

МГУПИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ

Кафедра 'Персональные компьютеры и сети'

В.М.Баканов, Д.В.Осипов

**ПАРАЛЛЕЛЬНОЕ
ПРОГРАММИРОВАНИЕ
В СТАНДАРТЕ MRI**

Учебно-методическое пособие по
выполнению лабораторных работ

Москва, 2006

УДК 681.3

Параллельное программирование в стандарте MPI: Учебно-методическое пособие по выполнению лабораторных работ / В.М.Баканов, Д.В.Осипов. -М.: МГУПИ, 2006. –78 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.

Также рассмотрена система ‘Интерактивная НОРМА’, предназначенная для автоматизации разработки ориентированных на решение задач математической физики разностными методами параллельных программ и позволяющая специалисту в конкретной проблемной области без навыков программирования генерировать исходные тексты на Fortran’е и С с использованием библиотеки MPI.

Разобраны основные функции библиотеки MPI, стандартные приемы MPI-программирования, возможности загрузчика исполняемых программ. Знание основ программирования на языке С и Fortran априори предполагается. При разработке стандартных MPI-программ дискусируются стратегии распараллеливания. Создаваемые сетевые приложения работоспособны в многопроцессорной среде архитектуры MPP (*Massively Parallel Processing*); рассматривается, в частности Linux-вычислительный кластер кафедры ИТ-4 МГУПИ.

Последняя версия данного методического пособия может быть получена в виде файла http://pilger.mgapi.ru/methods/1441/mpi_lab.pdf. Перед проведением работ желательно ознакомиться с конспектом лекций по дисциплине ‘Параллельные вычисления’ (см. <http://pilger.mgapi.ru/methods/1441/parr.pdf>).

Печатается по решению Редакционно-издательского совета Московского государственного университета приборостроения и информатики.

Рецензент: профессор, к.т.н. Зеленко Г.В.

Содержание

Стр.

Введение.....	
1 <i>Лабораторная работа 1.</i> Ознакомление с архитектурой вычислительного LINUX-кластера, основами его администрирования и принципами работы пользователей.....	
2 <i>Лабораторная работа 2.</i> Жизненный цикл процессов и простейший обмен данными между ними, тупиковые ситуации.....	
3 <i>Лабораторная работа 3.</i> Определение параметров коммуникационной сети вычислительного кластера.....	
4 <i>Лабораторная работа 4.</i> Простые MPI-программы (численное интегрирование).....	
5 <i>Лабораторная работа 5.</i> Умножение матриц – последовательная и параллельные версии	
6 <i>Лабораторная работа 6.</i> Автоматизация разработки параллельных MPI-программ с использованием проблемно-ориентированного языка NORMA.....	
Список литературы.....	

Введение

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 последовательностью вычислений, распределения данных и обменов информацией между процессами) 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 (*High 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 вызовов функций специальной библиотеки (заголовочные 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 переносимой 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. права 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 версии MPI 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 сторонним 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 вызовы 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 сложна 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 – выдача (часто в графическом виде) информации об обменах в параллельной программе.

Строго говоря, технология MPI подразумевает подход 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 громоздки при написании (для каждого ВУ требуется отдельный исходный 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 таковой 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 [4,5], для желающих серьезно заняться проблемой рекомендуются работы [1,3].

Т.к. время обмена данными между ветвями намного (на порядки) больше времени доступа к собственной (локальной) памяти, распределение работы между процессами должно быть ‘крупнозернистым’ [1]. Типичный размер этого зерна (*гранулы*) 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 par-*

allelism). На размер гранулы 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).

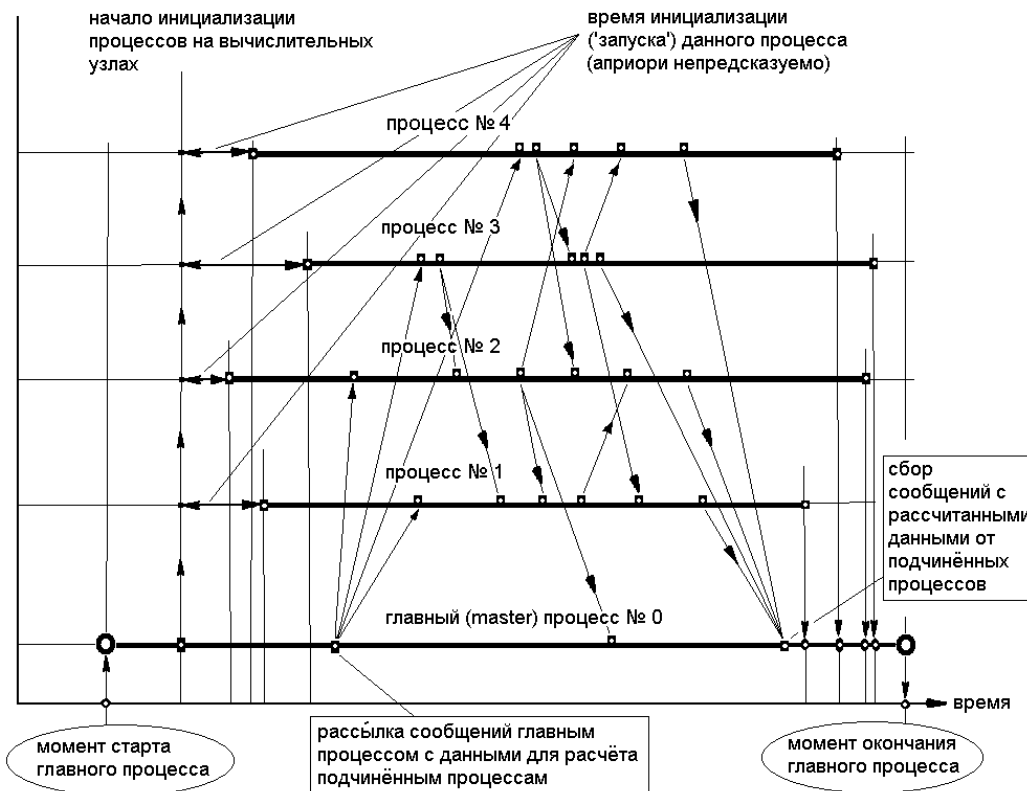


Рисунок 1 — Схема жизненного цикла процессов и обмена данными между ними.

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

С точки зрения программиста базовых методик работы (*парадигм*) существуют две - данные могут передаваться через разделяемую оперативную память (ОП, причем синхронизация доступа 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 двух парадигм может быть имитирована 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 противовес 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 коммутатора (о понятии коммутатора 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 'коммутатор', который можно рассматривать как дескриптор (номер) коллектива (он ограничивает область действия данной функции соответствующим коллективом). Коммутатор 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 коммутаторы. Имеют место включающий только самого себя как процесс коммутатор MPI_COMM_SELF и (заведомо неверный) коммутатор 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 коммутаторе процесса-получателя, а процесс-получатель должен вызвать одну из процедур приема с указанием того же коммутатора (в некоторых случаях необязательно знать точный номер процесса-отправителя в заданном коммутаторе).

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 операции. 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 одновременного 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 выдачей 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:

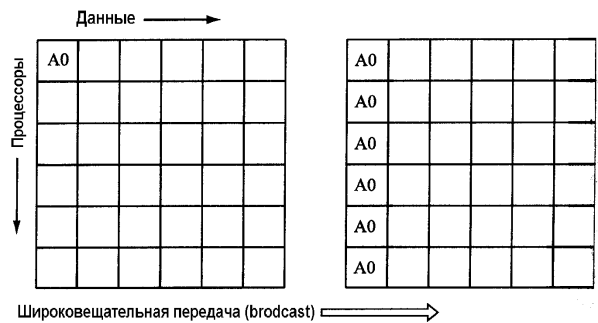
- Блокирующий режим - управление вызывающему процессу возвращается только после того, как данные приняты или переданы (или скопированы во временный буфер).
- Неблокирующий режим - управление вызывающему процессу возвращается немедленно (т.е. процесс блокируется до завершения операции) и фактическая приемопередача 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 ниже-расположенной таблицы схематично показаны процессоры (вычислительные

узлы) в виде строк и хранимые 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>):

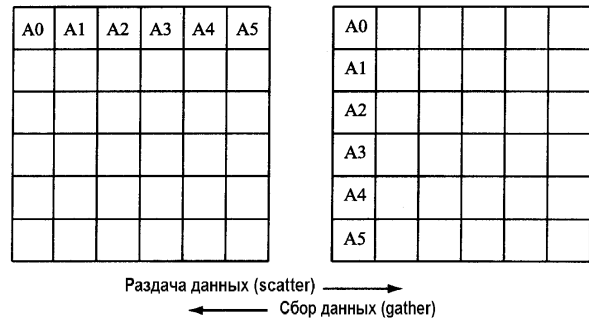
1. Тип **broadcast**: *один-всем*

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 послыки, `recvbuf, recvcount, recvtype` – то же для буфера сбора данных, `source` – номер процесса сбора данных, `comm` - коммуникатор

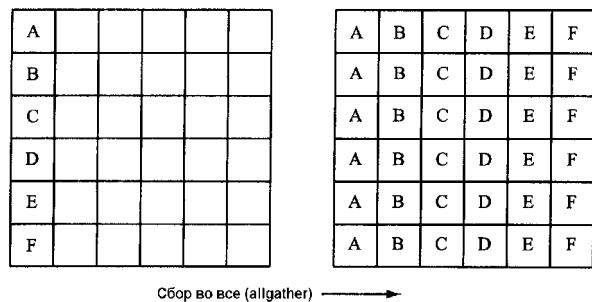


Тип **gather**: *каждый-одному*

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)`.

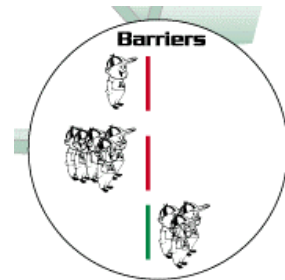
A0	A1	A2	A3	A4	A5
B0	B1	B2	B3	B4	B5
C0	C1	C2	C3	C4	C5
D0	D1	D2	D3	D4	D5
E0	E1	E2	E3	E4	E5
F0	F1	F2	F3	F4	F5

A0	B0	C0	D0	E0	F0
A1	B1	C1	D1	E1	F1
A2	B2	C2	D2	E2	F2
A3	B3	C3	D3	E3	F3
A4	B4	C4	D4	E4	F4
A5	B5	C5	D5	E5	F5

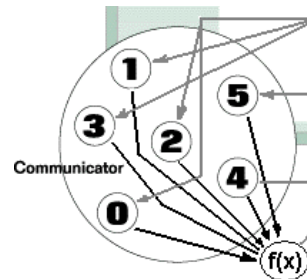
Раздача/сборка от всех всем (alltoall)

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 вызовов `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 всего 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 управление содержимое `buffer` гарантированно скопировано в `buffer` всех процессов.

Коллективные функции `MPI_Bcast` и `MPI_Reduce` можно выразить через парные операции `MPI_Send` и `MPI_Recv`. Например, операция `MPI_Bcast` для рассылки 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-1
    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 С список header-файлов обязательно должен быть дополнен строкой #include "mpi.h" (include 'mpif.h' в случае Fortran'a).

Общая структура простейшей параллельной С-программы приблизительно такова (типовой 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 С-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'a-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 ÷ 6] и некоторые другие (исходные тексты используемых в практикуме программ могут быть получены с http://pilger.mgapi.edu/metods/1441/mpi_src.zip).

Традиционно включаемые в руководства по MPI задачи сортировки, операций над графами, решение систем линейных алгебраических уравнений, использование возможностей MPI по поддержанию пользовательских топологий могут быть исследованы самостоятельно (рекомендуется работа [3,5,6] и ресурсы <http://pilger.mgapi.edu/metods/1441/paralab.pdf>, <http://pilger.mgapi.edu/metods/1441/paralab.zip> (программная система изучения и исследования методов параллельных вычислений *ParaLab*).

Единственным путем профессионального освоения MPI является постоянная практика (ручное составление параллельных программ и многократная компиляция/прогонка по принципу 'пока не получится'). Для желающих более полно освоить технологию программирования в MPI-стиле рекомендуется изучение руководств [1 ÷ 8].

1 Лабораторная работа 1. Ознакомление с архитектурой вычислительного LINUX-кластера, основами его администрирования и принципами работы пользователей

Цель работы – ознакомление с архитектурой, принципами работы кластерной вычислительной системы и практикой удаленного доступа пользователя к вычислительным ресурсам кластера.

Теоретическая часть. Вычислительный кластер является многопроцессорной вычислительной системой (МВС) архитектуры MPP (*Massively Parallel Processing*), отличием которой является наличие *оперативной памяти, распределенной по вычислительным узлам* (ВУ). Вычислительный кластер суть вычислительная система, состоящая из нескольких связанных между собой ЭВМ, расположенных в едином корпусе или соединенных скоростным каналом связи и используемых как *единый, унифицированный вычислительный ресурс*.

Специализированные вычислительные кластеры в большинстве случаев работают под управлением Linux как наиболее мобильной машинно-независимой ОС. Для реализации параллельных вычислений обычно не требуются многие возможности ОС, столь чрезмерно развитые в Windows (например, графический пользовательский интерфейс - GUI).

На рис.1.1 изображена *укрупненная схема* сетевых соединений для вычислительного кластера, удовлетворяющих минимуму вышеуказанных требований. Важно отметить наличие двух (*логически разделенных*) компьютерных сетей – *сеть управления* (служит для управления ВУ со стороны управляющей ЭВМ посредством инструкций типа `ssh`, данная сеть особенно нагружена в *период старта и окончания задания*) и *сеть обмена данными* (она же *коммуникационная*, служит для обмена данными между ВУ посредством MPI, нагружена в *период выполнения задания*). Для реализации особо мощных МВС применяют третью сеть для поддержки NFS (*Network File System*); при двух сетях NFS обычно использует *сеть управления*. Естественно, все эти сети могут быть виртуализированы (т.е. использовать *единую физическую сеть*).

