Министерство образования и науки Российской Федерации
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

Е.Н. Акимова

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Учебное пособие

Екатеринбург

УрФУ

2015

Оглавление

Введение

Характерная черта параллельных ЭВМ – возможность одновременного использования для обработки информации большого числа процессоров. Применение многопроцессорных вычислительных систем (МВС) ставит две задачи построения параллельных алгоритмов: распараллеливание существующих последовательных алгоритмов и создание новых алгоритмов с ориентацией на параллельные вычислительные системы. Проблемам распараллеливания алгоритмов посвящено множество статей и монографий отечественных и зарубежных авторов. Работы, посвященные параллельным вычислениям, идут по нескольким направлениям: исследование общих вопросов распараллеливания алгоритмов [1] – [2], построение алгоритмов для абстрактных параллельных вычислительных систем и реализация алгоритмов на реальных многопроцессорных вычислительных системах. Отметим обзоры В.Н. Фаддеевой и Д.К. Фаддеева [3], монографии Дж. Ортега [4], Е. Валяха [5], И.Н. Молчанова [6], сборник статей под редакцией Г. Родрига [7], коллективную монографию под редакцией Д. Ивенса [8], монографию В.Д. Корнеева [9], а также практические пособия-руководства [10] и [11].

Информация о многопроцессорном вычислительном комплексе МВС–1000 и среде параллельного программирования MPI доступна в Интернете на Web-сайтах: http://www.parallel.ru/, http://www.jscc.ru/, http://parallel.imm.uran.ru/.

Данное учебное пособие предназначено для студентов радиотехнического факультета специальности «Математическое обеспечение и администрирование информационных систем». Учебное пособие содержит теоретический и практический материал по разделам курса «Параллельные вычисления», читаемого в УГТУ–УПИ.

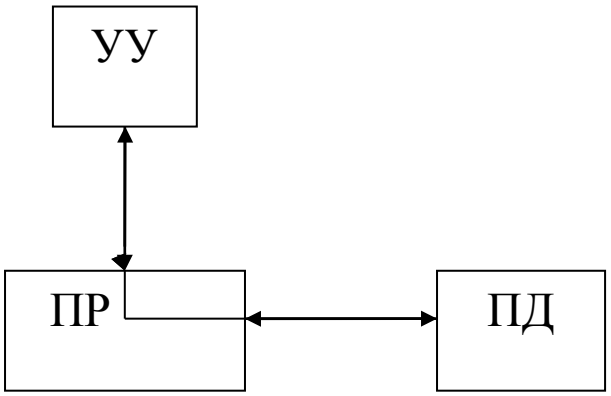
По своей структуре пособие состоит из трех частей. В первой части приводится обзор архитектур многопроцессорных вычислительных систем, рассматриваются понятия эффективности и ускорения параллельных алгоритмов. Во второй части описывается среда параллельного программирования – коммуникационная библиотека MPI, предназначенная для разработки параллельных программ на МВС. В третьей части приводятся примеры параллельных программ и виды курсовых работ для студентов радиотехнического факультета УГТУ–УПИ.

1. Обзор архитектур многопроцессорных вычислительных систем

1.1. Классификация параллельных вычислительных систем

Самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 г. М. Флинном [12]. Классификация базируется на понятии *потока*, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

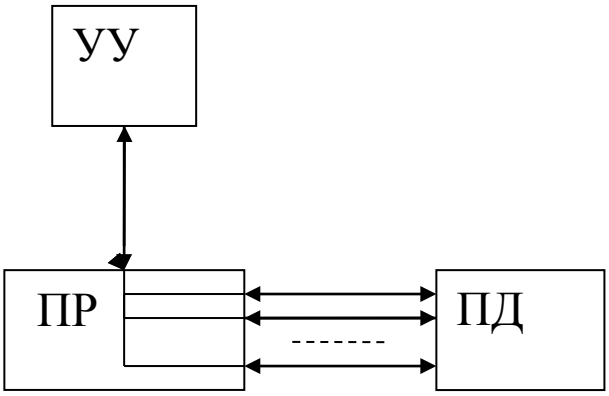
1. Класс **SISD** (single instruction stream / single data stream)



– одиночный поток команд и одиночный поток данных. К этому классу относятся прежде всего классические последовательные машины, или иначе – машины фон-неймановского типа, например PDP-11 или VAX 11/780. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним потоком данных. Не имеет значения тот факт, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка – как машина CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными попадают в этот класс.

Многими исследователями подмечено, что в этот класс можно включить и векторно-конвейерные машины, если рассматривать вектор как одно неделимое данное для соответствующей команды. В таком случае в этот класс попадут и такие системы, как CRAY-1, CYBER 205, машины семейства FACOM VP и многие другие.

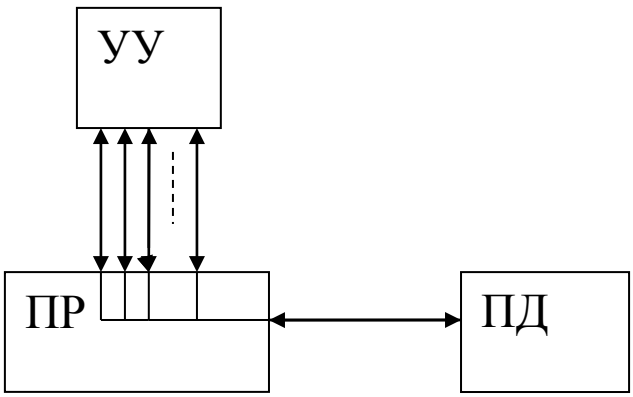
2. Класс **SIMD** (single instruction stream / multiple data stream)



– одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными – элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производиться либо процессорной матрицей, как в ILLIAC IV, либо с помощью конвейера, как, например, в машине CRAY-1. Бесспорными представителями класса SIMD считаются матрицы процессоров: ILLIAC IV, ICL DAP, Goodyear Aerospace MPP, Connection Machine 1 и т.п. В таких системах единое управляющее устройство контролирует множество процессорных элементов.

Каждый процессорный элемент получает от устройства управления в каждый фиксированный момент времени одинаковую команду и выполняет ее над своими локальными данными. Для классических процессорных матриц никаких вопросов не возникает, однако в этот же класс можно включить и векторно-конвейерные машины, например CRAY–1. В этом случае каждый элемент вектора надо рассматривать как отдельный элемент потока данных.

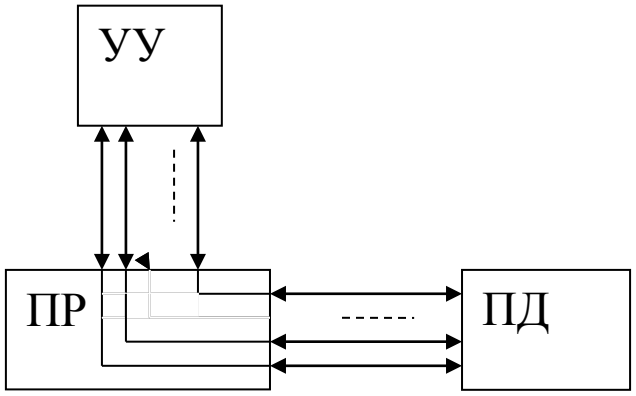
3. Класс **MISD** (multiple instruction stream / single data stream)



– множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако ни Флинн, ни другие специалисты в

области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на этом принципе. Ряд исследователей относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе. Будем считать, что пока данный класс пуст.

4. Класс **MIMD** (multiple instruction stream / multiple data stream)



– множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

Класс MIMD чрезвычайно широк, поскольку включает в себя всевозможные мультипроцессорные системы: Cm*, C.mmp, CRAY Y-MP, Denelcor HEP, BBN Butterfly, Intel Paragon, CRAY T3D и многие другие. Интересно то, что если конвейерную обработку рассматривать как выполнение множества команд (операций ступеней конвейера) не над одиночным векторным потоком данных, а над множественным скалярным потоком, то все рассмотренные выше векторно-конвейерные компьютеры можно расположить и в данном классе.

Предложенная схема классификации вплоть до настоящего времени является самой применяемой при начальной характеристике того или иного компьютера. Если говорится, что компьютер принадлежит классу SIMD или MIMD, то сразу становится понятным базовый принцип его работы, и в некоторых случаях этого бывает достаточно. Однако видны и явные недостатки. В частности, некоторые заслуживающие внимания архитектуры, например dataflow и векторно-конвейерные машины, четко не вписываются в данную классификацию. Другой недостаток – это чрезмерная заполненность класса MIMD. Необходимо средство, более избирательно систематизирующее архитектуры, которые по Флинну попадают в один класс, но совершенно различны по числу процессоров, природе и топологии связи между ними, по способу организации памяти и, конечно же, по технологии программирования.

Наличие пустого класса (MISD) не стоит считать недостатком схемы. Такие классы, по мнению некоторых исследователей в области классификации

архитектур, могут стать чрезвычайно полезными для разработки принципиально новых концепций в теории и практике построения вычислительных систем.

Дополнения Ванга и Бриггса к классификации Флинна

В книге К. Ванга и Ф. Бриггса [13] сделаны некоторые дополнения к классификации Флинна. Оставляя четыре ранее введенных базовых класса (SISD, SIMD, MISD, MIMD), авторы внесли следующие изменения.

Класс SISD разбивается на два подкласса:

- архитектуры с единственным функциональным устройством, например PDP-11;
- архитектуры, имеющие в своем составе несколько функциональных устройств CDC 6600, CRAY-1, FPS AP-120B, CDC Cyber 205, FACOM VP-200.

В класс SIMD также вводится два подкласса:

- архитектуры с пословно-последовательной обработкой информации – ILLIAC IV, PEPE, BSP;
- архитектуры с разрядно-последовательной обработкой – STARAN, ICL DAP.

В классе MIMD авторы различают:

- вычислительные системы со слабой связью между процессорами, к которым они относят все системы с распределенной памятью, например Cosmic Cube;
- вычислительные системы с сильной связью (системы с общей памятью), куда попадают такие компьютеры, как C.mmp, BBN Butterfly, CRAY Y-MP, Denelcor HEP.

Рассмотрим подробнее некоторые машины.

CDC 6600 (1964 г.)

Фирма Control Data Corporation (CDC) при непосредственном участии одного из ее основателей, Сеймура Р. Крэя (Seymour R. Cray), выпускает компьютер CDC-6600 – первый компьютер, в котором использовалось несколько независимых функциональных устройств. Для сравнения с сегодняшним днем приведем некоторые параметры компьютера:

- время такта 100 нс,
- производительность 2-3 млн. опер./сек,
- оперативная память разбита на 32 банка по 4096 60-разрядных слов,
- цикл памяти 1мкс,
- 10 независимых функциональных устройств.

Машина имела громадный успех на научном рынке, активно вытесняя машины фирмы IBM.

CDC 7600 (1969 г.)

CDC выпускает компьютер CDC-7600 с восемью независимыми конвейерными функциональными устройствами – сочетание параллельной и конвейерной обработки. Основные параметры:

- такт 27,5 нс,
- 10-15 млн. опер./сек,
- 8 конвейерных ФУ,
- 2-уровневая память.

Архитектура CRAY – 1

В 70-х гг. бывший сотрудник и один из руководителей фирмы CDC Seymour R. Cray (Сеймур Р. Крей) организовал собственную фирму, которая занялась проектированием сверхбыстродействующей ЭВМ, известной под названием Cray-1 с быстродействием, превосходящим 150 млн. опер./сек с широким использованием новой интегральной технологии.

Основная память машины Cray-1 в отличие от других высокопроизводительных машин не имеет иерархической структуры, она столь быстра, что в такой иерархии отпала необходимость. Считается, что машина Cray-1 является наиболее быстродействующей из класса однопроцессорных систем.

О структурной организации этой машины целесообразно рассказать более подробно. Изложение будет основано на интересной статье Ричарда М. Расселла, одного из разработчиков этой машины, в которой приведены основные технические данные, архитектурные особенности и некоторые соображения, положенные в основу принятых структурных решений (Richard M. Russell. The CRAY-1 Computer System. Communication of the ACM. January 1978, Volume 21, Number 1).

Расселл относит машину Cray-1 к классу сверхвысокопроизводительных векторных процессоров. К этому классу относятся также машины ILLIAC IV, STAR-100, ASC.

В состав центрального процессора Cray-1 входят:

- главная память, объемом до 1048576 слов, разделенная на 16 независимых по обращению блоков, емкостью 64К слов каждый;
- регистровая память, состоящая из пяти групп быстрых регистров, предназначенных для хранения и преобразования адресов, для хранения и обработки векторных величин;
- функциональные модули, в состав которых входят 12 параллельно работающих устройств, служащих для выполнения арифметических и логических операций над адресами, скалярными и векторными величинами;
- устройство, выполняющее функции управления параллельной работой модулей, блоков и устройств центрального процессора;
- 24 канала ввода-вывода, организованные в 6 групп с максимальной пропускной способностью 500000 слов в секунду (2 млн. байт/сек).

Двенадцать функциональных устройств машины Cray-1, играющих роль арифметико-логических преобразователей, не имеют непосредственной связи с главной памятью. Так же как и в машинах семейства CDC-6000, они имеют доступ только к быстрым операционным регистрам, из которых выбираются операнды и на которые засылаются результаты после выполнения соответствующих действий.

Математическое обеспечение машины Cray-1, так же как у ее предшественников – машин фирмы CDC, «фортранно-ориентировано». Это означает, что в качестве основного входного языка выбран Фортран, наиболее интенсивно используемый в научных расчетах. Для машины Cray-1 создан специальный оптимизирующий транслятор со стандартного Фортрана, учитывающий специфику этой векторной машины. При некоторых условиях, наложенных на программу, данный транслятор обеспечивает конвейерный параллелизм и готовит объектные программы, эффективно использующие эти возможности машины. Создана новая версия транслятора, который оптимизирует фортранные программы, составленные без каких-либо ограничений.

Операционная система **COS** (Cray Operating System) предназначена для режима пакетной обработки и дистанционной пакетной обработки заданий, полученных с удаленных терминалов. Операционная система рассчитана на мультипрограммную обработку одновременно до 63 активных задач. Для обеспечения работы в режиме дистанционного доступа в качестве машины-сателлита вычислительной системы Cray-1 используется мини-машина Eclipse. С этой машиной связаны внешние каналы центрального вычислителя с помощью специальных сопрягающих устройств. На мини-машину возлагаются функции управления приемом-передачей информации, управления линиями связи, т. е. функции процессора передачи данных. Фирма Cray разрабатывает для этих целей собственный специализированный микрокомпьютер, который в скором будущем должен заменить мини-машину другой фирмы. Машина Cray-1, как уже говорилось, на сегодняшний день считается наиболее быстродействующей в классе универсальных ЭВМ для научных расчетов. Ее производительность сильно зависит от характера решаемых задач. Экспериментальная проверка показала, что она колеблется от 20 до 160 млн. опер./сек. При выполнении операций с плавающей запятой диапазон ее быстродействия оценивается в пределах от 20 до 60 млн. опер./сек.

Проект ILLIAC IV

Система ILLIAC IV разработана в 1966 – 1972 гг. в Иллинойсском университете и изготовлена фирмой «Барроуз» (рис.1).

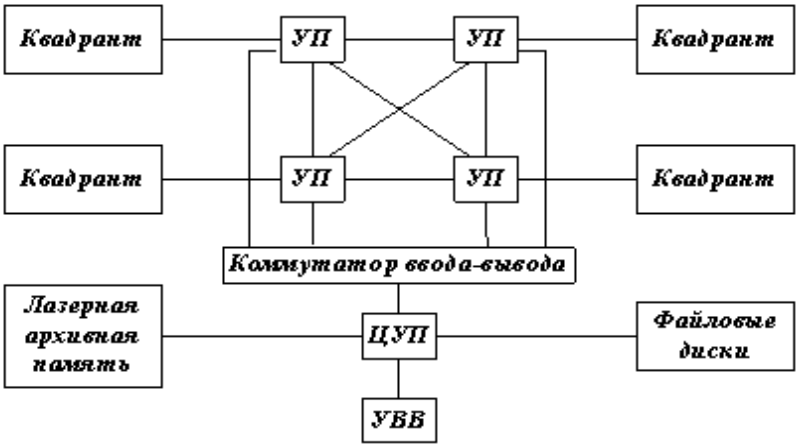


Рис. 1. Система ILLIAC-IV (проект)

По первоначальному проекту система ILLIAC-IV должна была включать в себя 256 ПЭ, разбитых на 4 группы, – квадранты, каждый из которых должен управляться специальным процессором (УП). Управление всей системой, содержащей кроме ПЭ и УП также внешнюю память и оборудование ввода – вывода, предполагалось от центрального управляющего процессора (ЦУП). Однако реализовать этот замысел не удалось из-за возникших технологических трудностей при создании интегральных схем, ОЗУ и удорожания всего проекта почти в два раза. В результате с опозданием на два года (в 1971 г.) система была создана в составе одного квадранта и одного УП и с начала 1974 г. введена в эксплуатацию (рис. 2). Ранее предполагалось получить на этой системе производительность примерно 1 млрд. опер./сек, однако реализовано 200 млн. опер./сек. Тем не менее этого оказалось достаточно, чтобы система в течение ряда лет считалась самой высокопроизводительной в мире.

