ПРОГРАММИРОВАНИЕ ЗАДАЧ ФИЗИКИ КОНДЕНСИРОВАННОГО СОСТОЯНИЯ С ИСПОЛЬЗОВАНИЕМ MPI

В.Н. Блинов, А.А. Севенюк

Московский государственный университет им. М.В. Ломоносова, механико-математический факультет

blinov.veniamin@gmail.com, kireeva.al@gmail.com

Поступила 01.06.2012

С недавних пор развитие вычислительной техники отклонилось от привычного курса наращивания тактовых частот в сторону многоядерных систем. Этот поворот отразился на методах высокопроизводительных вычислений, к каковым относится компьютерное моделирование в физике. В этом отношении задача микроскопического описания систем конденсированного состояния представляет особый интерес, поскольку реализация решения этого класса задач подходит для многоядерных архитектур. В данном тексте изложены основные идеи моделирования таких систем, и приведены основы программирования с использованием библиотеки MPI.

УДК 538.9, 519.245, 004.942

Предисловие

Данный обзор представляет собой базовое пособие по численному решению различных задач, в частности, физики мягкой материи, с применением параллельного программирования на основе MPI (Message Passing Interface). Читатель сможет ознакомиться с принципами параллельного программирования, а также разобрать примеры, основываясь на которых можно будет применять алгоритмы параллельных вычислений к другим задачам. Будут разобраны как простые примеры применения параллельных

вычислений к обучающим задачам (сложение векторов, умножение матриц, вычисление интеграла), так и две важнейшие сложные задачи: моделирование динамики системы N частиц (*молекулярная динамика*) и моделирование термодинамических ансамблей при помощи вероятностных алгоритмов, основанных на методе Монте-Карло. Последние повсеместно используются при моделировании физических систем.

Отметим, что на сегодняшний день существует достаточно большое количество готовых решений, как коммерческих, так и бесплатных, позволяющих осуществлять моделирование динамики и термодинамических ансамблей (например, пакеты *Gromacs [1], NAMD[2], LAMMPS[3]* и другие). Они имеют широкую функциональность, что позволяет проводить моделирование целого ряда задач. В этой работе мы подробно остановимся на описании алгоритмов и решений, которые могут быть использованы для написания авторских программ для моделирования физических систем. Большинство из описанных алгоритмов реализовано в названных готовых продуктах.

Помимо вопросов написания программ мы рассматриваем также и базовые идеи, используемые в моделировании задач физики конденсированного состояния. В работе представлены основные подходы к моделированию различных задач и приведены ссылки на источники, по которым читатель сможет при необходимости восстановить детальную реализацию конкретных методов.

1. Эволюция вычислительных систем

Развитие многопроцессорных вычислительных систем является приоритетным направлением развития компьютерной техники, поскольку существуют принципиальные ограничения максимально возможного быстродействия обычных последовательных процессоров. Одновременно с этим постоянно находятся задачи, для решения которых не достаточно существующих возможностей вычислительных средств [4], рис. 1. Такие задачи как, например, прогнозирование погоды, проектирование интегральных схем, генная инженерия, анализ загрязнения окружающей среды, создание лекарственных препаратов и многие другие требуют производительности более триллиона операций с плавающей точкой в секунду. Проблема создания высокопроизводительных вычислительных систем, способных подойти ближе к решению таких задач, относится к числу наиболее сложных научно-технических задач настоящего времени.

Несмотря на то, что суперкомпьютеры появились достаточно давно, применение параллельных вычислений получило более широкое распространение лишь около 2006 года, когда повсеместно стали использоваться многоядерные процессоры. Это позволило существенно ускорить решение некоторых задач, получая даже на одном персональном компьютере в несколько раз большую производительность расчётов.

Однако использование параллельных алгоритмов часто бывает сопряжено и с некоторыми трудностями. Так, параллельная реализация может оказаться медленнее последовательной в тех случаях, когда выбранный алгоритм плохо параллелизуется (то есть, не подходит для многоядерной архитектуры), или выбран неверный метод параллельной реализации. Кроме того, даже если вычисления организованы наилучшим образом, практически любая программа имеет последовательную часть, распараллеливание которой невозможно. Если выполнение этой части программы будет занимать львиную долю всего времени работы, распараллеливание остальной части может оказаться бессмысленным. В соответствии с законом Амдала [5], ускорение при использовании p процессов ограничивается величиной

$$S(p) = \frac{p}{1 + (p-1)f}$$

где f — доля последовательных вычислений в применяемом алгоритме.

Рис. 1. Проблемы компьютерного моделирования. Материал доклада “Grand Challenge”, [4].

Помимо программной части, важнейшую роль играет и архитектура вычислительных средств, перенос параллельных алгоритмов с одной архитектуры на другую может вызвать определенные трудности, а в ряде случаев оказаться невозможным.

Тем не менее, несмотря на все перечисленные трудности, параллельные вычисления незаменимы в тех ситуациях, когда решение задачи с использованием традиционных последовательных алгоритмов невозможно за разумное время.

2. Основы параллельных вычислений

Рассмотрим подробнее, что такое параллельные вычисления, и в каких случаях их использование оправдано.

Итак, пусть имеется список команд, выполняющихся последовательно на каком-либо устройстве, например, программа на компьютере. Для ясности дальнейшего изложения будем считать, что каждая команда представляет собой три элемента: инструкцию, входные данные и выходные данные, соответственно (F_i, IN_i, OUT_i), а программа есть последовательность таких троек. Процессор последовательно читает такие тройки, берёт из памяти данные IN_i, с которыми необходимо провести операцию F_i, выполняет указанную инструкцию и возвращает полученные данные OUT_i обратно в память. Схематичное изображение данного процесса приведено на рисунке 2.

систему, которая потенциально сможет решать некоторый класс задач значительно быстрее одного компьютера.

2.1. Зачем нужен MPI

Если программист захочет реализовать некоторый алгоритм так, чтобы он работал на нескольких компьютерах одновременно (параллельно), ему нужно сделать два важных шага: распределить работу между компьютерами и указать то, в какой момент какие данные должны передаваться. Он сможет осуществить первый из указанных шагов, а именно, реализовать на одном из языков программирования алгоритм каждого в отдельности компьютера. Но для передачи данных между компьютерами в ходе вычислений ему нужен некоторый механизм, которого обычно нет среди стандартных средств языка. Для решения проблемы передачи данных между компьютерами служит интерфейс MPI (*Message Passing Interface*, интерфейс передачи сообщений), позволяющий процессам в ходе работы обмениваться данными. Таким образом, программист, подключив к своему проекту библиотеку MPI, сможет сделать второй шаг реализации параллельных вычислений.

Стоит подробнее остановиться на словах *компьютер* и *процесс*. До этого момента рассматриваемой задачей было распределение вычислительной нагрузки между *компьютерами*. Для этого на каждом компьютере запускается программа (*процесс*), и обмен данными происходит через локальную сеть. Однако MPI позволяет даже больше: обмен данными между *процессами*, которые могут быть запущены и на одном и том же компьютере. Так, например, с помощью MPI можно успешно осуществить параллельную реализацию задачи на многоядерных процессорах и многопроцессорных компьютерах. В этом случае на одном процессоре запускается несколько процессов, выполнение которых распределяется по ядрам операционной системой.

В дальнейшем изложении *узлом* (*node*) мы будем называть один компьютер системы, состоящей из нескольких компьютеров. При этом он может содержать несколько процессоров, процессоры, в свою очередь, могут быть многоядерными. Принципиально узел отличает лишь то, что запущенные на нём процессы не используют для обмена данными локальную сеть, а обходятся внутренними шинами, которые, как привило, работают быстрее.

2.2. Архитектуры параллельных вычислений

Рассмотренный ранее случай нескольких компьютеров, соединённых между собой, представляет наиболее гибкую для программирования систему, поскольку каждый из них в отдельности является совершенно полноценным и может выполнять любые задачи независимо от работы остальных. Соответственно, круг задач, которые можно решать на такой системе, достаточно широк. Однако у таких систем есть узкое место: медленная передача данных. В связи с этим, распараллеливание задачи на достаточно большое число компьютеров может привести к замедлению выполнения даже по сравнению с одним компьютером. По этой причине при программировании для таких систем следует следить за количеством и объёмом передач данных.

Проблема медленной передачи данных решается на физическом уровне двумя способами. Первый – это многоядерные процессоры, в которых каждое ядро, по сути, является компьютером. Все ядра при этом имеют общую память, что значительно ускоряет обмен данными между процессами. Второй способ – это увеличение скорости передачи данных за счёт соединения компьютеров, например, при помощи InfiniBand, который в сотни раз быстрее обыкновенных сетей. Сочетание этих двух подходов реализовано в современных суперкомпьютерах, состоящих из многоядерных компьютеров-узлов, соединённых через шину InfiniBand.

систему, которая потенциально сможет решать некоторый класс задач значительно быстрее одного компьютера.

2.1. Зачем нужен MPI

Если программист захочет реализовать некоторый алгоритм так, чтобы он работал на нескольких компьютерах одновременно (параллельно), ему нужно сделать два важных шага: распределить работу между компьютерами и указать то, в какой момент какие данные должны передаваться. Он сможет осуществить первый из указанных шагов, а именно, реализовать на одном из языков программирования алгоритм каждого в отдельности компьютера. Но для передачи данных между компьютерами в ходе вычислений ему нужен некоторый механизм, которого обычно нет среди стандартных средств языка. Для решения проблемы передачи данных между компьютерами служит интерфейс MPI (*Message Passing Interface*, интерфейс передачи сообщений), позволяющий процессам в ходе работы обмениваться данными. Таким образом, программист, подключив к своему проекту библиотеку MPI, сможет сделать второй шаг реализации параллельных вычислений.

Стоит подробнее остановиться на словах *компьютер* и *процесс*. До этого момента рассматриваемой задачей было распределение вычислительной нагрузки между *компьютерами*. Для этого на каждом компьютере запускается программа (*процесс*), и обмен данными происходит через локальную сеть. Однако MPI позволяет даже больше: обмен данными между *процессами*, которые могут быть запущены и на одном и том же компьютере. Так, например, с помощью MPI можно успешно осуществить параллельную реализацию задачи на многоядерных процессорах и многопроцессорных компьютерах. В этом случае на одном процессоре запускается несколько процессов, выполнение которых распределяется по ядрам операционной системой.

В дальнейшем изложении *узлом* (*node*) мы будем называть один компьютер системы, состоящей из нескольких компьютеров. При этом он может содержать несколько процессоров, процессоры, в свою очередь, могут быть многоядерными. Принципиально узел отличает лишь то, что запущенные на нём процессы не используют для обмена данными локальную сеть, а обходятся внутренними шинами, которые, как привило, работают быстрее.

2.2. Архитектуры параллельных вычислений

Рассмотренный ранее случай нескольких компьютеров, соединённых между собой, представляет наиболее гибкую для программирования систему, поскольку каждый из них в отдельности является совершенно полноценным и может выполнять любые задачи независимо от работы остальных. Соответственно, круг задач, которые можно решать на такой системе, достаточно широк. Однако у таких систем есть узкое место: медленная передача данных. В связи с этим, распараллеливание задачи на достаточно большое число компьютеров может привести к замедлению выполнения даже по сравнению с одним компьютером. По этой причине при программировании для таких систем следует следить за количеством и объёмом передач данных.

Проблема медленной передачи данных решается на физическом уровне двумя способами. Первый – это многоядерные процессоры, в которых каждое ядро, по сути, является компьютером. Все ядра при этом имеют общую память, что значительно ускоряет обмен данными между процессами. Второй способ – это увеличение скорости передачи данных за счёт соединения компьютеров, например, при помощи InfiniBand, который в сотни раз быстрее обыкновенных сетей. Сочетание этих двух подходов реализовано в современных суперкомпьютерах, состоящих из многоядерных компьютеров-узлов, соединённых через шину InfiniBand.

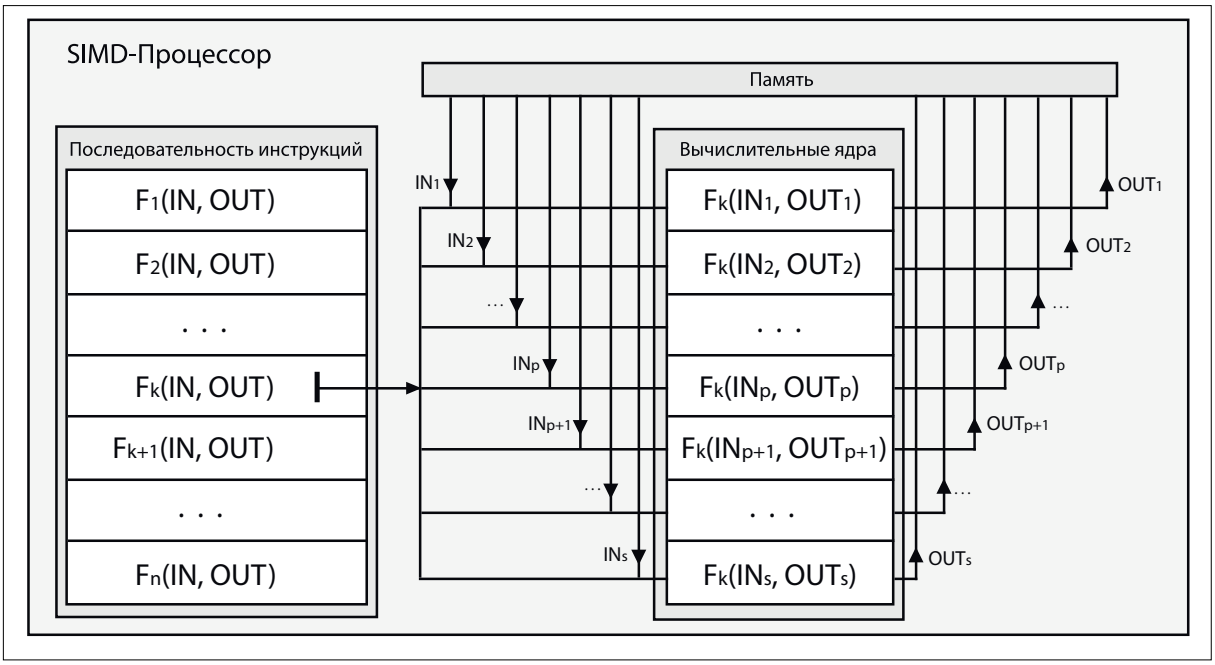

Рис. 3. Схема SIMD-процессора.

