Учреждение Российской академии наук
Институт проблем управления
им. В.А. Трапезникова РАН

А.М. Сальников, Е.А. Ярошенко,
О.С. Гребенник, С.В. Спиридонов

ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА MPI

Москва 2009

УДК 681.3.06

Сальников А.М., Ярошенко Е.А., Гребенник О.С., Спиридонов С.В. Введение в параллельные вычисления. Основы программирования на языке Си с использованием интерфейса MPI. – М.: ИПУ РАН, 2009. – 123 с.

Рассматривается архитектура многопроцессорных вычислительных систем. Приводится обзор современных суперкомпьютеров. Описывается работа на суперкомпьютерах кластерного типа и особенности параллельного программирования на языке Си с использованием интерфейса MPI. Среди рассматриваемых вопросов уделяется внимание необходимым навыкам разработки программ в операционной системе Linux. Изложенный материал адаптирован применительно к суперкомпьютеру ИПУ РАН.

Научное издание рассчитано на научных работников, аспирантов, студентов и разработчиков прикладных программ.

Рецензенты: д.т.н. Лебедев В. Г.
д.ф.-м.н. проф. Исламов Г. Г.

Утверждено к печати Редакционным советом Института.

Текст воспроизводится в виде, утверждённом
Редакционным советом Института

ISBN 978-5-91450-031-0

ОГЛАВЛЕНИЕ

1. АРХИТЕКТУРА МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

1.1. Введение

Термин «суперкомпьютер» существует со времени появления первых электронных вычислительных машин (ЭВМ) и фактически эволюционирует вместе с термином «компьютер». Сфера применения компьютеров охватывает абсолютно все области человеческой деятельности, и сегодня невозможно представить себе эффективную организацию работы без применения компьютеров. Но если компьютеры в целом развиваются разнонаправлено и самым непостижимым образом проникают в нашу жизнь, то суперкомпьютеры по-прежнему предназначены для того же, для чего разрабатывались первые электронные вычислительные машины, т.е. для решения задач, требующих выполнения больших объемов вычислений. Развитие топливно-энергетического комплекса, авиационной, ракетно-космической промышленности и многих других областей науки и техники требует постоянного увеличения объема производимых расчетов и, таким образом, способствует активному развитию суперкомпьютеров.

Считается, что термин «суперкомпьютер» (англ. supercomputer) впервые стали использовать Джордж Майкл (George Michael) и Сидней Фернбач (Sidney Fernbach), занимавшиеся проблемой параллельных вычислений в Ливерморской национальной лаборатории им. Э. О. Лоуренса (англ. Lawrence Livermore National Laboratory, LLNL) с конца пятидесятых годов двадцатого века.

Ливерморская национальная лаборатория в настоящее время входит в структуру Калифорнийского университета (англ. The University of California, UC) и наряду с Национальной лабораторией в Лос-Аламосе (англ. Los Alamos National Laboratory, LANL) «является главной научно-исследовательской и опытно-конструкторской организацией для решения проблем национальной безопасности США», т.е. одной из двух лабораторий, главной задачей которых служит разработка ядерного оружия. Также лаборатория занимается исследованиями в области наук,

напрямую не связанных с военными технологиями, таких как энергетика, экология и биология (в том числе биоинженерия). Именно в Ливерморской национальной лаборатории за многие годы было создано и успешно эксплуатировалось абсолютное большинство известных суперкомпьютеров, включая IBM Blue Gene/L – самый быстрый в мире суперкомпьютер 2004-2008 гг.

В общеупотребительный лексикон термин «суперкомпьютер» вошел в восьмидесятых годах благодаря феноменальной популярности в СМИ компьютерных систем Сеймура Крея (Seymour Cray), таких как Cray-1, Cray-2 и др. В то время в научно-популярной литературе суперкомпьютером назывался «любой компьютер, который создал Сеймур Крей», хотя сам Крей никогда не называл свои системы суперкомпьютерами, предпочитая использовать традиционное название «компьютер». Более того, еще при жизни Сеймура Крея его именем называли различные суперкомпьютеры, созданные другими талантливыми инженерами, среди которых был Стив Чен (Steve Chen), создатель самого производительного суперкомпьютера начала восьмидесятых Cray X-MP, породившего настоящий суперкомпьютерный бум в СМИ. В настоящее время имя Сеймура Крея носит компания Cray Inc., занимающая достойное место в ряду производителей суперкомпьютеров.

На волне триумфа и популярности суперкомпьютеров Сеймура Крея в конце восьмидесятых годов появилось множество небольших компаний, занимающихся созданием высокопроизводительных компьютеров. Однако уже к середине девяностых большинство из них было приобретено традиционными производителями компьютерного оборудования, такими как IBM и Hewlett-Packard.

Из-за шумихи в средствах массовой информации, созданной при активной «помощи» журналистов, термин «суперкомпьютер» некоторое время трактовался по-разному. Например, в 1989 году знаменитый компьютерный инженер и создатель архитектуры VAX Гордон Белл (Gordon Bell) в шутку предложил считать суперкомпьютером любой компьютер, весящий более тонны. Сегодня термин «суперкомпьютер» вернулся к истокам и по-прежнему обозначает компьютер, способный выполнять

очень большие объемы вычислений, т.е. более производительный и дорогой, чем любой серийно выпускаемый компьютер.

Большинство современных суперкомпьютеров – это кластерные системы, состоящие из большого числа серийно выпускаемых компьютеров, объединенных в единую систему с помощью серийно выпускаемых сетевых интерфейсов. Границы между специализированным программным обеспечением для суперкомпьютеров и типовым программным обеспечением сильно размыты. При этом суперкомпьютеры не ориентированы на работу с типовыми приложениями и этим они принципиально отличаются от других компьютеров с высокой общей производительностью. В отличие от серверов и мейнфреймов, призванных работать с типовыми задачами (управление базами данных, группами пользователей и т.п.), суперкомпьютеры призваны работать со сложными задачами (прогнозирование, моделирование и т.п.), требующими создания собственных приложений. Иногда суперкомпьютеры работают с одним-единственным приложением, использующим всю память и все процессоры системы; в других случаях они обеспечивают выполнение большого числа разнообразных пользовательских программ.

Вычислительное направление применения компьютеров всегда оставалось основным двигателем прогресса в компьютерных технологиях. Основным параметром, отличающим суперкомпьютер от обычного компьютера, является его производительность (вычислительная мощность) – количество арифметических операций за единицу времени. Именно этот показатель с наибольшей очевидностью демонстрирует масштабы прогресса, достигнутого в компьютерных технологиях. Производительность одного из первых суперкомпьютеров ABC, созданного в 1942 году в Университете штата Айова (англ. Iowa State University of Science and Technology, ISU) составляла всего 30 операций в секунду, тогда как пиковая производительность самого мощного суперкомпьютера 2008 года IBM Roadrunner в Национальной лаборатории в Лос-Аламосе составляет 1 квадриллион (10^{15}) операций в секунду.

Таким образом, за 65 лет произошло увеличение производительности суперкомпьютеров в 30 триллионов раз. Невозможно назвать другую сферу человеческой деятельности, где прогресс

был бы столь очевиден и так велик. И такой прогресс оказался возможным не столько за счет тысячекратного увеличения скорости работы электронных схем, сколько за счет максимального распараллеливания обработки данных.

Считается, что идея параллельной обработки данных как мощного резерва увеличения производительности вычислительных машин была высказана еще Чарльзом Бэббиджем примерно за сто лет до появления первого компьютера. Однако уровень развития технологий середины девятнадцатого века не позволил ему реализовать эту идею. С появлением первых компьютеров идеи распараллеливания неоднократно становились отправной точкой при разработке самых передовых и производительных вычислительных систем. Без преувеличения можно сказать, что вся история развития высокопроизводительных вычислений – это история реализации идей параллельной обработки данных на том или ином этапе развития компьютерных технологий, естественно, в сочетании с увеличением скорости работы электронных схем.

Принципиально важными решениями в повышении производительности вычислительных систем были:

- введение конвейерной организации выполнения команд;
- включение в систему команд векторных операций, позволяющих одной командой обрабатывать целые массивы данных;
- распределение вычислений на множество процессоров.

1.2. Традиционная классификация вычислительных систем

Большое разнообразие вычислительных систем породило естественное желание ввести для них какую-то классификацию. Эта классификация должна однозначно относить ту или иную вычислительную систему к некоторому классу, который, в свою очередь, должен достаточно полно ее характеризовать. Попыток такой классификации в разное время предпринималось множество. Одна из первых классификаций, ссылки на которую наиболее часто встречаются в литературе, была предложена М. Флинном в конце 60-х годов прошлого века. Она базируется на

понятиях потоков команд и потоков данных. На основе числа этих потоков выделяется четыре класса архитектур:

- SISD (англ. Single Instruction Single Data) – единственный поток команд и единственный поток данных. По сути дела это классическая машина фон Неймана. К этому классу относятся все однопроцессорные системы.
- SIMD (Single Instruction Multiple Data) – единственный поток команд и множественный поток данных. Типичными представителями являются матричные компьютеры, в которых все процессорные элементы выполняют одну и ту же программу, применяемую к своим (различным для каждого ПЭ) локальным данным. Некоторые авторы к этому классу относят и векторно-конвейерные компьютеры, если каждый элемент вектора рассматривать как отдельный элемент потока данных.
- MISD (Multiple Instruction Single Date) – множественный поток команд и единственный поток данных. М. Флинн не смог привести ни одного примера реально существующей системы, работающей на этом принципе. Некоторые авторы в качестве представителей такой архитектуры называют векторно-конвейерные компьютеры, однако такая точка зрения не получила широкой поддержки.
- MIMD (Multiple Instruction Multiple Date) – множественный поток команд и множественный поток данных. К этому классу относится большинство современных многопроцессорных систем.

Поскольку в этой классификации почти все современные многопроцессорные системы принадлежат одному классу, то вряд ли такая классификация представляет сегодня какую-либо практическую ценность. Тем не менее, используемые в ней термины достаточно часто упоминаются в литературе по параллельным вычислениям.

Эффективность использования современных компьютеров в решающей степени зависит от состава и качества программного обеспечения, установленного на них. В первую очередь это касается программного обеспечения, предназначенного для

разработки прикладных программ. Так, например, недостаточная развитость средств разработки для систем с распределенной памятью долгое время сдерживала их широкое использование. В настоящее время ситуация изменилась, и благодаря кластерным технологиям такие системы стали самой распространенной и доступной разновидностью высокопроизводительных вычислительных систем.

Основной характеристикой при классификации многопроцессорных систем является наличие общей или распределенной памяти. Это различие является важнейшим фактором, определяющим способы параллельного программирования и, соответственно, структуру программного обеспечения.

1.3. Классификация многопроцессорных вычислительных систем

В процессе развития суперкомпьютерных технологий идею повышения производительности вычислительной системы за счет увеличения числа процессоров использовали неоднократно. Если не вдаваться в исторический экскурс и обсуждение всех таких попыток, то можно следующим образом вкратце описать развитие событий.

Экспериментальные разработки по созданию многопроцессорных вычислительных систем (МВС) начались в семидесятых годах двадцатого века. Одной из первых таких систем стала разработанная в 1976 году в Университете Иллинойса в Урбане-Шампэйн (англ. University of Illinois at Urbana-Champaign, UIUC) МВС ILLIAC IV, которая включала 64 (в проекте до 256) процессорных элемента (ПЭ), работающих по единой программе, применяемой к содержимому собственной оперативной памяти каждого ПЭ. Обмен данными между процессорами осуществлялся через специальную матрицу коммуникационных каналов. Указанная особенность коммуникационной системы дала название «матричные суперкомпьютеры» соответствующему классу МВС. Более широкий класс МВС с распределенной памятью и с произвольной коммуникационной системой получил впоследствии название «многопроцессорные системы с массовым параллелизмом», или МВС с МРР-архитектурой (англ. Massive Parallel

Processing, MPP). При этом, как правило, каждый из ПЭ системы с МРР-архитектурой является универсальным процессором, действующим по своей собственной программе (в отличие от общей программы для всех ПЭ матричной МВС).

Первые матричные МВС выпускались с конца семидесятых годов буквально поштучно, поэтому их стоимость была фантастически высокой. Серийные образцы подобных систем, такие как первая коммерческая матричная МВС ICL DAP (англ. Distributed Array Processor), включавшая до 8192 ПЭ, появились примерно в это же время, однако не получили широкого распространения ввиду сложности программирования МВС с одним потоком управления (с одной программой, общей для всех ПЭ).

Первые промышленные образцы многопроцессорных систем появились на базе векторно-конвейерных компьютеров в середине восьмидесятых годов. Наиболее распространенными МВС такого типа были суперкомпьютеры Сеймура Крея. Однако такие системы были чрезвычайно дорогими и производились небольшими сериями. Как правило, в подобных компьютерах объединялось от 2 до 16 процессоров, которые имели равноправный (симметричный) доступ к общей оперативной памяти. В связи с этим они стали называться симметричными многопроцессорными системами (англ. Symmetric Multiprocessing, SMP).

Как альтернатива дорогим многопроцессорным системам на базе векторно-конвейерных процессоров была предложена идея строить эквивалентные по мощности многопроцессорные системы из большого числа дешевых серийно выпускаемых микропроцессоров. Однако очень скоро обнаружилось, что архитектура SMP обладает весьма ограниченными возможностями по наращиванию числа процессоров в системе из-за резкого увеличения числа конфликтов при обращении к общей шине памяти. В связи с этим оправданной представлялась идея снабдить каждый процессор собственной оперативной памятью, превращая компьютер в объединение независимых вычислительных узлов. Такой подход значительно увеличил масштабируемость многопроцессорных систем, но в свою очередь потребовал разработки специального способа обмена данными между вычислительными узлами, реализуемого обычно в виде механизма передачи сообщений (англ. Message Passing). Компьютеры с такой архи-

тектурой являются наиболее яркими представителями современных MPP-систем. В настоящее время оба этих направления (или их комбинации) являются доминирующими в развитии суперкомпьютерных технологий.

Нечто среднее между SMP и MPP представляют собой NUMA-архитектуры (англ. Non-Uniform Memory Access), в которых память физически разделена, но логически общедоступна. При этом время доступа к различным блокам памяти становится неодинаковым. В одной из первых NUMA-систем Cray T3D, выпущенной в 1993 году, время доступа к памяти соседнего процессора было в шесть раз больше, чем к памяти своего собственного процессора.

В настоящее время развитие суперкомпьютерных технологий идет по четырем основным направлениям:

- векторно-конвейерные системы;
- SMP- и NUMA-системы;
- MPP-системы;
- кластерные системы.

1.4. Векторно-конвейерные системы

Первый векторно-конвейерный компьютер Cray-1 появился в 1976 году. Архитектура его оказалась настолько удачной, что он положил начало целому семейству компьютеров. Название этому семейству компьютеров дали два принципа, заложенные в архитектуре процессоров:

- конвейерная организация обработки потока команд;
- введение в систему команд набора векторных операций, которые позволяют оперировать целыми массивами данных.

Длина одновременно обрабатываемых векторов в современных векторных системах составляет, как правило, 128 или 256 элементов. Очевидно, что векторные процессоры должны иметь гораздо более сложную структуру и по сути дела содержать множество арифметических устройств. Основное назначение векторных операций состоит в распараллеливании выполнения операторов цикла, в которых в основном и сосредоточена большая часть вычислительной работы. Для этого циклы подвергаются процедуре векторизации с тем, чтобы они могли реализовываться с использованием векторных команд. Как правило, это выполняется автоматически компиляторами при создании исполняемого кода программы. Поэтому векторно-конвейерные компьютеры не требуют какой-то специальной технологии программирования, что и явилось решающим фактором в их успехе на компьютерном рынке. Тем не менее, требуется соблюдение некоторых правил при написании циклов с тем, чтобы компилятор мог их эффективно векторизовать.

Исторически это были первые компьютеры, к которым в полной мере было применимо понятие суперкомпьютер. Как правило, несколько векторно-конвейерных процессоров (от двух до шестнадцати) работают в режиме с общей памятью (SMP), образуя вычислительный узел, а несколько таких узлов объединяются с помощью коммутаторов, образуя NUMA- или MPP-систему. Типичными представителями такой архитектуры являются компьютеры серий Cray J90 (1994 год), Cray T90 (1995 год), NEC SX-4 (1995 год), Cray SV1 (1998 год), NEC SX-5 (1999 год). Уровень развития микроэлектронных технологий долгое время не позволял производить однокристальные векторные процессоры, поэтому эти системы были довольно громоздки и чрезвычайно дороги. В связи с этим, начиная с середины 90-х годов, когда появились достаточно мощные суперскалярные микропроцессоры, интерес к этому направлению значительно ослабел.

Суперкомпьютеры с векторно-конвейерной архитектурой стали проигрывать системам с массовым параллелизмом. Однако в марте 2002 г. компания NEC представила систему Earth Simulator (ES) из 5120 векторно-конвейерных процессоров, которая в 5 раз превысила производительность предыдущего обладателя рекорда, очередной MPP-системы из Ливерморской национальной лаборатории ASCI White (2000 год), состоящей из 8192 суперскалярных микропроцессоров. Это заставило многих по-новому взглянуть на перспективы векторно-конвейерных систем. Производительность суперкомпьютера NEC ES оставалась самой высокой в мире до момента запуска в 2004 году первой версии суперкомпьютера IBM Blue Gene/L с MPP-архитектурой.

1.5. Симметричные многопроцессорные системы (SMP и NUMA)

Характерной чертой симметричных многопроцессорных систем является то, что все процессоры имеют прямой и равноправный доступ к любой точке общей памяти. Первые SMP-системы состояли из нескольких однородных процессоров и массива общей памяти, к которой процессоры подключались через общую системную шину. Однако очень скоро обнаружилось, что такая архитектура непригодна для создания каких-либо масштабных систем.

Первая возникшая проблема – это большое число конфликтов при обращении к общей шине. Остроту проблемы удалось частично снять разделением памяти на блоки, подключение к которым с помощью коммутаторов позволило распараллелить обращения от различных процессоров. Однако и в таком подходе неприемлемо большими казались накладные расходы для систем более чем с 32 процессорами.

Современные системы архитектуры SMP состоят, как правило, из нескольких однородных серийно выпускаемых микропроцессоров и массива общей памяти, подключение к которой производится либо с помощью общей шины, либо с помощью коммутатора.

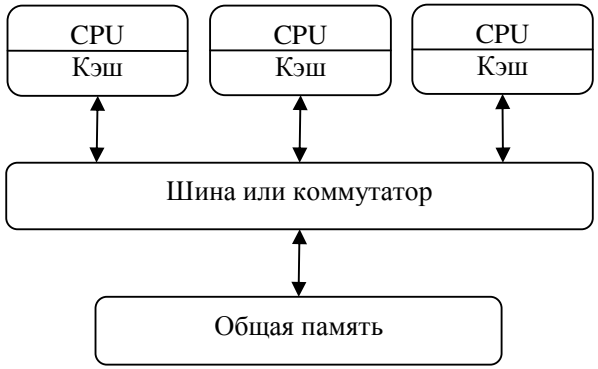

Рис. 1. Архитектура SMP-системы

Наличие общей памяти значительно упрощает организацию взаимодействия процессоров между собой и упрощает программирование, поскольку параллельная программа работает в едином адресном пространстве. Однако за этой кажущейся простотой скрываются проблемы, присущие системам этого типа. Все они, так или иначе, связаны с оперативной памятью. Дело в том, что в настоящее время даже в однопроцессорных системах самым узким местом является оперативная память, скорость работы которой отстает от скорости работы процессора. Для того чтобы сгладить этот разрыв, современные процессоры снабжаются скоростной буферной памятью (кэш-памятью), скорость работы которой значительно выше, чем скорость работы основной памяти.

В качестве примера приведем данные измерения пропускной способности кэш-памяти и основной памяти для персонального компьютера образца 1999 года на базе процессора Pentium III. В данном процессоре кэш-память имела два уровня:

- L1 (буферная память команд) – объем 32 Кб, скорость обмена 9976 Мб/с;
- L2 (буферная память данных) – объем 256 Кб, скорость обмена 4446 Мб/с.

В то же время скорость обмена с основной памятью составляла всего 255 Мб/с. Это означало, что для полной согласованности со скоростью работы процессора скорость работы основной памяти должна была быть минимум в 40 раз выше. Для сравнения, скорость обмена 10 Гбайт/с обеспечивает контроллер памяти (впервые интегрированный непосредственно в кристалл CPU) процессора Intel Core i7 образца 2009 года.

Очевидно, что при проектировании многопроцессорных систем эти проблемы еще более обостряются. Помимо хорошо известной проблемы конфликтов при обращении к общей шине памяти возникла и новая проблема, связанная с иерархической структурой организации памяти современных компьютеров. В многопроцессорных системах, построенных на базе микропроцессоров со встроенной кэш-памятью, нарушается принцип равноправного доступа к любой точке памяти. Данные, находящиеся в кэш-памяти некоторого процессора, недоступны для

других процессоров. Это означает, что после каждой модификации копии некоторой переменной, находящейся в кэш-памяти какого-либо процессора, необходимо производить синхронную модификацию самой этой переменной, расположенной в основной памяти.

С большим или меньшим успехом эти проблемы решаются в рамках общепринятой в настоящее время архитектуры ccNUMA (англ. Cache coherent Non-Uniform Memory Access). В такой архитектуре память физически распределена, но логически общедоступна. Это с одной стороны позволяет работать с единым адресным пространством, а с другой – увеличивает масштабируемость систем.

