Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Нижегородский государственный университет
им. Н.И. Лобачевского

В.А. Гришагин, А.Н. Свистунов

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ОСНОВЕ MPI

Учебное пособие

Нижний Новгород
Издательство Нижегородского госуниверситета
2005

УДК 004.421.2
ББК 32.973.26-018.2
Г 82

Г 82. Гришагин В.А., Свистунов А.Н. *Параллельное программирование на основе MPI.* Учебное пособие – Нижний Новгород: Изд-во ННГУ им.Н.И. Лобачевского, 2005. - 93 с.

ISBN 5-85746-758-6

Настоящее пособие содержит описание инструментальных средств разработки параллельных программ для многопроцессорных вычислительных комплексов на основе библиотеки MPI (Message Passing Interface), которая является одной из наиболее распространенных систем параллельного программирования. Наряду с рассмотрением основных возможностей библиотеки пособие включает примеры практических параллельных программ, иллюстрирующих основные принципы и приемы параллельного программирования в среде MPI. Для облегчения трудоемкого процесса поиска ошибок программирования излагаемый материал содержит раздел по отладке и профилированию параллельных программ.

Пособие предназначено для использования в учебном процессе при подготовке студентов по проблематике параллельных вычислений и может найти своего читателя в среде научных работников и специалистов широкого профиля, использующих современные высокопроизводительные параллельные вычислительные системы для решения сложных научных и прикладных задач.

Пособие разработано в рамках выполнения гранта Конкурсного центра фундаментального естествознания (КЦФЕ) Рособразования № Е02-1.0-58.

ББК 32.973.26-018.2

ISBN 5-85746-758-6

Введение

Создание *многопроцессорных (параллельных) вычислительных систем (ПВС)* является стратегической линией развития компьютерной техники, обусловливаемой существованием в любой текущий момент времени актуальных задач фундаментальной и прикладной науки, для анализа и исследования которых производительности существующих средств вычислительной техники оказывается недостаточно. Тем не менее практическое использование параллельных вычислительных систем не является столь широким, как это могло бы быть потенциально возможным. К основным сдерживающим факторам широкого распространения параллельных вычислений следует отнести *большую стоимость* и *разнообразие архитектурного построения* ПВС, существенно более высокую (по сравнению с последовательным программированием) *трудоемкость* разработки эффективных параллельных алгоритмов и программ.

Преодоление первого сдерживающего фактора по широкому использованию параллельных вычислений (высокая стоимость ПВС) может быть получено на пути построения *кластерных вычислительных систем* (*clusters*). Под *кластером* обычно понимается множество отдельных компьютеров, объединенных в сеть, для которых при помощи специальных аппаратно-программных средств обеспечивается возможность унифицированного управления (*single system image*), надежного функционирования (*availability*) и эффективного использования (*performance*). Кластеры могут быть образованы на базе уже существующих у потребителей отдельных компьютеров либо же сконструированы из типовых компьютерных элементов, что обычно не требует значительных финансовых затрат. Применение кластеров может также в некоторой степени снизить проблемы, связанные с разработкой параллельных алгоритмов и программ, поскольку повышение вычислительной мощности отдельных процессоров позволяет строить кластеры из сравнительно небольшого количества (несколько десятков) отдельных компьютеров (*lowly parallel processing*). Это приводит к тому, что для параллельного выполнения в алгоритмах решения вычислительных задач достаточно выделять только крупные независимые части расчетов (*coarse granularity*), что, в свою очередь, снижает сложность построения параллельных методов вычислений и уменьшает потоки передаваемых

данных между компьютерами кластера. Вместе с этим следует отметить, что организация взаимодействия вычислительных узлов кластера при помощи передачи сообщений обычно приводит к значительным временным задержкам, что накладывает дополнительные ограничения на тип разрабатываемых параллельных алгоритмов и программ.

Решение проблемы разнообразия архитектур параллельных вычислительных систем и обеспечение возможности создания мобильных (переносимых между различными компьютерными платформами) программ лежит на пути разработки стандартизованного базового системного программного обеспечения для организации параллельных вычислений. Основным стандартом, широко используемым в настоящее время в практических приложениях, является *интерфейс передачи сообщений* (*message passing interface - MPI*). Наличие такого стандарта позволило разработать стандартные библиотеки программ (*MPI-библиотеки*), в которых оказалось возможным скрыть большинство архитектурных особенностей ПВС и, как результат, существенно упростить проблему создания параллельных программ. Более того, стандартизация базового системного уровня позволила в значительной степени обеспечить мобильность параллельных программ, поскольку в настоящее время реализации MPI-стандарта имеются для большинства компьютерных платформ.

Использование передачи сообщений для организации параллельных вычислений ориентировано прежде всего на *многопроцессорные компьютерные системы с распределенной памятью.* Уменьшение времени расчетов (*ускорение*) при таком подходе может быть достигнуто только для тех научно-технических проблем, в которых объем вычислений превалирует над уровнем необходимых межпроцессорных взаимодействий (т.е. *для вычислительно-трудоемких задач с низкой коммуникационной сложностью*).

В данном пособии не ставится задача изложить основы параллельного программирования и полные функциональные возможности среды MPI (для этих целей можно рекомендовать высокопрофессиональные публикации [5-15]). Основное назначение пособия - дать актуальные практические сведения для разработки реальных параллельных MPI-программ. В связи с этим излагаемый материал содержит значительное количество примеров как по

отдельным функциям MPI, так и по реализации законченных параллельных алгоритмов.

Одной из основных проблем, возникающих при разработке параллельных программ, является их отладка. В отличие от последовательного случая трасса параллельной программы (временна́я диаграмма исполняемых операций) может меняться от запуска к запуску. Для уменьшения трудоемкости процесса анализа исполнения параллельных программ может быть рекомендована инструментальная среда MPE (*Multi Processing Environment*), адаптированная В.В.Травкиным и предназначенная для отладки и профилировки программ, созданных с использованием библиотеки MPI.

Разработка пособия выполнялась в рамках реализации образовательного проекта учебно-исследовательской лаборатории "Математические и программные технологии для современных компьютерных систем (Информационные технологии)", созданной в Нижегородском госуниверситете при поддержке компании Интел.

1. Параллельные вычисления для систем с распределенной памятью

1.1. Параллельные вычислительные модели

Параллельные вычислительные модели образуют сложную структуру. Они могут быть дифференцированы по нескольким направлениям: является ли память физически общей или распределенной; насколько обмен данными реализован в аппаратном и насколько в программном обеспечении; что в точности является единицей исполнения и т.д. Общая картина осложняется тем, что программное обеспечение позволяет реализовать любую вычислительную модель на любом оборудовании.

Для систем с распределенной памятью организация параллельных вычислений является возможной при использовании тех или иных способов передачи данных между взаимодействующими процессорами. *Модель с передачей сообщений* подразумевает множество процессов, имеющих только локальную память, но способных связываться с другими процессами, посылая и принимая сообщения. Определяющим свойством модели для передачи сообщений является тот факт, что передача данных из локальной памяти одного процесса в локальную память другого процесса требует выполнения операций обоими процессами.

1.2. Преимущества модели с передачей данных

Универсальность. Модель с передачей сообщений хорошо подходит к системам с отдельными процессорами, соединенными с помощью (быстрой или медленной) сети. Тем самым, она соответствует как большинству сегодняшних параллельных суперкомпьютеров, так и сетям рабочих станций.

Выразительность. Модель с передачей сообщений является полезной и полной для выражения параллельных алгоритмов. Она приводит к ограничению управляющей роли моделей, основанных на компиляторах и параллельности данных. Она хорошо приспособлена для реализации адаптивных алгоритмов, подстраивающихся к несбалансированности скоростей процессов.

Легкость отладки. Отладка параллельных программ обычно представляет собой непростую задачу. Модель с передачей сообщений управляет обращением к памяти более явно, чем любая другая модель

(только один процесс имеет прямой доступ к любым переменным в памяти), и, тем самым, облегчает локализацию ошибочного чтения или записи в память.

Производительность. Важнейшей причиной того, что передача сообщений остается постоянной частью параллельных вычислений, является производительность. По мере того, как CPU становятся все более быстрыми, управление их кэшем и иерархией памяти вообще становится ключом к достижению максимальной производительности. Передача сообщений дает программисту возможность явно связывать специфические данные с процессом, что позволяет компилятору и схемам управления кэшем работать в полную силу.

Все вышеперечисленное объясняет, почему передача сообщений становится одной из наиболее широко используемых технологий для выражения параллельных алгоритмов. Несмотря на некоторые недостатки, передача сообщений, более чем любая другая модель, приближается к тому, чтобы быть стандартным подходом для реализации параллельных приложений.

1.3. Стандаритизация модели передачи данных (MPI)

Использование модели передачи данных для разработки параллельных программ предполагает стандартизацию основных используемых понятий и применяемых средств. Подобные попытки унификации после разработки отдельных инструментальных коммуникационных библиотек привели к появлению в 1994 г. стандарта интерфейса передачи данных (*message passing interface - MPI*).

Основной целью спецификации MPI явилось сочетание в рамках единого подхода мобильных, эффективных и развитых средств передачи данных. Это означает возможность писать мобильные программы, использующие специализированное аппаратное или программное обеспечение, предлагаемое отдельными поставщиками. В то же время многие свойства, такие как ориентированная на приложения структура процессов или динамически управляемые группы процессов с широким набором коллективных операций, которые есть в любой реализации MPI, могут быть использованы в любой параллельной прикладной программе. Одна из наиболее критических групп пользователей - это разработчики параллельных библиотек, для которых эффективная, мобильная и

высокофункциональная программа особенно важна. MPI - это первая спецификация, которая позволяет им писать действительно мобильные библиотеки.

Выбранные требования - мобильность, эффективность, функциональность - диктуют многие проектные решения, которые определяют спецификацию MPI. Из дальнейшего описания будет видно, как эти решения влияют и на фундаментальные операции модели - *send* и *receive* - и на множество других, более развитых операций по передаче сообщений, включенных в MPI. MPI не является революционно новым способом программирования для параллельных компьютеров. Скорее, это попытка собрать лучшие свойства многих существующих систем передачи сообщений, улучшить их, если необходимо, и стандартизировать:

- MPI служит основой для разработки библиотек передачи данных. MPI специфицирует имена, последовательности вызова и результаты подпрограмм, вызываемых из программ на Фортране 77, и функций, вызываемых из Си-программ. Программы, написанные на Фортране 77 и Си, компилируются обычными компиляторами и связываются с библиотекой MPI;
- MPI относится к модели для передачи сообщений. Вычисление остается собранием *процессов*, связанных *сообщениями*.

2. Основные понятия MPI

2.1. Функции передачи сообщений

Многие понятия MPI, которые на первый взгляд могут показаться новыми, на самом деле являются необходимыми уточнениями хорошо знакомых понятий. Рассмотрим самую элементарную операцию в библиотеке передачи сообщений - основную операцию send. Во многих современных системах передачи сообщений она выглядит следующим образом:

```
send(address, length, destination, tag),
```

где

- `address` - адрес в памяти начала буфера, содержащего посылаемые данные;
- `length` - длина сообщения в байтах;

– `destination` - идентификатор процесса, которому посылается сообщение (обычно, как целое);

– `tag` - произвольное неотрицательное целое число, метка сообщения (иногда называемое также *типом*). Позволяет принимающей стороне "различать" сообщения и организовывать прием сообщений только нужного *типа.*

Эта конкретная совокупность параметров часто выбирается благодаря тому, что она представляет собой хороший компромисс между тем, что нужно программисту, и тем, что эффективно реализуется аппаратурой (передача непрерывной области памяти от одного процессора другому). В частности, можно ожидать, что операционная система обеспечивает обслуживание очередей так, чтобы операция приема

```
recv(address, maxlen, source, tag, datlen)
```

успешно завершалась только в случае получения сообщения с нужной меткой (`tag`). Другие сообщения помещаются в очередь до тех пор, пока не будет выполнена соответствующая операция приема. В большинстве систем `source` - это выходной аргумент, указывающий, откуда пришло сообщение, хотя в некоторых системах он также может быть использован для ужесточения отбора сообщений и помещения их в очередь. При приеме `address` и `maxlen` вместе описывают буфер, в который нужно поместить принимаемые данные, `datlen` представляет число принятых байтов.

Системы передачи сообщений с такого рода синтаксисом и семантикой показали себя чрезвычайно полезными, хотя и наложили ограничения, нежелательные для большого числа пользователей. MPI старался снять эти ограничения, предлагая более гибкие версии каждого из этих параметров, сохраняя при этом привычный смысл основных операций `send` и `receive`.

Описание буферов сообщений. Спецификация (`address, length`) посылаемого сообщения ранее хорошо соответствовала аппаратуре, но больше не является адекватной по двум причинам:

- Во многих ситуациях посылаемое сообщение не является *непрерывным*. В простейшем случае это может быть строка матрицы, хранящейся в столбцах. Вообще, сообщение может состоять из произвольно расположенных наборов структур различного размера. В

прошлом программисты (и библиотеки) предусматривали программы для упаковки этих данных в непрерывные буферы перед передачей и для распаковки их на приемном конце. Однако, поскольку появляются коммуникационные процессоры, которые могут непосредственно обращаться с данными, расположенными в памяти с равномерным шагом или еще более общим образом, становится особо важно с точки зрения производительности, чтобы упаковка производилась "на лету" самим коммуникационным процессором без дополнительных перемещений данных. Это невозможно сделать, если мы не представляем информацию в библиотеку связи в ее первоначальной (распределенной) форме;

- В последние несколько лет замечается рост популярности неоднородных вычислительных комплексов. Эта популярность возникла по двум причинам. Первая состоит в распределении сложной вычислительной задачи между различными специализированными компьютерами (например, SIMD, векторными, графическими). Вторая причина состоит в использовании сетей рабочих станций в качестве параллельных компьютеров. Сети рабочих станций, состоящие из машин, приобретенных в разное время, часто содержат компьютеры разных типов. В обеих этих ситуациях сообщениями обмениваются машины разной архитектуры, при этом `address`, `length` уже больше не являются адекватной спецификацией семантического содержания сообщения. Например, в векторах из чисел с плавающей точкой могут быть различными не только формат чисел с плавающей точкой, но даже их длина. Эта ситуация также может иметь место и для целых чисел. Библиотека связи может выполнять необходимое конвертирование, если ей точно сообщается, что именно передается.

Решение MPI обеих этих проблем состоит в том, чтобы специфицировать сообщения на более высоком уровне и более гибким образом, чем (`address, length`), и чтобы отразить тот факт, что сообщение содержит более структурированные данные, чем просто строку бит. Для этого буфер сообщения в MPI определяется тройкой (`address, count, datatype`), описывающей `count` элементов данных типа `datatype`, начинающегося с `address`. Мощь этого механизма происходит от гибкости в значениях типа данных (`datatype`).

Прежде всего, тип данных может принимать значения элементарных типов данных в принимающем языке. Таким образом,

(`A, 10, MPI_REAL`) описывает вектор **A** из 10 вещественных чисел на Фортране, вне зависимости от длины и формата числа с плавающей точкой. Реализация MPI для неоднородной сети гарантирует, что будут приняты те же 10 вещественных чисел, даже если принимающая машина имеет совершенно другой формат плавающей точки.

Используя средства MPI, пользователь может определять свои собственные типы данных, причем эти типы могут описывать "распределенные" по памяти (не непрерывные) данные.

2.2. Понятие коммуникаторов

Разделение семейств сообщений. Почти все системы передачи сообщений предусматривают аргумент `tag` для операций `send` и `receive`. Этот аргумент позволяет программисту обрабатывать приходящие сообщения упорядоченным образом, даже если сообщения поступают в порядке, отличном от желаемого. Системы передачи сообщений помещают сообщения, пришедшие "с неверной меткой", в очередь до тех пор, пока программа не будет готова принять их. Обычно имеется возможность специфицировать "универсальную" метку, которая сопоставляется с любой меткой.

Этот механизм оказался необходимым, но не достаточным, поскольку произвольность выбора метки означает, что по всей программе метки должны использоваться единым заранее определенным образом. Особые трудности возникают в случае библиотек, написанных далеко от прикладного программиста во времени и пространстве, так как сообщения этой библиотеки не должны быть случайно приняты прикладной программой.

Решение MPI состоит в расширении понятия метки до нового понятия *контекста*. Контексты размещаются во время исполнения программы системой в ответ на запрос пользователя (или библиотеки) и используются для сопоставления сообщений. От метки контексты отличаются тем, что вырабатываются системой, а не пользователем, и никакое "универсальное" сопоставление не допускается.

MPI сохраняет и обычное определение метки сообщения, включающего качество универсального сопоставления.

Наименование процессов. Процессы принадлежат *группам*. Если группа содержит *n* процессов, то процессы внутри группы идентифицируются *рангами* - целыми числами от 0 до *n*-1. В рамках стандарта MPI предполагается, что существует начальная группа,

которой принадлежат все процессы выполняемой параллельной программы. Внутри этой группы процессы нумеруются подобно тому, как они нумеруются во многих существующих системах передачи сообщений - от нуля до общего числа процессов минус один.

Коммуникаторы. Понятия контекста и группы комбинируются в единый объект, называемый *коммуникатором*, который становится аргументом к большинству двухточечных и коллективных операций. Таким образом, `destination` или `source`, специфицированные в операции посылки или приема, всегда указывают на ранг процесса в группе, идентифицированный с данным коммуникатором.

Иными словами, в MPI основная (синхронная) операция `send` выглядит следующим образом:

`MPI_Send(buf, count, datatype, dest, tag, comm)`,
где

- (`buf, count, datatype`) описывают count элементов вида `datatype`, начинающихся с `buf`;
- `dest` - ранг получателя в группе, связанной с коммуникатором `comm`;
- `tag` - имеет обычное значение;
- `comm` - идентифицирует группу процессов и контекст связи.

Операция `receive` превращается в

`MPI_Recv(buf, count, datatype, source, tag, comm, status)`.

Источник, метку и размер реально принятого сообщения можно извлечь из `status`.