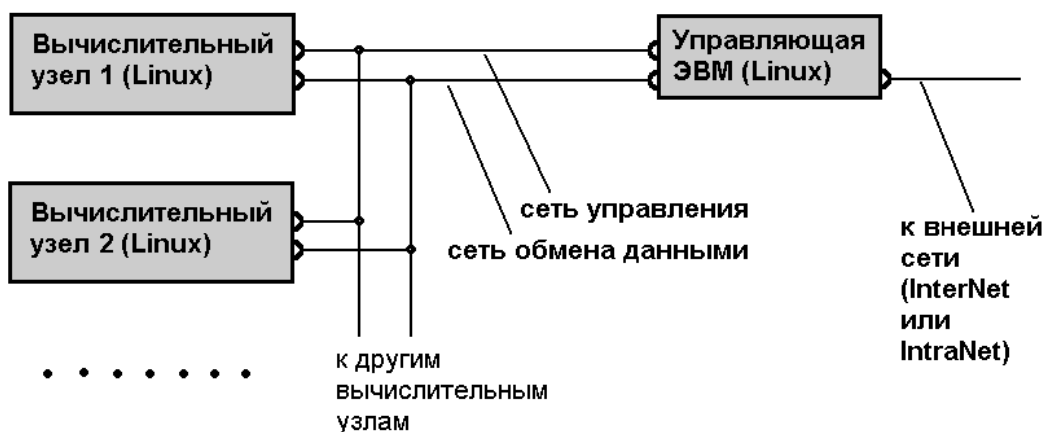


Рисунок 1.1 — Укрупненная схема сетевых коммуникаций многопроцессорной

ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ (ВЫЧИСЛИТЕЛЬНОГО КЛАСТЕРА).

Схема кафедрального кластера вместе с управляющей ЭВМ, управляющей и коммуникационной сетью приведена на рис.1.2 (данная МВС является *инструментальной*, используемой в первую очередь для целей отладки приложений и обучения).

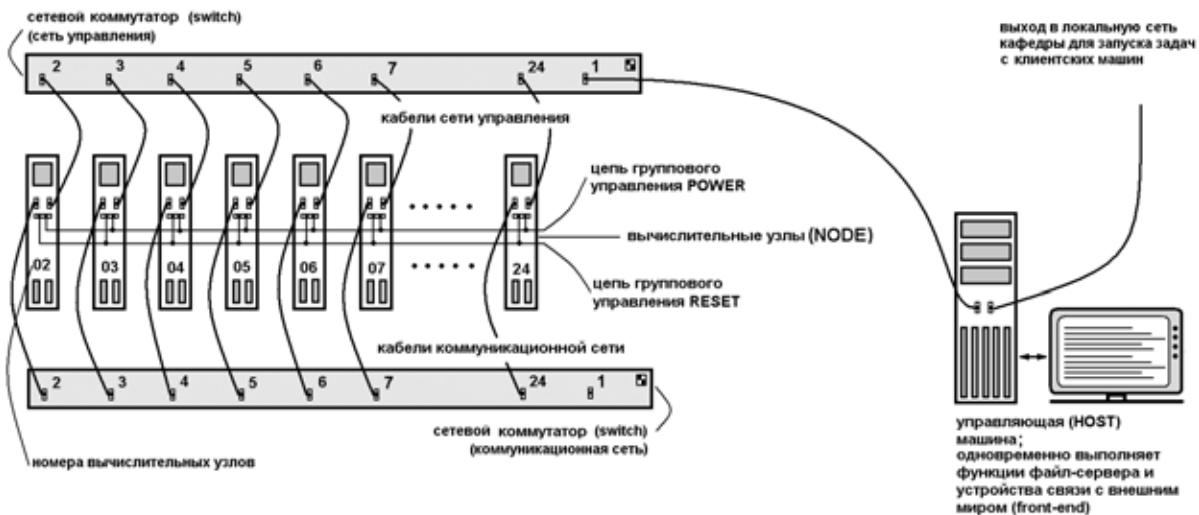


Рисунок 1.2 — Схема вычислительного кластера инструментального уровня кафедры ИТ-4 МГУПИ.

Кафедральный кластер представляет собой вычислительную систему, состоящую из системных узлов на базе процессоров AMD ATHLON, соединённых между собой через коммутаторы сетью Ethernet; данная МВС является типичным Beowulf-кластером (<http://www.beowulf.org>, название дано в честь героя древней скандинавской саги). На вычислительных узлах отсутствует аппаратура ввода/вывода (клавиатура/монитор); для технического обслуживания имеется один текстовый монохромный монитора и клавиатура. Системные блоки установлены в стандартный приборный шкаф с системой регулирования температуры путем вентиляции.

Для соединения ВУ между собой используется физическая топология 'звезда' относительно коммутатора, применяемый сетевой протокол – TCP/IP). ВУ не имеют адресов в InterNet, но обладают адресами в двух замкнутых, не маршрутизируемых наружу сетях – *сети управления* (используется для выполнения gsh) и *сети коммуникаций* (используется MPI).

Контроль и доступ пользователей к вычислительной системой осуществляется через отдельную ПЭВМ (*сервер доступа*), в данном случае выполняющую также роль управляющей ЭВМ (УМ) и файл-сервера; основные характеристики МВС приведены в табл. 1.1, вид аппаратуры – на рис.1.3.

Таблица 1.1. Основные технические характеристики МВС

Количество вычислительных узлов	до 24 блоками по 6
Число процессоров на одном узле	1
Рабочая частота процессора (GHz)	1,6

Cache-память процессора (kb)	256
Разрядность процессора (bit)	32
Объем оперативной памяти одного узла (Mb)	512
Объем дискового пространства одного узла (Gb)	80
Коммуникационная сеть	Gigabit Ethernet, 1000 Mbit/sec
Управляющая сеть	Fast Ethernet, 100 Mbit/sec
Латентность сети (мксек)	≈40
Энергопотребление всей системы (kW)	2
Тепловыделение (МДж/час)	7
Размер приборного шкафа (мм)	2100×800×600

На все узлы кластера установлена операционная система (ОС) Linux (*) Slackware 11.0 (ядро 2.6.17, файловая система ext3).

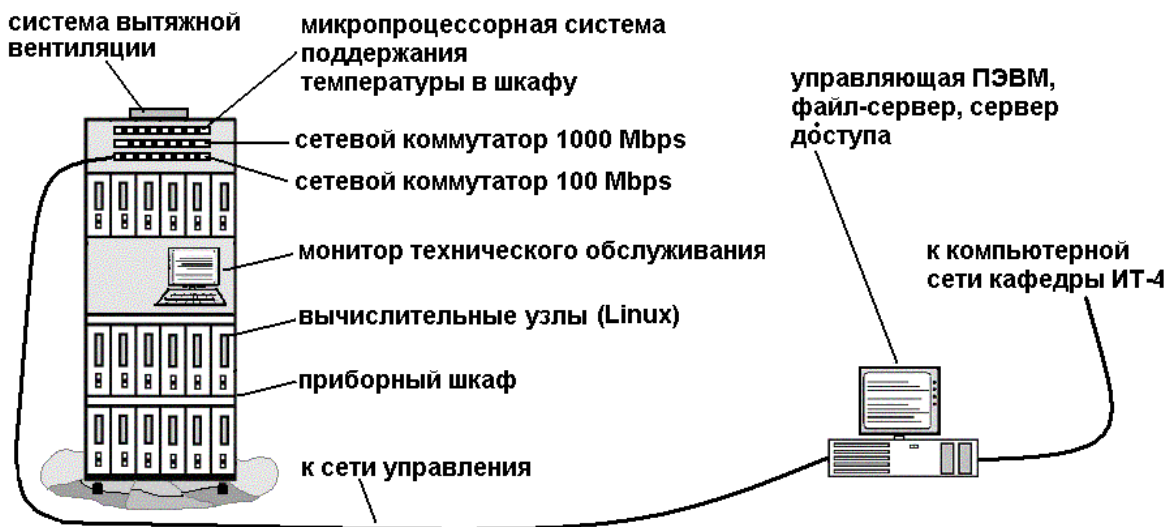


Рисунок 1.3 — Расположение аппаратуры вычислительного кластера кафедры ИТ-4 МГУПИ.

При данном конструктивном исполнении стоимость (в пересчете на один ВУ) составляет 430 \$US. Недостатком исполнения является неиспользование двух/четырёхядерных процессоров, в результате чего повысились габариты и энергопотребление МВС,

Пользовательские приложения компилируются и линкуются (с коммуникационной библиотекой – в данном случае это MPI версии LAM 7.0.6/MPI 2 C++ Indiana University) на управляющей ЭВМ, выполняемые файлы помещаются в каталог пользователя (напр., /home/quest), являющийся общим

* Windows-кластеры значительной мощности до настоящего времени остаются экзотикой (например, в 29-й редакции от июня 2007 г. списка Top500.org подобные МВС занимают только позиции 106 и 193) в силу известных причин (несмотря на активно продвигаемые MS решения класса *Windows Compute Cluster Server - WCCS*). Кроме того, автор разделяет известное правило ‘...не стоит класть все *имеющиеся* яйца в единую корзину’.

(*shared*) для УМ и всех вычислительных узлов (это обеспечивается средствами NFS – *Network File System*). Далее УМ специальным образом (см. ниже) стартует приложение на каждом из ВУ (необходимые системные библиотеки расположены в NFS-каталоге /*common*).

Управление и администрирование МВС реализовано с помощью WEB-интерфейса (рис.1.4, для этого на УМ функционирует WEB-сервер Apache); в качестве СУППЗ (*Система Управления Прохождением Параллельных Задач*) применена разработанная в ИПМ им.М.В.Келдыша РАН.

Администрирование МВС осуществляется с УМ или посредством удаленного доступа (УД) суперпользователем root, рядовые пользователи имеют ограниченный доступ (в соответствии с присвоенными администратором правами, обычно только к определенному каталогу файловой системы) к ресурсам кластера исключительно через УМ в режиме УД. Возможность *захода* (т.е. перехвата управления удаленной ЭВМ с собственного терминала посредством УД по сети) на ВУ имеет исключительно администратор (супер-пользователь root).

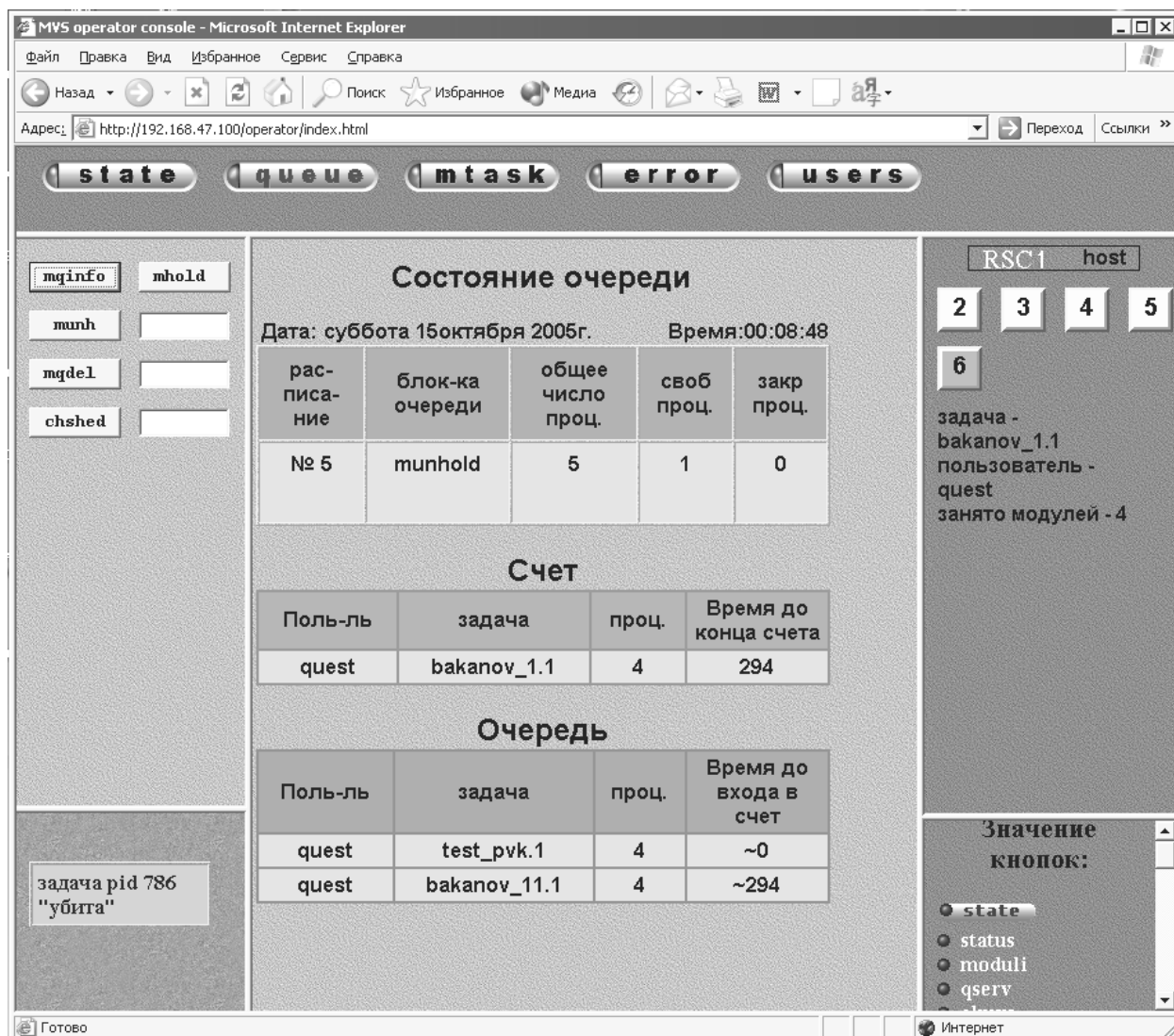


Рисунок 1.4 — WEB-базированный пульт оператора вычислительного кластера.

Система управления прохождением задач представляет собой совокупность взаимодействующих процессов на управляющей ЭВМ, основные из которых (см. рис.1.5) следующие (*):

- `runmvs` – клиент, вызываемый на сервере доступа, обеспечивает связь пользователя с системой управления прохождением задач (именно эта программа вызывается при выполнении всех пользовательских команд),
- `runmvsd` – сервер запросов, отвечающий за связь с клиентом и обработку клиентских запросов; функционирует на УМ,
- `qserver` – сервер очередей параллельных задач, который ведет очередь параллельных задач согласно заданной политике планирования очередей; функционирует на УМ,
- `mrun` – процесс-‘запускатель’ задачи, функциями которого являются определение свободных вычислительных модулей, выделение требуемого параллельного ресурса для задачи, порождение (с помощью системного вызова `fork`) специального процесса-менеджера `manager`, который запускает задачу и контролирует ее; функционирует на УМ,
- `sleeper` - процесс, контролирующей время выполнения задачи.