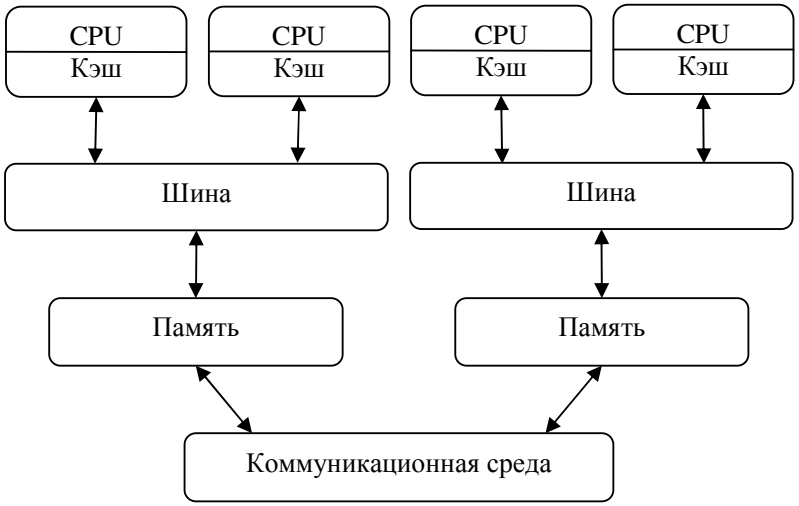

Рис. 2. Архитектура NUMA-системы

Когерентность кэш-памяти поддерживается на аппаратном уровне, что не избавляет, однако, от накладных расходов на ее поддержание. В отличие от классических SMP-систем память становится трехуровневой:

- кэш-память процессора;
- локальная оперативная память;
- удаленная оперативная память.

Время обращения к различным уровням может отличаться на порядок, что сильно усложняет написание эффективных параллельных программ для таких систем.

Перечисленные обстоятельства значительно ограничивают возможности по наращиванию производительности ccNUMA систем путем простого увеличения числа процессоров. Тем не менее, эта технология позволяет создавать системы, содержащие до 256 процессоров с общей производительностью порядка 200 млрд. операций в секунду. Системы этого типа серийно производятся многими компьютерными фирмами как многопроцессорные серверы с числом процессоров от 2 до 128 и до недавнего времени они прочно удерживали лидерство в классе малых суперкомпьютеров. Типичными представителями данного класса суперкомпьютеров являются компьютеры на базе процессоров AMD Opteron или Intel Itanium 2, например, HP Integrity Superdome (2002 год).

С конца 2008 года архитектура ccNUMA также поддерживается микроархитектурой Nehalem, преемником Core 2, первыми представителями которой стали процессоры Intel Core i7.

Неприятным свойством всех симметричных многопроцессорных систем является то, что их стоимость растет быстрее, чем производительность при увеличении числа процессоров в системе. Кроме того, из-за задержек при обращении к общей памяти неизбежно взаимное торможение при параллельном выполнении даже независимых программ.

1.6. Системы с массовым параллелизмом (MPP)

Проблемы, присущие многопроцессорным системам с общей памятью, простым и естественным образом устраняются в системах с массовым параллелизмом. Компьютеры этого типа представляют собой многопроцессорные системы с распределенной памятью, в которых с помощью некоторой коммуникационной среды объединяются однородные вычислительные узлы.

Каждый из узлов состоит из одного или нескольких процессоров, собственной оперативной памяти, коммуникационного оборудования, подсистемы ввода/вывода, т.е. обладает всем

необходимым для независимого функционирования. При этом на каждом узле может функционировать либо полноценная операционная система, либо урезанный вариант, поддерживающий только базовые функции ядра, а полноценная операционная система работает на специальном управляющем компьютере.

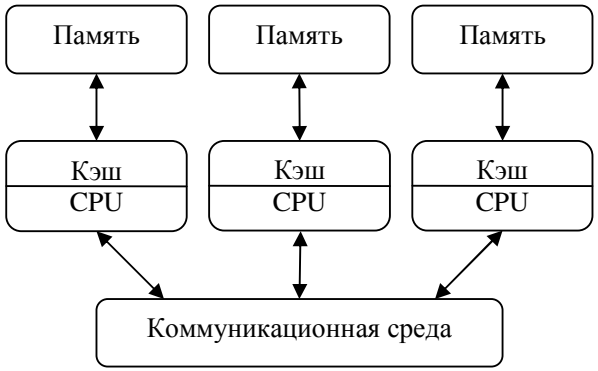

Рис. 3. Архитектура MPP-системы

Процессоры в таких системах имеют прямой доступ только к своей локальной памяти. Доступ к памяти других узлов реализуется обычно с помощью механизма передачи сообщений. Такая архитектура вычислительной системы устраняет одновременно как проблему конфликтов при обращении к памяти, так и проблему когерентности кэш-памяти. Это дает возможность практически неограниченно наращивать число процессоров в системе, увеличивая тем самым ее производительность. Успешно функционируют MPP-системы с сотнями и тысячами процессоров, производительность наиболее мощных из них достигает нескольких сотен триллионов операций в секунду. Важным свойством MPP-систем является их высокая масштабируемость. В зависимости от вычислительных потребностей для достижения необходимой производительности требуется просто собрать систему с нужным числом узлов.

Однако на практике устранение одних проблем, как это обычно бывает, порождает другие. Для MPP-систем на первый план выходит проблема эффективности коммуникационной

среды. Самым простым и наиболее эффективным было бы соединение каждого процессора с каждым. Но тогда на каждом узле суперкомпьютера, содержащего N процессоров, потребовалось бы $N - 1$ коммуникационных каналов, желательно двунаправленных. Различные производители MPP-систем использовали разные топологии. В суперкомпьютерах Intel Paragon (1992 год) процессоры образовывали прямоугольную двумерную сетку. Для этого на каждом узле достаточно было иметь четыре коммуникационных канала. В суперкомпьютерах Cray T3D (1993 год) и Cray T3E (1995 год) использовалась топология трехмерного тора. Соответственно, на каждом узле этого компьютера было по шесть коммуникационных каналов. Фирма nCUBE в начале девяностых годов использовала в своих компьютерах топологию n-мерного гиперкуба.

Каждая из упомянутых топологий имела свои преимущества и недостатки. Отметим, что при обмене данными между процессорами, не являющимися ближайшими соседями, происходит трансляция данных через промежуточные узлы. Очевидно, что в узлах должны быть предусмотрены какие-то аппаратные средства, которые освобождали бы центральный процессор от участия в трансляции данных. В последнее время для соединения вычислительных узлов чаще используется иерархическая система высокоскоростных коммутаторов, как это впервые было реализовано в компьютерах IBM SP2 (1994 год). Такая топология дает возможность прямого обмена данными между любыми узлами, без участия в этом промежуточных узлов.

Системы с распределенной памятью идеально подходят для параллельного выполнения независимых программ, поскольку при этом каждая программа выполняется на своем узле и никаким образом не влияет на выполнение других программ. Однако при разработке параллельных программ приходится учитывать более сложную, чем в SMP-системах, организацию памяти. Оперативная память в MPP-системах имеет трехуровневую структуру:

- кэш-память процессора;
- локальная оперативная память узла;
- оперативная память других узлов.

При этом отсутствует возможность прямого доступа к данным, расположенным на других узлах. Такие данные должны быть предварительно переданы в тот узел, который в них нуждается. Это значительно усложняет программирование. Кроме того, обмены данными между узлами выполняются значительно медленнее, чем обработка данных в локальной оперативной памяти узлов. Поэтому написание эффективных параллельных программ для таких компьютеров представляет собой более сложную задачу, чем для SMP-систем.

1.7. Кластерные системы

Кластерные технологии стали логическим продолжением развития идей, заложенных в архитектуре MPP. Если процессорный модуль в MPP-системе представляет собой законченную вычислительную систему, то следующий шаг напрашивается сам собой: почему бы в качестве таких вычислительных узлов не использовать обычные серийно выпускаемые компьютеры. Развитие коммуникационных технологий, а именно, появление высокоскоростного сетевого оборудования и специального программного обеспечения, такого как интерфейс MPI, реализующего механизм передачи сообщений над стандартными сетевыми протоколами, сделали кластерные технологии общедоступными. Сегодня не составляет большого труда создать небольшую кластерную систему, объединив вычислительные мощности нескольких компьютеров лаборатории.

Привлекательной чертой кластерных технологий является то, что они позволяют для достижения необходимой производительности объединять в единые вычислительные системы компьютеры разного типа, от персональных компьютеров до промышленных блейд-серверов.

Широкое распространение кластерные технологии получили как средство создания систем суперкомпьютерного класса из составных частей массового производства, значительно удешевляющих стоимость готовой системы. Одним из первых современных кластеров можно считать созданный в 1998 году в Университете штата Пенсильвания (англ. Pennsylvania State University) суперкомпьютер COCOA (англ. The Cost Effective

Computing Array), в котором на базе 25 двухпроцессорных персональных компьютеров общей стоимостью порядка 100 тыс. долл. США была создана система с производительностью, эквивалентной производительности 48-процессорного Cray T3D стоимостью несколько млн. долл. США.

Конечно, о полной эквивалентности упомянутых систем говорить не приходилось, потому что производительность систем с распределенной памятью очень сильно зависит от производительности коммуникационной среды. Коммуникационную среду можно достаточно полно охарактеризовать двумя параметрами: латентностью (временем задержки при посылке сообщения) и пропускной способностью (скоростью передачи информации). Для компьютера Cray T3D эти параметры составляли 1 мкс и 480 Мб/с соответственно. Для кластера COCOA, в котором в качестве коммуникационной среды использовалась стандартная на тот момент сеть Fast Ethernet, латентность и пропускная способность составляли 100 мкс и 10 Мб/с соответственно. При таких параметрах найдется не так много задач, эффективно решаемых на достаточно большом числе процессоров. Это обстоятельство долгое время отчасти объясняло чрезмерную стоимость суперкомпьютеров. Для сравнения, интерфейс InfiniBand 4X SDR, используемый в кластере ИПУ РАН, обеспечивает латентность 200 нс и пропускную способность 8 Гб/с (на уровне MPI – 1 мкс и 800 Мб/с соответственно).

Если говорить кратко, то кластер – это связанный набор полноценных компьютеров, используемый в качестве единого вычислительного ресурса. Преимущества кластерной системы перед набором независимых компьютеров очевидны. Во-первых, разработано множество диспетчерских систем пакетной обработки заданий, позволяющих послать задание на обработку кластеру в целом, а не какому-то отдельному компьютеру. Эти диспетчерские системы автоматически распределяют задания по свободным вычислительным узлам или буферизуют их при отсутствии таковых, что позволяет обеспечить более равномерную и эффективную загрузку компьютеров. Во-вторых, появляется возможность совместного использования вычислительных ресурсов нескольких компьютеров для решения одной задачи.

Для создания современных кластеров уже не используются однопроцессорные персональные компьютеры – сегодня их место заняли многопроцессорные, многоядерные и зачастую сами по себе высокопроизводительные SMP-серверы. При этом не накладывается никаких ограничений на состав и архитектуру узлов. Каждый из узлов может функционировать под управлением своей собственной операционной системы. Чаще всего используются UNIX-подобные ОС, причем как коммерческие (Solaris, Tru64 Unix), так и свободные (GNU/Linux, FreeBSD). В тех случаях, когда узлы кластера неоднородны, итоговая система называется гетерогенным кластером.

При создании кластера можно выделить два подхода:

1. В кластер объединяются полнофункциональные компьютеры, которые продолжают работать как самостоятельные единицы, например, рабочие станции лаборатории. Такой подход применяется при создании небольших кластерных систем или в целях тестирования технологий параллельной обработки данных.
2. Системные блоки компьютеров компактно размещаются в стандартных серверных стойках, а для управления системой и для запуска задач выделяется один или несколько полнофункциональных компьютеров. Такой подход применяется при целенаправленном создании мощных вычислительных ресурсов.

За последние годы разработаны специальные технологии соединения компьютеров в кластер. Наиболее широко в настоящее время применяется высокоскоростная коммутируемая последовательная шина InfiniBand. Это обусловлено простотой ее использования и относительно низкой стоимостью коммуникационного оборудования, которое обеспечивает приемлемую скорость обмена между узлами от 2,5 Гбит/с до 96 Гбит/с, в зависимости от типа установленного оборудования.

Разработчики пакета подпрограмм ScaLAPACK, предназначенного для решения задач линейной алгебры на многопроцессорных системах, в которых велика доля коммуникационных операций, сформулировали следующим образом требование к многопроцессорной системе: «Скорость межпроцессорных обменов между двумя узлами, измеренная в Мб/с, должна быть

не менее 1/10 пиковой производительности вычислительного узла, измеренной в мегафлопс». Таким образом, если в качестве вычислительных узлов использовать рабочие станции образца 2006 года на базе Intel Core 2 Duo с пиковой производительностью 15 гигафлопс, то аппаратура InfiniBand будет обеспечивать необходимый теоретический минимум, а все еще популярная Gigabit Ethernet – уже нет. В целом можно утверждать, что InfiniBand в настоящее время становится новым сетевым стандартом, потому что соответствует потребностям не только суперкомпьютеров кластерного типа, но и любых других высокопроизводительных серверных систем.

2. ПРОГРАММИРОВАНИЕ ДЛЯ МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

2.1. Программирование для систем с общей памятью

К системам с общей памятью относятся компьютеры с SMP-архитектурой, различные разновидности NUMA-систем и многопроцессорные векторно-конвейерные компьютеры. Характерным словом для этих компьютеров является «единый»: единая оперативная память, единая операционная система, единая подсистема ввода-вывода. Только процессоры образуют множество. Единая UNIX-подобная операционная система, управляющая работой всего компьютера, функционирует в виде множества процессов. Каждая пользовательская программа также запускается как отдельный процесс.

Операционная система сама каким-то образом распределяет процессы по процессорам. В принципе, для распараллеливания программ можно использовать механизм порождения процессов. Однако этот механизм не очень удобен, поскольку каждый процесс функционирует в своем адресном пространстве, и основное достоинство этих систем – общая память – не может быть использована простым и естественным образом.

Для распараллеливания программ используется механизм порождения нитей (англ. threads) – легковесных процессов, для которых не создается отдельного адресного пространства, но

которые на многопроцессорных системах также распределяются по процессорам. В языке программирования Си возможно прямое использование этого механизма для распараллеливания программ посредством вызова соответствующих системных функций, а в компиляторах с языка Fortran этот механизм используется либо для автоматического распараллеливания, либо в режиме задания распараллеливающих директив компилятору (такой подход поддерживают и компиляторы языка Си).

Все производители симметричных многопроцессорных систем в той или иной мере поддерживают стандарт PThreads (POSIX Threads) и включают в программное обеспечение распараллеливающие компиляторы для популярных языков программирования или предоставляют набор директив компилятору для распараллеливания программ. В частности, многие поставщики компьютеров SMP-архитектуры (Sun, HP, SGI) в своих компиляторах предоставляют специальные директивы для распараллеливания циклов. Однако эти наборы директив, во-первых, весьма ограничены и, во-вторых, несовместимы между собой. В результате этого разработчикам приходится распараллеливать прикладные программы отдельно для каждой платформы.

В последние годы все более популярной, особенно на платформах Sun Solaris и GNU/Linux, становится система программирования OpenMP (англ. Open Multi-Processing), являющаяся во многом обобщением и расширением этих наборов директив. Интерфейс OpenMP задумывался как стандарт для программирования в модели общей памяти. В OpenMP входят спецификации набора директив компилятору, процедур и переменных среды. По сути дела, он реализует идею «инкрементального распараллеливания, когда разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP-директивы. При этом система программирования OpenMP предоставляет разработчику большие возможности по контролю над поведением параллельного приложения.

Вся программа разбивается на последовательные и параллельные области. Все последовательные области выполняет главная нить, порождаемая при запуске программы, а при входе в параллельную область главная нить порождает дополнительные нити. Предполагается, что OpenMP-программа без какой-либо модификации должна работать как на многопроцессорных системах, так и на однопроцессорных. В последнем случае директивы OpenMP просто игнорируются.

Следует отметить, что наличие общей памяти не препятствует использованию технологий программирования, разработанных для систем с распределенной памятью. Многие производители SMP-систем предоставляют также такие технологии программирования, как MPI и PVM. В этом случае в качестве коммуникационной среды выступает распределенная память.

2.2. Программирование для систем с распределенной памятью

В системах с распределенной памятью на каждом вычислительном узле функционируют собственные копии операционной системы, под управлением которых выполняются независимые программы. Это могут быть как действительно независимые программы, так и параллельные ветви одной программы. В этом случае единственно возможным механизмом взаимодействия между ними является механизм передачи сообщений.

Стремление добиться максимальной производительности заставляет разработчиков при реализации механизма передачи сообщений учитывать особенности архитектуры многопроцессорной системы. Это способствует написанию более эффективных, но ориентированных на конкретный компьютер программ. Вместе с тем независимыми разработчиками программного обеспечения было предложено множество реализаций механизма передачи сообщений, независимых от конкретной платформы.

В 1995 г. был принят стандарт механизма передачи сообщений MPI (англ. Message Passing Interface). Он готовился с 1992 по 1994 гг. группой Message Passing Interface Forum, в которую вошли представители более чем сорока организаций из США и Европы. Основная цель, которую ставили перед собой разработчики MPI – это обеспечение полной независимости приложений, написанных с использованием MPI, от архитектуры многопроцессорной системы, без какой-либо существенной потери произ-

водительности. По замыслу авторов это должно было стать мощным стимулом для разработки прикладного программного обеспечения и стандартизованных библиотек подпрограмм для многопроцессорных систем с распределенной памятью. Подтверждением того, что эта цель была достигнута, служит тот факт, что в настоящее время этот стандарт поддерживается всеми производителями многопроцессорных систем. Реализации MPI успешно работают не только на классических MPP-системах, но также на SMP-системах и в сетях рабочих станций (в том числе неоднородных).

Реализация MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Большинство реализаций MPI поддерживают интерфейсы для языков Си, C++ и Fortran. Библиотека MPI включает в себя множество функций передачи сообщений типа точка-точка, развитый набор функций для выполнения коллективных операций и управления процессами параллельного приложения.

Основное отличие MPI от предшественников в том, что явно вводятся понятия групп процессов, с которыми можно оперировать как с конечными множествами, а также областей связи и коммуникаторов, описывающих эти области связи. Это предоставляет программисту очень гибкие средства для написания эффективных параллельных программ. В настоящее время MPI является основной технологией программирования для многопроцессорных систем с распределенной памятью.

Несмотря на значительные успехи в развитии технологии программирования с использованием механизма передачи сообщений, трудоемкость программирования с использованием этой технологии по-прежнему велика.

2.3. Параллельное программирование

Разработка параллельных программ является весьма трудоемким процессом, особенно для MPP-систем, поэтому, прежде чем приступать к этой работе, важно правильно оценить как ожидаемый эффект от распараллеливания, так и трудоемкость выполнения этой работы.

Решение на компьютере любой вычислительной задачи для выбранного алгоритма решения предполагает выполнение некоторого фиксированного объема арифметических операций. Ускорить решение задачи можно одним из трех способов:

- использовать более производительную вычислительную систему с более быстрым процессором и более скоростной системной шиной;
- оптимизировать программу, например, в плане более эффективного использования скоростной кэш-памяти;
- распределить вычислительную работу между несколькими процессорами, т.е. перейти на параллельные технологии.

Очевидно, что без распараллеливания не обойтись при программировании алгоритмов решения тех задач, которые в принципе не могут быть решены на однопроцессорных системах. Это может проявиться в двух случаях: либо когда для решения задачи требуется слишком много времени, либо когда для программы недостаточно оперативной памяти на однопроцессорной системе.

Для небольших задач зачастую оказывается, что параллельная версия работает медленнее, чем однопроцессорная. Заметный эффект от распараллеливания начинает наблюдаться при решении систем уравнений с большим количеством неизвестных. На кластерных системах ситуация еще хуже. Разработчики уже упоминавшегося ранее пакета ScaLAPACK для многопроцессорных систем с приемлемым соотношением между производительностью узла и скоростью обмена дают следующую формулу для количества процессоров P, которое рекомендуется использовать при решении задач линейной алгебры

(1) $P = m \times n / 106$,

где $m \times n$ – размерность матрицы.

Таким образом, на один процессор должен приходиться блок матрицы размером примерно 1000×1000. Формула (1) носит рекомендательный характер (особенно для процессоров последних поколений), при этом наглядно иллюстрируя масштаб задач, решаемых пакетами типа ScaLAPACK.

Рост эффективности распараллеливания при увеличении размера решаемой системы уравнений объясняется ростом объема вычислительной работы пропорционально n^3, а количества обменов между процессорами пропорционально n^2. Это снижает относительную долю коммуникационных затрат при увеличении размерности системы уравнений. Однако на эффективность параллельного приложения влияют не только коммуникационные издержки.

Параллельные технологии в MPP-системах допускают две модели программирования, похожие на традиционную классификацию вычислительных систем М. Флинна:

- SPMD (англ. Single Program Multiple Date) – на всех процессорах выполняются копии одной программы, обрабатывающие разные блоки данных;
- MPMD (англ. Multiple Program Multiple Date) – на всех процессорах выполняются разные программы, обрабатывающие разные данные.

Второй вариант иногда называют функциональным распараллеливанием. Такой подход, в частности, используется в системах обработки видеоинформации, когда множество квантов данных должны проходить несколько этапов обработки. В этом случае вполне оправданной будет конвейерная организация вычислений, при которой каждый этап обработки выполняется на отдельном процессоре.

Однако функциональное распараллеливание имеет весьма ограниченное применение, поскольку организовать достаточно длинный конвейер, да еще с равномерной загрузкой всех процессоров, весьма сложно. Наиболее распространенным режимом работы на системах с распределенной памятью является загрузка на некотором числе процессоров копий одной и той же программы.

Разработка параллельной программы подразумевает разбиение задачи на P подзадач, каждая из которых решается на отдельном процессоре.

В параллельном программировании термин «подзадача» имеет весьма широкий смысл. В MPMD-модели подзадачей называется функционально выделенный фрагмент программы.

В SPMD-модели подзадачей чаще называется обработка некоторого блока данных.

Таким образом, схему параллельной программы, использующей механизм передачи сообщений, можно упрощенно записать на языке Си следующим образом:

```
if (proc_id == 1) task1 ();
if (proc_id == 2) task2 ();
…
result = reduce (result1, result2, …);
```

Здесь *proc_id* – идентификатор процессора, а функция *reduce* формирует некий глобальный результат на основе полученных на каждом процессоре локальных результатов работы функций *task1*, *task2* и т. д. В этом случае одна и та же копия программы будет выполняться на *P* процессорах, но каждый процессор будет решать только свою подзадачу. Если разбиение на подзадачи достаточно равномерное, а накладные расходы на обмены не слишком велики, то можно ожидать близкого к *P* коэффициента ускорения решения задачи.