Некоторые другие системы передачи сообщений возвращают параметры "статуса" отдельными вызовами, которые косвенно ссылаются на последнее принятое сообщение. Метод MPI - один из аспектов усилий, направленных на то, чтобы надежно работать в ситуации, когда несколько потоков принимают сообщение от имени одного процесса.

3. Методы разработки параллельных программ с использованием интерфейса передачи сообщений MPI

При программировании с использованием модели передачи сообщений считают, что компьютер представляет собой набор независимых процессоров, каждый из которых обладает собственной локальной памятью. Работающая программа – это набор процессов (подзадач), каждая из которых выполняется на своем процессоре. В процессе работы подзадачи обмениваются данными между собой.

При разработке параллельной программы с использованием MPI исходная задача разбивается на подзадачи (декомпозиция). Обычная техника состоит в следующем: каждая из подзадач оформляется в виде отдельной структурной единицы (функции, модуля), на всех процессорах запускается одна и та же программа "загрузчик", которая, в зависимости от "номера" процессора загружает ту или иную подзадачу. Такой подход (запуск одной и той же программы на всех выделенных для решения задачи процессоров) получил в литературе наименование SPMD (*single program multiple data*).

Рассмотрим эту технологию подробнее на примере широко используемой демонстрационной программы типа "Hello world"

Первая программа

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[])
{
  int rank, numprocs, i, message;
// инициализация библиотеки MPI
  MPI_Init(&argc, &argv);
// получение количества процессов в программе
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
// получение ранга процесса
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf ("\n Hello from process %3d", rank);
// завершение библиотеки MPI
  MPI_Finalize();
  return 0;
}
```

Первым в каждой MPI программе должен быть вызов `MPI_Init`, который должен присутствовать в каждой программе MPI и предшествует всем другим вызовам MPI[1]. Он устанавливает "среду" (environment) MPI**.** Только одно обращение к `MPI_Init` допускается в исполнении программы. Его аргументами являются количество аргументов командной строки процесса и собственно командная строка процесса. Почти каждая функция MPI возвращает код ошибки, который выражен либо как `MPI_SUCCESS`, либо как зависимый от реализации код ошибки. В этом примере (и многих других примерах) для сокращения количества кода мы не будем особенно тщательны в проверке кода возврата, предполагая, что он всегда равен `MPI_SUCCESS`.

Весь обмен данных в MPI связан с коммуникатором. В этой программе мы будем использовать только предопределенный коммуникатор `MPI_COMM_WORLD`, определяющий единый контекст и весь набор процессов параллельной программы. Определение `MPI_COMM_WORLD` приводится в `mpi.h`**.**

Функция `MPI_Comm_size` возвращает (в `numprocs`) число запущенных для данной программы процессов. Каким способом пользователь запускает эти процессы -зависит от реализации, но любая программа может определить число запущенных процессов с помощью данного вызова. Значение `numprocs` - это, по сути, размер группы, связанной с коммуникатором `MPI_COMM_WORLD`. Процессы каждой группы пронумерованы целыми числами, начиная с 0, которые называются рангами (`rank`). Каждый процесс определяет свой номер в группе, связанной с данным коммуникатором, с помощью `MPI_Comm_rank`. Таким образом, каждый процесс получает одно и то же число в `numprocs`, но разные числа в `rank`. Каждый процесс печатает свой ранг и общее количество запущенных процессов, затем все процессы выполняют `MPI_Finalize`. Эта функция должна быть выполнена каждым процессом MPI и приводит к ликвидации "среды" MPI. Никакие вызовы MPI не могут быть осуществлены процессом после вызова `MPI_Finalize` (повторный `MPI_Init` также невозможен).

---

[1] За исключением MPI_Initialized, которым библиотека может определить, выполнялся ли MPI_INIT или нет.

Нужно заметить, что каждый процесс таким образом написанной и запущенной MPI-программы напечатает **одно** сообщение. Если все процессы были запущены на разных машинах, **на каждой машине** будет напечатано соответствующее сообщение[2]. Допустим, что мы хотим, чтобы все сообщения напечатал какой-то один процесс - например, процесс с рангом 0. Кроме того, мы хотим, чтобы ранги процессов, которые он должен напечатать, **передали** бы ему сами процессы. Таким образом, мы хотим реализовать систему с выделенным нулевым процессом, который должен принять сообщения от всех остальных процессов.

3.1. Передача сообщений между двумя процессами

```
A          #include <stdio.h>
           #include "mpi.h"
           int main(int argc, char* argv[]){
             int rank, n, i, message;
             MPI_Status status;
             MPI_Init(&argc, &argv);
             MPI_Comm_size(MPI_COMM_WORLD, &n);
             MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if (rank == 0){
B             printf ("\n Hello from process %3d",
        rank);
              for (i=1; i<n; i++){
                MPI_Recv(&message, 1, MPI_INT,
        MPI_ANY_SOURCE,
                   MPI_ANY_TAG, MPI_COMM_WORLD,
        &status);
                printf("\n Hello from process
        %3d", message);
          }
C     }else
   MPI_Send(&rank,1,MPI_INT,0,0,MPI_COMM_WORLD);
A          MPI_Finalize();
```

[2] На самом деле, в используемой реализации MPI все сообщения будут напечатаны на одной машине - на той, на которой окажется процесс с рангом 0. Подробнее см. п. 9.2. Запуск приложений с использованием MPI

9.2.1. Запуск программы при использовании Argonne National Lab

```
    return 0;
}
```

Каждый процесс, стартуя, опрашивает два параметра: общее количество процессов в группе (`n`) и свой ранг (`rank`). Делается это при помощи уже известных нам функций `MPI_Comm_size` и `MPI_Comm_rank`. То, что происходит далее, иллюстрирует широко применяемую при написании MPI программ технику - процессы начинают вести себя по разному в зависимости от своего ранга[3]. В тексте программы буквой `A` помечен код, выполняемый **всеми** процессами. Буквой `B` помечен код, выполняемый процессом с рангом, равным нулю. И, наконец, буквой `C` помечен код, выполняемый всеми другими процессами, с рангами, отличными от нуля. Для того, чтобы понять логику выполнения программы, нужно рассмотреть еще две функции MPI - функцию передачи сообщений (`MPI_Send`) и функцию приема сообщений (`MPI_Recv`).

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

Первые три аргумента `buffer, count, datatype` описывают сообщение в обычном для MPI стиле: адрес данных, их количество и тип. Следующий аргумент, `dest`, это адрес получателя, целое число, означающее номер процесса получателя в группе, определяемой коммуникатором, который задан параметром `comm`. Следующий аргумент — целое число, задающее тип сообщения или его *тег*, по терминологии MPI. Тег используется для передачи дополнительной информации вместе с передаваемыми данными. MPI гарантирует, что допустимыми значениями тега могут быть целые числа от 0 до 32767.

Сообщения от подчиненных процессов принимаются с помощью функции

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

Следует отметить, что получение сообщения происходит с блокировкой, т.е. управление не возвращается программе до тех пор,

---

[3] Напомним, что ранг процесса уникален в рамках группы (коммуникатора)

пока сообщение не будет принято. Первые три аргумента указывают, куда и в каком виде будет помещено сообщение. Процесс также может указать, что он будет ждать сообщения от какого-либо определенного процесса - в таком случае нужно указать ранг этого процесса в качестве параметра source. Если этого не нужно, можно использовать определенное в MPI значение `MPI_ANY_SOURCE`, чтобы указать, что процесс будет принимать сообщение от любого процесса из группы коммуникатора `comm`. Использование определенного в MPI значения `MPI_ANY_TAG` в качестве параметра `tag` указывает, что приемлемыми будут сообщения с любым тегом. Если необходимо принимать сообщения только с определенным тегом в качестве параметра `tag`, нужно использовать его.

Выходной параметр `stat` содержит информацию о принятом сообщении. Это структура типа `MPI_Stat`. Он должен быть определен в программе пользователя. Проанализировав значение полей `stat`, можно узнать ранг процесса, отправившего данные `stat.MPI_SOURCE`, и тег `stat.MPI_TAG`. Другие элементы `stat` служат для определения числа единиц реально полученных данных с использованием функции `MPI_Get_count`.

Итак, после определения своего ранга процесс в зависимости от него выполняет следующие действия. Все процессы, с рангами, отличными от нуля, передают на нулевой процесс свой ранг. Нулевой процесс в цикле принимает сообщения от любого процесса с любым тегом. Количество итераций цикла на 1 меньше количества процессов в группе (печать ранга нулевого процесса осуществляется до цикла).

Следует отметить также, что, несмотря на наличие в программе множества процессов, в каждый текущий момент времени в элементарном цикле обмена участвуют два процесса - отправитель и получатель (операция передачи "процесс-процесс"). В разные моменты времени в качестве отправителей выступают разные процессы.

Откомпилировав и запустив этот пример, мы можем получить, например, следующий вывод (было запущено четыре процесса):

```
Hello from process 0
Hello from process 2
Hello from process 1
Hello from process 3
```

Несложно заметить, что сообщения от процессов выводятся "не по порядку". Причем, порядок вывода сообщений может изменяться от

запуска к запуску. Причина происходящего кроется в том, как в приведенном примере описан цикл приема сообщений. В самом деле, нулевой процесс при приеме не указывает явно, с какого процесса он принимает сообщения. Передавая в качестве ожидаемого источника константу MPI_ANY_SOURCE, нулевой процесс принимает сообщение от **любого** другого процесса. Поскольку, будучи запущенными на разных машинах, процессы стартуют и выполняются с разными скоростями, порядок, в котором процессы будут входить в функцию передачи сообщений MPI_Send, предсказать сложно.

Приведенный пример легко модифицировать таким образом, чтобы обеспечить последовательный прием данных от процессов в порядке возрастания их рангов. Поскольку скоростью выполнения процессов (выполняющихся, возможно, на разных машинах) мы управлять не можем, для решения поставленной задачи мы изменим цикл приема сообщений. На каждой итерации цикла мы будем принимать сообщение от конкретного процесса.

Приведенный ниже код иллюстрирует эту технику:

```
        #include <stdio.h>
        #include "mpi.h"
        int main(int argc, char* argv[]){
          int rank, n, i, message;
          MPI_Status status;
          MPI_Init(&argc, &argv);
          MPI_Comm_size(MPI_COMM_WORLD, &n);
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     If (rank == 0){
            Printf ("\n Hello from process %3d",
     rank);
            for (i=1; i<n; i++){
            MPI_Recv(&message, 1, MPI_INT, i,
     MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
              Printf("\n Hello from process %3d",
     message);
       }
   }else
MPI_Send(&rank,1,MPI_INT,0,0,MPI_COMM_WORLD);
         MPI_Finalize();
           Return 0;
         }
```

Единственное отличие этой программы от предыдущей состоит в том, что в цикле приема сообщений вместо константы `MPI_ANY_SOURCE` в функцию `MPI_Recv` передается переменная `i`. Однако, если мы будем запускать эту модифицированную программу, порядок вывода сообщений всегда будет одним и тем же.

```
Hello from process 0
Hello from process 1
Hello from process 2
Hello from process 3
```

3.2. Основные типы операций передачи данных

Рассмотренные нами операции `MPI_Send` и `MPI_Recv` являются основными для обменов типа "процесс - процесс". Однако, их использование в "настоящих" (не учебных) программах сопряжено с известными трудностями и (в большинстве случаев) приводит к невысокой производительности полученных параллельных программ.

Все дело в том, каким образом происходит обмен сообщениями при использовании стандартных процедур передачи данных. Вообще в MPI поддерживаются следующие разновидности операций обмена:

- *Стандартная передача*. Считается завершенной, как только сообщение отправлено, независимо от того, получено оно или нет. Передача сообщения может начинаться, даже если не начался его прием.
- *Синхронная передача*. Не завершается до тех пор, пока не будет завершена приемка сообщения.
- *Буферизированная передача*. Завершается сразу же. При выполнении буферизированной передачи управление сразу же возвращается в вызывающую процедуру - сообщение при этом копируется в системный буфер и ожидает пересылки.
- *Передача по готовности*. Начинается только в том случае, если начата приемка сообщения и сразу завершается, не дожидаясь окончания приема.

Кроме того, каждый из этих четырех режимов может быть выполнен в блокирующей или неблокирующей форме[4]. Таким образом, всего в MPI 8 разновидностей операций передачи.

В MPI принято определенное соглашение об именах процедур, позволяющее определять тип используемой операции. Имя процедуры строится по следующему правилу:

`MPI_[I][R|S|B]Send`,

где

- `I (Immediate)` - обозначает неблокирующую операцию;
- `R` - передача по готовности;
- `S` – синхронный;
- `B` – буферизированный.

Отсутствие префикса означает, что используется стандартный режим. Следует отметить, что функция приема любого типа (`MPI_XXRecv`) способна принять сообщение, отосланное с помощью любой функции (`MPI_XXSend`).

Таким образом, функция `MPI_Send` выполняет стандартную блокирующую передачу. Она блокирует процесс до завершения операции передачи сообщения (это не гарантирует завершения приема!), что крайне неэффективно при пересылке большого массива данных, так как процессор, вместо того, чтобы выполнять вычисления, простаивает. То же самое относится и к функции `MPI_Recv`.

Таким образом, чтобы избежать замедления исполнения, нужно использовать другой способ организации работы - позволять пользователям начинать посылку (и получение) нескольких сообщений и продолжать выполнение других операций. MPI поддерживает этот подход, предоставляя неблокирующие посылки и прием.

3.3. Неблокирующий обмен

Неблокирующим аналогом `MPI_Send` является функция `MPI_Isend`.

---

[4] Блокирующий прием (передача) приостанавливает процесс на время приема сообщения

Функция MPI_Isend начинает неблокирующую операцию передачи. Аргументы этой процедуры такие же, как у MPI_Send, с добавлением handle вслед за последним аргументом. Обе функции (MPI_Send и MPI_Isend) ведут себя одинаково, за исключением того, что в случае MPI_Isend, буфер с посылаемым сообщением не должен модифицироваться до тех пор, пока сообщение не будет доставлено (более точно, до завершения операции, что можно выяснить с помощью процедур MPI_Wait или MPI_Test). Причины такого ограничения вполне понятны: в течение операции передачи данные не должны изменяться. Параметр handle имеет тип MPI_Request и используется для того, чтобы определить, было ли сообщение доставлено. Необходимая проверка осуществляется с помощью функции MPI_Test. Ниже представлен фрагмент кода, иллюстрирующий такую операцию:

```
...
MPI_Isend(buffer,count,datatype,dest,tag,comm,request
)
//продолжаем вычисления. buffer не должен модифицироваться.
...
//дожидаемся конца пересылки
While(!flag){
MPI_Test(request, flag, status)
}
```

Здесь сначала выполняется операция неблокирующей посылки данных, затем процесс, не дожидаясь окончания операции, продолжает вычисления, после чего входит в цикл ожидания завершения операции.

Часто возникает необходимость ждать завершения передачи. В этом случае вместо того, чтобы писать цикл с процедурой MPI_Test, используемой в предыдущем примере, можно использовать MPI_Wait:

```
MPI_Wait(request, status).
```

Процедура MPI_Irecv начинает неблокирующую операцию приема. Она имеет один дополнительный аргумент, handle, точно так же, как MPI_Isend. В то же время она имеет на один аргумент *меньше*: status-аргумент, который используется для возврата информации по окончанию приема, исключен из списка аргументов. Точно так же, как для MPI_Isend, для проверки завершения приема, начатого MPI_Irecv, можно использовать MPI_Test, а для ожидания завершения такого приема - MPI_Wait. Аргументы status этих двух

процедур возвращают информацию о завершенном приеме в том же виде, что и MPI_Recv для блокирующего приема.

Во многих случаях необходимо проверить или ждать завершения многих неблокирующих операций. Хотя можно просто циклически исполнять несколько таких операций, такой подход неэффективен, так как он приводит к бесцельной трате процессорного времени. MPI обеспечивает ожидание всех или какой-либо из набора неблокирующих операций (с помощью MPI_Waitall и MPI_Waitany) и проверки всех или какой-либо из набора неблокирующих операций (с помощью MPI_Testall и MPI_Testany). Например, чтобы начать два неблокирующих приема и затем ожидать их завершения, можно использовать:

```
MPI_Irecv( ..., requests[1]);
MPI_Irecv( ..., requests[2]);
...
MPI_Waitall( 2, requests, status);
```

Здесь status должен быть массивом из двух MPI_status объектов, он объявляется как

```
MPI_status status[2]
```

3.4. Синхронный блокирующий обмен

MPI предоставляет способ передачи сообщения, при котором возврат из процедуры передачи не происходит до тех пор, пока адресат не начнет принимать сообщение. Это процедура MPI_Ssend. Аргументы этой процедуры идентичны MPI_Send. Отметим, что MPI разрешает реализацию MPI_Send в виде MPI_Ssend; таким образом, максимальная переносимость гарантируется тем, что любое использование MPI_Send может быть замещено MPI_Ssend**.** Если процесс выполняет блокирующую синхронную передачу, то он будет приостановлен до приема адресатом сообщения. Синхронный обмен сообщениями происходит медленнее, чем стандартный, но он является более надежным и предпочтительным в плане обеспечения предсказуемости выполнения программ.

3.5. Буферизованный обмен

Буферизованная передача выполняется с использованием функции:

```
int MPI_Bsend(void* buffer,int count,MPI_Datatype
datatype,int dest,int tag,MPI_Comm comm).
```

Параметры идентичны `MPI_Send`. Управление процессу передается сразу же после копирования сообщения в буфер. Буфер для копирования данных выделяется системой или создается программистом при помощи функции:

```
int MPI_Buffer_attach(void* buffer,int size).
```

После завершения работы буфер уничтожается:

```
int MPI_Buffer_detach(void* buffer,int size).
```

Память для копирования сообщений должна быть достаточного размера; при переполнении буфера возникает ошибка.

3.6. Обмен по готовности

Обмен по готовности должен начинаться только в том случае, если начата соответствующая процедура приема - в противном случае, результат функции не определен. Завершается процедура сразу. Передача по готовности осуществляется с помощью процедуры