Как было отмечено выше, архитектуры, в которых каждый элемент представляет собой полноценный процессор, способный исполнять свой процесс независимо от остальных, могут быть использованы для параллельной реализации максимально широкого круга задач. Такие архитектуры иногда называют MIMD (*Multiple Instruction, Multiple Data*) архитектурами, что означает, что каждый элемент (ядро процессора) способен выполнять свою индивидуальную задачу на своих данных, то есть, на каждом такте выполняется сразу много разных инструкций на разных ядрах. Для программирования для таких архитектур обычно используется MPI.

Существуют и другие архитектуры, предназначенные, в основном, для решения более узкого класса задач. Например, графические процессоры, являющиеся представителями SIMD (*Single Instruction, Multiple Data*) архитектур. Класс задач, которые ставятся перед видеокартами, достаточно узок и обычно сводится к обработке изображений, что, в свою очередь, часто есть выполнение определённых действий с пикселями, образующими эти изображения. Для решения данных задач подходит следующая архитектура (см. рисунок 3).

Процессор содержит одну последовательность инструкций и большое число ядер. На каждом шаге все ядра выполняют одну инструкцию, но для разных данных, что позволяет быстро оперировать с массивами данных. По этой причине SIMD-процессоры также называют *векторными*. Отсутствие, в частности, отдельной последовательности инструкций у каждого ядра, с одной стороны, сужает круг задач, которые можно параллельно реализовать на таких системах, а с другой – освобождает место на плате процессора, что позволяет разместить большее число ядер (сотни) на одном процессоре, что приводит к значительному росту пиковой производительности.

Использование графических карт для вычислений стало достаточно популярной тенденцией последних лет, поскольку, во-первых, ряд интересных физических задач может быть реализован на графическом процессоре, во-вторых, соотношение цена/производительность у таких архитектур значительно меньше, и, в-третьих, потому что стали доступны средства программирования этих устройств, такие как OpenCL и CUDA.

В связи с развитием техник программирования графических процессоров и их доступностью, большое число суперкомпьютеров стали оснащать также и такими процессорами. Полученные системы часто называют *гибридными*, и правильное применение таких систем к решению вычислительных задач может дать значительное ускорение расчётов (хотя при этом необходимо достаточно изощрённо сочетать MPI с OpenCL/CUDA).

Распространение многоядерных систем привело к появлению достаточно большого числа библиотек, реализующих обмен данными между процессами: MPI, OpenMP, OpenCL, CUDA и другие. Возникает вопрос, почему MPI популярнее остальных? Ответ можно сформулировать следующим образом. Во-первых, программирование на MPI достаточно просто (по сравнению, например, с OpenCL или CUDA) как с точки зрения идейной, так и с точки зрения реализации, а во-вторых, MPI может использоваться для программирования более широкого круга вычислительных систем и задач. Подробнее про различные архитектуры процессоров, историю развития архитектур параллельных вычислительных систем а так же про архитектуры современных суперкомпьютеров можно прочитать в книгах В. П. Гергеля «Теория и практика параллельных вычислений» [6] и Барского «Архитектура параллельных вычислительных систем» [7].

2.3. Особенности построения программ

Написание программ на любом языке представляет собой некоторое искусство. Когда человек пишет свой первый «Hello world!», у него в голове пока нет мыслей о том, какой парадигмой программирования он пользуется для названия переменных, функций и синтаксиса в целом.

Постепенно, когда программы становятся сложнее и больше, появляется естественное желание строить программу понятным образом, и писать код так, чтобы его было удобно читать. Впрочем, это желание может и не появиться сразу. В этом случае оно приходит на следующей стадии, когда над программой начинает работать совместно сразу несколько человек. На этом этапе приходит понимание того, что разработка программы во многом аналогична строительству дома. Так, маленькие программы можно сразу сесть и написать, а если вдруг что-то получится плохо – переписать. В крайнем случае, можно просто начать всё сначала. Это процесс можно сравнить с построением скворечника: если вдруг что-то забудется или получится криво, всегда можно что-нибудь отпилить, приклеить или, в крайнем случае, переделать весь скворечник.

Однако для достаточно больших проектов подобный подход становится невозможным. Подобно строительству двадцатиэтажного здания, написание больших программных продуктов нельзя начинать без определённого плана. Любая ошибка на ранней стадии может привести к значительным потерям времени и ресурсов, а о переделывании всего проекта сначала не может быть и речи. Представим себе, например, что при закладке фундамента была допущена неточность в расчёте нагрузки на различные его участки, и эта неточность выявилась лишь после того, как достроили десятый этаж. Чтобы подобных проблем не возникало, следует как можно тщательней, с точностью до мельчайших деталей, продумывать структуру будущего программного продукта. Так, с опытом, программист вырабатывает негласные правила, следуя которым он разрабатывает качественное программное обеспечение.

В этом отношении, переход к параллельному программированию от привычных навыков построения последовательных программ является нетривиальным шагом. Обмены данными представляют собой существенное *идейное* усложнение задачи написания программ. Так, вместе с удобством и новыми возможностями, приходят и новые проблемы, например, отсутствие удобного отладчика (отладчик присутствует лишь в некоторых версиях). Кроме того, появляются новые виды ошибок, связанных с синхронизацией процессов. Возникает целый ряд новых вопросов, которые программист должен

держать в голове, чтобы сделать качественный продукт. И поскольку эти вопросы являются следствием перехода к новой парадигме программирования, ранее они не могли прийти в голову разработчика. Со временем, и для этого вида программирования будут выработаны мнемонические правила, которые, по сути, будут являться плодами опыта разработки приложений.

Таким образом, следует понимать, что разработка параллельных программ является новой деятельностью, расширяющей привычное понятие о программировании. Выделим основные моменты, отличающие разработку параллельных приложений с использованием MPI:

- разработка MPI-приложения, по сути, есть создание не одной программы, а нескольких, по копии на каждый запущенный процесс. При этом все эти программы содержатся в едином коде;
- при выполнении MPI-программы даже одного и того же кода для всех процессов, разные процессы выполняют свою последовательность команд с разной скоростью. По этой причине, следует всегда помнить о том, для каких операций необходима синхронизация процессов. При этом, синхронизирующие функции представляют собой контрольные точки, в которых все процессы догоняют друг друга;
- следует тщательно следить за отправкой и получением сообщений, особенно когда процесс-отправитель и процесс-получатель выполняют совершенно разный код;
- оптимизация зависит как от реализации, так и от системы. Оптимальность MPI-приложения почти никогда не кросс-платформенна, то есть, программа, оптимизированная под определённую вычислительную систему, может оказаться совершенно неоптимальной для другой.

Эти и другие моменты, появляющиеся в результате использования MPI, следует тщательно продумывать на стадии разработки сложных продуктов, в дополнение к структуре каждой из программ. Ввиду меньшей гибкости библиотеки MPI по сравнению, например, с C++, построение MPI-C++ проекта может повлечь за собой изменения некоторых привычных правил программирования на C++. Другими словами, замысловатое распараллеливание даже простой программы может стать трудным вызовом разработчику.

В заключение скажем, что, в отличие от «правил хорошего тона» построения последовательных программ, по которым сегодня существует достаточно много пособий, подобных руководств для параллельного программирования крайне мало. Это объясняется специфичностью построения оптимальных программ по отношению к системе, на которой предполагается вести расчёты.

3. Библиотека MPI

В данном разделе мы переходим непосредственно к практике параллельной реализации программ при помощи MPI. Существуют реализации библиотек функций MPI для языков Fortran, C и C++. Все программы в данной работе реализованы на языке C++, поэтому в дальнейшем предполагается, что читатель знаком с основами этого языка.

3.1. Общая схема

Итак, пусть в нашем распоряжении имеется система с установленным MPI (о том, как установить одну из существующих реализаций MPI-библиотек на Linux или Windows рассказано в соответствующих разделах). Покажем на простом примере, как работает параллельно реализованная программа на такой системе. Сначала предположим, что у нас уже есть написанная программа, которую мы хотим запустить. Обычно это делается строкой следующего вида

```
> mpirun -n 4  application.exe
```

При этом запускается четыре (за количество отвечает параметр -n) самостоятельных процесса application.exe (имя указанно вторым параметром). Эти процессы хоть и одинаковы, но могут быть различимы. В частности, каждый из них имеет уникальный идентификатор, присваиваемый ему операционный системой. Кроме того, команда mpirun заботится о том, чтобы все запущенные процессы знали общее число запущенных ей процессов и их идентификаторы.

Отметим, что если выполнить указанную выше команду дважды, будет запущено две группы по четыре процесса. Группы при этом будут независимы, и обмен данными будет возможен лишь внутри групп, но не между группами.

Вопросы о распределении процессов по компьютерам/процессорам/ядрам решаются на уровне настройки самого MPI и будут обсуждаться в последних разделах.

3.2. Различаем процессы

Теперь, когда нам известно, что происходит при запуске MPI-программ, рассмотрим код программы в примере 1. Это простейшая программа, позволяющая понять то, о чём следует помнить при написании параллельных программ.

```
1   #include <stdio.h>
2   #include "mpi.h"
3
4   int main()
5   {
6     int MyRank, Size;
7     MPI_Init(NULL, NULL);   // инициализировать MPI
8     MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);   //узнать ранг
9     MPI_Comm_size(MPI_COMM_WORLD, &Size);     //узнать число процессов
10    printf("Process %d [of %d] is initialized\n", MyRank, Size);
11                                              //вывести номер процесса на экран
12    MPI_Finalize();             //завершить работу MPI
13    return 0;                   //завершить работу программы
14  }
15
```

Пример 1. Пример простой MPI программы.

В первых двух строках подключаются заголовки библиотеки стандартного ввода/вывода stdio.h и библиотеки MPI mpi.h, необходимой для обмена данными между процессами. В строке 7 мы встречаем первую MPI-функцию: MPI_Init(). Эта функция выполняет две важные операции:

- активирует MPI, а потому должна располагаться до других MPI-функций (за небольшим исключением);
- инициализирует *коммуникатор* MPI_COMM_WORLD, содержащий информацию, о запущенных командой mpirun процессах.

К коммуникаторам мы вернёмся несколько позже, а пока следует запомнить, что есть особый коммуникатор MPI_COMM_WORLD , содержащий информацию о всех процессах.

У функции MPI_Init() два аргумента, которые пока несущественны, и мы положим их равными NULL. До функции MPI_Init() может быть написан любой код без использования функций MPI. Теперь, когда MPI инициализирован, мы вправе пользоваться его полным функционалом.

Первое, что теперь следует сделать – это научиться различать процессы, чтобы не дублировать вычисления, а распределять их оптимальных образом. Для этой цели вызываются функции MPI_Comm_rank() и MPI_Comm_size(), в результате чего каждый процесс узнаёт свой уникальный ранг и общее число процессов. В ходе исполнения программы ранг, полученный процессом в начале, не меняется. Общее число процессов

есть натуральное число N (задается пользователем при запуске программы, может быть любым), а ранги изменяются в пределах от 0 до N-1 и не повторяются (говоря точнее, это ранг в коммуникаторе `MPI_COMM_WORLD`).

Этот шаг крайне важен и присутствует в любой программе, потому как, используя эти функции, можно различать процессы внутри программы, разделяя работу между ними. Кроме того, разделение также может происходить по-разному в зависимости от заданного числа процессов.

Функции `MPI_Comm_rank()` и `MPI_Comm_size()` имеют по два параметра, первый из них – коммуникатор `MPI_COMM_WORLD`, а второй есть адрес переменной типа **`int`**, по которому записывается результат. Они будут подробнее рассмотрены в разделе о коммуникаторах. Следующей операцией каждый из запущенных процессов выводит строку со своим рангом и общим числом процессов. В результате, при N=4, например, поток вывода будет содержать в некотором порядке строки

```
Process 0 [of 4] is initialized
Process 1 [of 4] is initialized
Process 2 [of 4] is initialized
Process 3 [of 4] is initialized
```

Программа завершается функцией `MPI_Finalize()`, роль которой противоположна роли функции `MPI_Init()`: она заканчивает работу MPI и обычно выполняется непосредственно перед завершением работы программы.

Итак, подведём итог:

- для работы с MPI необходимо подключить заголовок `mpi.h`;
- все MPI-функции необходимо располагать между `MPI_Init()` и `MPI_Finalize()`;
- поток вывода у всех процессов общий;
- MPI имеет выделенный коммуникатор `MPI_COMM_WORLD`, который содержит описание всех задействованных в программе процессов. Его следует просто запомнить.
- на уровне кода запущенные процессы различаются по уникальному рангу в коммуникаторе `MPI_COMM_WORLD`, ранги принимают значения от 0 до N-1 и не повторяются (N – общее число процессов);

Кроме того важно понимать:

- запуск MPI-программы командой `mpirun` вызывает параллельное выполнение N экземпляров одного и того же кода;
- эти процессы часто удобно представлять как ячейки таблицы 1xN, пронумерованные числами от 0 до N-1 (например, слева направо). Это тривиальное соображение часто способствует пониманию процессов в MPI;
- исполнение кода может происходить (и происходит) неравномерно, поэтому при запуске программы примера 1 вывод процессов может происходить в разном порядке.

Умея выделять каждый процесс и зная общее их число, можно делить вычислительную нагрузку на части и выполнять её в параллельном режиме. Теперь необходимо сделать второй важный шаг — научиться передавать и получать данные от других процессов.

3.3. Функции передачи данных

Рассмотрим теперь механизмы передачи данных и соответствующие им функции MPI. Операции передачи сообщений составляют основу MPI, они, по сути, и есть MPI.

Среди существующих функций передачи сообщений различаются *парные* операции между двумя процессами и *коллективные* действия для одновременного взаимодействия нескольких процессов. Рассмотрим сначала парные операции.