На практике процедура распараллеливания чаще всего применяется к циклам. Тогда в качестве отдельных подзадач могут выступать экземпляры тела цикла, выполняемые для различных значений переменной цикла. Рассмотрим простейший пример:

```
for (i=1, i<=1000, i++) c[i] = c[i] + a[i+1];
```

В данном примере можно выделить 1000 независимых подзадач вычисления элементов массива *c*, каждая из которых теоретически может быть выполнена на отдельном процессоре.

Предположим, что в распоряжении программиста имеется вычислительная система, состоящая из 10 процессоров, тогда в качестве независимой подзадачи можно оформить вычисление 100 элементов массива *c*. При этом до выполнения вычислений необходимо принять решение о способе размещения этих массивов в памяти процессоров. Возможны два варианта размещения:

1. Все массивы целиком хранятся в каждом процессоре, тогда процедура распараллеливания сводится к вычислению стартового и конечного значений переменной цикла для каждого процессора. В каждом процессоре будет хра-

ниться своя копия всего массива, в которой будет модифицирована только часть элементов. В конце вычислений, возможно, потребуется сборка модифицированных частей со всех процессоров.

2. Все или часть массивов распределены по процессорам, т.е. в каждом процессоре хранится $1/P$ часть массива. Тогда может потребоваться алгоритм установления связи индексов локального массива в некотором процессоре с глобальными индексами всего массива, например, если значение элемента массива является некоторой функцией индекса. Если в процессе вычислений окажется, что какие-то требующиеся компоненты массива отсутствуют в данном процессоре, то потребуется их пересылка из других процессоров.

Вопрос распределения данных по процессорам и связь этого распределения с эффективностью параллельной программы является основным вопросом параллельного программирования.

Хранение копий всех массивов во всех процессорах во многих случаях уменьшает накладные расходы на пересылки данных, однако не дает выигрыша в плане объема решаемой задачи и создает сложности синхронизации копий массива при независимом изменении его элементов различными процессорами.

Распределение массивов по процессорам позволяет решать значительно более объемные задачи, наиболее подходящие для распараллеливания, но тогда на первый план выходит проблема минимизации пересылок данных.

Рассмотренный выше пример вычисления элементов массива достаточно хорошо укладывается в схему методологического подхода к решению задачи на многопроцессорной системе, изложенную Яном Фостером (Ian Foster) [4]. Автор выделяет четыре этапа разработки параллельного алгоритма:

1. Разбиение задачи на минимальные независимые подзадачи (англ. partitioning).
2. Установление связей между подзадачами (англ. communication).
3. Объединение подзадач с целью минимизации коммуникаций (англ. agglomeration).

4. Распределение укрупненных подзадач по процессорам с целью обеспечения равномерной загрузки процессоров (англ. mapping).

Подобные схемы – не более чем краткие изложения философии параллельного программирования, лишь подчеркивающие отсутствие какого-либо формализованного подхода в параллельном программировании для MPP-систем. Если 1-й и 2-й пункты имеют более или менее однозначное решение, то решение задач 3-го и 4-го пунктов основывается главным образом на интуиции разработчика.

Предположим, что требуется исследовать поведение определителя матрицы в зависимости от некоторого параметра. Один из подходов состоит в том, чтобы написать параллельную версию подпрограммы вычисления определителя и вычислить его значения для исследуемого интервала значений параметра. Однако, если размер матрицы относительно невелик, то может оказаться, что значительные усилия на разработку параллельной подпрограммы вычисления определителя не дадут сколь либо существенного выигрыша в скорости работы программы. В этом случае более продуктивным подходом будет использование обычной оптимизированной однопроцессорной подпрограммы, а по процессорам разложить исследуемый диапазон изменений параметра.

Теперь рассмотрим свойства многопроцессорной MPP-системы, необходимые для выполнения на ней параллельных программ. Минимальный набор функций невелик:

- процессоры в системе должны иметь уникальные идентификаторы (номера);
- должна существовать функция самоидентификации процессора;
- должны существовать функции обмена между двумя процессорами: посылка сообщения одним процессором и прием сообщения другим процессором.

Парадигма передачи сообщений подразумевает асимметрию функций передачи и приема сообщений. Инициатива инициализации обмена принадлежит передающей стороне. Принимающий процессор может принять только то, что ему было послано.

Различные реализации механизма передачи сообщений для облегчения разработки параллельных программ расширяют минимальный набор функций.

2.4. Оценка эффективности распараллеливания программ

В идеале решение задачи на P процессорах должно выполняться в P раз быстрее, чем на одном процессоре, или/и должно позволить решить задачу, содержащую в P раз больше данных. На практике такое недостижимо, что наглядно иллюстрируется законом Амдала:

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

где S – ускорение работы программы на P процессорах; f – доля непараллельного кода в программе.

Формула (2) справедлива как для систем с общей памятью, так и для систем с передачей сообщений, однако термин «доля непараллельного кода» в этих системах имеет разный смысл.

В программах, созданных для SMP-систем, долю непараллельного кода образуют только операторы главной нити программы. В программах, созданных для MPP-систем, доля непараллельного кода образуют операторы, выполняющиеся всеми процессорами.

Таблица 1 наглядно демонстрирует ускорение работы программы в зависимости от доли непараллельного кода по закону Амдала.

Таблица 1.

P	Доля непараллельного кода, %				
	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

Количественно оценить величину f путем анализа текста программы фактически невозможно. Такая оценка может быть получена только путем реальных запусков программы на разном числе процессоров. Из формулы (2) также следует, что P-кратное ускорение работы программы может быть достигнуто только в том случае, когда доля непараллельного кода равна нулю, что на практике недостижимо.

Закон Амдала наглядно демонстрирует, что увеличение числа процессоров для увеличения производительности работы программы не всегда оправданно. 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 формула (2) не учитывает накладные расходы на обмены между процессорами, поэтому в реальной системе ускорение будет еще меньше.

Следует также заметить, что распараллеливание программы – это лишь одно из средств ускорения ее работы. Не меньший, а иногда и больший, эффект дает оптимизация программы на одном процессоре. Актуальность такой оптимизации сохраняется из-за большого разрыва в скоростях работы кэш-памяти процессора и оперативной памяти системы. Многие разработчики не уделяют этой проблеме должного внимания, тратя значительные усилия на распараллеливание заведомо неэффективных программ.

2.5. Проблемы оптимизации программ

Рассмотрим проблемы оптимизации, возникающие при разработке высокоэффективных программ для современных вычислительных систем на примере элементарной задачи перемножения двух квадратных матриц. Поскольку эта задача кажется разработчику очень простой, она особенно наглядно демонстрирует нетривиальность проблем, возникающих при создании высокоэффективных приложений.

Отслеживать производительность компьютера на реальном приложении, выполняющем перемножение матриц, достаточно

просто. Для перемножения двух квадратных матриц размерности $n \times n$ нужно выполнить $(2 \times n - 1) \times n \times n$ арифметических операций. Таким образом, для оценки производительности компьютера достаточно замерить время выполнения этих операций и разделить одну величину на другую.

Следует отметить, что фиксация одного только времени выполнения программы не совсем удобна, поскольку программа может выполняться при различных исходных данных (при различных размерностях матриц), и тогда очень сложно сопоставлять результаты тестов. Кроме того, само по себе время выполнения программы мало что говорит об эффективности работы системы в целом. Намного важнее знать, с какой производительностью работает вычислительная система на этой программе.

Надо также признать, что тест на перемножение матриц не совсем подходит для оценки производительности системы, поскольку использует только операции умножения и сложения, но результаты этого теста весьма близки к реальности и потому интересны для анализа.

Выполним перемножение квадратных матриц размерности 1000×1000 и сравним полученную производительность с расчетной пиковой производительностью процессоров, которую так любят указывать в рекламных материалах продавцы вычислительных узлов для суперкомпьютеров.

В 2003 году В.Н. Дацюк, А.А. Букатов и А.И. Жегуло из Ростовского государственного университета опубликовали свои результаты проведения данного теста на трех компьютерах различной архитектуры; мы провели тест на одном из вычислительных узлов суперкомпьютера ИПУ РАН. Таким образом, в нашем распоряжении оказались результаты решения одной и той же задачи на четырех процессорах различной архитектуры:

- DEC Alpha с тактовой частотой 667 МГц;
- Intel Pentium III с тактовой частотой 500 МГц;
- SUN UltraSPARC II с тактовой частотой 300 МГц;
- Intel Quad Core Xeon с тактовой частотой 2,33 ГГц.

Параметры конфигурации компьютеров особого значения не имеют, поскольку тестовое приложение решает очень простую задачу и практически не загружает память. Расчетная пиковая производительность участвовавших в тесте процессоров равна:

- Alpha – 1 гигафлопс;
- Pentium – 500 мегафлопс;
- Ultra – 800 мегафлопс;
- Xeon – 37 гигафлопс (9 гигафлопс на одном ядре).

Для решения задачи перемножения квадратных матриц ростовские исследователи создали простую программу на языке Fortran, однако мы для удобства реализовали тот же самый алгоритм на языке Си. Наше решение было правомерным, поскольку результаты тестирования не использовались для сравнения производительности различных архитектур.

Создадим текстовый файл *matrix.c*, содержащий следующий код на языке Си:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

double dseconds (void);

int main (int argc, char *argv[])
{
  int i, j, k;
  double s;
  const n = 1000; // Размерность матрицы
  double *a, *b, *c; // Указатели на массивы
  double time; // Таймер

// Создание матриц (выделение памяти)
  a = (double *) malloc (n * n * sizeof (double));
  b = (double *) malloc (n * n * sizeof (double));
  c = (double *) malloc (n * n * sizeof (double));

// Инициализация матриц
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
```

```
    {
      a[i*n + j] = (double) i + 1;
      b[i*n + j] = (double) 1 / (j + 1);
    }
  }

// Перемножение матриц
  time = dseconds ();
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      s = 0.0;
      for (k=0; k<n; k++)
        s = s + a[i*n + k] * b[k*n + j];
      c[i*n + j] = s;
    }
  }
  time = dseconds () - time;

// Вывод результатов
  printf ("Время работы: %f c\n", time);
  printf ("Производительность: %f мегафлопс\n",
    (double) (n - 1) * n * n / time / 1000000);
  return (0);
}

// Текущее время в микросекундах
double dseconds (void)
{
  struct timeval tv;
  struct timezone tz;
  double time;

  gettimeofday (&tv, &tz);
  time = (double) tv.tv_sec +
    (double) tv.tv_usec / 1000000;
  return (time);
}
```


Откомпилируем программу в стандартном режиме, без каких-либо флагов и опций компилятора

```
> gcc matrix.c
```

Результаты работы программы при стандартной компиляции, приведенные в таблице 2, слегка обескураживают, поскольку реальная вычислительная мощность всех процессоров оказывается в десятки и сотни раз ниже заявленной производителями.

Таблица 2.

Решение на одном процессоре	Alpha	Pentium	Ultra	Xeon (1 ядро)
Время работы, с	108,41	153,21	261,42	14,39
Производительность, мегафлопс	18,44	13,05	7,65	69,40

Мы сознательно не включали оптимизацию при компиляции, чтобы продемонстрировать, насколько код программы влияет на ее производительность и как компиляторы могут самостоятельно повышать производительность программ с помощью автоматической оптимизации.

Для начала попробуем оптимизировать код программы самостоятельно, и обратим внимание на его вычислительную часть

```
s = 0.0;
for (k=0; k<n; k++)
s = s + a[i*n + k] * b[k*n + j];
c[i*n + j] = s;
```

В данном случае переменная *s* используется для промежуточного хранения результатов перемножения элементов матриц и от нее можно легко избавиться. Перепишем вычислительную часть программы следующим образом

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

Снова откомпилируем программу в стандартном режиме, без каких-либо флагов и опций компилятора.

Результаты работы «оптимизированной» программы на одном процессорном ядре вычислительного узла суперкомпьютера ИПУ РАН:

- время работы программы – 20,75 с;
- производительность ядра – 48,12 мегафлопс.

Эти цифры наглядно демонстрируют, что более элегантный и лаконичный код не всегда означает более производительный. К сожалению, для других вычислительных систем нет данных о величине падения производительности в результате отказа от промежуточной переменной, ведущего в действительности к более активной работе с памятью.

Теперь откомпилируем программу в режиме полной автоматической оптимизации. Заметим, что на производительность автоматически оптимизированной программы наши предыдущие манипуляции с вычислительной частью влияния не окажут

```
> gcc -O3 matrix.c
```

Результаты работы автоматически оптимизированной программы, приведенные в таблице 3, показывают, что автоматическая оптимизация существенно ускорила решение задачи. При этом производительность по-прежнему осталась во много раз ниже пиковой. Заметим также, что на процессорах фирмы Intel прирост производительности за счет автоматической оптимизации оказался наибольшим.

Таблица 3.

Решение на одном процессоре	Alpha	Pentium	Ultra	Xeon (1 ядро)
Время работы, с	58,84	134,76	90,84	7,30
Производительность, мегафлопс	33,97	14,83	22,00	136,69

Теперь попробуем еще раз оптимизировать код программы и снова посмотрим на его вычислительную часть. Обратим внимание, что при суммировании произведений во втором множителе выполняется выборка элементов столбца матрицы *b (k, j)* из отстоящих далеко друг от друга элементов массива (массивы в Си размещаются в памяти по строкам). Это не позволяет буферизовать их в быстрой кэш-памяти. Объявим указатель на промежуточный массив **d* типа *double*, куда мы будем предварительно выбирать столбец матрицы *b*, выделим под него память объемом $n \times sizeof\ (double)$ и перепишем вычислительную часть программы следующим образом:

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

Аналогичные действия были выполнены ростовскими исследователями с элементами строки матрицы *a (i, k)*, поскольку массивы в языке Fortran размещаются в памяти по столбцам. Снова откомпилируем программу в режиме автоматической оптимизации.

Результаты работы автоматически оптимизированной программы с предварительной выборкой столбца (строки) матрицы, приведенные в таблице 4, демонстрируют заметный рост производительности на всех компьютерах кроме системы на базе процессора Pentium III, обладающего очень маленьким кэшем. Это подтверждает наше предположение о важности эффективного использования кэш-памяти.

Таблица 4.

Решение на одном процессоре	Alpha	Pentium	Ultra	Xeon (1 ядро)
Время работы, с	54,41	33,76	11,16	1,32
Производительность, мегафлопс	36,74	59,21	179,13	754,13

Именно на эффективное использование кэш-памяти были в свое время нацелены поставляемые с высокопроизводительными системами оптимизированные математические библиотеки базового набора подпрограмм линейной алгебры BLAS (англ. Basic Linear Algebra Subprograms), и это обстоятельство до сих пор является одним из основных аргументов в конкурентной борьбе платформ для высокопроизводительных систем. В настоящее время написаны и бесплатно распространяются самонастраивающиеся библиотеки BLAS, которые могут устанавливаться на любом компьютере и под любой операционной системой.

В библиотеке BLAS из пакета автоматически настраиваемых программ линейной алгебры ATLAS (англ. Automatically Tuned Linear Algebra Software), установленного на суперкомпьютере ИПУ РАН, есть готовая процедура перемножения матриц *cblas_dgemm*. Чтобы использовать ее в самом начале программы подключим заголовочный файл */opt/atlas/include/cblas.h*, содержащий объявления необходимых функций и констант

```
#include "/opt/atlas/include/cblas.h"
```

и заменим весь вычислительный блок на вызов одной единственной функции

```
cblas_dgemm (CblasRowMajor,
CblasNoTrans,
CblasNoTrans,
n, n, n, 1, a, n, b, n, 0, c, n);
```

Откомпилируем программу в режиме автоматической оптимизации с подключением необходимых библиотек из пакета ATLAS

```
> gcc -O3 matrix.c -L"/opt/atlas/lib" -lcblas -latlas
```

Результаты работы программы с использованием библиотеки BLAS из пакета ATLAS, приведенные в таблице 5, наглядно демонстрируют практическую ценность использования пакета ATLAS, эффективность которого не только не уступает, но и в некоторых случаях превосходит эффективность некоторых

коммерческих библиотек аналогичного применения (Sun Performance Library, Common Extended Math Library).

Таблица 5.

Решение на одном процессоре	Alpha	Pentium	Ultra	Xeon (1 ядро)
Время работы, с	5,36	2,72	2,24	0,29
Производительность, мегафлопс	372,9	734,8	894,0	3372,17

В итоге мы получили производительность, очень близкую к расчетной пиковой. Единственным исключением стал установленный на вычислительных узлах суперкомпьютера ИПУ РАН процессор Intel Quad Core Xeon, пиковая производительность одного ядра которого была достигнута лишь на треть. Это произошло по трем основным причинам:

1. Неполноценное использование ресурсов ядра (чтобы раскрыть свой потенциал, ядро должно работать в связке со своей парой).
2. Недостаточная загрузка процессора (слишком маленькая задача).
3. Несовершенная настройка библиотеки BLAS из пакета ATLAS (пакет ATLAS не самый быстрый для данного процессора).

Для проверки первого предположения запустим программу на четырех ядрах, т.е. целиком на одном процессоре, расчетная пиковая производительность которого составляет 37 гигафлопс. Чтобы скомпилировать программу с использованием возможностей SMP-архитектуры процессора, т.е. для запуска на всех четырех ядрах, наберите в консоли

```
> gcc -O3 matrix.c -L"/opt/atlas/lib" -lm -lpthread
-lptcblas -latlas
```

В результате производительность программы выросла до 17 гигафлопс, т.е. достигла величины около половины пиковой и в процентном отношении стала выше, чем при запуске на одном ядре. Справедливости ради следует отметить, что такая производительность получается не всегда, и многократные запуски

программы демонстрируют разную производительность системы (вплоть до 6 гигафлопс). Разброс в показаниях в очередной раз свидетельствует в пользу второго предположения об использовании в качестве теста слишком простой задачи, несоответствующей вычислительной мощности процессора.

Для проверки второго предположения увеличим размерность матрицы до 10000 × 10000. В результате разбросы в расчетах производительности при многократных запусках программы прекратились, а производительность системы выросла до 20 гигафлопс, т.е. в процентном отношении стала еще выше, чем при запуске на одном ядре. Это ускорение свидетельствует о росте производительности системы на больших задачах, соответствующих вычислительной мощности процессора.

Наконец, для проверки третьего предположения заменим пакет ATLAS более совершенными библиотеками подпрограмм линейной алгебры. На суперкомпьютере ИПУ РАН установлена математическая библиотека Intel MKL (англ. Math Kernel Library), включающая в себя собственную версию BLAS.

Чтобы использовать функцию *cblas_dgemm* из библиотеки Intel MKL, заменим в коде программы соответствующую строку подключения заголовочного файла

```
    #include
"/opt/intel/mkl/10.1.0.015/include/mkl_cblas.h"
```

и откомпилируем программу для работы на одном процессоре (для работы на одном ядре необходимо подключить другой набор библиотек с помощью опций *-lmkl_sequential -lmkl_intel_lp64 -lmkl_core -lm*)

```
    > gcc -O3 mmatrix.c
-L"/opt/intel/mkl/10.1.0.015/lib/em64t"
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5
-lpthread -lm
```

В результате производительность системы при работе на одном процессоре выросла до 32,6 гигафлопс, т.е. вплотную приблизилась к расчетному пиковому значению. Это подтверждает важность правильного выбора математического обеспечения для высокопроизводительных вычислений.

3. СУПЕРКОМПЬЮТЕР ИПУ РАН

Суперкомпьютер ИПУ РАН – кластерная система, ориентированная на высокопроизводительные вычисления и предназначенная для решения сложных научно-технических задач. Система с архитектурой x64 на базе процессоров Intel Xeon построена в 2008 году на базе промышленных серверов Sun Fire X4150.

В состав суперкомпьютера входят:

- 12 вычислительных узлов (всего 96 процессорных ядер) на базе серверов Sun Fire X4150, каждый из которых оснащен двумя процессорами Intel Quad Core Xeon E5345 с тактовой частотой 2,33 ГГц, памятью PC2 5300 667 МГц ECC Fully Buffered DDR2 общим объемом 4 Гб, локальным жестким диском, двумя интерфейсами InfiniBand SDR и четырьмя интерфейсами Gigabit Ethernet;
- управляющий узел на базе сервера Sun Fire X4150 с дисковым массивом на базе жестких дисков Seagate Savvio 10K.2 SCSI;
- высокоскоростная вычислительная сеть InfiniBand SDR на базе коммутатора Mellanox InfiniScale III MT47396 (24 порта InfiniBand 4X со скоростью передачи до 8 Гб/с);
- управляющая сеть Fast Ethernet (для управления серверами средствами Sun ILOM) на базе коммутатора D Link DES-1026G;
- транспортная сеть Gigabit Ethernet (для запуска расчетных задач, мониторинга работы программ и оборудования, соединения с управляющим узлом и сетевой файловой системой) на базе двух коммутаторов D Link DGS 1024D;
- программное обеспечение, предоставляющее полный набор средств разработки и запуска расчетных задач:
  - операционная система OpenSUSE 11.0 (установлена на всех узлах кластера);
  - коммутирующее программное обеспечение OFED 1.4;
  - среда параллельного программирования MVAPICH2, реализующая интерфейс передачи сообщений MPI версии 2.0;

– компиляторы Си, C++, Fortran и другие инструментальные средства разработки программ из фонда свободного программного обеспечения GNU;
– математические библиотеки ATLAS, Intel MKL 10.1;
– менеджер управления очередями и вычислительными ресурсами SLURM.

Пиковая производительность суперкомпьютера ИПУ РАН составляет 0,89 терафлопс. Реально достигнутый максимум на тестах LINPACK составляет 0,62 терафлопс. Суперкомпьютер позволяет решать задачи с разделением вычислительных ресурсов между несколькими пользователями.

3.1. Операционная система GNU/Linux

Суперкомпьютер ИПУ РАН работает под управлением операционной системы openSUSE 11.0 из семейства GNU/Linux.

История операционной системы GNU/Linux началась в 1983 году, когда американец Ричард Столлман (Richard Stallman) основал проект GNU (англ. GNU's Not UNIX – «GNU – это не UNIX» – традиционный для компьютерных специалистов рекурсивный акроним). Проект GNU задумывался как свободная переносимая, многозадачная и многопользовательская UNIX-подобная операционная система. К 1991 году большинство прикладных программ, необходимых для новой операционной системы, были созданы, но собственное ядро, управляющее всеми процессами на низком уровне и взаимодействующее с оборудованием компьютера, так и не заработало. Это обстоятельство привело к появлению первой версии ядра Linux финна Линуса Торвальдса (Linus Torvalds). Таким образом, операционная система GNU/Linux – это ядро Linux, работающее в окружении необходимых прикладных программ из проекта GNU.

Операционная система GNU/Linux развивается по модели так называемого «свободного программного обеспечения» (англ. Free Software). Такая модель предусматривает свободу использования программы любым способом, свободу изучения принципов ее работы, свободу адаптации программы для нужд пользователя, свободу распространения копий программы, а также

свободу изменения программы и публикации внесенных изменений.

Дистрибутивов GNU/Linux достаточно много, однако наиболее распространенных, таких как openSUSE, не более десяти. Дистрибутивы разрабатываются и распространяются различными дистрибьюторами, в роли которых могут выступать как крупные компании, так и сообщества добровольцев. Операционная система openSUSE разрабатывается и распространяется под патронажем компании Novell. Наработки проекта openSUSE используются для развития коммерческой операционной системы на базе ядра Linux – SUSE Enterprise Linux (SLES), под управлением которой, в частности, работают узлы самого быстрого суперкомпьютера 2004-2008 гг. IBM Blue Gene/L.

Доступ к суперкомпьютеру ИПУ РАН осуществляется по протоколу SSH (TCP-порт 22). Уточните адрес суперкомпьютера в локальной сети института у вашего администратора.

Для доступа к суперкомпьютеру из среды Windows рекомендуется пользоваться программой PuTTY. Работа с кластером ИПУ РАН ведется только в текстовом (консольном) режиме, который в среде MS Windows принято называть режимом командной строки.

Первое, что видит пользователь суперкомпьютера ИПУ РАН в случае удачного соединения, – это запрос учетного имени пользователя системы