```
int MPI_Rsend(void* buffer, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm).
```

Параметры идентичны `MPI_Send`.

3.7. Выполнение операций приема и передачи одной функцией

В MPI есть группа интересных процедур, совмещающих функции приема и передачи. Они достаточно часто применяются при программировании "каскадных" или "линейных" схем, когда необходимо осуществлять обмен однотипными данными между процессорами. Примером является функция:

```
int MPI_Sendrecv (void* sendbuffer, int sendcount,
MPI_Datatype senddatatype, int dest, int sendtag, void*
recvbuffer, int recvcount, MPI_Datatype recvdatatype,
int src, int recvtag MPI_Comm comm, MPI_Status* status),
```

где

- `sendbuffer` - адрес массива передаваемых данных;

- `sendcount` - количество элементов в массиве;
- `senddatatype`- тип передаваемых элементов;
- `dest` - ранг адресата;
- `sendtag` - тег передаваемого сообщения;
- `recvbuffer` - адрес буфера для приема;
- `recvcount` - количество элементов в буфере приема;
- `recvdatatype` - тип элементов в буфере приема;
- `src` - ранг источника;
- `recvtag` - тег принимаемого сообщения;
- `comm` – коммуникатор;
- `status` - структура с дополнительной информацией.

Функция копирует данные из массива `sendbuffer` процесса с рангом `src` в буфер `recvbuffer` процесса с рангом `dest`.

Другая полезная функция:

```
int MPI_Sendrecv_replace (void* buffer, int count, MPI_Datatype datatype, int dest, int sendtag, int src, int recvtag MPI_Comm comm, MPI_Status* status).
```

Использует только один буфер, также передавая данные с процесса `src` на процесс `dest`. Примеры использования этих и других функций см. в разделе 7. Примеры параллельных программ.

4. Типы данных

При рассмотрении функций передачи сообщений MPI в качестве одного из аргументов этих функций передавался *тип данных* для посылаемых и получаемых сообщений. В рассмотренных до этого примерах использовались только элементарные типы данных, которые соответствуют базовым типам данных в используемом языке программирования - целым числам, числам с плавающей точкой и т.п., а также их массивам. В этом разделе мы обсудим весь набор базовых типов и, кроме того, рассмотрим способы конструирования производных типов MPI.

4.1. Базовые типы данных в MPI

MPI предоставляет широкий набор элементарных типов данных. Этот набор включает все базовые типы Си, а также два типа данных, специфичных для MPI: `MPI_BYTE` и `MPI_PACKED`. `MPI_BYTE` описывает *байт*, который определяется как 8 двоичных цифр. Этот тип не тождественен `MPI_CHAR` и `MPI_CHARACTER`. Во-первых, `MPI_BYTE` гарантированно представляет числа в диапазоне [0..255], в то время как char, в зависимости от реализации, может быть и 16-битным. Вторая причина в том, что в гетерогенной (неоднородной) среде машины литеры могут иметь различные кодировки. Например, система, использующая ASCII, имеет иное битовое представление литеры A, чем система, использующая EBCDIC. В **Таблица 1** представлено соответствие базовых типов MPI и типов данных языка Си.

Таблица 1. Базовые (предопределенные) типы данных MPI для Си

MPI Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	Double
MPI_FLOAT	Float
MPI_INT	Int
MPI_LONG	Long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	Short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

4.2. Типы данных MPI и типы данных в языках программирования

Тип данных в MPI - это объект, который определяет цепочку базовых типов данных и смещений в байтах каждого из этих типов. Такие смещения отсчитываются относительно буфера, который описывает тип данных. Мы будем представлять тип данных последовательностью пар базовых типов и смещений, в MPI эта последовательность называется *картой типа* (*typemap*).

Typemap= $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$

Например, тип MPI_INT может быть представлен картой типа (int,0).

Сигнатура типа некоего типа данных - это список базовых типов:

Type signature = $\{type_0, \ldots, type_{n-1}\}$

Сигнатура типа описывает, какие базовые типы данных образуют некоторый производный тип данных MPI, то есть сигнатура типа управляет интерпретацией элементов данных при посылке или получении. Другими словами, она сообщает MPI, как интерпретировать биты в буфере данных. Смещения сообщают MPI, где находятся эти биты. Чтобы понять, как MPI транслирует определенные пользователем типы данных, необходимо ввести несколько терминов. Допустим, тип данных MPI имеет карту типа $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$.

Введем ряд определений:

- *нижняя граница* типа

$$lb(\text{Typemap}) = \min_j (disp_j),$$

- *верхняя граница* типа

$$ub(\text{Typemap}) = \max_j (disp_j + sizeof(type_j)),$$

- *протяженность* типа

$$extent(\text{Typemap}) = ub(\text{Typemap}) - lb(\text{Typemap}) + pad.$$

Нижнюю границу можно рассматривать как расположение первого байта описанного типа данных. *Верхнюю границу* типа данных (предполагается, что адреса возрастают снизу вверх) можно рассматривать как адрес байта, располагающегося вслед за последним байтом описанного типа данных. И, наконец, протяженность между ними – это разница между этими двумя величинами, возможно, увеличенная, чтобы удовлетворить условию выравнивания. Оператор sizeof(type) возвращает размер *базового* типа в байтах.

Для понимания роли "pad" нам нужно обсудить *выравнивание* данных. Как Си, так и Фортран требуют, чтобы базовые типы были должным образом выровнены; то есть, чтобы числа, к примеру, целые или двойной точности, располагались только по тем адресам в памяти, где это допустимо. Каждая реализация этих языков определяет это "где

допустимо" (конечно, имеются какие-то ограничения). Одно из самых общих требований, которые налагают реализации этих языков, - это требование, чтобы адрес элемента в байтах был кратен длине этого элемента в байтах. Например, если `int` занимает четыре байта, то адрес `int` должен нацело делиться на четыре. Это требование отражается в определении протяженности типа данных MPI.

Рассмотрим карту типа

$\{(int,0),(char,4)\}$

на компьютере, который требует, чтобы целые были выровнены по 4-байтовой границе. Такая карта типа имеет $lb=\min(0,4)=0$ и $ub=\max(0+4,4+1)=5$. Но следующий `int` может быть размещен со смещением восемь от `int` в карте типа. Таким образом, протяженность рассматриваемой карты типа равна восьми на рассмотренном компьютере (более строго, протяженность служит для определения расположения "следующего" элемента указанного типа в буфере сообщения, заданного адресом, количеством и типом, т.е. является расстоянием (в байтах) между двумя последовательными элементами производного типа).

Для нахождения протяженностии типа данных MPI предоставляет процедуру `MPI_Type_extent`. Первым аргументом является тип данных MPI; протяженность возвращается во втором аргументе. В Си тип второго аргумента есть `MPI_Aint`; это - целый тип, который может вмещать произвольный адрес (на многих, но не всех системах, это будет `int`). Протяженность базовых типов равна числу байтов в них.

Размер типа данных - это число байтов, которые занимают данные. Он выдается с помощью `MPI_Type_size`; первым аргументом является тип данных, а размер возвращается во втором аргументе. Различие между протяженностью и размером типа данных иллюстрируется приведенной картой типа: размер равен пяти байтам, а протяженность (на компьютере, требующем выравнивания для целых по 4-байтовым границам) равняется восьми байтам. Процедурами получения границ типа данных MPI являются `MPI_Type_ub` - для получения верхней границы, и `MPI_Type_lb` - для получения нижней границы. Спецификации описанных здесь процедур для типов данных, даются в таблице 2.

Таблица 2. Функции MPI для получения характеристик производных типов данных

`int MPI_Type_contiguous (int count, MPI_Data_type oldtype, MPI_Datatype *newtype)`
`int MPI_Type_extent (MPI_Datatype datatype, MPI_Aint *extent)`
`int MPI_Type_size (MPI_Datatype datatype, MPI_Aint *size)`
`int MPI_Type_count (MPI_Datatype datatype, int *count)`
`int MPI_Type_lb (MPI_Datatype datatype, MPI_Aint *displacement)`
`int MPI_Type_ub (MPI_Datatype datatype, MPI_Aint *displacement)`

4.3. Определение пользовательских типов данных

Карта типа - это общий способ описания произвольных типов данных. Однако он может быть неудобен, особенно, если полученная карта типа содержит большое число элементов. MPI представляет несколько способов создания типов данных без явного конструирования карты типа.

- ***Непрерывный***: Это простейший способ конструирования нового типа. Он производит новый тип данных из `count` копий существующего со смещениями, увеличивающимися на протяженность `oldtype` (исходного типа). Простейший пример - массив элементов какого-то уже определенного типа.
- ***Векторный***: Это несколько обобщенный непрерывный тип, который допускает регулярные пробелы в смещениях. Элементы отделяются промежутками, кратными протяженности исходного типа данных. Пример - столбцы матрицы, при определении матрицы как двумерного массива;
- ***Н-векторный***: Похож на векторный, но расстояние между элементами задается в байтах.
- ***Индексный***: При создании этого типа используется массив смещений исходного типа данных; смещения

измеряются в терминах протяженности исходного типа данных.

- ***H-индексный***: Похож на индексный, но смещения задаются в байтах.
- ***Структурный***: Обеспечивает самое общее описание. Фактически, если исходные аргументы состоят из базовых типов MPI, задаваемая структура есть в точности карта создаваемого типа.

Рассмотрим более подробно создание непрерывного типа данных. Непрерывный тип создается с использованием функции `MPI_Type_contiguous`, производящей новый тип данных, делая `count` копий исходного со смещениями, увеличивающимися на протяженность `oldtype`. Например, если исходный тип данных (`oldtype`) имеет карту типа `{(int,0),(double,8)}`, то

```
MPI_Type_contiguous (2, oldtype, &newtype);
```

произведет тип данных `newtype` с картой типа

`{(int,0),(double,8),(int,16),(double,24)}`.

Использование аргумента `count` в процедурах MPI равносильно использованию непрерывного типа данных такого же размера. То есть,

```
MPI_Send (buffer, count, datatype, dest, tag, comm);
```

это то же самое, что

```
MPI_Type_contiguous (count, datatype, &newtype);
MPI_Type_commit (&newtype);
MPI_Send (buffer, 1, newtype, dest, tag, comm);
MPI_Type_free (&newtype);
```

4.4. Использование определенных пользователем типов данных

При решении широкого круга научно-технических задач используются алгоритмы, требующие для своей реализации выполнения тех или иных операций с матрицами. С точки зрения программиста, матрица представляет собой двумерный массив вида

```
DataType[N][M]
```

с возможностью доступа к элементам, строкам и столбцам. При написании же параллельной программы часто возникают сложности,

связанные с тем, что логически единый элемент модели - столбец матрицы - расположен в памяти не непрерывным образом (речь идет о реализации массивов в языке С). Таким образом, для реализации некоторых алгоритмов (перемножение двух матриц, например) необходимо уметь пересылать данные, расположенные в памяти некоторым регулярным, но не непрерывным образом.

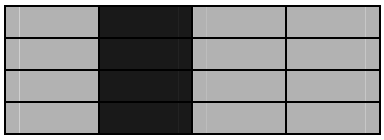

Рис. 1. Логическое представление матрицы

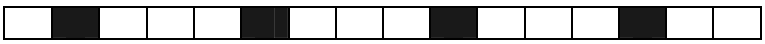

Рис. 2. Представление матрицы в памяти

На рис. 2 тпоказано, например, размещение в памяти второго столбца матрицы. Для передачи данных такого рода идеально

подходит рассмотренный выше векторный тип данных, создаваемый при помощи вызова

```
int MPI_Type_vector (int count, int blocklength, int
stride, MPI_Datatype oldtype, MPI_Datatype *newtype),
```

где

- `count` – количество блоков;
- `blocklen` – длина каждого блока;
- `stride` – количество элементов между блоками;
- `oldtype` - базовый тип.

В нашем случае, `count = 4, stride = 4, blocklen = 1.` Ниже приведена часть программного кода, создающего новый тип данных (столбец матрицы) и организующего обмен сообщениями этого типа.

```
…
double matrix[NUMCOLS][NUMROWS];
MPI_Datatype columntype;
int rank;
int i;
```

```
   int numprocs;
   MPI_Status status;
   …
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   …
   MPI_Type_vector(NUMROWS,1,NUMCOLS, MPI_DOUBLE,
&columntype ); //1
   MPI_Type_commit(&columntype);
                     //2
   …
   if (0==rank){
       for (i=1; i< numprocs; i++)
       MPI_Recv(&matrix[0][i],1,columntype,i,MPI_ANY_TAG,
MPI_COMM_WORLD,&status);
                        //3
   } else
MPI_Send(&matrix[0][rank],1,columntype,0,0,MPI_COMM_WORL
D);    //4
   …
   MPI_Type_free(&columntype);
                        //5
   …
```

Вызовы 1 и 2 создают новый тип данных и фиксируют его. Затем все процессы, кроме нулевого, высылают по одному столбцу матрицы на нулевой процесс. Высылается столбец с номером, соответствующим рангу процесса (вызов 4). На нулевом процессе организуется цикл приема сообщений от всех процессов, результат принимается в столбец, соответствующий рангу процесса, от которого было принято сообщение (вызов 3). Следует обратить внимание на то, что создание типа (вызовы 1 и 2), а также освобождение типа (вызов 5) должны происходить во всех процессах, которые участвуют в обмене сообщениями.

5. Коллективные операции

До сих пор рассматривались взаимодействия, в которых участвуют два процесса - один передает сообщение, другой принимает. MPI поддерживает более сложные типы взаимодействий, которые предполагают участие нескольких (возможно, более двух) процессов. Такие операции называют коллективными. Коллективные операции

широко используются при написании программ MPI. Это связано не только с естественностью таких операций (часто используемые алгоритмы предполагают рассылку данных всем вычислительным процессам, или наоборот, сборку рассчитанных данных на корневом процессе), но и высокой эффективностью таких операций. Однако, прежде чем перейти к рассмотрению этих операций MPI, необходимо более глубоко ознакомиться с понятием коммуникатора.

5.1. Коммуникаторы

Как уже говорилось ранее, весь обмен данных в MPI осуществляется в рамках коммуникаторов, которые определяют контекст обмена и группу процессов, с ними связанных. До сих пор мы использовали только предопределенный коммуникатор `MPI_COMM_WORLD`, определяющий единый контекст и совокупность всех выполняющихся процессов MPI. В MPI имеются средства создания и изменения коммуникаторов, предоставляющие возможность программисту создавать собственные группы процессов, что делает возможным использование в программах достаточно сложных типов взаимодействия. Типичной проблемой, которую может решить использование коммуникаторов, является проблема недопущения "пересечения" обменов по тегам. В самом деле, если программист использует какую-то библиотеку параллельных методов, то он часто не знает, какие теги использует библиотека при передаче сообщений. В таком случае существует опасность, что тег, выбранный программистом, совпадет с одним из тегов, используемых библиотекой. Чтобы этого не произошло, программист может создать собственный коммуникатор и выполнять операции обмена в его рамках. Несмотря на то, что этот коммуникатор будет содержать те же процессы, что и коммуникатор библиотеки, обмены в рамках этих двух разных коммуникаторов "не пересекутся".

5.2. Управление группами

С понятием коммуникатора тесно связано понятие группы процессов. Под *группой* понимают упорядоченное множество процессов. Каждому процессу в группе соответствует уникальный номер - *ранг*. Группа - отдельное понятие MPI, и операции с группами

могут выполняться отдельно от операций с коммуникаторами, но операции обмена для указания области действия всегда используют **коммуникаторы, а не группы**. Таким образом, один процесс или группа процессов могут входить в несколько различных коммуникаторов. С группами процессов в MPI допустимы следующие действия:

- Объединение групп;
- Пересечение групп;
- Разность групп.

Новая группа может быть создана только из уже существующих групп. В качестве исходной группы при создании новой может быть использована группа, связанная с предопределенным коммуникатором `MPI_COMM_WORLD`. При ручном конструировании групп может оказаться полезной специальная пустая группа `MPI_COMM_EMPTY`.

Для доступа к группе, связанной с коммуникатором `comm`.,. используется функция

```
int MPI_Comm_group (MPI_Comm comm,MPI_Group *group).
```

Возвращаемый параметр `group` - группа, связанная с коммуникатором. Далее, для конструирования новых групп могут быть применены следующие действия:

– создание новой группы `newgroup` из группы `oldgroup`, которая будет включать в себя `n` процессов, ранги которых задаются массивом `ranks`

```
int MPI_Group_incl(MPI_Group oldgroup, int n, int *ranks, MPI_Group *newgroup);
```

– создание новой группы newgroup из группы `oldgroup`, которая будет включать в себя `n` процессов, ранги которых не совпадают с рангами, перечисленными в массиве `ranks`

```
int MPI_Group_excll(MPI_Group oldgroup, int n, int *ranks, MPI_Group *newgroup);
```

– создание новой группы `newgroup` как разность групп `group1` и `group2`

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
```

– создание новой группы `newgroup` как пересечение групп `group1` и `group2`

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
```

– создание новой группы `newgroup` как объединение групп `group1` и `group2`

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
```

– удаление группы `group`

```
int MPI_Group_free(MPI_Group *group).
```

Получить информацию об уже созданной группе можно, используя следующие функции:

– получение количества процессов в группе

```
int MPI_Group_size(MPI_Group group, int *size);
```

– получение ранга текущего процесса в группе

```
int MPI_Group_rank(MPI_Group group, int *rank).
```

5.3. Управление коммуникаторами

В MPI различают два вида коммуникаторов – *интракоммуникаторы*, используемые для операций внутри одной группы процессов, и *интеркоммуникаторы* - для обмена между группами процессов. В большинстве случаев используются интракоммуникаторы, поэтому далее говориться будет только о них.

Создание коммуникатора - операция коллективная (и должна вызываться всеми процессами коммуникатора). Вызов `MPI_Comm_dup` копирует уже существующий коммуникатор `oldcom`

```
int MPI_Comm_dup (MPI_Comm oldcom,MPI_comm *newcomm),
```

в результате будет создан новый коммуникатор newcomm, включающий в себя те же процессы.

