Московский государственный университет им. М.В.Ломоносова
Научно-исследовательский вычислительный центр

ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

методическое пособие

А.С.Антонов

Москва, 2002

Данное пособие предназначено для начального освоения практического курса параллельных вычислений. Предполагается, что приводимой информации достаточно для начала серьезной работы на параллельных компьютерах (в первую очередь, кластерных системах). При этом основной упор делается на освоение практических навыков работы на вычислительном кластере НИВЦ МГУ. Курс включает в себя вводные сведения об операционной системе UNIX, архитектуре суперкомпьютеров и вычислительных кластеров, обнаружении и использовании параллелизма программ, технологиях параллельного программирования и многие практические сведения, необходимые для начала работы. Он рассчитан ориентировочно на 10 занятий, из которых несколько последних отводятся на отладку и оптимизацию модельной задачи. Методическое пособие содержит весь необходимый материал для начала работы на вычислительных кластерах и создания реальных эффективных параллельных программ.

Занятие 1. Введение. Коротко об операционной системе UNIX 5
1. Источники информации 5
2. План занятий, практические задания, политика доступа 5
3. Коротко об операционной системе UNIX 5
Задания: 9
Занятие 2. Вычислительный кластер НИВЦ МГУ 10
1. Архитектура кластера SCI 10
2. Вход на кластер 12
3. Компиляция 12
4. Система очередей 13
4. Web-интерфейс запуска задач на вычислительном кластере 15
5. Задания: 16
Занятие 3. Параллелизм и его использование 17
1. Параллелизм 17
2. Использование параллелизма 20
3. Эффективность распараллеливания 26
4. Обсуждение модельной задачи 27
5. Задания: 28
Занятие 4. Технология MPI 29
2. Введение 29
3. Общие функции MPI 31
4. Прием/передача сообщений между отдельными процессами 33
5. Задания: 46
Занятие 5. Технология MPI (продолжение) 47
1. Коллективные взаимодействия процессов 47
2. Синхронизация процессов 51
3. Работа с группами процессов 51
6. Задания: 52
Занятие 6. Технологии параллельного программирования (обзор) 53
1. Спецкомментарии 53
2. Расширения существующих языков программирования 54
3. Специальные языки программирования 55
4. Библиотеки и интерфейсы, поддерживающие взаимодействие параллельных процессов 55
5. Linda 56
6. Параллельные предметные библиотеки 56
7. Специализированные пакеты и программные комплексы 56
8. Задания: 57
Занятие 7. Технологии построения суперкомпьютеров. Кластерные технологии (обзор) 57
1. Производительность параллельных компьютеров 57
2. Классификация параллельных компьютеров 59
3. Вычислительные кластеры 60
4. Список TOP500 61

5. Сравнение коммуникационных технологий построения кластеров 62
6. Системы хранения данных..... 64
7. Высокопроизводительные вычисления в России 67
8. Задания:..... 68

Занятие 1. Введение. Коротко об операционной системе UNIX

1. Источники информации

В качестве основного источника информации по данному курсу рекомендуется Информационно-аналитический Центр по параллельным вычислениям в сети Интернет **http://parallel.ru**. В частности, многие вопросы, касающиеся использования вычислительного кластера НИВЦ МГУ, подробно освещены в разделе **http://parallel.ru/cluster/**. Для более подробного изучения вопросов, связанных с тематикой параллельных вычислений, рекомендуется книга: В.В.Воеводин, Вл.В.Воеводин "Параллельные вычисления" БХВ-Петербург, 2002, 608 стр.

Для оперативного разрешения вопросов, возникающих при использовании кластеров НИВЦ МГУ, обращайтесь по электронной почте в службу поддержки по адресу **support@parallel.ru**.

2. План занятий, практические задания, политика доступа

Первые занятия направлены на то, чтобы дать необходимый минимум информации для начала практической работы на вычислительном кластере. Настоятельно рекомендуется начинать выполнять практические задания с первого же занятия. Первое из предлагаемых после каждого занятия заданий является обязательным для выполнения, остальные задания рекомендуется выполнять по возможности. Для выполнения некоторых заданий могут потребоваться несколько больший объем знаний, чем затрагиваемый в данном пособии, что должно служить стимулом к более полному освоению темы.

Практическая работа на вычислительном кластере НИВЦ МГУ разрешается только с компьютеров выделенного для этого компьютерного класса и только в оговоренное время для проведения занятий. Для получения возможности доступа на кластер с других компьютеров или в другое время необходимо обращаться в службу поддержки кластера в индивидуальном порядке.

3. Коротко об операционной системе UNIX

UNIX – это многозадачная, многопользовательская система, обладающая широкими возможностями. Ее реализации существуют практически на всех распространенных компьютерных платформах. Более того, подавляющее большинство параллельных компьютеров работает под управлением той или иной разновидности UNIX, например, суперкомпьютеры Cray - под управлением UNICOS, Hewlett-Packard - под управлением HP-UX, IBM - под управлением AIX и т.д. В частности, вычислительный кластер НИВЦ МГУ работает под

управлением **LINUX**, одного из наиболее известных свободно распространяемых диалектов UNIX.

В рамках данного курса нет необходимости и возможности описывать все подробности и принципы организации операционной системы, остановимся на том, что требуется для нормальной работы пользователя в данной среде.

Все объекты в UNIX делятся на два типа: *файлы* и *процессы*. Все данные хранятся в файлах, доступ к периферийным устройствам осуществляется через специальные файлы. Вся же функциональность операционной системы определяется выполнением различных процессов.

Важнейшим пользовательским процессом является основной *командный интерпретатор (login shell)*. При входе пользователя в систему он спрашивает учетное имя пользователя (login) и пароль (password) пользователя. Первое, что нужно сделать после входа в систему с использованием стандартного или назначенного администратором пароля, - это поменять пароль при помощи команды **`passwd`**. Общим пожеланием при выборе нового пароля является следующее: пароль должен хорошо запоминаться и быть трудным для подбора. Для завершения сеанса работы нужно набрать команду **`exit`**.

Для того чтобы посмотреть встроенную в систему справку о командах, нужно использовать команду **`man`**. В частности, чтобы посмотреть описание и ключи самой команды **`man`**, нужно набрать команду **`man man`**.

Каждый пользователь системы имеет уникальное имя, с которым ассоциируется идентификатор пользователя (**UID**). Именно это имя используется при входе в систему (login). Пользователь является членом одной или нескольких групп, с которыми связаны идентификаторы групп (**GID**). Принадлежность к группе определяет дополнительные права, которыми обладают все пользователи группы. Информация о пользователях и группах обычно хранится в системных файлах **`/etc/passwd`**, **`/etc/shadow`** и **`/etc/group`**.

В UNIX все множество файлов организовано в виде древовидной структуры, называемой *файловой системой*. Каждый файл имеет имя, определяющее его расположение в файловой системе. Связь между именами файлов и собственно файлами обеспечивается при помощи *каталогов*. Корнем файловой системы является *корневой каталог*, имеющий имя "**`/`**". Имена всех остальных файлов содержат путь – список каталогов, которые нужно пройти, чтобы достичь файла. При этом имена каталогов разделяются знаком "**`/`**". Например, "**`/home/asa/myfile.txt`**". При перемещении по файловой системе текущий каталог называется "**`.`**", а каталог на единицу более высокого уровня "**`..`**". Кроме того, с каждым пользователем ассоциируется его *домашний каталог*, в котором по умолчанию хранятся его файлы.

Файлы и каталоги в UNIX имеют набор атрибутов, среди которых важно отметить владельца и группу. С их использованием организуется гибкое разграничение доступа к файлам и каталогам. Для просмотра всех атрибутов файла можно использовать команду **ls -l**. Результат выполнения команды для каждого файла будет иметь примерно следующий вид:

```
        1        2  3     4     5     6      7     8
-rwxr-xr--  1 asa group 3422 Feb 28 13:30 test
```

В первом столбце выдачи приводится список *прав доступа* к файлу. Первый символ обозначает тип файла (в данном случае "**-**" означает обычный файл; другие возможные значения: "**d**" – каталог, "**l**" – ссылка и др.), далее идут три группы по три символа, задающие собственно права доступа: первая группа из трех 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 нашем примере владелец файла (**asa**) имеет права на чтение, запись и выполнение (**rwx**), члены группы-владельца (**group**) имеют права на чтение и выполнение файла, но не имеют права на запись в файл (**r-x**), а все остальные пользователи имеют только право на чтение файла (**r--**). Существует также несколько дополнительных атрибутов, а для других типов файлов (например, для каталогов) приведенные атрибуты могут иметь несколько иное значение.

Сменить права доступа к файлу можно при помощи команды

chmod [u g o a][+ - =][r w x] file1…

При этом "**u**" означает смену права доступа для пользователя, "**g**" – для группы, "**o**" – для других пользователей, "**a**" – сразу для всех трех категорий. "**+**" означает добавление соответствующего права, "**-**" – удаление, а "**=**" – присвоение. Например, для добавления членам группы **group** права записи в файл **test** нужно выполнить команду **chmod g+w test**.

Владельцем-пользователем вновь созданного файла является пользователь, который его создал, а владельцем-группой – или первичная группа данного пользователя, или наследуется группа, приписанная текущему каталогу. Для смены владельца-пользователя и владельца-группы файла существуют команды **chown** и **chgrp** соответственно.

Рассмотрим основные операции, которые можно производить с файлами и каталогами:

- **cd [dir]** – переход в каталог **dir**. Если каталог не указан, то переход осуществляется в домашний каталог пользователя;
- **cp file1 file2** – копирование файла;

- **mv file1 file2** – перемещение (изменение имени) файла;
- **rm file1…** – удаление файлов;
- **rmdir dir1…** – удаление каталогов;
- **mkdir dir1…** – создание каталога;
- **pwd** – вывести имя текущего каталога;
- **cat file, more file, less file** – утилиты просмотра содержимого файла;
- **find dir** – поиск в файловой системе, начиная с каталога **dir**.
- **grep <рег_выражение> file1…** – поиск в файлах вхождений регулярного выражения **рег_выражение**.
- …

Второй тип объектов в UNIX – это процессы. Под процессом упрощенно можно понимать программу в стадии ее выполнения. Одновременно в системе выполняется достаточно большое число процессов, часть из которых является системными, а часть – пользовательскими. Для того чтобы просмотреть список выполняющихся процессов, нужно воспользоваться командой **ps**. Каждый процесс имеет некоторый набор атрибутов, в частности, уникальный идентификатор процесса (**PID**), который используется для управления его работой, например, посредством посылки *сигналов*. Для того чтобы завершить выполнение процесса, нужно послать ему соответствующий сигнал командой **kill -9 PID**. Список процессов, занимающих наибольшее количество процессорного времени или системных ресурсов, можно посмотреть, используя команду **top**.

Каждая запущенная программа получает три открытых *потока ввода/вывода*: стандартный ввод, стандартный вывод и стандартный вывод ошибок. По умолчанию все эти потоки ассоциированы с терминалом, однако могут быть перенаправлены на другое устройство, например, в файл. Для перенаправления стандартного ввода можно использовать символ "**<**", для стандартного вывода – "**>**" или "**>>**" (с добавлением), для потока ошибок – "**2>**". Например:

```
program > file.log
```

Здесь запускается программа **program**, а ее стандартный вывод перенаправляется в файл **file.log**.

Можно запустить сразу несколько программ так, что стандартный вывод одной передается непосредственно на стандартный ввод другой. Для этого используется конструкция "**|**", называемая *конвейером* команд:

```
program1 | program2 | program3…
```

Команды конвейера выполняются асинхронными процессами, связь между которыми осуществляется через соответствующие потоки ввода-вывода.

Можно запустить программу в *фоновом режиме*. В этом случае не будет ожидаться завершение ее выполнения, а сразу появится системное приглашение, и

можно будет продолжить работу в командном интерпретаторе. Для этого строку команды необходимо завершить символом "**&**":

```
program &
```

Список пользователей, работающих в данный момент в системе, можно посмотреть при помощи команды **who**. Некоторые сведения о системе выдаются по команде **uname**.

Для редактирования файлов в разных системах могут быть предусмотрены различные наборы средств. Чаще всего имеются редакторы **vi, joe** и другие, можно также пользоваться встроенным редактором файлового менеджера Midnight Commander (**mc**). Другим возможным вариантом является удаленное редактирование файлов с последующей их пересылкой на целевой компьютер.

Компилятор с языка Си обычно вызывается командой **cc** (**CC** для Си++), компилятор с языка Фортран – **f77** (**f90** для Фортрана 90).

Для удаленного доступа на другой компьютер можно пользоваться командой **telnet** (или **ssh** для защищенного доступа), а для копирования файлов – командой **ftp** (для защищенного **sftp** или **scp**).

Задания:

- Составить программу решения систем линейных алгебраических уравнений с квадратной невырожденной матрицей порядка **n** методом Гаусса (любой вариант метода) с использованием языка Си или Фортран.
- В чем существенные отличия ОС UNIX и Windows? Как Вы думаете, почему вычислительные кластеры, как правило, работают под управлением UNIX, а персональные компьютеры – под управлением Windows?
- Может ли один файл находится сразу в двух директориях в файловой системе UNIX?
- Какие права должны быть у файла, чтобы его могли читать, писать и исполнять все пользователи, кроме его владельца? Возможно ли такое вообще?
- Если программа запущена в фоновом режиме, как ее можно завершить?
- Можно ли одной командой разрешить чтение и выполнение, но запретить запись для какого-то файла?
- Как запретить запись в файл с уникальным именем, но неизвестным местонахождением?
- Можно ли одной командой удалить всю файловую систему, начиная с корневого каталога? Что может этому помешать?

- Вывести на экран строку, содержащую PID заданной программы.
- Выведите на дисплей содержимое Вашего домашнего каталога, включая подкаталоги.
- Как создать файл, в имени которого есть пробел? Приведите еще один способ, не изменяя имени команды, которой Вы воспользовались для решения.
- Выведите на экран список процессов, названия которых содержат '**getty**' (без кавычек), кроме процессов, порожденных Вашим поиском.
- Выведите на экран список всех исполняющихся процессов определенного пользователя.
- Программа **abcde** может завершаться успешно и неуспешно. Как запустить программу **abcde** так, чтобы при неуспешном завершении в файл **/tmp/run-result** выводилась строка “abcde failed”? Как модифицировать полученную Вами командную строку таким образом, чтобы в этот файл выводился и код завершения программы?
- Как сделать так, чтобы команда **rm** всегда запрашивала подтверждение удаления файлов? Как сделать так, чтобы это свойство сохранялось между сеансами работы?
- Как переименовать все файлы с именами **myproj*** в соответствующие **myproj*.old**, включая файлы во всех подкаталогах?