```
login as:
```

В ответ на этот запрос необходимо ввести имя пользователя, под которым вы зарегистрированы в системе, после чего система запросит пароль

```
Using keyboard-interactive authentication.
Password:
```

После ввода пароля выводится строка приглашения

```
username@main:~>
```

Появление приглашения означает, что пользователь успешно вошел в систему, и теперь консоль может принимать и выполнять его команды.

В приведенном выше примере строка приглашения включает в себя имя пользователя *username*, имя управляющего узла суперкомпьютера *main* и текущий каталог ~. Во всех последующих примерах для обозначения строки приглашения мы для краткости будем использовать только символ закрывающей угловой скобки.

Следует отметить, что в любой UNIX-подобной системе, в отличие от систем семейства MS Windows, учитывается регистр вводимых символов. Поэтому вводить все команды и их параметры следует именно с учетом этого обстоятельства, каждый раз обращая внимание на строчные и прописные буквы.

После первого входа в систему желательно сменить пароль, выданный администратором, на ваш собственный, знать который уже не будет никто кроме вас. Наберите в консоли

```
> passwd
```

В ответ программа *passwd* в интерактивном режиме поможет вам сменить пароль

```
Changing password for username.
Старый пароль:
Новый пароль:
Повторите Новый пароль:
Пароль изменен.
```

Если пользователь выбрал новый пароль не очень удачно (слишком короткий или очень простой), программа выдаст соответствующее предупреждение, но все равно примет пароль и позволит входить с ним в систему.

Главные команды, о которых нужно знать каждому пользователю GNU/Linux, – это команды *man* и *info*. Вместе эти команды образуют большой и подробный справочник по операционной системе и ее возможностям. В качестве параметров при запуске этих команд необходимо указывать название программы, системного файла или просто ключевое слово по интересующей тематике, например

```
> man passwd
```

или


```
> info passwd
```

В ответ пользователь получит описание программы *passwd*. Вся имеющаяся в системе информация по возможности предоставляется на русском языке, однако, существенная ее часть существует только на английском. Поскольку информация обычно не помещается на одном экране, для передвижения по документу следует пользоваться клавишами <Page Up> и <Page Dn>, а также клавишей <Пробел>. Нажатие клавиши <Q> в любой момент приводит к выходу из программы и возврату в режим ввода команд. Для более подробного знакомства с возможностями программ *man* и *info* наберите в консоли

```
> man man
```

или

```
> info info
```

Программы *man* и *info* не дублируют, как может показаться на первый взгляд, а дополняют друг друга, потому что хранят разную информацию в разных форматах. Программа *man* – это традиционная для любых UNIX-подобных систем справочная система, в то время как программа *info* входит в проект GNU и работает по своим правилам. Внешняя схожесть этих программ – исключительная заслуга составителей документации.

Ниже приведен краткий список команд, которые могут понадобиться начинающему пользователю при первом знакомстве с операционной системой:

- *whoami* – сообщает ваше учетное имя в системе;
- *who* или *w* – сообщает учетные имена пользователей, работающих в данный момент в системе;
- *pwd* – сообщает название текущей папки;
- *ls -l* или *ll* – выдает список файлов и папок в текущей папке с указанием их атрибутов;
- *cd* <*имя папки*> – осуществляет смену текущей папки;
- *ps -e* – выдает список всех программ (процессов), выполняющихся в данный момент в системе;
- *logout* или *exit* – завершает текущий сеанс работы с системой.

В суперкомпьютере ИПУ РАН используется оболочка Bash. Оболочка (англ. shell) – это программа, организующая общение операционной системы с пользователем. Именно оболочка принимает и передает на запуск все команды пользователя, поэтому оболочку часто называют командным процессором. Строку приглашения операционной системы тоже выводит оболочка, она же ожидает ввода очередной команды. Каждый раз, когда пользователь входит в систему, он попадает в ее оболочку.

Оболочка Bash – это мощный командный процессор, обладающий собственным языком программирования. В этой оболочке имеется большой набор встроенных (внутренних) команд и операторов, список которых можно получить по команде *help*. Чтобы получить детальную информацию по конкретной команде или оператору, укажите его название в качестве аргумента, например

```
> help echo
```

Оболочка Bash обеспечивает выполнение запросов пользователя: находит и вызывает исполняемые файлы, организует ввод/вывод, отвечает за работу с переменными окружения, выполняет некоторые преобразования (подстановки) аргументов при запуске программ и т.п. Главное свойство оболочки, которое делает ее мощным инструментом пользователя, – это наличие собственного языка программирования. Оболочка также использует все программы, доступные пользователю, как базовые операции поддерживаемого ею языка, обеспечивает передачу им аргументов, а также передачу результатов их работы другим программам и пользователю.

Ниже приведен список основных сочетаний клавиш, позволяющих эффективно работать с командной строкой Bash:

- <Стрелка вправо> или <Ctrl>+<F> – перемещение вправо по командной строке в пределах уже набранной цепочки символов плюс один символ справа (место для ввода следующего символа);
- <Стрелка влево> или <Ctrl>+<B> – перемещение на один символ влево;
- <Esc>+<F> – перемещение на одно слово вправо;

- <Esc>+<B> – перемещение на одно слово влево;
- <Home> или <Ctrl>+<A> – перемещение в начало набранной цепочки символов;
- <End> или <Ctrl>+<E> – перемещение в начало/конец набранной цепочки символов;
- <Del> или <Ctrl>+<D> – удаление символа, на который указывает курсор;
- <Backspase> – удаление символа, находящегося слева от курсора;
- <Ctrl>+<K> – удаление правой части строки, начиная с символа, на который указывает курсор;
- <Ctrl>+<U> – удаление левой части строки, включая символ, находящийся слева от курсор;
- <Enter> или <Ctrl>+<M> – запуск на выполнение набранной цепочки символов;
- <Ctrl>+<L> – очищение экрана и помещение набранной цепочки символов в верхней строке экрана;
- <Ctrl>+<T> – замена местами символа, на который указывает курсор, и символа, находящегося слева от курсора, с последующим перемещением курсора на один символ вправо;
- <Esc>+<T> – замена местами слова, на которое указывает курсор, и слова, находящегося слева от курсора;
- <Ctrl>+<K> – вырезание и сохранение в буфере части набранной цепочки символов, находящейся справа от курсора;
- <Esc>+<D> – вырезание и сохранение в буфере правой части слова, на которое указывает курсор;
- <Esc>+<Del> – вырезание и сохранение в буфере левой части слова, на которое указывает курсор;
- <Ctrl>+<W> – вырезание и сохранение в буфере части набранной цепочки символов, находящейся слева от курсора до ближайшего пробела;
- <Ctrl>+<Y> – вставка из буфера строки;
- <Esc>+<C> – замена строчного символа, на который указывает курсор, на прописной символ и перемещение курсора на ближайший пробел справа;

– <Esc>+<U> – замена строчных символов слова, на которое указывает курсор, прописными символами и перемещение курсора на ближайший пробел справа;
– <Esc>+<L> – замена прописных символов слова, на которое указывает курсор, на строчные и перемещение курсора на ближайший пробел справа;
– <Shift>+<Page Up> или <Shift>+<Page Dn> – просмотр страниц экранного вывода (количество страниц зависит от размера видеопамяти);
– <Ctrl>+<C> – прерывание выполнения программы;
– <Ctrl>+<D> – выход из оболочки (завершение сеанса).

Оболочка Bash имеет встроенную подпрограмму, предназначенную для облегчения ввода команд в командной строке. Эта подпрограмма вызывается по клавише <Tab> после того, как уже введено некоторое число символов. Если эти символы являются первыми символами в названии одной из стандартных команд, известных оболочке, то возможны разные варианты дальнейшего развития событий:

– если по введенным первым символам команда определяется однозначно, оболочка добавит окончание названия команды в командную строку;
– если однозначно восстановить имя команды по введенным первым символам невозможно, оболочка выдаст список всех подходящих вариантов продолжения для того, чтобы пользователь мог ввести оставшиеся символы самостоятельно, пользуясь выведенным списком как подсказкой;
– если среди введенных символов уже есть пробел, оболочка решит, что вы ищете имя файла, который должен передаваться как параметр, и выдаст в качестве подсказки список файлов текущего каталога, начинающихся с символов, следующих за пробелом.

Аналогичным образом можно получить список переменных окружения, если вместо клавиши <Tab> нажать комбинацию клавиш <Esc>+<$>.

Оболочка Bash запоминает крайнюю тысячу набранных последовательностей символов и позволяет вызывать их путем

выбора из списка, называемого историей ввода. Историю ввода можно просмотреть с помощью команды *history* (и воспользоваться упомянутыми ранее комбинациями клавиш <Shift>+<Page Up> и <Shift>+<Page Dn> для просмотра результата). История ввода сохраняется в файле, полное имя которого хранится в переменной окружения *HISTFILE*, (обычно *~/.bash_history*). Для работы с историей ввода в оболочке Bash используются следующие комбинации клавиш:

– <Стрелка вверх> или <Ctrl>+<P> – переход к предыдущей последовательности символов в списке;
– <Стрелка вниз> или <Ctrl>+<N> – переход к следующей последовательности символов в списке;
– <Page Up> – переход к самой первой последовательности символов в списке;
– <!>, <N> – выполнение (без нажатия клавиши <Enter>) N-ой последовательности символов из списка, считая от начала;
– <!>, <->, <N> – выполнение (без нажатия клавиши <Enter>) N-ой последовательности символов из списка, считая от конца;
– <!>, <строка> – выполнение первой найденной последовательности символов из списка, начинающейся на введенную строку (движение по списку осуществляется от конца к началу);
– <Ctrl>+<O> – аналогично нажатию клавиши <Enter>, но с последующим отображением очередной последовательности символов из списка.

3.2. Инструментарий разработчика

Инструментарий разработчика, созданный в рамках проекта GNU (англ. GNU Toolchain), является стандартным средством разработки программ в любой операционной системе на базе ядра Linux. Основу этого инструментария составляет набор GNU-компиляторов (англ. GCC, GNU Compiler Collection, ранее GNU C Compiler) для различных языков программирования, в том числе Си, C++ и Fortran. В набор GNU-компиляторов входят также компиляторы Java, Objective-C и Ada, однако эти языки

программирования не используются для высокопроизводительных вычислений.

Компиляция программ осуществляется с помощью утилиты *gcc*, по традиции называемой компилятором, но фактически являющейся интерфейсом целой системы компиляции, созданной проектом GNU. Выбор того или иного языка программирования обусловлен скорее личными предпочтениями разработчика и особенностями реализуемого алгоритма.

Инструментарий разработчика суперкомпьютера ИПУ РАН поддерживает языки программирования Си, C++ и Fortran, используемые во всех современных реализациях MPI для создания высокопроизводительных параллельных программ.

Создадим и запустим простую программу на Си. По сложившейся традиции первая программа будет выводить в консоль приветствие «Здравствуй, мир!».

Файлы с кодами программ – это обычные текстовые файлы, создавать их можно с помощью любого текстового редактора, в т.ч. с помощью традиционного для GNU/Linux консольного текстового редактора *vi*.

Создадим в домашней папке текстовый файл *hello.c*, содержащий следующий код на языке Си

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  printf ("Здравствуй, мир!\n");
  return (0);
}
```

Для компиляции программы наберите в консоли

```
> gcc hello.c
```

Если все сделано правильно, в домашней папке появится новый исполняемый файл под названием *a.out*. Чтобы запустить его наберите в консоли

```
> ./a.out
```

В ответ вы увидите следующее


```
Здравствуй, мир!
```

Компилятор *gcc* по умолчанию присваивает всем исполняемым файлам имя *a.out*. С помощью флага *-o* можно указать имя исполняемого файла самостоятельно. Наберите в консоли

```
> gcc hello.c –o hello
```

В результате в текущей папке появится исполняемый файл с именем *hello*. Чтобы запустить его наберите в консоли

```
> ./hello
```

Точка и слеш перед именем исполняемого файла означают, что исполняемый файл размещен в текущей папке. Без указания пути к исполняемому файлу запускаются только программы, размещенные в системных папках и являющиеся частью операционной системы. Для запуска всех остальных программ, в т.ч. пользовательских, необходимо указывать путь к исполняемому файлу (абсолютный или относительно текущей папки), например

```
> /home/nfs/username/hello
```

или

```
> ~/hello
```

Работа системы компиляции состоит из четырех последовательно выполняемых этапов:

1. Препроцессинг (англ. preprocessing).
2. Компиляция (англ. compilation).
3. Ассемблирование (англ. assembly).
4. Компоновка (англ. linking).

Препроцессинг подключает к программному коду содержимое заголовочных файлов, указанных в директивах *#include*, и заменяет макросы, определенные директивами *#define*, соответствующим программным кодом.

Во время компиляции происходит превращение исходного кода программы на языке Си в промежуточный код на Ассемблере. Этот промежуточный шаг очень важен для дальнейшей работы компилятора, потому что именно во время компиляции происходит детальный разбор исходного кода и поиск синтакси-

ческих ошибок. Возможно, поэтому, а также из-за некоторой исторически сложившейся терминологической путаницы, некоторые ошибочно полагают, что компиляция – это единственное, чем занимается компилятор (система компиляции).

Во время ассемблирования происходит превращение ассемблерного кода программы в объектный код, очень близкий к набору машинных команд. Результат сохраняется в объектных файлах.

На этапе компоновки происходит создание исполняемого файла путем связывания вызовов пользовательских и библиотечных функций с их объектным кодом, хранящимся в различных объектных файлах.

Для дальнейшей демонстрации работы инструментария разработчика GNU напишем небольшую программу-калькулятор.

Создадим в домашней папке текстовый файл *calc.c*, содержащий следующий код на языке Си

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  float x, y, z;
  char o;

  printf ("X = ");
  scanf ("%f", &x);
  printf ("Y = ");
  scanf ("%f", &y);
  printf ("Операция ( + - * / ): ");
  while ((o = getchar ()) != EOF)
  if (o == '+')
  {
    z = x + y; break;
  }
  else if (o == '-')
  {
    z = x - y; break;
  }
  else if (o == '*')
```



```
  {
    z = x * y; break;
  }
  else if (o == '/')
  {
    z = x / y; break;
  }
  printf ("X %c Y = %f\n", o, z);
  return (0);
}
```

Программа запрашивает у пользователя два числа и арифметическое действие, после чего печатает в консоль результат вычислений.

Флаг *-E* прерывает работу системы компиляции после завершения препроцессинга. Наберите в консоли

```
> gcc -E calc.c -o calc.i
```

Название конечного файла *calc.i* желательно указывать, потому что результаты препроцессинга по умолчанию печатаются в консоль. В файл *calc.i* будет добавлен код заголовочного файла *stdio.h*, указанного в директиве *#include*. Файл *stdio.h* содержит объявления встречающихся в программе функций ввода-вывода *printf*, *scanf* и *getchar*. В файл *calc.i* также будут добавлены некоторые теги, указывающие компилятору на способ связи с объявленными функциями. Код программы из файла *calc.c* без изменений будет добавлен в конец файла *calc.i*.

Описания стандартных функций и заголовочных файлов для всех языков программирования из инструментария разработчика, точно также как и описания всех стандартных программ операционной системы, можно получить с помощью команд *man* и *info*. Наберите в консоли

```
> man printf
```

или

```
> info stdio.h
```

Флаг *-S* прерывает работу системы компиляции после завершения компиляции программы в ассемблерный код. Наберите в консоли

```
> gcc –S calc.c
```

Название конечного файла можно не указывать, потому что компилятор *gcc* самостоятельно создаст объектный файл *calc.s*, т.е. назовет его также как и файл с исходным кодом, но поставит стандартное расширение для файла на Ассемблере.

Флаг *-c* прерывает работу системы компиляции после завершения ассемблирования. В результате получается объектный файл, состоящий из набора машинных команд, но без связи вызываемых функций с их определениями в подключаемых библиотеках. Наберите в консоли

```
> gcc –c calc.c
```

Название конечного файла можно не указывать, потому что компилятор *gcc* самостоятельно создаст объектный файл *calc.o*, т.е. назовет его также как и файл с исходным кодом, но поставит стандартное расширение для объектного файла.

Флаг *-x* сообщает компилятору с какого этапа следует начать обработку файла. Например, для файла *calc.i* (полученного ранее с помощью флага *-E*), наберите в консоли

```
> gcc –x cpp–output –c calc.i
```

В данном случае параметры *-x cpp-output* сообщает компилятору, что файл *calc.i* содержит обработанный препроцессором исходный код на языке Си (если указаны параметры *-x c++ cpp output* – на языке C++), поэтому компилятор начнет обработку файла сразу с этапа компиляции. Следует отметить, что расширение *.i* сообщает компилятору ту же информацию, что и параметры -x cpp output.

Компилятор *gcc* самостоятельно распознает языки программирования и этапы компиляции программы, основываясь на расширениях имен файлов, указанных пользователем. В частности, по умолчанию считается, что файлы с расширениями *.c* содержат исходный код на языке Си, файлы с расширениями *.C*, *.cc*, *.cp* или *.cpp* содержат исходный код на языке C++, файлы с

расширениями *.i* и *.ii* содержат обработанный препроцессором исходный код на языках Си и C++ соответственно и т. д. Файлы с неизвестными расширениями считаются объектными файлами и сразу попадают на этап компоновки.

Компилятор *gcc* позволяет проводить частичную обработку файла путем указания начального этапа с помощью флага *-x* и конечного этапа с помощью флагов *-c*, *-S*, и *-E*. Однако таким образом нельзя изменить последовательность прохождения этапов препроцессинга, компиляции, ассемблирования и компоновки. В частности, одновременное использование опций *-x -cpp-output -E* не запустит ни один из этапов.

К сожалению, реальные программы очень редко состоят из одного файла. Как правило, исходных файлов всегда несколько, причем в некоторых случаях программу приходится компоновать из нескольких частей, написанных на разных языках программирования. В таких случаях принято говорить о программе как о проекте. Компилятор *gcc* используется для последовательной обработки файлов проекта и последующей компоновки объектных файлов в исполняемый файл программы.

Добавим в программу-калькулятор операцию возведения в степень и разобьем код программы на несколько файлов:

- *calculator.c* – главное тело программы;
- *calculate.c* – расчетная функция;
- *calculate.h* – заголовочный файл.

Создадим в домашней папке текстовый файл *calculator.c*, содержащий следующий код на языке Си

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

int main (int argc, char *argv[])
{
  float x, y, z;
  char o;

  printf ("X = ");
  scanf ("%f", &x);
  printf ("Y = ");
  scanf ("%f", &y);
```

```
  printf ("Операция ( + - * / ^ ): ");
  while ((o = getchar ()) != EOF)
    if (o == '+' || o == '-' ||
      o == '*' || o == '/' || o == '^')
    {
      z = calculate (x, y, o); break;
    }
  printf ("X %c Y = %f\n", o, z);
  return (0);
}
```

Создадим в домашней папке текстовый файл *calculate.c*, содержащий следующий код на языке Си

```
#include <math.h>
#include "calculate.h"

float calculate (float x, float y, char o)
{
  float z;

  if (o == '+')
    z = x + y;
  else if (o == '-')
    z = x - y;
  else if (o == '*')
    z = x * y;
  else if (o == '/')
    z = x / y;
  else if (o == '^')
    z = powf (x, y);
  return (z);
}
```

Создадим в домашней папке текстовый файл *calculate.h*, содержащий следующий код на языке Си

```
float calculate (float x, float y, char o);
```

Создадим объектные файлы проекта *calculator.o* и *calculate.o*. Наберите в консоли


```
> gcc -c calculator.c
> gcc -c calculate.c
```

Наконец, скомпонуем исполняемый файл проекта *calc*. Наберите в консоли

```
> gcc calculator.o calculate.o -lm -o calc
```

Флаг *-lm* подключает в проект на этапе компоновки стандартную библиотеку математических функций *libm.so*, содержащую объектный код функции возведения в степень *powf*, вызываемой пользовательской функцией *calculate ()* и объявленной в заголовочном файле *math.h*. Без подключения этой библиотеки исполняемый файл не будет скомпонован и программа не заработает.