3.3.1. Передача сообщения от одного процесса другому

Обмен данными между процессами в MPI происходит путём посылки *сообщений* (о чём явно свидетельствует название *Message Passing Interface*). Соответственно, эти сообщения можно посылать (*send*) и принимать (*receive*). Рассмотрим следующий пример, поясняющий процесс передачи простого сообщения.

```
1   #include <stdio.h>
2   #include "mpi.h"
3
4   int main()
5   {
6     int MyRank, Size;
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
9     MPI_Comm_size(MPI_COMM_WORLD, &Size);
10  /*********************************************/
11    char message[20];
12    if (MyRank == 0) //если  ранг процесса 0, отправляем сообщение
13    {
14      message = "Hello from process 1";
15      MPI_Send(message, 20, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
16    }
17    if (MyRank == 1) //если  ранг процесса 1, получаем   сообщение
18    {
19      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
20  MPI_STATUS_IGNORE);
21      printf("Message %s received.\n", message);
22    }
23  /*********************************************/
24    MPI_Finalize();
25    return 0;
    }
```

Пример 2. Передача простого сообщения.

Пример 2 содержит немного изменённую программу примера 1, причём все изменения находятся между строками 10 и 22. Данная программа отправляет сообщение `message` от процесса с рангом 0 (`MyRank == 0`) процессу с рангом 1 (`MyRank == 1`). Соответственно, процесс с рангом 1 это сообщение получает. Разберёмся теперь с тем, какие аргументы имеют функции `MPI_Send()` и `MPI_Recv()`.

```
int MPI_Send(
             (void*)       buf,    //  адрес начала сообщения              input
              int          size,   //  количество отправляемых элементов   input
              MPI_Datatype type,   //  тип элементов сообщения             input
              int          rank,   //  ранг процесса-получателя            input
              int          tag,    //  метка сообщения                     input
              MPI_Comm     comm    //  коммуникатор                        input
            );
```

```
int MPI_Recv(
             (void*)       buf,   //  адрес записи принимаемого сообщения output
              int          size,  //  количество получаемых элементов      input
              MPI_Datatype type,  //  тип получаемых элементов             input
              int          rank,  //  ранг процесса-отправителя            input
              int          tag,   //  метка сообщения                      input
              MPI_Comm     comm,  //  коммуникатор                         input
              MPI_Status*  status //  статус получения                    output
            );
```

Первый аргумент обеих функций представляет собой адрес в памяти. В случае отправки это адрес буфера отправки, то есть, массива, содержащего отправляемое сообщение, а в случае получения – это адрес, по которому будет записано полученное сообщение (буфер получения). Так, в приведённом примере 2 это указатель на начало массива *message* и число 20 – количество символов. Обратим внимание на концептуальную особенность MPI: строка 14 содержит определение переменной сообщения только для процесса с рангом 0. Остальные процессы, хоть и выполняют этот же код, будут иметь другие значения переменной *message*. То есть, следует понимать, что одна и та же переменная в один и тот же момент может иметь разные значения на разных процессах.

Далее следует тип передаваемых элементов, причём в особом MPI-формате. Для большинства типов C++, однако, запись в этом формате получается легко: это название типа, написанное заглавными буквами, к которому добавлен префикс `MPI_`. Так, для типа **`int`** это `MPI_INT`, для **`double`** это `MPI_DOUBLE` и так далее. Ниже эти соответствия перечислены явно:

`MPI_CHAR`	**`signed char`**
`MPI_DOUBLE`	**`double`**
`MPI_FLOAT`	**`float`**
`MPI_INT`	**`int`**
`MPI_LONG`	**`long`**
`MPI_LONG_DOUBLE`	**`long double`**
`MPI_SHORT`	**`short`**
`MPI_UNSIGNED_CHAR`	**`unsigned char`**
`MPI_UNSIGNED`	**`unsigned int`**
`MPI_UNSIGNED_LONG`	**`unsigned long`**
`MPI_UNSIGNED_SHORT`	**`unsigned short`**

В примере 2 мы передаём символы и потому указываем тип `MPI_CHAR`. Помимо использования встроенных типов, MPI позволяет расширять этот список посредством механизма производных типов, который будет рассмотрен в соответствующем разделе ниже.

Смысл четвёртого аргумента обеих функций прост: при отправке следует указать, какому процессу отправляется сообщение, а при получении – от какого процесса должно прийти сообщение. Представим теперь, что процесс 0 отправляет два одинаковых по объёму и типу сообщения процессу 1, причём последний получил эти сообщения одновременно. Процессу 1 крайне важно знать, какое именно сообщение было отправлено первым. Для того, чтобы помочь процессу разобраться в этой ситуации, можно дать сообщениям различные пятые аргументы, которые являются *метками сообщений*. Отправленное сообщение будет получено адресатом, если оно имеет такую же метку, какую ждёт получатель сообщения. Таким образом, метка может использоваться как идентификатор сообщения. В примере 2 отправляемое сообщение имеет метку 99. Сообщение с такой же меткой ждёт процесс 1, поэтому сообщение будет получено. В случае если метка несущественна, в качестве этого аргумента можно указать `MPI_ANY_TAG`.

Далее следуют аргументы коммуникатора и статуса сообщения, которые мы пока всюду будем указывать равными `MPI_COMM_WORLD` и `MPI_STATUS_IGNORE`. Отметим лишь тот факт, что коммуникатор играет столь же важную роль в передаче сообщений, что и метка. Так, если в функциях посылки и приёма сообщения указать различные коммуникаторы, то сообщение не будет доставлено.

Обе эти функции *блокирующие* (*blocking*), то есть, останавливают программу до тех пор, пока не завершат своё выполнение. Для `MPI_Send` это означает окончание отправки сообщения, то есть, такой момент, когда буфер отправки можно вновь редактировать и это не повлияет на уже отправленное сообщение. Что касается `MPI_Recv`, эта функция является блокирующей в том смысле, что программа останавливается на ней до тех пор, пока не будет получено ожидаемое ей сообщение. Преимущество блокирующего механизма состоит в том, что он безопасен с точки зрения ошибок синхронизации процессов. Говоря иначе, с таким механизмом трудно что-то испортить. Цена этого удобства – сравнительно медленная работа, поскольку процессы вынуждены ждать полной отправки/получения сообщений. Если же в программе крайне важна производительность, имеет смысл задуматься об использовании *асинхронных*, или *неблокирующих*, передач. Подробнее различные протоколы обмена парными сообщениями рассмотрены в разделе, посвящённом оптимизации MPI-программ.

Объяснённого в данном разделе механизма передачи сообщений достаточно для создания полноценных параллельных версий многих алгоритмов. Однако в прикладных задачах передачи данных устроены достаточно сложным образом, и их реализация посредством приведённых выше функций отправки/получения может оказаться крайне затруднительной, особенно в том случае, когда скорость вычислений играет важную роль. По этой причине в MPI реализовано множество функций, рассмотренных в следующем разделе, которые помогают решить эту проблему.

Отметим важные моменты при написании парных функций приема-передачи сообщений. При приёме сообщения

- буфер памяти должен быть достаточным для приема сообщения, тип элементов передаваемого и принимаемого сообщения должны совпадать; при нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения;
- при необходимости приема сообщения от любого процесса-отправителя для параметра *`source`* может быть указано значение `MPI_ANY_SOURCE`,
- при необходимости приема сообщения с любым тегом (меткой) для параметра *`tag`* может быть указано значение `MPI_ANY_TAG`;
- функция `MPI_Recv` является блокирующей для процесса-получателя, т.е. его выполнение приостанавливается до завершения работы функции. Таким образом, если по каким-то причинам ожидаемое для приема сообщение будет отсутствовать, выполнение параллельной программы будет блокировано (*deadlock*).

При передаче сообщения

- процессы, между которыми выполняется передача данных, обязательно должны принадлежать коммуникатору, указываемому в функциях при отправке о получении;
- параметр *`tag`* используется только при необходимости различать передаваемые сообщения, в других случаях в качестве значения параметра может быть использовано произвольное целое число или `MPI_ANY_TAG`.

3.3.2. Одновременное выполнение передачи и приема

В параллельных программах достаточно часто необходимо отправить данные одному процессу и, в то же время, получить сообщение от другого процесса. Реализация

обменов такого рода при помощи обычных парных операций передачи данных неэффективна и достаточно трудоемка. Кроме того, реализация должна гарантировать отсутствие ситуаций, которые могут возникать, например, когда два процесса начинают передавать сообщения друг другу с использованием блокирующих функций передачи данных, то есть, каждый из процессов, прежде чем перейти к получению, должен дождаться полной отправки данных, что оказывается невозможным. Достижение эффективного и гарантированного одновременного выполнения операций передачи и приема данных может быть обеспечено при помощи функции MPI:

```
int MPI_Sendrecv (
            (void*)        sbuf,   // адрес начала передаваемого сообщения  input
             int           ssize,  // количество передаваемых элементов     input
             MPI_Datatype  stype,  // тип передаваемых данных               input
             int           dest,   // номер процесса-получателя             input
             int           stag,   // метка отправляемого сообщения         input
            (void*)        rbuf,   // адрес записи принимаемого сообщения  output
             int           rsize,  // количество принимаемых элементов      input
             MPI_Datatype  rtype,  // тип принимаемых элементов             input
             int           source, // номер процесса-отправителя            input
             int           rtag,   // метка принимаемого сообщения          input
             MPI_Comm      comm,   // коммуникатор                          input
             MPI_Status*   status  // статус выполнения операции           output
            );
```

Как следует из описания, функция `MPI_Sendrecv` передает сообщение, описываемое параметрами (*sbuf, ssize, stype, dest, stag*), процессу с рангом *dest* и принимает сообщение в буфер, определяемый параметрами (*rbuf, rcount, rtype, source, rtag*), от процесса с рангом *source*. В функции `MPI_Sendrecv` для передачи и приема сообщений применяются разные буферы. Когда сообщения имеют одинаковый тип, в MPI имеется возможность использования единого буфера для передачи и приема сообщений:

```
int MPI_Sendrecv_replace(
           (void*)       buf,    // адрес сообщения/места записи    input/output
            int          size,   // количество элементов                  input
            MPI_Datatype type,   // тип данных                            input
            int          dest,   // номер процесса-получателя             input
            int          stag,   // метка отправляемого сообщения         input
            int          source, // номер процесса-отправителя            input
            int          rtag,   // метка принимаемого сообщения          input
            MPI_Comm     comm,   // коммуникатор                          input
            MPI_Status*  status  // статус выполнения операции           output
            );
```

Пример использования функций для одновременного выполнения операций передачи и приема приведен в разделе 5 при разработке параллельной программы молекулярной динамики.

3.3.3. Синхронизация

Как было отмечено выше, процессы, запущенные MPI, не обязаны выполняться синхронно. Более того, они почти всегда выполняются асинхронно. Это означает, что если даже все они выполняют одну и ту же последовательность команд, то через некоторое время может возникнуть ситуация, в которой один процесс выполнил больше команд, чем остальные. Это явление может приводить к неправильной работе программы. Приведём следующий пример. Пусть имеется некоторый файл, с которым работают все

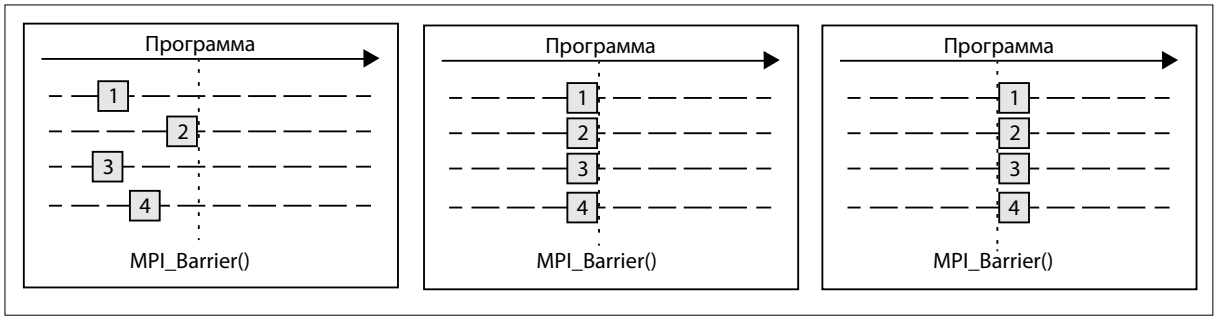

Рис. 4. Схема работы функции `MPI_Barrier()`.

процессы. Каждый из них должен сначала его прочитать, а затем что-то в него записать. В случае если один из процессов убежит вперёд, прочитает весь файл и что-то в него допишет, остальные процессы будут иметь дело с уже изменённым файлом. Чтобы избежать подобных проблем, используется функция

```
int MPI_Barrier(
                MPI_Comm     comm   //  коммуникатор                          input
               );
```

которая ждёт, пока все процессы коммуникатора (для `MPI_COMM_WORLD` это вообще все процессы) вызовут её. Таким образом, происходит синхронизация процессов, после чего они продолжают свою работу. В случае упомянутой выше совместной работы с файлом следует сначала произвести чтение, затем поставить барьер, а уже потом производить изменения.

С точки зрения оптимизации программы барьер обычно является фактором, замедляющим работу программы, а потому его обильного использования следует избегать.

