  содержится количество принятых элементов типа
                MPI_INT */
```

Удобно сортировать сообщения с помощью механизма ‘джокеров’; в этом случае вместо явного указания номера задачи-отправителя используется ‘джокер’ MPI_ANY_SOURCE (‘принимай от кого угодно’) и MPI_ANY_TAG вместо идентификатора получаемого сообщения (‘принимай что угодно’). Достоинство ‘джокеров’ в том, что приходящие сообщения извлекаются по мере поступления, а не по мере вызова MPI_Recv с нужными идентификаторами задач и/или сообщений (что экономит память и увеличивает скорость работы). Пользоваться ‘джокерами’ рекомендуется с осторожностью, т.к. возможна ошибка в приеме сообщения ‘не того’ или ‘не от того’.

При обмене данными в некоторых случаях возможны *вызванные взаимной блокировкой* т.н. тупиковые ситуации (используются также термины ‘deadlock’, ‘клинч’); в этом случае функции посылки и приема данных мешают друг другу и обмен не может состояться. Ниже рассмотрена deadlock-ситуация при использовании для пересылок разделяемой памяти.

| Вариант 1. | *Ветвь 1 :* | *Ветвь 2 :* |
|---|---|---|
| | Recv ( из ветви 2 ) | Recv ( из ветви 1 ) |
| | Send ( в ветвь 2 ) | Send ( в ветвь 1 ) |

Вариант 1 приведет к deadlock’у при любом используемом инструментарии, т.к. функция приема не вернет управления до тех пор, пока не получит данные; из-за этого функция передачи не может приступить к отправке данных, поэтому функция приема не вернет управление... и т.д. до бесконечности.

| Вариант 2. | *Ветвь 1 :* | *Ветвь 2 :* |
|---|---|---|
| | Send ( в ветвь 2 ) | Send ( в ветвь 1 ) |
| | Recv ( из ветви 2 ) | Recv ( из ветви 1 ) |

Казалось бы, что (если функция передачи возвращает управление только после того, как данные попали в пользовательский буфер на стороне приема) и здесь deadlock неизбежен. Однако при использовании MPI зависания во втором варианте не произойдет: функция MPI_Send, если на приемной сторо-

не нет готовности (не вызвана MPI_Recv), не станет дожидаться ее вызова, а скопирует данные во временный буфер и *немедленно вернет управление основной программе*. Вызванный далее MPI_Recv данные получит *не напрямую из пользовательского буфера, а из промежуточного системного* буфера (т.о. используемый в MPI способ буферизации повышает надежность - делает программу более устойчивой к возможным ошибкам программиста). Т.о. наряду с полезными качествами (см. выше) свойство блокировки может служить причиной возникновения (трудно локализуемых и избегаемых для непрофессионалов) тупик**о**вых ситуаций при обмене сообщениями между процессами.

Очень часто функции MPI_Send/MPI_Recv используются совместно и именно в таком порядке, поэтому в MPI специально введены две функции, осуществляющие одновременно пос**ы**лку одних данных и прием других. Первая из них - MPI_Sendrecv (у нее первые 5 формальных параметров такие же, как у MPI_Send, остальные 7 параметров аналогичны MPI_Recv). Следует учесть, что:

- как при приеме, так и при передаче используется один и тот же коммуникатор,
- порядок приема и передачи данных MPI_Sendrecv выбирает автоматически; при этом гарантировано отсутствие deadlock'a,
- MPI_Sendrecv совмест**и**ма с MPI_Send и MPI_Recv

Функция MPI_Sendrecv_replace помимо общего коммуникатора использует еще и общий для приема-передачи буфер. Применять MPI_Sendrecv_replace удобно с учетом:

- принимаемые данные должны быть заведомо *не длиннее* отправляемых
- принимаемые и отправляемые данные должны иметь одинаковый тип
- принимаемые данные записываются на место отправляемых
- MPI_Sendrecv_replace так же гарантированно не вызывает deadlock'a

В MPI предусмотрен набор процедур для осуществления *асинхронной передачи данных.* В отличие от блокирующих процедур, возврат из процедур данной группы происходит сразу после вызова без какой-либо остановки работы процессов; на фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной операции.

Обычно эта возможность исключительно полезна для создания эффективных программ - во многих случаях совершенно не обязательно дожидаться окончания посылки сообщения для выполнения последующих вычислений.

Асинхронным аналогом процедуры MPI_Send является MPI_Isend (для MPI_Recv, естественно, MPI_Irecv). Аналогично трем модификациям процедуры MPI_Send предусмотрены три дополнительных варианта процедуры

MPI_Isend:

- MPI_Ibsend - неблокирующая передача сообщения с буферизацией
- MPI_Issend - неблокирующая передача сообщения с синхронизацией
- MPI_Irsend - неблокирующая передача сообщения по готовности

*Цель работы* – приобретение практических знаний и навыков в компиляции и запуске простейших MPI-программ, практическое уяснение необходимости синхронизации ветвей,

*Необходимое оборудование* – вычислительный кластер под управлением UNIX-совместимой ОС, предустановленная поддержка MPI, рабочая консоль программиста для управления прохождением пользовательских задач.

*Порядок проведения работы* – студент подготавливает исходные тексты MPI-программ, компилирует их в исполнимое приложение, запускает на счет, анализирует выходные данные программы.

Часть 1 работы. Первой задачей является компиляция простейшей MPI-программы EXAMPLE_01.C, запуск ее на исполнение на заданном преподавателем числе ВУ и анализ результатов.

```
// source code of EXAMP_01.C program
#include "mpi.h"
#include <stdio.h>
#include <sys/timeb.h> // for ftime function

int main(int argc, char **argv)
{
 int CURR_PROC, ALL_PROC, NAME_LEN;
 char PROC_NAME[MPI_MAX_PROCESSOR_NAME];
 struct timeb t; // time contain structure

 MPI_Init (&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &CURR_PROC); /* current process */
 MPI_Comm_size (MPI_COMM_WORLD,  &ALL_PROC); /* all process */

// get processor name (really computer name)
 MPI_Get_processor_name(PROC_NAME, &NAME_LEN);

 ftime(&t); // get current time point
// t.time is time in sec since January 1,1970
// t.millitm is part of time in msec
// ATTENSION ! ftime may by used only after synchronization all processors time

// output to STDIN
 printf("I am process %d from %d and my name is %s (time: %.3f sec)\r\n",
        CURR_PROC, ALL_PROC, PROC_NAME, (t.time+1e-3*t.millitm));

 MPI_Finalize();
}  // end of EXAMPLE_01 program
```

Наиболее интересен анализ последовательности строк в**ы**дачи каждой ветви программы:

```
I am process 0 from 5 and my name is Comp_07 (time: 1110617937.329 sec)
I am process 2 from 5 and my name is Comp_06 (time: 1110617937.329 sec)
I am process 4 from 5 and my name is Comp_11 (time: 1110617939.003 sec)
I am process 1 from 5 and my name is Comp_02 (time: 1110617939.003 sec)
I am process 3 from 5 and my name is Comp_01 (time: 1110617938.997 sec)
```

Для определения момента времени с точностью $10^{-3}$ сек использована стандартная С-функция ftime (выдает *локальное* время для каждого процессора). Функция MPI_Wtime возвращает для каждого вызвавшего процесса *локальное* астрономическое время в секундах (double), прошедшее с некоторого момента (обычно 01.I.1970) в прошлом; гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время существования процесса. Функция MPI_Wtick возвращает double-значение, определяющее разрешение таймера (промежуток времени между 'т**и**ками') процессора. Без (принудительной) синхронизации час**о**в всех процессоров функции ftime и MPI_Wtime следует использовать лишь для определения относительных промежутков (но не для абсолютных моментов) времени. В случае MPI_WTIME_IS_GLOBAL=1 время процессоров синхронизировано, при =0 – нет.

Необходимо повторить запуск исполняемого модуля программы несколько раз с указанием различного числа процессоров; желательно предварительно попытаться ответить на вопрос о возможной последовательности в**ы**дачи данных (удивительно, но даже *при всей вышеприведенной* информации о правилах выполнения параллельных приложений правильный ответ нечт**а**т!).

Часть 2 работы. Задачей является формальное освоение использования простейших функций передачи и приема данных (MPI_Send и MPI_Recv). Эти функции осуществляют простейшую связь типа 'точка-точка' между процессами (одна ветвь вызывает функцию MPI_Send для передачи, другая – MPI_Recv для приема данных) с блокировкой. Обе эти функции (как и другие) возвращают код ошибки как целое (здесь не используется, успешному окончанию соответствует MPI_SUCCESS, подробнее см. файл mpi.h).

В следующем примере (файл EXAMP_02.C) нулевой процесс посылает (идентифицированную тегом 999) текстовую строку процессу 1 (обмен 'точка-точка'), там же строка и печатается:

```
// source code of EXAMP_02.C program
#include "mpi.h"
#include <stdio.h>
int main (int argc, char **argv)
{
 char message[20];
```

```
 int i=0, all_rank, my_rank;
 MPI_Status status;
 MPI_Init (&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &my_rank); // get number current process
 MPI_Comm_size (MPI_COMM_WORLD,  &all_rank); // all process

 if (my_rank==0)    /* code for process zero */
 {
  strcpy (message, "Hello, there!\0");
//for(i=0; i<all_rank; i++)
  MPI_Send(message, strlen(message), MPI_CHAR, i, 999, MPI_COMM_WORLD);
 }
 else   /* code for other processes */
 {
  MPI_Recv(message, 20, MPI_CHAR, 0, 999, MPI_COMM_WORLD, &status);
//MPI_Recv(message, 20, MPI_CHAR,  0, MPI_ANY_TAG, MPI_COMM_WORLD,
//                &status);
  printf ("I am %d process; received: %s\n", my_rank, message);
 }
  MPI_Finalize();
} // end of EXAMP_02 program
```

Так как использована *блокирующая* функция MPI_Recv, при запуске более чем на 2 процессорах программа *естественным образом не завершится* (исполнение закончится только по истечению отведенного времени).

Раскомментировав строку //for(i=0; i<all_rank; i++) и замен**и**в (сх**о**дным образом MPI_Recv(message, 20, MPI_CHAR, 0, 999, MPI_COMM_WORLD, &status); на MPI_Recv(message, 20, MPI_CHAR, 0, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);, получим на выходе подобные нижеприведенным строки):