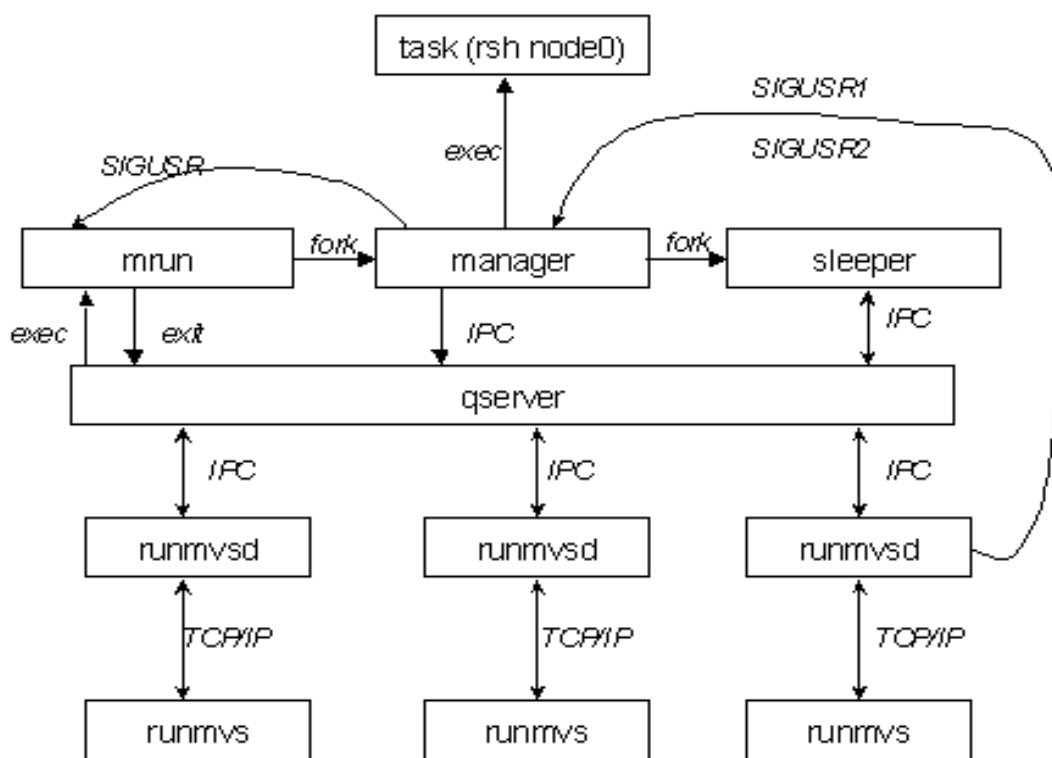


Рисунок 1.5 — Схема взаимодействия процессов.

* Баранов А.В., Лацис А.О., Сажин С.В., Храпцов М.Ю., Шарф С.В. Руководство системного программиста (администратора) системы управления прохождением задач МВС-1000/16 (версия 1.23).

Все задачи пользователей делятся на три категории – *отладочные, пакетные и фоновые*.

Отладочные задачи – это короткие по времени задачи, которые запускаются исключительно в целях отладки.

Пакетные задачи – это средние по времени задачи, которые производят реальные расчеты и выполняются, не прерываясь.

Фоновые задачи – задачи с большим временем счета, которые могут прерываться системой. Для фоновой задачи пользователь должен явно указать *квант* – минимальное время счета фоновой задачи, в течение которого задачу прерывать нельзя.

Планирование очередей в каждый момент времени производится в соответствии с параметрами текущего *режима планирования*. В системе может быть несколько *расписаний* режимов планирования, задающих некоторые общие параметры и определяющих все множество доступных на данный момент режимов. Переключение расписаний осуществляется по директиве оператора, а переключение режимов внутри расписания - автоматически.

Режим планирования определяется следующими параметрами:

- дата и время включения режима,
- максимальное время, отведенное для отладочных задач,
- максимальное время, отведенное для пакетных задач,
- число процессоров, которые ‘резервируются’ для отладочных задач,
- шкала приоритетов пользователей.

На всех ВУ используются два *сетевых интерфейса* (символьных имен, через которые можно обращаться к аппаратным средствам ЭВМ) – eth0 для *сети управления* (Fast Ethernet) и eth1 для коммуникационной сети (Gigabit Ethernet); причем интерфейс eth0 связан со смонтированным на системной плате сетевым адаптером, а eth1 – с дополнительной PCI-сетевой картой Gigabit Ethernet.

Директивы загрузки драйверов (используется *модульное ядро*) сетевых карт осуществляется в файле /etc/rc.netdevice

```
/sbin/modprobe forcedeth # драйвер для чипсета системной платы ASUS K8N-VM  
/sbin/modprobe 3c59x # драйвер встроенной сетевой карты Realtec ALC8201CL  
/sbin/modprobe skge # драйвер сет. карты D-Link DGE-530T (чип Marwell 88E001)
```

Для настройки сетевых интерфейсов применяется утилита с командной строкой ifconfig; для автоматизации настройки целесообразно нужные команды добавить в файл /etc/rc.d/rc.inet1 (выполняющийся при загрузке ОС скрипт, служащий для настройки сетевых интерфейсов). Эти команды могут быть, например, такими (первая строка соответствует управляющей сети, вторая - коммуникационной):

```
/etc/ifconfig eth0 10.100.37.N broadcast 10.100.63.255 netmask 255.255.224.0  
/etc/ifconfig eth1 10.100.5.N broadcast 10.100.31.255 netmask 255.255.224.0
```

где N – номер ВУ ($N=2 \div 24$). Соответственно символьные имена ВУ суть $rscu_2 \div rscu_24$ для сети управления и $rscs_2 \div rscs_24$ для цепи обмена данными, для управляющей машины $rscu_1$ и $rscs_1$ (см. рис. 1.1).

В случае использования единой физической сети для сетей $rscu_n$ и $rscs_n$ (режим *виртуализации* сетей) соответствующие настройки могут быть таковы (нотация $eth0:0$ как раз и дает возможность связать еще одну виртуальную сеть с тем же самым физическим сетевым адаптером, здесь коммуникационная сеть использует $eth0:0$):

```
/etc/ifconfig eth0 10.100.37.N broadcast 10.100.63.255 netmask 255.255.224.0  
/etc/ifconfig eth0:0 10.100.5.N broadcast 10.100.31.255 netmask 255.255.224.0
```

В конфигурационных файлах `/etc/hosts.allow` и `/etc/hosts.deny` находятся адреса удаленных ЭВМ, доступ которых по протоколу TCP/IP (с использованием служб `rlogin`, `rsh`, `telnet`, `finger`, `talk`) к данному компьютеру соответственно предоставляется или не предоставляется (если адреса нет ни в одном из указанных файлов, доступ разрешен). Список ЭВМ и их пользователей, которым разрешено выполнять команды `rlogin` (заход на удаленную ЭВМ) и `rsh` (выполнение заданной команды на удаленной ЭВМ), хранится в файле `/etc/hosts.equiv` (если ЭВМ или пользователя в этом файле нет, то с этой машины этот пользователь никакими силами ни `rsh`, ни `rlogin` на данную машину выполнить не сможет; причем обратное неверно: если в `/etc/hosts.equiv` пользователь и ЭВМ описаны, то попытка пользователя зайти подлежит дополнительной проверке). В рассматриваемом случае на ВУ в этом файле находится строка `rscu_1 root`, что дает возможность суперпользователю получать полное управление над ВУ. При этом выполняется дополнительная проверка – в файле `/etc/inetd.conf` в строках, отвечающих за настройку TCP-демонов `login` и `shell` необходимо указать следующее (сим подтверждается разрешение суперпользователю `root` заходить с УМ на ВУ без ввода пароля):

```
shell stream tcp nowait.10000 root /usr/sbin/tcpd in.rshd -lh  
login stream tcp nowait root /usr/sbin/tcpd in.rlogind -lh
```

Параметр настройки `nowait.10000` отменяет режим выключения InterNet-демона `inetd` на минуту в случае излишне частых обращений; этот режим по умолчанию включен для пресечения хакерских атак типа *DoS (Denial of Service, отказ в обслуживании)*.

Файловая система `ext2` преобразована в журналируемую `ext3` путем применения команды `tune2fs -j /dev/hda1` и соответствующей коррекции файла `/etc/fstab`.

Типовым приемом изменения настроек каждого ВУ является ‘заход’ суперпользователя `root` на этот ВУ с УМ с помощью команды `rlogin rscu_N` (возврат по `logout`); при вышеуказанных настройках пароль затребован не будет.

Для удаленной работы пользователей на МВС применяются клиентские программы TelNet, SSH, PuTTY и др. (для любителей FAR'a может быть интересен WinSCP-базирующийся Far SCP/SFTP client plugin; подробнее см. <http://winscp.net/eng/far.php>).

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 следующие 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 процесс `sleep` завершается (в случае завершения задачи 'естественным' путем менеджер завершает `sleep` и выполняет вышеперечисленные действия). В случае истечения заданного времени выполнения задания менеджер, получив сигнал о завершении `sleep`, 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 том же 'Руководство системного программиста').

Создание выполняемого файла для последовательного исполнения (компиляция и редактирование связей модуля `mm_ser.c`) осуществляется пользователем командой (при использовании компиляторов `gcc` и `icc`, исполняемый файл получит название `mm_ser` без расширения)

```
gcc -o mm_ser mm_ser.c
```

```
icc mm_ser.c -o mm_ser
```

То же для Fortran-файла `mm_ser.f`

```
f77 mm_ser.f -o mm_ser
```

запуск на исполнение – набором имени исполняемого модуля `mp_ser` (последовательная программа выполняется на управляющей машине).

Создание выполняемого файла для параллельного исполнения (компиляция и редактирование связей C- и Fortran-файлов `program.c` и `program.f` соответственно) осуществляется пользователем командой (исполняемый файл получит название `program` без расширения):

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

```
mpif77 -o program program.f
```

Запуск программы на параллельное выполнение осуществляется с помощью пакетного файла (*скрипта*) `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 среды MPI по умолчанию):

```
mpirun -np 7 my_program
```

Заметим, что при запуске задачи на `N` вычислительных узлах львиную долю 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 (выключены соответствующие ПЭВМ, не загружен Linux или узлы в данный момент заняты сторонними задачами) задание ставится в очередь до момента освобождения нужного количества узлов.

При работе в Linux могут возникнуть проблемы, связанные с некорректностью прав доступа к файлам и каталогам. Файлы и каталоги в UNIX-подобных операционных системах обладают набором атрибутов, среди кото-

рых важное значение имеет *владелец* и *группа* (это дает возможность организовать гибкое разграничение доступа к файлам и каталогам). Для просмотра всех атрибутов файла удобно использовать команду `ls -l` (или `dir -l`), выдающую организованную в виде столбцов информацию о файлах текущего каталога, для конкретного файла строка данных имеет такой вид:

<i>Права доступа к файлу</i>	<i>Число связей файла</i>	<i>Пользователь</i>	<i>Группа</i>	<i>Размер файла, байт</i>	<i>Дата последнего изменения файла</i>			<i>Имя файла</i>
-rwxrwxrwx	1	quest	users_1	2165	May	29	2005	pi_ser

В первом столбце **выдачи** приводится список *прав доступа* к файлу. Первый символ обозначает тип файла ('-' - обычный файл, 'd' - каталог, 's' - доменное гнездо (*socket*), 'p' - именованный канал, 'l' - символическая ссылка и т.д.); далее следуют три группы по три символа, определяющие собственно права доступа: первая группа трехсимвольная символов - права для владельца-пользователя, следующая группа - для владельца-группы и последняя группа - права для всех остальных пользователей. При этом '-' означает отсутствие права доступа, 'r' - право на чтение файла, 'w' - право на запись в файл или его удаление, 'x' - право на выполнение файла (в приведенном примере все имеют право читать, записывать и выполнять файл pi_ser). Сменить права доступа к файлу filename можно при помощи команды

```
chmod [u g o a] [+ - =] [r w x] filename
```

Здесь 'u' означает требование смены права доступа для пользователя, 'g' - для группы, 'o' - для иных пользователей, 'a' - сразу для всех трех категорий, '+' означает добавление соответствующего права, '-' - удаление, '=' - присвоение. При работе с оболочкой Midnight Commander удобно использовать механизм 'горячих клавиш' и меню (доступ к командам chmod и chown достигается через F9 → File). При работе без оболочки полезны команды:

- `cd dir` - переход в каталог dir (если dir не указан, то переход осуществляется в домашний каталог пользователя),
- `cp file1 file2` - копирование файла file1 в file2,
- `mv file1 file2` - перемещение (изменение имени) файла с file1 на file2,
- `rm file` -удаление файла file,
- `rmdir dir` - удаление каталога dir,
- `mkdir dir` - создание каталога dir,
- `pwd` - вывод имени текущего каталога,
- `cat file`, `more file`, `less file` – полезные утилиты для просмотра содержимого файла file,
- `grep <регулярное_выражение> file` - поиск в файле file вхождений регулярного выражения,
- `who` - список пользователей, работающих в данный момент в системе