Для создания нового коммуникатора служит функция `MPI_Comm_create`.

```
int MPI_comm_create (MPI_Comm oldcom, MPI_Group group, MPI_Comm *newcomm)
```

Вызов создает новый коммуникатор `newcomm`, который будет включать в себя процессы группы group коммуникатора `oldcomm`.

Рассмотрим пример простой программы.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char* argv[]){
    MPI_Group systemgroup;
    MPI_Group mygroup;
    MPI_Comm newcomm;
    int size,rank;
    int[2] ranks;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_group(MPI_COMM_WORLD,systemgroup);
    // получили группу процессов, связанную с
    // коммуникатором MPI_COMM_WORLD
    // в эту группу  входят все процессы
    ranks[0] = 1;
    ranks[1] = 3;
    MPI_Group_incl(systemgroup,2,ranks,&mygroup);
    //новая группа содержит 2 процесса  - 1 и 3
    MPI_Comm_create(MPI_COMM_WORLD,mygroup,&newcomm);
    //коммуникатор newcomm полностью готов к работе
    MPI_Comm_free(&newcomm);
    MPI_Group_free(&mygroup);
    MPI_Finalize();
}
```

В программе создается новый коммуникатор, включающий в себя процессы с рангами 1 и 3 из "старого" коммуникатора - `MPI_COMM_WORLD`. Стоит заметить, что теперь у 3-го процесса два ранга - один глобальный (в рамках коммуникатора `MPI_COMM_WORLD`), а другой - в рамках только что созданного коммуникатора `newcomm`.

Зачем нужны созданные пользователем коммуникаторы? Одна причина, побуждающая их использовать, уже называлась - без них не обойтись разработчикам библиотек. Вторая причина, по которой приходится создавать собственные коммуникаторы, состоит в том, что только с их помощью можно управлять более сложными, чем "точка - точка" типами обменов.

5.4. Передача данных от одного процесса всем. Широковещательная рассылка

При программировании параллельных задач часто возникает необходимость разослать какую-то порцию данных **всем процессам сразу.** Очевидно, что для решения этой задачи можно воспользоваться рассмотренными ранее операциями двупроцессного обмена.

```
MPI_Comm_size(MPI_COMM_WORLD,&size);
for (i=1; i<size; i++)
    MPI_Send(&buf,buflen,MPI_INT,i,0,MPI_COMM_WORLD);
```

Однако, такое решение неэффективно вследствие значительных затрат на синхронизацию процессов. Поэтому в MPI появилась специальная операция - операция широковещательной рассылки

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

Операция предполагает рассылку данных из буфера `buffer`, содержащего `count` элементов типа `datatype` с процесса, имеющего номер `root`, всем процессам, входящим в коммуникатор `comm`. Для пояснения сказанного рассмотрим простую задачу. Пусть нам требуется вычислить скалярное произведение двух векторов, причем вектора вводятся на нулевом процессе (например, считываются из файла). Ниже представлен текст соответствующей программы:

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

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

    double x[100], y[100];
double res, p_res = 0.0;

    MPI_Status status;
    int  n, myid, numprocs, i, N;
    N = 100;
    /* инициализация MPI */
```

```
   /* выяснение числа процессов и рангов */

       MPI_Init(&argc,&argv);
       MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
       MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   /* ввод векторов */
   if (0 == myid) read_vectors(x,y)
   /*рассылка векторов на все процессы*/
   MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(y, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   /*вычисление частичной суммы на каждом из процессов*/
   for (i = myid * N/numprocs; i < (myid + 1)*N/
numprocs; i++)
            p_res  = p_res + x[i] * y[i];
       /* сборка на 0-м процессе результата*/
       if (0 == myid){
                for (i=1; i< numprocs; i++)
                     MPI_Recv(&res, 1, MPI_DOUBLE, i,
     MPI_ANY_TAG,
                              MPI_COMM_WORLD, &status);
                     p_res = p_res+res;

   }else /*все процессы отсылают частичные суммы на 0-й*/
         MPI_Send(&p_res,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD
       );
       /*на 0-м печатаем результат*/
   if (0 == myid) printf("\nInner product =
%10.2f",p_res);
   MPI_Finalize();
   }
```

В рассмотренной программе сначала производится инициализация MPI. Затем выясняются ранг процесса и общее количество процессов. Затем на 0-м процессе происходит чтение данных и заполнение векторов. Нужно обратить внимание, что только на 0-м процессе массивы x и y содержат данные - на всех остальных процессах в этих массивах находятся не инициализированные данные! 0-й процесс рассылает данные на все остальные процессы. Это обеспечивается двумя вызовами функции `MPI_Bcast` - первый вызов рассылает вектор `x`, второй вызов - вектор `y`. Следует обратить внимание на следующие обстоятельства:

– Вызов функции `MPI_Bcast` присутствует во всех процессах. Таким образом, на 0-м процессе этот вызов обеспечивает отправку данных, а на остальных - их приемку. Ранг процесса, с которого осуществляется рассылка данных, задается параметром `root`;

– Данные отправляются из области памяти `buffer` (в нашем случае - из массива `x`) и принимаются в ту же область памяти, только на другом процессе;

– Данные рассылаются всем процессам в рамках коммуникатора.

Таким образом, если необходимо, например, на одну часть процессов разослать одно, а на другую - другое, необходимо будет создать два коммуникатора и выполнять соответствующую операцию широковещательной рассылки в каждом из них.

5.5. Передача данных от всех процессов одному. Операции редукции

В рассмотренной в предыдущем разделе задаче возникает еще одна проблема - собрать рассчитанные на каждом из процессов частичные суммы на 0-м процессе. Приведенный выше код решает эту проблему неэффективно - с помощью уже знакомого цикла приема сообщений от всех процессов. MPI предоставляет возможность решить эту задачу по-другому - используя обратную по отношению к широковещательной рассылке операцию - *операцию сбора данных* или *редукцию*. Операция редукции позволяет, собрав на одном из узлов данные, посланные остальными узлами, выполнить над ними какую-либо из групповых операций - типа сложения, поиска максимума, минимума, среднего значения и т.д.

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

где:

– `sendbuf` - данные, которые посылаются на каждом из процессов;

– `recvbuf` -адрес буфера, в который будет помещен результат;

– `count` - количество элементов в буфере sendbuf;

– `datatype` - тип элементов в буфере sendbuf;

– `op` - коллективная операция, которая должна быть выполнена над данными;

– `root` - ранг процесса, на котором должен быть собран результат;

– `comm` - коммуникатор, в рамках которого производится коллективная операция.

В качестве коллективных операций можно использовать предопределенные в MPI операции, такие как `MPI_SUM`, `MPI_MIN`, `MPI_MAX` и другие (всего их 12). Кроме того, имеется возможность определить свою операцию[5]. С учетом всего вышеизложенного, предыдущую программу можно переписать следующим образом:

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

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

    double x[100], y[100];
double res, p_res = 0.0;

    MPI_Status status;
    int  n, myid, numprocs, i, N;
    N = 100;
    /* инициализация MPI */
/* выяснение числа процессов и рангов */

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
/* ввод векторов */
if (0 == myid) read_vectors(x,y)
/*рассылка векторов на все процессы*/
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(y, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
/*вычисление частичной суммы на каждом из процессов*/
```

[5] Рассмотрение этого вопроса выходит за рамки настоящего пособия. Скажем лишь, что определение пользовательской операции выполняется с помощью функции MPI_Op_create.

```
   for (i = myid *N/numprocs; i < (myid + 1)*N/numprocs;
i++)
              p_res  = p_res + x[i] * y[i];
       /* сборка на 0-м процессе результата*/
       MPI_Reduce(p_res,res,N/numprocs,MPI_DOUBLE,MPI_SUM
,0,
          MPI_COMM_WORLD)
   if (0 == myid) printf("\nInner product =
%10.2f",res);
   MPI_Finalize();
   }
```

Так же, как и в случае с `MPI_Bcast`, функция `MPI_Reduce` должна вызываться во всех процессах, участвующих в сборке данных. `Sendbuf` на всех процессах содержит адрес области данных, которые будут участвовать в редукции. Значение `recvbuf` вычисляется только на процессе с рангом `root`[6]. В случае, если значение `recvbuf` необходимо получить на всех процессах, вместо процедуры `MPI_Reduce` следует использовать процедуру `MPI_Allreduce` со сходным синтаксисом.

5.6. Распределение и сбор данных

При программировании часто возникает задача распределения массива данных по процессам некоторыми регулярными "кусками". Например, распределение матрицы, нарезанной вертикальными лентами. Возникает и обратная задача - сбор на некотором выделенном процессе некоторого набора данных, распределенного по всем процессам.

---

[6] Следует отметить, что выполнение операции обработки сообщений может осуществляться и в ходе передачи сообщений на транзитных процессорах. Функция MPI_Reduce гарантирует не получение всех отправляемых сообщений, а лишь наличие результата указываемой операции.

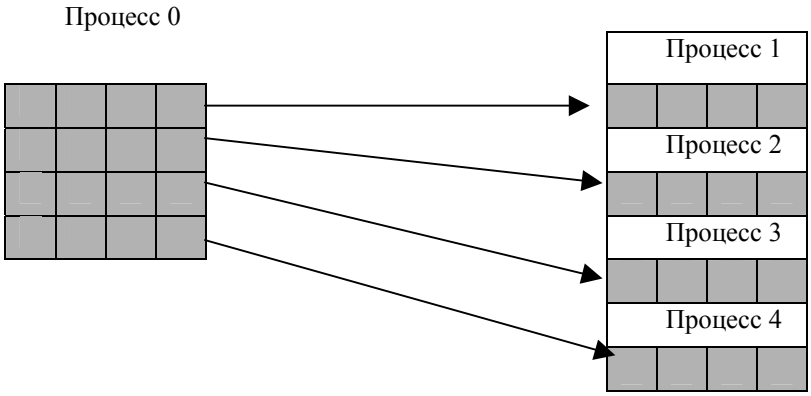

Рис. 3. Распределение данных

Распределение и сбор данных осуществляется с помощью вызовов процедур `MPI_Scatter` и `MPI_Gather`.

```
int MPI_Scatter(void* sendbuf, int sentcount, MPI_Datatype senddatatype, void* recbuf, int reccount, MPI_Datatype recdatatype,int root,MPI_Comm comm),
```

где:

- `sendbuf` - адрес буфера для передачи данных;
- `sentcount` - количество элементов, передаваемых на каждый процесс (общее количество элементов в буфере равно произведению `sentcount` на количество процессов в коммуникаторе);
- `senddatatype` - тип передаваемых данных;
- `recbuf` - буфер для приема данных;
- `reccount` - размер буфера `recbuf`;
- `recdatatype` - тип данных для приемки;
- `root` - ранг процесса, с которого рассылаются данные;
- `comm` – коммуникатор.

При вызове этой процедуры произойдет следующее. Процесс с рангом `root` произведет передачу данных всем другим процессам в коммуникаторе. Каждому процессу будет отправлено `sendcount` элементов. Процесс с рангом 0 получит порцию из `sendbuf`, начиная с

0-го и заканчивая sendcount-1 элементом. Процесс с рангом 1 получит порцию, начиная с sendcount, заканчивая 2* sendcount-1 и т.д.

Подпрограмма MPI_Gather собирает данные от остальных процессов.

```
int MPI_Gather(void* sendbuf, int sentcount, MPI_Datatype senddatatype, void* recbuf, int reccount, MPI_Datatype recdatatype,int root,MPI_Comm comm),
```

где:

- sendbuf - адрес буфера для передачи данных;
- sentcount - количество элементов, передаваемое на главный процесс;
- senddatatype - тип передаваемых данных;
- recbuf - буфер для приема данных;
- reccount - размер буфера recbuf;
- recdatatype - тип данных для приемки;
- root - ранг процесса, на котором собираются данные;
- comm – коммуникатор.

Посредством MPI_Gather каждый процесс в коммуникаторе передает данные из буфера sendbuf на процесс с рангом root. Этот "ведущий" процесс осуществляет склейку поступающих данных в буфере recbuf. Склейка данных осуществляется линейно, положение пришедшего фрагмента данных определяется рангом процесса, его приславшего. В целом процедура MPI_Gather обратна по своему действию процедуре MPI_Scatter.

Следует заметить, что при использовании MPI_Gather сборка осуществляется только на одном процессе. Во всех остальных процессах заполнение буфера recbuf не определено. Для некоторых задач необходимо, чтобы данные, рассчитанные на каждом из процессов, были собраны в единый объект опять же на каждом процессе. В таком случае, вместо функции MPI_Gather следует использовать функцию MPI_Allgather. При использовании функции MPI_Allgather на всех процессах в буфере recbuf будут собраны одинаковые данные - "большой" объект, полученный как объединение фрагментов, переданных с каждого из процессов.

Другая полезная процедура `MPI_Alltoall` пересылает данные по принципу "все - всем"

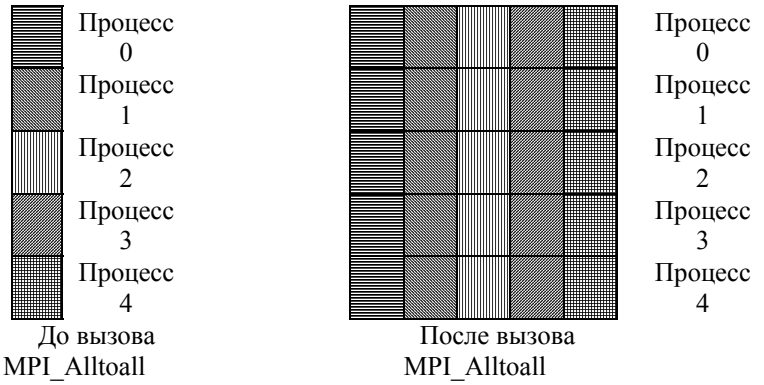

Рис. 4. Работа MPI_Alltoall

Кроме перечисленных, в MPI существует еще десяток функций, осуществляющих различные коллективные операции. При работе с ними следует помнить следующие основные моменты:

– Все коллективные операции выполняются в рамках коммуникатора. Если необходимо выполнить коллективную операцию над подмножеством процессов, следует создать для этой цели свой коммуникатор.

– Коллективные операции должны вызываться во всех процессах, которые в них участвуют.

– Разумное использование коллективных операций - хорошее средство повышения производительности.

Примеры подпрограмм распределения и сборки данных приведены в разделе 7. Примеры параллельных программ

6. Виртуальные топологии

Топология - дополнительный механизм MPI, который позволяет устанавливать дополнительную систему адресации для процессов. Топология в MPI является логической и никак не связана с топологией физической среды передачи данных. Введение в MPI поддержки топологий связано с тем, что большое число прикладных алгоритмов

устроено таким образом, что процессы оказываются упорядоченными в соответствии с некоторой топологией. В MPI поддерживаются два вида топологий - прямоугольная решетка произвольной размерности (декартова топология) и граф. Далее рассматриваются только декартовы топологии как наиболее часто используемые.

6.1. Решетки

Декартовы топологии часто применяются при решении прикладных задач. Известно большое количество алгоритмов, в которых используются "сетки". Один из наиболее широко представленных классов таких задач - сеточные методы решения дифференциальных уравнений в частных производных.

Для того, чтобы создать топологию вида "решетка", нужно воспользоваться функцией:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims, int *periods, int reoreder, MPI_Comm grid_comm),
```

где:

- `oldcomm` – исходный коммуникатор;
- `ndims` – размерность декартовой решетки;
- `dims` – целочисленный массив длины `ndims`, задает количество узлов в каждом измерении;
- `periods` – массив длины ndims, задает, является ли измерение замкнутым;
- `reoreder` - разрешено или нет системе менять нумерацию процессов.

Операция создания топологии является коллективной операцией - ее должны выполнить все процессы коммуникатора.

Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int MPI_Card_coords(MPI_Comm comm,int rank,int maxdims,int *coords),
```

где:

- `comm` – коммуникатор;

– `rank` - ранг процесса, для которого нужно определить координаты;

– `maxdims` - длина массива coords;

– `coords` - возвращаемые функцией декартовы координаты процесса.

Функция `MPI_Cart_rank` возвращает ранг процесса по его декартовым координатам.

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank),
```

где

– `comm` – коммуникатор;

– `coords` - декартовы координаты процесса;

– `rank` - возвращаемый функцией ранг процесса.

Процедура `MPI_Card_sub` расщепляет гиперкуб на гиперплоскости.

```
int MPI_Card_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm),
```

где:

– `comm` - исходный коммуникатор;

– `remain_dims` - массив, определяющий, какие измерения будут включены (1) или исключены (0) из гиперкуба;

– `newcomm` - возвращаемый коммуникатор.

Ниже приведены фрагменты кода, иллюстрирующие создание декартовой топологии.

```
…
MPI_Comm grid_comm, row_comm, col_comm;
int dim_sizes[2], wrap_around[2], reorder = 1;
int free_coords[2];
int p;
…
p=15;
dim_sizes[0] = dim_sizes[1] = p;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes,
wrap_around, reoreder, &grid_comm);
free_coords[0] = 0;
free_coords[1] = 1;
```

```
MPI_Cart_sub(grid_comm, free_coords, &row_comm);
free_coords[0] = 1;
free_coords[1] = 0;
MPI_Cart_sub(grid_comm, free_coords, &col_comm);
```

В программе сначала создается коммуникатор `grid_comm`, который представляет собой двумерную сетку (размерности 15x15), причем оба измерения периодические[7]. Затем с помощью процедуры `MPI_Card_sub` производится расщепление на гиперплоскости: сначала по одному измерению, а потом и по другому. В результате получаются еще два коммуникатора - один для строк, а другой для столбцов решетки.

Для процессов, организованных в гиперкуб, могут выполняться обмены особого рода - сдвиги. Имеется два вида сдвигов

- Циклический сдвиг на `k` элементов вдоль ребра решетки. Данные от процессора I пересылаются процессу `(k+i) mod dim`, где `dim` - размерность измерения, вдоль которого производится сдвиг;
- Линейный сдвиг на `k` позиций вдоль ребра решетки. Данные от процессора I пересылаются процессору `I+k` (если такой существует).