На суперкомпьютере ИПУ РАН объектные файлы стандартных библиотек компилятора располагаются в папке */usr/lib64*. Объектные файлы библиотеки ATLAS располагаются в папке */opt/atlas/lib* (соответствующие заголовочные файлы – в папке */opt/atlas/include*). Объектные файлы библиотеки Intel MKL располагаются в папке */opt/intel/mkl/10.1.0.015/lib/em64t* (соответствующие заголовочные файлы – в папке */opt/intel/mkl/10.1.0.015/include*).

Файлы библиотек с расширением *.a* называются статическими библиотеками. При компоновке объектный код функций из статических библиотек включается в код исполняемого файла. Файлы библиотек с расширением *.so* называются динамическими библиотеками. При компоновке в исполняемом файле размещаются только ссылки на динамические библиотеки, реального включения объектного кода вызываемых функций не происходит. Использование динамических библиотек ухудшает переносимость исполняемого кода, но экономит ресурсы системы, позволяя нескольким программам использовать одну и ту же библиотеку, загруженную в память.

Стандартные имена файлов библиотек состоят из префикса *lib*, названия библиотеки и расширения *.a* или *.so*. В параметрах запуска компилятора *gcc* префикс *lib* заменяется префиксом *-l*, а расширение файла не указывается. Таким образом, файл *libm.so* в параметрах запуска компилятора значится как *-lm*.

В системе имеются статические и динамические версии всех стандартных библиотек. По умолчанию компилятор *gcc* подключает к исполняемому файлу динамические библиотеки. Для компоновки исполняемого файла с использованием только статических библиотек используется параметр *-static*. Однако следует помнить, что некоторые библиотеки сами используют функции из других библиотек и при статической компоновке требуют явного указания всех используемых библиотек в параметрах запуска компилятора, что может представлять определенную сложность для неопытного разработчика.

Стандартные функции языка Си, такие как *printf*, находятся в библиотеке *libc.so*, которая автоматически участвует в компоновке всех программ, написанных на языке Си, и не требует упоминания в параметрах компилятора.

Для управления большим количеством файлов и компиляции больших программных проектов используется утилита *make* из инструментария разработчика GNU.

Создадим в домашней папке текстовый файл *Makefile*, содержащий следующие строки

```
calc: calculator.o calculate.o
   gcc calculator.o calculate.o –lm –o calc
calculator.o: calculator.c calculate.h
   gcc –c calculator.c
calculate.o: calculate.c calculate.h
   gcc –c calculate.c
clean:
   rm –f calc calculator.o calculate.o
install:
   cp calc ~/calc
uninstall:
   rm –f ~/calc
```

Обратите внимание на отступ слева, который должен быть сделан только с помощью символа табуляции.

Файл *Makefile* – это список действий (макросов, правил), которые утилита *make* может проделывать над многочисленными файлами программного проекта.

Для компиляции проекта наберите в консоли


```
> make
```

или

```
> make calc
```

Для копирования исполняемого файла в домашний каталог наберите в консоли

```
> make install
```

Для удаления исполняемого файла из домашнего каталога наберите в консоли

```
> make uninstall
```

Утилита *make* самостоятельно определяет, какие из файлов проекта требуют компиляции и, в случае необходимости, производит над ними действия, указанные в файле *Makefile*.

Для компиляции параллельных программ, использующих реализацию интерфейса MPI MVAPICH2, используется утилита *mpicc*. Утилита *mpicc* не является самостоятельным компилятором – это всего лишь небольшой скрипт оболочки Bash, запускающий компилятор *gcc* с параметрами подключения библиотеки функций MPI. Таким образом, компилятор *mpicc* использует тот же набор флагов и опций, что и система компиляторов *gcc*.

Чтобы скомпилировать параллельную программу *program.c* наберите в консоли

> *mpicc program.c*

Для запуска параллельных программ используется менеджер управления очередями и вычислительными ресурсами SLURM. Параллельная программа может быть запущена на суперкомпьютере ИПУ РАН только после постановки в очередь с помощью утилиты *srun*. Наберите в консоли

```
> srun –n2 a.out
```

Параметр *-n2* указывает менеджеру управления очередями, что программа должна быть запущена на двух вычислительных узлах, в результате чего будет создано два параллельных процесса. Количество вычислительных узлов для запуска параллельных программ на суперкомпьютере ИПУ РАН может изменяться в пределах от 2 до 96.

После постановки в очередь программа ждет освобождения вычислительных узлов, которые могут быть заняты другими программами. При наличии необходимого количества свободных вычислительных узлов программа, стоящая в очереди первой, немедленно уходит на исполнение.

Для постановки программы в очередь на исполнение и немедленный возврат в командную строку наберите в консоли

```
> srun -n2 a.out &
```

Для просмотра списка очередей используется утилита *sinfo*. По умолчанию все программы ставятся в очередь *debug*.

Для просмотра выполняемых программ (этапов программы) в очереди используется утилита *squeue*.

Утилита *srun* по умолчанию запускает экземпляр программы на каждом вычислительном узле (ядре процессора).

4. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА MPI

Данный раздел посвящен написанию параллельных программ с использованием программного интерфейса передачи сообщений MPI (англ. Message Passing Interface). На сегодняшний день MPI является наиболее распространенной технологией параллельного программирования для суперкомпьютеров. Большинство реализаций этого интерфейса поддерживают стандарты MPI 1.1 и MPI-2.0. Стандарт MPI-2.0 появился в 1997 году и существенно расширил возможности стандарта 1.1, но в течение долгого времени не был широко распространен. Тем не менее, для написания параллельных программ на суперкомпьютере ИПУ РАН был выбран именно стандарт MPI-2.0. Большая часть аспектов, которые будут описаны в данном разделе, является идентичной для обоих стандартов, поэтому далее будет предполагаться, что мы имеем дело со стандартом MPI-2.0. Если будет необходимо привести какие-то исключения для стандарта 1.1, то это будет оговорено дополнительно.

Существуют реализации MPI для языков Си, C++ и Fortran. Примеры и описания всех функций даны с использованием языка Си. Однако пользователи суперкомпьютера ИПУ РАН

имеют возможность компилировать и запускать программы также на языках C++ и Fortran. Основные идеи и правила создания программ с использованием интерфейса MPI схожи для всех языков. Дополнительную информацию о стандарте MPI можно получить в Интернете на сайте MPI Forum.

Главное отличие параллельной программы, использующей интерфейс MPI, от традиционной непараллельной программы состоит в том, что в параллельной программе в один момент времени могут выполняться несколько различных операций на разных процессорах/ядрах. Разработчик должен создать программный код для каждого ядра. Стандарт MPI предполагает возможность разработки приложений с уникальным кодом для каждой из параллельных ветвей алгоритма, т.е. со своим кодом для каждого ядра. Однако в большинстве случаев для всех ядер, участвующих в работе приложения, используется один и тот же программный код.

Таким образом, для создания параллельной программы достаточно написать программный код, в который будет включена библиотека MPI и реализованы все необходимые механизмы взаимодействия параллельных ветвей программы, откомпилировать этот программный код и обеспечить запуск исполняемого файла на нескольких ядрах.

Прежде чем перейти к описанию принципов написания программ введем некоторые обозначения:

- *процесс* – это исполнение программы на одном процессоре, на котором установлен MPI, безотносительно к тому, содержит ли эта программа внутри параллельные ветви или операции ввода/вывода или просто последовательный программный код;
- *группа* – это совокупность процессов, каждый из которых имеет внутри группы уникальное имя, используемое для взаимодействия с другими процессами группы посредством коммуникатора группы;
- *коммуникатор группы* – интерфейс синхронизации и обмена данными между процессами.

Коммуникатор выступает для прикладной группы как коммуникационная среда взаимодействия. Коммуникаторы бывают

внутригрупповыми (intra) и межгрупповыми (inter). Коммуникатор определяет контекст передачи сообщений. Сообщения, использующие разные коммуникаторы, не оказывают влияния друг на друга и не взаимодействуют друг с другом. Каждая группа процессов использует отдельный коммуникатор. Если в группе *n* процессов, то процессы внутри группы имеют номера от *0* до *n – 1*.

С точки зрения операционной системы процесс рассматривается как отдельное приложение, не взаимодействующее с другими процессами (приложениями). С точки зрения разработчика процесс – это одна из ветвей параллельной программы, которой необходимо обеспечить коммуникацию с другими ветвями (процессами) параллельного приложения. Поскольку зачастую все процессы одного приложения выполняют один и тот же код, то приходится реализовывать взаимодействие этого кода с самим собой с учетом его выполнения в разных процессах. Коммуникатор группы как раз и является той абстракцией, которая обеспечивает взаимодействие процессов между собой. При этом разработчика не должно интересовать, каким способом передается информация между процессами: заботу о передаче сообщений между процессами берет на себя интерфейс MPI.

4.1. Инициализация MPI

Любая программа на языке Си, использующая MPI, должна включать в себя заголовочный файл, в котором определены все функции, переменные и константы MPI. Для подключения библиотеки MPI необходимо внести в программу следующую строку

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

Директива *#include* подключает заголовочный файл *mpi.h* библиотеки MPI. Следует отметить, что при написании программ на языке C++ и использовании стандарта MPI-2.0 у пользователей могут возникнуть проблемы с подключением библиотеки MPI. На этапе компиляции программы компилятор может вывести следующее сообщение об ошибке


```
SEEK_SET is #defined but must not be for the C++
binding of MPI
```

Причиной этого является то, что в заголовочном файле стандартной библиотеки ввода/вывода *stdio.h* и в заголовочном файле *mpi.h* для C++ объявлены одни и те же константы *SEEK_SET*, *SEEK_CUR* и *SEEK_END*, что является ошибкой реализации MPI-2.0.

Для решения проблемы необходимо при запуске компилятора использовать флаг *DMPICH_IGNORE_CXX_SEEK* или подключать библиотеку MPI следующим образом

```
#undef SEEK_SET
#undef SEEK_END
#undef SEEK_CUR
#include "mpi.h"
```

После запуска параллельная программа должна инициализировать свою параллельную часть, т.е. подготовиться к работе с другими ветвями параллельного приложения. Для этого используется функция *MPI_Init*.

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

Параметры функции:

- *argc* – число аргументов в командной строке, вызвавшей программу;
- *argv* – указатель на массив символьных строк, содержащих эти аргументы.

Параметры обычно соответствуют аналогичным параметрам исходной программы, но не обязательно.

Функция *MPI_Init* обеспечивает инициализацию параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза, а если интерфейс MPI уже был инициализирован, то никакие действия не выполняются, а происходит немедленный выход из функции. Все оставшиеся функции MPI могут быть вызваны только после вызова *MPI_Init*.

Для успешного выполнения функции *MPI_Init* необходимо, чтобы приложение имело информацию, необходимую для создания группы процессоров, которые будут участвовать в выполнении приложения. Обычно такая информация передается в приложение при помощи специальных средств для запуска программы, и разработчику не нужно заботиться об этом.

Для завершения параллельной части приложения используется функция *MPI_Finalize*.

```
int MPI_Finalize (void);
```

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

Общая структура программы, использующей интерфейс MPI, на языке Си выглядит следующим образом

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

Иногда приложению необходимо выяснить, была ли уже вызвана функция *MPI_Init*. Для этого используется функция *MPI_Initialized*.

```
int MPI_Initialized  (
  int *flag
);
```

Функция *MPI_Initialized* возвращает значение в переменную *flag*. Если *MPI_Init* уже была вызвана, то *flag* имеет ненулевое значение, в противном случае *flag* равна нулю. В стандарте MPI 1.1 это единственная функция, которую можно вызывать до *MPI_Init*. Вызов функции *MPI_Finalize* не влияет на поведение *MPI_Initialized*.

Приведенная ниже программа демонстрирует работу функций инициализации интерфейса MPI

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

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

  MPI_Initialized (&flag);
  printf ("MPI_Initialized вернула %d\n", flag);
  MPI_Init (&argc, &argv);
  MPI_Initialized (&flag);
  printf ("MPI_Initialized вернула %d\n", flag);
  MPI_Finalize ();
  return (0);
}
```

Каждый из процессов этой программы печатает в консоль сначала информацию о том, что функция *MPI_Initialized* вернула значение 0, а затем 1. Обратите внимание, что сообщения каждый раз выводятся в хаотичном порядке. Это связано с тем, что без использования функций синхронизации процессы работают с разной скоростью.

Для работы над решением общей вычислительной задачи группе процессов необходимо выяснить, над какой частью задачи должен работать каждый процесс. Для этого каждый процесс должен уметь идентифицировать себя в группе. Работа с группами процессов будет рассмотрена далее, поэтому пока ограничимся лишь общим представлением о группах процессов.

Каждый процесс характеризуется уникальной парой: группа и номер процесса в группе. Каждый из процессов может принадлежать одной или нескольким группам, и в каждой из них иметь свой номер. После выполнения функции *MPI_Init* создается базовая группа с коммуникатором *MPI_COMM_WORLD*, содержащая все процессы приложения. Затем в рамках этой группы могут создаваться новые группы, которые будут включать часть процессов. Процессы могут общаться в рамках одной

группы. Для самоидентификации процесса в группе служат функции *MPI_Comm_size* и *MPI_Comm_rank*.

```
int MPI_Comm_size  (
  MPI_Comm comm,
  int *size
);
```

Параметры функции:

- *comm* – коммуникатор группы;
- *size* – размер группы.

Функция *MPI_Comm_size* определяет число процессов в группе.

```
int MPI_Comm_rank (
  MPI_Comm comm,
  int *rank
);
```

Параметры функции:

- *comm* – коммуникатор группы;
- *rank* – номер вызывающего процесса в группе.

Функция *MPI_Comm_rank* определяет номер вызывающего процесса в группе. Значение, возвращаемое по адресу *&rank*, лежит в диапазоне от *0* до *size – 1*.

Во время выполнения параллельной программы может возникнуть необходимость экстренно завершить работу всей группы процессов. Для этих целей используется функция *MPI_Abort*.

```
int MPI_Abort (
  MPI_Comm comm,
  int errorcode
);
```

Параметры функции:

- *comm* – коммуникатор группы;
- *errorcode* – код ошибки, с которой завершится процесс.

Функция *MPI_Abort* прерывает процессы, ассоциированные с коммуникатором *comm*, и возвращает управление внешней среде. Прерванный процесс всегда завершает свою работу с ошибкой, номер которой указан в параметре *errorcode*. Процесс,

вызывающий функцию *MPI_Abort*, должен быть членом группы с указанным коммуникатором. Если указанного коммуникатора группы не существует или вызывающий процесс не является ее членом, то прерывается вся базовая группа процессов *MPI_COMM_WORLD*.

Приведенная ниже программа демонстрирует работу функций идентификации и прерывания параллельных процессов

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

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

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  printf ("Размер группы %d, мой номер %d\n",
    size, me);
  printf ("Вызываю MPI_Abort\n");
  MPI_Abort (MPI_COMM_WORLD, 911);
  printf ("MPI_Abort была вызвана\n");
  MPI_Finalize();
  return (0);
}
```

Во время выполнения этой программы каждый из процессов печатает в консоль размер своей группы и свой номер в группе. Обратите внимание, что процессы выводят сообщения не по порядку своих номеров. Ни один из процессов не сообщает о завершении вызова функции *MPI_Abort*.

4.2. Прием/отправка сообщений с блокировкой

Помимо инициализации и идентификации все параллельные процессы должны обмениваться информацией друг с другом. При этом обмен информацией должен происходить не только до начала вычислений, но и в процессе вычислений, и после их завершения.

Обеспечение информационного обмена между процессами – основная задача интерфейса MPI. Для разных видов коммуникации MPI предлагает множество различных функций.

Во время коммуникации типа «точка-точка» с блокировкой соответствующие функции останавливают вызывающий процесс до тех пор, пока не будет завершена передача сообщения.

Для отправки сообщений используется функция *MPI_Send*.

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

Параметры функции:

- *buf* – отправляемое сообщение;
- *count* – число элементов в отправляемом сообщении;
- *datatype* – тип элементов;
- *dest* – номер процесса-получателя;
- *msgtag* – идентификатор сообщения;
- *comm* – коммуникатор группы.

Функция *MPI_Send* осуществляет блокирующую отправку сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest* в группе с коммуникатором *comm*. Все элементы сообщения содержатся в буфере *buf*. Значение *count* может быть нулевым. Тип передаваемых элементов *datatype* должен указываться с помощью предопределенных констант из библиотеки типов.

Например, вызов функции *MPI_Send*, отправляющей три элемента массива целого типа *int* нулевому процессу в группе с коммуникатором *MPI_COMM_WORLD* выглядит следующим образом

```
MPI_Send (&buf, 3, MPI_INT, 0, 1, MPI_COMM_WORLD);
```

В примере использован тип данных *MPI_INT*. Он соответствует типу *int* в языке Си (подробнее типы данных рассмотрены

далее). Значение идентификатора сообщения выбрано произвольным образом. Процессу разрешается отправлять сообщения самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из функции *MPI_Send*. Выбор способа реализации такой гарантии: копирование сообщения в промежуточный буфер или непосредственная передача сообщения процессу *dest*, остается за конкретной реализацией интерфейса MPI.

Следует отметить, что возврат из функции *MPI_Send* не означает, что сообщение передано процессу *dest* или уже покинуло вычислительный узел, на котором выполняется процесс-отправитель, вызвавший функцию *MPI_Send*.

Для приема сообщения используется функция *MPI_Recv*:

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

Параметры функции:

- *buf* – принимаемое сообщение;
- *count* – максимальное число элементов в принимаемом сообщении;
- *datatype* – тип элементов;
- *source* – номер процесса-отправителя;
- *msgtag* – идентификатор сообщения;
- *comm* – коммуникатор группы;
- *status* – параметры принятого сообщения.

Функция *MPI_Recv* осуществляет блокирующий прием сообщения с идентификатором *msgtag*, состоящего максимум из *count* элементов типа *datatype*, от процесса с номером *dest* в группе с коммуникатором *comm*.

Число элементов в принимаемом сообщении не должно превышать значения *count*. Если число элементов в принимаемом сообщении меньше значения *count*, то гарантируется, что в буфере *buf* будут размещены только принимаемые элементы, т.е. функция *MPI_Recv* не очищает буфер. Для определения точного числа элементов в принимаемом сообщении можно воспользоваться функцией *MPI_Probe*.

Блокировка гарантирует, что после возврата из функции *MPI_Recv* все элементы сообщения уже приняты и размещены в буфере *buf*.

В качестве номера процесса-отправителя можно указать *MPI_ANY_SOURCE* – признак готовности принять сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать *MPI_ANY_TAG* – признак готовности принять сообщение с любым идентификатором.

Если процесс-отправитель успел отправить несколько сообщений процессу-получателю, то однократный вызов функции *MPI_Recv* примет только одно сообщение, отправленное раньше других.

Например, вызов функции *MPI_Recv*, принимающей три элемента массива целого типа *int* от первого процесса в группе с коммуникатором *MPI_COMM_WORLD* выглядит следующим образом

```
MPI_Recv (&buf, 3, MPI_INT, 1,
  MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

Структура *status* типа *MPI_Status* (подробно рассматривается далее) содержит параметры принятого сообщения.

Пользуясь функциями отправки/получения сообщений с блокировкой, следует соблюдать осторожность. Как отмечалось ранее, в зависимости от реализации MPI сообщения отправляются напрямую или записываются в промежуточный буфер. Поэтому в некоторых реализациях MPI процессы могут блокировать друг друга. Если двум процессам необходимо одновременно отправить друг другу некоторый объем данных, и при этом сообщения пересылаются напрямую, минуя буфер, то оба процесса одновременно дойдут до функции отправки сообщения и заблокируются на ней, ожидая пока получатель дойдет до

функции получения сообщения. Однако ни один из процессов никогда не дойдет до функции получения сообщения, поскольку каждый процесс заблокирован на этапе отправки, программа остановится. В некоторых реализациях MPI подобные ситуации отслеживаются с помощью промежуточного буфера, но разработчик должен самостоятельно исключать взаимную блокировку в коде программы для обеспечения полноценной переносимости кода.

Функция *MPI_Sendrecv* объединяет функционал приема и отправки сообщений.

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

Параметры функции:

- *sbuf* – отправляемое сообщение;
- *scount* – число элементов в отправляемом сообщении;
- *stype* – тип элементов в отправляемом сообщении;
- *dest* – номер процесса-получателя;
- *stag* – идентификатор отправляемого сообщения;
- *rbuf* – принимаемое сообщение;
- *rcount* – максимальное число элементов в принимаемом сообщении;
- *rtype* – тип элементов в принимаемом сообщении;
- *source* – номер процесса-отправителя;
- *rtag* – идентификатор принимаемого сообщения;
- *comm* – коммуникатор группы;

– *status* – параметры принятого сообщения.

Функция *MPI_Sendrecv* отправляет и принимает сообщения. С помощью этой функции процессы могут отправлять сообщения самим себе. Сообщение, отправленное функцией *MPI_Sendrecv*, может быть принято функцией *MPI_Recv*, и точно также функция *MPI_Sendrecv* может принять сообщение, отправленное функцией *MPI_Send*. Буферы приема и отправки сообщений обязательно должны быть различными.

Если процесс ожидает получения сообщения, но не знает его параметров, можно воспользоваться функцией *MPI_Probe*.

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

Параметры функции:

– *source* – номер процесса-отправителя или *MPI_ANY_SOURCE*;
– *msgtag* – идентификатор принимаемого сообщения или *MPI_ANY_TAG*;
– *comm* – коммуникатор группы;
– *status* – параметры принимаемого сообщения.

Функция *MPI_Probe* получает параметры принимаемого сообщения с блокировкой процесса. Возврата из функции не происходит до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для приема. Параметры принимаемого сообщения определяются обычным образом с помощью параметра *status*. Функция *MPI_Probe* отмечает только факт получения сообщения, но реально его не принимает.

Для определения количества уже полученных сообщений используется функция *MPI_Get_Count*.