Занятие 2. Вычислительный кластер НИВЦ МГУ

1. Архитектура кластера SCI

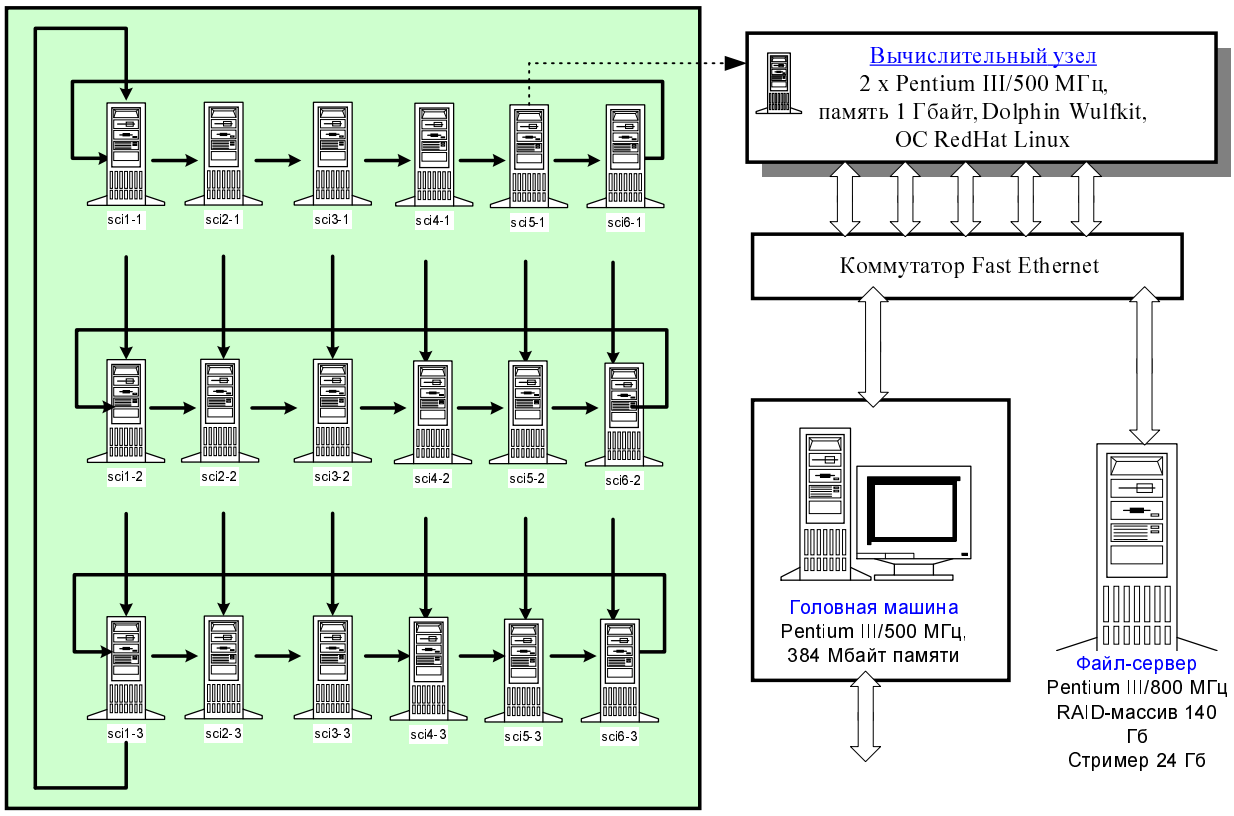
В настоящий момент в НИВЦ МГУ установлено три вычислительных кластера: SKY, SCI и AQUA. Кластер SKY представляет собой вычислительную ферму из двухпроцессорных узлов с процессорами Pentium III/850 МГц, соединенных коммуникационной сетью Fast Ethernet, и используется преимущественно для запуска однопроцессорных или слабосвязанных многопроцессорных задач. Для выполнения практических заданий предоставляется доступ к кластеру SCI, поэтому на его архитектуре остановимся ниже подробнее. Кластер AQUA похож по своей архитектуре на кластер SCI, но построен на более мощных процессорах, и в нем используются более новые программно-аппаратные решения. Подробную информацию о кластерах НИВЦ МГУ можно найти на странице **http://parallel.ru/cluster/**.

Кластер **SCI** построен на базе двухпроцессорных узлов с процессорами Pentium III. Узлы объединяются в двухмерную решетку с помощью высокоскоростной сети SCI (Scalable Coherent Interface). В качестве служебной сети используется Fast Ethernet. Дополнительно установлен однопроцессорный компьютер, используемый в качестве головной машины для загрузки задач на кла-

стер, а также мощный файл-сервер с RAID-массивом. Установлена также специальная рабочая станция, ориентированная на задачи визуализации.

Структура вычислительного кластера (кластер SCI)

36 процессоров Pentium III/500MHz, Dolphin SCI (двумерный тор), производительность 18 GFLOPS



В настоящее время 18 узлов кластера SCI объединены в двухмерную решетку 6x3. Каждый узел имеет однонаправленную связь со следующим по горизонтали и по вертикали (последний имеет связь с первым). Таким образом, каждый узел присутствует в двух однонаправленных кольцах, за организацию обменов по которым отвечают основная и дочерняя карты SCI. SCI-контроллеры в составе комплекта Wulfkit обеспечивают маршрутизацию между кольцами на аппаратном уровне. В случае выхода одного из узлов из строя обеспечивается автоматическая перенастройка маршрутизации, и вся остальная сеть продолжает работать.

Конфигурация узла кластера SCI:

- 2 процессора Pentium III/500 (550) МГц, кэш второго уровня 512 Кбайт;
- 1 Гбайт оперативной памяти SDRAM (4 модуля DIMM по 256 Мбайт);
- Жесткий диск IDE 4.3 Гбайт Quantum (на новых узлах 10 Гбайт Fujitsu);
- Адаптер Fast Ethernet;
- Основная и дочерняя карты SCI.

Суммарная пиковая производительность кластера SCI – около 18 Гфлопс (миллиардов операций с плавающей точкой в секунду). Производительность на реальных приложениях обусловлена, в первую очередь, параллельными свойствами самих приложений, а также производительностью коммуникационной среды. Латентность (время задержки сообщений) в рамках MPI поверх SCI составляет примерно 5.8 микросекунды, а максимальная достигнутая скорость однонаправленных пересылок составляет 81 Мбайт/сек. Для сравнения: в рамках MPI поверх Fast Ethernet скорость пересылок составляет до 10 Мбайт/сек, а время латентности - около 150 микросекунд. При обменах в рамках одного SMP-узла достигается латентность около 2.5 микросекунд и скорость обменов порядка 100 Мбайт/сек для сообщений среднего размера (до 256 Кбайт).

Тест LINPACK представляет собой решение больших систем линейных алгебраических уравнений методом LU-разложения. При решении системы уравнений с матрицей 44000x44000 на 36 процессорах кластера SCI была получена производительность около 10 Гфлопс (миллиардов операций с плавающей точкой в секунду).

2. Вход на кластер

Удаленный доступ к вычислительному кластеру осуществляется через головную машину (**cluster.parallel.ru**). Непосредственный терминальный доступ на узлы кластера невозможен. В настоящий момент удаленный терминальный доступ на кластер осуществляется по протоколу SSH версии 2 (SSH1 пока также поддерживается). В Unix этот протокол поддерживается стандартной командой **`ssh`**. В среде Windows можно пользоваться терминальной программой **TeraTerm** с дополнительным модулем для поддержки SSH или другими программами (**Telneat, SSHSecureShellClient, sshNT.exe, PuTTY** и др.).

В связи с проблемами безопасности протокола FTP, передача файлов на кластер осуществляется по другим протоколам, таким как **scp, Zmodem, SFTP** и др.

3. Компиляция

Для компиляции MPI-приложений рекомендуется пользоваться командами **`mpicc`** (для программ на C), **`mpiCC`** (для программ на C++), и **`mpif77/mpif90`** (для программ на Фортране 77/90). Опция компилятора "**`-o name`**" позволяет задать имя **`name`** для получаемого выполнимого файла, по умолчанию выполнимый файл называется **`a.out`**. Для оптимизации рекомендуется использовать ключ компилятора "**`-fast`**", например:

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

Если необходимо только скомпилировать один объектный модуль и не выполнять линковку, используется опция "**`-c`**", например:

```
mpiCC -c program2.C
```

Для сборки многомодульных приложений целесообразно пользоваться утилитой **make**. Простейшие примеры MPI-программ на языках Си и Фортран 77/90 доступны на кластере в каталоге **/usr/local/examples**.

4. Система очередей

Система управления очередями **Cleo (http://parallel.ru/cluster/batch.html)** предназначена для управления прохождением задач на многопроцессорных вычислительных установках (в том числе кластерных). Она позволяет автоматически распределять вычислительные ресурсы между задачами, управлять порядком их запуска, временем работы, получать информацию о состоянии очередей. В настоящее время на кластере SCI сформированы два независимых подраздела: "**long**" (для длительных задач) и "**short**" (для коротких задач). По умолчанию во всех командах используется очередь, указанная в переменной окружения **QS_QUEUE**. Значение этой переменной можно изменить в файле **~/.bash_profile** командой "**export QS_QUEUE=short**".

Задача ставится в очередь обычной командой запуска MPI-приложений (в квадратных скобках указаны необязательные параметры):

mpirun -np N [-q Q] [-maxtime T] [-p P] <программа с аргументами>,

где **N** - число процессоров, которое должно быть не более разрешенного числа процессоров для одной задачи, **Q** - это очередь, в которую будет поставлена задача, **T** – это максимальное время работы задачи в минутах, **P** - приоритет задачи в очереди.

Реально задача начнет выполняться, как только она окажется на верхушке очереди, если при этом будут свободны **N** процессоров. Система автоматически подбирает свободные узлы (процессоры) для запуска задачи. Гарантируется, что на каждом узле будет запущено прикладных процессов не более, чем реально доступно процессоров (2). Если задача поставлена в очередь, то система выдает подтверждение и присваивает задаче уникальный номер (ID). В дальнейшем ID может использоваться для получения информации о состоянии задачи и о результатах ее выполнения. После постановки задачи в очередь пользователь может отключиться от терминала, а затем войти на систему заново для просмотра результатов.

Задачи с большим приоритетом будут идти на счет раньше задач с меньшим приоритетом. Приоритет обычных задач по умолчанию равен 10. Если не важно, чтобы задача пошла на счет как можно быстрее, и целесообразно уступить очередь другим пользователям, то можно уменьшить значение приоритета. Например:

```
mpirun -np 32 -p 8 cg.A.32
```

Эта задача со значением приоритета 8 пойдет на счет не раньше, чем пойдут на счет все задачи с приоритетами 9 и 10.

Если, наоборот, требуется посчитать задачу как можно быстрее, то можно обратиться к администраторам с просьбой увеличить значение приоритета задачи. Самостоятельно повысить приоритет своей задачи более чем до 10 нельзя. Можно изменить приоритет задачи, стоящей в очереди, с помощью команды

chpri –p <приоритет> –n <номер задачи>

Параметр **–maxtime** устанавливает предельное время работы задачи в минутах. Это поможет оптимально планировать работу кластера и поможет другим пользователям ориентироваться при постановке задач в очередь. По умолчанию устанавливается очень небольшое предельное время. При истечении предельного времени задача будет сниматься со счета. Например, пусть на кластере свободно 8 процессоров, а в очереди уже стоит задача на 32 процессора (но все 32 процессора в ближайшие 7 часов не освободятся). Тогда если поставить в очередь короткую 4-процессорную задачу, указав максимальное время:

mpirun –np 4 –maxtime 10 program,

то эта задача сразу пойдет на счет. Таким образом, вычислительные ресурсы будут распределяться более оптимально.

Посмотреть текущее состояние очереди можно командой:

mps [параметры]

Сначала показываются задачи, работающие в текущий момент, а потом задачи, стоящие в очереди. Параметры команды **mps** следующие:

–q <очередь> - просмотр заданной очереди на текущем кластере или на указанном кластере, если очередь указана в виде **queue@cluster**, например, **long@sky–main**;

–q ALL - просмотр всех доступных очередей на всех кластерах;

–u <пользователь> - просмотр задач данного пользователя;

–t <имя задачи> - просмотр только задач с данным именем;

–n <номер задачи> - поиск задачи с данным номером;

–r - просмотр только работающих задач;

–? или **–h** - просмотр списка опций.

Параметры можно комбинировать, например, команда

mps –q ALL –r –u alex

выдаст список всех работающих задач пользователя **alex** на всех кластерах.

Удалить свою задачу, стоящую в очереди или выполняющуюся, можно командой

tasks [–q <очередь>] –d ID

Удалить все свои задачи можно командой

tasks [–q <очередь>] –d all

Для команды “**tasks –d**” также предусмотрена более короткая форма – “**qd**”.

По окончании работы задачи пользователю выдается сообщение на терминал. Выдача программы помещается в файл в рабочей директории с именем **<задача>.out-<номер>**. Кроме того, создается файл отчета **<задача>.rep-<номер>**, где указываются следующие данные: командная строка при запуске задачи, число процессоров, код возврата, имя выходного файла, рабочая директория, астрономическое время работы программы, имена узлов, на которых была запущена программа.

4. Web-интерфейс запуска задач на вычислительном кластере

Для того чтобы любой пользователь, не искушенный в вопросах распараллеливания, мог увидеть эффект от использования параллельных программ, на вычислительном кластере НИВЦ МГУ предусмотрена возможность запуска некоторых тестовых примеров посредством Web-интерфейса.

Тестовый полигон (**http://parallel.ru/polygon/**) предназначен для оперативного доступа к вычислительным ресурсам для проведения небольших предварительных экспериментов и выбора вычислительной платформы. Главное, что нужно понять пользователю в начале работы, это на какой тип вычислителя ориентироваться. Основных параметров несколько: выбор между компьютерами с общей или распределенной памятью, соотношение между скоростью процессоров и скоростью обмена данными в коммуникационной среде, выбор технологии параллельного программирования, выбор алгоритмического подхода и некоторые другие.

Запуская тестовые задачи на предоставляемых программно-аппаратных платформах, пользователь может оценить эффективность реализации тех или иных коммуникационных схем при использовании различных сетевых технологий (на настоящий момент SCI и Fast Ethernet), сравнить эффективность различных конструкций, предоставляемых технологиями параллельного программирования (например, различных вариантов пересылок или глобальных операций в MPI), а в конечном итоге, оценить предполагаемую эффективность возможных параллельных реализаций собственной задачи на предоставляемых вычислительных ресурсах.

На данный момент в Тестовом полигоне реализованы следующие типовые алгоритмические структуры межпроцессорного взаимодействия:

- пересчет элементов прямоугольной матрицы по известному закону через соседние элементы (с заданной шириной граничной области);

- различные варианты реализации пересылки данных от каждого процессора каждому;

- различные варианты реализации пересылки данных по кольцевой топологии;

- различные варианты реализации двунаправленных пересылок данных между двумя выделенными процессорами.

Используя web-интерфейс, пользователь формирует запрос на выбор целевой программно-аппаратной среды и указывает интересующее его тестовое приложение. Опираясь как на доступный парк вычислительных систем Центра, так и на возможность оперативного управления распределением заданий между системами, происходит запуск тестового приложения и возврат результатов пользователю. На данный момент в качестве аппаратных платформ Тестового полигона предоставляются вычислительные кластеры SCI и SKY, однако несложно добавить возможность выполнения пользовательских программ и на других платформах. Задача с нужными параметрами передается на выполнение системе очередей Cleo на выбранном кластере. В зависимости от выбранного пользователем режима получения результатов (непосредственно в окне браузера или по указанному адресу электронной почты) Web-интерфейс либо дожидается постановки задачи на выполнение и получения результатов, либо только информирует пользователя о постановке его задачи на выполнение.

Естественно, на запуск программ в таком режиме наложен целый ряд ограничений административного характера. Так, все задачи, выбираемые в качестве типовых алгоритмических структур, достаточно простые и даже при максимально допускаемых значениях параметров выполняются не более нескольких секунд, количество предоставляемых под эти задачи процессоров жестко ограничено сверху, задачи ставятся в очередь со стандартным приоритетом и т.д. Для предотвращения намеренной или случайной перегрузки вычислительных ресурсов многократной постановкой задач в очередь предусмотрена блокировка приема запросов к Web-интерфейсу с одного IP-адреса на определенный промежуток времени (на данный момент 5 минут). Кроме того, ведется полный протокол использования Тестового полигона всеми пользователями.

5. Задания:

- Проверить на тестовой матрице правильность выполнения программы решения систем линейных алгебраических уравнений методом Гаусса, замерить время выполнения, исследовать возможности компилятора по оптимизации выполнения данной программы.
- Откомпилировать и проверить эффективность выполнения программы вычисления числа Пи на различном числе процессоров (**`/usr/local/examples/mpi/cpi.c`** или **`fpi.f`**).
- Для чего на кластере нужна быстрая сеть? Для каких задач достаточно 100-мегабитной сети?

- Какие преимущества имеет технология SCI перед Fast Ethernet при построении кластерных систем?
- Каковы достоинства и недостатки расположения домашних каталогов пользователей на файл-сервере, а не на локальном диске?
- Как узнать, кто еще из пользователей работает в данный момент на кластере, и чьи задачи сейчас считаются?
- Целесообразно ли запускать на одном двухпроцессорном узле больше двух счетных процессов? Могут ли на одном а) кластере, б) узле, в) процессоре одновременно считаться задачи разных пользователей?
- Как Вы думаете, на что прежде всего повлияет изменение размерности решетки кластера с коммуникационной сетью SCI (например, сравните варианты конфигураций 6*6 и 3*12 узлов)?
- Для чего нужно указывать при запуске задачи параметр **`-maxtime`**?
- В очереди от Вашего имени поставлены задачи task1, task2, new1 и new2. Как удалить из очереди только задачи task1 и task2?
- Как узнать, на каких именно узлах запустилась Ваша задача до ее окончания?
- Как узнать, с каким кодом завершения закончилась Ваша задача?
- Предположим, для запуска Вашей задачи необходимо 4 процессора и около 10 минут времени на счет. В системе свободно 6 процессоров, но первой на счет стоит задача на 10 процессоров, а недостающие освободятся через несколько часов (по прогнозу системы). Как запустить Вашу задачу на счет раньше?
- Как сменить приоритет Вашей задаче, уже стоящей в очереди?
- Используя Web-интерфейс запуска задач, сравните эффективность кластеров SCI и SKY на задаче рассылки от каждого процесса каждому для разного количества процессоров и для разных длин сообщений.
- Используя Web-интерфейс запуска задач, выберите оптимальный вариант реализации пересылки данных от каждого процессора каждому.
- Используя Web-интерфейс запуска задач, выберите оптимальный вариант реализации пересылки данных по кольцевой топологии.