При регистрации в системе нового пользователя `user_name` создается его личный ('домашний') каталог `/home/user_name`, в котором и происходит вся работа пользователя (как сказано, каталог `/home` смонтирован под NFS). Выдача 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) 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'а/имя_задачи
```

Также можно пользоваться командами `tkill` (снятие задачи со счета, в случае `tkill *` снимается исполняющаяся задача пользователя), `mfree` (узнать 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 необходимо выполнять следующие действия:

- Обмен (двусторонний) файлами между своей (клиентской) машиной и УМ кластера.
- Управление файловой системой (создание/изменение/удаление файлов и каталогов и т.п.).
- Управление заданиями (запуск 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 файломенеджеров; удобно пользоваться вызываемым консольной командой `mc -as` 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 доступа изнутри

локальной сети и протокол с повышенной защищенностью SSH для доступа извне 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 аналог 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).

Все клиентские программы требуют указания адреса (IP или доменного) HOST-машины, далее вводятся login и password для захода собственно на Linux-управляющий узел. Дополнительно доступны базированная на WEB-технологии консоль оператора (рис.1.4) и реализующая пульт пользователя программа PULT (рис.1.6); обе системы разработаны в испытательной лаборатории проекта МВС ИПМ им. М.В.Келдыша РАН.

Исходные тексты программ могут быть подготовлены на сторонних 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 < data_in_file`). 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 только главным процессом.

Необходимое оборудование – вычислительный кластер под управлением UNIX-совместимой ОС, предустановленная поддержка MPI, система управления прохождением параллельных задач (СУППЗ) разработки ИПМ им. М.В.Келдыша РАН, рабочая консоль программиста для управления прохождением пользовательских задач.

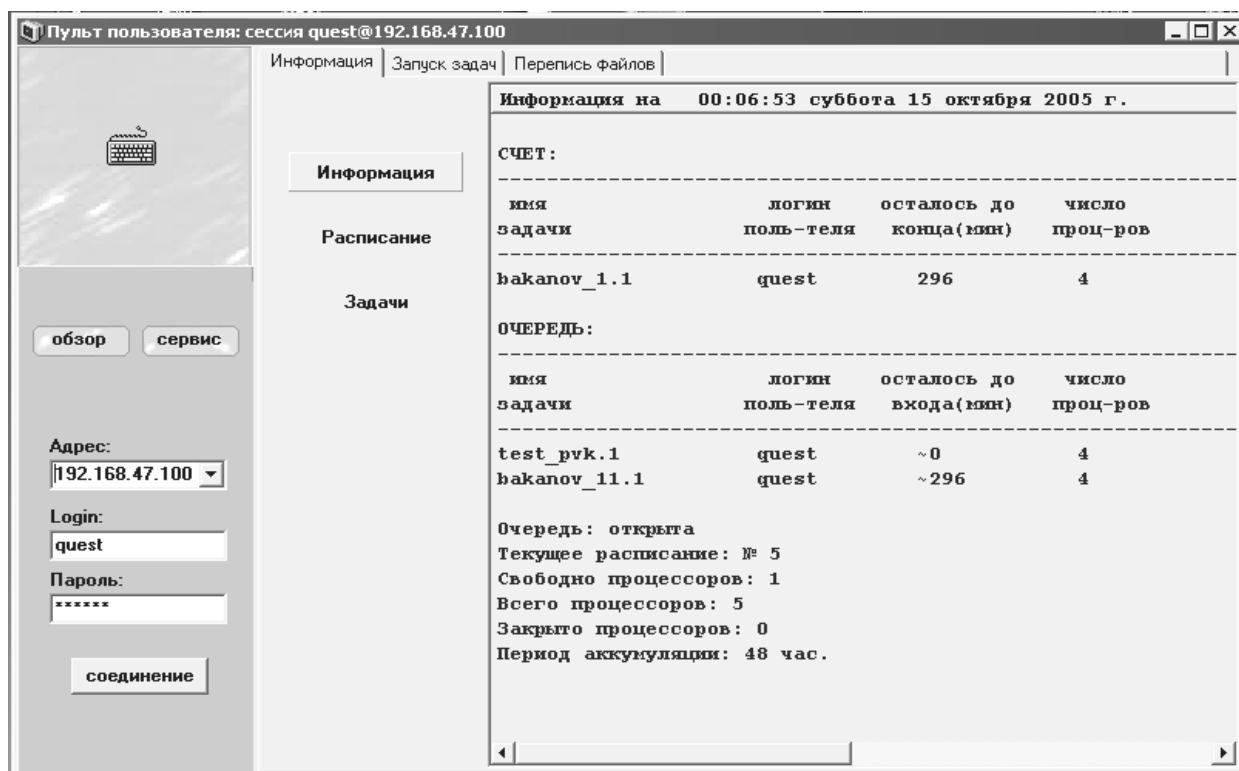


Рисунок 1.6 — Пульт пользователя вычислительного кластера (режим информации о задачах пользователя **quest**).

Порядок проведения работы :

- Студент ‘заходит’ на управляющий узел кластера с клиентской Windows-машины (пользуясь пультом пользователя или SSH-клиентом), регистрируется (login и password даются преподавателем).
- Пользователь проводит ознакомление с файловой структурой дисковой памяти управляющего узла (текущий каталог пользователя обычно /home/quest), практикуется в пересылке файлов между клиентской машиной и управляющим узлом (удобно при этом создать свой подкаталог с уникальным именем внутри /home/quest), переименовании файлов и др. операциям с файлами.
- Студент компилирует последовательное приложение (напр., mm_ser.c.), запускает его на выполнение (применяя перенаправление выдачи в файл mm_ser > mm_ser.out), анализирует **выдачу**.

Редактирование исходного файла может быть осуществлено с помощью встроенного в пульт пользователя редактора (закладка ‘Перепись файлов’, кнопка ‘Edit’), редактора другой используемой клиентской программы (в т.ч. с помощью запущенного Midnight Commander’a) или даже последовательностью копирования на клиентскую машину, редактирования средствами Windows и копирования файла обратно на управляющий узел кластера.

- Пользуясь описанными выше правилами, обучаемый компилирует параллельную программу (напр., pi_01.c), запускает ее на исполнение, анализирует содержимое файлов каталога **выдачи** (/home/guest/pi_01.1 в дан-

ном случае), при необходимости более тщательного анализа переписывает эти файлы на свою клиентскую машину.

Вопросы для самопроверки:

1. Чем отличаются архитектуры MPP и SMP? Каковы их преимущества и недостатки? Что такое NUMA-архитектура и каковы ее достоинства и недостатки?
2. Какие категории задач пользователей известны? Что такое 'режим планирования' очередей?
3. Каким образом реализуется ограничение времени выполнения задания по заданному максимальному времени счета?
4. Что такое 'сетевой интерфейс'? Какие параметры определяют его?
5. Какие сети необходимы при создании вычислительного кластера? Какие из них могут являться виртуальными? Какие из них критичны для времени выполнения параллельных программ?
6. В чем отличия администрирования вычислительного кластера от такового обычной последовательной ЭВМ?

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

Цель работы – приобретение практических знаний и навыков в компиляции и запуске простейших MPI-программ, доказательство необходимости синхронизации ветвей параллельной программы.

Теоретическая часть. 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.1 (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 (базовых) функций полезно привести прототипы (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 отправки 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* отсылку 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 predefined 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 отправки, 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 инициализировал 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.2.1).

Таблица 2.1 — 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.

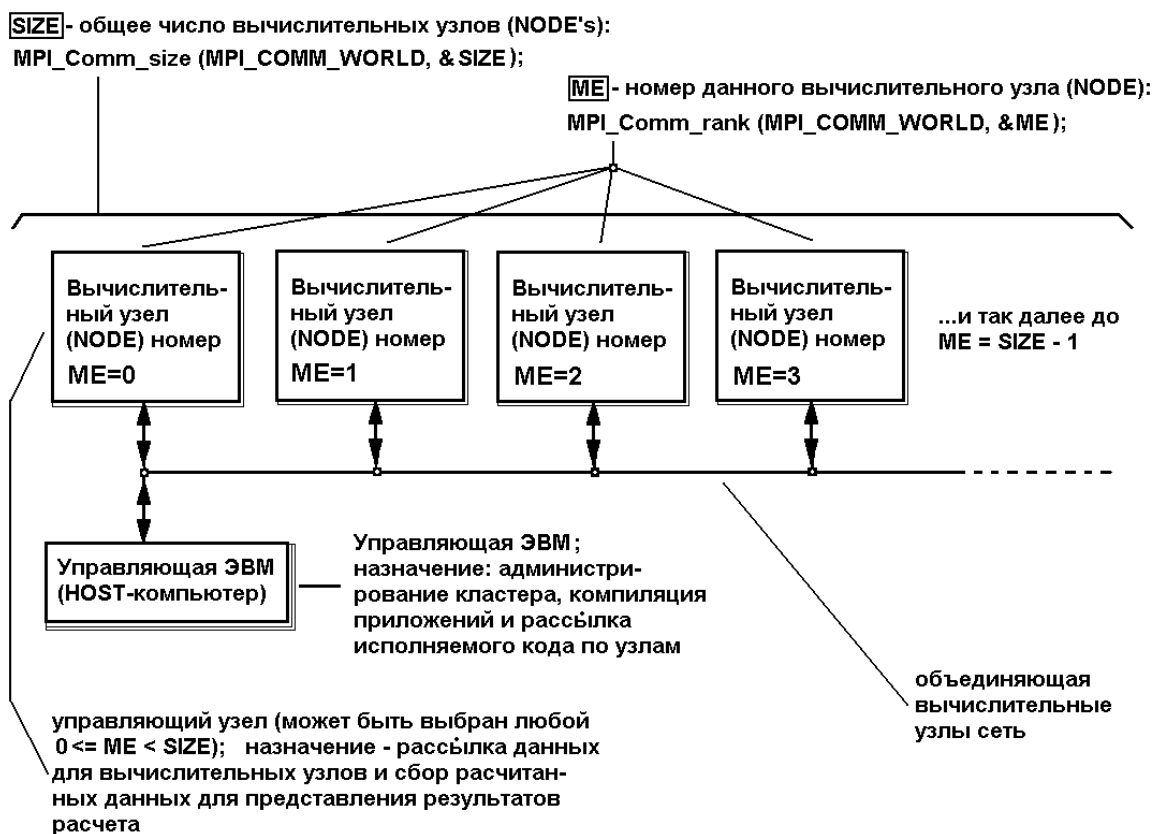


Рисунок 2.1 — Нумерация входящих в кластер вычислительных узлов (по умолчанию) и их стандартные назначения.

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.2.1). 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 адресов ветвей (например, при вычислении матриц иногда выгодно использовать картезианскую систему координат, где координаты вычислительной ветви 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 тем’ 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 отправки и приема данных мешают друг другу и обмен не может состояться. Ниже рассмотрена deadlock-ситуация при использовании для пересылок разделяемой памяти.

Вариант 1.	<u>Ветвь 1 :</u>	<u>Ветвь 2 :</u>
	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.	<u>Ветвь 1 :</u>	<u>Ветвь 2 :</u>
	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 происходит: функция MPI_Send, если на приемной стороне 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) тупиковых 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 посылку 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 коммутатор,
- порядок приема и передачи данных MPI_Sendrecv выбирает автоматически; при этом гарантировано отсутствие deadlock'а,
- MPI_Sendrecv совместима с MPI_Send и MPI_Recv

Функция MPI_Sendrecv_replace помимо общего коммутатора 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'а

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. 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

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 EXAMP_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
    // ATTENTION ! 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 выдачи 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 C-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 'тиками') процессора. Без (принудительной) синхронизации часов 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 выдачи данных (удивительно, но даже *при всей вышеприведенной* информации о правилах выполнения параллельных приложений правильный ответ нечаст!).

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 *корректным образом не завершится* (в зависимости от используемой версии MPI исполнение закончится с ошибкой или только по истечению отведенного времени).

Раскомментировав строку //for(i=0; i<all_rank; i++) и заменив (сходным 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 predefined 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 рассылки 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 повтора 1000 ÷ 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=10³
- t<число> - число повторений для относительно ‘быстрых’ операций (таких как измерение времени); умолчание t=10⁴
- R<число> - число повторений всего теста; умолчание R=1

Примерный вид выдачи 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 достоин тщательного исследования для дальнейшего квалифицированного применения.

Индивидуальные задания включает определение времени выполнения основных MPI-функций на различном оборудовании (напр., сравнить быстро-

действие высокопроизводительного кластера SCI НИВЦ МГУ и вычислительного кластера кафедры ИТ-4 МГУПИ).

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 коммутатор? 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 буферизованные 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**

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

Теоретическая часть. В вычислительных кластерах наиболее ‘узким местом’ обычно является сеть, осуществляющая связь между вычислительными узлами (фактически эта сеть является аналогом общей шины, связывающей различные узлы любой ЭВМ).

Даже весьма быстросействующая 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.1).

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) пропускной 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 об-

щей процедуры тестирования (более полный набор тестов можно получить с <http://parallel.ru/testmpi>).

Для виртуального кластера максимально достижимая реальная производительности коммуникационной сети достигается при размере сообщений более 128 ÷ 256 кбайт, при этом величина латентности может достигать 300 ÷ 400 мксек (что является платой за виртуальное совмещение сетей управления и коммуникаций единой физической Fast Ethernet-сетью).

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 PROG_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 prog_mpi 1048576
```

Образец выдачи (кластер кафедры ИТ-4 МГУПИ, 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 сводятся в табл.3.1, причем принято строить зависимость пропускной способности сети (столбец Speed таблицы, Мбайт/сек) от длины сообщения (столбец Len, кбайт) в логарифмической (по основанию 2) шкале абсцисс.

Таблица 3.1 — Данные экспериментов по определению зависимости реальной пропускной способности коммуникационной сети кластера от длины передаваемых сообщений

<i>№ №</i>	<i>Len (байт/кбайт)</i>	<i>Time (сек)</i>	<i>Rate (сек⁻¹)</i>	<i>Speed (байт*сек⁻¹)</i>
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 длинах (единицы/десятичные доли 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? Какова *количественно* цена обмена при определенной Вами величине латентности?
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?
5. Определите *экспериментально*, какое количество операций умножения (сложения, деления 'плавающих' чисел двойной точности) производит вычислительный узел Вашей МВС за промежуток времени, равный латентности? Сделайте *качественный* вывод о *рациональном* размере гранулы (зерна, блока) распараллеливания для предоставленного в Ваше распоряжение конкретного оборудования.

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

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

Теоретическая часть. 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.

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 $\frac{\pi}{4} = \arctg(1) - \arctg(0) = \int_0^1 \frac{dx}{1+x^2}$. 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 столь простой 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 C-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(MCW) управление в вызывающую программу не возвращается до тех пор, пока все процессы группы не вызовут MPI_Barrier(MCW), где MCW – 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 π определяется как $4.0 \times (\text{atanl}(1.0) - \text{atanl}(0.0))$ и сравнивается с численно вычисленным (относительная ошибка `rel.error` выводится в процентах).

Образец выдачи программы приведен ниже (запуск на процессорах кластера кафедры ИТ-4 МГУПИ):

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 π 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 $N=2,3,4,5,6,7,8,\dots$ с одинаковым числом intervals), при этом оценить точность вычисления числа π .
- 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 суммирование их в ветви 0).