```
int MPI_Get_Count (
  MPI_Status *status,
  MPI_Datatype datatype,
```



```
  int *count
);
```

Параметры функции:

– *status* – параметры сообщения;
– *datatype* – тип элементов;
– *count* – число элементов в сообщении.

Функция *MPI_Get_Count* определяет по значению параметра *status* число уже принятых (после обращения к *MPI_Recv*) или принимаемых (после обращения к *MPI_Probe* или *MPI_IProbe*) элементов сообщения типа *datatype*.

Приведенная ниже программа, где каждый процесс n считает сумму чисел от $n \times 10 + 1$ до $(n + 1) \times 10$, иллюстрирует работу функций приема/отправки сообщений с блокировкой

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

int main (int argc, char *argv[])
{
  int i;
  int size, me;
  int sum;
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  sum = 0;
  for (i=me*10+1; i<=(me+1)*10; i++)
    sum = sum + i;
  if (me == 0)
  {
    if (size > 1)
    {
      printf ("Процесс 0, сумма = %d\n", sum);
      for (i=1; i<size; i++)
      {
        MPI_Recv (&sum, 1, MPI_INT, i,
          1, MPI_COMM_WORLD, &status);
```

```
        printf ("Процесс %d, сумма = %d\n", i, sum);
      }
    }
  }
  else
    MPI_Send (&sum, 1, MPI_INT, 0,
      1, MPI_COMM_WORLD);
  MPI_Finalize ();
  return (0);
}
```

В этой программе каждый из процессов определяет свой номер в группе, вычисляет относящуюся к нему часть задачи, отправляет результат нулевому процессу и завершает работу. Нулевой процесс принимает данные от других процессов и печатает все результаты в консоль.

4.3. Прием/отправка сообщений без блокировки

В MPI предусмотрен набор функций для осуществления асинхронной передачи данных. Асинхронная передача предполагает, что возврат из функций приема/отправки происходит сразу же после их вызова. Наличие таких функций позволяет существенно разнообразить алгоритмы взаимодействия процессов. При асинхронной передаче нет необходимости ожидать приема или отправки очередного сообщения, время ожидания может быть использовано в других целях.

После инициирования асинхронной отправки или приема сообщения программа продолжает работу. При этом с помощью специальных вспомогательных функций программа может контролировать процесс приема/отправки и таким образом координировать свои дальнейшие действия.

К сожалению, асинхронные операции не всегда эффективно поддерживаются аппаратурой и системным окружением, поэтому на практике асинхронная передача не всегда повышает скорость вычислений, и эффект от выполнения вычислений на фоне пересылок сообщений может оказаться небольшим или вовсе нулевым. Однако это обстоятельство не умаляет достоинств асинхронной передачи, функциональность которой исключительно полезна для разработчиков и поэтому присутствует практически в каждой параллельной программе.

Для инициирования процесса асинхронной отправки сообщения используется функция *MPI_Isend*.

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

Параметры функции:

- *buf* – отправляемое сообщение;
- *count* – число элементов в отправляемом сообщении;
- *datatype* – тип элементов;
- *dest* – номер процесса-получателя;
- *msgtag* – идентификатор сообщения;
- *comm* – коммуникатор группы;
- *request* – идентификатор асинхронной передачи.

Функция *MPI_Isend* осуществляет отправку сообщения аналогично функции *MPI_Send*, однако возврат из функции происходит сразу после инициирования процесса отправки без ожидания обработки всего сообщения, находящегося в буфере *buf*. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении инициированной ранее отправки. Завершение процесса отправки сообщения можно определить с помощью параметра *request* в функциях *MPI_Wait* и *MPI_Test*. Только после успешной отправки сообщения буфер *buf* может быть использован повторно без опасения испортить передаваемые сообщения.

Сообщение, отправленное с помощью функций *MPI_Send* или *MPI_Isend*, может быть одинаково успешно принято с помощью функций *MPI_Recv* или *MPI_Irecv*.

Для инициирования процесса асинхронного приема сообщения используется функция *MPI_Irecv*.

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

- *buf* – принимаемое сообщение;
- *count* – максимальное число элементов в принимаемом сообщении;
- *datatype* – тип элементов;
- *source* – номер процесса-отправителя;
- *msgtag* – идентификатор сообщения;
- *comm* – коммуникатор группы;
- *request* – идентификатор асинхронной передачи.

Функция *MPI_Irecv* осуществляет прием сообщения аналогично функции *MPI_Recv*, однако возврат из функции происходит сразу после инициирования процесса приема без ожидания получения сообщения в буфере *buf*. Завершение процесса приема сообщения можно определить с помощью параметра *request* в функциях *MPI_Wait* и *MPI_Test*.

При работе с асинхронными функциями очень важно отслеживать состояние передачи сообщения. Отправляющей стороне это необходимо для корректного использования буфера, а принимающей стороне – для своевременной обработки данных. Функция *MPI_Wait* используется для того чтобы приостановить работу процесса до окончания передачи сообщения, т.е. фактически она позволяет осуществлять синхронизацию с данными.

```
int MPI_Wait (
  MPI_Request *request,
  MPI_Status *status
);
```

Параметры функции:

- *request* – идентификатор асинхронной передачи;
- *status* – параметры переданного сообщения.

Функция *MPI_Wait* осуществляет ожидание завершения асинхронных функций *MPI_Isend* и *MPI_Irecv*, инициировавших передачу с идентификатором *request*. В случае успешной передачи, атрибуты переданного сообщения можно определить с помощью параметра *status*.

Приведенная ниже программа, где каждый процесс отправляет сообщение соседу справа и принимает сообщение от соседа слева («соседи» вычисляются на основании номеров процессов), демонстрирует работу функций приема/отправки сообщений без блокировки

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

int main (int argc, char *argv[])
{
  int size, me, left, right;
  int rbuffer[10], sbuffer[10];
  double time;
  MPI_Request rrequest, srequest;
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  right = (me + 1) % size;
  left = me - 1;
  if (left < 0)
    left = size - 1;
  time = MPI_Wtime();
  printf ("Мой номер %d, время до паузы %.0f\n",
    me, time);
  sleep (5);
  if (me == 0)
    sleep(5);
  MPI_Irecv (rbuffer, 10, MPI_INT, left,
```

```
    123, MPI_COMM_WORLD, &rrequest);
  MPI_Isend (sbuffer, 10, MPI_INT, right,
    123, MPI_COMM_WORLD, &srequest);
  time = MPI_Wtime();
  printf ("Мой номер %d, время после паузы %.0f\n",
    me, time);
  MPI_Wait (&rrequest, &status);
  time = MPI_Wtime ();
  printf ("Мой номер %d, время запуска %.0f\n",
    me, time);
  MPI_Finalize ();
  return (0);
}
```

В этом примере используется функция *MPI_Wtime*, которая возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Гарантируется, что этот момент не будет изменен за время существования процесса. В данной программе функция *MPI_Wtime* используется для определения моментов времени, когда программа проходит различные этапы.

Если запустить программу на нескольких вычислительных узлах и проанализировать выводимые на экран сообщения, то можно заметить следующее:

1. До первого вывода сообщения на экран все процессы доходят в одно и то же время.
2. До второго вывода сообщения на экран одновременно доходят все процессы кроме нулевого. Нулевой процесс задерживается на 5 секунд.
3. До третьего вывода сообщения на экран одновременно доходят все процессы кроме первого и нулевого. Первый и нулевой процессы задерживаются на 5 секунд.

На основании этих фактов можно сделать вывод о том, как с помощью функции *MPI_Wait* происходит ожидание завершения передачи сообщения и синхронизация процессов.

Помимо функции *MPI_Wait* существует несколько ее модификаций, ожидающих завершения асинхронной передачи с различными условиями: *MPI_Waitall*, *MPI_Waitany* и *MPI_Waitsome*.


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

Параметры функции:

- *count* – число идентификаторов асинхронной передачи;
- *requests* – идентификаторы асинхронной передачи;
- *statuses* – параметры переданных сообщений.

Функция *MPI_Waitall* останавливает процесс до тех пор, пока все указанные передачи из массива *requests* не будут завершены. Если во время одной или нескольких передач возникнут ошибки, то коды ошибок будут записаны в соответствующие поля параметров передачи из массива *statuses*.

В случае успешной передачи всех сообщений, их атрибуты можно определить с помощью того же массива параметров *statuses*.

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

Параметры функции:

- *count* – число идентификаторов асинхронной передачи;
- *requests* – идентификаторы асинхронной передачи;
- *index* – номер завершенной передачи;
- *status* – параметры переданного сообщения.

Функция *MPI_Waitany* останавливает процесс до тех пор, пока какая-либо передача, из указанных в массиве *requests*, не будет завершена. Параметр *index* возвращает номер элемента в массиве *requests*, содержащего идентификатор завершенной передачи. Если завершены сразу несколько передач, то случайным образом выбирается одна из них.

В случае успешной передачи, атрибуты переданного сообщения можно определить с помощью параметра *status*.

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

Параметры функции:

- *incount* – число идентификаторов асинхронной передачи;
- *requests* – идентификаторы асинхронной передачи;
- *outcount* – число завершенных асинхронных передач;
- *indexes* – массив номеров завершенных передач;
- *statuses* – параметры переданных сообщений в завершенных передачах.

Функция *MPI_Waitsome* останавливает процесс до тех пор, пока хотя бы одна из передач, указанных в массиве *requests*, не будет завершена. В отличие от функции *MPI_Waitany*, функция *MPI_Waitsome* сохраняет информацию обо всех одновременно завершившихся передачах, таким образом, позволяя программе обработать большее количество данных. Параметр *outcount* содержит общее число завершенных передач, а первые *outcount* элементов массива *indexes* содержат номера элементов массива *requests* с идентификаторами завершенных передач. Первые *outcount* элементов массива *statuses* содержат параметры переданных сообщений.

Аналогично функциям *MPI_Wait*, *MPI_Waitall*, *MPI_Waitany* и *MPI_Waitsome* работают функции *MPI_Test*, *MPI_Testall*, *MPI_Testany* и *MPI_Testsome*, за исключением того, что эти функции не останавливают процесс.

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

Параметры функции:

- *request* – идентификатор асинхронной передачи;

- *flag* – признак завершения передачи;
- *status* – параметры переданного сообщения.

Функция *MPI_Test* осуществляет проверку завершения передачи с идентификатором *request*, инициированной асинхронными функциями *MPI_Isend* и *MPI_Irecv*. Параметр *flag* по умолчанию равен нулю и принимает значение 1, если соответствующая передача завершена. В случае успешной передачи, атрибуты переданного сообщения можно определить с помощью параметра *status*.

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

Параметры функции:

- *count* – число идентификаторов асинхронной передачи;
- *requests* – идентификаторы асинхронной передачи;
- *flag* – признак завершения передачи;
- *statuses* – параметры переданных сообщений.

Функция *MPI_Testall* проверяет завершение всех передач из массива *requests*. Параметр *flag* по умолчанию равен нулю и принимает значение 1, если все указанные передачи завершены. В случае успешной передачи, атрибуты переданных сообщений можно определить с помощью массива параметров *statuses*. В противном случае массив параметров *statuses* не определен.

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

Параметры функции:

- *count* – число идентификаторов асинхронной передачи;
- *requests* – идентификаторы асинхронной передачи;

– *index* – номер завершенной передачи;
– *flag* – признак завершения передачи;
– *status* – параметры переданного сообщения.

Функция *MPI_Testany* проверяет завершение передач из массива *requests*. Параметр *flag* по умолчанию равен нулю и принимает значение 1, если хотя бы одна из указанных передач завершена. Параметр *index* возвращает номер элемента в массиве *requests*, содержащего идентификатор завершенной передачи. Если могут быть завершены сразу несколько передач, то случайным образом выбирается одна из них.

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

Параметры функции:
– *incount* – число идентификаторов асинхронной передачи;
– *requests* – идентификаторы асинхронной передачи;
– *outcount* – число завершенных асинхронных передач;
– *indexes* – массив номеров завершенных передач;
– *statuses* – параметры переданных сообщений в завершенных передачах.

Функция *MPI_Testsome* проверяет завершение передач из массива *requests*. В отличие от функции *MPI_Testany*, функция *MPI_Testsome* сохраняет информацию обо всех завершившихся передачах. Параметр *outcount* содержит общее число завершенных передач, а первые *outcount* элементов массива *indexes* содержат номера элементов массива *requests* с идентификаторами завершенных передач. Первые *outcount* элементов массива *statuses* содержат параметры переданных сообщений. Если ни одна из указанных передач не завершена, параметр *outcount* равен нулю.

Использование функции *MPI_Test* и ее производных позволяет разработчику организовывать дополнительную активность процессов без создания дополнительных потоков. Реагирующий

на события потоковый планировщик легко реализуется с помощью циклического вызова *MPI_Test*. Код такого планировщика упрощенно выглядит следующим образом

```
MPI_Test (&request, &flag, &status);
while (!flag)
{
  /* Выполнение полезной работы */
  MPI_Test (&request, &flag, &status);
}
```

В одних и тех же случаях функция *MPI_Wait* возвращает управление, а функция *MPI_Test* возвращает *flag = true*; обе функции при этом возвращают одинаковые параметры *status*. Таким образом, блокирующая функция *MPI_Wait* может быть легко заменена функцией *MPI_Test*.

Для получения информации о формате принимаемого сообщения без блокировки процесса, можно воспользоваться функцией *MPI_Iprobe*.

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

Параметры функции:
– *source* – номер процесса-отправителя или *MPI_ANY_SOURCE*;
– *msgtag* – идентификатор принимаемого сообщения или *MPI_ANY_TAG*;
– *comm* – коммуникатор группы;
– *flag* – признак завершения передачи;
– *status* – параметры принимаемого сообщения.

Функция *MPI_Iprobe* получает информацию о структуре принимаемого сообщения без блокировки процесса. Параметр *flag* по умолчанию равен нулю и принимает значение 1, если сообщение с подходящим идентификатором и номером процес-

са-отправителя доступно для приема. Параметры принимаемого сообщения определяются обычным образом с помощью параметра *status*. Функция *MPI_Iprobe* отмечает только факт получения сообщения, но реально его не принимает.

Код планировщика с применением функции *MPI_Iprobe* может упрощенно выглядеть следующим образом

```
MPI_Iprobe (MPI_ANY_SOURCE, MPI_ANY_TAG,
  MPI_COMM_WORLD, &flag, &status);
while (!flag)
{
  /* Выполнение полезной работы */
  MPI_Iprobe (MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &flag, &status);
}
```

4.4. Объединение запросов на прием/отправку сообщений

На общее время выполнения каждого процесса (время работы параллельной программы на каждом вычислительном узле) оказывают влияние несколько факторов:

- время, затрачиваемое на выполнение вычислительных операций;
- время, затрачиваемое на установку соединения между процессами (вычислительными узлами);
- время, затрачиваемое на пересылку данных.

Длительность установки соединения между процессами, в свою очередь, зависит от латентности коммуникационной среды. Латентность – это интервал времени между моментом инициирования передачи сообщения процессом-отправителем и моментом приема первого байта отправленного сообщения процессом-получателем. Латентность зависит от длины передаваемых сообщений и их количества. Коммуникационная среда, передающая большое количество коротких сообщений и малое количество длинных, имеет разную латентность.

Таким образом, каждая установка соединения между процессами требует дополнительного времени, которое хотелось бы минимизировать.

Для снижения накладных расходов при передаче сообщений между процессами несколько запросов на прием и отправку сообщений предварительно объединяются вместе и затем одновременно выполняются. Для этого используются функции *MPI_Send_init*, *MPI_Recv_init*, *MPI_Start* и их производные.

Способ приема сообщения не зависит от способа его отправки: любое сообщение, отправленное обычным способом или с помощью объединения запросов, может быть принято как обычным способом, так и с помощью объединения запросов.

Параллельные процессы часто обмениваются сообщениями разного содержания, но одинаковой длины и с одинаковой внутренней структурой, например, в циклах. В таких случаях передача однократно инициализируется (подготавливается), после чего происходят многократные приемы/отправки сообщений без затрат дополнительного времени на подготовку данных.

Другим случаем использования функций объединения запросов на прием/отправку сообщений может быть параллельный процесс, способный самостоятельно производить вычисления, накапливая данные для информационного обмена. Впрочем, накапливание данных может привести к перегрузке коммуникационной сети.

Рассмотрим подробнее функции для объединения запросов на передачу сообщений.

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

Параметры функции:

- *buf* – отправляемое сообщение;
- *count* – число элементов в отправляемом сообщении;
- *datatype* – тип элементов;
- *dest* – номер процесса-получателя;

– *msgtag* – идентификатор сообщения;
– *comm* – коммуникатор группы;
– *request* – идентификатор асинхронной передачи.

Функция *MPI_Send_init* формирует запрос на отправку сообщения. Все параметры точно такие же, как у функции *MPI_Isend*, однако пересылка начинается только после вызова функции *MPI_Start*. Содержимое буфера нельзя менять до окончания передачи.

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

Параметры функции:
– *buf* – принимаемое сообщение;
– *count* – максимальное число элементов в принимаемом сообщении;
– *datatype* – тип элементов;
– *source* – номер процесса-отправителя;
– *msgtag* – идентификатор сообщения;
– *comm* – коммуникатор группы;
– *request* – идентификатор асинхронной передачи.

Функция *MPI_Recv_init* формирует запрос на прием сообщения. Все параметры точно такие же, как у функции *MPI_Irecv*, однако пересылка начинается только после вызова функции *MPI_Start*. Содержимое буфера нельзя менять до окончания передачи.

```
int MPI_Start (
  MPI_Request *request
);
```

Параметры функции:
– *request* – идентификатор асинхронной передачи.

Функция *MPI_Start* запускает отложенную передачу с идентификатором *request*, сформированную вызовами функций *MPI_Send_init* и *MPI_Recv_init*.

В момент обращения к функции *MPI_Start* необходимо, чтобы указатель на идентификатор асинхронной передачи указывал на полностью сформированный запрос.

```
int MPI_Start_all (
  MPI_Request *requests
);
```

Параметры функции:
– *requests* – идентификаторы асинхронной передачи.

Функция *MPI_Start_all* запускает все отложенные передачи с идентификаторами из массива *requests*, сформированные вызовами функций *MPI_Send_init* и *MPI_Recv_init*.

В момент обращения к функции *MPI_Start_all* необходимо, чтобы указатели на идентификаторы асинхронной передачи указывали на полностью сформированные запросы.

Для отмены отложенной передачи данных используется функция *MPI_Request_free*.

```
int MPI_Request_free (
  MPI_Request *request
);
```

Параметры функции:
– *request* – идентификатор асинхронной передачи.

Функция *MPI_Request_free* отменяет ранее сформированный запрос на передачу сообщения и освобождает все связанные с ним элементы памяти. Если асинхронная передача с идентификатором *request* происходит в момент вызова функции *MPI_Request_free*, то она не будет прервана и сможет успешно завершиться.

Функцией *MPI_Request_free* нужно пользоваться с осторожностью, потому что ее завершение невозможно отследить с помощью функций *MPI_Test* или *MPI_Wait*.

Приведенная ниже программа демонстрирует работу функций отложенного взаимодействия:

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

int main (int argc, char *argv[])
{
  int i, j;
  int me, size;
  int buffer[3];
  MPI_Request request;
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  if (size > 1)
  {
    if (me == 0)
    {
      MPI_Send_init (buffer, 3, MPI_INT, 1,
        1, MPI_COMM_WORLD, &request);
      for (i=0; i<4; i++)
      {
        for (j=0; j<=2; j++)
          buffer[j] = j * (i + 1);
        MPI_Start (&request);
        MPI_Wait (&request, &status);
      }
    }
    if (me == 1)
    {
      for (i=0; i<4; i++)
      {
        MPI_Recv (buffer, 3, MPI_INT, 0,
          1, MPI_COMM_WORLD, &status);
        printf ("Сообщение %d - %d, %d, %d\n",
          i + 1, buffer[0], buffer[1], buffer[2]);
      }
    }
  }
  MPI_Finalize ();
```



```
  return (0);
}
```

Данная программа использует только два параллельных процесса. Первый процесс формирует отложенный запрос на отправку сообщения второму процессу, после чего четырежды выполняет этот запрос. Перед каждой отправкой сообщения содержимое буфера меняется. Второй процесс принимает отправленные сообщения с блокировкой и выводит их на экран.

Данная программа демонстрирует алгоритм отказа от формирования запросов на отправку каждого типового сообщения. В результате экономится процессорное время.

Использование функции *MPI_Wait* гарантирует завершение передачи перед изменением содержимого буфера и повторной отправкой сообщения. Без функции *MPI_Wait* программа не сможет выполняться и будет прервана с соответствующими ошибками.

4.5. Барьерная синхронизация

Во всех предыдущих случаях процессы синхронизируются друг с другом в момент передачи сообщений с блокировкой или с помощью блокирующих функций типа *MPI_Wait*. Использование этих функций позволяет разработчику предсказывать поведение параллельных процессов в коммуникационной среде, т.е. фактически самостоятельно организовывать синхронизацию.

Функция *MPI_Barrier* предоставляет возможность синхронизации процессов без передачи сообщений.

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

Параметры функции:

– *comm* – коммуникатор группы.

Функция *MPI_Barrier* блокирует работу процесса до тех пор, пока все процессы группы *comm* не вызовут эту функцию.