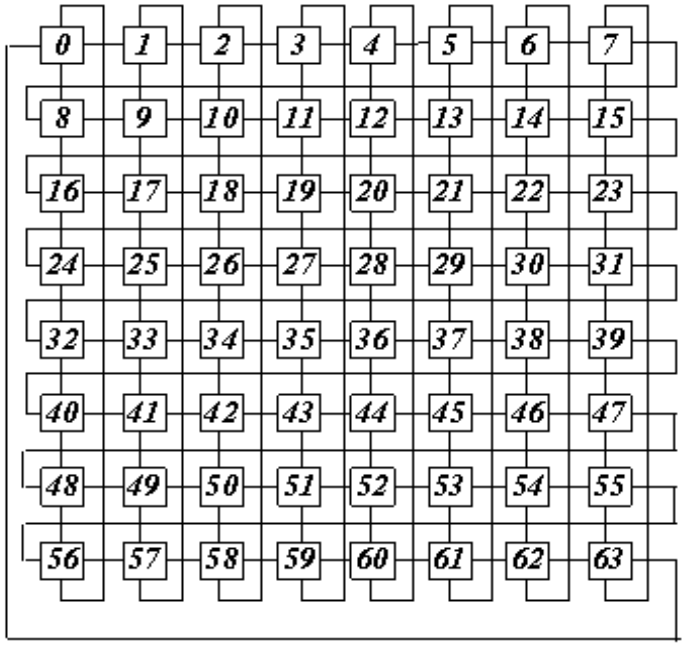


Рис. 2 Система ILLIAC-IV

Квадрант системы ILLIAC-IV

В каждом квадранте 64 ПЭ образуют матрицу размером 8x8. Схема связей между ПЭ похожа на схему системы SOLOMON, но связь с внешней средой имеют все ПЭ без исключения. Реально действующая система ILLIAC-IV состоит, таким образом, из двух частей: центральной с устройством управления и 64 ПЭ, а также подсистемы ввода – вывода, включающей в себя универсальную ЭВМ B-6700, файловые диски и лазерную архивную память большой емкости. Каждый ПЭ состоит из собственно процессора и ОЗУ. Процессор оперирует с 64-разрядными числами и выполняет универсальный набор операций. Быстродействие процессора достаточно высокое, операция сложения 64-разрядных чисел выполняется за 240 нс, а умножения – за 400 нс. Таким образом, процессор выполняет в среднем 3 млн. операций в секунду, а следовательно, производительность системы равна 3x64≈200 млн. опер./сек.

Емкость ОЗУ каждого ПЭ составляет 2048 64-разрядных слов, длительность цикла обращения к памяти 350 нс. Память выполнена на интегральных схемах. Каждый процессор имеет счетчик адресов и индексный регистр, так что конечный адрес в каждом процессоре может формироваться как сумма трех составляющих: адреса, содержащегося в команде для данного ПЭ, кода, содержащегося в центральном индексном регистре УУ, и кода, содержащегося в собственном индексном регистре. Это существенно повышает гибкость системы по сравнению с системой SOLOMON, где все ПЭ выбирают информацию по одному адресу. Каждый процессор кроме индексного регистра имеет в своем составе пять программно-адресуемых регистров: накапливающийся сумматор, регистр для операндов, регистр пересылок, используемый при передачах от одного ПЭ к другому, буферный регистр на одно слово и регистр управления состоянием ПЭ (аналогичный регистру моды в системе SOLOMON). Регистр управления имеет 8 разрядов. В зависимости от содержимого этого регистра ПЭ становится активным или пассивным, а также выполняет ряд пересылочных операций. Если вычисления не требуют полной разрядности, то процессор может быть разбит на два 32-разрядных подпроцессора или даже восемь 8-разрядных. Это позволяет в случае необходимости обрабатывать векторные операнды из 64, $2\times 64 = 128$ и $8\times 64 = 512$ элементов.

Каждый *i*-й ПЭ связан с четырьмя другими: $(i-1)$–, $(i+1)$–, $(i+8)-$ и $(i-8)-$м. При такой связи передача данных между любыми двумя ПЭ осуществляется не более чем за 7 шагов, а среднее число шагов равно 4.

По шине состояния ПЭ могут передавать сигналы о состоянии в УУ, которое таким образом всегда определяет состояние системы. Оперативное ЗУ каждого процессорного элемента связано со своим процессором, устройством управления центральной частью и подсистемой ввода – вывода.

Подсистема ввода – вывода включает в себя стандартную ЭВМ B-6700 (первоначально B-6500) и два уровня внешней памяти: на магнитных дисках с фиксированными дорожками и лазерную память. Накопители на дисках имеют

магнитные головки для каждой дорожки (128 головок на диск), и обмен данными осуществляется по 256-разрядной шине. Емкость каждого диска – около 1 млрд. бит. Для того чтобы согласовать скорость передачи информации с дисков и работу управляющей ЭВМ, в систему включено буферное ОЗУ, состоящее из четырех модулей памяти.

Лазерная память представляет собой одностороннее ЗУ очень большой емкости (1012 бит). Информация записывается на тонкой металлической пленке путем прожигания микроотверстий лазерным лучом. Емкость ЗУ – 1200 млрд. бит. Время доступа к данным от 0,2 до 5 с.

Машина B-6700 выполняет и различные другие функции: транслирует и компонует программы, управляет запросами на ресурсы, производит предварительную обработку информации и т. д.

Следует подчеркнуть, что сверхвысокая производительность системы достигается только на определенных типах задач, таких, например, как операции над матрицами, быстрое преобразование Фурье, линейное программирование, обработка сигналов, где как раз имеет место параллелизм данных или параллелизм независимых объектов. Необходимо отметить также и то, что разработка программ для систем ILLIAC-IV, обеспечивающих высокую производительность, является весьма сложным делом. Для упрощения этой задачи были разработаны специальные алгоритмические языки.

Система ILLIAC-IV была включена в состав вычислительной сети ARPA. В результате усовершенствования программного обеспечения производительность системы выросла до 300 млн. опер./сек.

1.2. Архитектура и программное обеспечение МВС–1000

Общее описание архитектуры

Основой системы является масштабируемый массив процессорных узлов. Каждый узел содержит вычислительный микропроцессор Alpha 21164 с производительностью 2 GFlops при тактовой частоте 500 МГц и оперативную память объемом 128 Мбайт с возможностью расширения.

Процессорные узлы взаимодействуют через коммуникационные процессоры TMS320C44 производства Texas Instruments, имеющие по 4 внешних канала (линка) с общей пропускной способностью 80 Мбайт/сек (20 Мбайт/сек каждый). Также разрабатывается вариант системы с использованием коммуникационных процессоров SHARC (ADSP 21060) компании Analog Devices, имеющих по 6 каналов с общей пропускной способностью до 240 Мбайт/сек (40 Мбайт/сек каждый).

Топология сети

Процессорные узлы связаны между собой по оригинальной схеме, сходной с топологией двумерного тора (для 4-линковых узлов). Структурный модуль (рис. 3) состоит из 16 вычислительных модулей, образующих матрицу 4×4. При этом четыре угловых элемента матрицы соединяются через транспьютерные линки по диагонали попарно. Оставшиеся 12 линков предназначаются для подсоединения внешних устройств (4 линка угловых ВМ)

и соединений с подобными ВМ. Максимальная длина пути в таком структурном модуле равна трем (против шести в исходной матрице 4×4).

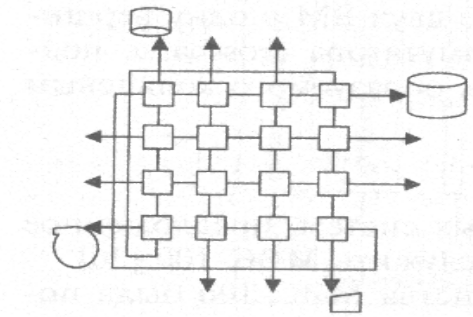


Рис. 3. Структурный модуль системы, решетка 4×4

Конструктивным образованием МВС–1000 является базовый вычислительный блок, содержащий 32 вычислительных модуля (рис. 4). Максимальная длина пути между любыми из 32 вычислительных модулей равна пяти, как в булевском гиперкубе. При этом число свободных линков после комплектации блока составляет 16, что позволяет продолжить процедуру объединения. Возможные схемы объединения базовых блоков в различные многопроцессорные системы приведены на рис. 5 и 6.

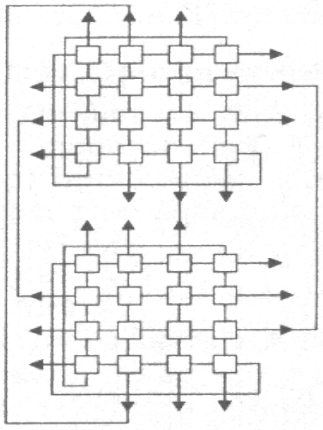


Рис. 4. Базовый вычислительный блок, 32 вычислительных модуля

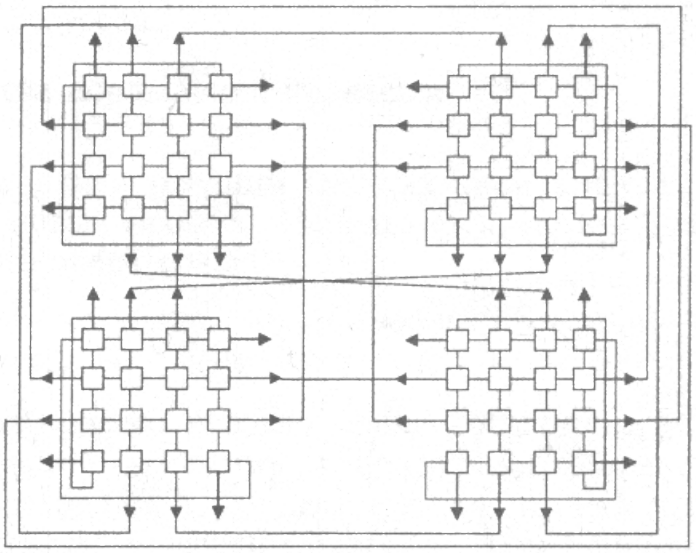


Рис. 5. Топология 64-процессорной системы МВС–1000

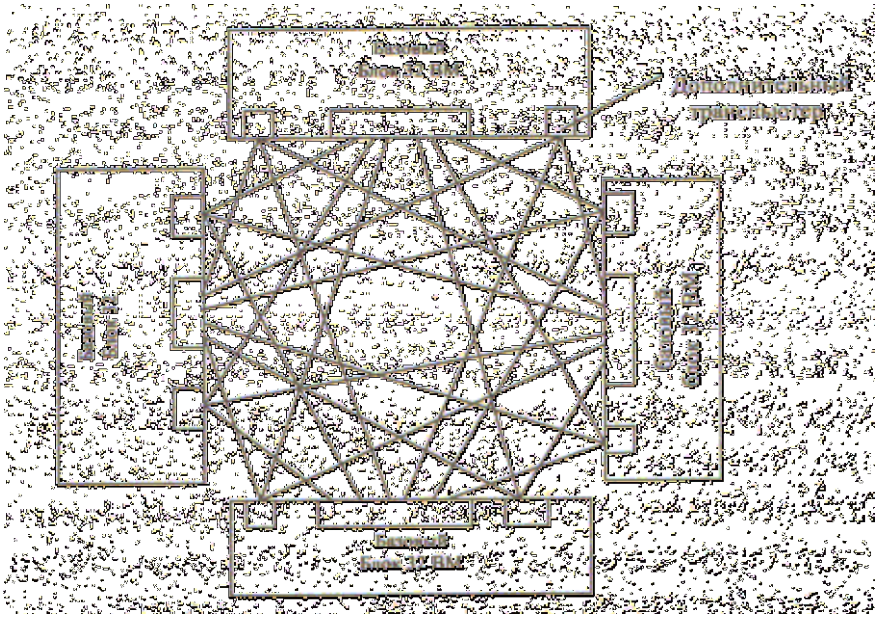


Рис. 6. Структура 128-процессорной системы МВС–1000, 4 базовых блока

Конструктивные решения МВС–1000

Процессорный узел смонтирован на типовой многослойной плате. В конструктивном модуле в виде стандартной стойки размером $0.6 \times 0.8 \times 2.2$ м^3 размещается до 64 узлов с системой электропитания и охлаждения. Вес стойки – 200 кг, электропотребление – 4 кВт. Система МВС–1000 с производительностью до 1 TFlops состоит из 8 стоек (512 узлов).

Внешний доступ и управление системой

Для управления массивом процессоров и внешними устройствами, а также для доступа к системе извне используется так называемый хост-компьютер (управляющая машина). Обычно это рабочая станция AlphaStation с процессором Alpha и операционной системой Digital Unix (Tru64 Unix) или ПК на базе Intel с операционной системой Linux.

Программное обеспечение

Пользователям предоставляются компиляторы языков Fortran 77 и C/C++. Коммуникационное ПО в МВС–1000 строится на основе специализированной транспортной службы Router+. На базе Router+ реализован адаптированный к МВС–1000 интерфейс параллельного программирования MPI. Аналогично могут быть реализованы интерфейсы PVM, GNS, DVM, HPF и др. Пользователи могут также вызывать функции Router+ непосредственно в своих программах. В планах разработчиков также поддержка на вычислительных узлах стандартных протоколов TCP/IP. Для задач визуализации разработана специализированная библиотека GraphLib.

В настоящее время производится разработка системы управления очередями и распределения нагрузки, а также адаптация системы профилирования и трассировки параллельных программ.

Реализован многопользовательский режим и удаленный доступ к системе через специальный промежуточный компьютер (gateway). Для пользователей обеспечивается Unix-совместимая среда компиляции и запуска программ.

Различные модификации МВС–1000

Суперкомпьютер (СК) МВС–1000М предназначен для решения сложных научно-технических задач.

Пиковая производительность СК МВС–1000М составляет 10^{12} операций с плавающей точкой с двойной точностью в секунду. Общий объем оперативной памяти решающего поля – 768 Гбайт. Для размещения СК МВС–1000М требуется 100 м^2. Потребляемая мощность составляет 120 кВА.

Программные и аппаратные средства СК МВС–1000М позволяют решать одну задачу с использованием всего вычислительного ресурса, а также разделять решающее поле на части требуемого размера и предоставлять их нескольким пользователям.

Общая структура МВС–1000М (рис.7)

В состав технических средств СК МВС–1000М входят:

- решающее поле из 768 процессоров Alpha 21264, разбитое на 6 базовых блоков, состоящих из 64 двухпроцессорных модулей;
- управляющая ЭВМ;
- файл-сервер NetApp F840;
- сеть Myrinet 2000;
- сети Fast/Gigabit Ethernet;
- сетевой монитор;
- система бесперебойного электропитания.

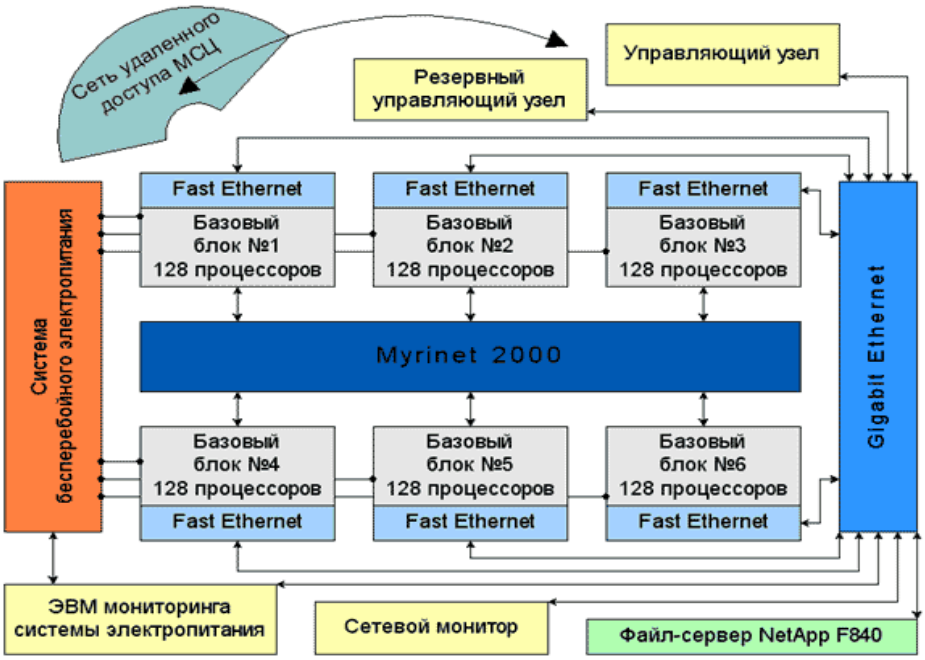
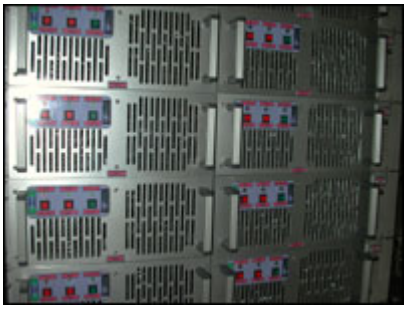


Рис. 7. МВС–1000М

Вычислительный модуль



Решающее поле СК МВС–1000М состоит из 384 двухпроцессорных вычислительных модулей (ВМ). Каждый ВМ включает:

- 2 процессора Alpha 21264 667 МГц с кэш-памятью 2-го уровня объемом 4 Мбайта;
- 2 Гбайта разделяемой оперативной памяти;
- жесткий диск объемом 20 Гбайт;
- интерфейсную плату сети Myrinet;
- интерфейсную плату сети Fast Ethernet;
- интерфейсную плату видеоконтроллера;
- источник питания мощностью 600 вт.

Пиковая производительность одного ВМ составляет 2,7 млрд. операций с плавающей точкой с двойной точностью в секунду.

Сетевые решения

Вычислительные модули связаны между собой высокоскоростной сетью Myrinet2000 (пропускная способность канала равна 2000 Мбит/сек) и сетью Fast Ethernet (пропускная способность канала равна 100 Мбит/сек).

Сеть Myrinet2000 предназначена для высокоскоростного обмена между ВМ в ходе вычислений.

Сеть Fast Ethernet предназначена для начальной загрузки программ и данных в ВМ, а также для передачи служебной информации о ходе вычислительного процесса.

Сеть Gigabit Ethernet предназначена для соединения решающего поля с управляющей ЭВМ и файл-сервером.