```
I am 2 process; received: Hello, there!
I am 1 process; received: Hello, there!
I am 4 process; received: Hello, there!
I am 3 process; received: Hello, there!
```

В последнем случае использованы предопределенные MPI-константы (‘джокеры’) MPI_ANY_SOURCE и MPI_ANY_TAG, означающие ‘принимай от любого источника’ и ‘принимай сообщения с любым тегом’. Теперь программа завершится естественным способом (т.к. будут *удовлетворены все блокирующие вызовы* MPI_Recv).

Дополнительные задания (самостоятельная работа студентов):

- усложнить структуру передаваемой в качестве сообщения текстовой строки (внести номер процесса, имя компьютера, момент времени передачи)
- для посылки сообщения всем процессам использовать функцию расс**ы**лки MPI_Bcast

- использовать вместо пар MPI_Send/MPI_Recv универсальные функции приема/передачи MPI_Sendrecv и MPI_Sendrecv_replace, оценить возможное уменьшение времени выполнения программы при этом (реализовать цикл повт**о**ра $1000 \div 10000$ раз и взять среднее, для оценки времени выполнения dt воспользоваться конструкцией t1=MPI_Wtime(); ..... t2=MPI_Wtime(); dt=t2-t1;)

Часть 3 работы. Задачей является определение времени выполнения основных MPI-функций. Для этого используется профессионально разработанная (лаборатория параллельных информационных технологий НИВЦ МГУ, parallel@parallel.ru) программа MPI_TEST.C (версия 2.0), исходный текст которой вследствие значительного размера (более 500 строк исходного кода) не приводится (программа работоспособна при число процессоров не менее 2).

Параметры командной строки имеют вид <буква><значение> и могут идти в командной строке в любом порядке (или вообще отсутствовать); смысл основных параметров следующий:

- T<число> - тестовая процедура повторяется указанное число раз (для каждого значения длины сообщения), результаты усредняются; по умолчанию $T=10^3$
- t<число> - число повторений для относительно ‘быстрых’ операций (таких как измерение времени); умолчание $t=10^4$
- R<число> - число повторений всего теста; умолчание R=1

Примерный вид в**ы**дачи программы приведен ниже (тест выполнялся на 5 процессорах высокопроизводительного кластера SCI НИВЦ МГУ, усредненное время выполнения MPI-вызовов указано в мксек, точность не лучше величины Timing):

```
Process 1 of 5 on sci2-4
Process 2 of 5 on sci3-1
Process 3 of 5 on sci3-2
Process 4 of 5 on sci3-3
MPItest/C 2.0: running 5 processes...
Process 0 of 5 on sci2-2
Testing basic MPI routines.
T = 1000, t = 10000

--- Size 1, Iteration 0 ---
** Timing for MPI operations in microseconds ***
0.330508        Timing; clock tick: 1
8.96987         Barrier
43.2285         Global sum of 1 numbers (Allreduce)
15.5733         Global sum of 1 numbers (Reduce)
25.436          Broadcast of 1 numbers
22.852          Gathering of 1 numbers
67.4485         All-gather of 1 numbers
```

```
77.4022         All-to-All send of 1 numbers
9.82214         Non-blocking send of 1 numbers with wait
8.62117         Blocking vector send of 1 numbers
14.0505         SendRecv, 1 numbers
17.3717         Send and receive of 1 numbers
45.0062         5 sends of 1 number in one
50.2882         5 async. sends of 1 number
13.9659         signal send and receive (double latency)

--- Size 10, Iteration 0 ---
** Timing for MPI operations in microseconds ***
0.330631        Timing; clock tick: 1
9.53711         Barrier
53.7914         Global sum of 10 numbers (Allreduce)
19.7596         Global sum of 10 numbers (Reduce)
30.2918         Broadcast of 10 numbers
12.8682         Gathering of 10 numbers
65.8808         All-gather of 10 numbers
64.604          All-to-All send of 10 numbers
11.5766         Non-blocking send of 10 numbers with wait
10.4252         Blocking vector send of 10 numbers
16.4718         SendRecv, 10 numbers
22.0297         Send and receive of 10 numbers
45.0127         5 sends of 1 number in one
50.2651         5 async. sends of 1 number
13.9314         signal send and receive (double latency)
…
…
MPItest/C complete in 7.79548 sec.
```

Если тест завершается чрезмерно быстро, то возникает вероятность большой погрешности в измерении времени и может потребоваться увеличение числа итераций (параметр T<число итераций>). Исходный текст MPI_TEST.C является превосходным образцом использования MPI-функций и дост**о**ин тщательного исследования для дальнейшего квалифицированного применения.

Индивидуальные задания включает определение времени выполнения основных MPI-функций на различном оборудовании (напр., сравн**и**ть быстродействие высокопроизводительного кластера SCI НИВЦ МГУ и вычислительного кластера кафедры ИТ-4 МГАПИ).

*Вопросы для самопроверки:*

1. Из каких стадий состоит жизненный цикл процессов при параллельном программировании?
2. Почему нельзя точно определить момент старта процессов на вычислительных узлах?
3. Что такое коммуникатор? Что такое *пользовательские топологии* и какие (кроме стандартной с линейной нумерацией) определены MPI?

4. В чем механизм блокирующей посылки сообщения? Какие MPI-функции осуществляют блокирующие посылки?
5. Что такое передача сообщений с буферизацией? Какие MPI-функции осуществляют буфферизованные посылки?
6. В чем заключается механизм ‘джокеров’ и каковы его преимущества? Какие проблемы могут возникнуть при использовании ‘джокеров’?
7. Что такое взаимная блокировка (тупиковая ситуация, ‘deadlock’, ‘клинч’)? В каких условиях это явление возникает?

## 3. *Лабораторная работа 3.* Определение параметров коммуникационной сети вычислительного кластера

*Общие сведения.* В вычислительных кластерах наиболее ‘узким местом’ обычно является сеть, осуществляющая связь между вычислительными узлами (фактически эта сеть является ан**а**логом общей шины, связывающей различные узлы любой ЭВМ).

Даже весьма быстродействующая сеть обладает свойством *латентности* (‘времени разгона’ до номинальной производительности); латентность особо сильно снижает реальную производительность сети при частом обмене (небольшими по объему) сообщениями (при задержке поступления очередного порции информации вычислительному узлу он вынужденно простаивает, рис.2).

В случае одностороннего обмена сообщениями между двумя узлами (обмен типа ‘точка-точка’) затрачиваемое на передачу время T (сек) оценивается как:

$$T=X/S+L,$$

где X – длина сообщения (Мбайт),

S – пропускная способность сетевого канала ‘точка-точка’ (мгновенная скорость передачи данных), Мбайт/сек,

L – время разгона операции обмена (не зависит от длины сообщения), сек.

Иногда бывает удобно оперировать латентностью, приведенной к скорости (*цена обмена* P, Мбайт):

$$P=L\times S,$$

Цена обмена – размер блока данных, которые канал ‘точка-точка’ мог бы передать при нулевой латентности.

При определении реальной (с учетом латентности) пропускн**о**й способности сети на операциях обмена типа ‘точка-точка’ используют пару простейших блокирующих (блокирующие функции возвращают управление вызывающему процессу только после того, как данные приняты или переданы или скопированы во временный буфер) MPI-предписаний MPI_Send/MPI_Recv, причем каждая операция повторяется много раз (с целью статистического усреднения).

Следует иметь в виду, что тестирование коммуникационной сети кластера на операциях ‘точка-точка’ является всего лишь важной, однако частью общей процедуры тестирования (более полный набор тестов можно получить с http://parallel.ru/testmpi).

Для виртуального кластера максимально достиж**и**мая реальная произво-

дительности коммуникационной сети достигается при размере сообщений более $128 \div 256$ кбайт, при этом величина латентности может достигать $300 \div 400$ мксек (что является пла**а**той за виртуальное совмещение сетей управления и коммуникаций единой физической Fast Ethernet-сетью).

*Цель работы* – замер реальной (с учетом латентности) производительности коммуникационной сети вычислительного кластера на блокирующих операциях ‘точка-точка’.

*Необходимое оборудование* – вычислительный кластер под управлением UNIX-совместимой ОС, предустановленная поддержка MPI, рабочая консоль программиста для управления прохождением пользовательских задач.

*Порядок проведения работы* – студент подготавливает исходные тексты MPI-программ, компилирует их в исполняемое приложение, запускает на счет, анализирует выходные данные программы и представляет их в графическом виде.

Исходный код простой C-программы PROG_MPI.C тестирования производительности сети приведен ниже:

```
// source code of PROG_MPI.C program
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

static int message[10000000]; // max length of message

int main (int argc, char *argv[])
{
 double  time_start,time_finish;
 int  i, n, nprocs, myid, len;
 MPI_Request rq;
 MPI_Status status;

 MPI_Init(&argc,&argv); /* initialization MPI */
 MPI_Comm_size(MPI_COMM_WORLD,&nprocs); /* all processes */
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);  /* current process */

 printf("PROGMPI: nprocs=%d myid=%d\n", nprocs, myid);
 fflush( stdout );
 len = atoi( argv[1] ); // get length message from command line
 printf( "length: %d\n", len );

 if (myid == 0) /* I a m MASTER-process ! */
 {
  n = 1000; /* cycles number */
  MPI_Send(&n, 4, MPI_CHAR, nprocs-1, 99,MPI_COMM_WORLD);
  time_start=MPI_Wtime();
  for (i=1;i<=n;i++)
```