Приведенная ниже программа демонстрирует работу функции *MPI_Barrier*:

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

int main (int argc, char *argv[])
{
  int i;
  int me, size;
  double buffer[3];
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  buffer[0] = MPI_Wtime();
  sleep (me);
  buffer[1] = MPI_Wtime();
  MPI_Barrier (MPI_COMM_WORLD);
  buffer[2] = MPI_Wtime();
  if (me > 0)
    MPI_Send (buffer, 3, MPI_DOUBLE, 0,
      1, MPI_COMM_WORLD);
  else
  {
    for (i=0; i<size; i++)
    {
      if (i > 0)
        MPI_Recv (buffer, 3, MPI_DOUBLE, i,
          1, MPI_COMM_WORLD, &status);
      printf ("Процесс %d, время в точке 1 - %.0f,
2 - %.0f, 3 - %.0f\n",
        i, buffer[0], buffer[1], buffer[2]);
    }
  }
  MPI_Finalize ();
  return (0);
}
```

В данной программе каждый из запускаемых параллельных процессов трижды фиксирует время:

1. До первого вывода сообщения на экран.
2. Перед вызовом функции MPI_Barrier.
3. После вызова функции MPI_Barrier.

Результаты первого замера времени демонстрируют синхронную работу всех процессов. Затем каждый процесс останавливается на время, равное его порядковому номеру в группе (в секундах). Второй замер времени демонстрирует соответствующую рассинхронизацию в работе всех процессов. Наконец, третий замер времени демонстрирует синхронный выход всех процессов из функции *MPI_Barrier*.

4.6. Группы процессов

Группа – это упорядоченное множество процессов. Каждому процессу в группе сопоставлено целое число – ранг или номер.

Группа с коммуникатором *MPI_GROUP_EMPTY* – пустая группа, не содержащая ни одного процесса.

Группа с коммуникатором *MPI_GROUP_NULL* – значение, используемое для ошибочной группы.

Новые группы можно создавать как на основе уже существующих групп, так и на основе коммуникаторов, но в операциях обмена могут использоваться только коммуникаторы. Базовая группа, из которой создаются все остальные группы процессов, связана с коммуникатором *MPI_COMM_WORLD*, в нее входят все процессы приложения. Операции над группами процессов являются локальными, в них вовлекается только вызвавший процедуру процесс, а выполнение не требует межпроцессорного обмена данными. Любой процесс может производить операции над любыми группами, в том числе над такими, которые не содержат данный процесс. При операциях над группами может получиться пустая группа *MPI_GROUP_EMPTY*.

Если при инициализации параллельной программы можно эффективно работать только с группой *MPI_COMM_WORLD*, то в дальнейшем при помощи функций-конструкторов групп можно создавать новые группы и коммуникаторы.

Конструкторы групп применяются к подмножеству и расширенному множеству существующих групп. Эти конструкторы

создают новые группы на основе существующих групп. Данные операции являются локальными и различные группы могут быть определены на различных процессах; процесс может также определять группу, которая не включает саму себя.

Интерфейс MPI не имеет механизма для формирования группы с нуля, группа может формироваться только на основе другой, предварительно определенной группы.

Базовая группа, на основе которой определены все другие группы – это группа, определяемая начальным коммуникатором *MPI_COMM_WORLD*. Доступ к этой группе, как и ко всем остальным группам, осуществляется с помощью функции *MPI_Comm_group*.

Следующие конструкторы предназначены для создания подмножеств и расширенных множеств существующих групп. Каждый конструктор группы ведет себя так, как будто он возвращает новый объект группы.

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

Параметры функции:

- *comm* – коммуникатор группы;
- *group* – группа в коммуникаторе.

Функция *MPI_Comm_group* возвращает в *group* дескриптор группы из *comm*. Данная функция используется для того, чтобы получить доступ к группе, соответствующей определенному коммуникатору. После этого над группами можно проводить операции.

Возможные операции над множествами определяются следующим образом:

- Объединение (англ. Union) – содержит все элементы первой группы и следующие за ними элементы второй группы, не входящие в первую группу.
- Пересечение (англ. Intersect) – содержит все элементы первой группы, которые также находятся во второй группе, упорядоченные как в первой группе.

- Разность (англ. Difference) – содержит все элементы первой группы, которые не находятся во второй группе, упорядоченные как в первой группе.

Заметим, что для этих операций порядок процессов в создаваемой группе определен прежде всего в соответствии с порядком в первой группе (если возможно) и затем, в случае необходимости, в соответствии с порядком во второй группе. Ни объединение, ни пересечение не коммутативны, но оба ассоциативны. Новая группа может быть пуста, то есть эквивалентна *MPI_GROUP_EMPTY*.

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

Параметры функции:

- *group1* – указатель на первую группу;
- *group2* – указатель на вторую группу;
- *newgroup* – указатель на создаваемую группу.

Функция *MPI_Group_union* создает объединение двух групп, и записывает результат в *newgroup*.

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

Параметры функции:

- *group1* – указатель на первую группу;
- *group2* – указатель на вторую группу;
- *newgroup* – указатель на создаваемую группу.

Функция *MPI_Group_intersection* образует группу, которая является пересечением *group1* и *group2*, и записывает результат в *newgroup*.

```
int MPI_Group_difference (
  MPI_Group group1,
```

```
  MPI_Group group2,
  MPI_Group *newgroup
);
```

Параметры функции:

- *group1* – указатель на первую группу;
- *group2* – указатель на вторую группу;
- *newgroup* – указатель на создаваемую группу.

Функция *MPI_Group_difference* образует группу, которая является результатом исключения *group2* из *group1*, и записывает результат в *newgroup*.

```
int MPI_Group_incl (
MPI_Group group,
int n,
int *ranks,
MPI_Group *newgroup)
```

Параметры функции:

- *group* – указатель на группу;
- *n* – количество процессов, входящих в новую группу;
- *ranks* – номера процессов из *group*, которые должны войти в создаваемую группу;
- *newgroup* – указатель на создаваемую группу.

Функция *MPI_Group_incl* создает группу *newgroup*, которая состоит из *n* процессов из *group* с номерами от *ranks[0]* до *ranks[n-1]*. Процесс с номером *i* в *newgroup* есть процесс с номером *ranks[i]* в *group*. Каждый из *n* элементов массива *ranks* должен быть правильным номером в *group*, и все элементы должны быть различными, иначе программа будет неверна. Если *n* равна нулю, то *newgroup* имеет значение *MPI_GROUP_EMPTY*. Функция *MPI_Group_incl* может использоваться, например, для переупорядочения элементов группы.

```
int MPI_Group_excl (
  MPI_Group group,
  int n,
  int *ranks,
  MPI_Group *newgroup
);
```


Параметры функции:

- *group* – указатель на группу;
- *n* – количество процессов, не входящих в новую группу;
- *ranks* – номера процессов из *group*, которые не должны войти в создаваемую группу;
- *newgroup* – указатель на создаваемую группу.

Функция *MPI_Group_excl* создает группу процессов *newgroup*, которая получается путем удаления из *group* процессов с номерами от *ranks[0]* до *ranks[n-1]*. Упорядочивание процессов в *newgroup* идентично упорядочиванию в *group*. Каждый из *n* элементов массива *ranks* должен быть правильным номером в *group*, и все элементы должны быть различными, иначе программа будет неверна. Если *n* равна нулю, то *newgroup* идентична *group*.

```
int MPI_Group_range_incl (
  MPI_Group group,
  int n,
  int ranges[][3],
  MPI_Group *newgroup
);
```

Параметры функции:

- *group* – указатель на группу;
- *n* – число триплетов в массиве *ranges*;
- *ranges* – массив триплетов, указывающий номера процессов в *group*, которые включены в *newgroup*;
- *newgroup* – указатель на создаваемую группу.

Функция *MPI_Group_range_incl* создает группу на основе последовательностей процессов из существующей группы. Результат записывается в *newgroup*.

Последовательности процессов вычисляются при помощи триплетов:

1. Первый номер.
2. Последний номер.
3. Шаг.

Если аргументы *ranges* состоят из триплетов вида $(first_1, first_2 + stride_1), \ldots, (first_n, first_n + stride_n)$,

то *newgroup* состоит из последовательности процессов с номерами

$first_1, first_1 + stride_1, \ldots, first_1 + stride_1 \times (last_1 - first_1) / stride_1,$

…

$first_n, first_n + stride_n, \ldots, first_n + stride_n \times (last_n - first_n) / stride_n,$

Каждый вычисленный номер должен быть правильным номером в *group*, и все вычисленные номера должны быть различными, иначе программа будет неверной. Заметим, что возможен случай, когда $first_i > last_i$, и $stride_i$ может быть отрицательным, но не может быть равным нулю.

Возможности этой функции позволяют расширить массив номеров до массива включенных номеров и передать результирующий массив номеров и другие аргументы в *MPI_Group_incl*. Вызов функции *MPI_Group_incl* эквивалентен вызову функции *MPI_Group_range_incl* с каждым номером *i* в *ranks*, замененным триплетом *(i, i, 1)* в аргументе *ranges*.

```
int MPI_Group_range_excl (
  MPI_Group group,
  int n,
  int ranges[][3],
  MPI_Group *newgroup
);
```

Параметры функции:

- *group* – указатель на группу;
- *n* – число триплетов в массиве *ranges*;
- *ranges* – массив триплетов, указывающий номера процессов в *group*, которые исключены из *newgroup*;
- *newgroup* – указатель на создаваемую группу.

Функция *MPI_Group_range_excl* создает группу процессов *newgroup*, которая получена путем удаления из *group* определенных последовательностей процессов. Эти последовательности задаются триплетами аналогично *MPI_Group_range_incl*. Каждый вычисленный номер должен быть правильным номером в *group*, и все вычисленные номера должны быть различными, иначе программа будет неверной.

Возможности функции *MPI_Group_range_excl* позволяют расширить массив номеров до массива исключенных номеров и

передать результирующий массив номеров и другие аргументы в *MPI_Group_excl*. Вызов функции *MPI_Group_excl* эквивалентен вызову функции *MPI_Group_range_excl* с каждым номером *i* в *ranks*, замененным триплетом *(i, i, 1)* в аргументе *ranges*.

Рассмотрим еще некоторые функции, помимо *MPI_Group_size* и *MPI_Group_rank*, которые позволяют работать с уже созданными группами.

```
int MPI_Group_translate_ranks (
  MPI_Group group1,
  int n,
  int *ranks1,
  MPI_Group group2,
  int *ranks2
);
```

Параметры функции:

- *group1* – указатель на первую группу;
- *n* – число элементов в массивах *ranks1* и *ranks2*;
- *ranks1* – массив из нуля или более правильных номеров в первой группе;
- *group2* – указатель на вторую группу;
- *ranks2* – массив соответствующих номеров во второй группе или *MPI_UNDEFINED*, если такое соответствие отсутствует.

Функция *MPI_Group_translate_ranks* переводит номера процессов из одной группы в другую. Эта функция важна для определения относительной нумерации одинаковых процессов в двух различных группах. Например, если известны номера некоторых процессов в группе коммуникатора *MPI_COMM_WORLD*, то можно узнать их номера в подмножестве этой группы.

```
int MPI_Group_compare (
  MPI_Group group1,
  MPI_Group group2,
  int *result
);
```

Параметры функции:

- *group1* – указатель на первую группу;

– *group2* – указатель на вторую группу;
– result – результат сравнения.

Функция *MPI_Group_compare* используется для сравнения двух групп. Если члены группы и их порядок в обеих группах совершенно одинаковы, возвращается результат *MPI_IDENT*. Это происходит, например, если *group1* и *group2* имеют тот же самый указатель (дескриптор группы). Если члены группы одинаковы, но порядок различен, то возвращается результат *MPI_SIMILAR*. В остальных случаях возвращается значение *MPI_UNEQUAL*.

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

Параметры функции:

– *group1* – указатель на группу.

Как и любой другой объект, созданный в памяти, группа должна быть удалена разработчиком после ее использования. Функция *MPI_Group_free* отправляет группу на удаление. Указатель на *group* устанавливается в состояние *MPI_GROUP_NULL*. Любая операция, использующая в это время удаляемую группу, завершится нормально.

4.7. Коммуникаторы групп

Коммуникатор предоставляет отдельный контекст обмена между процессами некоторой группы. Контекст обеспечивает возможность независимых обменов данными. Каждой группе процессов может соответствовать несколько коммуникаторов, но каждый коммуникатор в любой момент времени однозначно соответствует только одной группе.

Сразу после вызова функции *MPI_Init* создаются следующие коммуникаторы:

– *MPI_COMM_WORLD* – коммуникатор, объединяющий все процессы параллельной программы;
– *MPI_COMM_NULL* – значение, используемое для ошибочного коммуникатора;

– *MPI_COMM_SELF* – коммуникатор, включающий только вызвавший процесс.

В отличие от создания группы создание коммуникатора является коллективной операцией, требующей наличия межпроцессного обмена, поэтому такие функции должны вызываться всеми процессами некоторого существующего коммуникатора.

В модели клиент-сервер создается много программ, где один процесс (обычно нулевой процесс) играет роль распорядителя, а другие процессы служат вычислительными узлами. В такой структуре для определения ролей различных процессов коммуникатора полезны уже рассмотренные ранее функции *MPI_Comm_size* и *MPI_Comm_rank*. В дополнение к ним рассмотрим функцию *MPI_Comm_compare*.

```
int MPI_Comm_compare (
  MPI_Comm comm1,
  MPI_Comm comm2,
  int *result
);
```

Параметры функции:

– *comm1* – коммуникатор первой группы;
– *comm2* – коммуникатор второй группы;
– *result* – результат сравнения.

Функция *MPI_Comm_compare* используется для сравнения коммуникаторов двух групп. Результат *MPI_IDENT* появляется тогда и только тогда, когда *comm1* и *comm2* являются коммуникаторами одной и той же группы.

Результат *MPI_CONGRUENT* появляется, если исходные группы идентичны по компонентам и нумерации – эти коммуникаторы отличаются только контекстом.

Результат *MPI_SIMILAR* имеет место, если члены группы обоих коммуникаторов являются одинаковыми, но порядок их нумерации различен.

Во всех остальных случаях выдается результат *MPI_UNEQUAL*.

Рассмотрим функции, которые позволяют создать коммуникатор для вновь созданной группы. Первая из таких функций *MPI_Comm_create*.

```
int MPI_Comm_create (
  MPI_Comm comm,
  MPI_Group group,
  MPI_Comm *newcomm
);
```

Параметры функции:

- *comm* – коммуникатор группы;
- *group* – группа, для которой создается коммуникатор;
- *newcomm* – указатель на новый коммуникатор.

Функция *MPI_Comm_create* создает новый коммуникатор *newcomm* с коммуникационной группой, определенной аргументом *group* и новым контекстом. Из *comm* в *newcomm* не передается никакой кэшированной информации. Функция *MPI_Comm_create* возвращает *MPI_COMM_NULL* для процессов, не входящих в *group*. Запрос неверен, если *group* не является подмножеством группы, связанной с *comm*. Заметим, что запрос должен быть выполнен всеми процессами в *comm*, даже если они не принадлежат новой группе.

Еще одна функция *MPI_Comm_dup* дублирует существующий коммуникатор.

```
int MPI_Comm_dup (
  MPI_Comm comm,
  MPI_Comm *newcomm
);
```

Параметры функции:

- *comm* – коммуникатор группы;
- *newcomm* – указатель на новый коммуникатор.

Функция *MPI_Comm_dup* дублирует существующий коммуникатор *comm* вместе со связанными с ним значениями ключей. Для каждого значения ключа соответствующая функция обратного вызова для копирования определяет значение атрибута, связанного с этим ключом в новом коммуникаторе. Одно частное действие, которое может сделать вызов для копирования, состоит в удалении атрибута из нового коммуникатора. Операция *MPI_Comm_dup* возвращает в аргументе *newcomm*

новый коммуникатор с той же группой, любой скопированной кэшированной информацией, но с новым контекстом.

Функция *MPI_Comm_dup* используется, чтобы предоставить вызовам параллельных библиотек дублированное коммуникационное пространство, которое имеет те же самые свойства, что и первоначальный коммуникатор.

Обращение к *MPI_Comm_dup* имеет силу, даже если имеются ждущие парные обмены, использующие коммуникатор *comm*. Типичный вызов мог бы запускать *MPI_Comm_dup* в начале параллельного обращения и *MPI_Comm_free* этого дублированного коммуникатора – в конце вызова.

Отдельного внимания заслуживает функция MPI_Comm_split, поскольку она позволяет выполнить гораздо более сложное распределение процессов между различными коммуникаторами.

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

Параметры функции:

- *comm* – коммуникатор группы;
- *color* – управление соданием подмножества;
- *key* – управление назначением рангов;
- *newcomm* – новый коммуникатор.

Функция *MPI_Comm_split* делит группу, связанную с *comm*, на непересекающиеся подгруппы, по одной для каждого значения цвета *color*.

Каждая подгруппа содержит все процессы того же самого цвета. То есть для создания непересекающихся коммуникаторов каждый процесс должен вызвать эту функцию с некоторым значением параметра *color*. В результате все процессы, у которых было одинаковое значение параметра *color*, будут объединены одним коммуникатором.

В пределах каждой подгруппы процессы пронумерованы в порядке, определенном значением аргумента *key*, со связями, разделенными согласно их номеру в старой группе.

Для каждой подгруппы создается новый коммуникатор и возвращается в аргументе *newcomm*. Процесс может иметь значение цвета *MPI_UNDEFINED*, тогда *newcomm* возвращает *MPI_COMM_NULL*.

Обращение к *MPI_Comm_create* эквивалентно обращению к *MPI_Comm_split*, где все члены *group* имеют *color* равный нулю и *key* равный номеру в *group*, а все процессы, которые не являются членами *group*, имеют *color* равный *MPI_UNDEFINED*.

Функция *MPI_Comm_split* допускает более общее разделение группы на одну или несколько подгрупп с необязательным переупорядочением.

Значение *color* должно быть неотрицательным.

Функция *MPI_Comm_split* – это чрезвычайно мощный механизм для разделения единственной коммуникационной группы процессов на *k* подгрупп, где *k* неявно выбрано пользователем (количеством цветов для раскраски процессов). Полученные коммуникаторы будут не перекрывающимися. Такое деление полезно для организации иерархии вычислений, например для линейной алгебры.

Чтобы преодолеть ограничение на перекрытие создаваемых коммуникаторов, можно использовать множественные обращения к *MPI_Comm_split*. Этим способом можно создавать множественные перекрывающиеся коммуникационные структуры.

Заметим, что для фиксированного цвета ключи не должны быть уникальными. Функция *MPI_Comm_split* сортирует процессы в возрастающем порядке согласно этому ключу, и разделяет связи непротиворечивым способом. Если все ключи определены таким же образом, то все процессы одинакового цвета будут иметь относительный порядок номеров такой же, как и в породившей их группе. В общем случае они будут иметь различные номера.

Если значение ключа для всех процессов данного цвета сделано нулевым, то это означает, что порядок номеров процессов в новом коммуникаторе безразличен. Аргумент *color* не может

иметь отрицательного значения, чтобы не конфликтовать со значением, присвоенным *MPI_UNDEFINED*.

После завершения работы с коммуникатором его можно удалить при помощи функции *MPI_Comm_free*.