3.3.4. Коллективные передачи данных

BROADCAST

Рассмотрим ситуацию, когда у одного процесса есть некоторые данные, которые необходимо передать всем остальным процессам. Эта задача может быть решена путём использования рассмотренных выше функций передачи и получения сообщения, однако MPI позволяет решить эту проблему проще. Для этого используется функция `MPI_Bcast()` (*broadcast*, широковещание). Идейно эта функция эквивалентна последовательному вызову функции `MPI_Send()` для каждого процесса. Разница заключается в том, что, во-первых, использование этой функции более кратко и понятно, а, во-вторых, она может быть реализована в библиотеке MPI оптимальным образом, а потому значительно быстрее последовательных вызовов `MPI_Send()` выполнять поставленную задачу. Прототип данной функции выглядит следующим образом.

```
int MPI_Bcast(
              (void*)       buf,    //  адрес сообщения/места записи input/output
               int          size,   //  количество передаваемых элементов   input
               MPI_Datatype type,   //  тип передаваемых элементов          input
               int          root,   //  ранг процесса-отправителя           input
               MPI_Comm     comm    //  коммуникатор                        input
             );
```

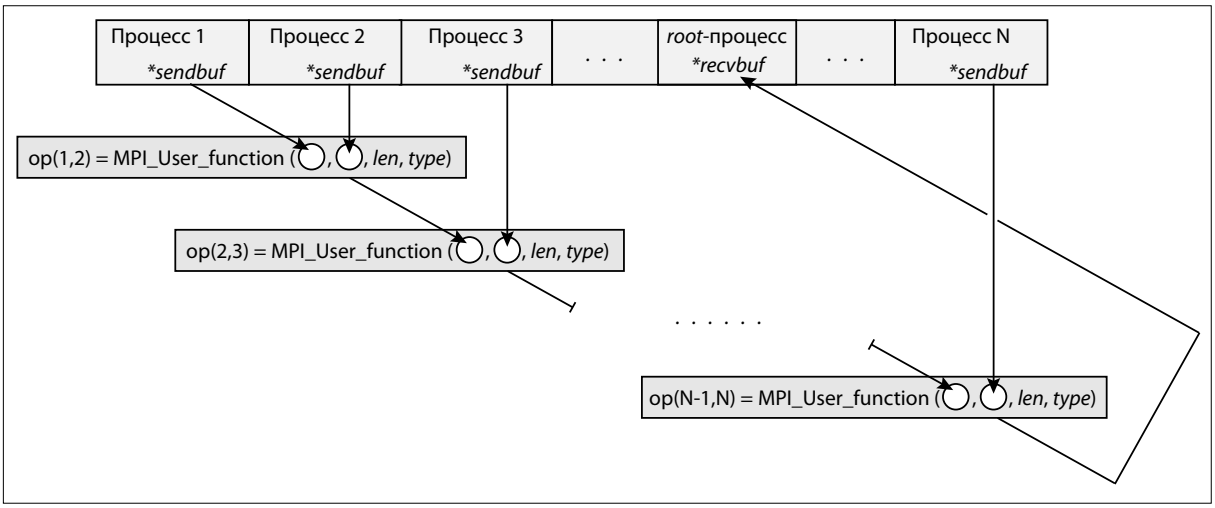

Рис. 5. Схема работы функции `MPI_Reduce`.

При её вызове осуществляется отправка процессом *root* сообщения *buf*, состоящего из *size* элементов типа *type*, всем процессам коммуникатора *comm* (в том числе и себе). Адрес исходного сообщения совпадает с адресом, по которому будет записан результат, и содержится в первом аргументе. Процесс, отправляющий сообщение, обычно называют главным или рут-процессом (*root*). У него совпадают адрес отправляемого сообщения и адрес, по которому будет записано полученное.

Для того, чтобы воспользоваться этой функцией, необходимо выполнить её вызов всеми процессами коммуникатора (в случае коммуникатора `MPI_COMM_WORLD` вообще всеми процессами). Схема работы данной функции приведена на верхней схеме рисунка 5. Ещё раз отметим, что функция `MPI_Bcast()` производит так называемую «широковещательную рассылку», то есть передает одни и те же данные от процессора с номером *root* ко всем процессам группы. Можно сказать, что эта функция является крайне популярной в MPI приложениях, в потому мы начали наше изложение коллективных передач данных именно с неё.

REDUCE

Вторая крайне важная коллективная операция осуществляет *редукцию*, то есть, собирает данные у процессов и производит с ними некоторое преобразование. Это может быть как вычисление некоторой формулы, например, суммы или произведения, так и некоторый анализ полученного набора, например, поиск максимального элемента. Данная функция имеет следующий синтаксис.

```
int MPI_Reduce(
               (void*)       sendbuf,   //  данные сообщения                  input
               (void*)       recvbuf,   //  адрес получения                  output
                int          size,      //  количество передаваемых элементов input
                MPI_Datatype type,      //  тип передаваемых элементов         input
                MPI_Op       operation, //  коллективная операция              input
                int          root,      //  ранг процесса-получателя           input
                MPI_Comm     comm       //  коммуникатор                       input
              );
```

При её вызове собираются данные из всех процессов, записанные по адресу *sendbuf* (их тип и объём, как и ранее, определяются параметрами *type* и *size*) и к ним применяется операция *operation*, после чего результат записывается процессу *rank* по

адресу *recvbuf*. При этом подразумевается, что применяемая операция ассоциативна, то есть, выполняется тождество

$$(a \circ b) \circ c = a \circ (b \circ c),$$

где через a, b, c обозначены объекты типа *type*, а через $\circ$ обозначена операция *operation*. В простейшем случае, когда, например, a, b и c есть просто числа, а в качестве операции взято сложение, данное тождество выполняется. Однако для более сложных типов данных и операций его следует проверять. Неассоциативность операции может привести к неопределённому результату, поскольку порядок применения операции не оговаривается в спецификации, а потому может зависеть от конкретной реализации библиотеки MPI.

Что касается порядка слагаемых, то предполагается, что они расположены по умолчанию в порядке возрастания рангов процессов, начиная с нулевого.

Применяемая операция *operation* может быть выбрана из следующего списка:

MPI_MAX	максимум	MPI_LOR	логическое «или»
MPI_MIN	минимум	MPI_BOR	побитовое «или»
MPI_SUM	сумма	MPI_LXOR	логическое исключающее «или»
MPI_PROD	произведение	MPI_BXOR	побитовое исключающее «или»
MPI_LAND	логическое «и»	MPI_MAXLOC	значение и ранг максимума
MPI_BAND	побитовое «и»	MPI_MINLOC	значение и ранг минимума

Если в качестве аргумента в функцию передаётся массив, то операция будет применена отдельно для каждого индекса, или, иначе говоря, покоординатно (если рассматривать массив как вектор).

Замечательной является возможность создания собственных операций, которые после можно использовать для редукции. Такая возможность снимает ограничения на действия, которые могут производиться с передаваемыми данными. Единственное условие, которое должно выполняться для новой операции — она должна быть ассоциативна, то есть, не зависеть от расстановки скобок. Операция создается с помощью следующей функции:

```
int MPI_Op_create(
                  MPI_User_function* function,  // указатель на функцию        input
                  int                commute,   // коммутативность[true/false] input
                  MPI_Op*            operation  // указатель на операцию      output
                 );
```

Здесь аргумент *commute* равен `true`, если определённая пользователем функция коммутативна и `false` в противном случае. Третий аргумент представляет собой указатель на объект операции, имеющий тип `MPI_Op`, и который можно передавать функции `MPI_Reduce()`. Передача в качестве аргумента того, является ли операция коммутативной, связана с оптимизацией выполнения редукции. Можно ожидать, что в случае коммутативной операции редукция будет работать быстрее по сравнению с некоммутативным аналогом. Это, однако, вновь зависит от того, какая именно библиотека MPI используется. Итак, мы разобрались со вторым и третьи аргументами.

Первый аргумент представляет собой указатель на функцию следующего вида

```
void MPI_User_function(
                       (void*)          arg,      // аргумент 1                 input
                       (void*)          argres,   // агрумент 2          input/output
                        int *           size,     // размер массивов аргуметов input
                        MPI_Datatype*   datatype  // тип данных аргументов      input
                        );
```

Пусть мы хотим выполнить редукцию с созданной нами операцией. Чтобы лучше понять, что обозначают аргументы функции, представим на секунду, что у нас уже есть операция *op*, которую мы хотим использовать. Посмотрим, что происходит, когда осуществляется вызов функции MPI_Reduce() с *op* в качестве пятого аргумента. В этот момент функция MPI_User_function, определяющая операцию *op*, в качестве четвёртого аргумента получает указатель на значение типа данных, указанного четвёртым аргументом MPI_Reduce(). После этого для каждой пары процессов (0, 1), (1, 2), ..., (*N*-2, *N*-1) происходит последовательное вычисление результата действия операции *op* на данные, полученные от процессов пары. При этом вычисление операции *op* для пары (i, j) эквивалентно вычислению функции MPI_User_function , которая получает в качестве первого аргумента *arg* данные от процесса с номером i, а в качестве второго аргумента *argres* – данные от процесса j. Идея данного процесса приведена на рисунке 5. Отметим также, что в некоторых случаях (например, для коммутативной операции) алгоритм действия функции MPI_Reduce может быть иным, но ответ будет получаться такой же, как и при описанном порядке действий.

Помимо требования ассоциативности, операция, задаваемая функцией MPI_User_function должна действовать так, чтобы её результат был представим в виде *argres[i]=arg[i]* op^i *argres[i]* для каждого *i* от 0 до *size*-1, где op^i есть некоторая операция (возможно, различная для различных i). Другими словами, определяемая функцией операция должна представлять собой *покоординатное* применение некоторой операции к *size*-мерным векторам (запрещается смешивать различные компоненты векторов). Например, функция не должна содержать выражения *argres[i]=arg[i]+argres[i+1]*. Нарушение этого правила может привести к некорректной работе программы.

После того, как операция была успешно применена и больше не нужна, можно удалить занимаемую ей память с помощью функции

```
int MPI_op_free(
                MPI_Op* op // указатель на определённую пользователем операцию  input
                );
```

ПРОЧИЕ ФУНКЦИИ КОЛЛЕКТИВНЫХ ПЕРЕДАЧ

Помимо перечисленных нами функций в MPI имеется возможность осуществлять и более сложные обмены данными. Ниже мы проиллюстрируем схему их работы, так же схемы работы можно увидеть на рис.6. Для более подробного описания читателю следует обратиться к [11].

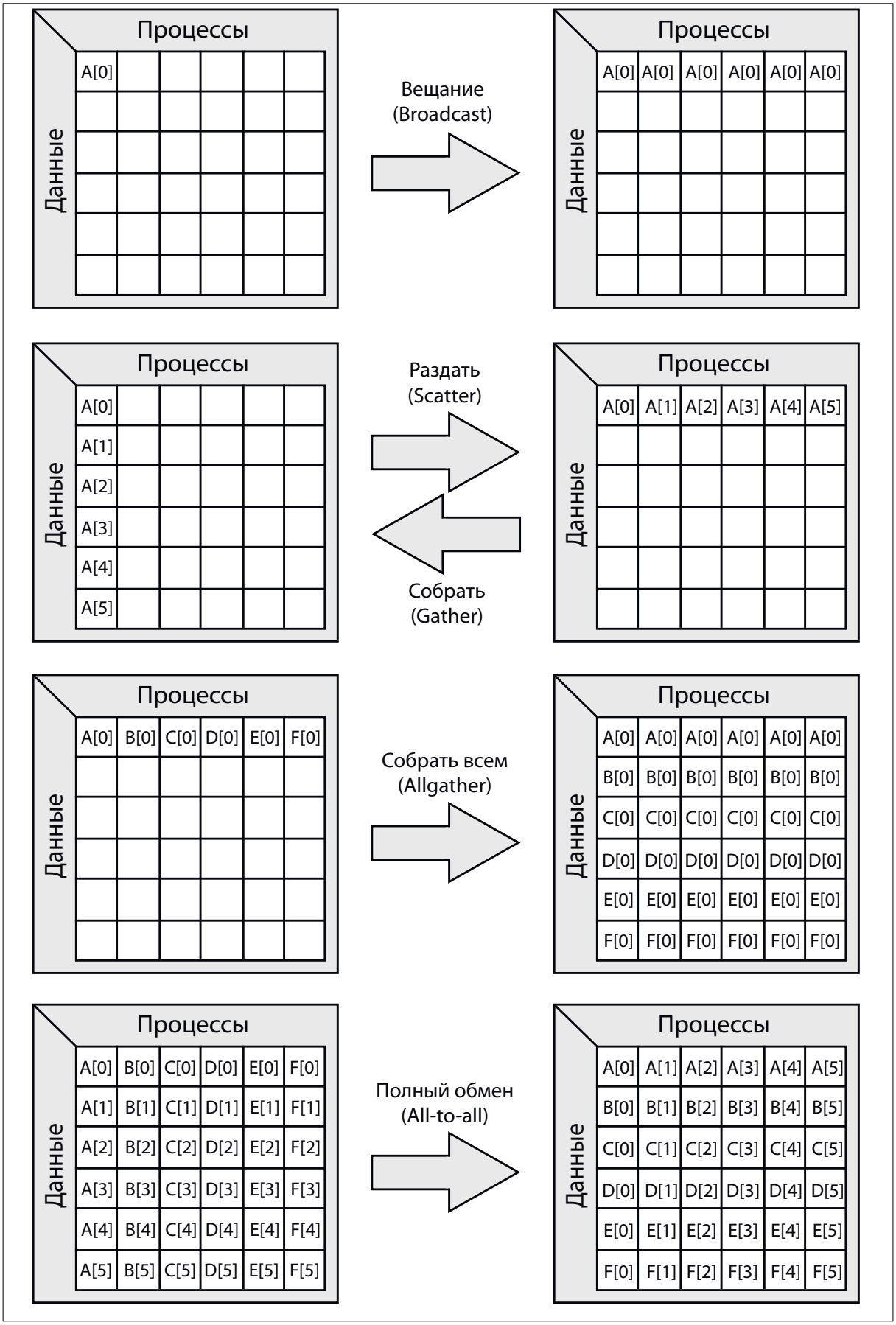

Рисунок 6. Схемы коллективных передач данных.

```
int MPI_Gather(
          (void*)       sendbuf,  // адрес буфера отправки                          input
           int          sendsize, // число элементов в отправляемом сообщении   input
           MPI_Datatype sendtype, // тип посылаемых данных                          input
          (void*)       recvbuf,  // адрес буфера получения                       output
           int          recvsize, // число элементов придёт от каждого процесса input
           MPI_Datatype recvtype, // тип получаемых данных                          input
           int          root,     // номер собирающего процесса                     input
           MPI_Comm     comm      // коммуникатор                                   input
           );
```