Сдвиг обеспечивается функцией:

```
int MPI_Card_shift(MPI_Comm comm, int direction,int displ, int* source, int* dst),
```

где:

- `comm` – коммуникатор;
- `direction` - направление сдвига;
- `disp` - величина сдвига;
- `source` - ранги источников;
- `dst` - ранги получателей.

[7] Тем самым, функция MPI_Cart_create позволяет определять наряду с решетками и топологии типа тор.

7. Примеры параллельных программ

7.1. Параллельная пузырьковая сортировка

7.1.1. Постановка задачи

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S=\{a_1, a_2, \ldots, a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{ (a'_1, a'_2, \ldots., a'_n): a'_1 \le a'_2 \le \ldots. \le a'_n \}.$$

Возможные способы решения этой задачи широко обсуждаются в литературе [1,2]. Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T \sim n^2 .$$

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

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

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из n значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких (p, $p>1$) процессоров. Исходный упорядочиваемый набор разделяется между процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются между собой. Результирующий набор, как правило, также разделен между процессорами; при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагающиеся на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

В рассматриваемом примере описан алгоритм параллельной пузырьковой сортировки.

7.1.2. Описание алгоритма решения задачи

Алгоритм последовательной пузырьковой сортировки

Дадим краткое описание алгоритма.

Последовательный алгоритм "пузырьковой" сортировки сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности

$(a_1, a_2, ..., a_n)$

алгоритм сначала выполняет n-1 операций *"сравнения-обмена" (compare-exchange)* для всех последовательных пар элементов (a_1, a_2), (a_2, a_3), ..., (a_{n-1},a_n). На этом шаге самый большой элемент перемещается в конец последовательности. Последний элемент в преобразованной последовательности затем исключается из сортировки, и последовательность "сравнений-обменов" применяется к возникающей в результате сокращенной последовательности $(a'_1, a'_2, ..., a'_{n-1})$.

Последовательность будет отсортирована после $n-1$ итерации. Мы можем улучшить эффективность "пузырьковой" сортировки, переходя к завершению процесса уже тогда, когда не происходит никаких обменов в течение итерации.

Алгоритм 1. Последовательный алгоритм "пузырьковой" сортировки

```
BUBBLE_SORT(n){
  for (i=n-1; i<1; i--)
    for (j=1; j<i; j++)
      compare_exchange(a_j, a_j+1);
}
```

Итерация внутреннего цикла "пузырьковой" сортировки выполняется за время $O(n)$, всего мы выполняем $O(n)$ итераций; таким образом, сложность "пузырьковой" сортировки - $O(n^2)$.

7.1.3. Алгоритм четной-нечетной перестановки

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как *метод чет-нечетной перестановки (odd-even transposition)* [1] .

Алгоритм "чет-нечетных перестановок" также сортирует n элементов за n итераций (n - четно), однако правила итераций различаются в зависимости от четности или нечетности номера итерации и, кроме того, каждая из итераций требует $n/2$ операций сравнения-обмена.

Пусть $(a_1,a_2, ...,a_n)$ - последовательность, которую нужно отсортировать. На итерациях с нечетными итерациями элементы с нечетными индексами сравниваются с их правыми соседями, и если они не находятся в нужном порядке, они меняются местами (т.е. сравниваются пары $(a_1, a_2), (a_3, a_4), ..., (a_{n-1},a_n)$ и, при необходимости, обмениваются (принимая, что n четно). Точно так же в течение четной фазы элементы с четными индексами сравниваются с их правыми соседями (т.е. сравниваются пары $(a_2, a_3), (a_4, a_5), ..., (a_{n-2},a_{n-1})$), и если они находятся не в порядке сортировки, их меняют местами. После n фазы нечетно-четных перестановок последовательность отсортирована. Каждая фаза алгоритма (и нечетная, и четная) требует $O(n)$ сравнений, а всего n фаз; таким образом, последовательная сложность алгоритма - $O(n^2)$.

Алгоритм2.Последовательный алгоритм "чет-нечетных перестановок"

```
ODD_EVEN(n) {
  for (i=1; i<n; i++){
    if (i%2==1) // нечетная итерация
      for (j=0; j<n/2-1; j++)
        compare_exchange(a2j+1,a2j+2);
    if (i%2==0) // четная итерация
      for (j=1; j<n/2-1; j++)
        compare_exchange(a2j,a2j+1);
  }
}
```

7.1.4. Параллельная реализация алгоритма

Получение параллельного вараинта алгоритма чет-нечетных перестановок не представляет каких-либо затрудений. Для каждой итерации алгоритма операции сравнения-обмена для всех пар элементов являются независимыми и выполняются одновременно. Рассмотрим случай "один элемент на процессор". Пусть $p=n$ - число процессоров (а также число элементов, которые нужно сортировать). Для простоты изложения материала будем предполагать, что вычислительная система имеет топологию кольца. Пусть далее элементы a_i $i = 1, 2, ..., n$, первоначально располагаются на процессорах p_i, $i = 1, 2, ..., n$. В течение нечетной итерации каждый процессор, который имеет нечетный номер, производит сравнение-обмен своего элемента с элементом, находящимся на процессоре-соседе справа. Аналогично, в течение четной итерации каждый процессор с четным номером производит сравнение-обмен своего элемента с элементом правого соседа.

Алгоритм 3. Параллельный алгоритм "чет-нечетных перестановок" на n-процессорном кольце

```
ODD_EVEN_PAR(n){
  id = GetProcId
  for (i=1; i<n; i++){
    if (i%2 == 1) // нечетная итерация
      if (id%2 == 1) // нечетный номер процессора
        compare_exchange_min(id+1); // сравнение-обмен с элементом на процессоре-соседе справа
      else
        compare_exchange_max(id-1); // сравнение-обмен с элементом на процессоре-соседе слева
    if (i%2 == 0) // четная итерация
      if(id%2 == 0) // четный номер процессора
        compare_exchange_min(id+1); // сравнение-обмен с элементом на процессоре-соседе справа
      else
        compare_exchange_max(id-1); // сравнение-обмен с элементом на процессоре-соседе слева
  }
}
```

В течение каждой итерации алгоритма операции "сравнения-обмена" выполняются только между соседними процессорами. Это требует $\Theta(1)$ времени. Общее количество таких итераций - n; таким образом, параллельное время выполнения алгоритма - $\Theta(n)$.

Рассмотрим теперь случай, когда число процессоров меньше, чем длина сортируемой последовательности. Пусть p - число процессоров, n - длина сортируемой последовательности, где p<n. Первоначально каждый процессор получает блок n/p элементов, который может быть упорядочен за время $\Theta((n/p)\log(n/p))$ каким-либо быстрым алгоритмом сортировки. После этого процессоры выполняют p ($p/2$ нечетных и $p/2$ четных) итераций, выполняя операцию "сравнить и разделить" (compare-split). Суть данной операции состоит в следующей последовательности действий:

- соседние по кольцу процессоры передают копии своих данных друг другу; в результате этих действий на каждом из пар соседних процессоров будет располагаться одинаковый набор двух блоков упорядочиваемых значений;
- далее на каждом процессоре имеющиеся блоки объединяются при помощи операции слияния;
- затем на каждом процессоре блок удвоенного размера разделяется на две части; после этого левый сосед процессорной пары оставляет первую половину элементов (с меньшими по значению элементами), а правый процессор пары - вторую (с большими значениями данных).

По окончании p итераций алгоритма исходная последовательность данных оказывается отсортированной.

7.1.5. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного метода. Длительность операций передачи данных между процессорами полностью определяется топологией вычислительной сети. Если логически соседние процессоры, участвующие в выполнении операции "сравнить и разделить", являются близкими (например, для кольцевой топологии сети), общая коммуникационная сложность алгоритма

пропорциональна количеству упорядоченных данных, т. е. n (объем пересылаемых на итерации данных определяется размером блока n/p, количество итераций равно p). Тем самым, вычислительная трудоемкость алгоритма определяется выражением:

$$T_p = \Theta((n/p)\log(n/p)) + \Theta(n) + \Theta(n),$$

где первая часть соотношения учитывает сложность первоначальной сортировки блоков, а вторая величина задает общее количество операций для слияния блоков в ходе исполнения операций "сравнить и разделить" (слияние двух блоков требует *2(n/p)* операций, всего выполняется p итераций сортировки). С учетом данной оценки показатели параллельного алгоритма имеют вид:

- для ускорения[8] - $S_p = \dfrac{\Theta(n\log n)}{\Theta((n/p)\log(n/p)) + \Theta(n)}$,
- для эффективности -

$$E_p = \frac{1}{1 - \Theta(\log p / \log n) + \Theta(p/\log n)}.$$

Анализ выражений показывает, что если количество процессоров совпадает с числом сортируемых данных, эффективность использования процессоров падает с ростом n; получение асимптотически ненулевого значения показателя E_p может быть обеспечено при количестве процессоров, пропорциональном величине *log n*.

7.1.6. Описание программы

Схема вызова функций программы, реализующей метод параллельной пузырьковой сортировки, показана на рисунке 5.

[8] Напомним, что под ускорением понимается отношение времени выполнения последовательной программы к времени выполнения параллельной программы. Под эффективностью понимают отношение ускорения к количеству используемых процессоров

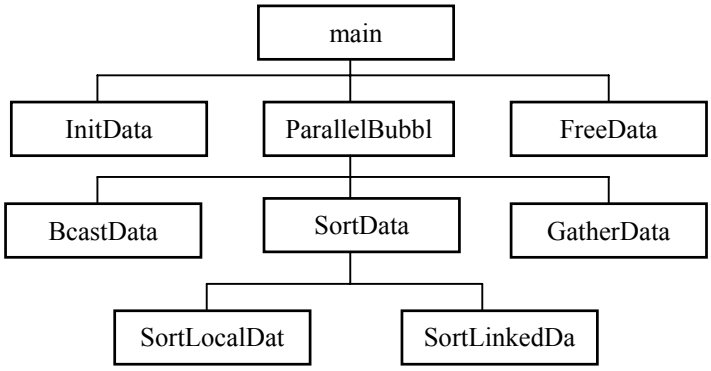

Рис. 5. Схема вызовов функций для метода параллельной пузырьковой сортировки

Рассмотрим процесс работы программы.

Управляющий процессор (процессор 0) выполняет:

- ввод длины сортируемой последовательности и ее генерацию случайным образом (функция *InitData*);
- разбиение последовательности на части и пересылку на соответствующие процессоры блоков последовательности (функция *BcastData*);
- выполнение итераций параллельной сортировки (эта часть действий является общей для всех процессоров и детально описана в алгоритме работы функциональных процессоров);
- сбор отсортированных частей от всех процессоров (функция GatherData);
- освобождение занимаемой памяти (функция *FreeData*).

Функциональные процессоры выполняют:

- прием от процессора 0 соответствующих элементов (функция BcastData);

– внутреннюю сортировку при помощи алгоритма пузырьковой сортировки на каждом процессоре (функция SortLocalData);

– реализацию параллельной чет-нечетной перестановки (функции SortData и SortLinkedData);

– обмен данными между соседними процессорами;

– объединение слиянием упорядоченных блоков;

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

– передача результата (отсортированного блока данных после выполнения всех итераций алгоритма) на управляющий процессор.

7.1.7. Текст программы

Файл bubble.h

```
#ifndef __BUBBLE_H
#define __BUBBLE_H

#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

struct PROCESS {  // данные процесса
  int ProcRank;  // ранг процесса
  int CommSize;  // общее число процессов
  int *pProcData;  // указатель на данные
  int BlockSize;  // количество значений
};

// генерация исходных данных
void InitData ( int **pData, int *pDataSize, PROCESS
*pProcs, int *argc, char **argv[] );
// параллельная сортировка
void ParallelBubble ( int *pData, int DataSize,
PROCESS *pProcs );
// завершение работы
void FreeData ( int *pData, int DataSize, PROCESS
*pProcs );
```

```
   //рассылка и прием исходных данных
   void BcastData ( int *pData, int DataSize, PROCESS
*pProcs );
   // сортировка данных
   void SortLocalData ( PROCESS *pProcs );
   // сбор отсортированных данных
   void GatherData ( int *pData, PROCESS *pProcs ) ;
   // локальная пузырьковая сортировка
   void SortLocalData ( PROCESS *pProcs );
   // выполнение операции "сравнить и разделить"
   void SortLinkedData ( int **pTemp1, int **pTemp2, int
BlockSize, int LinkedRank, int **pMergeData, int
Invert);

   #endif
```

Файл bubble.cpp

```
   #include "Bubble.h"

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

     PROCESS Procs;
     int *pData;
     int DataSize;

     // генерация исходных данных
     InitData ( &pData, &DataSize, &Procs, &argc, &argv
);
     // параллельная сортировка
     ParallelBubble ( pData, DataSize, &Procs );
     // завершение работы
     FreeData ( pData, DataSize, &Procs );

     return 0;
   }

   //генерация исходных данных
   void InitData ( int **pData, int *pDataSize, PROCESS
*pProcs, int *argc, char **argv[] ) {
     MPI_Init ( argc, argv );
```

```
     MPI_Comm_rank ( MPI_COMM_WORLD, &pProcs->ProcRank
);
     MPI_Comm_size ( MPI_COMM_WORLD, &pProcs->CommSize
);

     if ( pProcs->ProcRank == 0 ) {
       sscanf((*argv)[1], "%d", pDataSize);
         if ( *pDataSize%pProcs->CommSize != 0 )
           *pDataSize += pProcs->CommSize - ( *pDataSize
% pProcs->CommSize );
       *pData = new int[*pDataSize];
       srand(100);
       for(int i=0; i<(*pDataSize); i++) (*pData)[i] =
rand();
     }
   }

   // параллельная сортировка
   void ParallelBubble ( int *pData, int DataSize,
PROCESS *pProcs ) {
     //рассылка и прием исходных данных
     BcastData ( pData, DataSize, pProcs );
     // сортировка данных
     SortData ( pProcs );
     // сбор отсортированных данных
     GatherData ( pData, pProcs );
   }

   // завершение работы
   void FreeData ( int *pData, int DataSize, PROCESS
*pProcs ) {
     if ( pProcs->ProcRank == 0) delete [] pData;
     MPI_Finalize();
   }

   // рассылка и прием исходных данных
   void BcastData ( int *pData, int DataSize, PROCESS
*pProcs ) {
     if ( pProcs->ProcRank == 0 ) { // рассылка блоков данных
       pProcs->pProcData = pData;
       pProcs->BlockSize = DataSize / pProcs->CommSize;
       MPI_Bcast ( &pProcs->BlockSize, 1, MPI_INT, 0,
MPI_COMM_WORLD );
```

```
       MPI_Scatter ( &pData[(pProcs->ProcRank)*(pProcs-
>BlockSize)], pProcs->BlockSize, MPI_INT, pProcs-
>pProcData, pProcs->BlockSize, MPI_INT, 0,
MPI_COMM_WORLD );
     }
     else { // прием блоков данных
       MPI_Bcast ( &pProcs->BlockSize, 1, MPI_INT, 0,
MPI_COMM_WORLD );
       pProcs->pProcData = new int[pProcs->BlockSize *
2];
       MPI_Scatter ( &pData[(pProcs->ProcRank)*(pProcs-
>BlockSize)], pProcs->BlockSize, MPI_INT, pProcs-
>pProcData, pProcs->BlockSize, MPI_INT, 0,
MPI_COMM_WORLD );
     }
   }

   // сортировка данных
   void SortData ( PROCESS *pProcs ) {
     // данные с соседнего процессора
     int *pLinkedData = &(pProcs->pProcData[pProcs-
>BlockSize]);
     int *pMergeData = new int[ 2*pProcs->BlockSize ]; //
данные после слияния
     int Offset; // сдвиг номера процесса
     int Invert; // перестановка указателей

     SortLocalData ( pProcs ); // локальная сортировка на
процессоре

     for ( int i=1; i<=pProcs->CommSize; i++ ) { // чет-
нечетная перестановка
       if ( (i%2) == 1) { // нечетная итерация
         if ( (pProcs->ProcRank) % 2 == 1) { // нечетный
процесс
           if ( (pProcs->CommSize)-1 == pProcs->ProcRank
) continue;
          Offset=1; Invert=0;
         }
         else { // четный процесс
           if ( pProcs->ProcRank == 0 ) continue;
           Offset=-1; Invert=1;
        }
```

```
        }
        else { // четная итерация
          if ( (pProcs->ProcRank) % 2 == 1) { // нечетный
процесс
          Offset=-1; Invert=1;
       }
          else { // четный процесс
            if ( (pProcs->ProcRank) == (pProcs-
>CommSize)-1 ) continue;
            Offset=1; Invert=0;
         }
        }
        SortLinkedData ( &(pProcs->pProcData),
&pLinkedData, pProcs->BlockSize,
        pProcs->ProcRank + Offset, & pMergeData , Invert
);
      }
      delete [] pMergeData;
      MPI_Barrier ( MPI_COMM_WORLD );
    }

    // сбор отсортированных данных
    void GatherData ( int *pData, PROCESS *pProcs ) {
      MPI_Gather ( pProcs->pProcData, pProcs->BlockSize,
MPI_INT, pData, pProcs->BlockSize, MPI_INT, 0,
MPI_COMM_WORLD );
    }

    // локальная пузырьковая сортировка
    void SortLocalData  PROCESS *pProcs ) {
      int Tmp;
      for ( int i=0; i<(pProcs->BlockSize)-1; i++ )
        for ( int j=0; j<(pProcs->BlockSize)-1; j++) {
          if ( pProcs->pProcData[j] > pProcs-
>pProcData[j+1] ) {
            Tmp = pProcs->pProcData[j];
            pProcs->pProcData[j] = pProcs-
>pProcData[j+1];
            pProcs->pProcData[j+1] = Tmp;
          }
        }
    }

    // выполнение операции "сравнить и разделить"
```

```
   void SortLinkedData ( int **pTemp1, int **pTemp2, int
BlockSize, int LinkedRank, int **pMergeData, int Invert)
{
     MPI_Status status;
    // обмен данными между соседними процессорами
     MPI_Sendrecv ( *pTemp1, BlockSize, MPI_INT,
LinkedRank, 0, *pTemp2, BlockSize, MPI_INT, LinkedRank,
0, MPI_COMM_WORLD, &status );
     int i=0, j=0;
     while ( (i<BlockSize) && (j<BlockSize) ) { //
сортировка слиянием
       if ( (*pTemp1)[i] < (*pTemp2)[j] ) {
(*pMergeData)[i+j] = (*pTemp1)[i]; i++; }
         else { (*pMergeData)[i+j] = (*pTemp2)[j]; j++;
}
     }
     if ( i<BlockSize )
       for ( int k=i; k<BlockSize; k++ )
(*pMergeData)[k+j] = (*pTemp1)[k];
     if ( j<BlockSize )
       for ( int k=j; k<BlockSize; k++ )
(*pMergeData)[k+i] = (*pTemp2)[k];
     int *pTmp;
     if (Invert) { // перестановка указателей и разделение
последовательности
       pTmp = *pTemp1; *pTemp2 = * pMergeData;
       *pTemp1 = &( (*pMergeData)[BlockSize] );
     }
     else {
       pTmp = *pTemp2; *pTemp1 = * pMergeData;
       *pTemp2 = &( (*pMergeData)[BlockSize] );
     }
     *pMergeData = pTmp;
   }
```

7.1.8. Результаты вычислительных экспериментов

Описанная выше программа параллельной сортировки была реализована с помощью Microsoft Visual C++ 6.0 с использованием библиотеки параллельных вычислений MPICH.NT 1.2.2 фирмы Argonne National Labs. Эксперименты проводились на вычислительном кластере Нижегородского кластера на базе компьютеров Pentium 4 1.3

ГГц 256Mb RAM, сеть 100 Mbit Fast Ethernet, операционная система – MS Windows 2000 Professional.