```
  {
   MPI_Send(message, len, MPI_CHAR, nprocs-1, 99, MPI_COMM_WORLD);
   MPI_Recv(message, len, MPI_CHAR, nprocs-1, 99, MPI_COMM_WORLD,
                                                          &status);
  }
  time_finish=MPI_Wtime();
  printf( "Time %f rate %f speed %f\n", (float)(time_finish-time_start),
                                        (float)(2*n/(time_finish-time_start)),
                                        (float)(2*n*len/(time_finish-time_start)) );
 } /* end if (myid==0) */

 else
 if (myid == (nprocs-1)) /* I am is last of all processes ! */
 {
  MPI_Recv(&n,4, MPI_CHAR, 0, 99, MPI_COMM_WORLD,&status);
  for (i=1;i<=n;i++)
   {
    MPI_Recv(message, len, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    MPI_Send(message, len, MPI_CHAR, 0, 99, MPI_COMM_WORLD);
   }
 } /* end if (myid==(nproc-1)) */

fflush( stdout );
MPI_Finalize(); /* deinitialize MPI */
} // end of PROC_MPI program
```

Для запуска на исполнение применяется команда (запуск на 5 вычислительных узлах при максимальном времени выполнения 15 мин, последнее число – размер сообщений в байтах):

```
mpirun -np 5 –maxtime 15 proc_mpi 1048576
```

Образец в**ы**дачи (кластер кафедры ИТ-4 МГАПИ, задействованы 2 вычислительных узла):

```
LAM 7.0.6/MPI 2 C++ - Indiana University

PROGMPI: nprocs=2 myid=0
PROGMPI: nprocs=2 myid=1
length: 1048576
length: 1048576
Time 182.223991 rate 10.975503 speed 11508649.257698
```

Экспериментальные данные св**о**дятся в таблицу, причем принято строить зависимость пропускн**о**й способности сети (столбец Speed таблицы, Мбайт/сек) от длины сообщения (столбец Len, kбайт) в логарифмической (по основанию 2) шкале абсцисс.

Таблица 2.— Данные экспериментов по определению зависимости реальной пропускн**о**й способности коммуникационной сети кластера от длины передаваемых сообщений

| *№ №* | *Len (байт/кбайт))* | *Time (сек)* | *Rate ($сек^{-1}$)* | *Speed ($байт*сек^{-1}$)* |
|---|---|---|---|---|
| 1 | 64 (0,0625) | | | |
| 2 | 128 (0,125) | | | |
| 3 | 256 (0,25) | | | |
| … | … | … | … | … |
| … | 2’097’152 (2’048) | | | |
| … | 4’194’304 (4’096) | | | |
| … | 8’388’608 (8’192) | | | |

Столбец Rate показывает частоту обмена сообщениями в секунду. При малых дл**и**нах (единицы/десятки байт) сообщений латентность (в мксек) естественно вычисляется как $L=10^6/Rate$.

Требуется построить (удобно воспользоваться средствами Excel) зависимость реальной скорости коммуникационной сети от размера сообщений, оценить величину латентности, сделать выводы о размере сообщений, при которых реализуется теоретическая пропускная способность сети.

*Вопросы для самопроверки:*

1. Чем отличается производительность сети при обмене сообщениями различной длины от заявленной изготовителем?
2. Каков физический смысл латентности и цены обмена? В каких единицах они измеряются?
3. Как зависит реальная производительность сети от размера передаваемых сообщений? При сообщениях какой длины производительность сети достигает (теоретического) уровня?
4. Какова причина повышения латентности при виртуализации сетей управления и обмена данными?

## 4. *Лабораторная работа 4.* Простые MPI-программы (численное интегрирование)

*Общие сведения.* Среди задач численного анализа встречается немало задач, распараллеливание которых очевидно. Например, численное интегрирование сводится фактически к (многочисленному) вычислению подинтегральной функции (что естественно доверить отдельным процессам), при этом главная ветвь управляет процессом вычислений (определяет стратегию распределения точек интегрирования по процессам и 'собирает' частичные суммы). Подобным же 'интуитивным распараллеливанием' обладают задачи поиска и сортировки в линейном списке, численного нахождения корней функций, поиск экстремумов функции многих переменных, вычисление рядов и др. Трудности распараллеливания возникают, например, в случае рекурсивных алгоритмов и при последовательном просчете по формулам.

*Цель работы* – приобретение практических знаний и навыков в практическом программировании несложных MPI-программ, анализе точности вычисляемых значений.

*Необходимое оборудование* – вычислительный кластер под управлением UNIX-совместимой ОС, предустановленная поддержка MPI, рабочая консоль программиста для управления прохождением пользовательских задач.

*Порядок проведения работы* – студент подготавливает исходные тексты MPI-программ, компилирует их в исполнимое приложение, запускает на счет, анализирует (и по заданию преподавателя представляет в графическом виде) выходные данные.

Часть 1 работы. Широкоизвестно, что $\frac{\pi}{4} = \text{arctg}(1) - \text{arctg}(0) = \int_0^1 \frac{dx}{1+x^2}$. Заменяя вычисление интеграла конечным суммированием, имеем $\int_0^1 \frac{dx}{1+x^2} \approx \frac{1}{n} \sum_{j=1}^{j=n} \frac{1}{1+x_j^2}$, где $x_j = (j-0{,}5)/n$, n – число участков суммирования при численном интегрировании.

Площадь каждого участка (вертикальной 'полоски' - stripe) вычисляется функцией COMPUTE_INTERVAL как произведение ширины 'полоски' (width) на значение функции в центре 'полоски', далее площади суммируются главным процессом (используется равномерная сетка).

С целью уяснения принципов распределения вычислений рекомендуется проанализировать текст COMPUTE_INTERVAL (здесь j – номер участка интегрирования, myrank – номер данного вычислительного узла, intervals – общее число интервалов численного интегрирования, ntasks – общее число вычислительных узлов, значение локальных сумм накапливается в localsum):

```
localsum=0.0;
for (j=myrank; j<intervals; j+= ntasks)
{
 x = (j + 0.5) * width;
 localsum += 4 / (1 + x*x);
}
```

Заметим, что нагрузка каждого процесса вычислением конкретного значения подинтегральной функции было бы крайне нерациональным решением, т.к. время обмена сообщениями (посылка главным процессом значения параметра функции и прием вычисленного значения) сравнимо со временем вычисления функции и существенно повысило бы время выполнения программы (чрезмерно много коротких пересылок); подобный подход является примером излишне тонкозернистого (*fine-grained*) параллелизма. В целом следует стремиться к использованию возможно *меньшего количества максимально длинномерных* обменов.

В качестве функции времени совместно с MPI_Wtime применяется стандартная С-функция *локального для вычислительного узла* времени ftime (функция F_TIME). В программе использованы исключительно (блокирующие) функции обмена ‘точка-точка’ MPI_Send/MPI_Recv, активно используется барьерная функция MPI_Barrier (при вызове MPI_Barrier(comm) управление в вызывающую программу не возвращается до тех пор, пока все процессы группы не вызовут MPI_Barrier(comm), где comm - коммуникатор). Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован.

```
// source code PI_01.C program
#include "mpi.h"
#include <stdio.h>
#include <math.h>

#include <sys/timeb.h> // for ftime
double f_time(void); /* define real time by ftime function  */
#include “f_time.c”

double compute_interval (int myrank, int ntasks, long intervals);

int main(int argc, char ** argv)
{
 long intervals=100; // number of integration’s parts
 int i, myrank,ranksize;
 double pi, di, send[2], recv[2],
        t1,t2, t3,t4, t5,t6,t7, // time’s moments
        pi_prec=4.0*(atanl(1.0)-atan(0.0)); // pi precision
```

```
MPI_Status status;

t1=f_time();
MPI_Init (&argc, &argv);  /* initialize MPI-system */
t2=f_time();
MPI_Comm_rank (MPI_COMM_WORLD, &myrank); /* my place in MPI system */
MPI_Comm_size (MPI_COMM_WORLD, &ranksize);  /* size of MPI system */

if (myrank == 0)  /* I am the MASTER */
{
intervals = atoi( argv[1] ); /* get interval's count from command line */
printf ("Calculation of PI by numerical integration with %ld intervals\n", intervals);
}

MPI_Barrier(MPI_COMM_WORLD); /* make sure all MPI tasks are running */

if (myrank == 0) /* I am the MASTER */
{  /* distribute parameter */
printf ("Master: Sending # of intervals to MPI-Processes \n");
t3 = MPI_Wtime();
for (i=1; i<ranksize; i++)
 MPI_Send (&intervals, 1, MPI_LONG, i, 98, MPI_COMM_WORLD);
} /* end work MASTER */
else   /* I am a SLAVE */
{ /* receive parameters */
MPI_Recv (&intervals, 1, MPI_LONG, 0, 98, MPI_COMM_WORLD, &status);
} // end work SLAVE

/* compute my portion of interval */
t4 = MPI_Wtime();
pi = compute_interval (myrank, ranksize, intervals); /*======================*/
t5 = MPI_Wtime();
MPI_Barrier (MPI_COMM_WORLD); /* make sure all MPI tasks are running */
t6 = MPI_Wtime();

if (myrank == 0) /* I am the MASTER */
{  /* collect results add up & printing results */
 for (i=1; i<ranksize; i++)
 {
  MPI_Recv (&di, 1, MPI_DOUBLE, i, 99, MPI_COMM_WORLD, &status);
  pi += di;
 } /* end of collect results */
 t7 = MPI_Wtime();
 printf ("Master: Has collected sum from MPI-Processes \n");
 printf ("\nPi estimation: %.12lf (rel.error= %.5f %%)\n",
          pi, 1.0e2*(pi_prec-pi)/pi_prec);
 printf ("%ld tasks used,  execution time: %.3lf sec\n",ranksize, t7 -t3);
 printf("\nStatistics:\n");
 printf("Master: startup: %.0lf msec\n",t2-t1);
 printf("Master: time to send # of intervals:%.3lf sec\n",t4-t3);
 printf("Master: waiting time for sincro after calculation:%.2lf sec\n",t6-t5);
 printf("Master: time to collect: %.3lf sec\n",t7-t6);
```