Часть 2 работы. Для вычисления значения π можно использовать метод ‘стрельбы’ (рис.4.1) – вариант метод Монте-Карло. В применении к данному случаю метод заключается в генерации *равномерно распределенных* на двумерной области $[0 \leq x \leq 1, 0 \leq y \leq 1]$ точек и определении $\pi = 4 \times \frac{S_{OAC}}{S_{OABC}} \approx 4 \times \frac{\text{score}}{\text{darts}}$,

где S – площади фигур на рис.4.1, score – число попавших внутрь четверти окружности (фигура OAC) точек (условие $x^2 + y^2 \leq 1.0$, на рис.4.1 удовлетворяющие этому условию точки обозначены квадратами), darts – общее число точек (‘выстрелов’, throws). Вычисленное таким образом значение π является приближенным, в общем случае точность вычисления искомого 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).

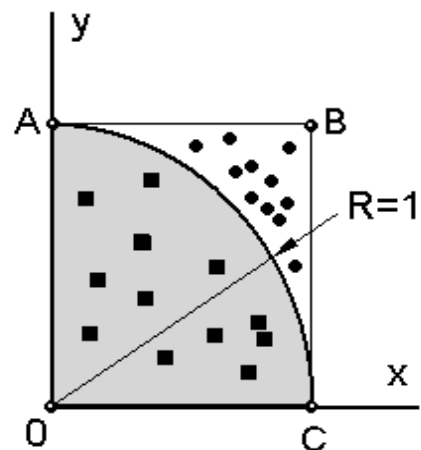


Рисунок 4.1 — Определение величины π методом ‘стрельбы’

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 π ; расчеты проводятся при начальном значении DARTS с последующим увеличением ROUNDS раз (заданы посредством `#define`).

Последовательный вариант программ вычисления π 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*(atan(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 0 as

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 π , 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*(atan(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
{
    psum = 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);
        psum = psum + pirecv;
    }
    /* Master calculates the average value of pi for this iteration */
    pi = (psum + 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
```

Выдача 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 π):

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 π являются весьма приближенными даже при большом числе ‘выстрелов’, причем не во всех случаях увеличение числа ‘выстрелов’ вызывает возрастание точности представления искомой 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 этом является буфером послылки, 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*(atan(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`, запуска на исполнение с выдачей данных в файл `pi_ser.out - pi_ser > pi_ser.out`), PI_02 и PI_03 программы вычисления значения π , ‘прогнать’ их в режиме выполнения на одном и нескольких процессорах, проанализировать выдачу.

Индивидуальные задания включает определение числа интервалов интегрирования, необходимое для представления числа π 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 π 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 $N=3,4,5,6,7,8\dots$ с одинаковым числом ‘выстрелов’).

Вопросы для самопроверки:

1. Привести примеры априори хорошо распараллеливаемых 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 π 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

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

Теоретическая часть. 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 (предполагая, что матрица является плотнозаполненной – т.е. число нулевых элементов мало; 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 $[C]=[A] \times [B]$ значение элементов результирующей матрицы вычисляется как (стандартный способ):

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

где $i=1 \div NRA$ – строки матриц $[A]$ и $[C]$,

$j=1 \div NCB$ – столбцы матриц $[B]$ и $[C]$,

$k=1 \div NCA$ – столбцы матрицы $[A]$ и строки матрицы $[B]$.

Хотя известны более эффективные алгоритмы (напр., *умножение матриц по Винограду* и *рекурсивный алгоритм Штрассена умножения квадратных матриц*), в дальнейшем используется стандартный (требующий N^3 операций умножения и $N^3 - N^2$ сложений, где N – характерный размер матрицы).

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 momemnts

/* 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]); в профессиональных пакетах линейной алгебры (ScaLAPACK, AZTEC) распределению содержимого обрабатываемых мат-

риц по вычислительным узлам уделяется серьезное внимание. В данной работе задача распределения содержимого матриц по ВУ не решается, *распараллеливаются только вычисления.*

Лежащий на поверхности способ распараллеливания вычислений заключается в перемножении каждым процессом значений a_{ik} и b_{kj} и суммировании главным процессом этих частичных сумм для получения c_{ij} . Однако этот подход наименее рационален по производительности, ибо каждое умножение (а их число пропорционально N^3) сопровождается минимум 3 обменами, причем длительность каждого намного 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 в каждом процессе вычислять произведение i -той строки [A] и j -того столбца [B], в результате сразу получаем готовое значение c_{ij} ; и при этом число обменов (пропорциональное N^2) все еще излишне высоко. 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.

Еще более эффективно пересылать каждому процессу *серию строк (ленту) матрицы [A] и серию столбцов [B]*; этим реализуется крупнозернистый (*coarse-grained*) параллелизм (рис.5.1), а метод умножения матриц именуется *ленточным*. Каждый процесс при этом вычисляет (определяемый условием $c_{ij}, i \in [i1...i1], i \in [j1...j2]$) прямоугольный блок матрицы [C]. При программировании на C для пересылки процессорам вертикальной ленты [B] необходимо использовать временный (рабочий) массив, при программировании на Fortran'e - то же для горизонтальной ленты [A]. Заметим, что нахождение в ОП процесса горизонтальной ленты [A] дает возможность вычислить не только прямоугольный блок $c_{ij}, i \in [i1...i1], i \in [j1...j2]$, но и всю вертикальную ленту $c_{ij}, i \in [1...NRA], i \in [j1...j2]$, см. рис.5.1; для этого необходимо дополнительно пересылать только новые горизонтальные ленты [A]. При программировании на C для пересылки процессорам вертикальной ленты [B] придется использовать временный (рабочий) массив, при программировании на Fortran'e - то же для горизонтальной ленты [A].

Т.к. обычно число столбцов матрицы [B] много меньше числа строк [A] (часто $NCB=1$ и матрица [B] фактически является вектором), в приведенной ниже программе MM_MPI_2.C матрица [B] пересылается всем процессам целиком.

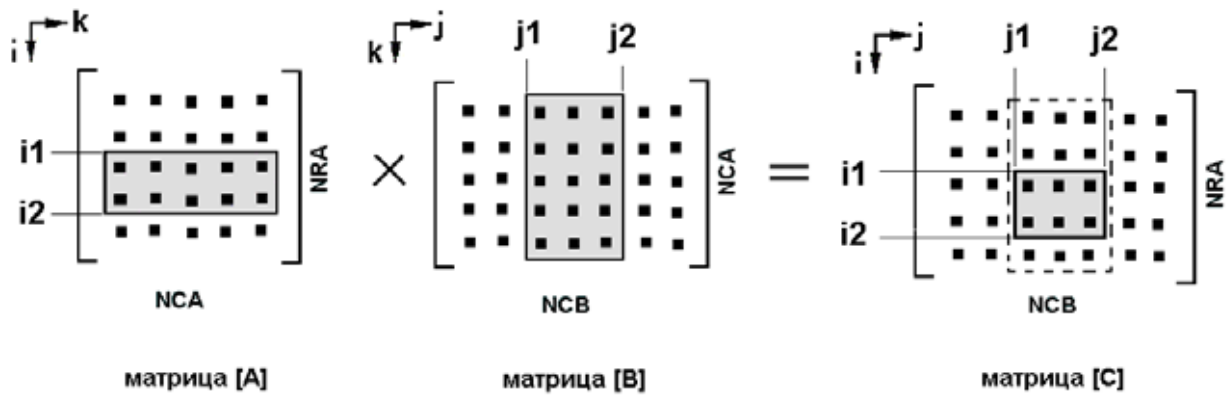


Рисунок 5.1 — Ленточная схема умножения матриц (пересылаемые ленты и вычисленный блок результирующей матрицы выделены серым фоном).

```
// source code of MM_MPI_2.C program
// Ros Leibensperger / Blaise Barney. Converted to MPI: George L.
#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 moment
    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_2.C program
```

Строки матрицы [A] распределяются между numworkers процессов по rows штук – каждому процессу $dest=1 \div numworkers$ 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 нацело и rows=averow+1 в противном 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 целиком.

После получения процессами данных выполняется умножение посредством внешнего цикла по $k=1 \div NCA$ (т.е. по столбцам [A] и строкам [B] и внутреннего цикла по $i=1 \div rows$ (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 выдачи программы при NRA=1000 и числе вычислительных узлов 12 (видно, что в случае невозможности деления NRA на numworkers нацело 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
```

Дальнейшим развитием (в сторону минимизации числа пересылок данных) ленточного алгоритма умножения матриц является алгоритм Фокса (Geoffrey Fox, et al., *Solving Problems on Concurrent Processors*, Englewood Cliffs, NJ, Prentice-Hall, 1998) и алгоритм Кэннона (в этом случае процессам пересылаются не ленты, а *прямоугольные блоки* исходных матриц; усложнением при этом является необходимость дополнительного межпроцессного обмена, ибо на основе прямоугольных блоков [A] и [B] невозможно вычислить такой же блок [C] – необходим циклический сдвиг фронта вычислений).

Алгоритм Фокса иногда именуется *клеточным* (ибо основан на распределении блоков матриц по ОП процессов по *принципу шахматной доски*), этот алгоритм описан также в работах [3,7]; ограничением его является (непрерывное) 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). Алгоритм Кэннона отличается более разумным распределением блоков исходных матриц по процессорам, при этом упрощается схема обмена данными между последними.

Например, при умножении матриц размером 6×6 по алгоритму Фокса при 9 процессах каждому из них передаются подматрицы размером $N' = N/Q = N/\sqrt{M} = 2$ следующим образом (процессоры объединены квадратной сеткой размерности Q , нумерация везде начинается с 0):

<u>Процесс 0</u> $A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$	<u>Процесс 1</u> $A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$	<u>Процесс 2</u> $A_{02} = \begin{pmatrix} a_{04} & a_{05} \\ a_{14} & a_{15} \end{pmatrix}$
<u>Процесс 3</u> $A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$	<u>Процесс 4</u> $A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$	<u>Процесс 5</u> $A_{12} = \begin{pmatrix} a_{24} & a_{25} \\ a_{34} & a_{35} \end{pmatrix}$
<u>Процесс 6</u> $A_{20} = \begin{pmatrix} a_{40} & a_{41} \\ a_{50} & a_{51} \end{pmatrix}$	<u>Процесс 7</u> $A_{21} = \begin{pmatrix} a_{42} & a_{43} \\ a_{52} & a_{53} \end{pmatrix}$	<u>Процесс 8</u> $A_{22} = \begin{pmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{pmatrix}$

В общем случае действия перемножения матриц иллюстрируются следующим алгоритмом [3]:

```
for(step=0; step<Q; step++)
{
    1. Выбрать подматрицу матрицы [A] в каждой строке процессов (в  $r$ -той строке находится подматрица  $A_{ru}$ , где  $u=(r+step) \bmod Q$ ).
    2. Для процессов каждой строки переслать всем иным процессом той же строки выбранную подматрицу.
    3. Каждый процесс умножает полученную подматрицу матрицы [A] на находящуюся в ОП процесса подматрицу матрицы [B].
    4. Для каждого процесса переслать подматрицу матрицы [B] процессу, находящемуся в решетке выше (из первой строки пересылка идет в последнюю строку).
}
```

Вариант умножения матриц в соответствие с алгоритмом Фокса `mm_fox.c` может быть получен у преподавателя (исходный текст не приводится).

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 обязательно 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).

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 $N=2,3,4,5\dots$).
- Выявить зависимость быстродействия (величина, обратная времени выполнения алгоритма) программы MM_MPI_2 от числа процессоров $N=3,4,5\dots$ (производительность программы при числе процессоров 2 принять единичной).

Дополнительные задания (самостоятельная работа студентов):

- Модифицировать программу MM_MPI_2 таким образом, чтобы матрицы [A], [B] и [C] не присутствовали в ОП каждого процесса, а только на главном.
- Реализовать программу умножения матриц с выполнением каждым процессором:
 - А) Умножений элементов $a_{ik} \times b_{kj}$ с целью получения частичной суммы c_{ij} и последующим суммирование частичных сумм главным процессом для получения c_{ij} (программа MM_MPI_0.C).
 - Б) Умножений строки матрицы [A] на столбец [B] с последующим суммированием на каждом процессоре для получения конечного значения c_{ij} (программа MM_MPI_1.C).

Какая особенность алгоритма выполнения действий на процессорах требуется в этом случае? Сравнить быстродействие этих программ с MM_MPI_2.C при различном числе процессоров.

- Сравнить эффективность алгоритма Фокса с программой MM_MPI_2.C (определить время выполнения при различной размерности умножаемых матриц и числе процессоров).

Вопросы для самопроверки:

1. В каком случае время умножения матриц будет больше – в случае выполнения MM_SER или MM_MPI_0 при числе процессоров $N=2$ (включая управляющую ЭВМ)?
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. Оценить (в единицах умножения элементов двумерного массива ‘плавающих’ чисел двойной точности с последующим сложением частичных сумм **онных**) размер *гранулы* (зерна, блока) параллелизма для случая параллельного умножения матриц ленточным способом. В каком случае *коэффициент ускорения вычислений* больше – при *большей* или *меньшей* размерности умножаемых матриц (предположение *проверить экспериментально*)?
5. Предложить и обосновать более эффективную (относительно вышерассмотренных) стратегию распределения больших матриц по вычислительным узлам (обязательно учесть некратность числа строк и столбцов матрицы числу процессоров).

6 Лабораторная работа 6. Автоматизация разработки параллельных МРІ-программ с использованием проблемно-ориентированного языка НОРМА

Цель работы – приобрести начальные навыки в разработке параллельных программ, предназначенных для решения задач математической физики разностными методами с использованием системы автоматизации создания параллельных программ ‘Интерактивная НОРМА’ и непроцедурного языка программирования НОРМА.

Теоретическая часть. Как сказано выше, программирование в технологии МРІ является достаточно низкоуровневым и поэтому требует высокой квалификации разработчика и значительных затрат (в т.ч. временных) при создании программного продукта. Имеющиеся системы автоматизации распараллеливания недостаточно эффективны (при высокой стоимости и узости 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 (*Непроцедурное Описание Разностных Моделей Алгоритмов* или *НОРМАльный уровень общения прикладного математика с компьютером*, <http://www.keldysh.ru/pages/norma>).

Общая схема разработки программ для задач математической физики включает следующие этапы:

- Постановка задачи. Выходом этого этапа является обычно система дифференциальных уравнений, описывающих задачу.
- Выбор пространственно-временной сетки и дискретизация уравнений с помощью одного из разностных методов.
- Выбор метода решения дискретных уравнений. Результат – формулы (соотношения), описывающие необходимые вычисления в узлах сетки.
- Полученные формулы оформляются в виде программы на некотором языке высокого уровня, обеспечивающем решение задачи на вычислительной машине.