Функция `MPI_Gather` собирает данные от каждого процесса и передаёт их root-процессу в последовательности, соответствующей порядку следования рангов процессов в коммуникаторе. Аргумент *recvbuf* игнорируется всеми процессами, отличными от root. Помимо `MPI_Gather()`, существует также вариант `MPI_Gatherv()`, отличающийся лишь тем, что позволяет передачу различных объёмов данных от различных процессов. В этом случае пятый аргумент становится массивом целых чисел, компоненты которого содержат число элементов, получаемых от процессов в порядке возрастания рангов. Кроме того, добавляется новый шестой аргумент *displacements* (прежний шестой становится седьмым и т.д.) – массив целых чисел, i-я компонента которого обозначает смещение по отношению к *recvbuf*, по которому будут записываться полученные от i-го процесса данные.

```
int MPI_Scatter(
          (void*)       sendbuf,  // адрес буфера отправки                       input
           int          sendsize, // число элементов в отправляемом сообщении input
           MPI_Datatype sendtype, // тип посылаемых данных                       input
          (void*)       recvbuf,  // адрес буфера получения                     output
           int          recvsize, // число элементов раздаётся
                                     каждому процесса                           input
           MPI_Datatype recvtype, // тип получаемых данных                       input
           int          root,     // номер собирающего процесса                  input
           MPI_Comm     comm      // коммуникатор                                input
           );
```

Обратная к gather-функциям `MPI_Scatter` раздает данные от root остальным процессам коммуникатора. В этом случае первый аргумент имеет смысл только для root-процесса. Как и в случае с `MPI_Gather()`, имеется вариант `MPI_Scatterv()`, позволяющий неравномерную раздачу данных по процессам. Отличие состоит лишь в том, что в этом случае *displacements* занимает место не шестого, а третьего аргумента.

```
int MPI_Allgather(
          (void*)       sendbuf,  // адрес буфера отправки                    input
           int          sendsize, // число элементов в отправляемом
                                     сообщении                                input
           MPI_Datatype sendtype, // тип посылаемых данных                    input
          (void*)       recvbuf,  // адрес буфера получения                  output
           int          recvsize, // число элементов придёт
                                     от каждого процесса                      input
           MPI_Datatype recvtype, // тип получаемых данных                    input
           MPI_Comm     comm      // коммуникатор                             input
           );
```

Сравнивая аргументы `MPI_Gather()`, видно отсутствие аргумента root. В этом смысле, рассматриваемая функция аналогична `MPI_Gather()`, но сбор данных осущест-

вляется каждым процессом от всех остальных. В идейном смысле, эта функция эквивалентна выполнению `MPI_Gather()` на каждом процессе. Запись в буфер получения происходит, как обычно, в порядке возрастания рангов. Несимметричная версия `MPI_Allgather()` имеет массив элементов типа **`int`** *displacements* шестым аргументом, и *`recvsize`* становится массивом.

```
int MPI_Alltoall(
          (void*)       sendbuf,  // адрес буфера отправки                  input
           int          sendsize, // число элементов раздаётся
                                     каждому процессу                       input
           MPI_Datatype sendtype, // тип посылаемых данных                  input
          (void*)       recvbuf,  // адрес буфера получения                output
           int          recvsize, // число элементов придёт
                                     от каждого процесса                    input
           MPI_Datatype recvtype, // тип получаемых данных                  input
           MPI_Comm     comm      // коммуникатор                           input
           );
```

Функция `MPI_Alltoall` является расширением `MPI_Gather()`, при котором каждый процесс отправляет различные данные различным адресатам. Так, i-й блок данных *`sendbuf`*, посланный процессом j окажется в результате на j-м месте массива *`recvbuf`* i-го процесса. В этом случае происходит полный обмен данными между всеми процессами. Из соображений оптимизации программ данную функцию не рекомендуется использовать ввиду того, что она задействует достаточно много памяти и в некоторых реализациях MPI может работать крайне медленно. В отличие от перечисленных выше функций коллективных передач данных, данная функция имеет сразу две функции-обобщения: `MPI_Alltoallv()` и `MPI_Alltoallw()`. Использование этих функций происходит крайне редко и нежелательно с точки зрения производительности, а потому мы не будем на них останавливаться.

```
int MPI_Allreduce(
          (void*)       sendbuf,  // адрес буфера отправки                  input
          (void*)       recvbuf,  // адрес буфера получения                output
           int          size,     // число отправляемых элементов           input
           MPI_Datatype type,     // тип данных у элементов                 input
           MPI_Op       op,       // операция [как в MPI_reduce()]          input
           MPI_Comm     comm      // коммуникатор                           input
           );
```

Аналог `MPI_Reduce()`, при котором полученный результат получают все процессы. Идейно аналогичен вызову `MPI_Reduce()` одним из узлов коммуникатора, за которым следует `MPI_Bcast()`.

```
int MPI_Reduce_scatter_block(
          (void*)       sendbuf,  // адрес буфера отправки                  input
          (void*)       recvbuf,  // адрес буфера получения                output
           int          size,     // число получаемых элементов             input
           MPI_Datatype type,     // тип данных у элементов                 input
           MPI_Op       op,       // операция [как в MPI_reduce()]          input
           MPI_Comm     comm      // коммуникатор                           input
           );
```

Сначала данная функция в каждом процессе берёт массив длиной *N*size* с началом в *sendbuf*, где *N* – число процессов в коммуникаторе. При этом тип элементов массива задан четвёртым параметром функции. После этого к полученному набору векторов применяется покоординатно `MPI_Reduce()`, как всегда в порядке возрастания рангов. Полученный в результате вектор делится на *N* частей, каждая из которых содержит *size* элементов, после чего эти части раздаются процессам (опять-таки, в порядке возрастания рангов). По принципу работы функция эквивалента последовательному вызову `MPI_Reduce()` и `MPI_Scatter()`, отсюда и берётся её название, однако реализация в библиотеке может оказаться несколько быстрее. Более общим аналогом этой функции является `MPI_Reduce_scatter()`. Её отличие состоит в том, что она позволяет неравномерно разделить результат редукции между процессами, то есть, разным процессам может достаться разное число компонент. Для этой цели третий аргумент из целого числа превращается в массив целых чисел, описывающих число компонент вектора, которое достанется различным процессам в результате выполнения функции.

```
int MPI_Scan(
          (void*)        sendbuf,  // адрес буфера отправки                 input
          (void*)        recvbuf,  // адрес буфера получения               output
           int           size,     // число отправляемых элементов          input
           MPI_Datatype  type,     // тип данных у элементов                input
           MPI_Op        op,       // операция [как в MPI_reduce()]         input
           MPI_Comm      comm      // коммуникатор                          input
           );
```

Аналог `MPI_Reduce()`, но i-й процесс получает результат применения операции *op* к набору данных из процессов с рангами 0, ..., i. Таким образом, каждый процесс получит индивидуальное значение в качестве результата. Разновидность `MPI_Exscan()` (*exclusive scan*) данной функции делает то же самое, но для i-го процесса результат будет содержать результат последовательного выполнения операции к набору данных процессом с рангами 0, ..., i-1. Результат для процесса к с нулевым рангом не определён.

Как уже было отмечено, эти функции несколько сложнее рассмотренных нами ранее и для их понимания рекомендуется изучить примеры их работы [11].

3.3.5. Пример использования коллективных передач

Рассмотрим пример вычисления числа π методом Монте-Карло. Идея алгоритма следующая: имеется генератор случайных пар чисел — координат точек, распределённых равномерно в квадрате $[-1;1]\times[-1;1]$. С помощью этого алгоритма мы кидаем точки в квадрат и смотрим, сколько из них попало в круг радиуса 1 с центром (0;0). Тогда имеет место следующее утверждение: отношение числа точек, попавших в круг, и общего числа точек будет равно π/4. Мы воспользуемся им без доказательства. Отсюда, зная число попавших и число не попавших точек, можно оценить число π. Чем больше точек будет рассмотрено по этому алгоритму, тем ближе результат будет к числу π. Организовать параллельную работу можно следующим образом: каждый процесс будет бросать в квадрат *N* точек и определять, сколько из случайных точек попало в круг. Таким образом, общее число рассмотренных точек будет равно *pN*, где *p* — число процессов. Потом данные со всех процессов собираются при помощи `MPI_Reduce()`.

Стоит отметить функцию `srand(MyRank)` в строке 26. Она обеспечивает различные случайные точки для каждого процесса, то есть последовательность случайных чисел начинается с разного места, и для разных процессов будут сгенерированы точки с разными координатами. Если не пользоваться подобной функцией, то каждый процесс бу-

дет генерировать одинаковые случайные точки и ценность параллельного алгоритма пропадает.

Реализацию на языке C++ можно посмотреть в примере 3.

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

int main()
{
  int    MyRank, Size, N, in = 0, total;
  double x, y, pi;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
  MPI_Comm_size(MPI_COMM_WORLD, &Size);

  if (MyRank == 0)   // нулевой процесс получает число точек с клавиатуры
  {
    printf("Enter number of points per process\n");
    scanf("%d", &N);
  }

  MPI_Barrier(MPI_COMM_WORLD); // ждём, пока первый процесс получит данные
  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);  // раздаём объём работ
процессам

  if (N == 0)       // если ноль - выходим
  {
    return 1;
  }
                    // иначе - инициализируем датчик случайных чисел
  srand(MyRank);
  for (int i=0; i<N; ++i)      // кидаем N точек в квадрат [-1;1]x[-1;1]
  {
    x = (2.0*rand())/RAND_MAX - 1;
    y = (2.0*rand())/RAND_MAX - 1;
    if (x*x + y*y < 1) ++in;  // если попали в круг увеличиваем счётчик
  }
  MPI_Reduce(&in, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
                  //складываем число попаданий и передаём сумму процессу 0

  if (MyRank == 0)
  {
    pi = (4.0*total)/N/Size;
    printf("Pi= %.12lf [error ~ %.12lf]\n", pi, pi- 3,141592653589);
  }

  MPI_Finalize();
  return 0;
}
```

Пример 3. Вычисление числа π: пример применения функций коллективных передач данных

3.4. Учет времени в MPI

Достаточно много было сказано о том, что написание программы с использованием MPI позволяет получить выигрыш во времени в несколько раз, но выигрыш будет сильно зависеть от задачи и от способа написания параллельного алгоритма (поскольку бывают разные подходы к написанию параллельных реализаций одной и той же задачи, обладающие разной эффективностью). Но пока еще не были описаны функции, позволяющие учитывать время работы программы с MPI. Поскольку может понадобиться явно рассматривать время работы одной из функций передачи данных, например, `MPI_Bcast()`, измерение времени работы функции лишь на одном процессе приведет, скорее всего, к неверным результатам, потому что данные могут приниматься процес-

сами не синхронно. Для корректного учета времени в библиотеке MPI используются функция

```
double MPI_Wtime(void); // расчет времени в MPI.
```

Значения на вход не подаются, в качестве выходных данных возвращается значение типа `double`. Функция вычисляет длину промежутка времени в секундах, начиная с определенного момента времени в прошлом. В течение программы этот начальный момент остается неизменным, поэтому величину промежутка времени можно вычислить, вычитая из величины `start = MPI_Wtime()` на момент запуска величину `end = MPI_Wtime()` на момент окончания. Функция запускается на каждом процессе. Рассмотрим пример использования этой функции.

```
1  #include <stdio.h>
2  #include <mpi.h>
3  int main()
4  {
5      int size, rank;
6      double start, end;
7      MPI_Init(NULL, NULL);                    //  инициализировать MPI
8      MPI_Comm_rank(MPI_COMM_WORLD, &rank); //  определить ранг
9      MPI_Comm_size(MPI_COMM_WORLD, &size); //  определить число процессов
10 //  MPI_Barrier(MPI_COMM_WORLD);
11     start = MPI_Wtime();                     //  запустить счетчик времени
12 //  MPI_Barrier(MPI_COMM_WORLD);
13     if (rank==0)
14     {
15         end = MPI_Wtime();              // если Root вычислить время и вывести
16         printf("Hello From Root #%d %lf \n", rank, (end - start));
17     }
18     else
19     {
20         end = MPI_Wtime();           // если не Root,вычислить время и вывести
21         printf("Hello From Slave #%d #%lf \n", rank, (end - start));
22     }
23     MPI_Finalize();                          //  завершить работу MPI
24     return 0;
25 }
```

Пример 4. Использование функции, учитывающей время.

Данный пример демонстрирует то, как меняется время в зависимости от того, синхронно ли начинают функции. Так, если запускать программу с закомментированными функциями `MPI_Barrier(MPI_COMM_WORLD)` (строки 10 и 12), то ото всех процессов придет одинаковый результат. Если же убрать комментарии в строках 10 и 12, результат будет иной за счет того, что процессы выполняются не синхронно, и время их работы будет отличаться в зависимости от процесса.

3.5. Выбор способа параллельной реализации

Параллельная организация задач обычно может быть осуществлена разными способами. Рассмотрим моделирование движения идеального газа из N частиц. Задача состоит в том, чтобы посчитать на каждом шаге положение каждой частицы в следующий момент времени и K макроскопических характеристик газа (температуру, давление, и т.д.), K обычно меньше N. Пусть для решения этой задачи имеется два компьютера, со-

ответственно нами будет запущено два процесса. Возникает вопрос, как следует разделить работу между ними, чтобы расчёты шли как можно быстрее.

Первый вариант – это разделить вычислительную нагрузку поровну: каждый вычисляет динамику *N/2* частиц, передаёт информацию другому, после чего каждый вычисляет *K/2* средних величин. При такой реализации оба процесса равнозначны, поскольку вычислительная нагрузка примерно одинакова.

Однако можно реализовать эту задачу по-другому: первый процесс вычисляет положения молекул в следующий момент, передаёт второму и считает дальше, в то время как второй занимается только расчётом средних величин. При этом узлы не равнозначны: основной счёт проходит на первом узле, а менее затратное вычисление средних величин проходит на втором. Эта неоднозначность позволяет выделить один процесс и назвать его главным, или root-процессом (обычно это тот, у которого нагрузка меньше).

Возникает вопрос, какая из реализаций лучше. Казалось бы, первая: вычисления распределены поровну, поэтому всё будет вычисляться быстрее. Однако это не всегда так: при правильной реализации асинхронных передач данных вторая реализация может оказаться в ряде случаев существенно быстрее.

Способ реализации параллельного алгоритма зависит от конкретной задачи и от конкретной вычислительной системы, на которой будут выполняться расчёты. Как уже не раз было подчёркнуто, оптимизация вычислений не является универсальной для всех вычислительных систем сразу. Напротив, её следует осуществлять индивидуально по отношению к используемым архитектурам.

3.6. Производные типы

Способ обмена сообщениями представляет собой удобный механизм передачи данных. При этом при каждой передаче необходимо уточнять то, что именно мы передаём: строки, целые числа и т.д. Что делать с базовыми типами ясно – в MPI для них имеются типы `MPI_CHAR`, `MPI_INT` и так далее. Однако C++ часто имеет дело и с более сложными типами данных, какими являются структуры и объекты, и далеко не для каждого из них найдётся подходящий тип для передачи. Отсутствие механизма передачи подобных объектов существенно бы усложнило портируемость программ на параллельные архитектуры. Представим на секунду ситуацию, что в нашем распоряжении имеется достаточно объёмная программа, написанная на C++. Кроме того, пусть эта программа осуществляет моделирование некоторого физического процесса. В этом случае логично предположить, что в ней достаточно распространено использование структур и классов, в той или иной мере описывающих реальные объекты. Если перед нами встанет задача параллельной реализации этой программы, то естественным будет желание не трогать существующую иерархию классов, а лишь переписать некоторые функции этих классов так, чтобы программа заработала в параллельном режиме. Но при вычислениях на нескольких процессах не обойтись без передач данных. Тогда имеется два варианта действий: сохранять структуру сложных типов памяти, или переписать всё в массивах. Первый случай часто оказывается более предпочтителен для больших программ.

В качестве простого варианта можно написать функцию передачи и приема данных, используя стандартную функцию передачи данных `MPI_Bcast()` с типом `MPI_BYTE`. Пусть есть структура данных `Object`, тогда «широковещательную рассылку» можно организовать с использованием двух функций из примера 4.