Занятие 3. Параллелизм и его использование

1. Параллелизм

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность. Оба вида параллельной обработки интуитивно понятны, поэтому сделаем лишь небольшие пояснения.

Параллельная обработка. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что имеется пять таких же независимых устройств, способных работать одновременно и независимо, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично, система из N устройств ту же работу выполнит примерно за 1000/N единиц времени.

Конвейерная обработка. Что необходимо для сложения двух вещественных чисел, представленных в форме с плавающей запятой? Целое множество мелких операций, таких как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.п. Процессоры первых компьютеров выполняли все эти “микрооперации” для каждой пары аргументов последовательно одна за другой до тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых.

Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передает результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если же каждую микрооперацию выделить в отдельный этап (или иначе говорят - ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находиться первые пять пар аргументов, первый результат будет получен через 5 единиц времени, каждый следующий – через одну единицу после предыдущего, а весь набор из ста пар будет обработан за 5+99=104 единицы времени, то есть будет получено ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера).

Приблизительно так же будет и в общем случае. Если конвейерное устройство содержит l ступеней, а каждая ступень срабатывает за одну единицу времени, то время обработки n независимых операций этим устройством составит $l+n-1$ единиц. Если это же устройство использовать в монопольном режиме (как последовательное), то время обработки будет равно $l \times n$. В результате получим ускорение почти в l раз за счет использования конвейерной обработки данных.

Казалось бы, конвейерную обработку можно с успехом заменить обычным параллелизмом, для чего продублировать основное устройство столько раз, сколько ступеней конвейера предполагается выделить. Однако стоимость и

сложность получившейся системы будет несопоставима со стоимостью и сложностью конвейерного варианта, а производительность будет почти такой же.

Для того чтобы написать параллельную программу, необходимо выделить в ней группы операций, которые могут вычисляться одновременно и независимо разными процессорами, функциональными устройствами или же разными ступенями конвейера.

Возможность этого определяется наличием или отсутствием в программе истинных информационных зависимостей. Две операции программы (а в данном случае под операцией можно понимать как отдельное срабатывание некоторого оператора, так и более крупные куски кода программы) называются *информационно зависимыми*, если результат выполнения одной операции используется в качестве аргумента в другой. Очевидно, что если операция B информационно зависит от операции A (то есть, использует какие-то результаты операции A в качестве своих аргументов), то операция B может быть выполнена только по завершении операции A. С другой стороны, если операции A и B не являются информационно зависимыми, то алгоритмом не накладывается никаких ограничений на порядок их выполнения, в частности, они могут быть выполнены одновременно. Таким образом, задача распараллеливания программы обычно сводится к нахождению в ней достаточного количества информационно независимых операций, распределению их между вычислительными устройствами, обеспечению синхронизации и необходимых коммуникаций.

Введенное понятие информационной зависимости является достаточно простым, но исследование всего набора информационных зависимостей, существующих в реальной программе, является весьма сложной задачей. Достаточно представить себе программу, состоящую из десятков тысяч операторов, и учесть, что каждый цикл может состоять из огромного количества итераций, чтобы примерно представить себе сложность возникающих проблем. Для того чтобы формализовать задачу и облегчить анализ, вводится понятие *графов информационных зависимостей.*

Вершинами в таких графах обычно являются некоторые операции программы, а в случае, если между двумя операциями существует информационная зависимость, то соответствующие этим операциям вершины соединяются направленной дугой, началом которой является вершина-поставщик информации, а концом – вершина-потребитель информации. Для упрощения исследования графа зависимостей в нем выделяется остовной подграф, имеющий минимальное число дуг при выполнении следующего условия: если две вершины графа зависимостей связаны путем, то они же должны быть связаны путем и в выделенном подграфе, который называется *минимальным графом зависимостей*. С помощью минимальных графов зависимостей можно решать все те же задачи, которые можно решать и с помощью обычных графов зависимостей, однако они

намного проще и в большинстве случаев позволяют производить эффективный анализ структуры зависимостей программы.

Если вершинам графа соответствуют отдельные срабатывания операторов программы, то такой граф называется *информационной историей* выполнения программы. Информационная история содержит максимально подробную информацию о структуре информационных зависимостей анализируемой программы, поэтому именно она используется при анализе программ с целью распараллеливания. Однако сложность анализа такова, что если в простых случаях можно построить и проанализировать получающийся граф вручную, то для больших реальных программ необходимы инструментальные программные средства.

2. Использование параллелизма

Предположим теперь, что мы научились каким-то образом строить и исследовать информационную историю выполнения программы. Выделив в ней множества информационно независимых операций, приходим к вопросу: каким же образом распределять эти множества между процессорами или другими обрабатывающими устройствами?

Достаточно очевидным способом будет следующий. Пометим в информационной истории те операции, которые зависят только от внешних данных программы и скажем, что такие операции принадлежат к первому ярусу информационной истории. На второй ярус поместим операции, зависящие только от операций первого яруса и внешних данных и так далее. Распределив таким образом операции информационной истории, получаем ее вид, называемый *ярусно-параллельной формой* программы. Количество ярусов ярусно-параллельной формы называется *длиной критического пути*. По построению операции, попавшие на один ярус, не могут состоять в отношении информационной зависимости, а значит, могут быть выполнены одновременно. Таким образом, процесс получения параллельной программы может быть следующим: сначала распределяем между процессорами операции первого яруса, после из завершения – операции второго яруса и т.д. Поскольку любая операция n-го яруса зависит хотя бы от одной операции (n-1)-го яруса, то ее нельзя начать выполнять раньше, чем завершится выполнение операций предыдущего яруса, а значит, ярусно-параллельная форма представляет собой в определенном смысле максимально параллельную реализацию программы. При этом длина критического пути характеризует количество параллельных шагов, необходимых для ее выполнения.

Однако ярусно-параллельная форма почти никогда не используется для практического распараллеливания программ. Причиной этого является то, что описанный способ распараллеливания плохо согласован как с конструкциями реальных языков программирования, так и с архитектурными особенностями современных компьютеров. Чтобы убедиться в этом, можете попробовать сконструировать таким способом параллельную реализацию практически любого ал-

горитма и оценить как сложность написания такой программы, так и количество необходимых коммуникаций между процессорами.

В реальности гораздо чаще используются другие способы распараллеливания, использующие характерные особенности наиболее распространенных языков программирования. Так, самым простым вариантом распределения работ между процессорами является примерно следующая конструкция (*крупноблочное распараллеливание*):

```
if (MyProc == 0) {/* операции, выполняемые 0-ым процессором */}
...
if (MyProc == K) {/* операции, выполняемые K-ым процессором */}
...
```

При этом предполагается, что каждый процессор каким-то образом может получить уникальный номер, присвоить его переменной **MyProc** и использовать в дальнейшем для получения участка кода для независимого исполнения. Таким образом, в приведенном примере операции в первых фигурных скобках будут выполнены только процессором с номером **0**, операции во вторых фигурных скобках – процессором с номером **K** и т.д. При этом, естественно, необходимо, чтобы одновременно разные процессоры могли выполнять только блоки информационно независимых операций, что может потребовать операций синхронизации процессоров, а также при необходимости нужно обеспечивать обмен данными между процессорами.

Однако далеко не всегда удается выделить в программе достаточно большое число блоков независимых операций, а значит, при значительном числе процессоров в используемом компьютере, часть из них будет простаивать. Поэтому наряду с предыдущим способом используют также более низкоуровневое распараллеливание. Как показывает практика, наибольший ресурс параллелизма в программах сосредоточен в циклах. Поэтому наиболее распространенным способом распараллеливания является то или иное *распределение итераций циклов*. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно тем или иным способом раздать разным процессорам для одновременного исполнения. Условно это может быть выражено примерно следующей конструкцией:

```
for (i = 0; i < N; i++) {
     if (i ~ MyProc) {
/* операции i-й итерации для выполнения процессором MyProc */
     }
}
```

Здесь конструкция **i ~ MyProc** применена для того, чтобы указать, что номер итерации **i** каким-то образом соотносится с номером процессора **MyProc**. Конкретный способ задания этого соотношения определяет то, какие итерации цикла на какие процессоры будут распределяться. В принципе, ничто не мешает, например, раздать всем процессорам по одной итерации, а все остальные

итерации выполнить каким-то одним процессором. Однако очевидно, что в подавляющем числе случаев такое распределение неэффективно, поскольку все процессоры, кроме одного, выполнив свою итерацию, будут, скорее всего, простаивать. Таким образом, одним из требований к распределению итераций (как, впрочем, и к процессу распараллеливания вообще) является по возможности *равномерная загрузка процессоров*. Наиболее распространенные способы распределения итераций циклов в той или иной степени удовлетворяют этому требованию.

Блочное распределение итераций предполагает, что распределение итераций цикла по процессорам ведется блоками по несколько последовательных итераций. В простейшем случае количество итераций цикла `N` делится на число процессоров `P`, результат округляется до ближайшего целого сверху и число `⌈N/P⌉` определяет количество итераций в блоке. При этом почти все процессоры получают одинаковое количество итераций, однако один или несколько последних процессоров могут простаивать. Выбор меньшего размера блока может уменьшить этот дисбаланс, однако при этом часть итераций останется нераспределенной. Если нераспределенные итерации снова начать распределять такими же блоками, начиная с первого процессора, то получим *блочно-циклическое распределение* итераций. Если уменьшать количество итераций дальше, то дойдем до распределения по одной итерации, которое называется *циклическим распределением*. Циклическое распределение позволяет минимизировать дисбаланс в загрузке процессоров, возникавший при блочных распределениях.

Рассмотрим небольшой пример. Пусть требуется распределить по процессорам итерации следующего цикла:

```
for (i = 0; i < N; i++)
     a[i] = a[i] + b[i];
```

Пусть в целевом компьютере имеется `P` процессоров с номерами `0...P-1`. Тогда блочное распределение итераций этого цикла можно записать следующим образом:

```
k = (N-1)/P + 1; /* размер блока итераций */
ibeg = MyProc * k; /* начало блока итераций процессора MyProc */
iend = (MyProc + 1) * k - 1; /* конец блока итераций процессора
MyProc */
if (ibeg >= N) iend = ibeg - 1; /* если процессору не досталось
итераций */
else if (iend >= N) iend = N - 1; /* если процессору досталось
меньше итераций */
for (i = ibeg; i <= iend; i++)
     a[i] = a[i] + b[i];
```

Циклическое распределение итераций того же цикла можно записать так:

```
for (i = MyProc; i < N; i+=P)
     a[i] = a[i] + b[i];
```

Нужно заметить, что все соображения о распределении итераций циклов с целью достижения равномерности загрузки процессоров имеют смысл только в предположении, что распределяемые итерации приблизительно равноценны по времени исполнения. Во многих реальных случаях (например, при решении матричных задач с треугольной матрицей) это может быть не так, а значит, могут потребоваться совершенно другие способы распределения.

Однако только равномерной загрузки процессоров обычно недостаточно для получения эффективной параллельной программы. Только в крайне редких случаях программа не сдержит информационных зависимостей вообще. Если же есть информационная зависимость между операциями, которые при выбранной схеме распределения попадают на разные процессоры, то потребуется пересылка данных. Обычно пересылки требуют достаточно большого времени для своего осуществления, поэтому другой важной целью при распараллеливании является *минимизация количества и объема необходимых пересылок данных*. Так, например, при наличии информационных зависимостей между **i**-й и **(i+1)**-й итерациями некоторого цикла блочное распределение итераций может оказаться эффективнее циклического, потому что при блочном распределении соседние итерации попадают на один процессор, а значит, потребуется меньшее количество пересылок, чем при циклическом распределении.

До сих пор говорилось о распределении итераций одномерных циклов, однако в программах часто встречаются многомерные циклические гнезда, причем каждый цикл такого гнезда может содержать некоторый ресурс параллелизма. Для его использования производят анализ и разбиение пространства итераций исследуемого фрагмента. *Пространством итераций* гнезда тесновложенных циклов называют множество целочисленных векторов **I**, координаты которых задаются значениями параметров циклов данного гнезда. Задача распараллеливания при этом сводится к разбиению множества векторов **I** на подмножества, которые выполняются последовательно друг за другом, но в рамках каждого такого подмножества итерации могут быть выполнены одновременно и независимо.

Среди методов анализа пространства итераций можно выделить несколько наиболее известных: методы гиперплоскостей, координат, параллелепипедов и пирамид.

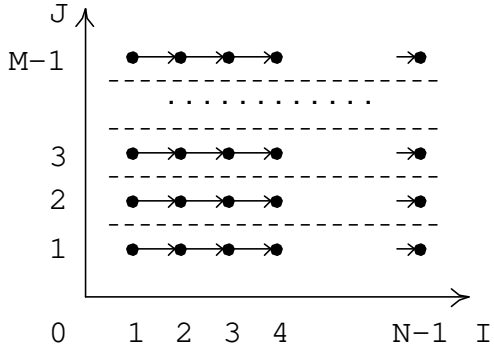
Метод гиперплоскостей заключается в том, что пространство итераций размерности **n** разбивается на гиперплоскости размерности **n–1** так, что все операции, соответствующие точкам одной гиперплоскости, могут выполняться одно-

временно и асинхронно. *Метод координат* заключается в том, что пространство итераций фрагмента разбивается на гиперплоскости, ортогональные одной из координатных осей. *Метод параллелепипедов* является логическим развитием двух предыдущих методов и заключается в разбиении пространства итераций на **n**-мерные параллелепипеды, объем которых определяет результирующее ускорение программы. В *методе пирамид* выбираются итерации, вырабатывающие значения, которые далее в теле гнезда циклов не используются. Каждая такая итерация служит основанием отдельной параллельной ветви, в которую также входят все итерации, влияющие информационно на выбранную. При этом зачастую информация в разных ветвях дублируется, что может привести к потере эффективности.

Рассмотрим небольшой пример:

```
for (i = 1; i < N; i++)
     for (j = 1; j < M; j++)
          a[i,j] = a[i-1,j] + a[i,j];
```

Пространство итераций данного фрагмента можно изобразить следующим образом:

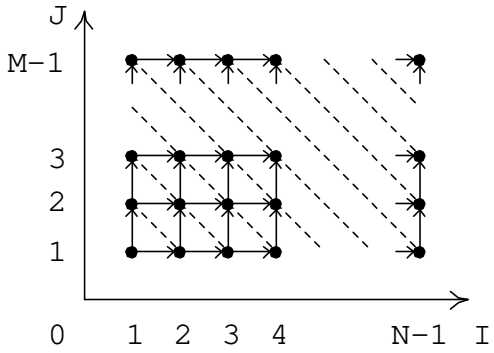


На этом рисунке кружки соответствуют отдельным срабатываниям оператора присваивания, а стрелки показывают информационные зависимости. Сразу видно, что разбиение пространства итераций по измерению **I** приведет к разрыву информационных зависимостей. Однако информационных зависимостей по измерению **J** нет, поэтому возможно применение метода координат с разбиением пространства итераций гиперплоскостями, ортогональными оси **J**, например, как показано на рисунке пунктирными линиями. Затем операции, попавшие в одну группу, распределяются для выполнения на один процессор целевого компьютера.

Немного усложним пример:

```
for (i = 1; i < N; i++)
     for (j = 1; j < M; j++)
          a[i,j] = a[i-1,j] + a[i,j-1];
```

Пространство итераций данного фрагмента можно изобразить следующим образом:



Очевидно, что метод координат в данном случае неприменим, потому что любое разбиение как по измерению **`I`**, так и по измерению **`J`** приведет к разрыву информационных зависимостей. Однако на рисунке пунктирными линиями показаны гиперплоскости **`I + J = const`**, которые содержат вершины, между которыми нет информационных зависимостей. Это означает возможность применения метода гиперплоскостей, при котором осуществляется перебор гиперплоскостей **`I + J = const`**, и для каждой из них соответствующие операции распределяются между процессорами целевого компьютера.