Главная идея, лежащая в основе языка Норма заключается в том, что полученное на третьем этапе описание решения задачи практически непосредственно используется для ввода его в вычислительную систему и проведения счета.

Язык 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 мыслит 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) языковых стандартов параллельного или последовательного программирования (в настоящее время существует возможность получения Fortran'MPI, Fortran'DVM, Fortran'77 или соответствующих C-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., [8]). Исходный текст на НОРМА в высшей степени близок 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. Более подробно описание языка см. на <http://pilger.mgapi.edu/methods/1441/norma.zip>, <http://www.keldysh.ru/norma> и в [8].

Важными конструкциями НОРМА являются описание разделов программы, областей и коллективных (в дальнейшем распараллеливаемых) операций. Ниже приведен текст программы, содержащий основной раздел (MAIN PART), двух одномерных областей O_i и O_j , двумерной области O_A (являющейся объединением областей O_i и O_j), определения отображаемой на область O_A переменной a , двух констант $iMAX$ и $jMAX$ и коллективной (выполняющейся над всеми значениями $a[i][j]$) операции присвоения $a[i][j]$ значения $i \times j$ (индексы нумеруются начиная с 1):

```
MAIN PART Mmatrix. ! Основной раздел)
BEGIN
  Oi: (i=1..iMAX. ! определение одномерных областей
  Oj: (j=1..jMAX).
  OA: (Oi; Oj). ! определение двумерной области
  VARIABLE a DEFINED ON OA DOUBLE. ! определение переменной на области OA
  ! здесь может находиться описание вывода распределенных на областях
  ! переменных, например, OUTPUT a(FILE='Mmatrix.out') ON OC.
  DOMAIN PARAMETERS iMAX=500. ! определение констант
  DOMAIN PARAMETERS jMAX=1000.
  FOR OA ASSUME a=i×j. ! коллективная операция для задания значений a[i][j]
  ! здесь может находиться заголовок (прототип) процедуры, совершающей
  ! некоторые действия над переменными, описанными на областях, например
  ! COMPUTE Pmatrix(a ON OA, b ON OB RESULT c ON OC).
END PART. ! конец основного раздела
```

Описанные в основном разделе операции не распараллеливаются и обычно являются подготовительными для последующих вычислений. Описание коллективной операции может выглядеть так (раздел PART содержит процедуру Pmatrix):

```
PART Pmatrix. ! начало раздела Pmatrix
  a,b RESULT c ! a,b – входные параметры, c – выходной
! тело процедуры Pmatrix
BEGIN
  Oi: (i=1..iMAX).
  Oj: (j=1..jMAX).
  Ok: (k=1..kMAX).
  OA: (Oi; Ok).
  OB: (Ok; Oj).
  OC: (Oi; Oj).
  VARIABLE a DEFINED ON OA DOUBLE.
  VARIABLE b DEFINED ON OB DOUBLE.
  VARIABLE c DEFINED ON OC DOUBLE.
  DOMAIN PARAMETERS iMAX=500.
  DOMAIN PARAMETERS jMAX=1000.
  DOMAIN PARAMETERS kMAX=1000.
  DISTRIBUTION INDEX i=1..12, j=1. ! описание топологии многопроцессорной системы
  FOR OC ASSUME c=SUM((Ok)a[i,k]×b[k,j]). ! коллективная операция вычисления c[i][j]
END PART. ! конец раздела Pmatrix
```

В начале раздела Pmatrix описаны переменные a,b как входные параметры и переменная c как выходной (ключевое слово RESULT) параметр. Умножаемые матрицы имеют размеры $a[i=1..iMAX][k=1..kMAX]$ и $b[k=1..kMAX][j=1..jMAX]$, результирующая матрица – $c[i=1..iMAX][j=1..jMAX]$.

Конструкция FOR OC ASSUME c=SUM((Ok)a[i,k]×b[k,j]). Описывает коллективную операцию вычисления произведения матриц по формуле

$$c_{ij} = \sum_{k=1}^{k=kMAX} a_{ik} \times b_{kj} \text{ для каждого элемента матрицы } c. \text{ Распределение данных и}$$

вычислений на двумерную сетку процессоров в данном случае описано посредством DISTRIBUTION INDEX i=1..12, j=1. (инструкция DISTRIBUTION INDEX бессмысленна в основном разделе); при этом исходные матрицы распределяются по процессорам путем разбиения их на горизонтальные ленты (в соответствие с индексом i). Критерий выбора индексных направлений (в текущей версии НОРМА их допускается два) и их сочетаний (если имеется возможность выбора из нескольких вариантов) в конструкции DISTRIBUTION INDEX с целью синтеза наиболее производительных программ описан, напр., в <http://pilger.mgapi.edu/metods/1441/norma.zip>. С целью обеспечения итераций в НОРМА имеется специальная инструкция, позволяющая организовать итерационный цикл с проверкой выхода на заданную точность.

Однако для создания текста НОРМА-программы все же требуется программист, знакомый с правилами языка и формально ‘набивающий’ исход-

ный 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 МГУПИ разработана система 'Интерактивная НОРМА', позволяющая создавать параллельные программы практически без написания исходных текстов на языке программирования.

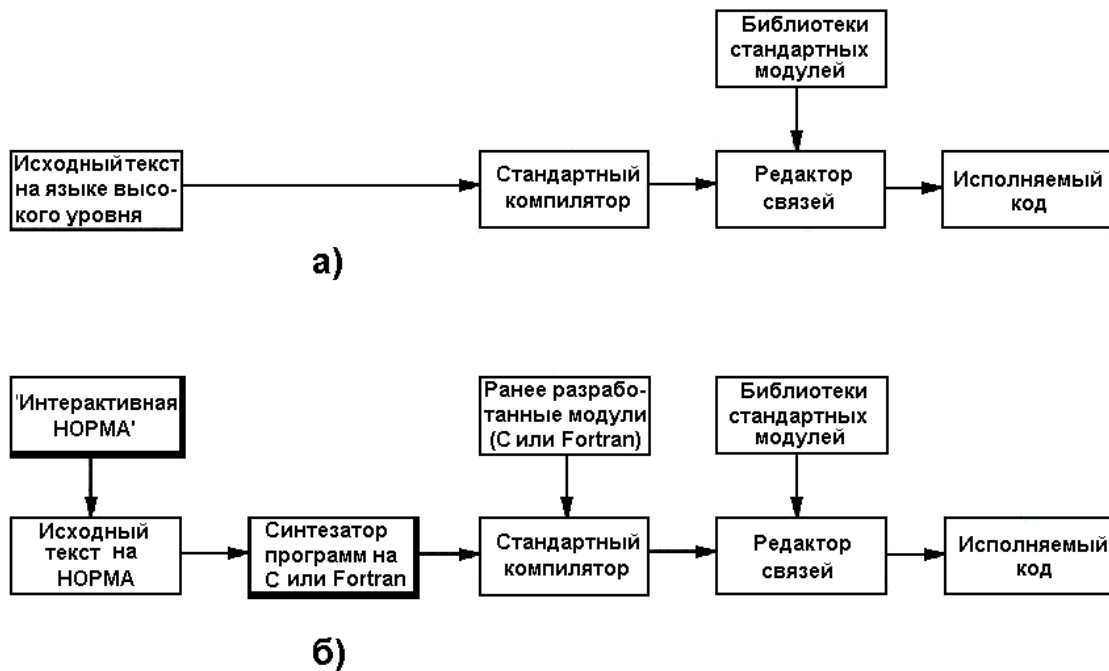


Рисунок 6.1 — Этапы создания исполняемых программ: а) – классический подход, б) – использование НОРМА-программирования совместно с оболочкой 'Интерактивная НОРМА'

Как видно из рис.6.1, при создании параллельных программ с использованием системы 'Интерактивная НОРМА' используется дополнительная программа–преобразователь (синтезатор) исходного кода НОРМА-программы в Fortran или С-программу). Система 'Интерактивная НОРМА' выполнена в технологии WEB-сайта <http://norma.deniz.ru> с применением PHP-4, нижеприведенные копии экранов получены в режиме offline-функционирования при поддержке комплексом программ 'Денвер-2' (<http://web.dklab.ru>, web@dklab.ru).

При формировании НОРМА-программы умножения матриц с помощью системы 'Интерактивная НОРМА' принято соответствие матрицам-сомножителям a, b и матрице произведения с двумерным областям OA, OB и OC соответственно; инициализация матриц проводится по формулам $a_{ik}=(i-1)+(k-1)$ и $b_{kj}=(k-1)\times(j-1)$. На рис.6.2 показано окно определения области в системе 'Интерактивная НОРМА' (следует определить три одномерные области

вещественных чисел двойной точности O_i, O_j, O_k с индексами i, j, k диапазоном от $1 \div 1000$).

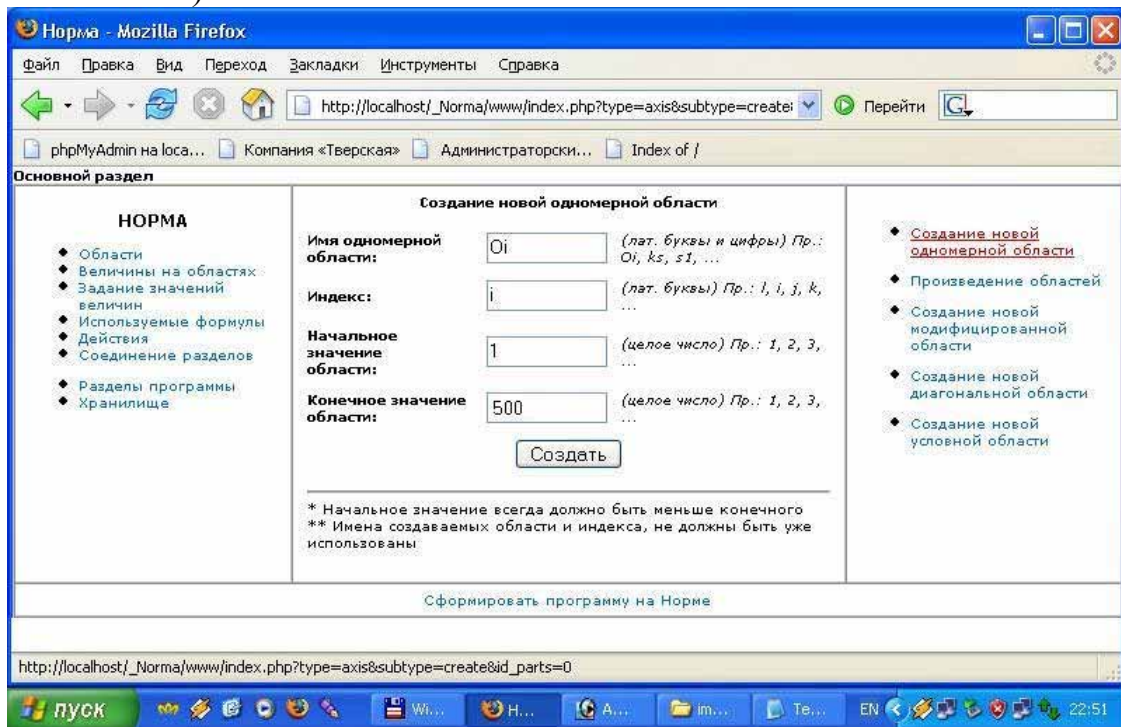


Рисунок 6.2 — Окно создания одномерной области O_i размером $1 \div 1000$ с индексом i в системе 'Интерактивная НОРМА'

Далее определяются три области O_A, O_B и O_C как объединение (произведение) ранее определенных областей O_i, O_k, O_k, O_j и O_i, O_j соответственно (рис.6.3).

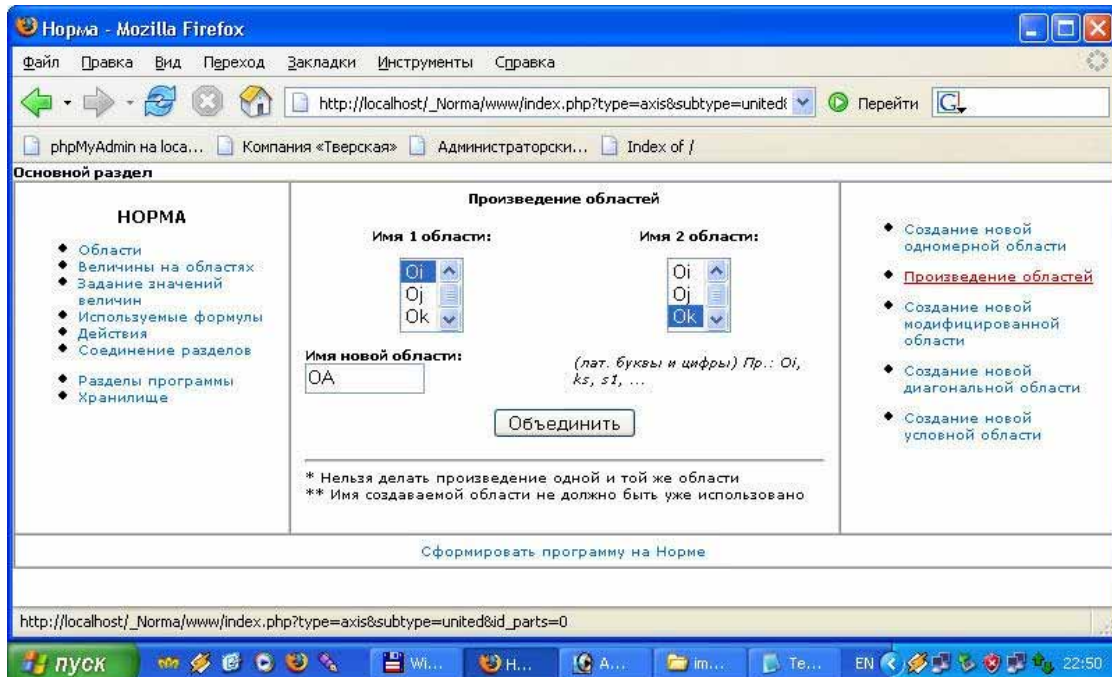


Рисунок 6.3 — Окно объединения (произведения) одномерных областей O_i и O_k в двумерную область O_A в системе 'Интерактивная НОРМА'

На следующем этапе задаются *величины на областях* – величина *a* задается на области OA (рис.6.4), *b* – на OB и *c* – на OC.