```
int MPI_Comm_free (
  MPI_Comm *comm.
);
```

Параметры функции:

– *comm* – удаляемый коммуникатор группы.

Коллективная функция *MPI_Comm_free* маркирует коммуникационный объект для удаления. Удаляемый коммуникатор *comm* принимает значение *MPI_COMM_NULL*.

Любые ждущие операции, которые используют удаляемый коммуникатор, будут завершены нормально. Объект фактически удаляется только в том случае, если не имеется никаких других активных ссылок на него.

Функция *MPI_Comm_free* используется в интра- и интеркоммуникаторах. Удаляемые функции обратного вызова для всех кэшируемых атрибутов вызываются в произвольном порядке.

Число ссылок на объект коммуникатора увеличивается с каждым вызовом *MPI_Comm_dup* и уменьшается с каждым вызовом *MPI_Comm_free*. Объект окончательно удаляется, когда число ссылок достигает нуля.

4.8. Функции коллективного взаимодействия

Функции коллективного взаимодействия используются для выполнения однообразных операций, в которых должны участвовать все процессы одного коммуникатора. Например, такие функции используются для отправки сообщений одного процесса всем остальным процессам.

Особенности коллективных функций:

– коллективные функции не используются для связи процессов типа точка-точка;
– коллективные функции выполняются в режиме с блокировкой;

- возврат из коллективной функции после завершения передачи в каждом процессе происходит независимо от завершения передачи в других процессах;
- количество и объем принимаемых сообщений должно быть равно количеству и объему отправляемых сообщений;
- типы элементов отправляемых и принимаемых сообщений должны совпадать или быть совместимы;
- сообщения не имеют идентификаторов.

К функциям коллективного взаимодействия относятся:

- синхронизация всех процессов одного коммуникатора с помощью функции MPI_Barrier;
- коллективные действия, в число которых входят:
  - отправка информации от одного процесса всем остальным членам некоторой области связи с помощью функции MPI_Bcast;
  - сборка (англ. Gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (англ. Root) процесса с помощью функций MPI_Gather и MPI_Gatherv;
  - сборка распределенного массива в один массив с рассылкой его всем процессам некоторой области связи с помощью функций MPI_Allgather и MPI_Allgatherv;
  - разбиение массива и рассылка его фрагментов (англ. Scatter) всем процессам области связи с помощью функций MPI_Scatter и MPI_Scatterv;
  - совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами в свой буфер приема с помощью функций MPI_Alltoall и 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 с помощью функции MPI_Reduce;
  - с рассылкой результата всем процессам с помощью функции MPI_Allreduce;
  - совмещенная операция Reduce/Scatter с помощью функции MPI_Reduce_scatter;
  - префиксная редукция с помощью функции MPI_Scan.

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

1. Простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов.
2. «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 функций имеют постфикс «v» в конце имени функции.

4.9. Типы данных

Сообщение – это некая порция информации, которая передается от одного процесса другому. Помимо пересылаемой информации сообщение содержит служебные данные, необходимые для обеспечения передачи сообщения адресату. Служебная информация включает идентификатор процесса отправителя сообщения и этот идентификатор называется рангом процесса. Ранг процесса это целое неотрицательное число, значение его назначается системой от нуля и выше. Любой запущенный процесс может с помощью процедуры MPI определить свой ранг. Далее идет адрес, по которому размещаются данные, пересылаемые отправителем – адрес буфера отправления сообщения. Необходимо указывать тип пересылаемых данных.

Особенностью MPI является то, что в сообщении пересылаются, как правило, однотипные данные. Необходимо указывать количество данных – это размер буфера сообщения. При-

нимающий процесс должен зарезервировать необходимое количество памяти для приема. Необходимо указывать идентификатор процесса, который должен получить сообщение. Также указывается адрес, по которому должны быть размещены передаваемые данные у получателя. И, наконец, указывается идентификатор коммуникатора, в рамках которого выполняется пересылка сообщения.

Данные, содержащиеся в сообщении, в общем случае организованы в массив элементов, каждый из которых имеет один и тот же тип. Кроме пересылки данных система передачи сообщений поддерживает пересылку информации о состоянии процессов коммуникации. Это может быть, например, уведомление о том, что прием данных, отправленных другим процессом, завершен. Вообще говоря, из-за присутствия разнородных форматов хранения данных на узлах гетерогенной системы для того, чтобы между нитями был возможен обмен сообщениями, данные могут кодироваться, могут упаковываться с помощью некоторого универсального алгоритма.

В MPI принята своя унифицированная схема обозначения типов данных. В таблице 6 представлено соответствие между типами данных MPI и типами данных Си.

Таблица 6.

Константы MPI	Тип данных Си
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	нет соответствия
MPI_PACKED	нет соответствия

Не имеет смысла подробно останавливаться на предопределенных типах MPI или разбирать соответствующие аналоги в Си. Далее речь пойдет о методах работы с пользовательскими типами данных, о построении карты типа, регистрации и аннулировании и будет приведено несколько примеров использования пользовательских типов в несложных MPI программах. Также будут затронуты вопросы упаковки и распаковки данных.

Как уже говорилось, сообщение в MPI представляет собой массив однотипных данных, элементы которого расположены в последовательных ячейках памяти. Такая структура не всегда удобна, поскольку часто приходится работать с массивами и пересылать, например, не целиком массив, а его фрагмент, либо подмножество его элементов. Расположение элементов в массивах иногда может не укладываться в такую модель организации пересылки данных. Примером может служить пересылка структур данных, содержащих разнотипные элементы. В MPI-программе для пересылки данных, расположенных не последовательно, или для пересылки неоднородных данных придется создать производный тип данных.

В качестве примера, иллюстрирующего подобную проблему, можно рассмотреть следующую ситуацию. В численных расчетах часто приходится иметь дело с матрицами. В языке Си используется линейная модель памяти, в которой матрица хранится в виде последовательно расположенных строк. Так, например, для параллельного умножения матриц требуются пересылки столбцов матрицы, а в линейной модели Си элементы столбцов расположены в оперативной памяти не непрерывно, а с промежутками.

Решать проблему пересылки разнотипных данных или данных, расположенных не в последовательных ячейках памяти, можно разными средствами. Можно «уложить» элементы исходного массива во вспомогательный массив так, чтобы данные располагались непрерывно. Это неудобно и требует дополнительных затрат памяти и процессорного времени. Можно различные элементы данных пересылать по отдельности. Это медленный и неудобный способ обмена. Более эффективным решением является использование производных типов данных.

Производные типы данных создаются во время выполнения программы (а не во время ее трансляции), как правило, перед их использованием.

Создание типа – последовательный процесс, состоящий из двух шагов:

1. Конструирование типа (создание его структуры).
2. Регистрация типа (после регистрации типа его можно использовать наряду со стандартными типами MPI).

После завершения работы с производным типом, он аннулируется. При этом все производные от него типы остаются и могут использоваться дальше, пока и они, в свою очередь, не будут уничтожены.

Производные типы данных создаются из базовых типов с помощью функций-конструкторов.

Производный тип данных в MPI характеризуется последовательностью базовых типов и набором целочисленных значений смещения.

Смещения отсчитываются относительно начала буфера обмена и определяют те элементы данных, которые будут участвовать в обмене. Не требуется, чтобы они были упорядочены (по возрастанию или по убыванию). Отсюда следует, что порядок элементов данных в производном типе может отличаться от исходного, кроме того, один элемент данных может появляться в новом типе многократно.

Последовательность пар (базовый тип, смещение) называется картой типа.

В таблице 7 представлена карта некоторого типа. В карте содержится *n* пар, значит, пользовательский тип содержит *n* элементов, каждый элемент задается своим базовым типом. Базовый тип соответствует либо стандартному предопределенному типу MPI, либо другому пользовательскому типу.

Второй компонент пары – это смещение относительно начала буфера.

Расстояние задается в количестве ячеек между началами элементов в последовательности. Таким образом, допускается ситуация, когда элементы располагаются с разрывами, с некоторым расстоянием между собой, они могут и пересекаться.

Таблица 7.

1-й элемент пары	2-й элемент пары
Базовый тип 0	Смещение базового типа 0
Базовый тип 1	Смещение базового типа 1
Базовый тип 2	Смещение базового типа 2
…	…
Базовый тип n-1	Смещение базового типа n-1

Наиболее общим конструктором типов в MPI является функция *MPI_Type_struct* – конструктор структурного типа. В случае использования этой функции программист может использовать полное описание каждого элемента типа. Если пересылаемые данные содержат подмножество элементов массива, такая детальная информация не нужна, поскольку у всех элементов массива один и тот же базовый тип. MPI предлагает три конструктора, которые можно использовать в такой ситуации: *MPI_Type_contiguous*, *MPI_Type_vector* и *MPI_Type_indexed*.

Первый из них создает производный тип, элементы которого являются непрерывно расположенными элементами массива. Второй создает тип, элементы которого расположены на одинаковых расстояниях друг от друга. Третий создает тип, содержащий произвольные элементы.

Рассмотрим далее интерфейс конструктора *MPI_Type_vector*.

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

Параметры функции:

– *count* – неотрицательное целое значение, задающее количество блоков данных в новом типе;
– *blocklen* – задает длину каждого блока в количестве элементов (неотрицательное целое значение);

– *stride* – определяет количество элементов, расположенных между началом предыдущего и началом следующего блока;
– *oldtype* – базовый тип.

Таким образом, входные параметры позволяют дать исчерпывающее описание того, каким образом должна быть произведена выборка элементов базового типа. Выходным параметром является идентификатор нового типа *newtype*. Этот идентификатор назначается программистом. Исходные данные здесь однотипные.

```
int MPI_Type_hvector (
  int count,
  int blocklen,
  MPI_Aint stride,
  MPI_Datatype oldtype,
  MPI_Datatype *newtype
);
```

Конструктор *MPI_Type_hvector* представляет собой некоторую модификацию предыдущего, в котором смысл параметров сохраняется, а смещения задаются в байтах.

```
int MPI_Type_struct (
  int count,
  int blocklengths[],
  MPI_Aint indices[],
  MPI_Datatype oldtypes[],
  MPI_Datatype *newtype
);
```

Параметры функции:
– *count* – задает количество элементов в производном типе, а также длину массивов *oldtypes*, *indices* и *blocklengths*;
– *blocklengths* – количество элементов в каждом блоке;
– *indices* – смещение каждого блока в байтах;
– *oldtypes* – тип элементов в каждом блоке;
– *newtype* – идентификатор производного типа.

Функция *MPI_Type_struct* является конструктором структурного типа. Она позволяет создать тип, содержащий элементы различных базовых типов.

Базовые типы могут быть разными. Задавая их расположение, можно сконструировать структурный тип.

```
int MPI_Type_indexed (
  int count,
  int blocklens[],
  int indices[],
  MPI_Datatype oldtype,
  MPI_Datatype *newtype
);
```

Параметры функции:
– *count* – количество блоков, одновременно длина массивов *indices* и *blocklens*;
– *blocklens* – количество элементов в каждом блоке;
– *indices* – смещение каждого блока, которое задается в количестве ячеек базового типа (целочисленный массив);
– *oldtype* – базовый тип;
– *newtype* – идентификатор производного типа.

Функция *MPI_Type_indexed* является конструктором индексированного типа. При создании индексированного типа блоки располагаются по адресам с разным смещением и его можно считать обобщением векторного типа.

Функция *MPI_Type_hindexed* также является конструктором индексированного типа, однако смещения indices задаются в байтах.

```
int MPI_Type_hindexed (
  int count,
  int blocklens[],
  MPI_Aint indices[],
  MPI_Datatype oldtype,
  MPI_Datatype *newtype
);
```

Функция *MPI_Type_contiguous* используется для создания типа данных с непрерывным расположением элементов.

```
int MPI_Type_contiguous (
  int count,
  MPI_Datatype oldtype,
  MPI_Datatype *newtype
);
```

Параметры функции:

- *count* – счетчик повторений;
- *oldtype* – базовый тип;
- *newtype* – идентификатор нового типа.

Функция *MPI_Type_contiguous* фактически создает описание массива.

Пример использования производного типа:

```
MPI_Datatype a;
float b[10];
…
MPI_Type_contiguous (10, MPI_REAL, &a);
MPI_Type_commit (&a);
MPI_Send (b, 1, a,…);
MPI_Type_free (&a);
```

Выше приведен фрагмент раздела описаний, в котором содержится описание идентификатора нового типа *MPI_Datatype* и пересылаемого массива *b[10]*. Работа с пользовательским типом выглядит следующим образом:

1. Вызывается конструктор, который позволяет создать описание нового типа. Он создает тип *a* – однородный тип, состоящий из некоторого количества последовательно расположенных элементов типа *MPI_REAL*.
2. Конструктор обязательно должен быть зарегистрирован, и регистрация выполняется функцией *MPI_Type_commit* с единственным параметром идентификатором нового типа. После этого новый тип можно использовать в операциях пересылки.
3. В конце вызывается деструктор *MPI_Type_free*, который аннулирует пользовательский тип.

Тип данных, соответствующий подмассиву многомерного массива, можно создать с помощью функции *MPI_Type_create_subarray*:


```
int MPI_Type_create_subarray (
  int ndims,
  int *sizes,
  int *subsizes,
  int *starts,
  int order,
  MPI_Datatype oldtype,
  MPI_Datatype *newtype
);
```

Параметры функции:

- *ndims* – размерность массива;
- *sizes* – количество элементов типа *oldtype* в каждом измерении полного массива;
- *subsizes* – количество элементов типа *oldtype* в каждом измерении подмассива;
- *starts* – стартовые координаты подмассива в каждом измерении;
- *order* – флаг, задающий переупорядочение;
- *oldtype* – базовый тип.

Конструктор *MPI_Type_create_subarray* используется, когда надо выбирать подмножества, например, при работе с многомерными решетками или при выборе гиперплоскостей.

Ограничимся рассмотренными конструкторами пользовательских типов и обратимся к функциям их регистрации и удаления. С помощью вызова функции *MPI_Type_commit* производный тип *datatype*, сконструированный программистом, регистрируется. После этого он может использоваться в операциях обмена:

```
int MPI_Type_commit (
  MPI_Datatype *datatype
);
```

Аннулировать производный тип datatype можно с помощью вызова функции *MPI_Type_free*:

```
int MPI_Type_free (
  MPI_Datatype *datatype
);
```

Следует напомнить, что предопределенные (базовые) типы данных не могут быть аннулированы.

Существует некоторое количество вспомогательных функций, которые могут использоваться при работе с пользовательскими типами. В качестве примера можно привести *MPI_Type_size*, позволяющую определить размер того или иного типа в байтах, т.е. определить объем памяти, который занимает один элемент данного типа.

```
int MPI_Type_size (
  MPI_Datatype datatype,
  int *size
);
```

Количество элементов данных в одном объекте типа *datatype* (его экстент) можно определить с помощью вызова функции *MPI_Type_extent*.

```
int MPI_Type_extent (
  MPI_Datatype datatype,
  MPI_Aint *extent
);
```

Смещения могут даваться относительно базового адреса, значение которого содержится в константе *MPI_BOTTOM*.

Адрес (англ. address) по заданному положению (англ. location) можно определить с помощью функции *MPI_Address*.

```
int MPI_Address (
  void *location,
  MPI_Aint *address
);
```

С помощью функции *MPI_Type_get_contents* можно определить фактические параметры, использованные при создании производного типа.

```
int MPI_Type_get_contents (
  MPI_Datatype datatype,
  int max_integers,
  int max_addresses,
  int max_datatypes,
```



```
  int *integers,
  MPI_Aint *addresses,
  MPI_Datatype *datatypes
);
```

Параметры функции:

- *datatype* – идентификатор типа;
- *max_integers* – количество элементов в массиве *integers*;
- *max_addresses* – количество элементов в массиве *addresses*;
- *max_datatypes* – количество элементов в массиве *datatypes*;
- *integers* – содержит целочисленные аргументы, использованные при конструировании указанного типа;
- *addresses* – содержит аргументы *address*, использованные при конструировании указанного типа;
- *datatypes* – содержит аргументы *datatype*, использованные при конструировании указанного типа.

Приведенная ниже программа демонстрирует процесс создания и использования структурного типа.

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

struct newtype {
  float a;
  float b;
  int n;
};

int main (int argc, char *argv[])
{
  int myrank;
  MPI_Datatype NEW_MESSAGE_TYPE;
  int block_lengths[3];
  MPI_Aint displacements[3];
  MPI_Aint addresses[4];
  MPI_Datatype typelist[3];
  int blocks_number;
```

```
struct newtype indata;
int tag;
MPI_Status status;

MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
tag = 0;
typelist[0] = MPI_FLOAT;
typelist[1] = MPI_FLOAT;
typelist[2] = MPI_INT;
block_lengths[0] =
  block_lengths[1] =
    block_lengths[2] = 1;
MPI_Address(&indata, &addresses[0]);
MPI_Address(&(indata.a), &addresses[1]);
MPI_Address(&(indata.b), &addresses[2]);
MPI_Address(&(indata.n), &addresses[3]);
displacements[0] = addresses[1] - addresses[0];
displacements[1] = addresses[2] - addresses[0];
displacements[2] = addresses[3] - addresses[0];
blocks_number = 3;
MPI_Type_struct (blocks_number, block_lengths,
  displacements, typelist, &NEW_MESSAGE_TYPE);
MPI_Type_commit (&NEW_MESSAGE_TYPE);
if (myrank == 0)
{
  indata.a = 3.14159;
  indata.b = 2.71828;
  indata.n = 2009;
  MPI_Send (&indata, 1, NEW_MESSAGE_TYPE, 1,
    tag, MPI_COMM_WORLD);
  printf ("Процесс %i отправил: %f %f %i\n",
    myrank, indata.a, indata.b, indata.n);
}
else
{
  MPI_Recv (&indata, 1, NEW_MESSAGE_TYPE, 0,
    tag, MPI_COMM_WORLD, &status);
  printf ("Процесс %i принял: %f %f %i, %d\n",
    myrank, indata.a, indata.b, indata.n,
```



```
      status.MPI_ERROR);
  }
  MPI_Type_free (&NEW_MESSAGE_TYPE);
  MPI_Finalize ();
  return (0);
}
```

В программе задаются типы членов производного типа. Затем количество элементов каждого типа. Вычисляются адреса членов типа indata и определяются смещения трех членов производного типа относительно адреса первого, для которого смещение равно нулю. Располагая этой информацией, можно определить производный тип, что и делается с помощью функций *MPI_Type_struct* и *MPI_Type_commit*. Созданный таким образом производный тип можно использовать в любых операциях обмена. Тип *MPI_Aint* представляет собой скалярный тип, длина которого имеет размер, одинаковый с указателем.

Функции упаковки и распаковки данных *MPI_Pack* и *MPI_Unpack* являются альтернативой пользовательских типов. Функция *MPI_Pack* позволяет явным образом хранить произвольные (в том числе и расположенные не в последовательных ячейках) данные в непрерывной области памяти (буфере передачи). Функцию *MPI_Unpack* используют для копирования данных из буфера приема в произвольные (в том числе и не расположенные непрерывно) ячейки памяти. Упаковка и распаковка используется для решения таких задач как обеспечение совместимости с другими библиотеками обмена сообщениями, для приема сообщений по частям, для буферизации исходящих сообщений в пользовательское пространство памяти, что дает независимость от системной политики буферизации.

Сообщения передаются по коммуникационной сети, связывающей узлы вычислительной системы. Сеть работает медленно, поэтому, чем меньше в параллельной программе обменов, тем меньше потери на пересылку данных. С учетом этого полезен механизм, который позволял бы вместо отправки трех разных значений тремя сообщениями, отправлять их все вместе. Такие механизмы есть: это параметр *count* в функциях обмена, производные типы данных и функции *MPI_Pack* и *MPI_Unpack*.

С помощью аргумента *count* в функциях *MPI_Send*, *MPI_Recv*, *MPI_Bcast* и *MPI_Reduce* можно отправить в одном сообщении несколько однотипных элементов данных. Для этого элементы данных должны находиться в непрерывно расположенных ячейках памяти.

Если элементы данных – простые переменные, они могут не находиться в последовательных ячейках памяти. В этом случае можно использовать производные типы данных или упаковку.

Функция упаковки *MPI_Pack* может вызываться несколько раз перед передачей сообщения, содержащего упакованные данные, а функция распаковки *MPI_Unpack* в этом случае также будет вызываться несколько раз после выполнения приема.

Для извлечения каждой порции данных применяется новый вызов.

При распаковке данных текущее положение указателя в буфере сохраняется.

```
int MPI_Pack (
  void *inbuf,
  int incount,
  MPI_Datatype datatype,
  void *outbuf,
  int outcount,
  int *position,
  MPI_Comm comm.
);
```

Параметры функции:

- *inbuf* – начальный адрес входного буфера;
- *incount* – количество входных данных;
- *datatype* – тип каждого входного элемента данных;
- *outcount* – размер выходного буфера в байтах;
- *position* – текущее положение в буфере в байтах;
- *comm* – коммуникатор для упакованного сообщения;
- *outbuf* – стартовый адрес выходного буфера.

При вызове функции упаковки incount элементов указанного типа выбираются из входного буфера и упаковываются в выходном буфере, начиная с положения *position*.


```
int MPI_Unpack (
  void *inbuf,
  int insize,
  int *position,
  void *outbuf,
  int outcount,
  MPI_Datatype datatype,
  MPI_Comm comm.
);
```

Параметры функции:

- *inbuf* – начальный адрес входного буфера;
- *insize* – размер входного буфера в байтах;
- *position* – текущее положение в байтах;
- *outcount* – количество данных, которые должны быть распакованы;
- *datatype* – тип каждого выходного элемента данных;
- *comm* – коммуникатор для упаковываемого сообщения;
- *outbuf* – стартовый адрес выходного буфера.

Существуют некоторые вспомогательные функции, которые облегчают работу с функциями *MPI_Pack* и *MPI_Unpack*. Важнейшей из таких функций является *MPI_Pack_size*, которая позволяет определить объем памяти *size* (в байтах), необходимый для распаковки сообщения.

```
int MPI_Pack_size (
  int incount,
  MPI_Datatype datatype,
  MPI_Comm comm,
  int *size
);
```

Параметры функции:

- *incount* – аргумент *count*, использованный при упаковке;
- *datatype* – тип упакованных данных;
- *comm* – коммуникатор.

Приведенная ниже программа демонстрирует работу функций широковещательной рассылки сообщений, упаковки и распаковки данных.

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

int main (int argc, char *argv[])
{
  int myrank;
  float a, b;
  int n;
  char buffer[100];
  int position;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  if (myrank == 0)
  {
    printf ("Введите a, b и n\n");
    scanf ("%f %f %i", &a, &b, &n);
    position = 0;
    MPI_Pack (&a, 1, MPI_FLOAT, &buffer,
      100, &position, MPI_COMM_WORLD);
    MPI_Pack (&b, 1, MPI_FLOAT, &buffer,
      100, &position, MPI_COMM_WORLD);
    MPI_Pack (&n, 1, MPI_INT, &buffer,
      100, &position, MPI_COMM_WORLD);
    MPI_Bcast (&buffer, 100, MPI_PACKED, 0,
      MPI_COMM_WORLD);
  }
  else
  {
    MPI_Bcast (&buffer, 100, MPI_PACKED, 0,
      MPI_COMM_WORLD);
    position = 0;
    MPI_Unpack (&buffer, 100, &position, &a,
      1, MPI_FLOAT, MPI_COMM_WORLD);
    MPI_Unpack (&buffer, 100, &position, &b,
      1, MPI_FLOAT, MPI_COMM_WORLD);
    MPI_Unpack (&buffer, 100, &position, &n,
      1, MPI_INT, MPI_COMM_WORLD);
    printf ("Процесс %i принял a=%f, b=%f, n=%i\n",
      myrank, a, b, n);
```



```
  }
  MPI_Finalize ();
  return (0);
}
```

В данной программе нулевой процесс копирует в буфер значение *a*, затем дописывает туда *b* и *n*. После выполнения широковещательной рассылки главным процессом остальные используют функцию *MPI_Unpack* для извлечения *a*, *b*, и *n* из буфера. При вызове функции *MPI_Bcast* необходимо использовать тип данных *MPI_PACKED*.

Типы данных *MPI_PACKED* и *MPI_BYTE* не имеют аналогов в языке Си. Они позволяют хранить данные в «сыром» формате, имеющем одинаковое двоичное представление на стороне отправителя сообщения и на стороне получателя. Представление символа может различаться на разных машинах, а байт везде одинаков.

Тип *MPI_BYTE*, например, может использоваться, если необходимо выполнить преобразование между разными представлениями в гетерогенной вычислительной системе.

ЛИТЕРАТУРА

1. Антонов А.С. Введение в параллельные вычисления: методическое пособие. – М.: Изд-во МГУ, 2002.
2. Антонов А.С. Параллельное программирование с использованием технологии MPI: учебное пособие. – М.: Изд-во МГУ, 2004.
3. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. – Ростов-на-Дону: Изд-во ООО «ЦВВР», 2004.
4. Дацюк В.Н., Букатов А.А., Жегуло А.И. Многопроцессорные системы и параллельное программирование: методическое пособие. – Ростов-на-Дону: Изд-во РГУ, 2003.
5. Шпаковский Г.И., Серикова Н.В., Липницкий А.С., Верхотуров А.Е., Гришанович А.Н., Орлов А.В. Стандарт MPI: A Message-Passing Interface Standart, June 12, 1995. – Минск: Изд-во БГУ, 2004.
6. Шпаковский Г.И., Серикова Н.В., Липницкий А.С., Верхотуров А.Е., Гришанович А.Н., Орлов А.В. Стандарт MPI-2: Extension to the Message-Passing Interface, July 18, 1997. – Минск: Изд-во БГУ, 2004.
7. Amdahl G. Validity of the single-processor approach to achieving large-scale computing capabilities. – AFIPS Conf., AFIPS Press, 1967.
8. Foster I. Designing and Building Parallel Programs. // http://www.hensa.ac.uk/parallel/books/addison-wesley/dbpp – 1995.

ISBN 978-5-91450-031-0

9 785914 500310

Введение в параллельные вычисления.
Основы программирования на языке Си
с использованием интерфейса MPI.
Научное издание

В печать от 14.04.2009
Формат бумаги 60x84/16.Уч.-изд. л. 5,1
Тираж 100. Заказ 19
117997, Москва, Профсоюзная, 65
Учреждение Российской академии наук
Институт проблем управления
им. В.А. Трапезникова РАН