Результаты экспериментов в числовой форме представлены в таблице 3 и виде графиков зависимостей времени выполнения и ускорения вычислений на рисунке 6 и рисунке 7.

Таблица 3. Результаты экспериментов

Размер массива	Время работы последовательного алгоритма	2 процессора		4 процессора	
		Время выполнения	Ускорение	Время выполнения	Ускорение
1000	0.0178	0.0101	1.76	0.3318	0.05
5000	0.4703	0.2296	2.04	0.4847	0.97
10000	1.7747	0.7295	2.43	0.5930	2.99
20000	7.1495	2.7839	2.57	1.4897	4.80
50000	43.6683	17.7828	2.46	5.5284	7.90
100000	186.2004	71.6852	2.60	18.4546	10.09

Сверхлинейный характер ускорения для ряда экспериментов обусловливается квадратичной сложностью алгоритма сортировки в зависимости от объема упорядочиваемых данных (сортируемые блоки в параллельном варианте алгоритма имеют меньший размер по сравнению с исходным набором значений).

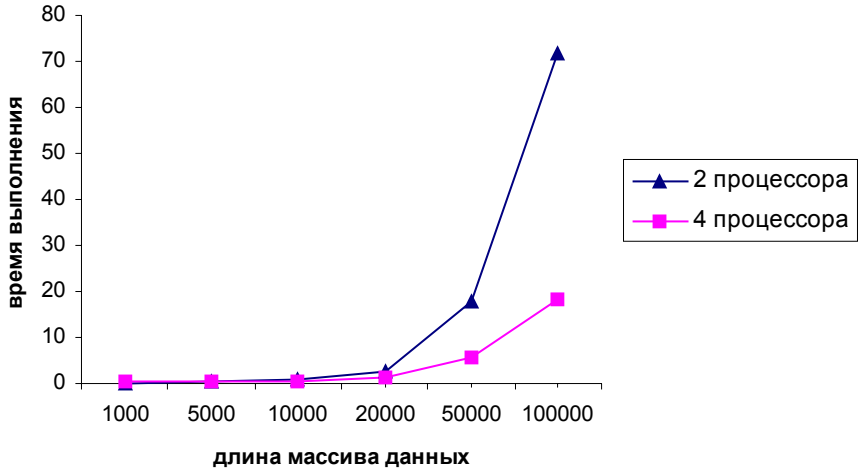

Рис. 6. Зависимость времени выполнения от размера массива

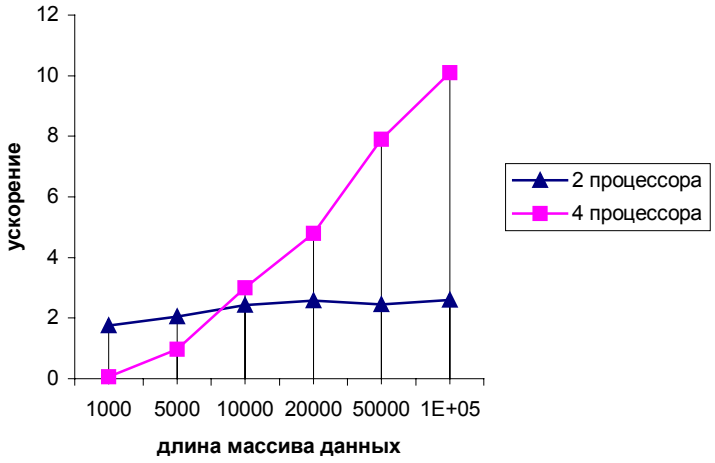

Рис. 7. Зависимость ускорения от размера массива

7.2. Умножение матриц. Параллельный алгоритм Фокса

7.2.1.Постановка задачи

Заданы две матрицы $A = ||a_{ij}||_{n\times m}$ и $B = ||b_{jk}||_{m\times k}$. Получить матрицу $C = ||c_{ik}||_{n\times k}$, которая является результатом перемножения матриц A и B, используя параллельный алгоритм Фокса.

7.2.2. Параллельная реализация алгоритма

При построении параллельных способов выполнения матричного умножения широко используется *блочное представление* матриц. Пусть количество процессоров составляет $p = k^2$, а количество строк и столбцов матрицы является кратным величине $k = \sqrt{p}$, т.е. $n = mk$. Представим исходные матрицы A, B и результирующую матрицу C в виде наборов прямоугольных блоков размера $m \times m$. Тогда операцию матричного умножения матриц A и B в блочном виде можно представить следующим образом:

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

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

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

При анализе данного выражения можно обратить внимание на взаимную независимость вычислений блоков C_{ij} матрицы C. Как результат, возможный подход для параллельного выполнения вычислений может состоять в выделении для расчетов, связанных с получением отдельных блоков C_{ij}, разных процессоров. Применение подобного подхода позволяет получить многие *эффективные*

параллельные методы умножения блочно-представленных матриц; один из алгоритмов данного класса рассматривается ниже.

Для организации параллельных вычислений предположим, что процессоры образуют логическую прямоугольную решетку размером $k \times k$ (обозначим через p_{ij} процессор, располагаемый на пересечении i строки и j столбца решетки). Основные положения параллельного метода, известного как *алгоритм Фокса* (*Fox*) [1], состоят в следующем:

- каждый из процессоров решетки отвечает за вычисление одного блока матрицы C;
- в ходе вычислений на каждом из процессоров p_{ij} располагается четыре матричных блока:

  - блок C_{ij} матрицы C, вычисляемый процессором;

  - блок A_{ij} матрицы A, размещенный в процессоре перед началом вычислений;

  - блоки A'_{ij}, B'_{ij} матриц A и B, получаемые процессором в ходе выполнения вычислений;

Выполнение параллельного метода включает:

- **этап инициализации**, на котором на каждый процессор p_{ij} передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех процессорах;
- **этап вычислений**, на каждой итерации l, $1 \le l \le k$, которого выполняется:

  - для каждой строки i, $1 \le i \le k$, процессорной решетки блок A_{ij} процессора p_{ij} пересылается на все процессоры той же строки i; индекс j, определяющий положение процессора p_{ij} в строке, вычисляется по соотношению

$$j = (i + l - 1) \bmod k + 1,$$

($\bmod$ есть операция получения остатка от целого деления);

- полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждого процессора p_{ij} перемножаются и прибавляются к блоку C_{ij}

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij};$$

- блоки B'_{ij} каждого процессора p_{ij} пересылаются процессорам p_{ij}, являющимися соседями сверху в столбцах процессорной решетки (блоки процессоров из первой строки решетки пересылаются процессорам последней строки решетки).

-

7.2.3. Время выполнения алгоритма

Рассмотрим выполнение данного алгоритма при топологии сети в виде гиперкуба. После k ($k = \sqrt{p}$) шагов процессор $P_{i,j}$ совершил перемножение $A_{i,l} \times B_{l,j}$ для каждого l от 0 до $k-1$. Каждый процессор совершает вычисления за время, равное n^3 / p. Время для рассылки "один-ко-всем"блоков матрицы A преобладает над временем одношагового сдвига блока матрицы B. Если логическая решетка процессоров отображается на гиперкуб, то каждая рассылка может быть совершена за время

$$(t_н + t_к n^2 / p)\log\sqrt{p},$$

где $t_н$ есть время начальной подготовки и характеризует длительность подготовки сообщения для передачи, $t_к$ есть время передачи одного слова данных по линии передачи данных (длительность подобной передачи определяется полосой пропускания коммуникационных каналов в сети). Так как эта операция повторяется k раз, общее время коммуникаций составляет

$$(t_н + t_к n^2 / p)\sqrt{p}\log\sqrt{p}.$$

Тем самым, общее время параллельного выполнения программы составляет

$$T_p = \frac{n^3}{p} + (t_н + t_к n^2 / p)\sqrt{p}\log\sqrt{p}\,.$$

7.2.4. Описание программы

Схема вызова функций программы, реализующей метод Фокса, показана на рисунке 8.

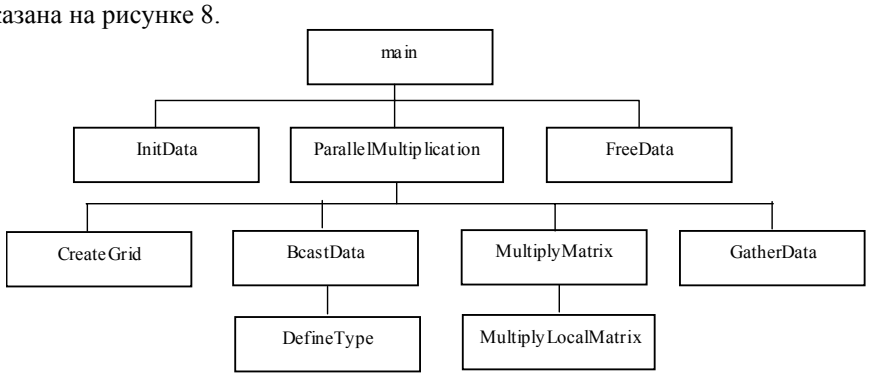

Рис. 8. Схема вызовов функций

Рассмотрим процесс работы программы.

Управляющий процессор (процессор 0) выполняет:

- ввод размерности перемножаемых матриц и ввод элементов матриц (функция *InitData*);
- создание топологии ″решетка″ (функция CreateGrid);
- создание производного типа - ″блок матрицы″ (функция DefineType) и пересылку блоков на соответствующие процессоры в топологии ″решетка″ (функция BcastData);
- выполнение итераций параллельного перемножения матриц (эта часть действий является общей для всех процессоров и детально описана в разделе Параллельная реализация алгоритма);

– сбор результатов (перемноженных блоков) от всех процессоров (функция GatherData);

– освобождение занимаемой памяти (функция *FreeData*).

Функциональные процессоры выполняют:

– создание топологии "решетка" (функция CreateGrid);

– создание производного типа "блок матрицы" (функция DefineType);

– прием от процессора 0 соответствующих блоков матриц (функция BcastData);

– выполнение итераций параллельного перемножения матриц (функция MultiplyMatrix);

– передача результата (перемноженного блока) на управляющий процессор.

–

7.2.5. Текст программы

Файл Fox.h

```
#ifndef __FOX_H
#define __FOX_H

#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

struct PROCESS {      // данные процесса
  int ProcRank;       // ранг процесса
  int CommSize;       // общее количество процессов
  int *pProcMatrixA;  // подматрица A
  int *pProcMatrixB;  // подматрица B
  int *pProcMatrixC;  // подматрица C
  int BlockSize;      // длина подматрицы
  };

struct GRIDINFO {    // параметры решетки
  int DimGrid ;      // порядок решетки
  int NumRow;        // номер строки
  int NumCol;        // номер столбца
  int RankGrid;      // ранг процесса в коммуникаторе решетки
```

```
    MPI_Comm CommGrid; // коммуникатор решетки
    MPI_Comm RowComm; // коммуникатор строки
    MPI_Comm ColComm; // коммуникатор столбца
  };

  // генерация исходных данных
  void InitData ( int **pMatrixA, int **pMatrixB, int
*pDataSize, PROCESS *pProcs, int *argc, char **argv[] );
  // параллельное умножение матриц
  void ParallelMultiplication ( int *pMatrixA, int
*pMatrixB, int **pMatrixC, int DataSize, PROCESS
*pProcs, GRIDINFO *Grid );
  // завершение работы
  void FreeData ( int *pMatrixA, int *pMatrixB, int
DataSize, PROCESS *pProcs );
  // установка параметров решетки
  void CreateGrid ( PROCESS *pProcs, GRIDINFO *Grid );
  // рассылка и прием блоков данных
  void BcastData ( int *pMatrixA, int *pMatrixB, int
DataSize, PROCESS *pProcs, MPI_Datatype *MatrixBlock );
  // перемножение матриц
  void MultiplyMatrix ( int DataSize, PROCESS *pProcs,
GRIDINFO *Grid, MPI_Datatype MatrixBlock);
  // сбор данных
  void GatherData ( PROCESS *pProcs, GRIDINFO *Grid,
int *pMatrixC, int DataSize, MPI_Datatype MatrixBlock );
  // определить тип данных - блок матрицы
  void DefineType ( PROCESS *pProcs, int DataSize,
MPI_Datatype *MatrixBlock );
  // локальное умножение матриц
  void MultiplyLocalMatrix ( int *pMatrixA, int
*pMatrixB, int *pMatrixC, int Size, int Stride );

  #endif
```