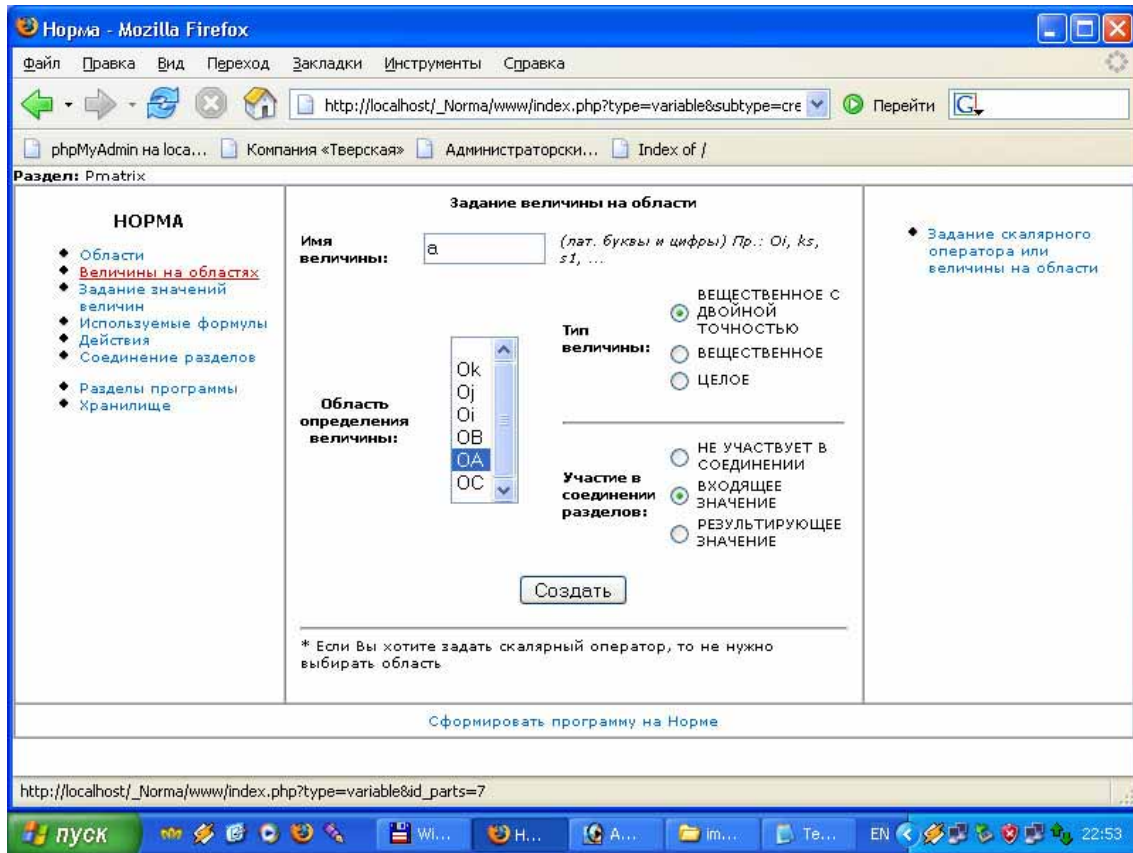


Рисунок 6.4 — Окно задания величины *a* на области **OA** в системе 'Интерактивная НОРМА'

Задание нового раздела состоит из определения его имени (окно рис.6.5), указания, к какому из существующих разделов присоединяется новый (рис.6.6) и непосредственное объединение разделов (рис.6.7); указанные рисунки иллюстрируют последовательность действий по присоединению раздела Pmatrix к основному разделу).

Для определения действия (*коллективной операции*) $c = \text{SUM}((Ok)a[i,k] \times b[k,j])$ (эквивалент операции перемножения матриц $c_{ij} = \sum_{k=1}^{k=kMAX} a_{ik} \times b_{kj}$) служит окно рис.6.8, причем раздел может содержать любое количество коллективных операций (в точности таким же образом созданы действия по инициализации величин *a* и *b*).

Последним этапом является определение решетки процессоров для выполнения параллельной программы (окно рис.6.9), в дальнейшем программу необходимо запускать на определенном (равном произведению числа процессоров по обоим направлениям плюс единица) числе процессоров.

Синтез НОРМА-программы выполняется при щелчке мышью по строке 'Сформировать программу на НОРМЕ'.

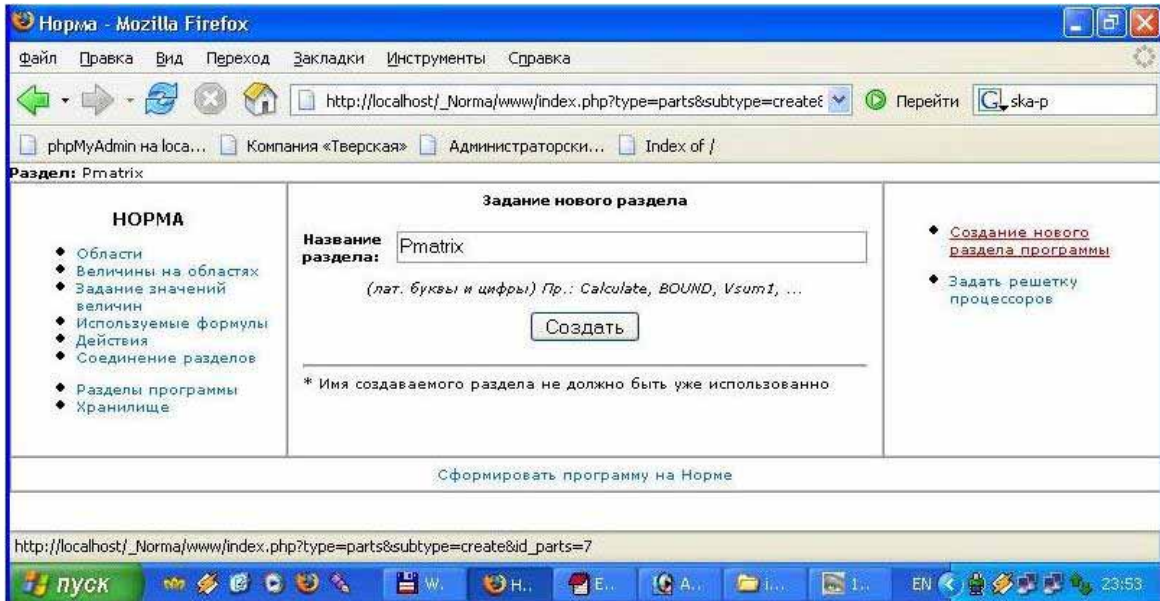


Рисунок 6.5 — Окно задания наименования нового раздела **Pmatrix** в системе 'Интерактивная НОРМА'

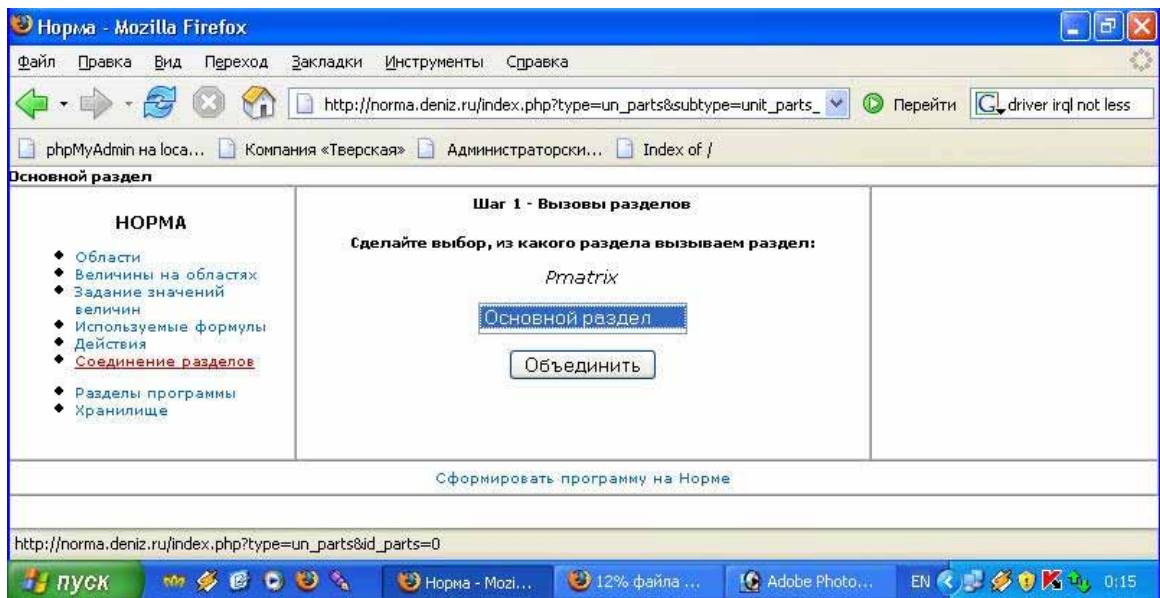


Рисунок 6.6 — Окно объединения разделов в системе 'Интерактивная НОРМА'

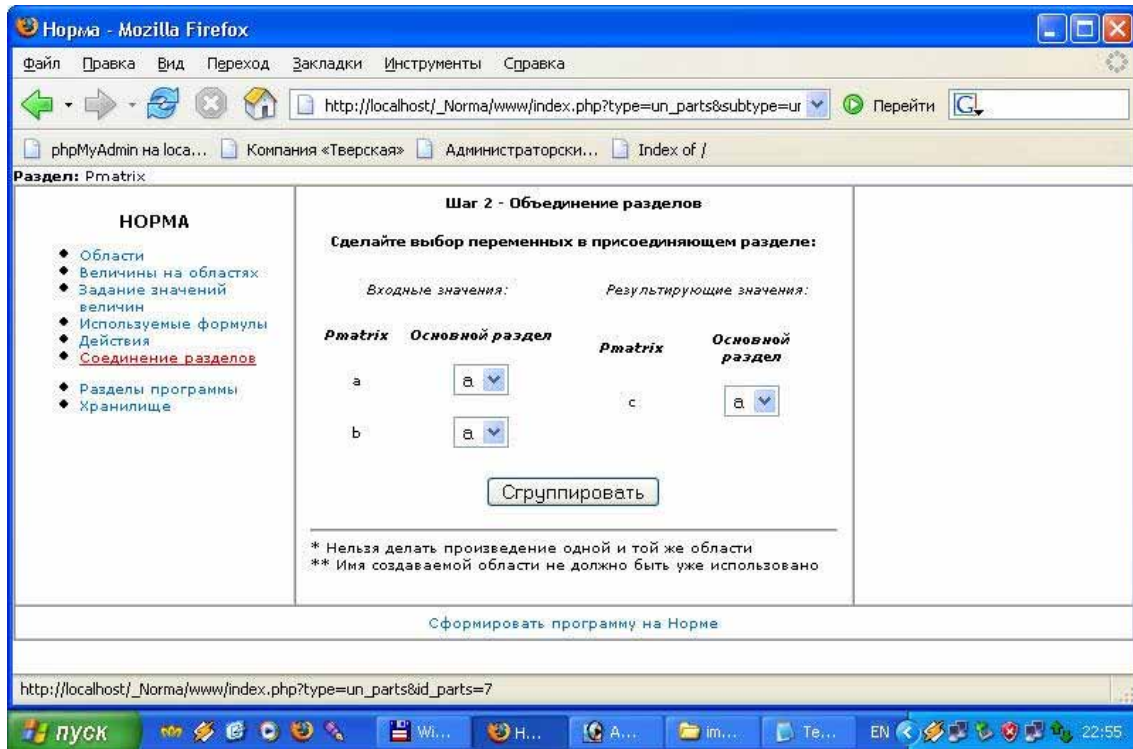


Рисунок 6.7 — Окно выбора параметров объединения в присоединенном разделе в системе 'Интерактивная НОРМА'

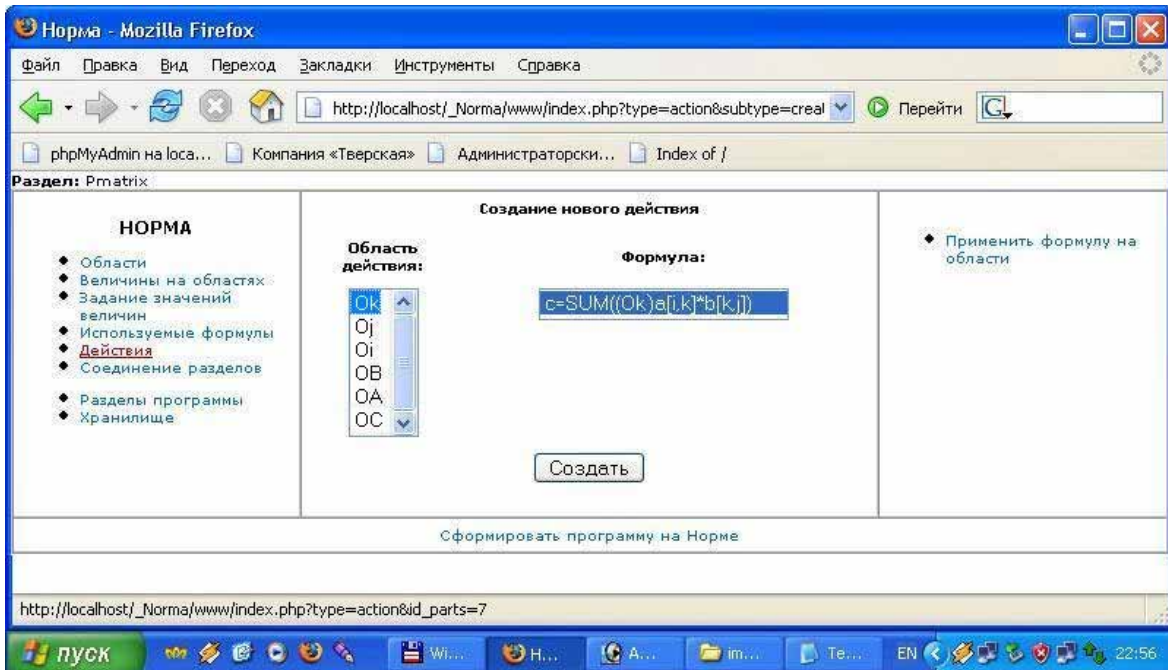


Рисунок 6.8 — Окно создания новой коллективной операции $c = \text{SUM}((\text{Ok})a[i,k] \times b[k,j])$ в системе 'Интерактивная НОРМА'