Если переходить от распараллеливания отдельных циклических конструкций к распараллеливанию целой программы, то может оказаться невыгодным использовать весь найденный ресурс параллелизма (в особенности, на компьютерах с распределенной памятью), поскольку потребуется большое количество перераспределений данных между выполнением циклов.

В некоторых случаях можно добиться более эффективного распараллеливания программы при помощи *эквивалентных преобразований* – таких преобразований кода программы, при которых полностью сохраняется результат ее выполнения. Существует достаточно большое число подобных преобразований, полезных в различных случаях, например: перестановки циклов, схлопывание циклов, расщепление циклов и т.д.

Приведем один небольшой пример. Допустим, что в программе содержится такой цикл:

```
for (i = 1; i < N; i++) {
     a[i] = a[i] + b[i];
     c[i] = c[i–1] + b[i];
}
```

В данном цикле есть информационные зависимости между срабатываниями второго оператора из тела цикла на **`i`**-й и **`(i–1)`**-й итерациях. Поэтому нельзя просто раздать итерации исходного цикла для выполнения различным процессорам. Однако, достаточно очевидно, что этот цикл можно разбить на два, первый из которых станет параллельным:

```
for (i = 1; i < N; i++)
     a[i] = a[i] + b[i];
for (i = 1; i < N; i++)
     c[i] = c[i-1] + b[i];
```

В некоторых случаях и эквивалентные преобразования бессильны помочь в распараллеливании программы, но допустимы другие методы при предположении, что можно пренебречь ошибками округления. Например, пусть нужно произвести суммирование элементов массива:

```
for (i = 0, s = 0; i < N; i++)
     s+=a[i];
```

Если подойти к этому циклу формально, то распараллелить его не получится, так как между итерациями цикла существуют информационные зависимости. Однако, если ошибками округления, вызванными разным порядком суммирования элементов массива **a**, допустимо пренебречь, то данный фрагмент можно распараллелить при помощи широко известной *схемы сдваивания*: на первом шаге первый процессор суммирует элементы **a[0]** и **a[1]**, второй процессор суммирует элементы **a[2]** и **a[3]** и т.д., на следующем шаге попарно суммируются эти частичные суммы и так до получения окончательного результата. Если имеется **N/2** процессоров, то весь алгоритм суммирования выполняется за $\log_2 N$ параллельных шагов.

3. Эффективность распараллеливания

Естественно, что, используя параллельную систему с **p** вычислительными устройствами, пользователь ожидает получить ускорение своей программы в **p** раз по сравнению с последовательным вариантом. Но действительность практически всегда оказывается далека от идеала.

Предположим, что структура информационных зависимостей программы определена (что в общем случае является весьма непростой задачей), и доля операций, которые нужно выполнять последовательно, равна **f**, где $0 \le f \le 1$ (при этом доля понимается не по статическому числу строк кода, а по времени выполнения последовательной программы). Крайние случаи в значениях **f** соответствуют полностью параллельным **(f = 0)** и полностью последовательным **(f = 1)** программам. Тогда для того, чтобы оценить, какое ускорение **S** может быть получено на компьютере из **p** процессоров при данном значении **f**, можно воспользоваться *законом Амдала:*

$$S \le \frac{1}{f + \frac{1-f}{p}}$$

Например, если 9/10 программы исполняется параллельно, а 1/10 по-прежнему последовательно, то ускорения более 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0). Отсюда можно сделать вывод, что не любая программа может быть эффективно распараллелена. Для того чтобы это было возможно, необходимо, чтобы доля информационно независимых операций была очень большой. В принципе, это не должно отпугивать от параллельного программирования, потому что, как показывает практика, большинство вычислительных алгоритмов устроено в этом смысле достаточно хорошим образом.

Предположим теперь, что в программе относительно немного последовательных операций. Казалось бы, в данном случае все проблемы удалось разрешить. Но представьте, что доступные вам процессоры разнородны по своей производительности. Значит, будет такой момент, когда кто-то из них еще трудится, а кто-то уже все сделал и бесполезно простаивает в ожидании. Если разброс в производительности процессоров большой, то и эффективность всей системы при равномерной загрузке будет крайне низкой.

Но предположим, что все процессоры одинаковы. Проблемы кончились? Опять нет! Процессоры выполнили свою работу, но результатами чаще всего надо обмениваться для продолжения вычислений, а на передачу данных уходит время, и в это время процессоры опять простаивают... Кроме указанных, есть и еще большое количество факторов, влияющих на эффективность выполнения параллельных программ, причем все они действуют одновременно, а значит, все в той или иной степени должны учитываться при распараллеливании.

Таким образом, заставить параллельную вычислительную систему или супер-ЭВМ работать с максимальной эффективностью на конкретной программе - это задача не из простых, поскольку *необходимо тщательное согласование структуры программ и алгоритмов с особенностями архитектуры параллельных вычислительных систем.*

4. Обсуждение модельной задачи

Предлагаемая для реализации модельная программа (метод Гаусса решения систем линейных алгебраических уравнений, **http://parallel.ru/vvv/tasks/**, первая) является достаточно простой и в то же время содержательной с точки зрения распараллеливания. На ее примере можно пройти все основные этапы процесса написания параллельных программ и встретиться со многими подстерегающими на этом пути трудностями.

Предлагается следующая последовательность действий по написанию и оптимизации программы (в общем случае, не обязательная при создании параллельной программы):

- математическая постановка задачи, запись в формульном виде;
- построение вычислительного алгоритма;
- создание и оптимизация последовательной программы;
- исследование ресурса параллелизма программы и выбор метода распараллеливания;
- написание “какой-то” параллельной программы с использованием любых конструкций любых технологий параллельного программирования;
- оптимизация параллельной структуры программы, выбор наиболее подходящих конструкций, возможно, принятие другого решения по методу распараллеливания;
- дополнительная оптимизация программы, связанная, например, с экономией памяти или другими требованиями со стороны аппаратуры или программного обеспечения.

5. Задания:

- Исследовать информационную структуру программы, реализующей метод Гаусса решения систем линейных алгебраических уравнений, предложить различные методы распределения операций между процессорами, выбрать один из них для реализации и мотивировать свой выбор.
- Пусть конвейерное устройство состоит из **n** ступеней, каждая из которых выполняется за t_i **(i=1…n)** тактов. Пусть на данном устройстве нужно совершить **N** операций. За какое минимальное количество тактов это может быть сделано?
- Что такое балансировка загрузки и как она влияет на ускорение работы программы?
- Опишите ситуацию, при которой использование кэш-памяти процессоров может нарушить закон Амдала (суперлинейное ускорение).
- Предположим, что нам нужно ускорить работу программы в 20 раз. 1/10 часть этой программы можно ускорить не более чем в 10 раз. Во сколько раз нужно ускорить оставшиеся 9/10 программы, чтобы достичь поставленной цели?
- Чему равна длина критического пути информационного графа следующих фрагментов:

а)

```
for (i = 1; i < N; i++) {
     for (j = 1; j < N; j++) {
          a[i][j] = (a[i-1][j] + a[i][j-1])/2;
     }
```

```
}
```

б)

```
for (i = 1; i <= n; i++) {
     for (j = 1; j <= n; j++) {
          u[i+j] = u[2*n+1-i-j];
     }
}
```

- Можно ли выполнить параллельно фрагменты из предыдущего задания и если да, то как?
- Верно ли утверждение, что если некоторый фрагмент программы может быть эффективно реализован на конвейерном устройстве, то он может быть эффективно реализован и в параллельном режиме на наборе последовательных устройств (и наоборот)?
- Придумайте задачу, которая бы эффективно выполнялась на макроконвейере, ступенями которого являются отдельные компьютеры локальной сети.

Занятие 4. Технология MPI

2. Введение

Наиболее распространенной технологией программирования параллельных компьютеров с распределенной памятью в настоящее время является MPI. Основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии данной технологии — *Message Passing Interface*. Стандарт MPI фиксирует интерфейс, который должна соблюдать как система программирования MPI на каждой вычислительной системе, так и пользователь при создании своих программ. Современные реализации, чаще всего, соответствуют стандарту MPI версии 1.1. В 1997—1998 годах появился стандарт MPI-2.0, значительно расширивший функциональность предыдущей версии. Однако до сих пор этот вариант MPI не получил широкого распространения. Везде далее, если иного не оговорено, мы будем иметь дело со стандартом 1.1.

MPI поддерживает работу с языками Си и Фортран. В данном пособии все примеры и описания всех функций будут даны с использованием языка Си. Однако это совершенно не является принципиальным, поскольку основные идеи MPI и правила оформления отдельных конструкций для этих языков во многом схожи. Полная версия интерфейса содержит описание более 120 функций. Наша задача — объяснить идею технологии и помочь освоить необходимые на практике компоненты. Более подробно об интерфейсе MPI можно почитать на странице **http://parallel.ru/tech/tech_dev/mpi.html.**

Интерфейс поддерживает создание параллельных программ в стиле MIMD, что подразумевает объединение процессов с различными исходными текстами. Однако на практике программисты гораздо чаще используют SPMD-модель, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Все дополнительные объекты: имена функций, константы, предопределенные типы данных и т.п., используемые в MPI, имеют префикс **`MPI_`**. Если пользователь не будет использовать в программе имен с таким префиксом, то конфликтов с объектами MPI заведомо не будет. Все описания интерфейса MPI собраны в файле **`mpi.h`**, поэтому в начале MPI-программы должна стоять директива **`#include <mpi.h>`**.

MPI-программа — это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается. Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Для локализации взаимодействия параллельных процессов программы можно создавать *группы процессов*, предоставляя им отдельную среду для общения — *коммуникатор*. Состав образуемых групп произволен. Группы могут полностью входить одна в другую, не пересекаться или пересекаться частично. При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя **`MPI_COMM_WORLD`**. Этот коммуникатор существует всегда и служит для взаимодействия всех процессов MPI-программы.

Каждый процесс MPI-программы имеет уникальный атрибут *номер процесса*, который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда становятся понятными *два основных атрибута процесса: коммуникатор и номер в коммуникаторе*. Если группа содержит n процессов, то номер любого процесса в данной группе лежит в пределах от 0 до $n - 1$.

Основным способом общения процессов между собой является посылка сообщений. *Сообщение* — это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и другие. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до 32767. Для работы с атрибутами сообщений введена структура **MPI_Status**, поля которой дают доступ к их значениям.

3. Общие функции MPI

Прежде чем переходить к описанию конкретных функций, сделаем несколько общих замечаний. При описании функций мы всегда будем пользоваться словом OUT для обозначения выходных параметров, через которые функция возвращает результаты. Даже если результатом работы функции является одно число, оно будет возвращено через один из параметров. Связано это с тем, что практически все функции MPI возвращают в качестве своего значения информацию об успешности завершения. В случае успешного выполнения функция вернет значение **MPI_SUCCESS**, иначе — код ошибки. Вид ошибки, которая произошла при выполнении функции, можно будет понять из описания каждой функции. Предопределенные возвращаемые значения, соответствующие различным ошибочным ситуациям, определены в файле **mpi.h**. В дальнейшем при описании конкретных функций, если ничего специально не сказано, то возвращаемое функцией значение будет подчиняться именно этому правилу.

В данном разделе мы остановимся на общих функциях MPI, необходимых практически в каждой программе.

int MPI_Init(int *argc, char ***argv)

Инициализация параллельной части программы. Все другие функции MPI могут быть вызваны только после вызова **MPI_Init**. Необычный тип аргументов **MPI_Init** предусмотрен для того, чтобы иметь возможность передать всем процессам аргументы функции **main**. Инициализация параллельной части для каждого приложения должна выполняться только один раз.

int MPI_Finalize(void)

Завершение параллельной части приложения. Все последующие обращения к любым MPI-функциям, в том числе к **MPI_Init**, запрещены. К моменту вызова **MPI_Finalize** каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Общая схема MPI-программы выглядит так:

```
main(int argc, char **argv)
{
        …
        MPI_Init(&argc, &argv);
        …
        MPI_Finalize();
        …
}
```

int MPI_Comm_size(MPI_Comm comm, int *size)

- **comm** — идентификатор коммуникатора,
- **OUT size** — число процессов в коммуникаторе **comm**.

Определение общего числа параллельных процессов в коммуникаторе **comm**. Результат возвращается через параметр **size**, для чего функции передается адрес этой переменной. Поскольку коммуникатор является сложной структурой, перед ним стоит имя предопределенного типа **MPI_Comm**, определенного в файле **mpi.h**.

int MPI_Comm_rank(MPI_Comm comm, int *rank)

- **comm** — идентификатор коммуникатора,
- **OUT rank** — номер процесса в коммуникаторе **comm**.

Определение номера процесса в коммуникаторе **comm**. Если функция **MPI_Comm_size** для того же коммуникатора **comm** вернула значение **size**, то значение, возвращаемое функцией **MPI_Comm_rank** через переменную **rank**, лежит в диапазоне от **0** до **size–1**.

double MPI_Wtime(void)

Эта функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время существования процесса. Заметим, что эта функция возвращает результат своей работы не через параметры, а явным образом.

Простейший пример программы, в которой использованы описанные выше функции, выглядит так:

```
main(int argc, char **argv)
{
      int me, size;
      …
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &me);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      printf("Process %d size %d \n", me, size);
      …
      MPI_Finalize();
      …
}
```

Строка, соответствующая функции **printf**, будет выведена столько раз, сколько процессов было порождено при вызове **MPI_Init**. Порядок появления строк заранее не определен и может быть, вообще говоря, любым. Гарантируется только то, что содержимое отдельных строк не будет перемешано друг с другом.

4. Прием/передача сообщений между отдельными процессами

Все функции передачи сообщений в MPI делятся на две группы. В одну группу входят функции, которые предназначены для взаимодействия двух процессов программы. Такие операции называются индивидуальными или операциями типа точка-точка. Функции другой группы предполагают, что в операцию должны быть вовлечены все процессы некоторого коммуникатора. Такие операции называются коллективными. Начнем описание функций обмена сообщениями с обсуждения операций типа точка-точка. Все функции данной группы, в свою очередь, так же делятся на два класса: функции с блокировкой (с синхронизацией) и функции без блокировки (асинхронные).

Прием/передача сообщений с блокировкой задаются конструкциями следующего вида.

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

- **buf** — адрес начала буфера с посылаемым сообщением;
- **count** — число передаваемых элементов в сообщении;
- **datatype** — тип передаваемых элементов;
- **dest** — номер процесса-получателя;
- **msgtag** — идентификатор сообщения;
- **comm** — идентификатор коммуникатора.

Блокирующая посылка сообщения с идентификатором **msgtag**, состоящего из **count** элементов типа **datatype**, процессу с номером **dest**. Все элементы посылаемого сообщения расположены подряд в буфере **buf**. Значение **count** может быть нулем. Разрешается передавать сообщение самому себе. Тип передаваемых элементов **datatype** должен указываться с помощью предопределенных констант типа, например, **MPI_INT**, **MPI_LONG**, **MPI_SHORT**, **MPI_LONG_DOUBLE**, **MPI_CHAR**, **MPI_UNSIGNED_CHAR**, **MPI_FLOAT** и т.п. Для каждого типа данных языков Фортран и Си есть своя константа. Полный список предопределенных имен типов можно найти в файле **mpi.h**.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Это означает, что после возврата из данной функции можно использовать любые присутствующие в вызове функции переменные без опасения испортить передаваемое сообщение. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу **dest**, остается за разработчиками конкретной реализации MPI.

Следует специально отметить, что возврат из функции **MPI_Send** не означает ни того, что сообщение получено процессом **dest**, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший **MPI_Send**. Предоставляется только гарантия безопасного изменения переменных, использованных в вызове данной функции. Подобная неопределенность далеко не всегда устраивает пользователя. Чтобы расширить возможности передачи сообщений, в MPI введены дополнительные три функции. Все параметры у этих функций такие же, как и у функции **MPI_Send**, однако у каждой из них есть своя особенность.

MPI_Bsend — передача сообщения с буферизацией. Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в буфер, и произойдет немедленный возврат из функции. Выполнение данной функции никак не зависит от соответствующего вызова функции приема сообщения. Тем не менее, функция может вернуть код ошибки, если места под буфер недостаточно.