Сеть Myrinet2000 в СК МВС–1000М реализована на базе шести 128-входовых полносвязных коммутаторов. При обмене данными между двумя ВМ с использованием протоколов MPI достигается пропускная способность на уровне 110 – 150 Мбайт/сек.

Подсистема удаленного управления и непрерывного мониторинга

Подсистема удаленного управления и непрерывного мониторинга работы СК МВС–1000М обеспечивает:

1) мониторинг состояния процессоров ВМ;
2) мониторинг состояния оборудования сети Myrinet;
3) мониторинг доступности ВМ по сети Myrinet;
4) мониторинг доступности ВМ по сети Fast Ethernet;
5) мониторинг загруженности ВМ;
6) обработку сигналов от датчиков, имеющихся в ВМ (температуры процессоров, состояния вентиляторов и т.п.);
7) пересылку «критической» информации журналов (логов) ВМ на управляющую ЭВМ;
8) инициацию отключения питания ВМ при возникновении аварийных ситуаций;
9) мониторинг доступности сетевых файловых систем;

10) оповещение администратора системы о выявленных неисправностях по электронной почте;

11) сбор статистики и графическую визуализацию активности в сетях Fast Ethernet Myrinet.

Подсистема коллективного доступа

Подсистема коллективного доступа к ресурсам суперкомпьютера обеспечивает:

1) прием заданий пользователей и постановку этих заданий в очередь;

2) динамическое распределение ресурсов суперкомпьютера по запросам пользователей, при этом единицей ресурсов является один процессор СК МВС–1000М;

3) выполнение заданий пользователей в пакетном режиме;

4) выполнение на СК МВС–1000М задач пользователей как содержащих, так и не содержащих функции MPI;

5) сбор статистики о выполнении заданий пользователей и формирование соответствующих отчётов для анализа характеристик пользовательских задач;

6) графический интерфейс мониторинга производительности СК МВС–1000М и управления заданиями.

Взаимодействие удаленных пользователей с управляющей ЭВМ осуществляется по протоколу ssh.

Коммуникационная среда Myrinet поддерживается в современных реализациях интерфейса параллельного программирования MPI. В качестве программных средств коммуникационной среды Myrinet используется коммуникационная система GM. В ее состав входят:

- драйвер;
- служебные программы;
- тестовые программы;
- библиотека функций и заголовочный файл GM API;
- демонстрационные программы.

Система питания и охлаждения

Общая потребляемая мощность СК МВС–1000М составляет около 120 кВА. Охлаждение стоек – воздушное. Охлажденный воздух поступает на вход стоек из подстоечного пространства.

СК МВС–1000М оснащен системой бесперебойного питания. В состав системы бесперебойного питания включен монитор системы электропитания.

1.3. Вычислительные ресурсы ИММ УрО РАН

ИММ УрО РАН предоставляет пользователям для работы следующие вычислительные средства:

1) **МВС–1000/16** – многопроцессорный вычислитель кластерного типа (14 процессоров PENTIUM3 (800 МГц) с двумя картами FastEthernet) с распределенной памятью (MPP) (рис. 8) (производство НИИ «Квант»);

2) **МВС–1000/17ЕК** – многопроцессорный вычислитель кластерного типа (16 2-процессорных модулей Xeon (2,4 ГГц) с двумя картами GbitEthernet) (рис. 9) (производство НИИ «Квант»);
3) **PrimePower 850** – 8-процессорный вычислитель с общей памятью (SMP) 32 Гбайт (страна-производитель – Германия);
4) **МВС–1000М/266** – многопроцессорный вычислитель кластерного типа (128 2-процессорных модулей Alpha (667 МГц) с картой Myrinet).

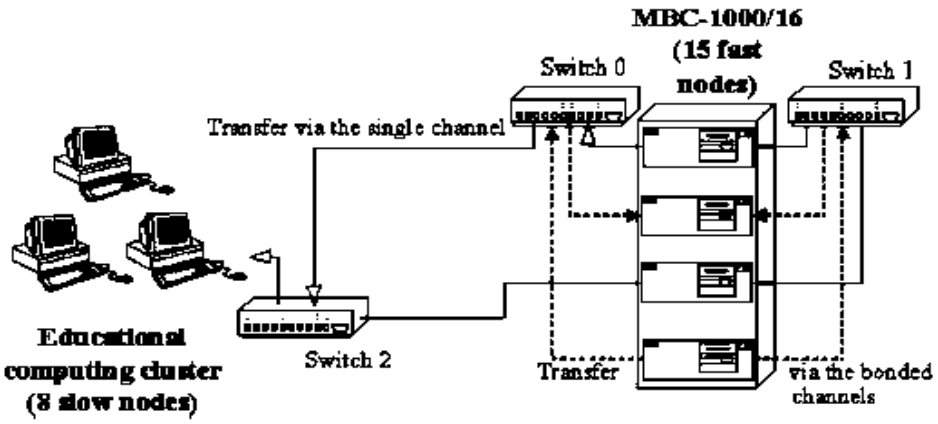
№ n/n	Модель вычислителя	Архитектура	Количество процессоров	Тип процессора	Объем ОП, Гбайт	Коммуникационная среда	Пиковая производительность, Gflops
1	МВС-1000/16 (UM16)	MPP	14	Pentium3 (800.0 МГц)	2	Fast Ethernet x2	13
2	МВС-1000/17ЕК (UM32)	MPP	2x16	Xeon (2.4 ГГц)	68	Gbit Ethernet x2	160
3	PrimePower 850 (UMF)	SMP	8	SPARC 64 (1.08 ГГц)	32	Коммутатор «точка-точка»	48
4	МВС-1000М/266 (UMA)	MPP	2x128	Alpha (667 МГц)	256	Myrinet	330

Операционная система Linux используется для 1, 2, 4-го, а Solaris – для 3-го.

Для организации межпроцессорного взаимодействия можно использовать следующие **библиотеки обмена сообщениями:**
MPI (реализация MPICH);
MPI (реализация LAM);
библиотека TCP Router .

Для компиляции программ доступны **Intel компиляторы** (бесплатно для некоммерческого использования в версиях Linux)
и свободно распространяемые **компиляторы серии gcc** (Gnu Compiller Collection) с языков **C , C++ , Fortran95**.

Установлена **система DVM** – полноценная система параллельного программирования на языках **C-DVM и Fortran-DVM**.



MBC–1000/16

Рис. 8. MBC–1000/16

Рис. 9. MBC–1000/32

Рис. 9. МВС–1000/32 (продолжение)

2. Понятие параллельных вычислений

Под *параллельными вычислениями (parallel or concurrent computations)* можно понимать процессы решения задач, в которых в один и тот же момент времени могут выполняться одновременно несколько вычислительных операций.

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

Параллельная обработка. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за 1000/N единиц времени. Подобные аналогии можно найти и в жизни: если один солдат вскопает огород за 10 часов, то рота солдат из пятидесяти человек с такими же способностями, работая одновременно, справятся с той же работой за 12 минут – принцип параллельности в действии!

Пионером в параллельной обработке потоков данных был академик А.А. Самарский, выполнявший в начале 50-х гг. расчеты, необходимые для моделирования ядерных взрывов. Самарский решил эту задачу, посадив несколько десятков барышень с арифмометрами за столы. Барышни передавали данные друг другу просто на словах и откладывали необходимые цифры на арифмометрах. Таким образом, в частности, была рассчитана эволюция взрывной волны. Это и была первая параллельная система. Хотя расчеты водородной бомбы были мастерски проведены, точность их была очень низкая, потому что узлов в используемой сетке было мало, а время счета получалось слишком большим.

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

Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передавал бы результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется

за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если каждую микрооперацию выделить в отдельный этап (или иначе говорят – ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находиться первые пять пар аргументов, а весь набор из ста пар будет обработан за 5+99=104 единицы времени – ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера).

Казалось бы, конвейерную обработку можно с успехом заменить обычным параллелизмом, для чего продублировать основное устройство столько раз, сколько ступеней конвейера предполагается выделить. В самом деле, пять устройств предыдущего примера обработают 100 пар аргументов за 100 единиц времени, что быстрее времени работы конвейерного устройства! В чем же дело? Ответ прост: увеличив в пять раз число устройств, мы значительно увеличиваем как объем аппаратуры, так и ее стоимость. Представьте себе, что на автозаводе решили убрать конвейер, сохранив темпы выпуска автомобилей. Если раньше на конвейере одновременно находилась тысяча автомобилей, то, действуя по аналогии с предыдущим примером, надо набрать тысячу бригад, каждая из которых (1) в состоянии полностью собрать автомобиль от начала до конца, выполнив сотни разного рода операций, и (2) сделать это за то же время, в течение которого машина прежде находилась на конвейере.

Общая схема организации:

- разделение процесса вычислений на части, которые могут быть выполнены одновременно;
- распределение вычислений по процессорам;
- обеспечение взаимодействия параллельных вычислений.

Возможные постановки проблемы

Формирование методов параллельных вычислений:

- разработка новых параллельных алгоритмов решения;
- распараллеливание последовательного метода;
- распараллеливание последовательной программы.

Учет особенностей вычислительной системы:

- выбор наилучшей архитектуры параллельной системы под процесс вычислений;
- создание параллельных методов, учитывающих свойства конкретной вычислительной системы.

Уровни организации параллелизма

1. Низкоуровневый способ распараллеливания (*fine granularity*):

- уровень команд (*instruction level*);
- уровень блоков команд или циклов (*block* or *loop level*).

2. Высокоуровневый способ распараллеливания (*coarse granularity*):

- уровень подпрограмм (*procedure level*);
- уровень отдельных заданий (*task level*).

3. Эффективность и ускорение параллельного алгоритма

Для исследования параллельных свойств и сравнения работы последовательного и параллельного алгоритмов вводятся некоторые характеристики. Основные – это коэффициенты ускорения и эффективности.

Показатели эффективности:

- ускорение

$$S_p = T_1 / T_p \le p\ ;$$

- эффективность

$$E_p = S_p / p \le 1,$$

где T_p – время выполнения параллельного алгоритма на МВС с числом процессоров p ($p > 1$); T_1 – время выполнения последовательного алгоритма на одном процессоре. T_p представляет собой совокупность чистого времени счета и накладных расходов на межпроцессорные обмены, т.е. $T_p = T_c + T_o$.

В общем случае эффективность распараллеливания меняется в пределах $0 < E_p < 1$. В идеальном случае при равномерной и сбалансированной загрузке процессоров и минимальном времени обменов между ними $E_p \to 1$, но при решении практических задач она уменьшается за счет накладных расходов.

Основной целью при построении параллельных алгоритмов является получение максимального ускорения и эффективности.

Но существует ряд зависимых друг от друга факторов, препятствующих этому:

1) различная степень параллелизма на разных этапах параллельного алгоритма;

2) несбалансированность нагрузки процессоров;

3) задержки, вызванные обменами, конфликтами в памяти и временем синхронизации.

Условия высокой эффективности:

- равномерная загрузка процессоров (отсутствие простоев);
- низкая интенсивность взаимодействия процессоров (независимость);
- масштабируемость.

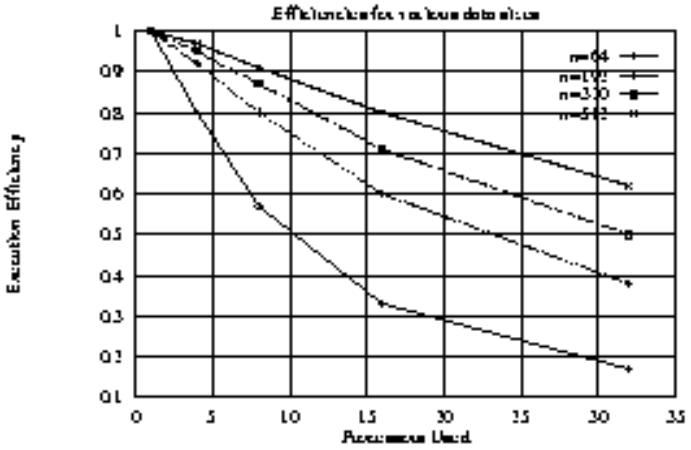
Под **масштабируемостью** (scalability) параллельного алгоритма понимается его возможность ускорения вычислений пропорционально увеличению числа используемых процессоров:

$p \to 2p \ \Rightarrow\ S \to 2S$ для некоторых $m \ge n$.

Масштабируемость есть возможность алгоритма обеспечения постоянного значения эффективности вычислений при увеличении числа процессоров (возможно, за счет увеличения размера задачи).

На рисунке приведен пример сложения n значений с эффективностью $E = 0.8$

- для $p = 4$ при $n = 64$,
- для $p = 8$ при $n = 192$,
- для $p = 16$ при $n = 512$.



Функция **изоэффективности** (isoefficiency fucntion) есть зависимость размера решаемой задачи от числа используемых процессоров для обеспечения постоянного уровня эффективности параллельных вычислений:

$$W = K f(p),$$

где W – показатель вычислительной сложности задачи (количество операций);
K – коэффициент, зависящий только от значения показателя эффективности.

Накладные расходы изоэффективности (overhead function):

$$T_0 = p T_p - W.$$

Тогда

$$W = p T_p - T_0,$$

$$E = W / p T_p = W / (W + T_0),$$

$$W = (E / (1 - E)) T_0 = K T_0 = K (p T_p - W).$$

В идеале решение задачи на p процессорах должно выполняться в p раз быстрее, чем на одном процессоре, или/и должно позволить решить задачу с объемами данных, в p раз большими. На самом деле такое ускорение практически никогда не достигается. Причина этого хорошо иллюстрируется законом Амдала:

$$S \le 1/(f + (1 - f)/p),$$

где S – ускорение работы программы на p процессорах, а f – доля непараллельного кода в программе.

Эта формула справедлива при программировании и в модели общей памяти, и в модели передачи сообщений. Несколько разный смысл вкладывается в понятие *доля непараллельного кода*. Для SMP систем (модель общей памяти) эту долю образуют те операторы, которые выполняет только главная нить программы. Для MPP систем (механизм передачи сообщений) непараллельная часть кода образуется за счет операторов, выполнение которых дублируется всеми процессорами. Оценить эту величину из анализа текста программы

практически невозможно. Такую оценку могут дать только реальные расчеты на различном числе процессоров. Из формулы следует, что p-кратное ускорение может быть достигнуто, только когда доля непараллельного кода равна 0. Очевидно, что добиться этого практически невозможно. Наглядно действие закона Амдала демонстрирует табл. 1.

Таблица 1

Ускорение работы программы в зависимости от доли непараллельного кода

Число процессоров	Доля последовательных вычислений, %				
	50	25	10	5	2
	Ускорение работы программы				
2	1.33	1.60	1.82	1.90	1.96
4	1.60	2.28	3.07	3.48	3.77
8	1.78	2.91	4.71	5.93	7.02
16	1.88	3.36	6.40	9.14	12.31
32	1.94	3.66	7.80	12.55	19.75
512	1.99	3.97	9.83	19.28	45.63
2048	2.00	3.99	9.96	19.82	48.83

Из таблицы хорошо видно, что если, например, доля последовательного кода составляет 2 %, то более чем 50-кратное ускорение в принципе получить невозможно. С другой стороны, по-видимому, нецелесообразно запускать такую программу на 2048 процессорах с тем, чтобы получить 49-кратное ускорение. Тем не менее такая задача достаточно эффективно будет выполняться на 16 процессорах, а в некоторых случаях потеря 37 % производительности при выполнении задачи на 32 процессорах может быть вполне приемлемой. В некотором смысле закон Амдала устанавливает предельное число процессоров, на котором программа будет выполняться с приемлемой эффективностью в зависимости от доли непараллельного кода. Заметим, что эта формула не учитывает накладные расходы на обмены между процессорами, поэтому в реальной жизни ситуация может быть еще хуже.

Не следует забывать, что распараллеливание программы – это лишь одно из средств ускорения ее работы. Не меньший эффект, а иногда и больший, может дать оптимизация однопроцессорной программы. Чрезвычайную актуальность эта проблема приобрела в последнее время из-за большого разрыва в скорости работы кэш-памяти и основной памяти. К сожалению, зачастую этой проблеме не уделяется должного внимания.

Пример. В качестве примера исследования эффективности приведем сравнение коэффициентов эффективности и ускорения последовательного [14] и параллельного алгоритма матричной прогонки [15] для МВС–1000/16.

Составлено несколько программ с помощью библиотеки MPI на языке Фортран 90: последовательный классический алгоритм матричной прогонки (ПКА), последовательный модифицированный алгоритм матричной прогонки (ПМА) и параллельный алгоритм матричной прогонки. По сравнению с ПКА время счета ПМА больше, но устойчивость к ошибкам округления выше.

При этом была написана на языке Фортран вспомогательная библиотека для работы с матрицами и векторами, содержащая операции умножения, сложения, вычитания, обращения матриц, сложения и вычитания векторов, умножения матрицы на вектор.

В табл. 2 приведены времена счета и коэффициенты ускорения и эффективности решения системы уравнений с помощью классического и модифицированного последовательных, а также параллельного алгоритма матричной прогонки для блочно–трехдиагональных матриц размерности 30000´ 30000 с квадратными блоками размерности 25.

Таблица 2

Алгоритм матричной прогонки для матриц 30000´ 30000

Число процессоров m	Время T_m, сек	Ускорение S_m	Эффективность E_m
1	62.28 / 80.76	ПКА /ПМА	ПКА / ПМА
2	39.38	1.56 / 2.0	0.78 / 1.0
3	26.43	2.32 / 3.0	0.78 / 1.0
4	19.70	3.11 / 4.0	0.78 / 1.0
5	15.62	3.92 / 5.0	0.78 / 1.0
6	13.22	4.64 / 6.0	0.77 / 1.0
10	8.08	7.71 / 10.0	0.77 / 1.0
15	5.56	11.2 / 14.5	0.75 / 0.97

Результаты вычислений показывают, что параллельный алгоритм матричной прогонки имеет высокую эффективность распараллеливания $E_m \geq 0.75$ для числа процессоров $m \leq 15$ при сравнении с ПКА и $E_m \approx 1$ при сравнении с ПМА.