```
void BcastObject(Object &Ob)                //передать Object
{
       int size = sizeof(Object);            // вычислить размер Object
       char *buff = new char[size];          // завести буфер размера size
       memcpy(buff, &Ob, sizeof(Object));    // записать Object в буфер
```

```
        MPI_Bcast(buff, size, MPI_BYTE, 0, MPI_COMM_WORLD ); // передать
        delete[] buff;                          // удалить буфер
}

void ReceiveObject(Object &Ob)              //получить Object
{
        int size = sizeof(Object);           // вычислить размер Object
        char *buff = new char[size];         // завести буфер размера size
        MPI_Bcast(buff, size, MPI_BYTE, 0, MPI_COMM_WORLD ); // получить
        Ob =*((Object*)((char*)buff));  //собрать из полученного сообщения Object
        delete[] buff;                          //удалить буфер
}
```

Пример 4. Возможная реализация функции `MPI_Bcast()` для объекта.

Однако в MPI существует и более удобные (в некотором смысле) специальные средства, позволяющие решить проблему передачи сложных типов. Этот механизм носит название *производных типов* (*derived datatypes*). Это название означает то, что пользователь может на основании имеющегося у него класса или структуры создать новый тип, который будет понятен MPI. После этого построенный тип данных можно будет использовать во всех функциях наравне со встроенными типами, такими как `MPI_DOUBLE` или `MPI_CHAR`.

Для решения этой проблемы в MPI существует специальный механизм *производных типов*, который позволяет MPI, как передавать созданные пользователем структуры и классы.

3.6.1.Схемы типов

Рассмотрим сначала набор пар S следующего вида, который назовём *схемой типа*: $S = \{(t_0, s_0), (t_1, s_1), ..., (t_k, s_k)\}$, где t_i – основной тип (double, int и т.д.), s_i – смещение (в байтах). Такая схема и начальный адрес *buf* полностью определяют разметку сообщения в том смысле, что по адресу $buf + s_i$ находится объект типа t_i. Отметим, что смещения всегда целые, но не обязательно являются положительными. Отрицательное смещение будет означать расположение данных в памяти до точки отсчёта. MPI работает только с такими схемами, описание которых, по сути, и содержат объекты класса `MPI_Datatype`. Поэтому наша задача сводится к тому, чтобы описать созданный нами класс в рамках подобных схем. На уровне реализации мы будем действовать следующим образом. На основе существующей структуры (класса) мы будем создавать схему, которую после будем использовать для передачи данных этой структуры (класса).

Сам MPI не различает, что он передаёт в сообщении. При желании, можно передать массив из чисел типа `double`, объявив его для MPI как массив из символов с элементами `MPI_CHAR`. На этом основан принцип работы примера 4. Зачем тогда нужны производные типы? Для того, чтобы иметь больший контроль над тем что мы передаём. Пример 4 осуществляет передачу «вслепую» всех имеющихся данных, в то время как механизм производных типов позволит отличать друг от друга различные элементы классов и передавать их с большей вариативностью.

Рассмотрим теперь функции, используемые для создания производных типов. Первая функция, которую мы рассмотрим, создаёт новый тип данных, состоящий из последовательного повторения исходного типа. Её прототип имеет следующий вид (*contiguous* - смежный).

```
int MPI_Type_contiguous(
                        int           count,   // число повторений, >=0     input
                        MPI_Datatype  oldtype, // исходный тип              input
                        MPI_Datatype* newtype  // указатель на новый тип   output
                       );
```

Получая в качестве аргументов число и тип данных (то есть, схему *oldtype*), данная функция возвращает новый тип данных, полученный (*count*-1)-кратным приписыванием исходного типа к концу предыдущего. Так, если мы имеем схему *{(double, 0)}* и значение *count* равное 2 на выходе (*newtype*) получим *{(double, 0), (double, 8)}*, поскольку размер исходного типа равен 8 байтам. Отметим, что в некоторых случаях, связанных с выравниванием типов в памяти, приписывание может происходить с пробелами, то есть так, что перед началом очередной копии *oldtype* находилось бы несколько пустых битов. Подробнее этот вопрос будет разобран позже, в разделе о ширине типа.

Следующая функция предлагает другой способ построения карты типа.

```
int MPI_Type_vector(
                int           count,       // число блоков, >=0             input
                int           blocklength, // число элементов в блоке, >=0  input
                int           stride,      // элементов помещается в блоке  input
                MPI_Datatype  oldtype,     // исходный тип                  input
                MPI_Datatype* newtype      // указатель на новый тип       output
               );
```

Полученный в результате тип имеет следующую структуру. Сначала выделяются *count* блоков, каждый из которых имеет размер *stride*size*, где *size* – размер *oldtype* в байтах. Затем в начале каждого блока размечается *blocklength* элементов типа *oldtype*, а остальное место блока остаётся свободным. Иллюстрация такой разметки представлена на рисунке 7.

В качестве примера можно рассмотреть результат выполнения функции MPI_Type_vector(2, 2, 3, *oldtype*, *newtype*), где в качестве исходного типа взят *{(double, 0)}*. Тогда полученный тип будем иметь разметку *{(double, 0), (double, 8), (double, 24), (double, 32)}*.

Наконец, самая важная функция, которая позволяет создать наиболее широкий класс различных схем типов

```
int MPI_Type_create_struct(
             int           count,          // число блоков >=0              input
             int           blocklengths[], // числа элементов в блоках >=0  input
             MPI_Aint      shifts[],       // смещения блоков               input
             MPI_Datatype  oldtypes[],     // массив исходных типов         input
             MPI_Datatype* newtype         // указатель на новый тип       output
            );
```

Тип, который будет создан в результате работы этой функции, строится следующим образом. Для каждого i от 0 до *count-1* выделяется блок, состоящий из *blocklengths*[*i*] элементов типа *oldtypes*[*i*]. Это построение производится так же, как сделала бы функция MPI_Type_contiguous. Затем каждый построенный блок смещается на *shifts*[*i*] (с типом MPI_Aint можно обращаться как с обычным int). Отметим разницу с предыдущей функцией: поскольку тут участвует сразу несколько типов, здесь единица смещения – байт, а не размер исходного элемента. Рассмотри пример:

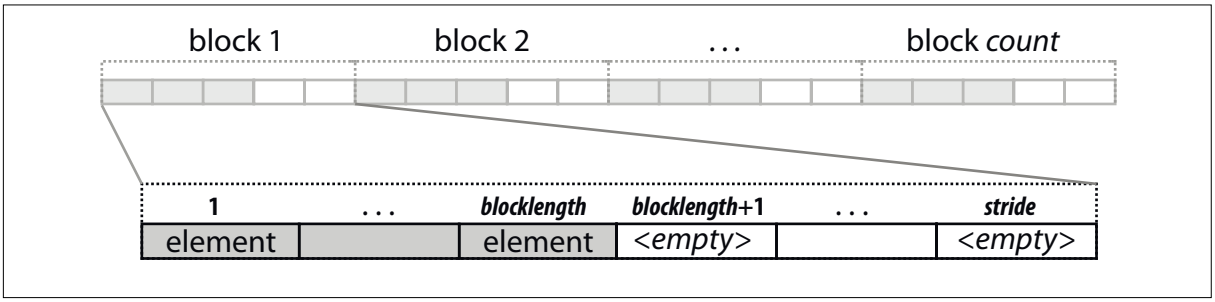

Рис. 7. Иллюстрация типа данных, создаваемого `MPI_Type_vector`.

пусть B = (2, 1, 3), D = (0, 16, 26), T = (`MPI_FLOAT, MPI_DOUBLE, MPI_CHAR`). Тогда вызов `MPI_Type_create_struct(3, B, D, T, newtype)` возвращает тип со схемой *{(float, 0), (float, 4), (double, 16), (double, 24), (char, 26), (char, 27), (char, 28)}*.

Помимо перечисленных, существуют и другие функции, конструирующие новые типы данных для MPI-функций, [11]. Ввиду того, что нам они не пригодятся, мы опустим их описание.

3.6.2. Размеры типов

Представим теперь, что нам необходимо использовать функцию `MPI_Gather` и выделить необходимую для сбора данных память на root-процессе. Возникает вопрос, каков же размер создаваемых нами типов? Можно выделить два различных понятия размера типа.

Первое – это количество байтов, занимаемых именно данными. То есть, если разметка типа имеет вид *{(float, 0), (float, 94)}*, то при подсчёте размер такого типа будут учитываться лишь те участки, которые заняты интересующим нас типом. Так, размер приведённого в примере типа равен 8 байтам. Такой показатель будем называть *размером* (*size*). Получить размер типа можно функцией

```
int MPI_Type_size(
                  MPI_Datatype  datatype,      //    тип данных            input
                  int *         size           //    размер               output
                 );
```

Дать второе определение несколько сложнее. Его идея такова: мы хотим назвать размером это расстояние между первым и последним байтом, занимаемым типом. Этот показатель мы будем называть *шириной* (extent). Для определения ширины типа в MPI используются псевдотипы MPI_UB (upper bound) и MPI_LB (lower bound). Это ненастоящие типы, поскольку они не занимают памяти, то есть, имеют нулевую ширину, а потому не влияют и на размер типа, в который они входят в качестве подтипа. Единственное, зачем нужны эти псевдотипы – это указать начало и конец области памяти, занимаемой всем типом (то есть, границы, определяющие extent).

Рассмотрим следующий пример: пусть `B=(1,1,1), D=(-4,16,26), T=(MPI_LB,MPI_FLOAT,MPI_UB)`. Тогда вызов `MPI_Type_create_struct (3,B,D,T,newtype)` возвращает тип со схемой `{(lb,-4),(float,16),(ub,26)}`. Как мы уже отметили ранее, `ub` и `lb` являются не более чем метками начала и конца области, занимаемой данным типом. Ширина возвращённого данной функции типа равна 26-(-4)=30 (от -4 до 25 включительно).

В общем случае границы типа определяются следующим образом. Рассмотрим тип $S=\{(t_0, s_0),(t_1, s_1), \ldots, (t_k, s_k)\}$, t_i- тип, s_i-смещение. Тогда его нижняя граница lb определяется из следующей системы

$lb(S) = \min_j s_j$, если нет элементов типа MPI_LB,
$lb(S) = \min_j \{s_j$ таких что t_j=MPI_LB$\}$ в противном случае.

Похожим образом, верхняя граница ub

$ub(S) = \max_j s_j + sizeof(t_j) + \varepsilon$, если нет элементов типа MPI_UB,
$ub(S) = \min_j \{s_j$ таких что t_j=MPI_LB$\}$ в противном случае.

Тогда ширина $extent(S)=ub(S)-lb(S)$. Величина ε в последнем выражении есть минимальное целое число, такое что extent(S) делится нацело на $\max_i \{alignof(t_i)\}$ (для большинства типов значения $alignof(t_i)$ и $sizeof(t_i)$ совпадают).

Вернёмся на минуту к функции MPI_Type_contiguous().При вызове этой функции типы записываются друг за другом, исходя не из адреса первого свободного байта после первой копии исходного типа, а из значения ширины типа. Другими словами, последовательная запись объектов исходного типа S происходит через каждые extent(S) байтов. При этом если исходный тип содержал метки MPI_LB/MPI_UB, все они удалятся, кроме самой первой и самой последней (условно говоря, стоящие рядом MPI_LB и MPI_UB «сокращаются»).

Получить значения нижней границы и ширины созданного типа можно при помощи функции MPI_Type_get_extent() со следующим прототипом.

```
int MPI_Type_get_extent(
                   MPI_Datatype   datatype,   // интересующий тип данных    input
                   MPI_Aint *     lb,         // нижняя граница типа       output
                   MPI_Aint *     extent      // ширина типа               output
                  );
```

В качестве аргументов эта функция получает производный тип данных, информация о котором нас интересует, и адреса двух целых чисел, по которым будет записаны нижняя граница и ширина типа-аргумента.

Помимо этой функции, есть ещё одна, MPI_Type_get_true_extent(), которая делает то же, что и MPI_Type_get_extent(), но игнорирует метки MPI_LB и MPI_UB. Игнорирование меток означает, что ширина теперь «истинная», то есть, определяется только данными, отсюда и слово true в названии функции. Порядок и смысл аргументов этой функции совпадают с аргументами аналога, соответственно, прототип имеет следующий вид.