MPI_Ssend — передача сообщения с синхронизацией. Выход из данной функции 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, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Если прием сообщения так же выполняется с

блокировкой, то функция **MPI_Ssend** сохраняет семантику блокирующих вызовов.

MPI_Rsend — передача сообщения по готовности. Данной функцией можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов функции, вообще говоря, является ошибочным и результат ее выполнения не определен. Во многих реализациях функция **MPI_Rsend** сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи.

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

- **OUT buf** — адрес начала буфера для приема сообщения;
- **count** — максимальное число элементов в принимаемом сообщении;
- **datatype** — тип элементов принимаемого сообщения;
- **source** — номер процесса-отправителя;
- **msgtag** — идентификатор принимаемого сообщения;
- **comm** — идентификатор коммуникатора;
- **OUT status** — параметры принятого сообщения.

Прием сообщения с идентификатором **msgtag** от процесса **source** с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения **count**. Если число принятых элементов меньше значения **count**, то гарантируется, что в буфере **buf** изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в принимаемом сообщении, то можно воспользоваться функцией **MPI_Get_count**. Блокировка гарантирует, что после возврата из функции все элементы сообщения уже будут приняты и расположены в буфере **buf**.

Ниже приведен пример программы, в которой нулевой процесс посылает сообщение процессу с номером один и ждет от него ответа. Если программа будет запущена с б**о**льшим числом процессов, то реально выполнять пересылки все равно станут только нулевой и первый процессы. Остальные процессы после их порождения функцией **MPI_Init** сразу завершатся, выполнив функцию **MPI_Finalize**.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
      int numtasks, rank, dest, src, rc, tag=1;
```

```
      char inmsg, outmsg='x';
      MPI_Status Stat;

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

      if (rank == 0) {
           dest = 1;
           src = 1;
           rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
           MPI_COMM_WORLD);
           rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag,
           MPI_COMM_WORLD, &Stat);
      }
      else
      if (rank == 1) {
           dest = 0;
           src = 0;
           rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag,
           MPI_COMM_WORLD, &Stat);
           rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
           MPI_COMM_WORLD);
      }
      MPI_Finalize();
}
```

В следующем примере каждый процесс с четным номером посылает сообщение своему соседу с номером на единицу большим. Дополнительно поставлена проверка для процесса с максимальным номером, чтобы он не послал сообщение несуществующему процессу. Показана только схема программы.

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char **argv)
{
      int me, size;
      int SOME_TAG=0;
      MPI_Status status;
      …
      MPI_Init (&argc, &argv);
      MPI_Comm_rank (MPI_COMM_WORLD, &me);
      MPI_Comm_size (MPI_COMM_WORLD, &size);
```

```
        if ((me % 2) == 0)  {
              if ((me+1) < size)   /* посылают все процессы, кроме
              последнего */
                    MPI_Send (…, me+1, SOME_TAG, MPI_COMM_WORLD);
        }
        else
              MPI_Recv (…, me–1, SOME_TAG, MPI_COMM_WORLD, &status);
        …
        MPI_Finalize();
  }
```

В качестве номера процесса-отправителя можно указать предопределенную константу **MPI_ANY_SOURCE** — признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу **MPI_ANY_TAG** — это признак того, что подходит сообщение с любым идентификатором. При одновременном использовании этих двух констант будет принято любое сообщение от любого процесса.

Параметры принятого сообщения всегда можно определить по соответствующим полям структуры **status**. Предопределенный тип **MPI_Status** описан в файле **mpi.h**. В языке Си параметр **status** является структурой, содержащей поля с именами **MPI_SOURCE**, **MPI_TAG** и **MPI_ERROR**. Реальные значения номера процесса-отправителя, идентификатора сообщения и кода ошибки доступны через **status.MPI_SOURCE**, **status.MPI_TAG** и **status.MPI_ERROR**. В Фортране параметр **status** является целочисленным массивом размера **MPI_STATUS_SIZE**. Константы **MPI_SOURCE**, **MPI_TAG** и **MPI_ERROR** являются индексами по данному массиву для доступа к значениям соответствующих полей, например, **status(MPI_SOURCE)**.

Обратим внимание на некоторую несимметричность операций посылки и приема сообщений. С помощью константы **MPI_ANY_SOURCE** можно принять сообщение от любого процесса. Однако в случае посылки данных требуется явно указать номер принимающего процесса.

В стандарте оговорено, что если один процесс последовательно посылает два сообщения другому процессу, и оба эти сообщения соответствуют одному и тому же вызову **MPI_Recv**, то первым будет принято сообщение, которое было отправлено раньше. Вместе с тем, если два сообщения были одновременно отправлены разными процессами, и оба сообщения соответствуют одному и тому же вызову **MPI_Recv**, то порядок их получения принимающим процессом заранее не определен.

Последнее замечание относительно использования блокирующих функций приема и посылки связано с возможным возникновением тупиковой ситуации. Предположим, что работают два параллельных процесса и они хотят обменяться данными. Было бы вполне естественно в каждом процессе сначала воспользоваться функцией **MPI_Send**, а затем **MPI_Recv**. Но именно этого и не стоит делать. Дело в том, что мы заранее не знаем, как реализована функция **MPI_Send**. Если разработчики для гарантии корректного повторного использования буфера посылки заложили схему, при которой посылающий процесс ждет начала приема, то возникнет классический тупик. Первый процесс не может вернуться из функции посылки, поскольку второй не начинает прием сообщения. А второй процесс не может начать прием сообщения, поскольку сам по похожей причине застрял на посылке. Выход из этой ситуации прост. Нужно использовать либо неблокирующие функции приема/передачи, либо функцию совмещенной передачи и приема. Оба варианта мы обсудим ниже.

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

- **status** — параметры принятого сообщения;
- **datatype** — тип элементов принятого сообщения;
- **OUT count** — число элементов сообщения.

По значению параметра **status** данная функция определяет число уже принятых (после обращения к **MPI_Recv**) или принимаемых (после обращения к **MPI_Probe** или **MPI_Iprobe**) элементов сообщения типа **datatype**. Данная функция, в частности, необходима для определения размера области памяти, выделяемой для хранения принимаемого сообщения.

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 коммуникатора;
- **OUT status** — параметры найденного подходящего сообщения.

Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из функции не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра **status**. Следует особо обратить внимание на то, что функция определяет только факт прихода сообщения, но реально его не принимает.

Прием/передача сообщений без блокировки. В отличие от функций с блокировкой, возврат из функций данной группы происходит сразу без какой-либо блокировки процессов. На фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной операции. В принципе, данная возможность исключительно полезна для создания эффективных программ. В самом деле, программист знает, что в некоторый момент ему потребуется массив, который вычисляет другой процесс. Он заранее выставляет в программе асинхронный запрос на получение данного массива, а до того момента, когда массив реально потребуется, он может выполнять любую другую полезную работу. Опять же, во многих случаях совершенно не обязательно дожидаться окончания посылки сообщения для выполнения последующих вычислений.

Если есть возможность операции приема/передачи сообщений скрыть на фоне вычислений, то этим, вроде бы, надо безоговорочно пользоваться. Однако на практике не все согласуется с теорией. Многое зависит от конкретной реализации. К сожалению, далеко не всегда асинхронные операции эффективно поддерживаются аппаратурой и системным окружением. Поэтому не стоит удивляться, если эффект от выполнения вычислений на фоне пересылок окажется нулевым. Сделанные замечания касаются только вопросов эффективности. В отношении предоставляемой функциональности асинхронные операции исключительно полезны, поэтому они присутствуют практически в каждой реальной программе.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
int msgtag, MPI_Comm comm, MPI_Request *request)
```

- **buf** — адрес начала буфера с посылаемым сообщением;
- **count** — число передаваемых элементов в сообщении;
- **datatype** — тип передаваемых элементов;
- **dest** — номер процесса-получателя;
- **msgtag** — идентификатор сообщения;
- **comm** — идентификатор коммуникатора,
- **OUT request** — идентификатор асинхронной операции.

Передача сообщения аналогична вызову **MPI_Send**, однако возврат из функции **MPI_Isend** происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере **buf**. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации, подтверждающей завершение данной посылки. Определить тот момент времени, когда можно повторно использовать буфер **buf** без опасения испортить передаваемое сообщение, можно с помощью параметра **request** и функций **MPI_Wait** и **MPI_Test**.

Аналогично трем модификациям функции **MPI_Send**, предусмотрены три дополнительных варианта функций **MPI_Ibsend**, **MPI_Issend**, **MPI_Irsend**. К изложенной выше семантике работы этих функций добавляется асинхронность.

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

- **OUT buf** — адрес начала буфера для приема сообщения;
- **count** — максимальное число элементов в принимаемом сообщении;
- **datatype** — тип элементов принимаемого сообщения;
- **source** — номер процесса-отправителя;
- **msgtag** — идентификатор принимаемого сообщения;
- **comm** — идентификатор коммуникатора;
- **OUT request** — идентификатор операции асинхронного приема сообщения.

Прием сообщения, аналогичный **MPI_Recv**, однако возврат из функции происходит сразу после инициализации процесса приема без ожидания получения всего сообщения и его записи в буфере **buf**. Окончание процесса приема можно определить с помощью параметра **request** и процедур типа **MPI_Wait** и **MPI_Test**.

Сообщение, отправленное любой из функций **MPI_Send**, **MPI_Isend** и любой из трех их модификаций, может быть принято любой из процедур **MPI_Recv** и **MPI_Irecv**.

С помощью данной функции легко обойти возможную тупиковую ситуацию, о которой говорилось ранее. Заменим вызов функции приема сообщения с блокировкой на вызов функции **MPI_Irecv**. Расположим его перед вызовом функции **MPI_Send**, т.е. преобразуем фрагмент следующим образом:

```
…                                  …
MPI_Send (…)          ──►          MPI_Irecv (…)
MPI_Recv(…)                        MPI_Send (…)
…                                  …
```

В такой ситуации тупик гарантированно не возникнет, поскольку к моменту вызова функции **MPI_Send** запрос на прием сообщения уже будет выставлен.

int MPI_Wait(MPI_Request *request, MPI_Status *status)

- **request** — идентификатор операции асинхронного приема или передачи;
- **OUT status** — параметры сообщения.

Ожидание завершения асинхронной операции, ассоциированной с идентификатором **request** и запущенной функцией **MPI_Isend** или **MPI_Irecv**. Пока асин-

хронная операция не будет завершена, процесс, выполнивший функцию **MPI_Wait**, будет заблокирован. Если речь идет о приеме, атрибуты и длину принятого сообщения можно определить обычным образом с помощью параметра **status**.

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)

- **count** — число идентификаторов асинхронных операций;
- **requests** — идентификаторы операций асинхронного приема или передачи;
- **OUT statuses** — параметры сообщений.

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива **statuses** будет установлено в соответствующее значение.

Ниже показан пример программы, в которой все процессы обмениваются сообщениями с ближайшими соседями в соответствии с топологией кольца.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
      int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
      MPI_Request reqs[4];
      MPI_Status stats[4];

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

      prev = rank - 1;
      next = rank + 1;
      if (rank == 0) prev = numtasks - 1;
      if (rank == (numtasks - 1)) next = 0;

      MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD,
      &reqs[0]);
      MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD,
      &reqs[1]);
```

```
        MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD,
        &reqs[2]);
        MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD,
        &reqs[3]);

        MPI_Waitall(4, reqs, stats);
        MPI_Finalize();
    }
```

int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status)

- **count** — число идентификаторов асинхронных операций;
- **requests** — идентификаторы операций асинхронного приема или передачи;
- **OUT index** — номер завершенной операции обмена;
- **OUT status** — параметры сообщений.

Выполнение процесса блокируется до тех пор, пока какая-либо асинхронная операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если завершились несколько операций, то случайным образом будет выбрана одна из них. Параметр **index** содержит номер элемента в массиве **requests**, содержащего идентификатор завершенной операции.

int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

- **incount** — число идентификаторов асинхронных операций;
- **requests** — идентификаторы операций асинхронного приема или передачи;
- **OUT outcount** — число идентификаторов завершившихся операций обмена;
- **OUT indexes** — массив номеров завершившихся операции обмена;
- **OUT statuses** — параметры завершившихся сообщений.

Выполнение процесса блокируется до тех пор, пока хотя бы одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр **outcount** содержит число завершенных операций, а первые **outcount** элементов массива **indexes** содержат номера элементов массива **requests** с их идентификаторами. Первые **outcount** элементов массива **statuses** содержат параметры завершенных операций.

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

- **request** — идентификатор операции асинхронного приема или передачи;

- **OUT flag** — признак завершенности операции обмена;
- **OUT status** — параметры сообщения.

Проверка завершенности асинхронных функций **MPI_Isend** или **MPI_Irecv**, ассоциированных с идентификатором **request**. В параметре **flag** функция **MPI_Test** возвращает значение **1**, если соответствующая операция завершена, и значение **0** в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра **status**.

int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses)

- **count** — число идентификаторов асинхронных операций;
- **requests** — идентификаторы операций асинхронного приема или передачи;
- **OUT flag** — признак завершенности операций обмена;
- **OUT statuses** — параметры сообщений.

В параметре **flag** функция возвращает значение **1**, если все операции, ассоциированные с указанными идентификаторами, завершены. В этом случае параметры сообщений будут указаны в массиве **statuses**. Если какая-либо из операций не завершилась, то возвращается **0**, и определенность элементов массива **statuses** не гарантируется.

int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)

- **count** — число идентификаторов асинхронных операций;
- **requests** — идентификаторы операций асинхронного приема или передачи;
- **OUT index** — номер завершенной операции обмена;
- **OUT flag** — признак завершенности операции обмена;
- **OUT status** — параметры сообщения.

Если к моменту вызова функции **MPI_Testany** хотя бы одна из операций асинхронного обмена завершилась, то в параметре **flag** возвращается значение **1**, **index** содержит номер соответствующего элемента в массиве **requests**, а **status** — параметры сообщения. В противном случае в параметре **flag** будет возвращено значение **0**.

int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

- **incount** — число идентификаторов асинхронных операций;

- **requests** — идентификаторы операций асинхронного приема или передачи;
- **OUT outcount** — число идентификаторов завершившихся операций обмена;
- **OUT indexes** — массив номеров завершившихся операции обмена;
- **OUT statuses** — параметры завершившихся операций.

Данная функция работает так же, как и **MPI_Waitsome**, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение **outcount** будет равно нулю.

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

- **source** — номер процесса-отправителя или **MPI_ANY_SOURCE**;
- **msgtag** — идентификатор ожидаемого сообщения или **MPI_ANY_TAG**;
- **comm** — идентификатор коммуникатора;
- **OUT flag** — признак завершенности операции обмена;
- **OUT status** — параметры подходящего сообщения.

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре **flag** возвращается значение **1**, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично **MPI_Probe**), и значение **0**, если сообщения с указанными атрибутами еще нет.

Объединение запросов на взаимодействие. Процедуры данной группы позволяют снизить накладные расходы, возникающие в рамках одного процессора при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером. Несколько запросов на прием и/или передачу могут объединяться вместе для того, чтобы далее их можно было бы запустить одной командой. Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

int MPI_Send_init(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

- **buf** — адрес начала буфера с посылаемым сообщением;
- **count** — число передаваемых элементов в сообщении;
- **datatype** — тип передаваемых элементов;
- **dest** — номер процесса-получателя;

- **msgtag** — идентификатор сообщения;
- **comm** — идентификатор коммуникатора;
- **OUT request** — идентификатор асинхронной передачи.

Формирование запроса на выполнение пересылки данных. Все параметры точно такие же, как и у подпрограммы **MPI_Isend**, однако в отличие от нее пересылка не начинается до вызова подпрограммы **MPI_Startall**. Как и прежде, дополнительно предусмотрены варианты и для трех модификаций посылки сообщений: **MPI_Bsend_init**, **MPI_Ssend_init**, **MPI_Rsend_init**.

```
int MPI_Recv_init( void *buf, int count, MPI_Datatype datatype, int
source,  int msgtag, MPI_Comm comm, MPI_Request *request)
```

- **OUT buf** — адрес начала буфера приема сообщения;
- **count** — число принимаемых элементов в сообщении;
- **datatype** — тип принимаемых элементов;
- **source** — номер процесса-отправителя;
- **msgtag** — идентификатор сообщения;
- **comm** — идентификатор коммуникатора;
- **OUT request** — идентификатор асинхронного приема.