При дальнейшем увеличении числа процессоров время обменов увеличивается, эффективность уменьшается.

4. Среда параллельного программирования MPI

Наиболее распространенной технологией программирования для параллельных компьютеров с распределенной памятью в настоящее время является MPI. Основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии данной технологии – Message Passing Interface (интерфейс передачи сообщений). Стандарт MPI фиксирует интерфейс, который должен соблюдаться как системой программирования на каждой вычислительной платформе, так и пользователем при создании своих программ. Коммуникационная библиотека MPI стала общепризнанным стандартом в параллельном программировании с использованием механизма передачи сообщений.

4.1. Общая организация MPI

MPI-программа представляет собой набор независимых процессов, каждый из которых выполняет свою собственную программу (не обязательно одну и ту же), написанную на языке C или FORTRAN. Появились реализации MPI для C++, однако разработчики стандарта MPI за них ответственности не несут. Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Как правило, каждый процесс выполняется в своем собственном адресном пространстве, однако допускается и режим разделения памяти. MPI не специфицирует модель выполнения процесса – это может быть как последовательный процесс, так и многопотоковый (не во всех реализациях). MPI не предоставляет никаких средств для распределения процессов по вычислительным узлам и для запуска их на исполнение. Эти функции возлагаются на разработчика конкретной реализации стандарта MPI. MPI не накладывает каких-либо ограничений на то, как процессы будут распределены по процессорам, в частности, возможен запуск MPI-программы с несколькими процессами на обычной однопроцессорной системе.

Интерфейс MPI поддерживает создание параллельных программ в стиле *MIMD (Multiple Instruction Multiple Data)*, что подразумевает объединение процессов с различными исходными текстами. Однако писать и отлаживать такие программы очень сложно, поэтому на практике программисты гораздо чаще используют *SPMD-модель (Single Program Multiple Data)* параллельного программирования, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Поскольку MPI является библиотекой, то при компиляции программы необходимо прилинковать соответствующие библиотечные модули. Это можно сделать в командной строке или воспользоваться предусмотренными в большинстве систем командами или скриптами **mpicc** (для программ на языке Си), **mpiCC** (для программ на языке Си++), **mpif77/mpif90** (для программ на языках Фортран 77/90). Опция компилятора "-o name" позволяет задать имя

name для получаемого выполнимого файла, по умолчанию выполнимый файл называется a.out, например:

mpif77 –o program program.f

После получения выполнимого файла необходимо запустить его на требуемом количестве процессоров. Для этого обычно предоставляется команда запуска MPI-приложений mpirun, например:

mpirun –np N <программа с аргументами>,

где N – число процессов, которое должно быть не более разрешенного в данной системе числа процессов для одной задачи. После запуска одна и та же программа будет выполняться всеми запущенными процессами, результат выполнения в зависимости от системы будет выдаваться на терминал или записываться в файл с определенным именем.

Для идентификации наборов процессов вводится понятие *группы*, объединяющей все или какую-то часть процессов. Каждая группа образует *область связи*, с которой связывается специальный объект – *коммуникатор* области связи. Процессы внутри группы нумеруются целым числом в диапазоне 0..groupsize-1. Все коммуникационные операции с некоторым коммуникатором будут выполняться только внутри области связи, описываемой этим коммуникатором. При инициализации MPI создается предопределенная область связи, содержащая все процессы MPI-программы, с которой связывается предопределенный коммуникатор **MPI_COMM_WORLD**. В большинстве случаев на каждом процессоре запускается один отдельный процесс, и тогда термины *процесс* и *процессор* становятся синонимами, а величина groupsize становится равной NPROCS – числу процессоров, выделенных данной задаче.

Итак, если сформулировать коротко, MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Это достаточно объемная и сложная библиотека, состоящая примерно из 130 функций, в число которых входят:

- функции инициализации и закрытия MPI-процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Все это множество функций предназначено для облегчения разработки эффективных параллельных программ. Пользователю принадлежит право самому решать, какие средства из предоставляемого арсенала использовать, а какие нет. В принципе любая параллельная программа может быть написана с использованием всего 6 MPI-функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций.

Каждая из MPI функций характеризуется способом выполнения:

1) ***локальная функция*** **–** выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций;

2) ***нелокальная функция*** – для ее завершения требуется выполнение MPI-процедуры другим процессом;

3) ***глобальная функция*** – процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи;

4) ***блокирующая функция*** – возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить;

5) ***неблокирующая функция*** – возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

Использование библиотеки MPI имеет некоторые отличия в языках C и FORTRAN.

В языке C все процедуры являются функциями, и большинство из них возвращает код ошибки. При использовании имен подпрограмм и именованных констант необходимо строго соблюдать регистр символов. Массивы индексируются с 0. Логические переменные представляются типом int (true соответствует 1, a false – 0). Определение всех именованных констант, прототипов функций и определение типов выполняется подключением файла mpi.h. Введение собственных типов в MPI было продиктовано тем обстоятельством, что стандартные типы языков на разных платформах имеют различное представление. MPI допускает возможность запуска процессов параллельной программы на компьютерах различных платформ, обеспечивая при этом автоматическое преобразование данных при пересылках. В табл. 3 приведено соответствие предопределенных в MPI типов стандартным типам языка C.

Таблица 3

Соответствие между MPI-типами и типами языка C

Тип MPI	Тип языка C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

В языке FORTRAN большинство MPI-процедур являются подпрограммами (вызываются с помощью оператора CALL), а код ошибки возвращают через дополнительный последний параметр процедуры. Несколько процедур, оформленных в виде функций, код ошибки не возвращают. Не требуется строгого соблюдения регистра символов в именах подпрограмм и именованных констант. Массивы индексируются с 1. Объекты MPI, которые в языке C являются структурами, в языке FORTRAN представляются массивами целого типа. Определение всех именованных констант и определение типов выполняется подключением файла mpif.h. В табл. 4 приведено соответствие предопределенных в MPI типов стандартным типам языка FORTRAN.

Таблица 4

Соответствие между MPI-типами и типами языка FORTRAN

Тип MPI	Тип языка FORTRAN
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	8 бит, используется для передачи нетипизированных данных
MPI_PACKED	Тип для упакованных данных

В табл. 3 и 4 перечислен обязательный минимум поддерживаемых стандартных типов, однако если в базовой системе представлены и другие типы, то их поддержку будет осуществлять и MPI (например, если в системе есть поддержка комплексных переменных двойной точности DOUBLE COMPLEX, то будет присутствовать тип MPI_DOUBLE_COMPLEX). Типы MPI_BYTE и MPI_PACKED используются для передачи двоичной информации без какого-либо преобразования. Кроме того, программисту предоставляются средства создания собственных типов на базе стандартных.

4.2. Базовые функции MPI

<table>
<tr><th colspan="2">Функция инициализации</th><th>MPI_Init</th><th colspan="2">Синтаксис</th></tr>
<tr><th colspan="3"></th><th>C</th><th>FORTRAN</th></tr>
<tr><td colspan="3">Функция инициализирует параллельные части программы. Все другие процедуры MPI могут быть вызваны только после вызова MPI _Init. Инициализация параллельной части для каждого приложения должна выполняться только один раз. В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы-приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе. Кроме этого создается предопределенный коммуникатор MPI_COMM_SELF, описывающий свою область связи для каждого отдельного процесса</td><td>int MPI_Init(int *argc, char ***argv)

Каждому процессу при инициализации передаются указатели на аргументы командной строки программы argc и argv, из которых системой могут извлекаться и передаваться в параллельные процессы некоторые параметры запуска программы</td><td>MPI_INIT(IERROR)
INTEGER IERROR

Параметр IERROR является выходным и возвращает код ошибки</td></tr>
<tr><th colspan="2">Функция завершения MPI-программ</th><th>MPI_Finalize</th><th colspan="2">Синтаксис</th></tr>
<tr><th colspan="3"></th><th>C</th><th>FORTRAN</th></tr>
<tr><td colspan="3">Функция закрывает все MPI-процессы и ликвидирует все области связи. Все последующие обращения к любым процедурам MPI, в том числе к MPI_Init, запрещены. К моменту вызова</td><td>int MPI_Finalize(void)</td><td>MPI_FINALIZE (IERROR)
INTEGER IERROR</td></tr>
</table>

MPI_Finalize каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены			
Функция определения числа процессов в области связи	***MPI_Comm_size***	**Синтаксис**	
		C	**FORTRAN**
Функция возвращает количество процессов в области связи коммуникатора comm. До создания явным образом групп и связанных с ними коммуникаторов единственно возможными значениями параметра COMM являются MPI_COMM_WORLD и MPI_COMM_SELF, которые создаются автоматически при инициализации MPI. Подпрограмма является локальной		int MPI_Comm_size (MPI_Comm comm, int *size)	MPI_COMM_SIZE (COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR IN comm – коммуникатор; OUT size – число процессов в области связи коммуникатора comm
Функция определения номера процесса	***MPI_Comm_rank***	**Синтаксис**	
		C	**FORTRAN**
Функция возвращает номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..size-1 (значение size может быть определено с помощью предыдущей функции). Подпрограмма является локальной		int MPI_Comm_rank (MPI_Comm comm, int *rank)	MPI_COMM_RANK (COMM, RANK, IERROR) INTEGER COMM, RANK, IERROR IN comm – коммуникатор; OUT rank – номер процесса, вызвавшего функцию
Функция передачи сообщения	***MPI_Send***	**Синтаксис**	
		C	**FORTRAN**
Функция выполняет посылку		int MPI_Send(void*	MPI_SEND(BUF,

count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm. Переменная buf – это, как правило, массив или скалярная переменная. В последнем случае значение count = 1	buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR IN buf – адрес начала расположения пересылаемых данных; IN count – число пересылаемых элементов; IN datatype – тип посылаемых элементов; IN dest – номер процесса-получателя в группе, связанной с коммуникатором comm; IN tag – идентификатор сообщения (аналог типа сообщения функций nread и nwrite PSE nCUBE2); IN comm – коммуникатор области связи
Функция приема сообщения ***MPI_Recv***	**Синтаксис**	
	C	**FORTRAN**
Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm	int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR OUT buf – адрес начала расположения принимаемого сообщения;

		IN count – максимальное число принимаемых элементов; IN datatype – тип элементов принимаемого сообщения; IN source – номер процесса-отправителя; IN tag – идентификатор сообщения; IN comm – коммуникатор области связи; OUT status – атрибуты принятого сообщения

Теперь рассмотрим функцию, которая не входит в очерченный нами минимум, но которая важна для разработки эффективных программ. Речь идет о функции отсчета времени – таймере. С одной стороны, такие функции имеются в составе всех операционных систем, но, с другой стороны, существует полнейший произвол в их реализации. Опыт работы с различными операционными системами показывает, что при переносе приложений с одной платформы на другую первое (а иногда и единственное), что приходится переделывать, – это обращения к функциям учета времени. Поэтому разработчики MPI, добиваясь полной независимости приложений от операционной среды, определили и свои функции отсчета времени.

Функция отсчета времени (таймер)	***MPI_Wtime***	**Синтаксис**	
		C	**FORTRAN**
Функция возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом (точки отсчета). Гарантируется, что эта точка отсчета не будет изменена в течение жизни процесса. Для хронометража участка программы вызов функции делается в начале и конце участка и определяется разница между показаниями таймера		double MPI_Wtime(void)	DOUBLE PRECISION MPI_WTIME()

Пример:
{
double starttime, endtime;
starttime = MPI_Wtime();
… хронометрируемый участок …
endtime = MPI_Wtime();
printf(“Выполнение заняло %f секунд\n”, endtime-starttime);
}		

4.3. Работа с коммуникаторами

Часто в приложениях возникает потребность ограничить область коммуникаций некоторым набором процессов, которые составляют подмножество исходного набора. Для выполнения каких-либо коллективных операций внутри этого подмножества из них должна быть сформирована своя область связи, описываемая своим коммуникатором. Для решения таких задач MPI поддерживает два взаимосвязанных механизма. Во-первых, имеется набор функций для работы с группами процессов как упорядоченными множествами и, во-вторых, набор функций для работы с коммуникаторами для создания новых коммуникаторов как описателей новых областей связи.

Группа представляет собой упорядоченное множество процессов. Каждый процесс идентифицируется переменной целого типа. Идентификаторы процессов образуют непрерывный ряд, начинающийся с 0. В MPI вводится специальный тип данных MPI_Group и набор функций для работы с переменными и константами этого типа. Существует две предопределенных группы:

MPI_GROUP_EMPTY – группа, не содержащая ни одного процесса;

MPI_GROUP_NULL – группа, возвращающая значение в случае, когда она не может быть создана.

Созданная группа не может быть модифицирована (расширена или усечена), может быть только создана новая группа. Интересно отметить, что при инициализации MPI не создается группа, соответствующая коммуникатору **MPI_COMM_WORLD**. Она должна создаваться специальной функцией явным образом.

Коммуникатор представляет собой скрытый объект с некоторым набором атрибутов, а также правилами его создания, использования и уничтожения. Коммуникатор описывает некоторую область связи. Одной и той же области связи может соответствовать несколько коммуникаторов, но даже в этом случае они не являются тождественными и не могут участвовать во взаимном обмене сообщениями. Если данные посылаются через один коммуникатор, процесс-получатель может получить их только через тот же самый коммуникатор.

В MPI существует **два типа коммуникаторов**:

intracommunicator – описывает область связи некоторой группы процессов;

intercommunicator – служит для связи между процессами двух различных групп.

Следующие коммуникаторы создаются сразу после вызова процедуры MPI_INIT:

- **MPI_COMM_WORLD** – коммуникатор, объединяющий все процессы приложения;
- **MPI_COMM_NULL** – коммуникатор, значение которого используемое для ошибочного коммуникатора;
- **MPI_COMM_SELF** – коммуникатор, включающий только вызвавший процесс.

Тип коммуникатора можно определить с помощью специальной ***функции MPI_Comm_test_inter.***

Функция определения типа коммуникатора	***MPI_Comm_ test_inter***	**Синтаксис**	
		C	**FORTRAN**
Функция возвращает значение «истина», если коммуникатор является inter коммуникатором. При инициализации MPI создается два предопределенных коммуникатора: MPI_COMM_WORLD – описывает область связи, содержащую все процессы; MPI_COMM_SELF – описывает область связи, состоящую из одного процесса		MPI_Comm_test_inter(MPI_Comm comm, int *flag)	MPI_COMM_TEST_INTER(COMM, FLAG, IERROR) INTEGER COMM, IERROR LOGICAL FLAG IN comm – коммуникатор; OUT flag – возвращает true, если comm – intercommunicator

Функции работы с коммуникаторами

В данном разделе рассматриваются функции работы с коммуникаторами. Они разделяются на функции доступа к коммуникаторам и функции создания коммуникаторов. Функции доступа являются локальными и не требуют коммуникаций, в отличие от функций создания, которые являются коллективными и могут потребовать межпроцессорных коммуникаций.

Две основные функции доступа к коммуникатору (***MPI_Comm_size*** – опрос числа процессов в области связи и ***MPI_Comm_rank*** – опрос идентификатора, или номера процесса в области связи) уже были рассмотрены среди базовых функций MPI. Кроме них, имеются нижеследующие функции:

Функция сравнения двух коммуникаторов	***MPI_Comm_ compare***	**Синтаксис**	
		C	**FORTRAN**
		MPI_Comm_compare (MPI_Comm comm1,MPI_Comm comm2, int *result)	MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) INTEGER COMM1, COMM2, RESULT, IERROR IN comm1 – первый коммуникатор; IN comm2 – второй коммуникатор; OUT result – результат сравнения. Возможные значения результата сравнения: MPI_IDENT – коммуникаторы идентичны, представляют один и тот же объект; MPI_CONGRUENT – коммуникаторы конгруэнтны, две области связи с одними и теми же атрибутами группы; MPI_SIMILAR – коммуникаторы подобны, группы содержат одни и те же процессы, но другое упорядочивание; MPI_UNEQUAL – во всех других случаях. Создание нового коммуникатора возможно с помощью одной из трех функций: MPI_Comm_dup, MPI_Comm_create, MPI_Comm_split.

Функция дублирования коммуникатора	***MPI_Comm_dup***	**Синтаксис**	
		C	**FORTRAN**
Функция создает новый коммуникатор NEWCOMM с той же группой процессов и атрибутами, что и у коммуникатора COMM		MPI_Comm_dup (MPI_Comm comm, MPI_Comm *newcomm)	MPI_COMM_DUP (COMM, NEWCOMM, IERROR) INTEGER COMM, NEWCOMM, IERROR IN comm – коммуникатор; OUT newcomm – копия коммуникатора
Функция создания коммуникатора	***MPI_Comm_create***	**Синтаксис**	
		C	**FORTRAN**
Эта функция создает коммуникатор для группы group. Для процессов, которые не являются членами группы, возвращается значение MPI_COMM_NULL. Функция возвращает код ошибки, если группа group не является подгруппой родительского коммуникатора		MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)	MPI_COMM_CREATE (COMM, GROUP, NEWCOMM, IERROR) INTEGER COMM, GROUP, NEWCOMM, IERROR IN comm – родительский коммуникатор; IN group – группа, для которой создается коммуникатор; OUT newcomm – новый коммуникатор
Функция расщепления коммуникатора	***MPI_Comm_split***	**Синтаксис**	
		C	**FORTRAN**
Разбиение коммуникатора COMM на несколько новых коммуникаторов по числу значений параметра COLOR. В один коммуникатор попадают процессы с одним значением COLOR. Процессы с б**о**льшим значением параметра KEY получат больший ранг в новой группе, при одинаковом значении параметра KEY порядок нумерации процессов выбирается системой. Процессы, которые не		MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)	MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR) INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR IN comm – родительский коммуникатор; IN color – признак подгруппы; IN key – управление упорядочиванием;

должны войти в новые коммуникаторы, указывают в качестве параметра COLOR константу MPI_UNDEFINED. Им в параметре NEWCOMM вернется значение MPI_COMM_NULL		OUT newcomm – новый коммуникатор

Приведем алгоритм расщепления группы из восьми процессов на три подгруппы:

```
MPI_comm comm, newcomm;
int myid, color;
……
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm).
```

В данном примере первую подгруппу образовали процессы, номера которых делятся на 3 без остатка, вторую – для которых остаток равен 1 и третью – для которых остаток равен 2. Отметим, что после выполнения функции **MPI_Comm_split** значения коммуникатора newcomm в процессах разных подгрупп будут отличаться.