```
 printf("Master: calculation time:%.3lf sec\n",t5-t4);

 MPI_Barrier (MPI_COMM_WORLD); /* make sure all MPI tasks are running */

 for (i=1; i<ranksize; i++)
 {  /* collect there calculation time */
 MPI_Recv(recv, 2, MPI_DOUBLE, i, 100, MPI_COMM_WORLD, &status);
 printf("process %d: calculation time: %.3lf sec,\twaiting time for sincro.: %.3lf sec\n",
         i,recv[0],recv[1]);
 } // end of collection
} // end work MASTER

 else /* I am a SLAVE */
{ /* send my result back to master */
 MPI_Send (&pi, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
 MPI_Barrier (MPI_COMM_WORLD); /* make sure all MPI tasks are running */
 send[0]=t5-t4;
 send[1]=t6-t5;
 MPI_Send (send, 2, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD);
} // end work SLAVE

 MPI_Finalize ();
} // end MAIN function

/* ============================================================= */
double compute_interval (int myrank, int ntasks, long intervals)
{ /* calculate integral's localsum */
 double x, width, localsum=0.0;
 long j;
 width = 1.0 / intervals; /* width of single stripe */
 for (j=myrank; j<intervals; j+= ntasks)
 {
  x = (j + 0.5) * width;
  localsum += 4 / (1 + x*x);
 }
 return (localsum * width);  /* area of stripe */
} // end of COMPUTE_INTERVAL function
// end of PI_01.C program
```

Функция f_time содержится в файле F_TIME.C (подключается посредством #include “f_time.c”):

```
double f_time() /* return time since 01 jan 1970 as double 'sec, msec' */
{
 struct timeb tp;
 ftime(&tp);
 return ((double)(tp.time)+1.0e-3*(double)(tp.millitm));
} // end f_time function
```

Возможно точное (double при данной разрядной сетке ЭВМ) значение $\pi$ определяется как 4.0*(atanl(1.0)-atanl(0.0)) и сравнивается с численно вычисленным (относительная ошибка rel.error выдается в процентах).

Образец в**ы**дачи программы приведен ниже (запуск на процессорах кластера кафедры ИТ-4 МГАПИ):

```
LAM 7.0.6/MPI 2 C++ - Indiana University

  Calculation of PI by numerical Integration with 10000 intervals
  Master: Sending # of intervals to MPI-Processes
  Master: Has collected sum from MPI-Processes

  Pi estimation: 3.141592654423 (rel.error= -0.00001 %)
  5 tasks used - Execution time: 0.017 sec

  Statistics:
  Master: startup: 4243 msec
  Master: time to send # of intervals: 0.000 sec
  Master: waiting time for sincro after calculation: 0.00 sec
  Master: time to collect: 0.000 sec
  Master: calculation time: 0.017 sec
  process 1: calculation time: 0.016 sec,  waiting time for sincro.: 0.001 sec
  process 2: calculation time: 0.017 sec,  waiting time for sincro.: 0.000 sec
  process 3: calculation time: 0.016 sec,  waiting time for sincro.: 0.001 sec
  process 4: calculation time: 0.015 sec,  waiting time for sincro.: 0.002 sec
```

Индивидуальные задания включает определение числа интервалов интегрирования, необходимое для представления числа $\pi$ с заданной относительной точностью (линейка $10^{-2}$, $10^{-3}$, $10^{-4}$, $10^{-5}$, $10^{-6}$), что реализуется методом повторных расчетов при увеличивающемся intervals).

Дополнительные задания (самостоятельная работа студентов):

- Обеспечить автоматический выбор оптимального числа интервалов вычисления интеграла (методом удвоения начального значения intervals до момента непревышения точности вычислений точности представления double-вещественного числа).
- Оценить возможное ускорение вычислений при увеличении числа процессоров (‘прогнать’ программу на числе процессоров N=2,3,4,5,6,7,8… с одинаковым числом intervals), при этом оценить точность вычисления числа $\pi$.
- Изменить программу вычисления определенного интеграла путем замены функций обмена ‘точка-точка’, коллективными функциями MPI_Bcast (передача ‘один-всем’ из ветви 0, функция передает число интервалов всем параллельным ветвям) и MPI_Reduce (получение данных от всех ветвей и суммирования их в ветви 0).

Часть 2 работы. Для вычисления значения $\pi$ можно использовать метод ‘стрельбы’ (рис.5) - вариант метод Монте-Карло. В применении к данному случаю метод заключается в генерации *равномерно распределенных* на двумерной области $[0 \le x \le 1,\ 0 \le y \le 1]$ точек и определении $\pi = 4 \times \frac{S_{0AC}}{S_{0ABC}} \approx 4 \times \frac{score}{darts}$, где S – площади фигур на рис.3, score – число попавших внутрь четверти окружности (фигура 0AC) точек (условие $x^2+y^2 \le 1.0$, на рис.5 удовлетворяющие этому условию точки обозначены квадратами), darts – общее число точек (‘выстрелов’, throws). Вычисленное таким образом значение $\pi$ является приближенным, в общем случае точность вычисления иск**о**мого значения повышается с увеличением числа ‘выстрелов’ и качества датчика случайных чисел; подобные методы используются в случае трудностей точной числовой оценки (например, для вычисления определенных интегралов большой кратности).

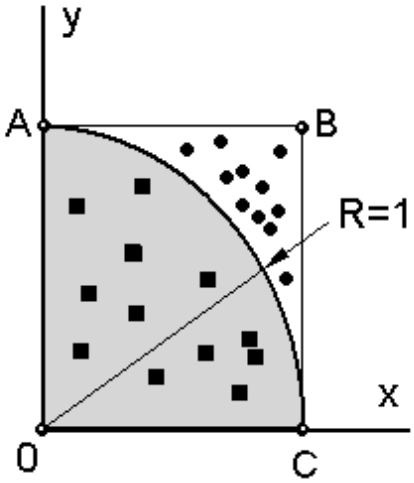


Рисунок 5.— Определение значения $\pi$ методом ‘стрельбы’

Именно функция dboard (‘мишень’) генерирует DARTS пар равномерно распределенных на отрезке $[0 \div 1]$ случайных чисел, проверяет принадлежность нахождения точки (определяемой координатами последней двойки случайных чисел) внутренней области четверти окружности и вычисляет текущее значение $\pi$; расчеты проводятся при начальном значении DARTS с последующим увеличением ROUNDS раз (з**а**даны посредством #define).

Последовательный вариант программ вычисления $\pi$ методом ‘стрельбы’ приведен ниже (файл PI_SER.C, функция DBOARD расположена в файле dboard.c и подключается посредством #include ‘dboard.c’):

```
// source code PI_SER.C program
#include <stdio.h>
#include <math.h>

#include <sys/timeb.h> // for ftime
double f_time(void); /* define real time by ftime function  */

void srandom (unsigned seed);
long random(void);
double dboard (int darts);
#include “dboard.c” // including dboard.c file
#include “f_time.c”  // including f_time.c file

#define DARTS    10000  /* number of throws at dartboard */
#define ROUNDS 100     /* number of times "darts" is iterated */
```

```
int main(int argc, char *argv[])
{
double pi,         /* average of pi after 'darts' is thrown */
          avepi,   /* average pi value for all iterations */
          t1, t2, // time's moments
          pi_prec=4.0*(atanl(1.0)-atan(0.0)); // pi precision
int i, n;

t1=f_time(); // fix start time calculated

srandom (5); // init of random generator
avepi = 0;
for (i=0; i<ROUNDS; i++) // rounds times call dboard function
{
 pi = dboard(DARTS);
 avepi = ((avepi * i) + pi)/(i + 1);
 t2=f_time(); // fix end time calculated
 printf("%7d throws aver.value of PI= %.12lf (rel.error= %.5lf %%, time= %.3lf sec)\n",
        (DARTS * (i+1)), avepi, 1.0e2*(pi_prec-avepi)/pi_prec, t2-t1);
} // end for (i=0; i<ROUNDS; i++)
} // end of MAIN function
// end of PI_SER.C program
```

Текст функции DBOARD (подключается посредством #include "dboard.c"):

```
double dboard(int darts)
{
double x_coord,    /* x coordinate, between -1 and 1  */
          y_coord,    /* y coordinate, between -1 and 1  */
          pi,          /* pi  */
          r;           /* random number between 0 and 1  */
int score,   /* number of darts that hit circle */
         n;
unsigned long cconst;

cconst = 2 << (31 - 1); /* used to convert integer random number
                           between 0 and 2^31 to double random number between 0 and 1 */
score = 0; // start summ

for (n=1; n<=darts; n++)  // cicles at random 2D-points
{
 r = (double)random() / cconst;
 x_coord = (2.0 * r) - 1.0; // x-coord of 2D-point
 r = (double)random() / cconst;
 y_coord = (2.0 * r) - 1.0; // y-coord of 2D-point
 if (((x_coord*x_coord) + (y_coord*y_coord)) <= 1.0) // 2D-point in circle?
   score++;
} // end for (n=1; n<=darts; n++)
 pi = 4.0 * (double)score / (double)darts;
 return(pi);
} // end of dboard function
```

При первом варианте распараллеливания (программа PI_02.C) используются только (блокирующие) функции обмена ‘точка-точка’ MPI_Send/MPI_Recv: вычислительные узлы посылают MASTER-узлу частичные (определенные на основе выполненных данным узлом ‘выстрелов’) значения $\pi$, главный процесс суммирует их и выдает на печать усредненные значения:

```
// source code PI_02.C program
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define DARTS    10000  /* number of throws at dartboard */
#define ROUNDS 100     /* number of times ‘darts’ is iterated */
#define MASTER   0       /* task ID of master task */

void srandom (unsigned seed);
double dboard (int darts);
#include “dboard.c” // including dboard.c file

int main(int argc, char *argv[])
{
double homepi,   /* value of pi calculated by current task */
        pi,           /* average of pi after ‘darts’ is thrown */
        avepi,       /* average pi value for all iterations */
        pirecv,      /* pi received from worker */
        pisum,      /* sum of workers PI values */
        t1, t2, // time’s moments
        pi_prec=4.0*(atanl(1.0)-atan(0.0)); // pi precision
int     taskid,       /* task ID - also used as seed number */
        numtasks, /* number of tasks */
        source,      /* source of incoming message */
        rc,           /* MPI’s return code */
        i, n;
MPI_Status status;

rc = MPI_Init(&argc,&argv);
rc = MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
rc = MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

t1=MPI_Wtime(); // store start time calculated

srandom (taskid); // initialization random number generator by taskid value

avepi = 0;
for (i=0; i<ROUNDS; i++)
{
 homepi = dboard(DARTS); // all tasks calculate pi by dartboard algorithm

if (taskid != MASTER) // It’s NOT MASTER!
{
```

```
 rc = MPI_Send(&homepi, 1, MPI_DOUBLE, MASTER, i, MPI_COMM_WORLD);
 if (rc != MPI_SUCCESS) // if error MPI_Send function
  printf("%d: Send failure on round %d\n", taskid, i);
}
else // I am MASTER
{
 pisum = 0;
 for (n=1; n<numtasks; n++)
 {
  rc = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, i,
                        MPI_COMM_WORLD, &status);
  if (rc != MPI_SUCCESS) // if error MPI_Recv function
    printf("%d: Receive failure on round %d\n", taskid, i);
  pisum = pisum + pirecv;
 }
/* Master calculates the average value of pi for this iteration */
 pi = (pisum + homepi)/numtasks;
/* Master calculates the average value of pi over all iterations */
 avepi = ((avepi * i) + pi) / (i + 1);
 t2=MPI_Wtime(); // fix end time calculated
 printf("%9d throws aver.value of PI= %.12lf (rel.error= %.5lf %%, time= %.3lf sec)\n",
        (DARTS * (i+1)), avepi, 1.0e2*(pi_prec-avepi)/pi_prec, t2-t1);
  }
} // end for (i=0; i<ROUNDS; i++)
 MPI_Finalize();
 return 0;
} // end of MAIN function
// end of PI_02.C program
```

**Вы**дача программы (вычислительный кластер кафедры ИТ-4, число ‘выстрелов’ от 1000 до 10’000) приведена ниже (напомним, что каждое вычисленное значение следует рассматривать как одну из реализаций случайной величины, нормально распределенной вокруг истинного значения $\pi$):

```
LAM 7.0.6/MPI 2 C++ - Indiana University

10000 throws aver.value of PI= 3.119600000000 (rel.error= 0.70005 %, time= 0.003 sec)
20000 throws aver.value of PI= 3.137800000000 (rel.error= 0.12072 %, time= 0.006 sec)
30000 throws aver.value of PI= 3.140000000000 (rel.error= 0.05070 %, time= 0.037 sec)
40000 throws aver.value of PI= 3.144100000000 (rel.error= -0.07981 %, time= 0.040 sec)
50000 throws aver.value of PI= 3.143440000000 (rel.error= -0.05880 %, time= 0.043 sec)
60000 throws aver.value of PI= 3.139333333333 (rel.error= 0.07192 %, time= 0.046 sec)
70000 throws aver.value of PI= 3.141200000000 (rel.error= 0.01250 %, time= 0.048 sec)
80000 throws aver.value of PI= 3.145250000000 (rel.error= -0.11642 %, time= 0.051 sec)
90000 throws aver.value of PI= 3.144933333333 (rel.error= -0.10634 %, time= 0.053 sec)
…
100000 throws aver.value of PI= 3.143760000000 (rel.error= -0.06899 %, time= 0.056 sec)
…
200000 throws aver.value of PI= 3.143840000000 (rel.error= -0.07154 %, time= 0.082 sec)
…
400000 throws aver.value of PI= 3.142310000000 (rel.error= -0.02283 %, time= 0.133 sec)
```

…
800000 throws aver.value of PI= 3.143230000000 (rel.error= -0.05212 %, time= 0.236 sec)
…
1000000 throws aver.value of PI= 3.142872000000 (rel.error= -0.04072 %, time= 0.293 sec)

Рассчитанные значения $\pi$ являются весьма приближенными даже большом числе ‘выстрелов’, причем не во всех случаях увеличение числа ‘выстрелов’ вызывает возрастание точности представления иск**о**мой величины.

Ниже приведен вариант программы (файл PI_03.C) с использованием коллективных функций (вместо функций семейства ‘точка-точка’); при этом в MASTER-процессе функция MPI_Reduce суммирует величины homepi от всех задач (сумма помещается в pisum; homepi при является буфером пос**ы**лки, pisum – приемным буфером):

```
// source code PI_03.C program
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define DARTS    10000  /* number of throws at dartboard */
#define ROUNDS 100     /* number of times ‘darts’ is iterated */
#define MASTER   0       /* task ID of MASTER task */

void srandom (unsigned seed);
double dboard (int darts);
#include “dboard.c” // including dboard.c file

int main(int argc, char *argv[])
{
double homepi, pisum, pi, avepi,
          t1, t2, // time’s moments
          pi_prec=4.0*(atanl(1.0)-atan(0.0)); // pi precision
int       taskid, numtasks, rc, i:
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

t1=MPI_Wtime(); // fix start time calculated

avepi = 0;
for (i=0; i<ROUNDS; i++)
{
 homepi = dboard(DARTS);
 rc = MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE, MPI_SUM, MASTER,
                         MPI_COMM_WORLD);
 if (rc != MPI_SUCCESS)
  printf("%d: failure on MPI_Reduce\n", taskid);
```

```
if (taskid == MASTER)
{
  pi = pisum/numtasks;
  avepi = ((avepi * i) + pi)/(i + 1);
  t2=MPI_Wtime(); // store end time calculated
  printf("%9d throws aver.value of PI= %.12lf (rel.error= %.5lf %%, time= %.3lf sec)\n",
         (DARTS * (i+1)), avepi, 1.0e2*(pi_prec-avepi)/pi_prec, t2-t1);
}
} // end for (i=0; i<ROUNDS; i++)
 MPI_Finalize();
 return 0;
} // end of MAIN function
// end of PI_03 program
```

При выполнении второй части работы следует скомпилировать варианты PI_SER (на кластере кафедры ИТ-4 МГАПИ команда компиляции и разрешения ссылок icc –o pi_ser pi_ser.c, з**а**пуска на исполнение с выдачей данных в файл pi_ser.out – pi_ser > pi_ser.out), PI_02 и PI_03 программы вычисления значения $\pi$, ‘прогнать’ их в режиме выполнения на одном и нескольких процессорах, проанализировать в**ы**дачу.

Индивидуальные задания включает определение числа интервалов интегрирования, необходимое для представления числа $\pi$ с заданной относительной точностью (линейка $10^{-2}$, $10^{-3}$, $10^{-4}$, $10^{-5}$, $10^{-6}$), что реализуется путем повторных расчетов при увеличении числа ‘выстрелов’.

Дополнительные задания (самостоятельная работа студентов):

- Оценить снижение времени вычисления $\pi$ параллельной версией программы по отношению к последовательной (файл PI_SER.C) при различных числах ‘выстрелов’.
- Оценить возможное ускорение вычислений при увеличении числа процессоров (‘прогнать’ программу на числе процессоров N=3,4,5,6,7,8… с одинаковым числом ‘выстрелов).

*Вопросы для самопроверки:*

1. Привести примеры априори хорошо распараллеливающихся алгоритмов. В чем заключается условие близкой к идеальной распараллеливаемости?
2. Будет ли результат численного интегрирования (программа PI_01.C) монотонно стремиться к точному значению соответствующего определенного интеграла при неограниченном увеличении числа интервалов интегрирования?
3. Каким образом и от каких параметров датчика случайных чисел при методе ‘стрельбы’ зависит точность вычисления определенного интеграла? Какой оценки числа $\pi$ следует ожидать при *статистически неравномерном распределении* координат точек ‘выстрелов’: а) в случае расположения

максимума гистограммы распределения вблизи 0? б) вблизи 1? в) ровно в центре отрезка $[0 \div 1]$?

## 5. *Лабораторная работа 5.* Умножение матриц – последовательная и параллельные версии

*Общие сведения.* Операции с матрицами большой (тысячи/сотни тысяч) размерности являются основой многих численных методов (например, МКЭ – *метода конечных элементов*). Только одна (далеко не самая крупная) квадратная матрица размером 3000×3000 double-чисел требует для размещения 72 Мбайт ОП (предполагаем, что матрица является плотнозаполненной – т.е. число нулевых элементов мал**о**; иначе потребуются специальные методы хранения ненулевых элементов ). Ясно, что в этом случае программист обязан распределить по процессорам кластера не только вычислительные процедуры, но и данные (те же матрицы); использование для хранения матриц дисковой памяти катастрофически замедляет процедуру.

Одна из простейших матричных операций – умножение. Известно, что в случае [C]=[A]×[B] значение элементов результирующей матрицы вычисляется как (стандартный способ):

$$c_{ij} = \sum_{k=1}^{k=NCA} a_{ik} b_{kj},$$

где i=1÷NRA – строки матриц [A] и [C],
j=1÷NCB – столбцы матриц [B] и [C],
k=1÷NCA – столбцы матрицы [A] и строки матрицы [B].

Хотя известны более эффективные алгоритмы (напр., *умножение матриц по Вин**о**граду* и *рекурсивный алгоритм Штрассена умножения квадратных матриц*), в дальнейшем используется стандартный (требующий NRA×NCB×NCA операций умножения и NRA×(NCB-1)×NCA сложений).

Тривиальная программа MM_SER.C умножения матриц стандартным способом (используется последовательный вариант вычислений одним процессором) приведена ниже:

```
// source code of MM_SER.C program
#include <stdio.h>

#include <sys/timeb.h> // for ftime
double f_time(void); /* define real time by ftime function  */
#include "f_time.c"

#define NRA 3000  /* number of rows in matrix A */
#define NCA 3000  /* number of columns in matrix A */
#define NCB 10    /* number of columns in matrix B */

int main(int argc, char *argv[])
{
int   i, j, k; /* indexes */
```

```
double a[NRA][NCA],        /* matrix A to be multiplied */
          b[NCA][NCB],        /* matrix B to be multiplied */
          c[NRA][NCB],        /* result matrix C */
          t1,t2; // time's momemts

/* Initialize A, B, and C matrices */
 for (i=0; i<NRA; i++)
  for (j=0; j<NCA; j++)
    a[i][j] = i+j;
 for (i=0; i<NCA; i++)
  for (j=0; j<NCB; j++)
    b[i][j] = i*j;
 for(i=0; i<NRA; i++)
  for(j=0; j<NCB; j++)
   c[i][j] = 0.0;

t1=f_time(); // get start time's moment

/* Perform matrix multiply */
 for(i=0; i<NRA; i++)
  for(j=0; j<NCB; j++)
    for(k=0; k<NCA; k++)
      c[i][j] += a[i][k] * b[k][j];

t2=f_time(); // get ended time's moment

printf ("Multiplay time= %.3lf sec\n\n", t2-t1);
printf("Here is the result matrix:\n");
 for (i=0; i<NRA; i++)
  {
   printf("\n");
   for (j=0; j<NCB; j++)
    printf("%6.2f  ", c[i][j]);
   }
 printf ("\n");
} // end of MM_SER.C program
```

Теоретически (без учета затрат времени на обмен данными) задача умножения матриц распараллеливается идеально (выходные данные вычисляются по единому алгоритму на основе исходных и в процессе вычисления не изменяют последних), однако при практическом распараллеливании (разработке программы) необходимо учитывать как задержки времени при обменах, так и распределение по вычислительным узлам вычислений и блоков данных. Т.о. стратегия распределения блоков матриц по ВУ должна основываться на анализе структуры процесса вычисления (т.н. *тонкой информационной структуре алгоритма*, [1]).

Находящийся на пересечении i-той строки и j-того столбца элемент матрицы [C] представляет собой сумму последовательных произведений всех (k=1÷NCA) элементов i-той строки матрицы [A] на все (k=1÷NCA) элементы j-того столбца матрицы [B]; см. рис.4.

Лежащий на поверхности способ распараллеливания заключается в пос**ы**лке (в цикле по k-i индексам [A] и соответственно k-j индексам [B]) каждому процессу значений $a_{ik}$ и $b_{kj}$, перемножению их процессом, отс**ы**лке произведению главному процессу и суммированию в нем для получения $c_{ij}$. Однако этот подход наименее рационален по производительности, ибо каждое умножение (а их число пропорционально $N^3$ , где N – характерный размер матриц) сопровождается минимум 3 обм**е**нами, причем длительность каждого сравн**и**ма или превышает длительность операции умножения. Кроме того, при таком способе велика *избыточность пересылок* (и $a_{ik}$ и $b_{kj}$ участвуют в формировании не одного $c_{ij}$, а многих).

Более рационально при вычислении каждого $c_{ij}$ пересылать каждому процессу i-тую строку [A] и j-тый столбец [B]. При этом в результате сложения частичных произведений процессом получается готовое $c_{ij}$, которое и отсылается главному процессу; однако и при этом число обменов (пропорциональное $N^2$) все еще излишне высок**о** (но оперативная память процессоров также малонагружена). Известно, что в языке C/C++ элементы матрицы располагаются в ОП по строкам (в Fortran'e – по столбцам), поэтому в C пересылка строк записывается тривиально, а обмен столбцами потребует некоторых ухищрений.

Более рационально пересылать каждому процессу *серию строк матрицы* [A] и *серию столбцов* [B]; этим реализуется крупнозернистый (*coarse-grained*) параллелизм. Каждый процесс при этом вычисляет серию строк матрицы [C], аналогичную по размерам таков**о**й матрицы [A]. При программировании на C для перес**ы**лки процессорам столбцов [B] необходимо использовать временный (рабочий) массив, при программировании на Fortran'e - то же для серии строк [A]. Недостатком способа является необходимость пересылки серии столбцов [B] между процессорами (алгоритм должен предусматривать определение, в каком процессоре находится нужный столбец матрицы [B] и пересылку его процессору, который нуждается в нем в данный момент; такой проблемы не возникает, если матрицу [B] целиком хранить на каждом процессоре).

Т.к. обычно число столбцов матрицы [B] много меньше числа строк [A] (часто NCB=1 и матрица [B] фактически является вектором), приведенной ниже программе MM_MPI.C матрица [B] пересылается всем процессам цели**к**ом (рис.6). При $NCB \cong NCA$ или затруднениях в размещении матрицы [B] цели**к**ом в ОП узла целесообразно будет разделить [B] на столбцы (см. выше).

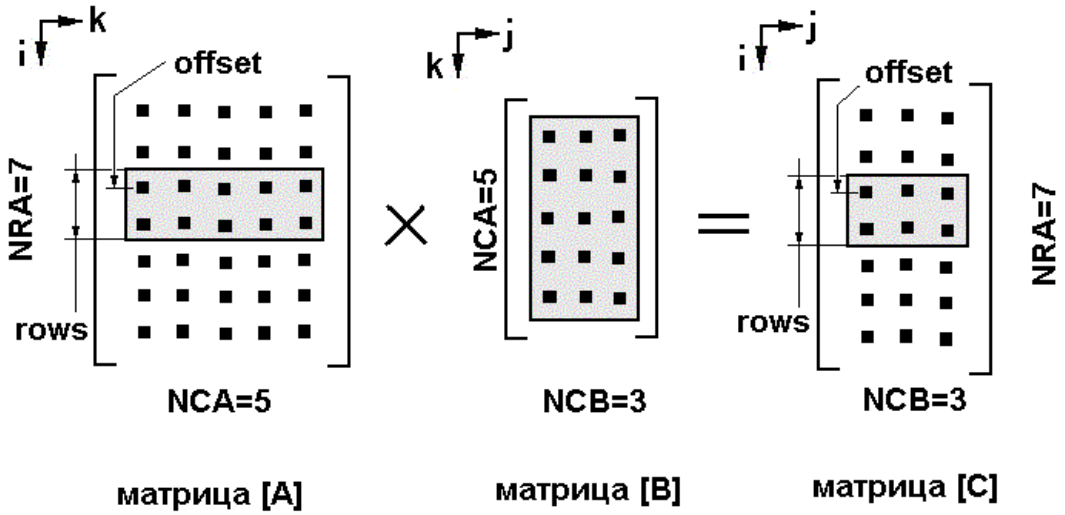



Рисунок 6.— Схема распределения блоков умножаемых матриц по процессам (пересыл**а**емые части выделены серым фоном)

```
// source code of MM_MPI_0.C program
#include "mpi.h"
#include <stdio.h>

#define NRA 3000  /* number of rows in matrix A */
#define NCA 3000  /* number of columns in matrix A */
#define NCB 10      /* number of columns in matrix B */

#define MASTER 0  /* taskid of MASTER task */
#define FROM_MASTER  1  /* setting a message type */
#define FROM_WORKER 2  /* setting a message type */
#define M_C_W  MPI_COMM_WORLD

int main(int argc, char *argv[])
{
int      numtasks,        /* number of tasks in partition */
         taskid,          /* a task identifier */
         numworkers,      /* number of worker tasks */
         source,          /* task id of message source */
         dest,            /* task id of message destination */
         rows,            /* rows of matrix A sent to each worker */
         averow, extra, offset, /* used to determine rows sent to each worker */
         i, j, k, rc;  /* indexes */
double a[NRA][NCA],          /* matrix A to be multiplied */
          b[NCA][NCB],          /* matrix B to be multiplied */
          c[NRA][NCB],          /* result matrix C */
          t1,t2; // time's momemts
MPI_Status status;

 rc = MPI_Init(&argc,&argv);
 rc|= MPI_Comm_size(M_C_W, &numtasks);
 rc|= MPI_Comm_rank(M_C_W, &taskid);
```

```
 if (rc != MPI_SUCCESS)
    printf ("error initializing MPI and obtaining task ID information\n");
 else
    printf ("task ID = %d\n", taskid);
 numworkers = numtasks-1;
/************************** master task ************************************/
  if (taskid == MASTER)
   {
      printf("Number of worker tasks = %d\n",numworkers);
      for (i=0; i<NRA; i++)
         for (j=0; j<NCA; j++)
            a[i][j]= i+j;
      for (i=0; i<NCA; i++)
         for (j=0; j<NCB; j++)
            b[i][j]= i*j;

 /* send matrix data to the worker tasks */
      averow = NRA/numworkers;
      extra = NRA%numworkers;
      offset = 0;

t1=MPI_Wtime(); // get start time's moment

      for (dest=1; dest<=numworkers; dest++)
      {
         if(dest <= extra)
           rows = averow + 1;
         else
           rows = averow;