```
int MPI_Type_get_true_extent(
                  MPI_Datatype  datatype,   // интересующий тип данных        input
                  MPI_Aint *    truelb,     // истинная нижняя граница типа  output
                  MPI_Aint *    trueextent  // истинная ширина типа           output
                   );
```

Игнорирование меток может понадобиться, например, в следующем случае.

Рассмотрим простой тип со схемой `t={(int,-4),(lb,0),(int,4),(ub,8)}`. Его ширина (extent) равна 8 байтам, в то время как для хранения одного экземпляра этого типа потребуется 12 байтов. Последняя величина и есть истинная ширина (true extent). Зачем же тогда нужны метки границ в этом типе? Для того чтобы при последовательной передаче нескольких элементов этого типа в буфере сообщения не было пробелов. Ясно, что подобный пример достаточно искусственный, и в простых программах часто можно обойтись без подобных трюков.

Помимо приведённых нами операций, связанных с производными типами, MPI позволяет также копировать производные типы без изменения, копировать с изменением ширины, а также производить прочие полезные операции над типами, подробнее о которых узнать можно в [10] и [11].

3.6.3. Глобальные смещения переменных

В предыдущем разделе были описаны функции, которые можно применять для построения производных типов. Тем не менее, пока они позволяют лишь построить некоторые схемы, которые никак не связанные со структурами (классами) программы. Для того чтобы создавать производные типы, которые отвечали бы имеющимся структурам данных, необходимо получить информацию об организации в памяти элементов внутри этих структур. В решении этой задачи может помочь функция

```
int MPI_Get_address(
                  void *       location, // положение в памяти            input
                  MPI_Aint *   address   // смещение этого положения     output
                  );
```

Предназначение этой функции заключается в том, чтобы возвращать смещения произвольного адреса *location* относительно начала адресного пространства `MPI_BOTTOM`. А именно, если мы объявим новый класс, состоящий из нескольких переменных, применение этой функции помогает понять, каким образом расположены в памяти элементы класса. Это позволит создать производный тип со схемой, дублирующей расположения в памяти данных класса. При этом в большинстве C++ реализаций значение адреса, возвращаемое данной функцией, совпадает с тем, которое возвращает оператор `&`. Однако это, вообще говоря, неверно ряда систем, чья память устроена нестандартным образом. В этом смысле значение функции `MPI_Get_address()` состоит в обеспечении переносимости кода для таких систем.

3.6.4. Подтверждение и удаление новых типов

В процессе изготовления схем типов, некоторые из получаемых схем могут быть временными, созданные лишь с целью облегчения дальнейшего создания. Для MPI нет смысла учиться передавать объекты, соответствующие временным типам. Напротив, типы, которые планируется передавать, MPI может попытаться каким-либо оптимизировать для передачи.

По этой причине, перед передачей данных нового типа следует подтвердить его «полезность», после чего MPI примет меры для осуществления возможности его оптимальной передачи. Данное подтверждение производится функцией с прототипом следующего вида.

```
int MPI_Type_commit(
                   MPI_Datatype *   datatype      // тип данных            input
                   );
```

После вызова этой функции новый тип *datatype* может быть полноценно использован в программе. Когда созданный и подтверждённый тип отработал своё, его можно удалить командой

```
int MPI_Type_free(
                  MPI_Datatype *     datatype      // тип данных            input
                  );
```

В результате вызывается деструктор объекта *datatype*, а сама переменная-ссылка на него получает значение MPI_DATATYPE_NULL.

3.6.5. Передача массивов элементов производного типа

Рассмотрим теперь подробнее, что происходит при отправке от одного процесса другому сообщения, содержащего массив объектов определённого типа,. В частности, рассмотрим вызов функции MPI_Send(*buf*,*size*,*datatype*,*rank*,*tag*,*comm*), где третьим аргументом передаётся указатель на созданный тип. В MPI-реализации этот вызов эквивалентен последовательности трёх вызовов

```
MPI_Type_contiguous(size,datatype,newtype);
MPI_Type_commit(newtype);
MPI_Send(buf,1,newtype,rank,tag,comm);
```

Другими словами, сначала весь массив интерпретируется как один элемент сложного типа, получающегося *size* –кратным приписыванием элементов *datatype* друг за другом. Отправка происходит лишь после этого.

При этом значение *buf* интерпретируется как нулевое (не нижняя граница!) для *datatype*, то есть, для производного типа $S=\{(t_0, s_0),(t_1, s_1), \ldots, (t_k, s_k)\}$ будем иметь соответствие t_i по адресу *buf* $+s_i$. Подобным образом происходит передача всех сообщений, функция отправки которых содержит параметры *size* и *datatype*.

В данном разделе мы рассмотрели механизм производных типов, который позволяет осуществлять передачу данных с практически любой структурой. C++, как объектно-ориентированный язык, работает преимущественно со структурами и классами, а потому механизм производных типов крайне важен для полноценной работы с объектами. Их использование делает код понятнее, что часто позволяет избежать ошибок. С другой стороны, их использование может привести к замедлению работы программы, поэтому иногда бывает целесообразно воздержаться от их использования. Для дальнейшего ознакомления читатель может обратиться к спецификации MPI [11].

Завершим рассмотрение производных типов примерами, иллюстрирующими работу основных функций данного раздела. Рассмотрим пример 5, в котором для структуры Particle, состоящей из характеристик частицы: массы, координат и скорости, строится производный тип *Particletype*. После этого построенный производный тип используется для передачи массива структур от одного процесса другому.

Опишем кратко построенный алгоритм. Сначала определимся, что производный тип мы создадим при помощи единственного вызова функции MPI_Type_struct(). Вызов это, однако, следует тщательно подготовить. Структура Particle состоит из элемента типа **int**, трёх элементов типа **double** и трёх элементов типа **float** . В соответствии с этим, нас производный тип будет состоять из трёх блоков, первый – из одного элемента

типа **int**, второй – из трёх элементов типа **double**, и последний – из трёх элементов типа **float**. В соответствии с этой информацией, мы можем сразу заполнить три аргумента будущей функции `MPI_Type_struct()`. Так, мы определились, что число блоков будет равно трём, длины блоков равны 1, 3, 3, а их типы соответственно `MPI_INT`, `MPI_DOUBLE` и `MPI_FLOAT`. Эту информацию мы заносим в соответствующие аргументы практически в самом начале программы, в строках 26-27, поскольку для их определения никаких вычислений не требуется.

Осталось сделать последний, сравнительно сложный шаг, состоящий в определении того, как именно расположены в памяти элементы структуры `Particle`. Поскольку при объявлении массивов в C++ для них всегда выделяются последовательные участки памяти, то для того, чтобы знать всё об устройстве нашей структуры, необходимо лишь выяснить адреса переменной *m*, а также адреса первых элементов каждого из массивов *x* и *v*. Именно это и осуществляют строки 32-34, записывая полученные адреса в массив *disp*. Поскольку использование абсолютных адресов крайне неудобно, мы будем отсчитывать смещения не от начала адресного пространства `MPI_BOTTOM`, а от адреса одного из элементов структуры, например, от адреса *m*. Эта задумка осуществляется строками 36-39.

После произведённых операций у нас есть все необходимые данные для того, чтобы создать производный тип, соответствующий структуре `Particle`. Сделав это в 43 строке, нам следует сразу же верифицировать новый тип (строка 44), показав MPI, что мы планируем использовать его при передаче сообщений.

Оставшаяся часть кода не нуждается в подробном рассмотрении и содержит лишь передачу сообщения, представляющего собой массив из объектов нового типа. После передачи работа с MPI заканчивается, и управление возвращается операционной системе.

```
1   #include <stdio.h>
2   #include "mpi.h"
3
4   struct Particle
5   {
6     int    m;          // масса атома
7     double x[3];       // координаты
8     float  v[3];       // скорость
9   };
10
11
12  int main()
13  {
14    int MyRank, Size;
15    MPI_Init(NULL, NULL);
16    MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
17    MPI_Comm_size(MPI_COMM_WORLD, &Size);
18
19  /*********************************************/
20    Particle particle[1000];
21    int i, dest=1, rank=0, tag=99;
22
23  // Готовим производный тип для структуры Particle
24
25    MPI_Datatype Particletype;
26    MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_FLOAT}; // массив типов
27    int          blocklen[3] = {1,3,3};                      // размеры блоков
28    MPI_Aint     disp[3];                                   // массив смещений
```

```
29
30  // Заполняем массив смещений в соответствии с расположением данных в структуре
31
32    MPI_Address( particle,      disp[0]);
33    MPI_Address( particle[0].d, disp[1]);
34    MPI_Address( particle[0].b, disp[2]);
35
36    for (i=2; i>=0; --i)
37    {
38       disp[i] -= disp[0];
39    }
40
41  // Всё готово, создаём производный тип
42
43    MPI_Type_struct(3, blocklen, disp, type, &Particletype);
44    MPI_Type_commit(&Particletype);
45
46  // Посылаем производный тип от процесса rank процессу dest
47
48    if (MyRank == rank)
49    {
50      MPI_Send(particle,1000,Particletype,dest,tag,MPI_COMM_WORLD);
51    }
52
53    if (MyRank == dest)
54    {
55
56  MPI_Recv(particle,1000,Particletype,rank,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
57    }
58
59  /*************************************************/
60    MPI_Finalize();
61    return 0;
    }
```

Пример 5. Использование производных типов

Если эталонная структура, на основе который необходимо построить производный тип, имеет более сложный вид, и состоит из нескольких частей, состоящих в свою очередь из более простых структур, то можно пойти по другому пути и создавать производный тип последовательно. Так, можно сначала создать производный тип вектора, после чего для каждой из составных частей создать свой тип, и, наконец, создать тип для первоначальной структуры. Пусть, например, есть структура `Structure`, состоящая из трех векторов.

```
struct Structure
{
  Vector V1;
  Vector V2;
  Vector V3;
};
```

Тогда, чтобы построить производный тип для этой структуры, можно сначала написать функцию, которая будет строить производный тип для вектора. Потом, используя новый тип для вектора, можно построить производный тип для `Structure`.

`Build_derived_type_Vec(Vector* v, MPI_Datatype& Vector_type)` в строке 15 – функция, которая строит производный тип для вектора, написать ее можно аналогично тому, как описано в примере 5.