Функция уничтожения коммуникатора	***MPI_Comm_ free***	**Синтаксис**	
		C	**FORTRAN**
		MPI_Comm_free (MPI_Comm *comm)	MPI_COMM_FREE (COMM, IERROR) INTEGER COMM, IERROR IN comm – уничтожаемый коммуникатор

4.4. Обзор коммуникационных операций типа точка-точка

К операциям этого типа относятся две представленные в предыдущем разделе коммуникационные процедуры. В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. Многообразие объясняется возможностью организации таких обменов множеством способов. Описанные в предыдущем разделе функции реализуют стандартный режим с блокировкой.

Блокирующие функции подразумевают выход из них только после полного окончания операции, т.е. вызывающий процесс блокируется, пока операция не будет завершена. Для функции посылки сообщения это означает,

что все пересылаемые данные помещены в буфер (для разных реализаций MPI это может быть либо какой-то промежуточный системный буфер, либо непосредственно буфер получателя). Для функции приема сообщения блокируется выполнение других операций, пока все данные из буфера не будут помещены в адресное пространство принимающего процесса.

Неблокирующие функции подразумевают совмещение операций обмена с другими операциями, поэтому неблокирующие функции передачи и приема по сути дела являются функциями инициализации соответствующих операций. Для опроса завершенности операции (и завершения) вводятся дополнительные функции.

Как для блокирующих, так и неблокирующих операций MPI поддерживает четыре режима выполнения. Эти режимы касаются только функций передачи данных, поэтому для блокирующих и неблокирующих операций имеется по четыре функции посылки сообщения. В таблице перечислены имена базовых коммуникационных функций типа точка-точка, имеющихся в библиотеке MPI.

Таблица 5

Список коммуникационных функций типа точка-точка

Режимы выполнения	С блокировкой	Без блокировки
Стандартная посылка	MPI_Send	MPI_Isend
Синхронная посылка	MPI_Ssend	MPI_Issend
Буферизованная посылка	MPI_Bsend	MPI_Ibsend
Согласованная посылка	MPI_Rsend	MPI_Irsend
Прием информации	MPI_Recv	MPI_Irecv

Из табл. 5 хорошо виден принцип формирования имен функций. К именам базовых функций Send/Recv добавляются различные префиксы.

1. Префикс S (synchronous) – означает синхронный режим передачи данных. Операция передачи данных заканчивается только тогда, когда заканчивается прием данных. Функция нелокальная.
2. Префикс B (buffered) – означает буферизованный режим передачи данных. В адресном пространстве передающего процесса с помощью специальной функции создается буфер обмена, который используется в операциях обмена. Операция посылки заканчивается, когда данные помещены в этот буфер. Функция имеет локальный характер.
3. Префикс R (ready) – согласованный или подготовленный режим передачи данных. Операция передачи данных начинается только тогда, когда принимающий процессор выставил признак готовности приема данных, инициировав операцию приема. Функция нелокальная.
4. Префикс I (immediate) – относится к неблокирующим операциям.

Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом. Функции передачи, находящиеся в одном столбце, имеют совершенно одинаковый синтаксис и отличаются только внутренней реализацией. Поэтому в дальнейшем будем рассматривать только стандартный режим, который в обязательном порядке поддерживают все реализации MPI.

4.4.1. Блокирующие коммуникационные операции

В стандартном режиме выполнение операции обмена включает три этапа.

1. Передающая сторона формирует пакет сообщения, в который помимо передаваемой информации упаковываются адрес отправителя (source), адрес получателя (dest), идентификатор сообщения (tag) и коммуникатор (comm). Этот пакет передается отправителем в системный буфер, и на этом функция посылки сообщения заканчивается.

2. Сообщение системными средствами передается адресату.

3. Принимающий процессор извлекает сообщение из системного буфера, когда у него появится потребность в этих данных. Содержательная часть сообщения помещается в адресное пространство принимающего процесса (параметр buf), а служебная – в параметр status.

Поскольку операция выполняется в асинхронном режиме, адресная часть принятого сообщения состоит из трех полей:

- коммуникатора (comm), т.к. каждый процесс может одновременно входить в несколько областей связи;
- номера отправителя в этой области связи (source);
- идентификатора сообщения (tag), который используется для взаимной привязки конкретной пары операций посылки и приема сообщений.

Параметр count (количество принимаемых элементов сообщения) в процедуре приема сообщения должен быть не меньше, чем длина принимаемого сообщения. При этом реально будет приниматься столько элементов, сколько находится в буфере. Такая реализация операции чтения связана с тем, что MPI допускает использование расширенных запросов:

- для идентификаторов сообщений (MPI_ANY_TAG – читать сообщение с любым идентификатором);
- для адресов отправителя (MPI_ANY_SOURCE – читать сообщение от любого отправителя).

Не допускается расширенных запросов для коммуникаторов. Расширенные запросы возможны только в операциях чтения. Интересно отметить, что таким же образом организованы операции обмена в PSE nCUBE2. В этом отражается фундаментальное свойство механизма передачи сообщений: асимметрия операций передачи и приема сообщений, связанная с тем, что инициатива в организации обмена принадлежит передающей стороне.

Таким образом, после чтения сообщения некоторые параметры могут оказаться неизвестными, а именно: число считанных элементов, идентификатор сообщения и адрес отправителя. Эту информацию можно получить с помощью параметра status. Переменные status должны быть явно объявлены в MPI-программе. В языке C status – это структура типа MPI_Status с тремя полями MPI_SOURCE, MPI_TAG, MPI_ERROR. В языке FORTRAN status – массив типа INTEGER размера MPI_STATUS_SIZE. Константы MPI_SOURCE, MPI_TAG и MPI_ERROR определяют индексы элементов. Назначение полей переменной status представлено в табл. 6.

Таблица 6

Назначение полей переменной status

Поля status	C	FORTRAN
Процесс-отправитель	status.MPI_SOURCE	status(MPI_SOURCE)
Идентификатор сообщения	status.MPI_TAG	status(MPI_TAG)
Код ошибки	status.MPI_ERROR	status(MPI_ERROR)

Как видно из таблицы, количество считанных элементов в переменную status не заносится.

Функция определения числа фактически полученных элементов сообщения	**MPI_Get_count**	**Синтаксис**	
		C	**FORTRAN**
Подпрограмма MPI_Get_count может быть вызвана либо после чтения сообщения (функциями MPI_Recv, MPI_Irecv), либо после опроса факта поступления сообщения (функциями MPI_Probe, MPI_Iprobe). Операция чтения безвозвратно уничтожает информацию в буфере приема. При этом попытка считать сообщение с параметром count меньшим, чем число элементов в буфере, приводит к потере сообщения		int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)	MPI_GET_COUNT (STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR IN status – атрибуты принятого сообщения; IN datatype – тип элементов принятого сообщения; OUT count – число полученных элементов
Функция определения параметров полученного сообщения без его чтения	**MPI_Probe**	**Синтаксис**	
		C	**FORTRAN**
Подпрограмма MPI_Probe выполняется с блокировкой, поэтому завершится она лишь тогда, когда сообщение с подходящим идентификатором и номером процесса-отправителя будет доступно для получения. Атрибуты этого сообщения возвращаются в переменной status.		int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)	MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR IN source – номер процесса-отправителя;

Следующий за MPI_Probe вызов MPI_Recv с теми же атрибутами сообщения (номером процесса-отправителя, идентификатором сообщения и коммуникатором) поместит в буфер приема именно то сообщение, наличие которого было опрошено подпрограммой MPI_Probe		IN tag – идентификатор сообщения; IN comm – коммуникатор; OUT status – атрибуты опрошенного сообщения

При использовании блокирующего режима передачи сообщений существует потенциальная опасность возникновения тупиковых ситуаций, в которых операции обмена данными блокируют друг друга. Приведем пример некорректной программы, которая будет зависать при любых условиях:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

В этом примере оба процесса (0-й и 1-й) входят в режим взаимного ожидания сообщения друг от друга. Такие тупиковые ситуации будут возникать всегда при образовании циклических цепочек блокирующих операций чтения.

Приведем вариант правильной программы:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

Другие комбинации операций SEND/RECV могут работать или не работать в зависимости от реализации MPI (буферизованный обмен или нет).

В ситуациях, когда требуется выполнить взаимный обмен данными между процессами, безопаснее использовать совмещенную операцию MPI_Sendrecv.

Функция совмещения выполнения операций передачи и приема	**MPI_Sendrecv**	**Синтаксис**	
		C	**FORTRAN**
Функция MPI_Sendrecv совмещает выполнение операций передачи и приема. Обе операции используют один и тот же коммуникатор, но идентификаторы сообщений могут различаться. Расположение в адресном пространстве процесса принимаемых и передаваемых данных не должно пересекаться. Пересылаемые данные могут быть различного типа и иметь разную длину		int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype ecvtype, int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)	MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR IN sendbuf – адрес начала расположения посылаемого сообщения; IN sendcount – число посылаемых элементов; IN sendtype – тип посылаемых элементов; IN dest – номер процесса-получателя; IN sendtag – идентификатор посылаемого сообщения; OUT recvbuf – адрес начала расположения принимаемого сообщения; IN recvcount –

		максимальное число принимаемых элементов; IN recvtype – тип элементов принимаемого сообщения; IN source – номер процесса-отправителя; IN recvtag – идентификатор принимаемого сообщения; IN comm – коммуникатор области связи; OUT status – атрибуты принятого сообщения

Функция обмена данными одного типа с замещением посылаемых данных на принимаемые	**MPI_Sendrecv_replace**	**Синтаксис**	
		C	**FORTRAN**
В данной операции посылаемые данные из массива buf замещаются принимаемыми данными. В качестве адресатов source и dest в операциях пересылки данных можно использовать специальный адрес MPI_PROC_NULL. Коммуникационные операции с таким адресом ничего не делают. Применение этого адреса бывает удобным вместо использования логических конструкций для анализа условий посылать/читать сообщение или нет. Этот прием будет использован нами далее в одном из примеров, а именно в программе решения уравнения Лапласа методом Якоби		MPI_Sendrecv_replace(void* buf, intcount, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)	MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR INOUT buf – адрес начала расположения посылаемого и принимаемого сообщения;

		IN count – число передаваемых элементов; IN datatype – тип передаваемых элементов; IN dest – номер процесса-получателя; IN sendtag – идентификатор посылаемого сообщения; IN source – номер процесса-отправителя; IN recvtag – идентификатор принимаемого сообщения; IN comm – коммуникатор области связи; OUT status – атрибуты принятого сообщения

4.4.2. Неблокирующие коммуникационные операции

Использование неблокирующих коммуникационных операций повышает безопасность с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций. Эти задачи решаются разделением коммуникационных операций на две стадии: инициирование операции и проверку завершения операции.

Неблокирующие операции используют специальный скрытый (opaque) объект «запрос обмена» (request) для связи между функциями обмена и функциями опроса их завершения. Для прикладных программ доступ к этому объекту возможен только через вызовы MPI-функций. Если операция обмена завершена, подпрограмма проверки снимает «запрос обмена», устанавливая его в значение MPI_REQUEST_NULL. Снять запрос без ожидания завершения операции можно подпрограммой MPI_Request_free.

Функция передачи сообщения без блокировки	**MPI_Isend**	**Синтаксис**	
		C	**FORTRAN**
Возврат из подпрограммы		int	MPI_ISEND(BUF,

происходит немедленно (immediate), без ожидания окончания передачи данных. Этим объясняется префикс I в именах функций. Поэтому переменную buf повторно использовать нельзя до тех пор, пока не будет погашен «запрос обмена». Это можно сделать с помощью подпрограммы MPI_Wait или MPI_Test, передав им параметр request		MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR IN buf – адрес начала расположения передаваемых данных; IN count – число посылаемых элементов; IN datatype – тип посылаемых элементов; IN dest – номер процесса-получателя; IN tag – идентификатор сообщения; IN comm – коммуникатор; OUT request – «запрос обмена»
Функция приема сообщения без блокировки	**MPI_Irecv**	**Синтаксис**	
		C	**FORTRAN**
Возврат из подпрограммы происходит немедленно, без ожидания окончания приема данных. Определить момент окончания приема можно с помощью подпрограммы MPI_Wait или MPI_Test с соответствующим параметром request		int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)	MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR OUT buf – адрес для принимаемых данных; IN count – максимальное число принимаемых элементов;

			IN datatype – тип элементов принимаемого сообщения; IN source – номер процесса-отправителя; IN tag – идентификатор сообщения; IN comm – коммуникатор; OUT request – «запрос обмена»
Неблокирующая функция чтения параметров полученного сообщения	**MPI_Iprobe**	**Синтаксис**	
		C	**FORTRAN**
Если flag=true, то операция завершилась и в переменной status находятся атрибуты этого сообщения		int MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag MPI_Status *status)	MPI_IPROBE (SOURCE, TAG, COMM, FLAG, STATUS, IERROR) LOGICAL FLAG INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR IN source – номер процесса-отправителя; IN tag – идентификатор сообщения; IN comm – коммуникатор; OUT flag – признак завершенности операции; OUT status – атрибуты опрошенного сообщения

Воспользоваться результатом неблокирующей коммуникационной операции или повторно использовать ее параметры можно только после ее полного завершения. Имеется два типа функций завершения неблокирующих операций (ожидание завершения и проверка завершения):

1) операции семейства WAIT блокируют работу процесса до полного завершения операции;

2) операции семейства TEST возвращают значения TRUE или FALSE в зависимости от того, завершилась операция или нет. Они не блокируют работу процесса и полезны для предварительного определения факта завершения операции.

Функция ожидания завершения неблокирующей операции	**MPI_Wait**	**Синтаксис**	
		C	**FORTRAN**
Это нелокальная блокирующая операция. Возврат происходит после завершения операции, связанной с запросом request. В параметре status возвращается информация о законченной операции		int MPI_Wait(MPI_Request *request, MPI_Status *status)	MPI_WAIT(REQUEST, STATUS, IERROR) INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR INOUT request – «запрос обмена»; OUT status – атрибуты сообщения
Функция проверки завершения неблокирующей операции	**MPI_Test**	**Синтаксис**	
		C	**FORTRAN**
Это локальная неблокирующая операция. Если связанная с запросом request операция завершена, возвращается flag = =true, a status содержит информацию о завершенной операции. Если проверяемая операция не завершена, возвращается flag = false, а значение status в этом случае не определено		int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)	MPI_TEST(REQUEST, FLAG, STATUS, IERROR) LOGICAL FLAG INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR INOUT request – «запрос обмена»; OUT flag – признак завершенности проверяемой операции; OUT status – атрибуты сообщения, если операция завершилась