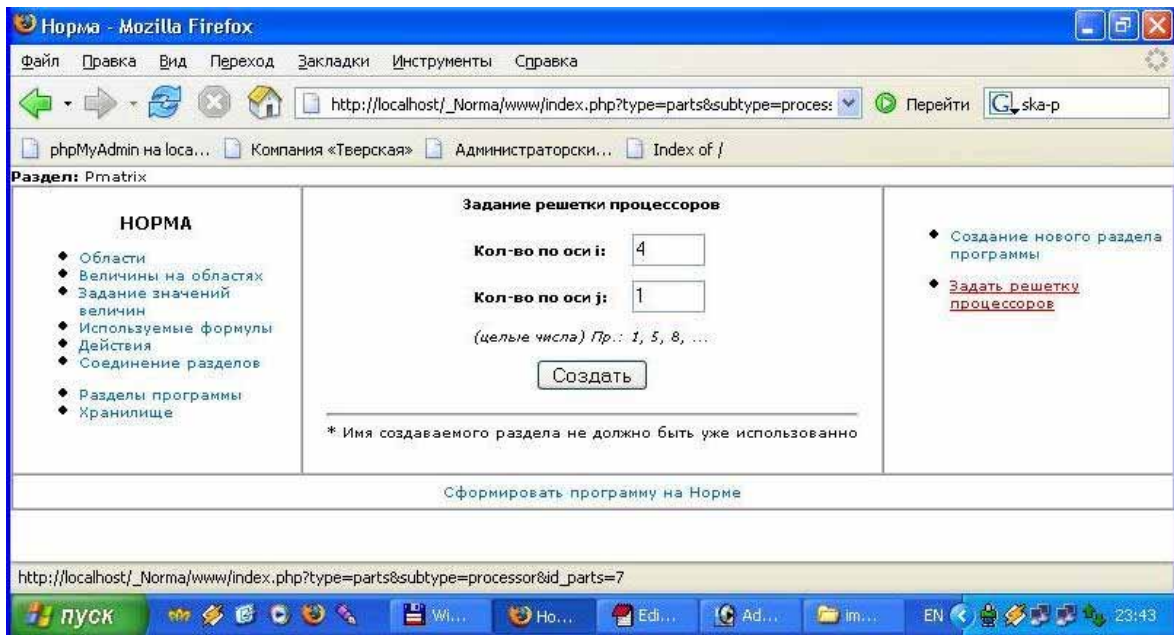


Рисунок 6.9 — Окно задания решетки процессоров для выполнения параллельной программы

Конечный результат работы синтезатора (программа на языке НОРМА) имеет такой вид (комментарии выполнены вручную):

MAIN PART Listing.

! Программа умножения матриц (основной раздел)
BEGIN

O_i: (i=1..500). ! определение одномерных областей

O_j: (j=1..1000).

O_k: (k=1..1000).

OA: (O_i; O_k). ! определение двумерных областей

OB: (O_k; O_j).

OC: (O_i; O_j).

VARIABLE a DEFINED ON OA DOUBLE. ! определение переменных на областях

VARIABLE b DEFINED ON OB DOUBLE.

VARIABLE c DEFINED ON OC DOUBLE.

OUTPUT c(FILE='Listing.out') ON OC. ! операция вывода результирующей матрицы [C]

FOR OA ASSUME a=(i-1)+(k-1). ! коллективная операция для задания

FOR OB ASSUME b=(k-1)×(j-1). ! начальных значений элементам a[i][k] и b[k][j] матриц

COMPUTE Pmatrix(a ON OA, b ON OB RESULT c ON OC). ! вызов раздела Pmatrix

END PART. ! конец основного раздела

PART Pmatrix. ! начало раздела Pmatrix

a,b RESULT c ! a,b – входные параметры, c – выходной

! тело процедуры Pmatrix

BEGIN

O_i: (i=1..500).

O_j: (j=1..1000).

O_k: (k=1..1000).

OA: (O_i; O_k).

OB: (O_k; O_j).

OC: (O_i; O_j).

VARIABLE a DEFINED ON OA DOUBLE.
 VARIABLE b DEFINED ON OB DOUBLE.
 VARIABLE c DEFINED ON OC DOUBLE.

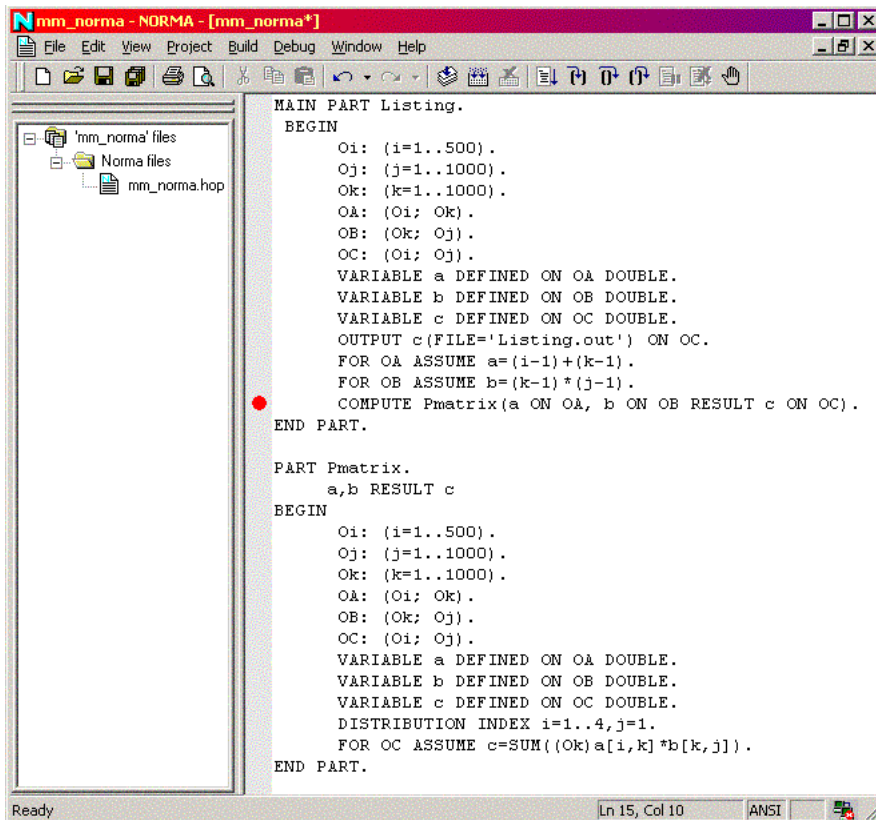
DISTRIBUTION INDEX i=1..4, j=1. ! описание топологии многопроцессорной системы

FOR OC ASSUME c=SUM((Ok)a[i,k]×b[k,j]). ! коллективная операция $C_{ij} = \sum_{k=1}^{k=kMAX} a_{ik} \times b_{kj}$

END PART. ! конец раздела Pmatrix

В синтезированном Fortran или C-коде по этой НОРМА-программе исходные массивы распределяются по процессорам в виде горизонтальных лент (по первому индексу), в случае DISTRIBUTION INDEX k=1..4, i=1. распределение происходит в виде вертикальных лент (по второму индексу); для доказательства этого полезно проанализировать синтезированные Fortran/C – тексты.

Окно синтезатора (программы, генерирующей исходные тексты програм-



мы на языках Fortran/C на основе НОРМА-текста) дано на рис.6.10 (программа синтезатора norma.exe функционирует в среде Windows). Синтез программ инициируется выбором соответствующих вариантов в выпадающем меню Build, выбор формата выходной программы (варианты языков Fortran или C) доступен в окне Project → Settings... → General, каталоги расположения отладочной (Debug) и рабочей (Release) версий синтезированной

Рисунок 6.10 — Окно синтезатора Fortran/C – программ по исходному тексту на языке НОРМА.

программы определяются в окне Project → Settings... → Local Directories. Синтезатор поддерживает также режим отладки (вариант Debug главного меню) при наличии соответствующей поддержки.

Следует удостовериться, что в диалоговом окне File → Options... корректно указаны пути к файлам компилятора и конфигулятора (файлам normarc.exe и normasnf.exe соответственно, обычно находящимся в едином с norma.exe каталоге). При появлении сообщения вида --- (38) size of a 2000000 bytes grater than size of message buffer 1024000 bytes следует увеличить размер рабочего

буфера синтезатора до нужного значения (Project→Settings→Compiler Options→Bufsize (Kb)).

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

Порядок проведения работы:

- Студент с помощью системы ‘Интерактивная НОРМА’ подготавливает программу умножения матриц `mm_norma` на языке НОРМА согласно описанию выше.
- Исходный текст программы на НОРМА переносится в синтезатор НОРМА, на его основе синтезируется MPI-программа на Fortran’е (*рекомендуется*) или C (выбираются варианты Project→Settings...→General→Fortran MPI или ...→C MPI соответственно).
- Студент с помощью программы PULT (рис.1.3,1.4) заходит на кластер (вводя `192.162.47.100/quest/пароль_пользователя_quest`) и переносит синтезированные файлы `mm_norma.f` или `mm_norma.c` на вычислительный кластер (закладка Перепись файлов правого окна программы PULT); в качестве рабочего *рекомендуется* использовать каталог `/home/quest/mm_norma` (напомним, что пользователю `quest` разрешено работать только внутри каталога `/home/quest`).
- Fortran или C – файлы компилируются в исполнимое приложение посредством ввода команд (для Fortran’а или C соответственно):

```
mpif77 -o program mm_norma.f
```

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

Для ввода команд компиляции удобно использовать закладку **Запуск задач** с выбором радиокнопки **Выполнение произвольной команды** программы PULT, команды вводятся в строке **Команды**, соответствующий вывод (`stdout`, в данном случае протокол компиляции) появляется на Экране выдачи сообщений в нижней части окна.

- Исполняемые файлы запускаются на счет (число процессоров должно равняться произведению указанных в инструкции DISTRIBUTION INDEX параметров плюс единица); команда вводится аналогично предыдущему случаю, за прохождением задачи на кластере удобно следить с помощью программы PULT (закладка **Информация**, вариант **Информация** или **Задачи**):

```
mpirun -np 5 mm_norma
```

- Студент по заданию преподавателя проводит анализ полученных данных (для этого файл выдачи Listing.out переписывается на клиентскую машину – ранее описанная закладка Перепись файлов программы PULT; для файлов размером менее 64 kb возможен просмотр без переписи – закладка Перепись файлов, кнопка edit).

Индивидуальные задания для студентов:

- Реализовать счетчик времени выполнения операции умножения матриц. Для этого в C-программе область вызова функции Pmatrix(a,b,c); следует окаймить вызовами MPI_WTIME() следующим образом (переменную t включить в описания DOUBLE-переменных):

```
t=MPI_Wtime();
Pmatrix(a,b,c); // этот оператор уже существует
t=MPI_Wtime() - t;
file8=fopen("Listing.out", "wt"); // этот оператор уже существует
fprintf(file8, "Time=%10.3f sec\n", t);
```

В Fortran-программе соответствующий текст должен выглядеть так (необходимо дополнить идентификатором t описание переменных типа DOUBLE PRECISION):

```
t=PMPI_WTIME()
CALL Pmatrix(a,b,c)
t=PMPI_WTIME() - t
OPEN(UNIT=8, FILE='Listing.out')
write(8, 113) t
113 format(f10.3)
```

- Выполнить замеры времени выполнения операции произведения матриц размерностью $10^3 \times 10^3$ чисел двойной точности на 2,3,4...10 процессорах; полученные данные (первая строка файла Listing.out) свести в таблицу и представить графически, сделать выводы о характере роста производительности с увеличением числа процессоров.
- Выполнить замеры времени выполнения операции произведения матриц размерностью $N \times N$ чисел двойной точности ($N=100, 300, 500, 700, 1000$) на фиксированном числе процессоров (обычно $5 \div 10$); полученные данные свести в таблицу и представить графически, сделать выводы о характере снижения производительности с увеличением размерности умножаемых матриц.
- Обдумать варианты комбинации индексов в инструкции DISTRIBUTION INDEX, обеспечивающие максимум производительности (при линейной топологии возможные варианты – ij, ik, jk плюс случаи множественности измерений по i,j,k при единичности измерения по другому параметру), составить план эксперимента (общее число процессоров

должно во всех случаях быть одинаковым) и проделать соответствующие опыты; полученные данные свести в таблицу, сделать выводы об оптимальном (обеспечивающем максимум производительности) сочетании индексов для Fortran или С-программ.

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 языка НОРМА эффективен принцип разработки программ с использованием интерактивной оболочки? Чем отличается подобная оболочка от известных систем визуального программирования (например, Delphi/C++Builder, Visual C и т.п.)?

Список литературы

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2004. –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. (электронный ресурс http://www.cluster.bsu.by/download/book_PDF.pdf, http://pilger.mgapi.edu/methods/1441/pos_mpi.pdf)
4. Антонов А.С. Введение в параллельное программирование (методическое пособие). –М.: НИВЦ МГУ, 2002, -69 с. (электронный ресурс <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. (электронный ресурс http://parallel.ru/tech/tech_dev/MPI, http://pilger.mgapi.edu/metods/1441/mpi_book.pdf)
6. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем (учебное пособие, изд. 2, дополненное). –Н.Новгород.: изд. ННГУ им. Н.И.Лобачевского, 2003 (электронный ресурс http://pilger.mgapi.edu/metods/1441/basic_pr.zip).
7. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. –Ростов-на-Дону.: Издательство ООО 'ЦВВР', 2003, -208 с. (электронный ресурс <http://pilger.mgapi.edu/metods/1441/book.pdf>)
8. 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**

**Параллельное программирование
в стандарте MPI**

Учебно-методическое пособие
по выполнению лабораторных работ

ЛР № 020418 от 08 октября 1997 г.

Подписано в печать 02.02.2006 г.
Формат 60 × 84. 1/16.
Объем 4,75 п.л. Тираж 100 экз. Заказ 17.

Московский государственный
университет приборостроения и информатики.
107996, Москва, ул. Стрмынка, 20.