Файл Fox.cpp

```
  #include "Fox.h"

  int main(int argc, char *argv[]) {
    PROCESS Procs;
    GRIDINFO Grid;
    int *pMatrixA, *pMatrixB, *pMatrixC;
    int DataSize;
```

```
      // генерация исходных данных
      InitData ( &pMatrixA, &pMatrixB, &DataSize,
&Procs, &argc, &argv );
      // параллельное умножение матриц
      ParallelMultiplication ( pMatrixA, pMatrixB,
&pMatrixC, DataSize, &Procs, &Grid);
      // завершение работы
      FreeData ( pMatrixA, pMatrixB, DataSize, &Procs );

      return 0;
   }

   // генерация исходных данных
   void InitData ( int **pMatrixA, int **pMatrixB, int
*pDataSize, PROCESS *pProcs, int *argc, char **argv[] )
{
     MPI_Init ( argc, argv );
     MPI_Comm_rank ( MPI_COMM_WORLD, &pProcs->ProcRank
);
     MPI_Comm_size ( MPI_COMM_WORLD, &pProcs->CommSize
);

     if ( pProcs->ProcRank == 0 ) {
       sscanf((*argv)[1], "%d", pDataSize);
       if ( *pDataSize % (int)sqrt(pProcs->CommSize) !=
0 )
         *pDataSize += (int)sqrt(pProcs->CommSize) - (
*pDataSize % (int)sqrt(pProcs->CommSize) );
       *pMatrixA = new int[(*pDataSize)*(*pDataSize)];
       *pMatrixB = new int[(*pDataSize)*(*pDataSize)];

       srand(100);
       for( int i = 0; i < (*pDataSize)*(*pDataSize);
i++ ) {
         (*pMatrixA)[i] = (int) rand()/1000;
         (*pMatrixB)[i] = (int) rand()/1000;
       }
     }
   }

   // параллельное умножение матриц
   void ParallelMultiplication ( int *pMatrixA, int
*pMatrixB, int **pMatrixC, int DataSize, PROCESS
*pProcs, GRIDINFO *Grid ) {
```

```
      // установить параметры решетки
      CreateGrid ( pProcs, Grid );
      MPI_Datatype MatrixBlock;
      // разослать данные
      BcastData ( pMatrixA, pMatrixB, DataSize, pProcs,
&MatrixBlock );
      // перемножить матрицы
      MultiplyMatrix ( DataSize,pProcs, Grid, MatrixBlock
);
      // собрать данные
      GatherData ( pProcs, Grid, *pMatrixC, DataSize,
MatrixBlock );
   }

   // завершение работы
   void FreeData ( int *pMatrixA, int *pMatrixB, int
DataSize, PROCESS *pProcs ) {
     if ( pProcs->ProcRank == 0 ) {  delete [] pMatrixA,
pMatrixB; }
     MPI_Finalize();
   }

   // установка параметров решетки
   void CreateGrid ( PROCESS *pProcs, GRIDINFO *Grid ) {
     int Dimensions [2];
     int WrapAround [2];
     int Coordinates[2];
     int FreeCoords [2];
     Grid->DimGrid = (int)sqrt(pProcs->CommSize); //
порядок решетки
     Dimensions [0] = Dimensions [1] = Grid->DimGrid;
     WrapAround [0] = WrapAround [1] = 1;
     // создать решетку
     MPI_Cart_create ( MPI_COMM_WORLD, 2, Dimensions,
WrapAround , 1, &(Grid->CommGrid));
     MPI_Comm_rank ( Grid->CommGrid, &(Grid->RankGrid)
);
     MPI_Cart_coords ( Grid->CommGrid, Grid->RankGrid,
2, Coordinates );
     Grid->NumRow = Coordinates[0]; Grid->NumCol =
Coordinates[1];
     FreeCoords[0] = 0;             FreeCoords[1] = 1;
     // коммуникатор для строк
```

```
     MPI_Cart_sub ( Grid->CommGrid, FreeCoords, &(Grid-
>RowComm) );
     FreeCoords[0] = 1;              FreeCoords[1] = 0;
     // коммуникатор для столбцов
     MPI_Cart_sub ( Grid->CommGrid, FreeCoords, &(Grid-
>ColComm) );
   }

   // рассылка и прием блоков данных
   void BcastData ( int *pMatrixA, int *pMatrixB, int
DataSize, PROCESS *pProcs, MPI_Datatype *MatrixBlock ) {
     if ( pProcs->ProcRank == 0 ) { // разослать размерность
подматриц и сами подматрицы
       pProcs->BlockSize = DataSize / (int)sqrt(pProcs-
>CommSize);
       MPI_Bcast ( &pProcs->BlockSize, 1, MPI_INT, 0,
MPI_COMM_WORLD );

       // определить тип - блок матрицы
       DefineType ( pProcs, (pProcs->BlockSize) *
(int)sqrt(pProcs->CommSize), MatrixBlock );

       pProcs->pProcMatrixA = pMatrixA;
       pProcs->pProcMatrixB = pMatrixB;
       for ( int NumProc = 1; NumProc < pProcs-
>CommSize; NumProc++ ) {
         MPI_Send ( &pMatrixA[( NumProc /
(int)sqrt(pProcs->CommSize) ) * DataSize * pProcs-
>BlockSize + pProcs->BlockSize * ( NumProc % (int)
sqrt(pProcs->CommSize) )], 1, *MatrixBlock, NumProc, 1,
MPI_COMM_WORLD);
         MPI_Send ( &pMatrixB[( NumProc /
(int)sqrt(pProcs->CommSize) ) * DataSize * pProcs-
>BlockSize + pProcs->BlockSize * ( NumProc % (int)
sqrt(pProcs->CommSize) )], 1, *MatrixBlock, NumProc, 1,
MPI_COMM_WORLD);
       }
     }
     else { // принять порядок подматриц и сами подматрицы
       MPI_Status status;
       MPI_Bcast ( &pProcs->BlockSize, 1, MPI_INT, 0,
MPI_COMM_WORLD );
       // определить тип - блок матрицы
```

```
       DefineType(pProcs, (pProcs->BlockSize) *
(int)sqrt(pProcs->CommSize),MatrixBlock);
       pProcs->pProcMatrixA = new int[ (pProcs-
>BlockSize) * (pProcs->BlockSize) * (int)sqrt(pProcs-
>CommSize) ];
       pProcs->pProcMatrixB = new int[ (pProcs-
>BlockSize) * (pProcs->BlockSize) * (int)sqrt(pProcs-
>CommSize) ];
       MPI_Recv ( pProcs->pProcMatrixA, 1, *MatrixBlock,
0, 1, MPI_COMM_WORLD, &status );
       MPI_Recv ( pProcs->pProcMatrixB, 1, *MatrixBlock,
0, 1, MPI_COMM_WORLD, &status );
     }
   }

   // перемножение матриц
   void MultiplyMatrix ( PROCESS *pProcs, GRIDINFO
*Grid, MPI_Datatype MatrixBlock) {
     MPI_Status status;
     int *TempA;
     // номер рассылающего процесса
     int Source = ( Grid->NumRow + 1 ) % Grid->DimGrid;
     // номер принимающего процесса
     int Dest = ( Grid->NumRow + Grid->DimGrid - 1) %
(Grid->DimGrid);
     if (pProcs->ProcRank == 0)
       pProcs->pProcMatrixC = new int [ (pProcs-
>BlockSize) * (int)sqrt(pProcs->CommSize) * (pProcs-
>BlockSize) * (int)sqrt(pProcs->CommSize) ];
     else
       pProcs->pProcMatrixC = new int [ (pProcs-
>BlockSize) * (pProcs->BlockSize) * (int)sqrt(pProcs-
>CommSize) ];

     for( int j = 0; j < pProcs->BlockSize * pProcs-
>BlockSize * (int)sqrt(pProcs->CommSize); j++)
       (pProcs->pProcMatrixC)[j] = 0;

     for ( int l = 0; l < Grid->DimGrid; l++ ) {
       // разослать соответствующий блок матрицы A внутри i-ой строки
       for (int i = 0; i < Grid->DimGrid; i++) {
         j = ( Grid->NumRow + l ) % (Grid->DimGrid);
         if ( j == Grid->NumCol ) {
           TempA = pProcs->pProcMatrixA;
```

```
          MPI_Bcast ( pProcs->pProcMatrixA, 1,
MatrixBlock, j, Grid->RowComm );
         }
         else {
          TempA = new int [ (pProcs->BlockSize) *
(pProcs->BlockSize) * (int)sqrt(pProcs->CommSize) ];
          MPI_Bcast ( TempA, 1, MatrixBlock, j, Grid-
>RowComm );
        }
       }
       // перемножить блоки матриц A и B
       MultiplyLocalMatrix ( TempA, pProcs-
>pProcMatrixB, pProcs->pProcMatrixC, pProcs->BlockSize,
pProcs->BlockSize*(int)sqrt(pProcs->CommSize)-pProcs-
>BlockSize );
       // циклически передвинуть блоки матрицы B внутри каждого
столбца
       MPI_Sendrecv_replace ( pProcs->pProcMatrixB, 1,
MatrixBlock, Dest,0, Source,0, Grid->ColComm, &status );
     }
     delete [] TempA;
   }

   // сбор данных
   void GatherData ( PROCESS *pProcs, GRIDINFO *Grid,
int *pMatrixC, int DataSize, MPI_Datatype MatrixBlock )
{
     if ( pProcs->ProcRank == 0 ) {
       MPI_Status status;
       pMatrixC = pProcs->pProcMatrixC;
       for ( int NumProc = 1; NumProc < pProcs-
>CommSize; NumProc++ )
         MPI_Recv ( &pMatrixC[( NumProc /
(int)sqrt(pProcs->CommSize) ) * DataSize * pProcs-
>BlockSize + pProcs->BlockSize * ( NumProc % (int)
sqrt(pProcs->CommSize) )], 1, MatrixBlock, NumProc, 1,
MPI_COMM_WORLD, &status );
     }
     else
       MPI_Send ( pProcs->pProcMatrixC, 1, MatrixBlock,
0, 1, MPI_COMM_WORLD );
   }

   // определить тип данных - блок матрицы
```

```
   void DefineType ( PROCESS *pProcs, int DataSize,
MPI_Datatype *MatrixBlock ) {
     MPI_Type_vector ( pProcs->BlockSize, pProcs-
>BlockSize, DataSize, MPI_INT, MatrixBlock );
     MPI_Type_commit ( MatrixBlock );
   }

   // локальное умножение матриц
   void MultiplyLocalMatrix ( int *pMatrixA, int
*pMatrixB, int *pMatrixC, int Size, int Stride ) {
     for ( int i = 0; i < Size; i++ )
       for ( int j = 0; j < Size; j++ )
         for ( int k = 0; k < Size; k++ )
           pMatrixC[i*Size+j+i*Stride] +=
pMatrixA[i*Size + k + i*Stride] * pMatrixB[k*Size + j +
k*Stride];
   }
```

7.2.6. Результаты вычислительных экспериментов

Описанная выше программа параллельного умножения матриц была реализована с помощью Microsoft Visual C++ 6.0 с использованием библиотеки параллельных вычислений MPICH.NT 1.2.2 фирмы Argonne National Labs. Эксперименты проводились на вычислительном кластере ННГУ на базе компьютеров Pentium4 1.3ГГц 256Mb RAM, сеть 100 Mbit Fast Ethernet, операционная система – MS Windows 2000 Professional.

Результаты экспериментов в численной форме представлены в таблице 4 и в виде графиков зависимостей времени выполнения и ускорения вычислений на рисунке 9 и рисунке 10.

Таблица 4. Результаты экспериментов для алгоритма Фокса, сек

Порядок матрицы	Время работы послед. алгоритма	4 процессора		9 процессоров		16 процессоров	
		Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
500	6,87	2,99	2,30	3,53	1,95	5,26	1,31
000	58,63	18,67	3,14	27,72	2,12	28,14	2,08
1300	133,78	38,76	3,45	60,00	2,23	75,23	1,78
1500	368,93	73,39	5,03	85,44	4,32	83,55	4,42

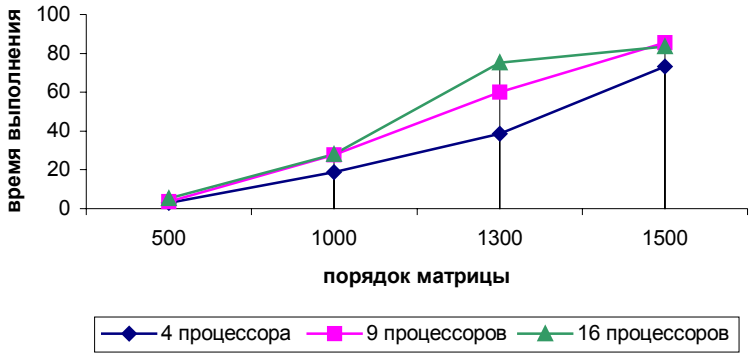

Рисунок 9. Зависимость времени выполнения алгоритма от порядка матрицы для алгоритма Фокса

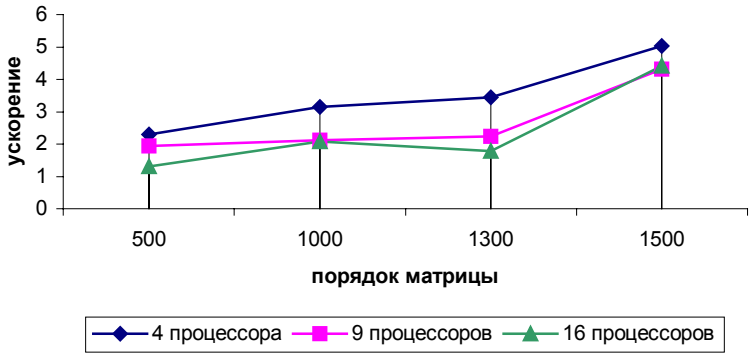

Рисунок 10. Зависимость ускорения от порядка матрицы для алгоритма Фокса

8. Отладка и профилировка программ, разработанных с использованием библиотеки MPI

8.1. Среда выполнения параллельных программ

Одной из основных задач, которые ставились при разработке MPI, была возможность лёгкого создания различных библиотек параллельных подпрограмм, помогающих решить проблему построения больших параллельных приложений. Среда MPE (*Multi Processing Environment*) использует эту возможность MPI и реализует такой полезный механизм, как изучение производительности параллельных программ. MPE даёт пользователю общий интерфейс для разработки библиотеки, позволяющей параллельной программе создавать трассировочные файлы (*log-файлы*), содержащие информацию о характеристиках выполнения, таких, например, как размер посылаемых сообщений, время работы функций MPI и др. [3].

Главными компонентами MPE являются следующие:

1. Набор процедур для создания лог-файлов в различных форматах, содержащих информацию о выполнении параллельного приложения.
2. Набор графических приложений для визуализации содержимого лог-файлов, сгенерированных после завершения параллельного приложения.
3. Библиотека визуализации работы параллельного приложения в момент его исполнения, основанная на X-Windows протоколе операционной системы Unix.

Рассмотрим более подробно эти компоненты.

8.2. Создание лог-файлов

Главная идея, используемая в MPE, – замена стандартных функций интерфейса MPI на специальные функции, носящие те же имена, что и функции MPI, но выполняющие дополнительные к стандартным действия. Будем далее называть такие "подменённые" функции *обёртками* (*wrappers*). Таким образом, обёртка, к примеру, над функцией

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

может не только отправлять сообщение другому MPI-процессу, но ещё и записывать некие интересующие пользователя данные (например, размер сообщения - MPI_Type_size(datatype) * count) в память для последующего сбора их на том узле, откуда был произведён запуск параллельного приложения, и записи в лог-файл в определённом формате для дальнейшего просмотра одним из средств визуализации.

Для создания обёрток MPE предоставляет специальный формат описания обёрток и генератор, позволяющий по описаниям создать обёртки в виде функций на языке C.

Для того, чтобы пользователю не нужно было "с нуля" разрабатывать механизмы сбора данных о выполнении параллельной программы с каждого из вычислительных узлов и от каждого из MPI-процессов, MPE предоставляет готовый набор функций (*API – CLOG*), позволяющий:

1. Хранить в памяти информацию, сгенерированную обёртками во время их вызова, как связный лист "состояний программы во времени" с описанием, где каждое "состояние" может отвечать исполнению одной из MPI-функций, или фрагмента кода программы пользователя (последнее потребует непосредственной вставки вызовов CLOG-функций в программу пользователя, использования обёрток будет недостаточно).

2. Автоматически "собирать" данную информацию на "головном" узле после завершения MPI-программы (точнее, после вызова обёртки над функцией MPI_Finalize).

3. Записывать собранную информацию в файл в формате CLOG для дальнейшего анализа данной информации с помощью визуализатора, поддерживающего данный формат.

На API CLOG основан коммерческий пакет для анализа производительности и визуализации параллельных приложений VAMPIR .

В стандартную поставку MPI входит библиотека mpe.lib, содержащая обёртки над функциями MPI, позволяющие лишь узнать время начала и конца работы каждой из MPI-функций.

В рамках работы над вычислительным кластером ННГУ была создана альтернативная библиотека `mpe.lib`, содержащая обёртки, позволяющие протоколировать аргументы функций MPI, а также размеры сообщений, которыми обмениваются MPI-процессы.

8.3. Визуализация содержимого лог-файлов

MPE предоставляет возможность просмотра созданных с помощью API CLOG лог-файлов. К сожалению, программа jumpshot-2, позволяющая просматривать лог-файлы в формате CLOG, поставляется только в Unix-версии MPICH и в данный момент работы над ней остановлены в силу появления более нового и перспективного формата представления информации, собранной во время работы параллельных приложений – SLOG. Появление данного формата вызвано назревшей серьёзной проблемой, связанной с форматом CLOG – крайним замедлением просмотра лог-файлов при росте их величины. Разработчики MPE утверждают, что падение производительности визуализатора начинается уже с лог-файлов размером около 10 мегабайт, и при некотором размере лог-файла визуализатор jumpshot-2 "зависает». Надо отметить, что лог-файлы в формате CLOG достаточно больших и долго исполняющихся MPI-программ часто занимают десятки мегабайтов, что делает невозможным их анализ. Причина сильного падения производительности визуализатора jumpshot-2 в том, что он загружает в память весь CLOG файл целиком, что отнимает огромное количество памяти при анализе файла. Формат SLOG был создан несколько в стороне от MPI и MPE как формат, позволяющий представлять данные в виде фреймов, что даёт возможность новому визуализатору, поддерживающему просмотр файлов в формате SLOG, - jumpshot-3 – визуализировать файлы размеров в несколько гигабайтов (http://www.mcs.anl.gov/perfvis/software/log_format/index.htm#SLOG-1). Формат SLOG находится в развитии и при его создании не было одновременно создано API, подобного CLOG. Новое API, поставляемое с SLOG-форматом, не позволяет напрямую использовать SLOG при написании обёрток для MPI-функций, что вынуждает использовать API CLOG во время работы параллельной MPI-программы, а затем, после завершения программы, конвертировать данные из формата CLOG в SLOG. При этом, однако, в состав MPE не входит полноценный конвертер из формата CLOG в формат SLOG,

позволяющий конвертировать в SLOG-файл информацию о размере сообщений, которыми обмениваются MPI-процессы, а также, к примеру, информацию об аргументах вызывавшихся MPI-функций. С другой стороны, API SLOG содержит все необходимые функции для создания полноценного конвертера из CLOG в SLOG, и требуемый конвертер был создан в рамках работы над вычислительным кластером ННГУ.

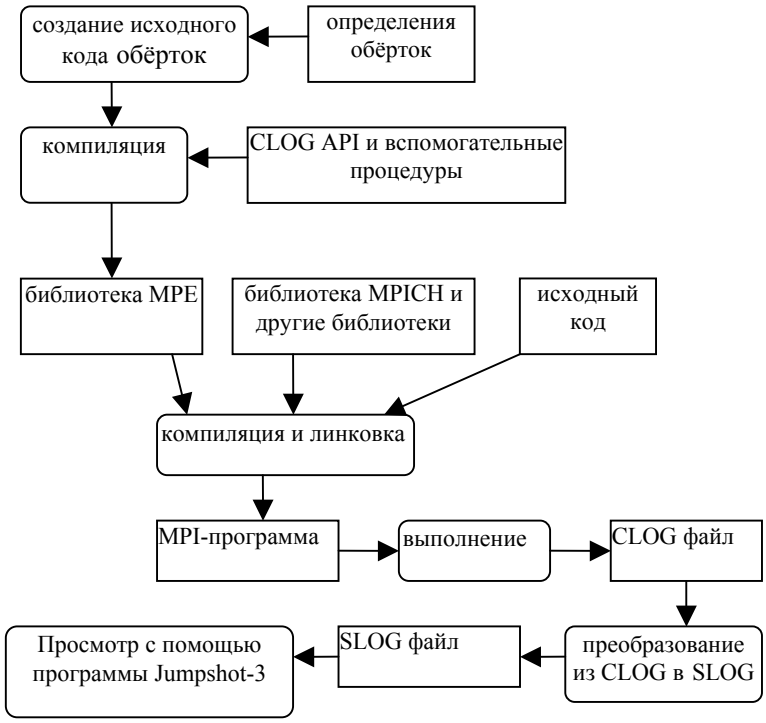

Рисунок 11. Общее представление о сборе, преобразовании и отображении информации о выполнении параллельной программы

Рассмотрим подробнее составные элементы процесса сбора, преобразования и отображения информации о выполнении

параллельной программы с помощью средств MPE, представленные на рисунке 11:

- **Определения обёрток**

"Прототипы" обёрток, записанные в специальном формате (не являющимся полноценным языком), позволяющие одним определением описать целый класс обёрток, служат для упрощения написания обёрток.

- **Создание исходного кода обёрток**

Процесс преобразования определений обёрток в "готовые" обёртки на языке C при помощи специальных генераторов исходного кода.

- **CLOG API и вспомогательные процедуры**

Процедуры API CLOG для сохранения в памяти на каждом вычислительном узле, сбора на "головном" узле, записи в файл в формате CLOG информации о выполнении параллельной программы (см. подробнее п. "Создание лог-файлов").