Рассмотрим пример использования неблокирующих операций и функции MPI_Wait.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** Выполнение вычислений во время передачи сообщения ****
    CALL MPI_WAIT(request, status, ierr)
ELSE
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** Выполнение вычислений во время приема сообщения ****
    CALL MPI_WAIT(request, status, ierr)
END IF
```

Функция снятия запроса без ожидания завершения неблокирующей операции	**MPI_Request _free**	**Синтаксис**	
		C	**FORTRAN**
Параметр request устанавливается в значение MPI_REQUEST_NULL. Связанная с этим запросом операция не прерывается, однако проверить ее завершение с помощью MPI_Wait или MPI_Test уже нельзя. Для прерывания коммуникационной операции следует использовать функцию MPI_Cancel(MPI_Request *request)		int MPI_Request_free (MPI_Request *request)	MPI_REQUEST_FREE (REQUEST, IERROR) INTEGER REQUEST, IERROR INOUT request – «запрос обмена»

В MPI имеется набор подпрограмм для одновременной проверки на завершение нескольких операций (табл. 7).

Таблица 7

Функции коллективного завершения неблокирующих операций

Выполняемая проверка	Функции ожидания (блокирующие)	Функции проверки (неблокирующие)
Завершились все операции	MPI_Waitall	MPI_Testall
Завершилась по крайней мере одна операция	MPI_Waitany	MPI_Testany
Завершилась одна из списка проверяемых	MPI_Waitsome	MPI_Testsome

Кроме того, MPI позволяет для неблокирующих операций формировать целые пакеты запросов на коммуникационные операции MPI_Send_init и MPI_Recv_init, которые запускаются функциями MPI_Start или MPI_Startall.

Проверка на завершение выполнения производится обычными средствами с помощью функций семейства WAIT и TEST.

4.5. Обзор коллективных операций

Набор операций типа точка-точка является достаточным для программирования любых алгоритмов, однако MPI вряд ли завоевал бы такую популярность, если бы ограничивался только этим набором коммуникационных операций. Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективной операцией MPI_Bcast. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Набор коллективных операций включает:

- синхронизацию всех процессов с помощью барьеров (MPI_Barrier);
- коллективные коммуникационные операции, в число которых входят:
  - ✓ рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast);
  - ✓ сборка (gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv);
  - ✓ сборка (gather) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI_Allgather, MPI_Allgatherv);
  - ✓ разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI_Scatter, MPI_Scatterv);
  - ✓ совмещенная операция Scatter/Gather (All-to-All), при которой каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами, в свой буфер приема (MPI_Alltoall, MPI_Alltoallv);
- глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
  - ✓ с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
  - ✓ с рассылкой результата всем процессам (MPI_Allreduce);

- ✓ совмещенная операция Reduce/Scatter(MPI_Reduce_scatter);
- ✓ префиксная редукция (MPI_Scan).

Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:

- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- «векторный» вариант, который предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом «v» в конце имени функции.

Отличительные особенности коллективных операций:

- коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка;
- коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию;
- количество получаемых данных должно быть равно количеству посланных данных;
- типы элементов посылаемых и получаемых сообщений должны совпадать;
- сообщения не имеют идентификаторов.

Примечание. В данной главе часто будут использоваться понятия *буфер обмена, буфер передачи, буфер приема.* Не следует понимать эти понятия в буквальном смысле как некую специальную область памяти, куда помещаются данные перед вызовом коммуникационной функции. На самом деле, это, как правило, используемые в программе обычные массивы, которые непосредственно могут участвовать в коммуникационных операциях. В вызовах подпрограмм передается адрес начала непрерывной области памяти, которая будет участвовать в операции обмена.

Изучение коллективных операций начнем с рассмотрения двух функций, стоящих особняком: MPI_Barrier и MPI_Bcast.

Функция синхронизации процессов	**MPI_Barrier**	**Синтаксис**	
		C	**FORTRAN**
Блокирует работу вызвавшего ее процесса до тех пор, пока все другие процессы группы также не вызовут эту функцию. Завершение работы этой функции возможно только всеми процессами		int MPI_Barrier(MPI_ Comm comm)	MPI_BARRIER (COMM, IERROR) INTEGER COMM, IERROR IN comm – коммуникатор

одновременно (все процессы «преодолевают барьер» одновременно)		

Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован. Неявную синхронизацию процессов выполняет любая коллективная функция.

Функция широковещательной рассылки данных	**MPI_Bcast**	**Синтаксис**	
		C	**FORTRAN**
Процесс с номером root рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm. После завершения подпрограммы каждый процесс в области связи коммуникатора comm, включая и самого отправителя, получит копию сообщения от процесса-отправителя root		int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)	MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR INOUT buffer – адрес начала расположения в памяти рассылаемых данных; IN count – число посылаемых элементов; IN datatype – тип посылаемых элементов; IN root – номер процесса-отправителя; IN comm – коммуникатор

На рис. 10 представлена графическая интерпретация операции Bcast.

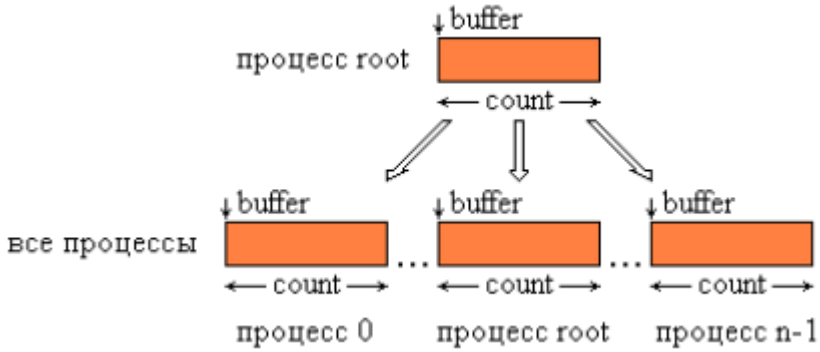


Рис. 10. Графическая интерпретация операции Bcast

Пример использования функции MPI_Bcast:

```
    IF ( MYID .EQ. 0 ) THEN
    PRINT *, 'ВВЕДИТЕ ПАРАМЕТР N : '
    READ *, N
    END IF
    CALL MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD,
IERR)
```

4.5.1. Функции сбора блоков данных от всех процессов группы

Семейство функций сбора блоков данных от всех процессов группы состоит из четырех подпрограмм: MPI_Gather, MPI_Allgather, MPI_Gatherv, MPI_Allgatherv. Каждая из указанных подпрограмм расширяет функциональные возможности предыдущих.

Функция сборки блоков данных	**MPI_Gather**	**Синтаксис**	
		C	**FORTRAN**
Производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения. Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых элементов, а число sendcount		int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	MPI_GATHER(SEND BUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR IN sendbuf – адрес начала размещения посылаемых данных; IN sendcount – число

должно равняться числу recvcount. То есть, recvcount в вызове из процесса root – это число собираемых от каждого процесса элементов, а не общее количество собранных элементов		посылаемых элементов; IN sendtype – тип посылаемых элементов; OUT recvbuf – адрес начала буфера приема (используется только в процессе-получателе root); IN recvcount – число элементов, получаемых от каждого процесса (используется только в процессе-получателе root); IN recvtype – тип получаемых элементов; IN root – номер процесса-получателя; IN comm – коммуникатор

Графическая интерпретация операции Gather представлена на рис. 11.

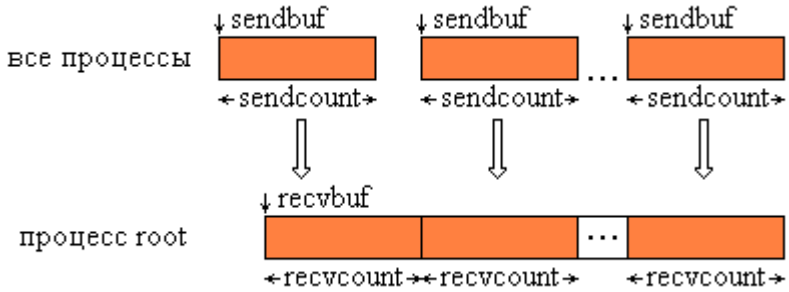


Рис. 11. Графическая интерпретация операции Gather

Пример программы с использованием функции MPI_Gather:

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

Функция сборки	**MPI_Allgather**	**Синтаксис**	
		C	**FORTRAN**
Выполняется так же, как MPI_Gather, но получателями		int MPI_Allgather(void	MPI_ALLGATHER (SENDBUF,

являются все процессы группы. Данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf каждого процесса. После завершения операции содержимое буферов приема recvbuf у всех процессов одинаково	* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)	SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR IN sendbuf – адрес начала буфера посылки; IN sendcount – число посылаемых элементов; IN sendtype – тип посылаемых элементов; OUT recvbuf – адрес начала буфера приема; IN recvcount – число элементов, получаемых от каждого процесса; IN recvtype – тип получаемых элементов; IN comm – коммуникатор

Графическая интерпретация операции Allgather представлена на рис. 12. На этой схеме ось Y образуют процессы группы, а ось X – блоки данных.

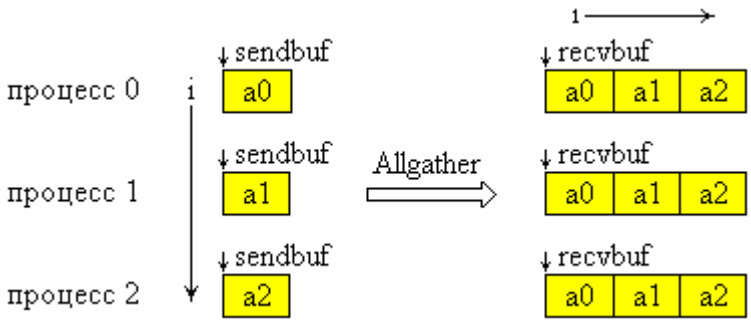


Рис. 12. Графическая интерпретация операции Allgather

Функция сборки	MPI_Gatherv	Синтаксис	
		C	FORTRAN
Функция MPI_Gatherv позволяет собирать блоки с разным числом элементов от каждого процесса, так как количество элементов, принимаемых от каждого процесса, задается индивидуально с помощью массива recvcounts. Эта функция обеспечивает также большую гибкость при размещении данных в процессе-получателе, благодаря введению в качестве параметра массива смещений displs. Сообщения помещаются в буфер приема процесса root в соответствии с номерами посылающих процессов, а именно: данные, посланные процессом i, размещаются в адресном пространстве процесса root, начиная с адреса rbuf + displs[i]		int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* rbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)	MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RBUF, RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, COMM, IERROR IN sendbuf – адрес начала буфера передачи; IN sendcount – число посылаемых элементов; IN sendtype – тип посылаемых элементов; OUT rbuf – адрес начала буфера приема; IN recvcounts – целочисленный массив (размер равен числу процессов в группе), i-й элемент массива определяет число элементов, которое должно быть получено от процесса i; IN displs – целочисленный массив (размер равен числу процессов в группе), i-е значение определяет смещение

		i-го блока данных относительно начала rbuf; IN recvtype – тип получаемых элементов; IN root – номер процесса-получателя; IN comm – коммуникатор

Графическая интерпретация операции Gatherv представлена на рис. 13.

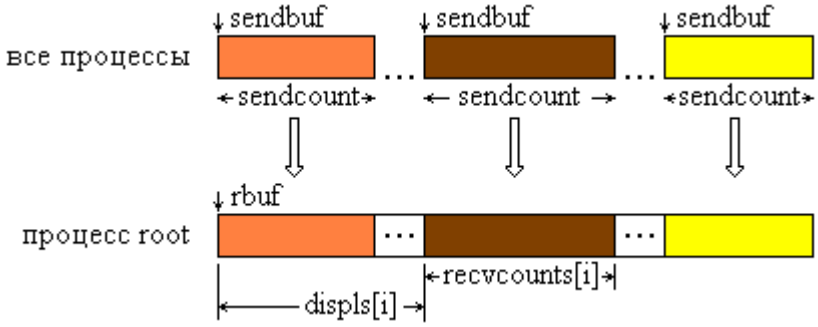


Рис. 13. Графическая интерпретация операции Gatherv

Функция сборки	**MPI_Allgatherv**	**Синтаксис**	
		C	**FORTRAN**
Функция MPI_Allgatherv является аналогом функции MPI_Gatherv, но сборка выполняется всеми процессами группы, поэтому в списке параметров отсутствует параметр root		int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* rbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)	MPI_ALLGATHERV (SENDBUF, SENDCOUNT, SENDTYPE, RBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, IERROR IN sendbuf – адрес начала буфера передачи; IN sendcount – число посылаемых элементов; IN sendtype – тип посылаемых элементов;

		OUT rbuf – адрес начала буфера приема; IN recvcounts – целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, которое должно быть получено от каждого процесса; IN displs – целочисленный массив (размер равен числу процессов в группе), i-е значение определяет смещение относительно начала rbuf i-го блока данных; IN recvtype – тип получаемых элементов; IN comm – коммуникатор

4.5.2. Функции распределения блоков данных по всем процессам группы

Семейство функций распределения блоков данных по всем процессам группы состоит из двух подпрограмм: MPI_Scatter и MPI_Scatterv.

Функция распределения	**MPI_Scatter**	**Синтаксис**	
		C	**FORTRAN**
Функция MPI_Scatter разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе). Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммуникатором comm являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, несущественны.		int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR IN sendbuf – адрес начала размещения

Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых элементов, а число посылаемых элементов sendcount должно равняться числу принимаемых recvcount. Следует обратить внимание, что значение sendcount в вызове из процесса root – это число посылаемых каждому процессу элементов, а не общее их количество. Операция Scatter является обратной по отношению к Gather		блоков распределяемых данных (используется только в процессе-отправителе root); IN sendcount – число элементов, посылаемых каждому процессу; IN sendtype – тип посылаемых элементов; OUT recvbuf – адрес начала буфера приема; IN recvcount – число получаемых элементов; IN recvtype – тип получаемых элементов; IN root – номер процесса-отправителя; IN comm – коммуникатор

На рис. 14 представлена графическая интерпретация операции Scatter.

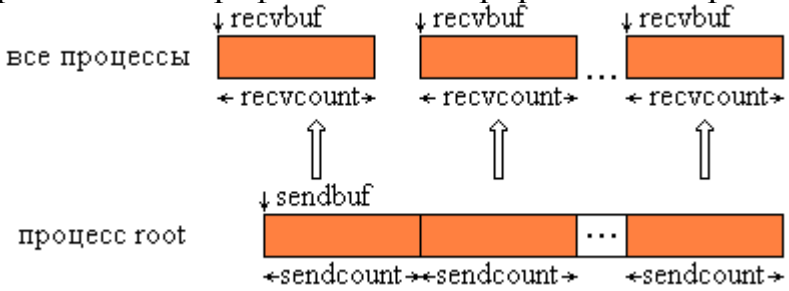


Рис. 14. Графическая интерпретация операции Scatter

Пример использования функции MPI_Scatter:

```
MPI_Comm comm;
int rbuf[100], gsize;
int root, *array;
… …
MPI_Comm_size(comm, &gsize);
array = (int *) malloc( gsize * 100 * sizeof (int));
MPI_Scatter(array, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm).
```

Функция распределения	MPI_Scatterv	**Синтаксис**	
		C	**FORTRAN**
Является векторным вариантом функции MPI_Scatter, позволяющим посылать каждому процессу различное количество элементов. Начало расположения элементов блока, посылаемого i-му процессу, задается в массиве смещений displs, а число посылаемых элементов – в массиве sendcounts. Эта функция является обратной по отношению к функции MPI_Gatherv		int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	MPI_SCATTERV (SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR IN sendbuf – адрес начала буфера посылки (используется только в процессе-отправителе root); IN sendcounts – целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, посылаемых каждому процессу; IN displs – целочисленный массив (размер равен числу процессов в группе), i-е значение определяет смещение относительно начала sendbuf для данных, посылаемых процессу i; IN sendtype – тип

		посылаемых элементов; OUT recvbuf – адрес начала буфера приема; IN recvcount – число получаемых элементов; IN recvtype – тип получаемых элементов; IN root – номер процесса-отправителя; IN comm – коммуникатор

На рис. 15 представлена графическая интерпретация операции Scatterv.

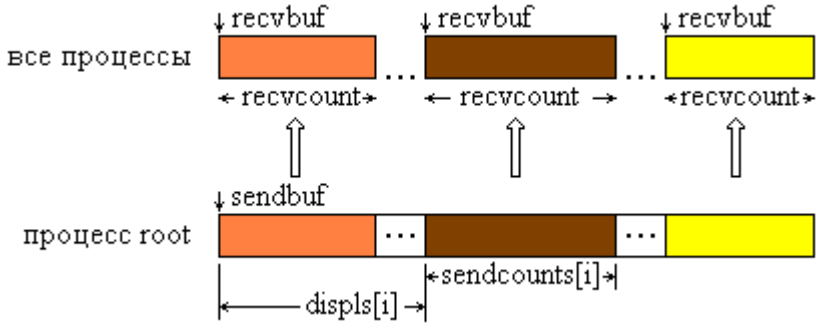


Рис. 15. Графическая интерпретация операции Scatterv

4.5.3. Совмещенные коллективные операции

Совмещенные операции	MPI_Alltoall	**Синтаксис**	
		C	**FORTRAN**
Совмещает в себе операции Scatter и Gather и является по сути дела расширением операции Allgather, когда каждый процесс посылает различные данные разным получателям. Процесс i посылает j-й блок своего буфера sendbuf процессу j, который помещает его в i-й блок своего буфера recvbuf. Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов		int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)	MPI_ALLTOALL (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE,

		RECVCOUNT, RECVTYPE, COMM, IERROR IN sendbuf – адрес начала буфера посылки; IN sendcount – число посылаемых элементов; IN sendtype – тип посылаемых элементов; OUT recvbuf – адрес начала буфера приема; IN recvcount – число элементов, получаемых от каждого процесса; IN recvtype – тип получаемых элементов; IN comm – коммуникатор

Графическая интерпретация операции Alltoall представлена на рис. 16.

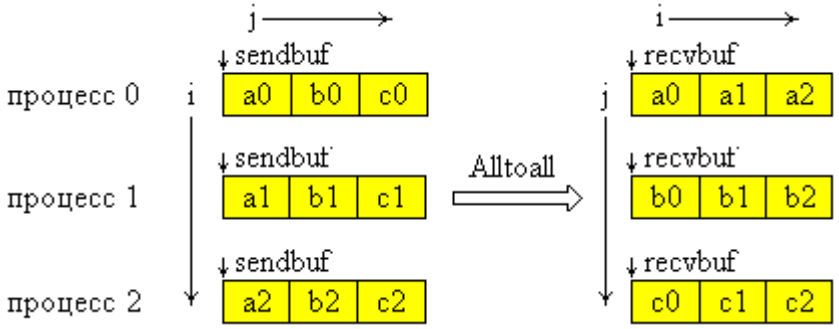


Рис. 16. Графическая интерпретация операции Alltoall

Функция MPI_Alltoallv реализует векторный вариант операции Alltoall, допускающий передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных.

4.5.4. Глобальные вычислительные операции над распределенными данными

В параллельном программировании математические операции над блоками данных, распределенных по процессорам, называют глобальными операциями редукции. В общем случае операцией редукции называется

операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора. В частности, если компоненты вектора расположены в адресных пространствах процессов, выполняющихся на различных процессорах, то в этом случае говорят о глобальной (параллельной) редукции. Например, пусть в адресном пространстве всех процессов некоторой группы имеются копии переменной var (необязательно имеющие одно и то же значение), тогда применение к ней операции вычисления глобальной суммы или, другими словами, операции редукции SUM возвратит одно значение, которое будет содержать сумму всех локальных значений этой переменной. Использование операций редукции является одним из основных средств организации распределенных вычислений.

В MPI глобальные операции редукции представлены в нескольких вариантах:

- с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
- с сохранением результата в адресном пространстве всех процессов (MPI_Allreduce);
- префиксная операция редукции, которая в качестве результата операции возвращает вектор. i-я компонента этого вектора является результатом редукции первых *i* компонент распределенного вектора (MPI_Scan);
- совмещенная операция Reduce/Scatter (MPI_Reduce_scatter).

Совмещенные операции	**MPI_Reduce**	**Синтаксис**	
		C	**FORTRAN**
Операция глобальной редукции, указанная параметром op, выполняется над первыми элементами входного буфера, и результат посылается в первый элемент буфера приема процесса root. Затем то же самое делается для вторых элементов буфера и т.д.		int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR IN sendbuf – адрес начала входного буфера; OUT recvbuf – адрес начала буфера результатов (используется только в процессе-

		получателе root); IN count – число элементов во входном буфере; IN datatype – тип элементов во входном буфере; IN op – операция, по которой выполняется редукция; IN root – номер процесса-получателя результата операции; IN comm – коммуникатор

На рис. 17 представлена графическая интерпретация операции Reduce. На данной схеме операция ‘+’ означает любую допустимую операцию редукции.

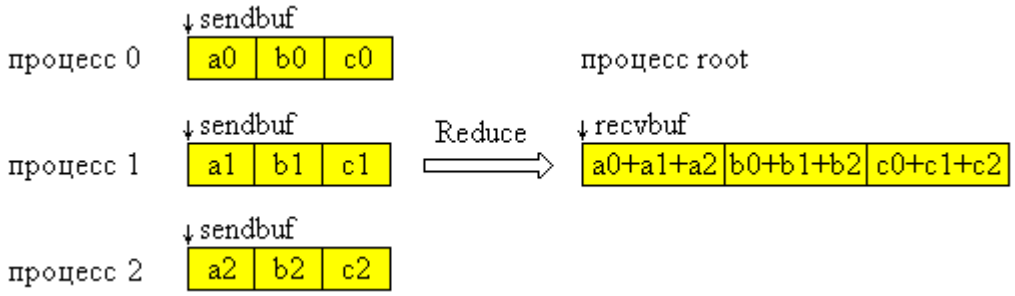


Рис. 17. Графическая интерпретация операции Reduce

В качестве операции op можно использовать либо одну из предопределенных операций, либо операцию, сконструированную пользователем. Все предопределенные операции являются ассоциативными и коммутативными. Сконструированная пользователем операция, по крайней мере, должна быть ассоциативной. Порядок редукции определяется номерами процессов в группе. Тип datatype элементов должен быть совместим с операцией op. В табл. 8 представлен перечень предопределенных операций, которые могут быть использованы в функциях редукции MPI.

Таблица 8

Предопределенные операции в функциях редукции MPI

Название	Операция	Разрешенные типы
MPI_MAX MPI_MIN	Максимум Минимум	C integer, FORTRAN integer, Floating point
MPI_SUM MPI_PROD	Сумма Произведение	C integer, FORTRAN integer, Floating point, Complex
MPI_LAND MPI_LOR MPI_LXOR	Логическое AND Логическое OR Логическое исключающее OR	C integer, Logical
MPI_BAND MPI_BOR MPI_BXOR	Поразрядное AND Поразрядное OR Поразрядное исключающее OR	C integer, FORTRAN integer, Byte
MPI_MAXLOC MPI_MINLOC	Максимальное значение и его индекс Минимальное значение и его индекс	Специальные типы для этих функций

В таблице используются следующие обозначения:
C integer: MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG
FORTRAN integer: MPI_INTEGER
Floating point: MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
Logical: MPI_LOGICAL
Complex: MPI_COMPLEX
Byte: MPI_BYTE

Операции MAXLOC и MINLOC выполняются над специальными парными типами, каждый элемент которых хранит две величины: значения, по которым ищется максимум или минимум, и индекс элемента. В MPI имеется 9 таких предопределенных типов.

C:

MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	int and int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

FORTRAN:

MPI_2REAL	REAL and REAL
MPI_2DOUBLE_PRECISION	DOUBLE PRECISION and DOUBLE PRECISION
MPI_2INTEGER	INTEGER and INTEGER

Совмещенные операции	MPI_Allreduce	**Синтаксис**	
		C	**FORTRAN**
Сохраняет результат редукции в адресном пространстве всех процессов, поэтому в списке параметров функции отсутствует идентификатор корневого процесса root. В остальном набор параметров такой же, как и в предыдущей функции		int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR IN sendbuf – адрес начала входного буфера; OUT recvbuf – адрес начала буфера приема; IN count – число элементов во входном буфере; IN datatype – тип элементов во входном буфере; IN op – операция, по которой выполняется редукция; IN comm – коммуникатор

На рис. 18 представлена графическая интерпретация операции Allreduce.

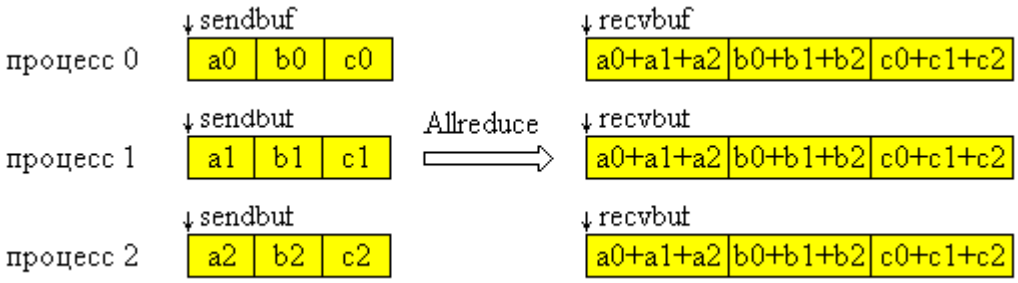


Рис. 18. Графическая интерпретация операции Allreduce

Совме-щенные операции	MPI_Reduce_ scatter	Синтаксис	
		C	FORTRAN
Совмещает в себе операции редукции и распределения результата по процессам. Функция MPI_Reduce_scatter отличается от MPI_Allreduce тем, что результат операции разрезается на непересекающиеся части по числу процессов в группе, i-я часть посылается i-му процессу в его буфер приема. Длины этих частей задает третий параметр, являющийся массивом		MPI_Reduce_scatter (void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR IN sendbuf – адрес начала входного буфера; OUT recvbuf – адрес начала буфера приема; IN revcount – массив, в котором задаются размеры блоков, посылаемых процессам; IN datatype – тип элементов во входном буфере; IN op – операция, по которой выполняется редукция; IN comm – коммуникатор

На рис. 19 представлена графическая интерпретация операции Reduce_scatter.

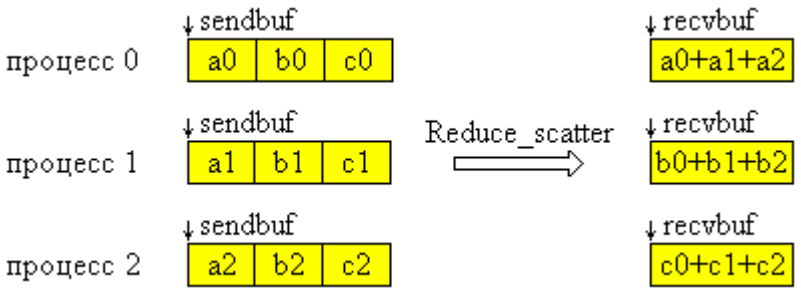


Рис. 19. Графическая интерпретация операции Reduce_scatter

Совмещенные операции	**MPI_Scan**	**Синтаксис**	
		C	**FORTRAN**
Выполняет префиксную редукцию. Параметры такие же, как в MPI_Allreduce, но получаемые каждым процессом результаты отличаются друг от друга. Операция пересылает в буфер приема i-го процесса редукцию значений из входных буферов процессов с номерами 0…i включительно		int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	MPI_MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR IN sendbuf – адрес начала входного буфера; OUT recvbuf – адрес начала буфера приема; IN count – число элементов во входном буфере; IN datatype – тип элементов во входном буфере; IN op – операция, по которой выполняется редукция; IN comm – коммуникатор

На рис. 20 представлена графическая интерпретация операции Scan.

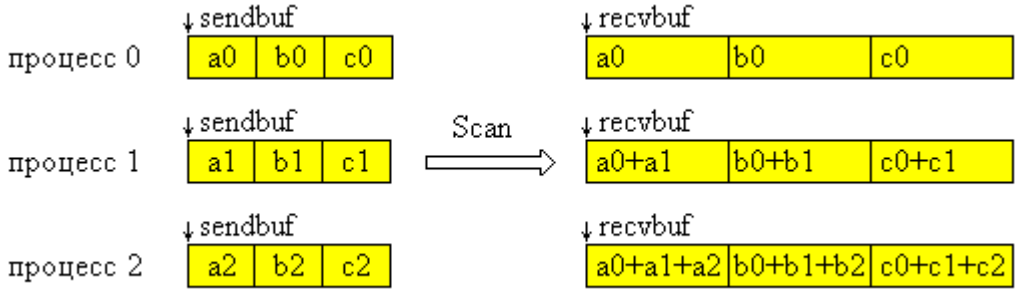


Рис. 20. Графическая интерпретация операции Scan

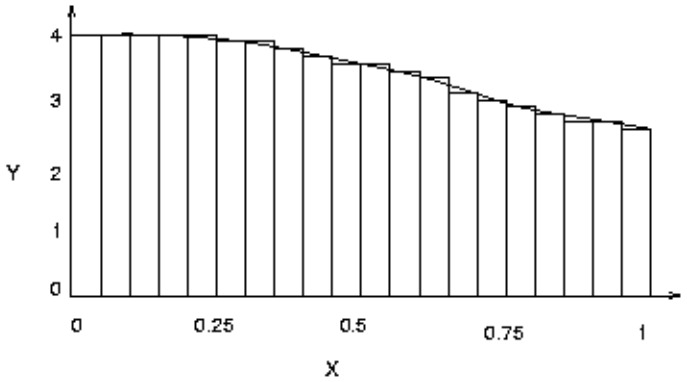
5. Примеры параллельных программ на языке Фортран

5.1. Параллельное вычисление числа π

Значение числа π может быть получено при помощи интеграла

$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx.$$

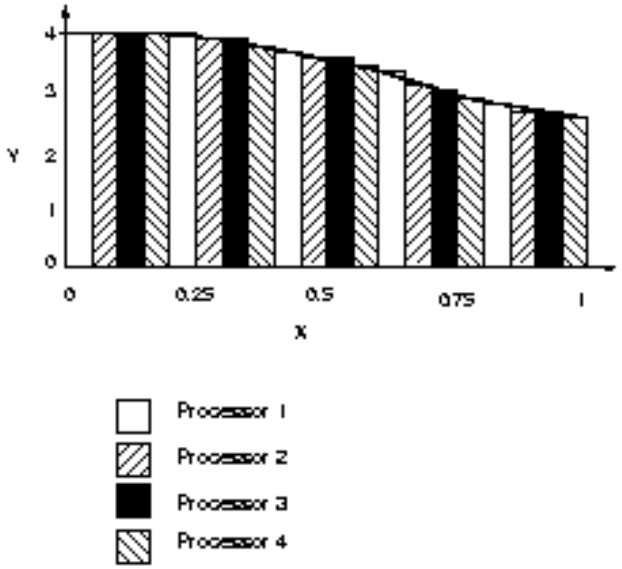
Для численного интегрирования используется метод прямоугольников:



$$\pi = \sum_{i=1}^{n} \frac{4}{1+x_i^2} \cdot \frac{1}{n},$$

где n – число точек разбиения отрезка $[0,1]$; x_i – координаты точек сетки.

Распределим вычисления между m процессорами (как показано на рисунке) и просуммируем получаемые на отдельных процессорах частные суммы.



В табл. 9 приведены времена счета и коэффициенты ускорения и эффективности вычисления числа π при $n = 1000000000$ для МВС–1000/32.

Таблица 9

Вычисление числа π

Число процессоров m	Время T_m, сек	Ускорение S_m	Эффективность E_m
1	15,93	–	–
2	7,96	1,999	0,999
3	5,31	2,998	0,999
4	3,99	3,991	0,997
5	3,19	4,999	0,999
6	2,66	5,987	0,997
7	2,28	6,985	0,997
8	1,99	7,976	0,998
9	1,87	8,508	0,945
10	1,60	9,979	0,997
11	1,46	10,906	0,991
12	1,44	11,061	0,921

Ниже приводится **пример программы** вычисления числа π.

```
 program main

include 'mpif.h'

double precision  PI25DT
parameter       (PI25DT = 3.14159265358979323846264338327950288d0)

double precision  mypi, pi, h, sum, x, f, a
double precision  t1,t2
integer n, myid, numprocs, i, rc

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
print *, 'Process ', myid, ' of ', numprocs, ' is alive'

t1 = MPI_WTime()

if ( myid .eq. 0 ) then
 n=1000000000
```

```
      endif

      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)


c     calculate the interval size
      h = 1.0d0/n

      sum  = 0.0d0

      do 20 i = myid+1, n, numprocs
         x = h * (i - 0.5d0)
         x= 4.d0 / (1.d0 + x*x)
         sum = sum + x
20    continue
      mypi = h * sum

c     collect all the partial sums

      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $     MPI_COMM_WORLD,ierr)

      t2 = MPI_WTime()
c                             node 0 prints the answer.
      if (myid .eq. 0) then
         write(*, 97) pi, abs(pi - PI25DT)
 97      format('  pi is approximately: ', F18.16,
     +          '  Error is: ', F18.16)

         t=t2-t1
         write(*,*)'time =',t,'sec'

      endif


 30   call MPI_FINALIZE(rc)
      stop
      end
```

5.2. О распараллеливании итерационных методов. Параллельное умножение матрицы на вектор

Итерационные методы решения систем линейных алгебраических уравнений (СЛАУ) вида $Az = b$, где A - матрица размерности $n \times n$; z и b - векторы размерности n, являются методами последовательных приближений. В них необходимо задать некоторое приближенное решение - начальное приближение. После этого проводится цикл вычислений – итерация, в результате которой находят новое приближение. Итерации проводятся до получения решения с требуемой точностью. Примером итерационного метода является метод наискорейшего спуска [16]

$$z^{k+1} = z^k - \frac{\left\| A^T A z^k - A^T b \right\|^2}{\left\| A(A^T A z^k - A^T b) \right\|^2} A^T (A z^k - b), \qquad z^0 = 0. \tag{1}$$

Условием останова итерационных процессов является

$$\frac{\left\| A z^k - b \right\|}{\left\| b \right\|} < \varepsilon .$$

Итерационные методы решения СЛАУ сводятся к операциям над матрицами и векторами: сложению и вычитанию матриц и векторов, умножению матриц, умножению матрицы на вектор.

Распараллеливание итерационных методов основано на разбиении матрицы A горизонтальными полосами на m блоков, а вектора решения z и вектора правой части b – на m частей так, что $n = m \times L$, где n - размерность системы уравнений; m - число процессоров, L - число строк матрицы в блоке. На каждой итерации каждый из m процессоров вычисляет свою часть вектора решения. Host- процессор отвечает за пересылки данных и также вычисляет свою часть вектора решения.

В случае матричного умножения $A^T A$ каждый из m процессоров умножает свою часть строк транспонированной матрицы A^T на всю матрицу A. Host-процессор отвечает за пересылки данных и также вычисляет свою часть вектора решения.

В случае умножения матрицы на вектор каждый из m процессоров умножает свою часть строк матрицы A на вектор z, результирующий вектор собирается на Host- процессоре.

Host – processor
1 – processor
2 – processor
…
m – processor

В табл. 10 приведены времена счета и коэффициенты ускорения и эффективности умножения заполненной матрицы размерности 10000´ 10000 на вектор для МВС–1000/32. Эффективность распараллеливания умножения матрицы на вектор меняется в пределах $0.54 \le E_m \le 0.57$. Вектор b собирается на одном процессоре, и эта операция передачи данных снижает эффективность распараллеливания.

Таблица 10

Умножение матрицы на вектор

Число процессоров m	Время T_m, сек	Ускорение S_m	Эффективность E_m
1	19,38	–	–
2	16,93	1,14	0,57
3	11,30	1,71	0,57
4	8,71	2,22	0,55
5	6,78	2,85	0,57
6	5,65	3,43	0,57
7	5,11	3,79	0,54
8	4,21	4,60	0,57
9	3,91	4,95	0,55
10	3,57	5,42	0,54

Ниже приводится **пример программы** параллельного умножения матрицы на вектор.