         // rows = (dest <= extra) ? averow+1 : averow;
         printf("...sending %d rows to task %d\n", rows, dest);
         MPI_Send(&offset, 1, MPI_INT, dest, FROM_MASTER, M_C_W);
         MPI_Send(&rows,  1, MPI_INT, dest, FROM_MASTER,  M_C_W);
         MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, FROM_MASTER,
                                                              M_C_W);
         MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, FROM_MASTER, M_C_W);
         offset += rows;
      }

 /* wait for results from all worker tasks */
      for (source=1; source<=numworkers; i++)
      {
         MPI_Recv(&offset, 1, MPI_INT, source, FROM_WORKER, M_C_W, &status);
         MPI_Recv(&rows,  1, MPI_INT, source, FROM_WORKER,  M_C_W, &status);
         MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, FROM_WORKER,
                                                              M_C_W, &status);
      }

t2=MPI_Wtime(); // get ended time's momemt
printf ("Multiply time= %.3lf sec\n\n", t2-t1);
```

```
    printf("Here is the result matrix:\n");
    for (i=0; i<NRA; i++)
    {
      printf("\n");
      for (j=0; j<NCB; j++)
        printf("%6.2f  ", c[i][j]);
     }
    printf ("\n");
}

/***************************** worker task ************************************/
  if (taskid > MASTER)
  {
    MPI_Recv(&offset, 1, MPI_INT, MASTER, FROM_MASTER, M_C_W, &status);
    MPI_Recv(&rows,  1, MPI_INT, MASTER, FROM_MASTER,  M_C_W, &status);
    MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, FROM_MASTER,
                                                    M_C_W, status);
    MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, FROM_MASTER,
                                                    M_C_W, &status);

    for (k=0; k<NCB; k++)
      for (i=0; i<rows; i++)
      {
        c[i][k] = 0.0;
        for (j=0; j<NCA; j++)
          c[i][k] += a[i][j] * b[j][k];
      }

    MPI_Send(&a_offset, 1, MPI_INT, MASTER, FROM_WORKER, M_C_W);
    MPI_Send(&a_rows,  1, MPI_INT, MASTER, FROM_WORKER,  M_C_W);
    MPI_Send(&c, a_rows*NCB, MPI_DOUBLE, MASTER, FROM_WORKER,
                                                      M_C_W);
  }
  MPI_Finalize();
} // end of MM_MPI_0.C program
```

Стр**о**ки матрицы [A] распределяются между numworkers процессов по averows штук; причем каждому процессу dest=1÷numworkers посредством вызова функции MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD) пересылаются rows строк (rows=averow, если NRA делится на numworkers н**а**цело и rows=averow+1  в прот**и**вном случае, offset – номер первой пересылаемой строки). Матрица [B] пересылается посредством MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD) всем numworkers  процессам целик**о**м.

После получения процессами данных выполняется умножение посредством внешнего цикла по k=1÷NCA (т.е. по столбцам [A] и строкам  [B] и внутреннего цикла по i=1÷rows (т.е. переданной части строк [A] и [C]):

```
  for (k=0; k<NCB; k++)
   for (i=0; i<rows; i++) // …на узле присутствует только rows строк матрицы [A]
```

```
{
 c[i][k] = 0.0;
  for (j=0; j<NCA; j++)
    c[i][k] += a[i][j] * b[j][k];
}
```

В конце работы все numworkers процессов возвращают rows*NCB строк [C] главному процессу посредством выполнения MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, FROM_MASTER, MPI_COMM_WORLD); причем предварительно номер первой строки offset передается посредством MPI_Send(&offset, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD), а число строк rows - MPI_Send(&rows,1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD). MASTER-процесс принимает эти данные вызовом MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, FROM_WORKER, MPI_COMM_WORLD, &status).

Ниже приводится начало в**ы**дачи программы при NRA=1000 и числе вычислительных узлов 12 (видно, что в случае невозможности деления NRA на numworkers н**а**цело равный extra=NRA%numworkers отстаток распределяется равномерно по первым 4 вычислительным узлам):