Формирование запроса на выполнение приема сообщения. Все параметры точно такие же, как и у функции **MPI_Irecv**, однако в отличие от нее реальный прием не начинается до вызова функции **MPI_Startall**.

```
MPI_Startall(int count, MPI_Request *requests)
```

- **count** — число запросов на взаимодействие;
- **OUT requests** — массив идентификаторов приема/передачи

Запуск всех отложенных операций передачи и приема, ассоциированных с элементами массива запросов **requests** и инициированных функциями **MPI_Recv_init**, **MPI_Send_init** или ее тремя модификациями. Все отложенные взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью функций семейств **MPI_Wait** и **MPI_Test**.

Совмещенные прием и передача сообщений. Совмещение приема и передачи сообщений между процессами позволяет легко обходить множество подводных камней, связанных с возможными тупиковыми ситуациями. Предположим, что в линейке процессов необходимо организовать обмен данными между **i**-м и **(i+1)**-ым процессами. Если воспользоваться стандартными блокирующими функциями посылки сообщений, то возможен тупик, обсуждавшийся ранее.

Один из способов обхода такой ситуации состоит в использовании функции совмещенного приема и передачи.

```
int MPI_Sendrecv( void *sbuf, int scount, MPI_Datatype stype,  int
dest, int stag, void *rbuf, int rcount,  MPI_Datatype rtype, int
source, MPI_Datatype rtag,  MPI_Comm comm, MPI_Status *status)
```

- **sbuf** — адрес начала буфера с посылаемым сообщением;
- **scount** — число передаваемых элементов в сообщении;
- **stype** — тип передаваемых элементов;
- **dest** — номер процесса-получателя;
- **stag** — идентификатор посылаемого сообщения;
- **OUT rbuf** — адрес начала буфера приема сообщения;
- **rcount** — число принимаемых элементов сообщения;
- **rtype** — тип принимаемых элементов;
- **source** — номер процесса-отправителя;
- **rtag** — идентификатор принимаемого сообщения;
- **comm** — идентификатор коммуникатора;
- **OUT status** — параметры принятого сообщения.

Данная операция объединяет в едином запросе посылку и прием сообщений. Естественно, что реализация этой функции гарантирует отсутствие тупиков, которые могли бы возникнуть между процессами при использовании обычных блокирующих операций **MPI_Send** и **MPI_Recv**. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Буфера приема и посылки обязательно должны быть различными. Сообщение, отправленное операцией **MPI_Sendrecv**, может быть принято обычным образом, и точно также операция **MPI_Sendrecv** может принять сообщение, отправленное обычной операцией **MPI_Send**.

5. Задания:

- Начните создавать параллельный вариант программы, реализующей метод Гаусса решения систем линейных алгебраических уравнений, используя изученные функции MPI.
- Пинг-понг. Смоделировать последовательный обмен сообщениями между двумя процессами, замерить время на одну итерацию обмена, определить зависимость времени от длины сообщения.
- Сравнить эффективность реализации различных видов пересылок данных между двумя выделенными процессорами.
- Можно ли в процессе работы MPI-программы порождать новые процессы, если освободились свободные процессоры?

- Могут ли различные процессы MPI-программы записывать одновременно: а) на один и тот же диск? б) в один и тот же файл?
- Может ли MPI-программа продолжать работу после аварийного завершения одного из процессов?
- Могут ли группы процессов иметь непустое пересечение, не совпадающее ни с одной из них полностью?
- Можно ли сообщение, отправленное с помощью блокирующей операции посылки, принять неблокирующей операцией приема?
- Что гарантирует блокировка при отправке/приеме сообщений?
- Как принять любое сообщение от любого процесса?

Занятие 5. Технология MPI (продолжение)

1. Коллективные взаимодействия процессов

В операциях коллективного взаимодействия процессов *участвуют все процессы коммуникатора*. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки. Асинхронных коллективных операций в MPI нет.

В коллективных операциях можно использовать те же коммуникаторы, что и были использованы для операций типа точка-точка. MPI гарантирует, что сообщения, вызванные коллективными операциями, никак не повлияют на выполнение других операций и не пересекутся с сообщениями, появившимися в результате индивидуального взаимодействия процессов.

Вообще говоря, нельзя рассчитывать на синхронизацию процессов с помощью коллективных операций. Если какой-то процесс уже завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммуникатора, ни даже того, что она ими начата (конечно же, если это возможно по смыслу операции).

В коллективных операциях не используются идентификаторы сообщений.

`int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)`

- **`OUT buf`** — адрес начала буфера посылки сообщения;
- **`count`** — число передаваемых элементов в сообщении;

- **datatype** — тип передаваемых элементов;
- **source** — номер рассылающего процесса;
- **comm** — идентификатор коммуникатора.

Рассылка сообщения от процесса **source** всем процессам данного коммуникатора, включая рассылающий процесс. При возврате из процедуры содержимое буфера **buf** процесса **source** будет скопировано в локальный буфер каждого процесса коммуникатора **comm**. Значения параметров **count**, **datatype**, **source** и **comm** должны быть одинаковыми у всех процессов. В результате выполнения следующего оператора всеми процессами коммуникатора **comm**:

```
MPI_Bcast(array, 100, MPI_INT, 0, comm);
```

первые сто целых чисел из массива **array** нулевого процесса будут скопированы в локальные буфера **array** каждого процесса коммуникатора **comm**.

```
int MPI_Gather( void *sbuf, int scount, MPI_Datatype stype,  void
*rbuf, int rcount, MPI_Datatype rtype,  int dest, MPI_Comm comm)
```

- **sbuf** — адрес начала буфера посылки;
- **scount** — число элементов в посылаемом сообщении;
- **stype** — тип элементов отсылаемого сообщения;
- **OUT rbuf** — адрес начала буфера сборки данных;
- **rcount** — число элементов в принимаемом сообщении;
- **rtype** — тип элементов принимаемого сообщения;
- **dest** — номер процесса, на котором происходит сборка данных;
- **comm** — идентификатор коммуникатора.

Сборка данных со всех процессов в буфере **rbuf** процесса **dest**. Каждый процесс, включая **dest**, посылает содержимое своего буфера **sbuf** процессу **dest**. Собирающий процесс сохраняет данные в буфере **rbuf**, располагая их в порядке возрастания номеров процессов. На процессе **dest** существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров **sbuf**, **scount**, **stype**, **dest** и **comm**. Значения параметров **dest** и **comm** должны быть одинаковыми у всех процессов. Параметр **rcount** у процесса **dest** обозначает число элементов типа **rtype**, принимаемых не от всех процессов в сумме, а от каждого процесса. С помощью похожей функции **MPI_Gatherv** можно принимать от процессов массивы данных различной длины.

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype, void
*rbuf, int rcount, MPI_Datatype rtype, int source, MPI_Comm comm)
```

- **sbuf** — адрес начала буфера посылки;
- **scount** — число элементов в посылаемом сообщении;

- **stype** — тип элементов отсылаемого сообщения;
- **OUT rbuf** — адрес начала буфера сборки данных;
- **rcount** — число элементов в принимаемом сообщении;
- **rtype** — тип элементов принимаемого сообщения;
- **source** — номер процесса, на котором происходит сборка данных;
- **comm** — идентификатор коммуникатора.

Функция **MPI_Scatter** по своему действию является обратной к **MPI_Gather**. Процесс **source** рассылает порции данных из массива **sbuf** всем **n** процессам приложения. Можно считать, что массив **sbuf** делится на **n** равных частей, состоящих из **scount** элементов типа **stype**, после чего **i**-я часть посылается **i**-му процессу. На процессе **source** существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров **rbuf**, **rcount**, **rtype**, **source** и **comm**. Значения параметров **source** и **comm** должны быть одинаковыми у всех процессов. Аналогично функции **MPI_Gatherv**, с помощью функции **MPI_Scatterv** процессам можно отослать порции данных различной длины.

В следующем примере показано использование функции **MPI_Scatter** для рассылки строк массива. Напомним, что в языке Си, в отличие от Фортрана, массивы хранятся в памяти по строкам.

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4
int main(int argc, char **argv)
{
      int numtasks, rank, sendcount, recvcount, source;
      float sendbuf[SIZE][SIZE] = {
            {1.0, 2.0, 3.0, 4.0},
            {5.0, 6.0, 7.0, 8.0},
            {9.0, 10.0, 11.0, 12.0},
            {13.0, 14.0, 15.0, 16.0}  };
      float recvbuf[SIZE];

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

      if (numtasks == SIZE) {
            source = 1;
            sendcount = SIZE;
```

```
            recvcount = SIZE;
            MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,
            recvcount,
            MPI_FLOAT,source,MPI_COMM_WORLD);

            printf("rank= %d  Results: %f %f %f %f\n", rank,
            recvbuf[0],recvbuf[1],recvbuf[2],recvbuf[3]);
        }
        else
            printf("Число процессов должно быть равно %d. \n",
            SIZE);

        MPI_Finalize();
    }
```

К коллективным операциям относятся и *редукционные операции*. Такие операции предполагают, что на каждом процессе хранятся некоторые данные, над которыми необходимо выполнить единую операцию, например, операцию сложения чисел или операцию нахождения максимального значения. Операция может быть либо предопределенной операцией MPI, либо операцией, определенной пользователем. Каждая предопределенная операция имеет свое имя, например, **MPI_MAX** - глобальный максимум, **MPI_MIN** – глобальный минимум, **MPI_SUM** – глобальная сумма, **MPI_PROD** – глобальное произведение и т.п.

int MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- **sbuf** — адрес начала буфера для аргументов операции **op**;
- **OUT rbuf** — адрес начала буфера для результата операции **op**;
- **count** — число аргументов у каждого процесса;
- **datatype** — тип аргументов;
- **op** — идентификатор глобальной операции;
- **comm** — идентификатор коммуникатора.

Данная функция задает выполнение **count** независимых глобальных операций **op**. Предполагается, что в буфере **sbuf** каждого процесса расположено **count** аргументов, имеющих тип **datatype**. Первые элементы массивов **sbuf** участвуют в первой операции **op**, вторые элементы массивов **sbuf** участвуют во второй операции **op** и т.д. Результаты выполнения всех **count** операций записываются в буфер **rbuf** на каждом процессе. Значения параметров **count**, **datatype**, **op** и **comm** у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция **op** обладает свойствами ассоциативности и коммутативности.

```
int MPI_Reduce( void *sbuf, void *rbuf, int count,  MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm)
```

- **sbuf** — адрес начала буфера для аргументов;
- **OUT rbuf** — адрес начала буфера для результата;
- **count** — число аргументов у каждого процесса;
- **datatype** — тип аргументов;
- **op** — идентификатор глобальной операции;
- **root** — процесс-получатель результата;
- **comm** — идентификатор коммуникатора.

Функция аналогична предыдущей, но результат операции будет записан в буфер **rbuf** не у всех процессов, а только у процесса **root**.

2. Синхронизация процессов

Синхронизация процессов в MPI осуществляется с помощью единственной функции **MPI_Barrier**.

```
int MPI_Barrier(MPI_Comm comm)
```

- **comm** — идентификатор коммуникатора.

Функция блокирует работу вызвавших ее процессов до тех пор, пока все оставшиеся процессы коммуникатора **comm** также не выполнят эту процедуру. Только после того, как последний процесс коммуникатора выполнит данную функцию, все процессы будут разблокированы и продолжат выполнение дальше. Данная функция является коллективной. Все процессы должны вызвать **MPI_Barrier**, хотя реально исполненные вызовы различными процессами коммуникатора могут быть расположены в разных местах программы.

3. Работа с группами процессов

Несмотря на то, что в MPI есть значительное множество функций, ориентированных на работу с коммуникаторами и группами процессов, мы не будем подробно останавливаться на данном разделе. Представленная функциональность позволяет сравнить состав групп, определить их пересечение, объединение, добавить процессы в группу, удалить группу и т. д. В качестве примера приведем лишь один способ образования новых групп на основе существующих.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm
*newcomm)
```

- **comm** — идентификатор существующего коммуникатора;

- **color** — признак разделения на группы;
- **key** — параметр, определяющий нумерацию в новых группах;
- **OUT newcomm** — идентификатор нового коммуникатора.

Данная процедура разбивает все множество процессов, входящих в группу **comm**, на непересекающиеся подгруппы — одну подгруппу на каждое значение параметра **color**. Значение параметра **color** должно быть неотрицательным целым числом. Каждая новая подгруппа содержит все процессы, у которых параметр **color** имеет одно и тоже значение, т.е. в каждой новой подгруппе будут собраны все процессы “одного цвета”. Всем процессам подгруппы будет возвращено одно и тоже значение **newcomm**.

Предположим, что нужно разделить все процессы программы на две подгруппы в зависимости от того, является ли номер процесса четным или нечетным. В этом случае, в поле **color** достаточно поместить выражение **My_Id%2**, где **My_Id** — это номер процесса. Значением данного выражения может быть либо **0**, либо **1**. Все процессы с четными номерами автоматически попадут в одну группу, а процессы с нечетными в другую.

int MPI_Comm_free(MPI_Comm comm)

- **OUT comm** — идентификатор коммуникатора.

Появление нового коммуникатора всегда вызывает создание новой структуры данных. Если созданный коммуникатор больше не нужен, то соответствующую область памяти необходимо освободить. Данная функция уничтожает коммуникатор, ассоциированный с идентификатором **comm**, который после возвращения из функции будет иметь предопределенное значение **MPI_COMM_NULL**.

6. Задания:

- Получить работающий параллельный вариант программы, реализующей метод Гаусса решения систем линейных алгебраических уравнений, исследовать его эффективность, предложить варианты возможных улучшений.
- Верно ли, что в коллективных взаимодействиях участвуют все процессы приложения?
- Могут ли возникать конфликты между сообщениями, посылаемыми процессами друг другу, и сообщениями коллективных операций? Если да, как они разрешаются?
- Смоделировать барьерную синхронизацию при помощи пересылок точка-точка и сравнить эффективность такой реализации и стандартной функции.

- Смоделировать глобальное суммирование методом сдваивания и сравнить эффективность такой реализации с функцией **`MPI_Reduce`**.
- Напишите свой вариант процедуры **`Mpi_Gather`**, используя функции посылки сообщений типа точка-точка.
- Подумайте, как организовать коллективный асинхронный обмен данными, аналогичный функции: а) **`MPI_Reduce`**; б) **`MPI_AlltoAll`**.
- Подумайте, какая характеристика сетевой среды больше всего влияет на время выполнения барьерной синхронизации.
- Ознакомьтесь со стандартом MPI-2.0 и назовите его ключевые отличия от MPI-1.1.

Занятие 6. Технологии параллельного программирования (обзор)

1. Спецкомментарии

Начнем с использования в традиционных языках программирования *специальных комментариев*, добавляющих “параллельную” специфику в изначально последовательные программы. Предположим, что вы работаете на векторно-конвейерном компьютере Cray T90. Вы знаете, что все итерации некоторого цикла программы независимы и, следовательно, его можно векторизовать, т.е. очень эффективно исполнить с помощью векторных команд на конвейерных функциональных устройствах. Если цикл простой, то компилятор и сам определит возможность преобразования последовательного кода в параллельный. Если уверенности в высоком интеллекте компилятора нет, то перед заголовком цикла лучше вставить явное указание на отсутствие зависимости и возможность векторизации. В частности, для языка Фортран это выглядит так:

```
CDIR$ NODEPCHK
```

По правилу Фортрана, буква '**`C`**' в первой позиции говорит о том, что вся строка является комментарием, последовательность '**`DIR$`**' указывает на то, что это спецкомментарий для компилятора, а часть '**`NODEPCHK`**' как раз и говорит об отсутствии информационной зависимости между итерациями последующего цикла.

Следует отметить, что использование спецкомментариев не только добавляет возможность параллельного исполнения, но и полностью сохраняет исходный вариант программы. На практике это очень удобно — если компилятор ничего не знает о параллелизме, то все спецкомментарии он просто пропустит, взяв за основу последовательную семантику программы.

На использование комментариев опирается и широко распространенный в настоящее время стандарт **Open_MP**. Основная ориентация сделана на работу с общей памятью, нитями (threads) и явным описанием параллелизма. В Фортране признаком спецкомментария Open_MP является префикс **`!$OMP`**, а в языке Си используют директиву **`#pragma omp`**. В настоящее время практически все ведущие производители SMP-компьютеров поддерживают Open_MP в компиляторах на своих платформах.

2. Расширения существующих языков программирования

Кроме использования комментариев для получения параллельной программы, часто идут на *расширение существующих языков программирования.* Вводятся дополнительные операторы и новые элементы описания переменных, позволяющие пользователю явно задавать параллельную структуру программы и в некоторых случаях управлять исполнением параллельной программы. Так язык **High Performance Fortran (HPF)**, помимо традиционных операторов Фортрана и системы спецкомментариев, содержит новый оператор **`FORALL`**, введенный для описания параллельных циклов программы. Наиболее интересной чертой HPF представляется многоуровневое отображение массив - массив-шаблон - виртуальный процессорный массив - физические процессоры, позволяющее максимально гибко отображать пользовательские данные на реальный компьютер. На вычислительном кластере НИВЦ МГУ установлена система **HPF Adaptor**, переводящая программу с HPF на Fortran с MPI.

Другим примером служит язык **mpC**, разработанный в Институте системного программирования РАН как расширение ANSI C. Основное назначение mpC — создание эффективных параллельных программ для неоднородных вычислительных систем. Пользователь может задать топологию сети, распределение данных и вычислений и необходимые пересылки данных. Посылка сообщений организована с использованием интерфейса MPI.

DVM-система предназначена для создания переносимых и эффективных вычислительных приложений на языках C-DVM и Fortran-DVM для параллельных компьютеров с различной архитектурой. Аббревиатура DVM соответствует двум понятиям: Distributed Virtual Memory и Distributed Virtual Machine. Первое отражает наличие единого адресного пространства. Второе отражает использование виртуальных машин для двухступенчатой схемы отображения данных и вычислений на реальную параллельную машину. Модель программирования предполагает задание DVM-указаний с помощью спецкомментариев, а значит, один вариант программы для последовательного и параллельного исполнения. Поддерживаются три группы директив: директивы распределения данных, директивы распределения вычислений и спецификации удаленных данных. Компилятор переводит программу на язык Фортран или Си, используя для организации межпроцессорного взаимодействия одну из существующих технологий

параллельного программирования (MPI, PVM, Router). В систему DVM также входят библиотека поддержки LIB-DVM, DVM-отладчик, предсказатель выполнения DVM-программ, анализатор производительности DVM-программ. Система разработана в Институте прикладной математики им. М.В.Келдыша РАН.

3. Специальные языки программирования

Если нужно точнее отразить либо специфику архитектуры параллельных систем, либо свойства какого-то класса задач некоторой предметной области, то используют *специальные языки параллельного программирования.* Для программирования транспьютерных систем был создан язык **Occam**, для программирования потоковых машин был спроектирован язык однократного присваивания **Sisal**. Очень интересной и оригинальной разработкой является декларативный язык **НОРМА**, созданный под руководством И.Б.Задыхайло в Институте прикладной математики им. М.В.Келдыша РАН для описания решения вычислительных задач сеточными методами. Высокий уровень абстракции языка позволяет описывать задачи в нотации, близкой к исходной постановке проблемы математиком, что условно авторы языка называют программированием без программиста. Язык с однократным присваиванием, не содержит традиционных конструкций языков программирования, фиксирующих порядок вычисления и тем самым скрывающих естественный параллелизм алгоритма.

4. Библиотеки и интерфейсы, поддерживающие взаимодействие параллельных процессов

С появлением массивно-параллельных компьютеров широкое распространение получили *библиотеки и интерфейсы, поддерживающие взаимодействие параллельных процессов*. Типичным представителем данного направления является интерфейс **Message Passing Interface (MPI)**, реализация которого есть практически на каждой параллельной платформе, начиная от векторно-конвейерных супер-ЭВМ до кластеров и сетей персональных компьютеров. Программист сам явно определяет какие параллельные процессы приложения в каком месте программы и с какими процессами должны либо обмениваться данными, либо синхронизировать свою работу. Обычно адресные пространства параллельных процессов различны. В частности, такой идеологии следуют MPI и **PVM**. В других технологиях, например **Shmem**, допускается использование как локальных (private) переменных, так и общих (shared) переменных, доступных всем процессам приложения, и реализуется схема работы над общей памятью с помощью операций типа Put/Get.

5. Linda

Несколько особняком стоит система **Linda**, добавляющая в любой последовательный язык лишь четыре дополнительные функции **`in`**, **`out`**, **`read`** и **`eval`**, что и позволяет создавать параллельные программы. К сожалению, простота заложенной идеи оборачивается большими проблемами в реализации, что делает данную красивую технологию скорее объектом академического интереса, чем практическим инструментом.

6. Параллельные предметные библиотеки

Часто на практике прикладные программисты вообще не используют никаких явных параллельных конструкций, обращаясь в критических по времени счета фрагментах к подпрограммам и функциям *параллельных предметных библиотек*. Весь параллелизм и вся оптимизация спрятаны в вызовах, а пользователю остается лишь написать внешнюю часть своей программы и грамотно воспользоваться стандартными блоками. Примерами подобных библиотек являются **Lapack**, **ScaLapack**, **Cray Scientific Library**, **HP Mathematical Library**, **PETSc** и многие другие.

Параллельные предметные библиотеки, реализации которых установлены на вычислительном кластере НИВЦ МГУ:

- **BLAS** и **LAPACK** – библиотеки, реализующие базовые операции линейной алгебры, такие как перемножение матриц, умножение матрицы на вектор и т.д.
- **ScaLAPACK** включает подмножество процедур LAPACK, переработанных для использования на MPP-компьютерах, включая: решение систем линейных уравнений, обращение матриц, ортогональные преобразования, поиск собственных значений и др.
- **FFTW**, **DFFTPack** – быстрое преобразование Фурье.
- **PETSc** - набор процедур и структур данных для параллельного решения научных задач с моделями, описываемыми в виде дифференциальных уравнений с частными производными.

7. Специализированные пакеты и программные комплексы

И, наконец, последнее направление, о котором стоит сказать, это *использование специализированных пакетов и программных комплексов*. Как правило, в этом случае пользователю вообще не приходится программировать. Основная задача — это правильно указать все необходимые входные данные и правильно воспользоваться функциональностью пакета. Так, многие химики для выполнения квантово-химических расчетов на параллельных компьютерах пользуются пакетом **GAMESS**, не задумываясь о том, каким образом реализована параллельная обработка данных в самом пакете.

8. Задания:

- Проверить эффективность различных модификаций программы, реализующей метод Гаусса решения систем линейных алгебраических уравнений.
- Назовите преимущества и недостатки параллельных расширений языков программирования, использующих спецкомментарии.
- Можно ли реализовать базовые процедуры MPI посредством Linda (включая асинхронные процедуры и глобальные операции)?
- Почему возможности mpC не были реализованы в виде библиотеки, как MPI?
- Почему использование технологии MPI на компьютерах с общей памятью чаще всего оказывается не самым эффективным вариантом?
- Могут ли на одном и том же компьютере использоваться сразу все из описанных выше технологий?
- Выберите технологию, наиболее подходящую для реализации Вашей задачи. Обоснуйте свой выбор.

Занятие 7. Технологии построения суперкомпьютеров. Кластерные технологии (обзор)

1. Производительность параллельных компьютеров

Сначала коротко постараемся ответить на вопрос, что можно считать суперкомпьютером. В разное время пытались дать разное определение данного понятия, но мы остановимся на наиболее простом: к классу супер-ЭВМ принадлежат лишь те компьютеры, которые имеют максимальную производительность в настоящее время.

Производительность компьютеров обычно измеряют в количестве производимых ими операций в единицу времени. Если в качестве операций берутся операции с плавающей запятой, то единицей производительности считается FLOPS (Floating point Operations per Second), то есть число таких операций в секунду. Поскольку мощность современных компьютеров весьма велика, обычно используют производные единицы: MFLOPS (миллион операций с плавающей запятой в секунду), GFLOPS (миллиард операций в секунду) и TFLOPS (триллион операций в секунду).

Естественной характеристикой любого компьютера является его *пиковая производительность*. Данное значение определяет тот максимум, на который теоре-

тически способен компьютер. Вычисляется оно очень просто. Для этого достаточно предположить, что все устройства компьютера работают в максимально производительном режиме. Если в процессоре есть два конвейерных устройства, то рассматривается режим, когда оба конвейера одновременно работают с максимальной нагрузкой. Если в компьютере есть 1000 таких процессоров, то пиковая производительность одного процессора просто умножается на 1000. Иногда пиковую производительность компьютера называют его теоретической производительностью. Это нюанс в названии лишний раз подчеркивает тот факт, что производительность компьютера на любой реальной программе никогда не только не превысит этого порога, но и не достигнет его точно.

Пиковая производительность компьютера вычисляется однозначно и подсознательно всегда возникает связь между пиковой производительностью компьютера и его возможностями в решении задач. Чем больше пиковая производительность, тем, вроде бы, быстрее пользователь сможет решить свою задачу. Однако с момента появления первых параллельных компьютеров пользователи убедились, что разброс в значениях реальной производительности может быть огромным. На одних задачах удавалось получать 90% от пиковой производительности, а на других лишь 2%. Например, если кто-то мог использовать независимость и конвейерность всех функциональных устройств компьютера CDC 7600, то производительность получалась высокой. Если в вычислениях были информационные зависимости, то конвейерность не использовалась, и производительность снижалась. Если в алгоритме явно преобладал один тип операций, то часть устройств простаивала, вызывая дальнейшее падение производительности.

Для того чтобы оценить реальную производительность компьютера, нужно отойти от характеристик аппаратуры и оценить эффективность работы программно-аппаратной среды на фиксированном наборе задач. Бессмысленно для каждого компьютера показывать свой набор. Разработчики без труда придумают такие программы, на которых их компьютер достигает производительности, близкой к пиковой. Такие примеры никого не убедят. Набор тестовых программ должен быть зафиксирован. Эти программы будут играть роль эталона, по которому будут судить о возможностях вычислительной системы.

Такие попытки неоднократно предпринимались. На основе различных критериев формировались тестовые наборы программ или фиксировались отдельные эталонные программы (такие программы иногда называют *бенчмарками*, отталкиваясь от английского слова “benchmark”). Программы запускались на различных системах, замерялись те или иные параметры, на основе которых в последствии проводилось сравнение компьютеров между собой. В дальнейшем изложении для подобных программ мы чаще всего будем использовать слово тест, делая акцент не на проверке правильности работы чего-либо, а на тестировании эффективности работы вычислительной системы.

Что имело бы смысл взять в качестве подобного теста? Что-то не сложное и известное всем. Одним из таких примеров стал известный тест LINPACK. Данный тест является программой решения системы линейных алгебраических уравнений с плотной матрицей с выбором главного элемента по строке. Простой алгоритм, регулярные структуры данных, значительная вычислительная емкость, возможность получения показателей производительности близких к пиковым — все эти черты сделали тест исключительно популярным.

Поскольку никакое одно число или даже набор чисел не будут универсальной характеристикой производительности компьютера, то при необходимости получения истинной картины о свойствах компьютера идут по пути *комплексного тестирования программно-аппаратной среды в целом*. Определяют параметры работы вычислительного комплекса на большом наборе программ, имеющих различные вычислительно-коммуникационные характеристики. Такая работа, как правило, тяжела и трудоемка, но другого пути в настоящий момент нет.

Другая достаточно популярная характеристика компьютеров, отражающая и экономическую составляющую – *отношение производительность/стоимость*. При построении компьютера естественно желание получить максимально возможную производительность за минимальную стоимость.

2. Классификация параллельных компьютеров

Существует много различных способов организации параллельных вычислительных систем. Здесь можно назвать векторно-конвейерные компьютеры, массивно-параллельные и матричные системы, компьютеры с широким командным словом, спецпроцессоры, кластеры, компьютеры с многопоточной архитектурой, систолические массивы, dataflow компьютеры и т.п. Если же к подобным названиям для полноты описания добавить и сведения о таких важных параметрах, как организация памяти, топология связи между процессорами, синхронность работы отдельных устройств или способ исполнения операций, то число различных архитектур станет и вовсе необозримым.

Мы приведем здесь с определенной долей условности простую качественную классификацию параллельных компьютеров, главной целью которой является продемонстрировать место вычислительных кластеров во всем множестве возможных архитектур.

На верхнем уровне этой классификации находятся наиболее мощные компьютеры своего времени, назовем их *“классическими” суперкомпьютерами*. Главное, что их характеризует, - это принятые при их конструировании уникальные решения: обычно используются как специально сконструированные процессоры, так и уникальные коммуникации для связи между процессорами. Уникаль-

ные, специально разработанные решения зачастую позволяют достичь рекордных показателей производительности, но надо также помнить, что все уникальное обычно является дорогим, поэтому стоимость таких суперкомпьютеров оказывается очень и очень высокой. В качестве примера таких систем приведем компьютеры CRAY Y-MP C90/T90, HP Superdome, NEC SX-6. Основными областями их применения являются те, в которых необходимо достижение максимально большой производительности и надежности, не считаясь с огромными затратами (космическая, военная отрасль и др.).

Если при создании нового компьютера не ставить цель разработать новый уникальный процессор, а взять один из существующих и выпускаемых серийно, то можно значительно удешевить процесс разработки и уменьшить стоимость получающегося компьютера. За счет выбора типа и количества процессоров можно добиться необходимого уровня пиковой производительности, но во многих случаях критичным является процесс коммуникации процессоров. Если коммуникационная сеть создается специально под новый компьютер, то назовем получающуюся систему *системой с массовым параллелизмом*. Использование серийных процессоров позволяет заметно удешевить систему, но разработка уникальных коммуникаций все же удерживает цену на достаточно высоком уровне. К этому классу отнесем, например, компьютеры CRAY T3D/T3E, IBM SP2 и др.

Если же строить компьютер только на основе серийных компонентов, используя как существующие серийные процессоры, так и какую-то серийную коммуникационную сеть, то мы приходим к *кластерным системам*. Кластерные системы, естественно, характеризуются минимальной стоимостью, а для достижения приемлемого уровня производительности пользуются обычно их хорошей масштабируемостью, наращивая количество процессоров. Дешевизна и легкость создания и наращивания кластерных систем приводят к тому, что они приобретают все большую популярность и распространенность.