```
 program matrix_v
include 'mpif.h'
integer ierr, rank, i,j, size, n, h
parameter (n = 10000)
double precision time_start, time_finish
double precision a(n,n), b(n), buf(n), c(n)
integer status(MPI_STATUS_SIZE)
call MPI_INIT(ierr)

call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

h = n/size

OPEN(1,FILE = 'out')
```

```
  do i = 1, n
     do j = 1 , n
           a(i,j) = 1.
      end do
  end do

  do i = 1, n
     a(i,i) = i
    b(i) = 1.
     c(i) = 0.
     buf(i) = 0.
  end do

     call MPI_BARRIER(MPI_COMM_WORLD,ierr)
     if(rank = 0)
     time_start = MPI_WTIME(ierr)
     endif

    do i = rank*h+1, h+rank*h
    do j = 1, n
        c(i) = c(i) + a(i,j)*b(j)
     end do
  end do

  call MPI_BARRIER(MPI_COMM_WORLD,ierr)
 call MPI_GATHER(c(rank*h+1), h, MPI_DOUBLE_PRECISION,
buf(1), h, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)

  if(rank == 0) then
     time_finish = MPI_WTIME(ierr) - time_start
     write(2,*) "time = ",time_finish,"size = ",size
     do i = 1,n
     write(2,*) buf(i)
     end do
  endif

  call MPI_FINALIZE(ierr)
  end
```

6. Виды курсовых работ для студентов

6.1. Решение краевой задачи Дирихле для уравнения Пуассона

Рассматривается задача Дирихле для уравнения Пуассона:

$$\Delta u(x,y) = f(x,y), \qquad u(x,y)|_{\partial D} = g(x,y). \tag{2}$$

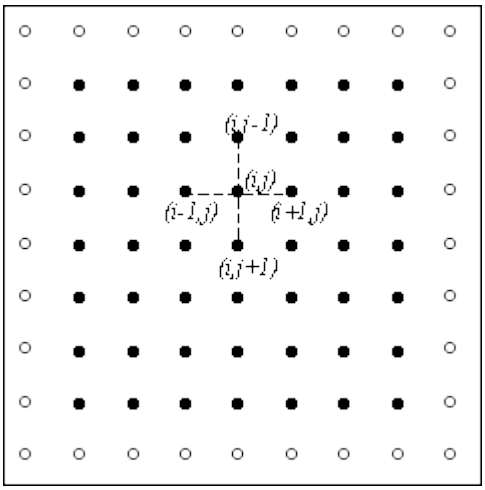
Применяя метод конечных разностей с использованием пятиточечного шаблона на сетке $N \times N$, реализовать на МВС–1000 параллельный итерационный процесс Гаусса-Зейделя для уравнения Пуассона в конечно-разностной форме:

$$u_{ij}^{k+1} = 0.25(u_{i-1,j}^{k+1} + u_{i+1,j}^{k} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k} - h^2 f_{ij}); \quad i, j = 1, \ldots, N-1 \tag{3}$$

или параллельный метод верхней релаксации по схеме

$$u_{ij}^{k+1} = \frac{\omega}{4}(u_{i-1,j}^{k+1} + u_{i+1,j}^{k} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k} - h^2 f_{ij}) + (1-\omega)u^{k}{}_{ij}; \quad i, j = 1, \ldots, N-1, \tag{4}$$

где $\omega -$ параметр верхней релаксации, например ω=1/2 или ω=1/4 (при $\omega = 1$ получаем метод Гаусса-Зейделя).



В качестве области задания D функции $u(x,y)$ будем использовать единичный квадрат

$$D = \{(x,y) \in D:\ 0 \le x, y \le 1\}.$$

Граничные условия и правая часть, например, задаются в виде

$$f(x,y) = 4 + 2x^2 - 2x + 2y^2 - 2y, \quad (x,y) \in D;$$

$$u(x,y)|_{\partial D} = y^2 - y, \quad x = 0, \quad x = 1;$$

$$u(x,y)|_{\partial D} = x^2 - x, \quad y = 0, \quad y = 1.$$

В качестве начального приближения u_{ij} будем использовать значения, сгенерированные датчиком случайных чисел из диапазона $[-1,1]$.

Итерационный процесс (2) продолжается до достижения заданной точности, а именно

$$\max_{ij=1,\ldots,N} \left\{ \left| u_{ij}^{k+1} - u_{ij}^{k} \right| \right\} < \varepsilon, \qquad \varepsilon = 0.01, \quad N = 1000.$$

Точное решение задачи имеет вид

$$u(x,y) = (x^2 - x + 1)(y^2 - y + 1).$$

После решения задачи необходимо построить функцию u_{ij} на сетке с помощью программы SURFER и сравнить приближенное решение с точным.

Для построения параллельного алгоритма решения задачи необходимо провести разбиение исходной расчетной области на подобласти, количество которых соответствует количеству задействованных в расчете процессоров.

Для решения задачи в каждой области необходимы граничные условия. При этом, учитывая шаблон разностной схемы, потребуется организация передачи значений приближений решения в узлах, являющихся границами организованных подобластей.

Разбиение происходит не по строкам, а по столбцам в силу особенностей хранения массивов в памяти при использовании языка Fortran. Значения функции на границах должны быть получены из результатов счета «соседних» процессоров. Таким образом, каждый процессор, кроме первого и последнего, должен постоянно (на каждой итерации) обмениваться данными с двумя «соседями», а крайние процессоры – только с одним. Счет не должен продолжаться, пока не получены значения функции на границах. Одновременная отправка и получение данных обеспечивается функцией библиотеки MPI mpi_sendrecv, с помощью которой синхронно происходит обмен данными. Если при вызове этой функции в качестве номера процессора-получателя или процессора-отправителя указан специальный номер mpi_proc_null, соответственно получение или отправка данных не производится – данная возможность используется крайними процессорами. Кроме того, на каждой итерации необходимо проверять, не достигнута ли нужная точность. Это возможно благодаря функции mpi_allreduce, которая собирает данные (в данном случае – значение невязки) со всех процессоров и производит над ними какую-либо операцию (в данном случае – сложение).

Заметим, что метод верхней релаксации представляет собой рекуррентные вычислительные формулы, затрудняющие его распараллеливание. Тем не менее метод все же удается распараллелить. Для этого разобьем все узлы в шахматном порядке на красные и черные, как показано на рис. 21 (в литературе эта процедура носит название красно-черного упорядочения). Нетрудно заметить, что для каждого цвета узлов расчеты будут проводиться независимо. На первом этапе будем вычислять значения сеточной функции $u_{i,j}$ в красных узлах, используя значения в черных узлах на предыдущей итерации:

$$u_{R,i,j}^{k+1} = \frac{\omega}{4}\left(u_{B,i-1,j}^{k} + u_{B,i+1,j}^{k} + u_{B,i,j-1}^{k} + u_{B,i,j+1}^{k}\right) + (1-\omega)u_{R,i,j}^{k} \ .$$

На втором этапе можно рассчитать значения $u_{i,j}$ в черных узлах, используя значения в красных узлах, но уже новые, вычисленные на первом этапе:

$$u_{B,i,j}^{k+1} = \frac{\omega}{4}\left(u_{R,i-1,j}^{k} + u_{R,i+1,j}^{k} + u_{R,i,j-1}^{k} + u_{R,i,j+1}^{k}\right) + (1-\omega)u_{B,i,j}^{k} \ .$$

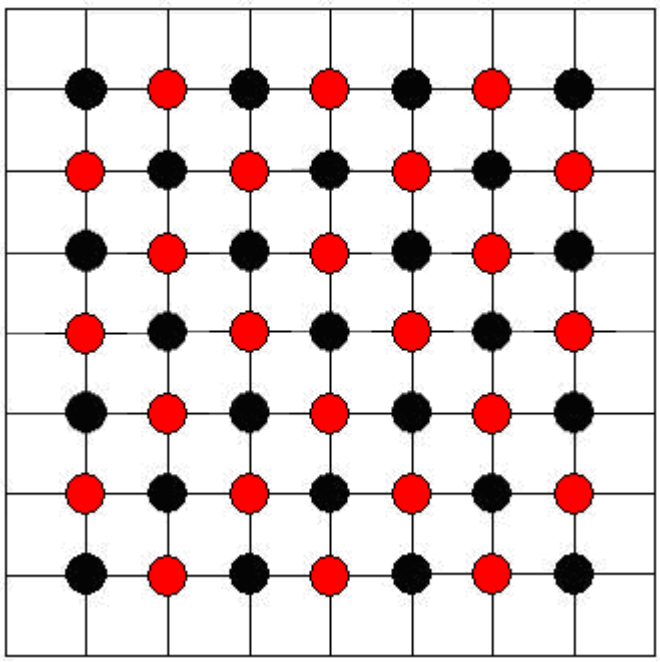


Рис. 21. Шаблон красно-черного разбиения

6.2. Решение обратной задачи гравиметрии

Рассматривается обратная задача гравиметрии о нахождении переменной плотности в горизонтальном и криволинейном слое $\Pi = \{(x, y, z) : H_1 \le z \le H_2\}$ по гравитационным данным, измеренным на площади земной поверхности. Здесь H_1, H_2 - либо константы (горизонтальный слой), либо функции от x, y (криволинейный слой). Используется априорная информация об отсутствии аномалий плотности вне слоя [17]. Предполагается, что распределение плотности $\sigma(x, y)$ внутри слоя не зависит от z (ось z направлена вниз).

Задача нахождения неизвестной плотности $\sigma(x, y)$ сводится к решению линейного двумерного интегрального уравнения Фредгольма первого рода

$$A\sigma \equiv f \int_a^b \int_c^d \left\{ \frac{1}{\left[\left(x - x'\right)^2 + \left(y - y'\right)^2 + H_1^{\,2} \right]^{1/2}} - \frac{1}{\left[\left(x - x'\right)^2 + \left(y - y'\right)^2 + H_2^{\,2} \right]^{1/2}} \right\} \sigma(x', y') dx' dy' = \Delta g(x, y), \tag{5}$$

где f - гравитационная постоянная; $\Delta g(x, y)$ - гравитационный эффект, порождаемый источниками в горизонтальном или криволинейном слое.

Некоторые необходимые понятия

Задача называется корректно поставленной, если для любых значений исходных данных ее решение существует, единственно и устойчиво по исходным данным.

Задача называется устойчивой по исходным данным, если малое изменение исходных данных приводит к малому изменению решения. Отсутствие устойчивости означает, что даже незначительные погрешности в исходных данных приводят к большим погрешностям в решении или к неверному результату (чувствительность к погрешностям исходных данных).

СЛАУ неустойчива, если ее решение сильно изменяется при малом изменении как коэффициентов, так и свободных членов.

Задача (5) является некорректно поставленной. В настоящее время развиты методы решения некорректных задач – методы регуляризации [18],[19]. Они основываются на замене исходной задачи корректно поставленной задачей. Вводится параметр регуляризации α, при стремлении которого к нулю решение этой задачи сходится к решению исходной задачи при определенной связи параметра α с погрешностью исходных данных.

Для характеристики численной неустойчивости СЛАУ определим число обусловленности матрицы [20]:

$$\text{cond(A)} = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}},$$

где $\lambda_{\max}$ и $\lambda_{\min}$ - максимальное и минимальное собственные значения матрицы $A^T A$.

В случае симметричной матрицы $\mathrm{cond(A)} = \frac{\lambda_{\max}}{\lambda_{\min}}$, $\lambda_{\max}$ и $\lambda_{\min}$ - максимальное и минимальное собственные значение матрицы A.

Рассмотрим СЛАУ $Az = b$ и возмущенную по правой части СЛАУ $A(z + \Delta z) = b + \Delta b$. Справедливо следующее неравенство: $\frac{\Delta z}{z} \le \mathrm{cond}(A)\frac{\Delta b}{b}$.

Чем больше число обусловленности СЛАУ, тем более неустойчиво решение СЛАУ. СЛАУ с плохо обусловленной матрицей является неустойчивой, и в этом случае необходимо использовать регуляризацию.

Число обусловленности является важной характеристикой СЛАУ, поэтому его нахождение - важная задача.

Вернемся к обратной задаче гравиметрии о восстановлении плотности.

После дискретизации уравнения на сетке, где задана $\Delta g(x, y)$, задача (5) сводится к решению системы линейных алгебраических уравнений (СЛАУ) либо с симметричной положительно определенной матрицей (горизонтальный слой), либо с несимметричной матрицей (криволинейный слой). Так как уравнение (5) относится к классу некорректно поставленных задач, то СЛАУ

$$Az = b, \tag{6}$$

возникающее в результате дискретизации уравнения, является плохо обусловленной и преобразуется к виду

$$(A + \alpha E)z = b, \tag{7}$$

где α - параметр регуляризации.

Полученную систему уравнений (7) решить на МВС–1000 параллельным итерационным методом, например методом простой итерации:

$$z^{k+1} = z^k - \frac{1}{\lambda_{\max}}[(A + \alpha E)z^k - b], \quad z_0 = 0, \tag{8}$$

где $\lambda_{\max}$ - максимальное собственное значение матрицы $A + \alpha E$.

$\lambda_{\max}$ для матрицы A вычислить с помощью степенного метода.

В случае криволинейного слоя исходная СЛАУ преобразуется к системе уравнений с симметричной матрицей

$$(A^T A + \alpha E)z = A^T b, \tag{9}$$

где A^T - транспонированная матрица.

Полученную систему уравнений (9) решить на МВС–1000 параллельным итерационным методом, например методом простой итерации:

$$z^{k+1} = z^k - \frac{1}{\lambda_{\max}}[(A^T A + \alpha E)z^k - A^T b], \quad z_0 = 0, \tag{10}$$

где $\lambda_{\max}$ - максимальное собственное значение матрицы $A^T A + \alpha E$.

$\lambda_{\max}$ для матрицы A^TA вычислить с помощью степенного метода.

Для проверки сходимости методов (8) и (10) на каждой итерации выдавать в файл номер итерации k и относительную норму невязки $\frac{\|Az^k - b\|}{\|b\|}$, которая должна уменьшаться (процесс сходится).

После решения СЛАУ необходимо построить линии уровня искомой плотности и саму плотность по программе SURFER.

Алгоритм вычисления $\lambda_{\max}$ с помощью степенного метода:

1) выбрать единичный вектор X_0 такой, что $\|X_0\| = 1$;
2) для $k = 1, 2, 3, \ldots$
   вычислить $Y_k = AX_{k-1}$;
   нормировать $X_k = \frac{Y_k}{\|Y_k\|}$;
   проверить условие сходимости $\|X_k - X_{k-1}\| < \varepsilon$;
3) в случае выполнения условия сходимости $\lambda_{\max} \approx \|AX_k\|$.

6.3. Нахождение числа обусловленности матрицы (при решении обратной задачи гравиметрии)

Рассматривается обратная задача гравиметрии о нахождении переменной плотности в горизонтальном и криволинейном слое (см. пункт 6.2, (5)).

Для СЛАУ (7) и (9) вычислить числа обусловленности матриц A и A^TA на МВС–1000 с помощью параллельного алгоритма:

$$\text{cond(A)} = \frac{\lambda_{\max}}{\lambda_{\min}} \quad \text{и} \quad \text{cond}(A^TA) = \frac{\lambda_{\max}}{\lambda_{\min}} \ .$$

Здесь $\lambda_{\max}$ и $\lambda_{\min}$ - максимальное и минимальное собственные значение матрицы A (в случае СЛАУ (7)) или матрицы A^TA (в случае СЛАУ (9)).

$\lambda_{\max}$ вычисляется с помощью степенного метода (см. пункт 6.2),

$\lambda_{\min}$ вычисляется методом обратной итерации.

Алгоритм вычисления $\lambda_{\min}$ методом обратной итерации [21] :

1) выбрать единичный вектор X_0 такой, что $\|X_0\| = 1$;
2) для $k = 1, 2, 3, \ldots$
   вычислить $Y_k = A^{-1}X_{k-1}$ (решить СЛАУ $AY_k = X_{k-1}$ параллельным методом);
   нормировать $X_k = \frac{Y_k}{\|Y_k\|}$;
   проверить условие сходимости $\|X_k - X_{k-1}\| < \varepsilon$;
3) в случае выполнения условия сходимости $\lambda_{\min} \approx \frac{1}{\|A^{-1}X_k\|} = \frac{1}{\|Y_k\|}$.

Здесь норма вектора - $\|X\| = \sqrt{\sum_{k=1}^{n} |x_k|^2}$.

Список литературы

1. Воеводин В.В. Математические модели и методы в параллельных процессах / В.В. Воеводин. М.: Наука, 1986. 296 с.
2. Воеводин В.В. Параллельные вычисления / В.В. Воеводин, Вл.В. Воеводин. СПб.: БХВ-Петербург, 2002. 599 с.
3. Фаддеева В.Н. Параллельные вычисления в линейной алгебре - 1,2 / В.Н. Фаддеева, В.К. Фаддеев // Кибернетика. 1977. № 6. С.28-40; 1982. № 3. С.18- 31.
4. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем / Дж. Ортега. М.: Мир, 1991. 366 с.
5. Валях Е. Последовательно-параллельные вычисления / Е. Валях. М.: Мир, 1985. 456 с.
6. Молчанов И.Н. Введение в алгоритмы параллельных вычислений / И.Н. Молчанов. Киев: Наукова Думка, 1990. 127 с.
7. Параллельные вычисления / под ред. Г. Родрига. М.: Наука, 1986. 374 с.
8. Системы параллельной обработки / под ред. Д. Ивенса. М.: Мир, 1985. 414 с.
9. Корнеев В.Д. Параллельное программирование в MPI / В.Д. Корнеев. Москва; Ижевск: Институт компьютерных исследований, 2003. 303 с.
10. Букатов А.А. Программирование многопроцессорных вычислительных систем / А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. Ростов н/Д.: ЦВВР, 2003. 208 с.
11. Антонов А.С. Параллельное программирование с использованием технологии MPI / А.С. Антонов. М.: МГУ, 2004. 71 с.
12. Flynn M. Very high-speed computing system / M. Flynn // Proc. IEEE. 1966. No. 54. P. 1901- 1909.
13. Hwang K. Computer Architecture and Parallel Processing / K. Hwang, F.A. Briggs. New York, 1984. P. 32- 40.
14. Самарский А.А. Методы решения сеточных уравнений / А.А. Самарский, Е.С. Николаев. М.: Наука, 1978. 590 с.
15. Акимова Е.Н. Распараллеливание алгоритма матричной прогонки / Е.Н. Акимова // Математическое моделирование. 1994. Т. 6, № 9. С. 61- 67.
16. Васин В.В. Операторы и итерационные процессы Фейеровского типа. Теория и приложения / В.В. Васин, И.И. Еремин. Екатеринбург: УрО РАН, 2005. 210 с.
17. Martyshko P.S. On the construction of the density sections using gravity data / P.S. Martyshko, D.E. Koksharov // Extended Abstracts of 66th EAGE Conference & Exhibition. Paris, 7- 12 June 2004. P- 143.
18. Васин В.В. Катастрофы и парадоксы при решении неустойчивых задач на ЭВМ / В.В. Васин, А.Л. Агеев // Математика и кибернетика. Серия 10. М.: Знание, 1991. С. 11- 25.
19. Васин В.В. Методы решения неустойчивых задач: учебное пособие / В.В. Васин. Свердловск: УрГУ, 1989. 94 с.
20. Фаддеев В.К. Вычислительные методы линейной алгебры / В.К. Фаддеев, В.Н. Фаддеева. М.: Гос. издат. физ.- мат. литературы, 1963. 734 с.
21. Парлетт Б. Симметричная проблема собственных значений / Б. Парлетт. М.: Мир, 1983. 382 с.