```
Number of worker tasks = 12
task ID = 5
task ID = 11
task ID = 10
task ID = 1
task ID = 0
task ID = 6
task ID = 7
task ID = 3
task ID = 2
task ID = 9
task ID = 8
task ID = 12
task ID = 4
...sending 84 rows to task 1
...sending 84 rows to task 2
...sending 84 rows to task 3
...sending 84 rows to task 4
...sending 83 rows to task 5
...sending 83 rows to task 6
...sending 83 rows to task 7
...sending 83 rows to task 8
...sending 83 rows to task 9
...sending 83 rows to task 10
...sending 83 rows to task 11
...sending 83 rows to task 12
```

Эффективный (т.н. *клеточный*, основанный на распределении блоков матриц по ОП процессов по принципу *шахматной доски*) описан в работе [3];

ограничением его является (непременное) условие $M=Q^2$ и $N \% Q = 0$ (N – порядок перемножаемых матриц, M – число процессов, виртуально распределенных на квадратную сетку размеров $Q \times Q$, причем каждый процесс связан с блоком размером $N/Q \times N/Q$ элементов матриц).

На практике процедура умножения матриц является всего лишь частью разрабатываемой проблемно-ориентированной системы; при этом исходные матрицы [A] и [B] вычисляются на предварительных стадиях расчета и необязательно присутствуют в полном виде в ОП главного процесса; программист должен разработать интерфейсные процедуры, позволяющие заполнять исходные матрицы непосредственно в ОП вычислительных узлов (и соответственно получать значения коэффициентов матрицы-произведения).

*Цель работы* – уяснение принципов распределения данных по вычислительным узлам кластера, приобретение практических знаний и навыков в создании MPI-программ.

*Необходимое оборудование* – вычислительный кластер под управлением UNIX-совместимой ОС, предустановленная поддержка MPI, рабочая консоль программиста для управления прохождением пользовательских задач.

*Порядок проведения работы* – студент подготавливает исходные тексты MPI-программ, компилирует их в исполнимое приложение, запускает на счет, проводит анализ полученных данных по заданию преподавателя.

Индивидуальные задания для студентов:

- Определить (ограниченный возможностями компилятора или размером ОП) максимальный размер матрицы [A] при умножении на вектор-столбец [B] для данного вычислительного комплекса (последовательный и параллельный варианты при числе процессоров N=2,3,4,5).
- Выявить зависимость быстродействия (величина, обратная времени выполнения алгоритма) программы MM_MPI от числа процессоров N=2,3,4… (производительность программы при числе процессоров 2 принять единичной).

Дополнительные задания (самостоятельная работа студентов):

- Реализовать программу умножения матриц с выполнением процессорами: а) Умножений элементов $a_{ik} \times b_{kj}$ с целью получения частичной суммы $c_{ij}$

  б) Умножений строки матрицы [A] на столбец [B] с последующим суммированием для получения конечного значения $c_{ij}$. Какая особенность алгоритма выполнения действий на процессорах потребуется в этом случае? Сравнить быстродействие этих программ с MM_MPI_0.C при различном числе процессоров и применении широковещательных операций обмена.
- Запрограммировать (описанный в работе [3], стр.179÷180) ‘шахматный’ алгоритм умножения матриц; сравнить его производительность с

MM_MPI_0.C.

*Вопросы для самопроверки:*

1. В каком случае время умножения матриц будет больше – в случае выполнения MM_SER или MM_MPI_0 при числе процессоров N=2?
2. Какими соображениями следует пользоваться при разработке стратегии распределения вычислений и блоков данных между процессами при распараллеливании алгоритмов?
3. Каким образом целесообразно распределить блоки исходных и вычисленной матриц в случае процедуры нахождения матричного произведения? При умножении/делении матрицы на скаляр? При транспонировании матрицы?
4. Предложить более эффективную (относительно вышеразобранных) стратегию распределения больших матриц по вычислительным узлам (учесть некратность числа строк и столбцов матрицы числу процессоров).

## 5. *Лабораторная работа 6.* Автоматизация разработки параллельных MPI-программ с использованием проблемно-ориентированного языка НОРМА

*Общие сведения.* Как сказано выше, программирование в технологии MPI является достаточно низкоуровневым и поэтому требует высокой квалификации разработчика и значительных затрат (в т.ч. временны**х**) при создании программного продукта. Имеющиеся системы автоматизации распараллеливания недостаточно эффективны (при высокой стоимости и **у**зости области применения). Кроме того, существуют фундаментальные соображения (характерный размер зерна распараллеливания программы намного превосходит характерный размер типичного оператора языка программирования), следствием которых является практическая невозможность качественного распараллеливания последовательной программы, записанной на привычных языках программирования [2].

Т.о. исключение этапа последовательной алгоритмизации потенциально может привести к возможности качественного автоматического создания параллельных программ. Один из примеров реализации такого подхода – предложенная еще в 60-х г.г. в ИПМ им.М.В.Келдыша РАН концепция непроцедурного языка НОРМА (***Н****епроцедурное* ***О****писание* ***Р****азностных* ***М****оделей* ***А****лгоритмов* или *НОРМАльный уровень общения прикладного математика с компьютером*, http://www.keldysh.ru/pages/norma).

*Я*зык НОРМА называют *декларативным* вследствие упора именно на описание правил вычисления значений, а не на исчерпывающе подробную конкретизацию алгоритма. Разработчик прикладных программ абстрагируется от особенностей конкретных ЭВМ и м**ы**слит в привычных терминах своей предметной области (сеточные методы математической физики, в основном метод конечных разностей - МКР). Система НОРМА включает синтезатор, назначением которого является преобразование НОРМА-текста в (один из привычных) язык**о**вых стандартов параллельного или последовательного программирования (в настоящее время существует возможность получения Fortran-MPI, Fortran-DVM, Fortran-77 или соответствующих С-текстов). Эффективный автоматический синтез параллельного кода на основе НОРМА-программы достигается определенными ограничениями языка, важное из которых – отсутствие многократного присваивания (при этом несущественна последовательность операторов, отсутствуют глобальные переменные, запрещена рекурсия, нет побочных эффектов при вычислениях и др., [7]). Исходный текст на НОРМА в высшей степени бл**и**зок к записи численного метода решения конкретной задачи. В записи на языке НОРМА отсутствуют избыточные информационные связи (полный анализ которых как раз и является ‘ахиллесовой пятой’ систем выявления скрытого параллелизма), что и позволяет реализовать эффективное автоматическое распараллеливание.

Однако для создания НОРМА-программы все же требуется программист, знакомый с правилами языка и формально ‘набивающий’ исходный текст программы. Особенности НОРМА позволяют сделать следующий шаг в сторону, противоположную операции разработки программы в виде последовательной записи операторов, т.е. вообще не применять текстовое представление программы. Логично снабдить синтезатор НОРМА интерактивной графической подсистемой, позволяющей манипулировать с программой как с отображением в виде графики и гипертекста совокупности объектов (включая формулы, вводимые пользователем в отдельные поля гипертекстовых форм) [2].

С этой целью на кафедре ИТ-4 МГАПИ разработана система ‘Интерактивная НОРМА’ (http://norma.deniz.ru), позволяющая разрабатывать параллельные программы практически без написания исходных текстов на языке программирования.

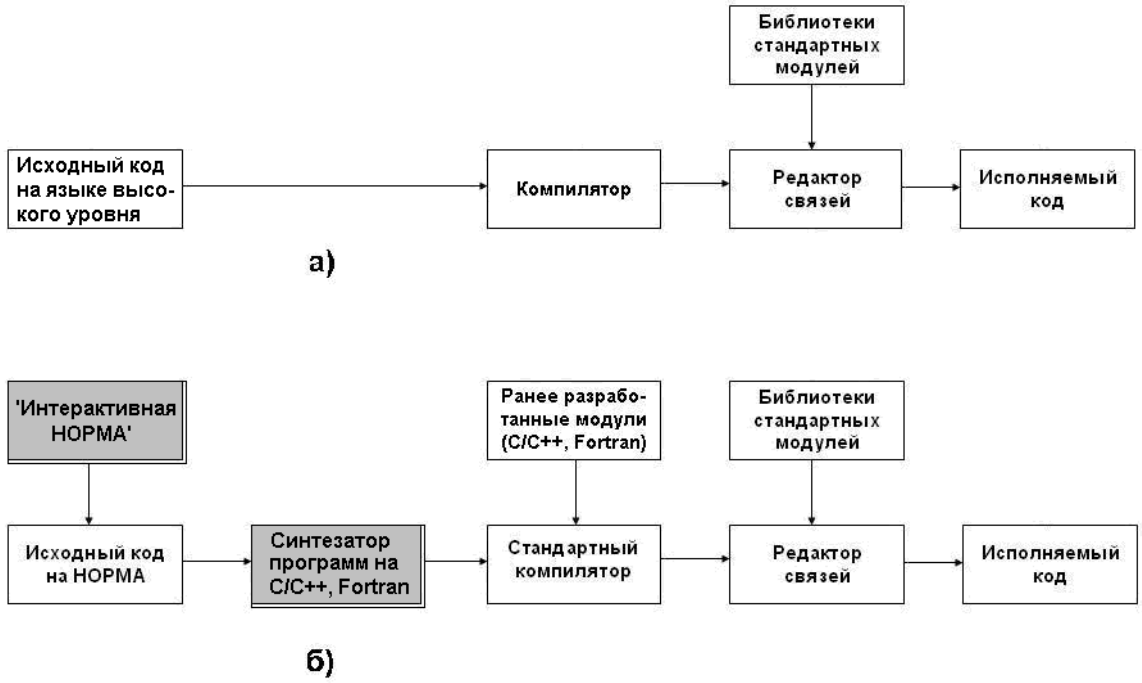



Рисунок 7.— Этапы создания исполняемых программ: а) - классический подход, б) - использование НОРМА-программирования и оболочки ‘Интерактивная НОРМА’)

Как видно из рис.7, при создании параллельных программ с использованием системы ‘Интерактивная НОРМА’ используется дополнительная программа–преобразователь (синтезатор) исходного кода НОРМА-программы в C/C++ и/или Fortran-программу).

*Цель работы* – приобрести начальные навыки разработки направленных на решение задач матфизики параллельных программ с использованием системы программирования ‘Интерактивная НОРМА’.

*Необходимое оборудование* – подключенная к сети InterNet персональная ЭВМ под под управлением ОС Windows, вычислительный кластер под управлением UNIX-совместимой ОС с предустановленной поддержкой MPI, рабочая консоль программиста.

*Порядок проведения работы* – студент с помощью системы ‘Интерактивная НОРМА’ подготавливает программу на языке НОРМА, синтезирует MPI-программу на C или Fortran’е, переносит ее на вычислительный кластер, компилирует в исполнимое приложение, запускает на счет, проводит анализ полученных данных по заданию преподавателя.

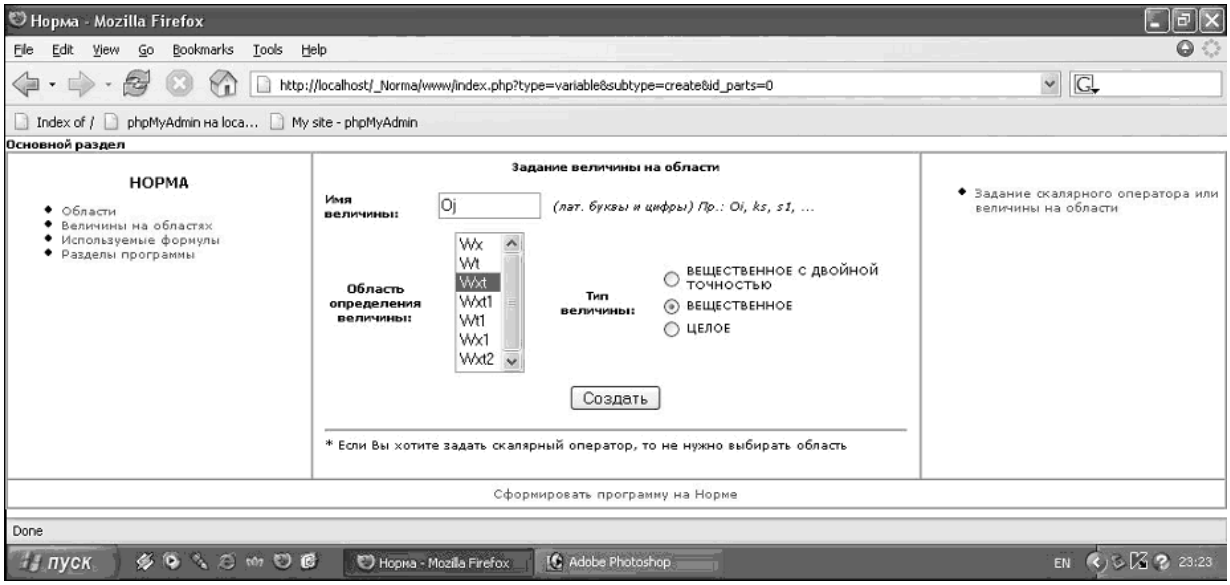



Рисунок 8.— Окно определения величин на области в системе ‘Интерактивная НОРМА’

Индивидуальные задания для студентов:

*Вопросы для самопроверки:*

1. В чем заключаются трудности автоматизации процесса распараллеливания алгоритмов? Какие системы подобного рода известны?
2. Каков основной принцип распараллеливания алгоритмов с использованием языка НОРМА?
3. В чем проявляется декларативность языка НОРМА? Каковы ограничения НОРМА?
4. Почему именно для НОРМА эффективен принцип разработки программ с использованием интерактивной оболочки? Чем отличается подобная оболочка от известных систем (например, Delphi/C++Builder, Visual C и т.п.)?

**Список литературы**


1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. - СПб.: БХВ-Петербург, 2002. -608 с.
2. Лацис А.О. Как построить и использовать суперкомпьютер. –М.: Бестселлер, 2003. –240 с.
3. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI. –Минск:, БГУ, 2002. -325 с. (http://pilger.mgapi.edu/metods/1441/pos_mpi.pdf)
4. Антонов А.С. Введение в параллельное программирование (методическое пособие). –М.: НИВЦ МГУ, 2002, -69 с. (http://pilger.mgapi.edu/metods/1441/antonov.pdf)
5. Антонов А.С. Параллельное программирование с использованием технологии MPI (учебное пособие). –М.: НИВЦ МГУ, 2004, -72 с. (http://parallel.ru/tech/tech_dev/MPI, http://pilger.mgapi.edu/metods/1441/mpi_book.pdf)
6. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. –Ростов-на-Дону.: Издательство ООО “ЦВВР”, 2003, -208 с. (http://pilger.mgapi.edu/metods/1441/book.pdf)
7. Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Задыхайло И.Б. Норма. Описание языка. Рабочий стандарт. Препринт ИПМ им. М.В.Келдыша РАН, № 120, 1995, -50 с.

Учебное издание.

**Баканов Валерий Михайлович,**
**Осипов Денис Васильевич**

# Введение в практику разработки параллельных программ в стандарте MPI

Учебно-методическое пособие
по выполнению лабораторных работ

---

Подписано в печать __.____.2005 г.
Формат 60 × 84 1/16.
Объем ?? п.л. Тираж 100 экз. Заказ ___.

Отпечатано в типографии Московской государственной
Академии приборостроения и информатики.
107846, Москва, ул.Стромынка, 20.