Крайней точкой данной классификации можно считать обычные *локальные вычислительные сети*, которые потенциально тоже можно считать единым компьютером и даже использовать в таком качестве при помощи специального программного обеспечения.

3. Вычислительные кластеры

В последнее время все большую популярность в мире получают вычислительные кластеры. Сразу оговоримся, что в компьютерной литературе понятие “кластер” употребляется в различных значениях. В частности, “кластерная” технология используется для повышения скорости работы и надежности серверов баз данных или web-серверов. Здесь мы будем говорить только о кластерах, ориентированных на решение задач вычислительного характера.

Если говорить кратко, то *вычислительный кластер* - это совокупность компьютеров, объединенных в рамках некоторой сети для решения одной задачи. В качестве вычислительных узлов обычно используются доступные на рынке однопроцессорные компьютеры, двух или четырехпроцессорные SMP-серверы. Каждый узел работает под управлением своей копии операционной системы, в качестве которой чаще всего используются стандартные ОС: Linux, Windows NT, Solaris и т.п. Состав и мощность узлов может меняться даже в рамках одного кластера, давая возможность создавать неоднородные системы. Выбор конкретной коммуникационной среды определяется многими факторами: особенностями класса решаемых задач, доступным финансированием, необходимостью последующего расширения кластера и т.п. Возможно включение в конфигурацию кластера специализированных компьютеров, например, файл-сервера. Как правило, предоставляется возможность удаленного доступа на кластер через Internet.

Ясно, что простор для творчества при проектировании кластеров огромен. Узлы могут не содержать локальных дисков, коммуникационная среда может одновременно использовать различные сетевые технологии, узлы не обязаны быть одинаковыми и т.д. Рассматривая крайние точки, кластером можно считать как пару ПК, связанных локальной 10-мегабитной Ethernet-сетью, так и вычислительную систему, создаваемую в рамках проекта Cplant в Национальной лаборатории Sandia: 1400 рабочих станций на базе процессоров Alpha объединены высокоскоростной сетью Myrinet.

С появлением вычислительных кластеров параллельные вычисления стали доступны многим. Если раньше параллельные компьютеры стояли в больших центрах, то сейчас кластер может собрать и поддерживать небольшая лаборатория. Стоимость кластерных решений значительно ниже стоимости традиционных суперкомпьютеров. Для их построения, как правило, используются массовые процессоры, стандартные сетевые технологии и свободно распространяемое программное обеспечение. Если есть желание, минимум средств и знаний, то принципиальных препятствий для построения собственной параллельной системы нет.

4. Список TOP500

Top500 – официальный список 500 мощнейших компьютеров мира. Производительность компьютеров оценивается на тесте LINPACK. Список обновляется 2 раза в год. Последняя (20-я) редакция списка опубликована в ноябре 2002 г.

Тест LINPACK представляет собой решение больших систем линейных алгебраических уравнений методом LU-разложения. Параллельная реализация LINPACK основана на библиотеке SCALAPACK. Производительность определяется как количество “содержательных” операций с плавающей точкой в еди-

ницу времени, и выражается в Мфлопс. Число выполненных операций с плавающей точкой оценивается по формуле $2n^3/3 + 2n^2$ (здесь n - размерность матрицы). В настоящее время LINPACK используется для формирования списка Top500 пятисот самых мощных компьютеров мира. Кроме пиковой производительности R_{peak} для каждой системы указывается величина R_{max}, равная производительности компьютера на тесте LINPACK с матрицей максимального для данного компьютера размера N_{max}. По значению R_{max} отсортирован весь список. Как показывает анализ представленных данных, величина R_{max} составляет 50—70% от значения R_{peak}. Интерес для анализа функционирования компьютера представляет и указанное в каждой строке значение $N_{1/2}$, показывающее, на матрице какого размера достигается половина производительности R_{max}.

В последнюю, 20-ю редакцию списка 500 наиболее мощных компьютеров мира (июнь 2002 года, **http://parallel.ru/computers/top500.list20.html**) вошло уже 93 кластерных системы.

5. Сравнение коммуникационных технологий построения кластеров

Понятно, что различных вариантов построения кластеров очень много. Одно из существенных различий состоит в используемой сетевой технологии, выбор которой определяется, прежде всего, классом решаемых задач.

Первоначально Beowulf-кластеры строились на базе обычной 10-мегабитной сети Ethernet. Сегодня часто используется сеть Fast Ethernet, как правило, на базе коммутаторов. Основное достоинство такого решения — это низкая стоимость. Вместе с тем, большие накладные расходы на передачу сообщений в рамках Fast Ethernet приводят к серьезным ограничениям на спектр задач, эффективно решаемых на таких кластерах. Если от кластера требуется б**о**льшая универсальность, то нужно переходить на другие, более производительные коммуникационные технологии. Исходя из соображений стоимости, производительности и масштабируемости, разработчики кластерных систем делают выбор между Fast Ethernet, Gigabit Ethernet, SCI, Myrinet, cLAN, ServerNet и рядом других сетевых технологий.

Какими же числовыми характеристиками выражается производительность коммуникационных сетей в кластерных системах? Необходимых пользователю характеристик две: латентность и пропускная способность сети. *Латентность* — это время начальной задержки при посылке сообщений. *Пропускная способность сети* определяется скоростью передачи информации по каналам связи. Если в программе много маленьких сообщений, то сильно скажется латентность. Если сообщения передаются большими порциями, то важна высокая пропускная способность каналов связи. Наличие латентности определяет и тот

факт, что максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной.

Рис. Латентность и пропускная способность канала

На практике пользователям не столько важны заявляемые производителем пиковые характеристики, сколько реальные показатели, достигаемые на уровне приложений, например из программ, использующих технологию MPI. После вызова пользователем функции посылки сообщения **`Send()`** сообщение последовательно проходит через целый набор слоев, определяемых особенностями организации программного обеспечения и аппаратуры. Этим, в частности, определяются и множество вариаций на тему латентности реальных систем. Установили MPI на компьютере плохо, латентность будет большая, купили дешевую сетевую карту от неизвестного производителя, ждите дальнейших сюрпризов. Кстати, наличие латентности определяет и тот факт, что максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной.

В следующей таблице приводятся некоторые характеристики наиболее известных коммуникационных технологий, используемых при построении кластерных систем.

Таблица. **Сравнительные характеристики коммуникационных технологий**

	SCI	**Myrinet**	**cLAN**	**Server-Net**	**Fast Ethernet**
Латентность (MPI)	5,6 мкс	17 мкс	30 мкс	13 мкс	170 мкс
Пропускная способность (MPI)	80 Мбайт/с	40 Мбайт/с	100 Мбайт/с	180 Мбайт/с	10 Мбайт/с
Пропускная способность (аппаратная)	400 Мбайт/с	160 Мбайт/с	150 Мбайт/с	н/д	12,5 Мбайт/с
Реализация MPI	ScaMPI	HPVM, MPICH-GM и др.	MPI/Pro	MVICH	MPICH
Производитель	Dolphin	Myricom	Giganet	Compaq	Intel, Com и др.

6. Системы хранения данных

Производительность кластера, как и любого другого компьютера зависит не только от характеристик его процессора. Не менее важное значение имеет и организация систем ввода/вывода, которым зачастую уделяется недостаточное внимание при построении высокопроизводительных вычислительных систем. Многие современные задачи требуют хранения очень больших объемов данных с быстрым доступом к ним. Немалое значение имеет также повышение надежности хранения данных.

Наиболее распространенные в настоящее время системы хранения данных основаны на использовании магнитных и магнитооптических дисков. Простые системы основаны на использовании *магнитных дисков* стандарта **IDE** и некоторых его разновидностей. Например, жесткий диск компании Seagate Technology ST340016A (Barracuda ATA IV) имеет емкость 40 Гбайт, среднее время доступа 9 мс при скорости вращения 7200 оборот/мин, внешнюю скорость передачи данных 100 Мбайт/сек., время наработки на отказ 600000 часов. *Магнитооптические диски* уступают обычным жестким магнитным дискам лишь по времени доступа к данным. Стоимость хранения единицы данных на них в несколько раз меньше стоимости хранения того же объема данных на жестких магнитных дисках, однако магнитооптика еще далека от массового применения, главным образом из-за высокой цены самого дисковода. Для примера, дисковод Sierra 1.3 Гбайт обеспечивает среднее время доступа 19 мс и среднее время наработки на отказ 80000 часов.

Дальнейшее повышение надежности и коэффициента готовности дисковых подсистем достигается построением избыточных дисковых массивов **RAID (Redundant Array of Independent Disks)**. *Дисковый массив* - это набор дисковых устройств, работающих вместе, чтобы повысить скорость и надежность системы ввода/вывода. Этим набором устройств управляет специальный RAID-контроллер (контроллер массива), который инкапсулирует в себе функции размещения данных по массиву; а для всей остальной системы позволяет представлять весь массив как одно логическое устройство ввода/вывода. За счет параллельного выполнения операций чтения и записи на нескольких дисках, массив обеспечивает повышенную скорость обменов по сравнению с одним большим диском.

Массивы также могут обеспечивать избыточное хранение данных, с тем, чтобы данные не были потеряны в случае выхода из строя одного из дисков. В зависимости от уровня RAID проводится зеркалирование или распределение данных по дискам. Каждый из четырех основных уровней RAID использует уникальный метод записи данных на диски, и поэтому все уровни обеспечивают различные преимущества. Уровни RAID 1, 3 и 5 обеспечивают зеркалирование или хранение битов четности; и поэтому позволяют восстановить информацию в случае сбоя одного из дисков.

Технология RAID 0 также известна как *распределение данных (data striping)*. При применении этой технологии информация разбивается на куски (фиксированные объемы данных, обычно именуемых блоками), эти куски записываются на диски и считываются с них в параллель. С точки зрения производительности это означает два основных преимущества: повышается пропускная способность последовательного ввода/вывода за счет одновременной загрузки нескольких интерфейсов и снижается латентность случайного доступа, поскольку несколько запросов к различным небольшим сегментам информации могут выполняться одновременно. Недостатком является то, что уровень RAID 0 предназначен исключительно для повышения производительности и не обеспечивает избыточности данных. Поэтому любые дисковые сбои потребуют восстановления информации с резервных носителей.

Технология RAID 1 также известна как *зеркалирование (disk mirroring)*. В этом случае копии каждого куска информации хранятся на отдельном диске. Обычно каждый используемый диск имеет “двойника”, который хранит точную копию этого диска. Если происходит сбой одного из основных дисков, то он замещается своим “двойником”. Производительность произвольного чтения может быть улучшена, если для чтения информации будет использоваться тот из “двойников”, головка которого расположена ближе к требуемому блоку. Время записи может оказаться несколько больше, чем для одного диска, в зависимости от стратегии записи: запись на два диска может производиться либо в параллель (для скорости), либо строго последовательно (для надежности).

Уровень RAID 1 хорошо подходит для приложений, которые требуют высокой надежности, низкой латентности при чтении, а также если не требуется минимизация стоимости. RAID 1 обеспечивает избыточность хранения информации, но в любом случае следует поддерживать резервную копию данных, т.к. это единственный способ восстановить случайно удаленные файлы или директории.

Технология RAID уровней 2 и 3 предусматривает параллельную (“в унисон”) работу всех дисков. Эта архитектура требует хранения битов четности для каждого элемента информации, распределяемого по дискам. Отличие RAID 3 от RAID 2 состоит только в том, что RAID 2 использует для хранения битов четности несколько дисков, тогда как RAID 3 использует только один. RAID 2 используется крайне редко.

Если происходит сбой одного диска с данными, то система может восстановить его содержимое по содержимому остальных дисков с данными и диска с информацией четности. Производительность в этом случае очень велика для больших объемов информации, но может быть весьма скромной для малых объемов, поскольку невозможно перекрывающееся чтение нескольких небольших сегментов информации.

RAID 4 исправляет некоторые недостатки технологии RAID 3 за счет использования больших сегментов информации, распределяемых по всем дискам, кроме диска с информацией четности. При этом для небольших объемов информации используется только диск, на котором находится нужная информация. Это означает, что возможно одновременное исполнение нескольких запросов на чтение. Однако запросы на запись порождают блокировки при записи информации четности. RAID 4 используется крайне редко.

Технология RAID 5 очень похожа на RAID 4, но устраняет связанные с ней блокировки. Различие состоит в том, что информация четности распределяется по всем дискам массива. В данном случае возможны как одновременные операции чтения, так и записи. Данная технология хорошо подходит для приложений, которые работают с небольшими объемами данных, например, для систем обработки транзакций.

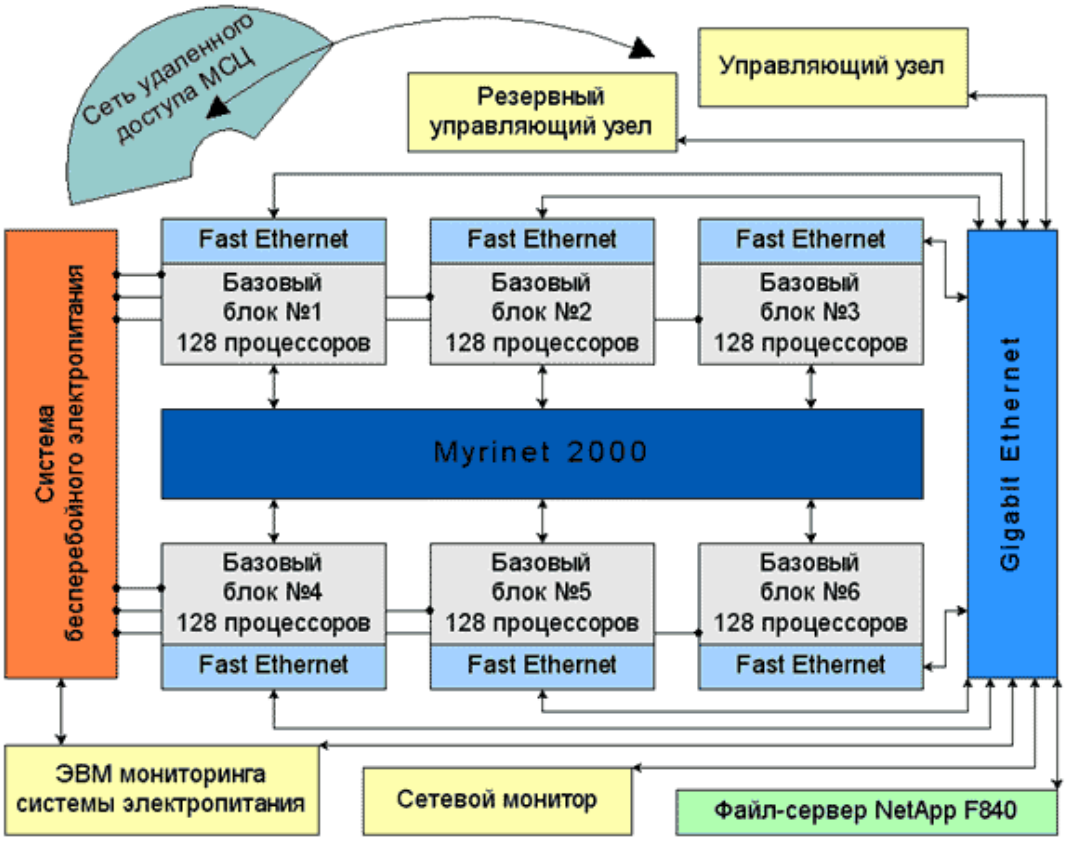
Каждый узел вычислительного кластера НИВЦ МГУ содержит жесткий диск IDE 4.3 Гбайт Quantum (или 10 Гбайт Fujitsu). Кроме того, предоставляется возможность использования RAID-массива из 6 дисков Ultra160 SCSI объемом по 36 Гбайт (IBM), установленного на файл-сервере. Все файлы из домашней директории пользователя хранятся на файл-сервере, для сохранения файлов на локальный диск нужно записывать файлы в каталог **/tmp**. На другом вычислительном кластере (SKY) на каждом узле установлены два жестких диска, подключенные к различным IDE-каналам, что позволяет использовать их парал-

лельно двумя прикладными процессами, за счет чего можно получить заметное ускорение задач, требующих интенсивного потока ввода/вывода.

7. Высокопроизводительные вычисления в России

В последнее время и в России происходит бурный рост интереса к высокопроизводительным вычислениям. Связан он, в первую очередь, с кластерными технологиями, позволяющими получать потенциально очень большую производительность за относительно невысокую стоимость. Кластерные установки появляются повсеместно, их число и мощность растет очень быстро.

Так, например, кластером является и самый мощный российский суперкомпьютер **МВС-1000М**, установленный в Межведомственном суперкомпьютерном центре в Москве (**http://www.jscc.ru/**). Компьютер состоит из шести базовых блоков, содержащих по 64 двухпроцессорных модуля. Каждый модуль имеет два процессора Alpha 21264/667 МГц (кэш-память второго уровня 4 Мбайта), 2 Гбайта оперативной памяти, разделяемой процессорами модуля, жесткий диск. Общее число процессоров в системе равно 768, а пиковая производительность МВС-1000М превышает 1 TFLOPS. В 20-ой редакции списка 500 наиболее мощных компьютеров мира Top500 компьютер МВС-1000М занимает 74-е место.



Все модули МВС-1000М связаны двумя независимыми сетями. Сеть Myrinet 2000 используется программами пользователей для обмена данными в процессе вычислений. При использовании MPI пропускная способность каналов сети достигает значений 110-150 Мбайт/с. Сеть Fast Ethernet используется операционной системой для выполнения сервисных функций.

8. Задания:

- Модифицировать программу, реализующую метод Гаусса решения систем линейных алгебраических уравнений таким образом, чтобы на каждом процессоре выделялась только такая часть массивов данных, которая необходима для выполнения распределенных на данный процессор операций, исследовать эффективность и масштабируемость полученной программы.
- Чем отличается кластер от традиционного суперкомпьютера? Что у них общего?
- Какие известны классификации вычислительных систем? Суперкомпьютеров? По каким параметрам имеет смысл строить классификации?
- На какие характеристики программно-аппаратной среды кластера нужно обращать особое внимание при разработке эффективной параллельной программы?

- Какова максимальная производительность кластеров в настоящее время? Сколько кластеров имеют производительность более 1 TFLOPS? (Используйте данные из списка Top500)
- Вам нужно определить конфигурацию кластера для Вашей организации. Какими критериями Вы будете руководствоваться в процессе проектирования?
- Выберите уровень технологии RAID, наиболее, на Ваш взгляд, подходящий для хранения данных в Вашей организации. Обоснуйте выбор.
- Из чего складывается пиковая производительность суперкомпьютера МВС-1000М (более 1 TFLOPS)?