- **Библиотека MPE**

Библиотека `mpe.lib`, содержащая откомпилированные обёртки над функциями MPI; эта библиотека должна компоноваться с программой библиотекой MPI для того, чтобы все переопределения функций MPI, являющиеся обёртками, брались именно из данной библиотеки.

- **Библиотека MPICH -**

Библиотека `mpich.lib`, содержащая оригинальные функции интерфейса MPI.

- **Исходный код**

Исходный код программы пользователя, использующей MPI.

- **MPI-программа**

Программа, полученная после компиляции исходного кода программы пользователя и линковки с библиотеками MPE, MPICH и необходимыми системными библиотеками.

- **Выполнение**

Выполнение программы пользователя приводит к сбору и записи информации в файл в формате CLOG.

- **CLOG файл**

Файл, содержащий информацию о выполнении параллельной программы (см. подробнее п. "Создание лог-файлов").

- **Преобразование из CLOG в SLOG**

С помощью специального преобразователя (*converter*) CLOG файл преобразуется в SLOG файл.

- **SLOG файл**

Файл, содержащий информацию о выполнении параллельной программы в формате SLOG.

- **Просмотр с помощью программы Jumpshot-3**

Просмотр информации о выполнении параллельной программы, сохранённой в формате SLOG, с помощью просмотрщика (*viewer*) Jumpshot-3.

8.4. Методика организации профилирования функций MPI

В MPI включены спецификации того, каким образом должен быть организован перехват вызовов библиотеки MPI для выполнения необходимых дополнительных действий (например, для целей профилирования) без знания при этом исходных кодов реализации MPI. Идея заключается в том, чтобы выполнить перехват вызовов на этапе линковки, а не на этапе компиляции. Стандарт MPI требует, чтобы каждая процедура вида MPI_Xxx могла быть вызвана по альтернативному имени PMPI_Xxx.

Эта схема позволяет пользователю написать ограниченное число обёрток для процедур MPI и выполнить все необходимые действия в этих обёртках. Чтобы вызвать "настоящую" процедуру MPI, к ней следует обращаться с префиксом "PMPI_". Нужна гарантия того, что линкер выберет обёртку, а не оригинальную функцию MPI при разрешении ссылок на MPI-функцию из исходного кода MPI-программы. Это обеспечивается порядком библиотек, подаваемых на вход линкеру: вначале должна следовать библиотека `mpe.lib`, содержащая обёртки, потом – `mpich.lib`, содержащая оригинальные функции.

8.5. Программа просмотра SLOG файлов Jumpshot-3

Jumpshot-3 позволяет просматривать SLOG файлы в графическом виде. Документация по Jumpshot-3, написанная его авторами, с подробным описанием использования располагается по адресу http://www-unix.mcs.anl.gov/perfvis/software/viewers/jumpshot-3/, а также в комплекте поставки библиотеки MPICH в документе

<MPICH installation directory>\Jumpshot\jumpshot3_tour.pdf

8.6. Пример использования

Рассмотрим пример: анализ работы программы transf1 из тестового пакета НИВЦ МГУ (http://parallel.ru/testmpi/).

8.6.1. Замечания по компиляции программы

Добавьте в строку линковки "mpe.lib mpich.lib" с сохранением порядка следования библиотек.

8.6.2.Выполнение программы

Пусть программа расположена по пути path\transf1, а MPI инсталлирован по пути path2MPI. Запустите программу на двух узлах без дополнительных аргументов. После завершения программы на головном узле будет создан лог-файл в CLOG-формате path\transf1.exe.clog.

8.6.3. Преобразование из CLOG в SLOG

Используйте конвертер из CLOG в SLOG: path2MPI\mpe\clog2slog\clog2slog path\transf1.exe.clog. Будет создан файл transf1.exe.slog.

8.6.4. Просмотр результатов

Запустите jumpshot-3 следующей командой (для этого нужно проинсталлировать Java SDK версии не ниже, чем 1.1.7):

```
java -jar path2MPI\Jumpshot\jumpshot3.jar
```

Загрузите SLOG-файл transf1.exe.slog, после чтения данных которого на экране должно быть представлено окно выбора фреймов.

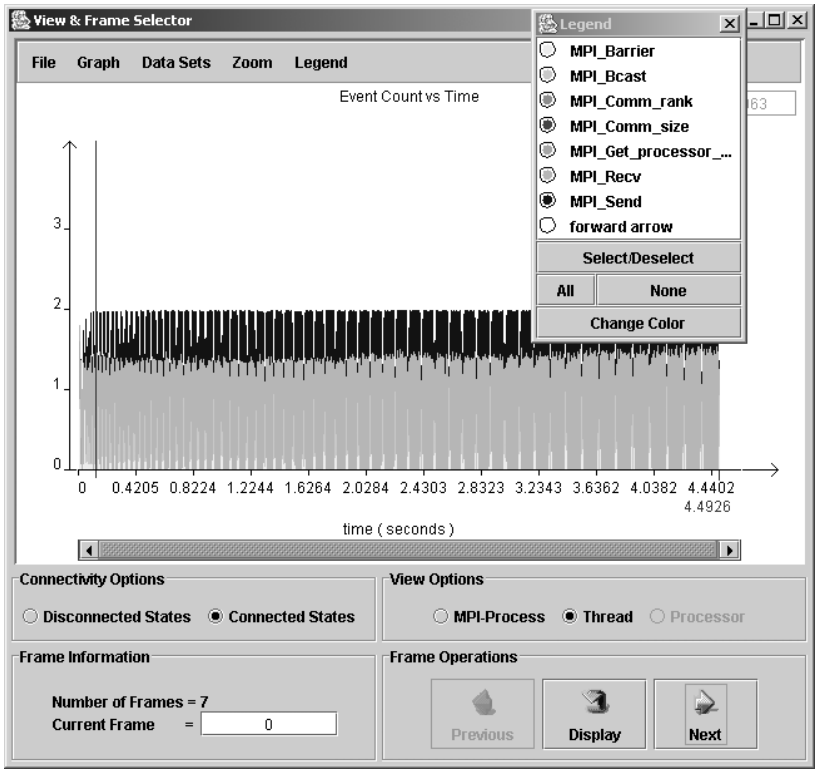

Рисунок 12. Окно выбора фреймов

Выберите временное окно (кнопками "Next" и "Previous") и нажмите кнопку "Display". В результате выбора данного режима на экране будет представлено окно временных диаграмм (см. рис. 12). В нём каждому параллельному процессу (в нашем эксперименте их было два) соответствует ось времени, на которой отложены состояния, в которых находился данный процесс.

В течение своей работы, каждый процесс параллельной программы проходит через ряд "состояний», которые характеризуются исполняемыми в данный момент процедурами MPI. Выделение состояний осуществляется при помощи цветовой маркировки (в нижней части окна временных диаграмм имеются пояснения, какие цвета соответствуют каким состояниям процесса), названия состояний

совпадают с названиями функций MPI. Набором отображенных на экране состояний можно управлять, включая или скрывая высветку тех или иных состояний.

Визульное представление состояний и их чередование по шкале времени позволяет изучать динамику развития процессов параллельной программы. Взаимодействие процессов при передаче сообщений отображается стрелками, направленными от процесса-источника к процессу-приемнику данных.

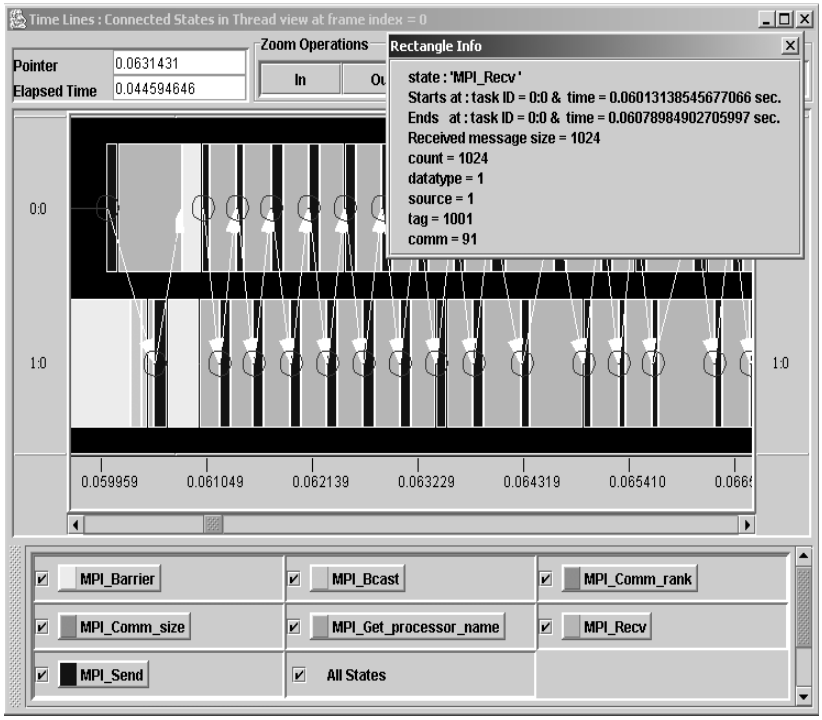

Рисунок 13. Окно временных диаграмм

При щелчке мышью на прямоугольнике, отвечающем конкретному состоянию, в диалоговом окне отображается информация о данном состоянии, включая аргументы соответствующей MPI-функции и размер посланного/полученного сообщения, а также время начала/конца состояния.

Пользуясь данным инструментом, можно проследить ход выполнения программы, найти "узкие"места и получить весьма наглядное представление о параллельной топологии, на которой исполняется программа.

9. Приложения

9.1. Подключение библиотеки MPI к среде программирования

9.1.1. Последовательность установки и подключения к среде разработки
Borland C++Builder 5.0 библиотеки MPI

Соглашения и замечания:

1. В документе рассматривается реализация библиотеки выполненная **Argonne National Laboratory Group** для ОС **Windows NT/2000** (версия библиотеки 1.2.0.4).

2. Здесь и далее термины **MPI** и **MPICH** используются как синонимы.

3. Здесь и далее используется сокращенное название среды разработки BCBuilder.

4. В документе используются названия и термины английской версии ОС **Windows NT/2000**.

5. Здесь и далее полагается, что <BCB> – путь к папке, в которую установлена среда разработки **BCBuilder**; <MPI_PATH> – путь к папке, в которую установлена библиотека MPI.

6. Изначально данная версия библиотеки **MPI** не обладает средствами для компиляции **MPI**-приложений в **BCBuilder**. Для этой цели необходимо использовать специально подготовленный шаблон проекта **MpiProject**.

Действия:

1. Создать в папке <BCB>\Include каталог **Mpi** и скопировать в него содержимое каталога **<MPI_PATH>\Sdk\Include** (4 h.-файла).

2. Создать в папке <BCB>\ObjRepos каталог **MpiApp** и скопировать в него шаблон проекта **MpiProject** (4 файла).

3. В среде **BCBuilder** добавить в репозитарий шаблон проекта **MpiProject**:

- запустить **BCBuilder** (Start/Programs/Borland C++Builder 5/C++Builder 5).
- открыть в среде проект **MpiProject**: (в окне **Open Project** – File/Open Project найти папку <BCB>\ObjRepos\MPIApp\Release, выбрать файл **MpiProject.bpr**).

- в диалоговом окне **Add to Repository** зарегистрировать проект:
  - в поле **Title** ввести название, например, **MPI Application**.
  - в поле **Description** ввести описание проекта;
  - в выпадающем списке **Page** выбрать элемент **Projects**.
  - нажать **OK**,

4. Закрыть проект **MpiProject** (File/Close All).

9.1.2. Последовательность установки и подключения к среде разработки Microsoft Visual C++ библиотеки MPI

Соглашения и замечания:

7. В документе рассматривается реализация библиотеки выполненная Argonne National Laboratory Group для ОС Windows NT/2000 (версия библиотеки 1.2.5.1).

8. Здесь и далее термины MPI и MPICH используются как синонимы.

9. Здесь и далее используется сокращенное название среды разработки MSVC.

10. В документе используются названия и термины английской версии ОС Windows NT/2000.

11. Здесь и далее полагается, что <MPI_PATH> – путь к папке, в которую установлена библиотека MPI.

Действия:

12. Создать в среде MSVC новый проект и включить в него необходимые файлы с исходными текстами программы.

13. Установить пути к директориям, содержащим файлы заголовков и библиотек библиотеки MPI. Для этого следует выбрать пункт меню Tools\Options и перейти на вкладку Directories. Выбрать в выпадающем списке пункт Include files, вставить путь <MPI_PATH>\SDK\include, затем выбрать в выпадающем списке пункт Library files, вставить путь <MPI_PATH>\SDK\lib и нажать ОК.

14. Связать с проектом библиотеку с реализацией функций MPI. Для этого выбрать пункт меню Project\Settings, перейти на вкладку Linker и добавить к списку библиотек библиотеку mpich.lib.

15. Сохранить проект.

16. Теперь проект может быть откомпилирован.

Замечание: Не запускайте **MPI**-приложение непосредственно из среды (см. документ "Запуск **MPI**-приложений").

9.2. Запуск приложений с использованием MPI

9.2.1. Запуск программы при использовании Argonne National Lab

Запуск на локальном компьютере:

17. Запустите файловый менеджер **Far**.
18. Перейдите в папку с MPI-приложением.
19. В командной строке файлового менеджера Far наберите:

```
mpirun -localonly #processes exe [args...]
```

Здесь:

- `mpirun` – программа, осуществляющая запуск **MPI**-приложений.
- `-localonly` – ключ локального запуска.
- `#processes` – количество процессов, которое будет создано.
- exe – имя исполняемого файла **MPI**-приложения.
- [args...] – аргументы командной строки **MPI**-приложения, если они имеются.

В этом варианте запуска выполнение программы производится на одном компьютере в режиме разделения времени процессора для всех процессов программы.

Сетевой запуск:

Запускаемое в сетевом варианте **MPI**-приложение должно быть раскопировано с сохранением путей на все машины сети, на которых установлен **MPI**. Предположим для определенности, что у нас имеется сеть из 4 машин с именами: **ws1**, **ws2**, **ws3**, **ws4**. Пусть на машине **ws1** в папке d:\user\exec\ создано **MPI**-приложение **exam.exe**. На машинах **ws1**, **ws2**, **ws3** установлен **MPI**. Для сетевого запуска на указанных машинах файл **exam.exe** должен быть скопирован в каталог d:\user\exec\ на машины **ws2**, **ws3**.

Следующие пункты являются общими:

20. Запустите приложение **MPI Configuration Tool** (Start/Program/Argonne National Lab/MPICH.NT.1.2.0.4/Remote

Shell/MPI Configuration Tool) - в окне должен появиться список из четырех пунктов с названиями машин в сети.

21. Нажмите кнопку **Find**, в списке выделятся машины, на которых установлен **MPI** (точнее, компонента **Remote Shell**).

22. Снимите выделение с названий машин, на которых не будет запускаться **MPI**-приложение.

23. Нажмите кнопку **Set**.

24. В появившемся окне **MPICH Registry settings** можно осуществить некоторые настройки, однако чаще всего здесь просто необходимо нажать **OK**.

25. Запустите файловый менеджер **Far**.

26. Перейдите в папку с **MPI**-приложением.

Для продолжения действий по запуску MPI-программы возможны два различных варианта.

Способ 1:

В командной строке файлового менеджера **Far** наберите:

```
mpirun -np #processes exe [args...]
```

Способ 2:

Создайте конфигурационный файл по образцу:

```
exe d:\user\exec\ex.exe
hosts
ws1 #processes1
ws2 #processes2
ws3 #processes3
```

В командной строке наберите:

```
mpirun configfile [args...]
```

В результате данного варианта запуска программы на каждом вычислительном узле будет создано указанное количество процессов.

Литература

1. Kumar V., Grama A., Karypis G. Introduction to Parallel Computing. – The Benjamin/Cummings Publishing Company, Inc., 1994.
2. MPI:A Message-Passing Interface Standard (Version 1.1).
3. Performance Visualization for Parallel Programs (http://www-unix.mcs.anl.gov/perfvis/).
4. Web pages for MPI and MPE. (http://www.csa.ru/~il/mpich_doc/)
5. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону: Изд-во ООО "ЦВВР", 2003. 208 с.
6. Антонов А.С. Введение в параллельные вычисления (методическое пособие). М.: Изд-во физического факультета МГУ, 2002. 70 с.
7. Антонов А.С. Параллельное программирование с использованием технологии MPI. Изд-во Московского университета, 2004.
8. Корнеев В.В., Киселев А.В. Современные микропроцессоры. М.: Изд-во "Нолидж".
9. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. БХВ-Петербург, 2002. 608 с.
10. Корнеев В.В.. Параллельные вычислительные системы. М.:Изд-во "Нолидж", 1999.
11. Корнеев В.Д.Параллельное программирование в MPI. Новосибирск: Изд-во СО РАН, 2000. 213 с.
12. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Нижний Новгород: Изд-во ННГУ им. Н.И.Лобачевского, 2003. 184 с.
13. Высокопроизводительные параллельные вычисления на кластерных системах. Материалы Международного научно-практического семинара./ Под ред. проф. Р.Г. Стронгина. Нижний Новгород: Изд-во ННГУ им.Н.И.Лобачевского, 2002. 217 с.
14. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. БХВ-Петербург, 396 с., ISBN 5-94157-188-7.
15. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие. Минск: Изд-во БГУ, 2002. 323 с. ISBN 985-445-727-3.

СОДЕРЖАНИЕ

Владимир Александрович Гришагин
Алексей Николаевич Свистунов

Параллельное программирование на основе MPI

Учебное пособие

Редактор Е.В. Тамберг

Формат 60x84 1/16 Бумага офсетная . Печать офсетная.
Гарнитура Таймс. Усл. печ. л. 5,4. Уч.-изд. л. 6,4.
Тираж 200 экз. Заказ .

Издательство Нижегородского государственного
университета им. Н.И. Лобачевского
603950. Н. Новгород, пр.Гагарина, 23

Типография ННГУ. 603000, Н. Новгород, ул. Б. Покровская, 37