```
1  void Build_derived_type_Structure(Structure* X, MPI_Datatype& Structure_type)
2  {
3      int           block_length[3];         // размеры блоков
4      MPI_Aint      displacements[3];        // массив смещений
5      MPI_Datatype  typelist[3];             // массив типов
6      MPI_Datatype  vec;                     // производный тип для вектора
7      MPI_Aint      addresses[4];
8
9      int numb_elements = 3;
10
11     block_length[0] = 1;
12     block_length[1] = 1;
13     block_length[2] = 1;
14
15     Build_derived_type_Vec(&(X->V1),  vec );   //построить тип вектора
16     typelist[0] = vec;
17     typelist[1] = vec;
18     typelist[2] = vec;
19
20     MPI_Address(X, &addresses[0]);
21     MPI_Address(&(X->V1), &addresses[1]);
22     MPI_Address(&(X->V2), &addresses[2]);
23     MPI_Address(&(X->V3), &addresses[3]);
24
25     displacements[0] = addresses[1] - addresses[0];
26     displacements[1] = addresses[2] - addresses[0];
27     displacements[2] = addresses[3] - addresses[0];
28
29     //построить тип структуры
30 MPI_Type_struct(numb_elements,block_length,displacements,typelist,
   &Structure_type);
31 MPI_Type_commit(&Structure_type);
32 }
```

3.7. Коммуникаторы

Мы завершим наше введение в MPI последним, но крайне важным, понятием коммуникатора.

При рассмотрении коллективных обменов данных мы использовали всеобщий коммуникатор `MPI_COMM_WORLD`. В результате, все групповые коммуникации затрагивали все узлы. Представим теперь, что наше MPI-приложение *гетерогенно*, то есть состоит из нескольких принципиально разных частей. При этом каждая часть выполняется определённым количеством процессов, и обмены данными между частями необходимы достаточно редко по сравнению с обменами внутри групп, работающих над одной и той же частью задачи. Чтобы принять во внимание подобное перераспределение работы процессов, в MPI используется механизм коммуникаторов. Коммуникатору соответствует некоторое подмножество процессов, которое называется *группой*, однако понятие группы шире понятия коммуникатора: одной группе может соответствовать несколько коммуникаторов. Если коммуникатор передаётся в качестве аргумента в какую-либо MPI-функцию, то это означает, что все операции должны производиться в рамках группы процессов данного коммуникатора. С группами можно работать независимо от коммуникаторов, но, поскольку все функции передачи сообщений требуют именно коммуникатор в качестве аргумента, созданные группы всё равно необходимо привязывать к коммуникаторам. Кроме того, коммуникаторы делятся на две категории: внутренние и

внешние. Внутренние коммуникаторы используются для обмена данными внутри группы, в то время как обмен между группами требует внешнего коммуникатора.

ВНУТРЕННИЕ КОММУНИКАТОРЫ

(*внутригрупповые* коммуникаторы или *интракоммуникаторы*)

Этот тип коммуникаторов наиболее часто используется при передачах данных в MPI. Он употребляется в тех ситуациях, когда необходимо выделить подмножество всех процессов и обеспечить обмен сообщениями таким образом, чтобы передача сообщений касалась только процессов этого подмножества. Внутренний коммуникатор содержит в себе *группу*, *контекст* и прочие *атрибуты*. Атрибуты определяют информацию, которую пользователь или библиотека добавили к коммуникатору для обращения в дальнейшем. Для создания, модификации и удаления атрибутов в MPI предусмотрены специальные средства, подробнее узнать о них можно, например в [12].

Группа представляет собой совокупность процессов, которые выделены для общения между собой. По сути, это просто подмножество всех процессов.

Контекст является независимым пространством для обмена MPI-сообщениями. Он служит дополнительной меткой, которая отличает все сообщения в рамках данного коммуникатора.

Подобный механизм является решающим в вопросах, связанных с написанием MPI-библиотек, поскольку вызовы внешних функций не должны никак взаимодействовать с коммуникацией в исходном контексте программы. В некотором смысле, подобное решение можно сравнить с использованием пространств имён в C++.

Как было отмечено ранее, запуск функции `MPI_Init` инициализирует коммуникатор `MPI_COMM_WORLD`. Кроме того, по умолчанию для каждого процесса определяется коммуникатор `MPI_COMM_SELF`, состоящий лишь из одного этого процесса, и константа `MPI_COMM_NULL`, значение которой передаётся переменной коммуникатора в случае, если соответствующий ей объект был удалён или ещё не был создан.

ВНЕШНИЕ КОММУНИКАТОРЫ

(*межгрупповые* коммуникаторы или *интеркоммуникаторы*)

Данный тип коммуникаторов предназначен для передачи сообщений между двумя непересекающимися группами. Как было упомянуто выше, в случае, если задача разбивается на несколько частей, каждая из которых выполняется отдельной группой процессов, бывает удобно использовать данный тип коммуникаторов для обмена сообщениями между такими группами. Но если каждая из групп задаётся внутренним коммуникатором, внешний должен иметь возможность каким-то образом связывать эти две группы для общения.

Как и в предыдущем случае, *контекст* представляет собой независимое пространство для обмена сообщениями. Однако в случае внешних коммуникаторов, контекст определяет обмен сообщениями между двумя непересекающимися группами. Более того, отправители сообщений и их получатели всегда находятся в разных группах. Из вышесказанного следует, что для работы данному типу коммуникаторов необходимо иметь две *группы*.

Коммуникатор изолирует выбранную группу процессов и помогает избежать ошибок при активном обмене MPI-сообщениями в рамках различных частей задачи. Поскольку внешние коммуникаторы являются более сложными в использовании, их стоит применять лишь в тех случаях, когда это позволять значительно облегчить реализацию. Мы не будем останавливаться на них более подробно. Эту информацию можно найти в спецификации, [11].

Отметим ещё раз тот факт, что во всех функциях передачи сообщений, где указывается ранг, указывается также и коммуникатор. Это происходит потому, что ранг процес-

са не определён сам по себе (то есть, у процесса нет ранга), а лишь имеет смысл в рамках рассматриваемого коммуникатора.

3.7.1. Контекст

Контекст – свойство коммуникаторов, которое позволяет разделять пространство обмена сообщениями. В то время как понятие группы позволяет выделить некоторый класс процессов и предоставить удобный механизм обращения к ним, контекст предоставляет механизм для выделения сообщений. Так, сообщения одного контекста не могут быть перепутаны с сообщениями другого. Сообщения, отправленные в рамках одного контекста, могут быть получены лишь элементами этого же контекста и будут проигнорированы всеми остальными контекстами. Как уже было замечено выше, контекст ставит свою индивидуальную метку на каждое отправляемое сообщение. В отличие от групп, контексты создаются автоматически вместе с коммуникатором и не являются отдельными объектами.

3.7.2. Группы

Группы являются отдельными объектами типа `MPI_Group`, содержащими подмножество процессов. При этом процессы в группе упорядочены и пронумерованы последовательными числами от 0 до уменьшенного не единицу размера группы, которые ранее мы назвали *рангами* процессов.

Обратим внимание на предопределённые объект `MPI_GROUP_EMPTY` и константу `MPI_GROUP_NULL`. Первый соответствует ссылке на объект пустой группы, в то время как вторая константа соответствует ссылке на несуществующий объект (например, удалённый за ненадобностью).

Пусть имеется некоторая группа, про которую нам пока ничего не известно. В этом случае можно получить информацию о размере группы и о том, каков ранг данного процесса в этой группе с помощью приведённых ниже функций `MPI_Group_size` и `MPI_Group_rank`. Схема их работы весьма прозрачна: получая на вход ссылку на группу, они возвращают, соответственно, её размер и ранг данного процесса в ней.

```
int MPI_Group_size(
                  MPI_Group   group,    // группа                          input
                  int*        size      // размер группы                  output
                  );

int MPI_Group_rank(
                  MPI_Group   group,    // группа                          input
                  int*        rank      // ранг процесса в группе         output
                  );
```

Что касается построения новых групп, их можно конструировать лишь из уже имеющихся. В этом смысле, исходным материалом таких построений являются коммуникаторы `MPI_COMM_SELF` и `MPI_COMM_WORLD`. Чтобы извлечь из коммуникатора соответствующую группу, используют функцию

```
int MPI_Comm_group(
                  MPI_Comm    comm,      // коммуникатор                    input
                  MPI_Group*  group      // ссылка на объект группы        output
                  );
```

Далее, если нужно сконструировать собственную группу, необходимо уметь выполнять простейшие операции объединения, пересечения и исключения, которые задаются функциями

```
int MPI_Group_union(
                  MPI_Group   group1,    // первая группа                   input
                  MPI_Group   group2,    // вторая группа                   input
                  MPI_Group*  group      // ссылка на группу-результат     output
                  );

int MPI_Group_intersection(
                  MPI_Group   group1,    // первая группа                   input
                  MPI_Group   group2,    // вторая группа                   input
                  MPI_Group*  group      // ссылка на группу-результат     output
                  );

int MPI_Group_difference(
                  MPI_Group   group1,    // первая группа                   input
                  MPI_Group   group2,    // вторая группа                   input
                  MPI_Group*  group      // ссылка на группу-результат     output
                  );
```

Первая функция возвращает группу, полученную объединением двух групп, причём сначала идут процессы первой группы, а затем – процессы второй.

Вторая функция возвращает группу, состоящую лишь из тех элементов, которые присутствуют в обеих группах-аргументах. При этом их порядок определяется в соответствии с их порядком в первой группе.

Третья из указанных функций возвращает группу, составленную из элементов первой группы, не входящих во вторую. Порядок их следования такой же, как в первой группе.

Если в результате перечисленных операций получается пустая группа, результат будет иметь значение `MPI_GROUP_EMPTY`. Более важные функции, которые позволяют выделить подгруппу группы явно по их рангам, выглядят следующим образом

```
int MPI_Group_incl(
                  MPI_Group  oldgroup,   // исходная группа                 input
                  int        n,          // число элементов в новой группе  input
                  int*       ranks,      // массив рангов элементов         input
                  MPI_Group* group       // новая группа                   output
                  );

int MPI_Group_excl(
                  MPI_Group  oldgroup,   // исходная группа                 input
                  int        n,          // число элементов в новой группе  input
                  int*       ranks,      // массив рангов элементов         input
                  MPI_Group* group       // новая группа                   output
                  );
```

Обе функции принимают на вход исходную группу, массив рангов элементов и длину этого массива. Первая функция возвращает в *group* ссылку на новую группу, состоящую из элементов группы *oldgroup* с рангами *ranks*[0], ..., *ranks*[n-1]. Вторая функция, напротив, конструирует группу из всех элементов исходной группы кроме элементов *ranks*[0], ..., *ranks*[n-1].

С помощью указанных функций может быть реализована группа из любого набора процессов. Существуют и другие функции для работы с группами, которые могут оказаться полезными в ряде случаев [11].

Наконец, чтобы удалить группу, которая больше не нужна, вызывается функция

```
int MPI_Group_free(
                  MPI_Group*  group      // ссылка на удаляемую группу      input
                  );
```

3.7.3. Работа с коммуникаторами

Теперь, когда мы определись с тем, как создавать группы, и зачем нужен контекст, можно начать обсуждение коммуникаторов. Как и в случае групп, имея существующий коммуникатор, можно узнать размер и ранг процесса в рамках этого коммуникатора. Эту задачу решают функции, совершенно аналогичные `MPI_Group_size` и `MPI_Group_rank` с точностью до типа первого аргумента.

```
int MPI_Comm_size(
                  MPI_Comm    comm,     // коммуникатор                    input
                  int*        size      // размер коммуникатора           output
                  );

int MPI_Comm_rank(
                  MPI_Group   group,    // коммуникатор                    input
                  int*        rank      // ранг процесса в коммуникаторе  output
                  );
```

Именно эти две функции мы последовательно вызывали во всем приведённых примерах. По сути, именно после вызова второй из них процессы становятся различимыми.

Ситуация с созданием собственных коммуникаторов обстоит следующим образом. Сам по себе коммуникатор является достаточно сложным объектом. По этой причине, создание нового коммуникатора «с нуля» потребовало бы хорошего знания MPI и усложнения последнего. На практике же в основном используется достаточно узкий круг коммуникаторов, поэтому разработчиками спецификации MPI было принято решение о том, что создавать новый коммуникатор следует на основе уже имеющегося, каковым является `MPI_COMM_WORLD`.

Рассмотрим основные функции, которые используются для создания коммуникаторов. Крайне важным является то, что все эти функции должны вызываться всеми процессами исходного коммуникатора *comm*. Первая из них просто копирует уже имеющийся коммуникатор.

```
int MPI_Comm_dup(
                  MPI_Comm      comm,  // коммуникатор                    input
                  MPI_Comm*  newcomm   // ссылка на новый
                                       коммуникатор-копию             output
                  );
```

При этом созданный коммуникатор является новым в том смысле, что передача сообщений старого коммуникатора игнорируется новым и наоборот.

Вторая функция

```
int MPI_Comm_create(
                  MPI_Comm       comm,      // коммуникатор                    input
                  MPI_Group     group,      // группа                          input
                  MPI_Comm*   newcomm       // ссылка на новый
                                               коммуникатор-копию            output
                  );
```

требует, чтобы группа *group* была подмножеством группы коммуникатора *comm*. Если исходный коммуникатор был внутренним, данная функция создаст новый коммуникатор с группой *group*. Однако это случится лишь в том случае, если все элементы *comm* произведут вызов этой функции с идентичным аргументом *group*, причём идентичность означает совпадения множества пар ранг-процесс. В случае если какое-то из условий не выполняется, *newcomm* будет содержать значение MPI_COMM_NULL.

Следующая крайне полезная функция MPI_Comm_split позволяет разбить один коммуникатор на несколько произвольным образом. Как и все остальные функции создания коммуникаторов, эта функция должна вызваться всеми функциями исходного коммуникатора. Однако на выходе разные процессы могут иметь разные значения коммуникатора.

```
int MPI_Comm_split(
                  MPI_Comm       comm,    // исходный коммуникатор          input
                  int           color,    // цвет  >=0                      input
                  int             key,    // параметр для нового ранга      input
                  MPI_Comm*   newcomm     // ссылка на новый коммуникатор  output
                  );
```

Особенность данного вызова состоит в том, что разные процессы передают собственные значения параметров цвета и ключа. В результате, все процессы одного цвета получают на выходе коммуникатор *newcomm*, состоящий ровно из них, а процессы разного цвета окажутся в разных коммуникаторах. Так происходит выбор разбиение всех процессов на группы. Чтобы группы стали полноценными коммуникаторами, нам осталось указать ранг каждого процесса в новой группе с помощью третьего аргумента. Процессы в каждой новой образовавшейся группе одного цвета упорядочиваются по возрастанию параметра *key*, после чего их ранги определяются, исходя из этого порядка. Так, процесс с наименьшим значением приобретает ранг 0, следующий 1 и так далее. Отметим, что значение цвета должно быть неотрицательным, а значения ключей могут повторяться. В последнем случае процессы с совпадающими рангами будут упорядочены так же, как они были упорядочены в исходном коммуникаторе. Таким образом, данная функция даёт нам полную свободу действий: по цветовому признаку происходит разбиение исходного коммуникатора на произвольные группы с произвольными рангами. Наконец, чтобы освободить память, занимаемую ненужным более коммуникатором, используется функция

```
int MPI_Comm_free(
                  MPI_Comm*   comm      // ссылка на удаляемый коммуникатор   input
                  );
```

Механизм коммуникаторов может служить для создания иерархичности и гетерогенности параллельной реализации программы и является крайне полезным в некото-

рых случаях, например, при работе на гибридных вычислительных системах CPU/GPU, когда графические процессоры есть лишь на некоторых узлах. В этом случае сама по себе вычислительная система гетерогенна, и коммуникаторы могут помочь использовать её наиболее эффективно.

4. Примеры

Рассмотрим теперь, как можно организовывать вычисления параллельно в применении к задачам.

Общие принципы разработки параллельно работающих алгоритмов.

1. Выделение блоков, которые вычисляются в значительной степени независимо друг от друга.
2. Определение зависимостей, то есть получение представления о том, какими данными предстоит обмениваться процессам в ходе решения задачи.
3. Получение соответствия количества процессов и количества частей задачи, выполняющихся параллельно. В идеальном случае все процессы должны быть равномерно нагружены для получения максимальной отдачи.

Программа будет более гибкой, если в ней будет предусмотрено различное количество процессов для расчета, это нужно предусмотреть при разделении работы между процессами.

В качестве задачи, которая может проиллюстрировать различные алгоритмы распараллеливания можно рассмотреть задачу умножения матрицы на вектор.

4.1. Умножение матрицы на вектор

Рассмотрим квадратную матрицу размера $n \times n$, которую нужно умножить на вектор длины n. Посмотрим, какие блоки можно выделить в алгоритме умножения этих объектов так, чтобы их выполнение было бы по большей части независимо.

Введем следующие обозначения:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}; b = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}; \begin{pmatrix} a_{1i} \\ a_{2i} \\ \dots \\ a_{ni} \end{pmatrix} = \hat{A}_i; \begin{pmatrix} a_{j1} & a_{j2} & \dots & a_{jn} \end{pmatrix} = \overline{A}_j$$

Тогда можно вычислить k-ю координату получающегося в результате умножения вектора по следующим формулам:

$$(Ab)_k = \sum_{i=1}^{n} a_{ki} b_i = \hat{A}_k b = \sum_{i=1}^{n} (\hat{A}_i)_k \; b_k \qquad (1)$$

Исходя из формул (1), можно предложить три способа распараллеливания задачи умножения матрицы на вектор.

СПОСОБ 1

Результирующий вектор будет состоять из координат, полученных как скалярные произведения строк матрицы на вектор. В этом случае для получения каждой из координат результата требуется ровно одна строка из матрицы и вектор, на который

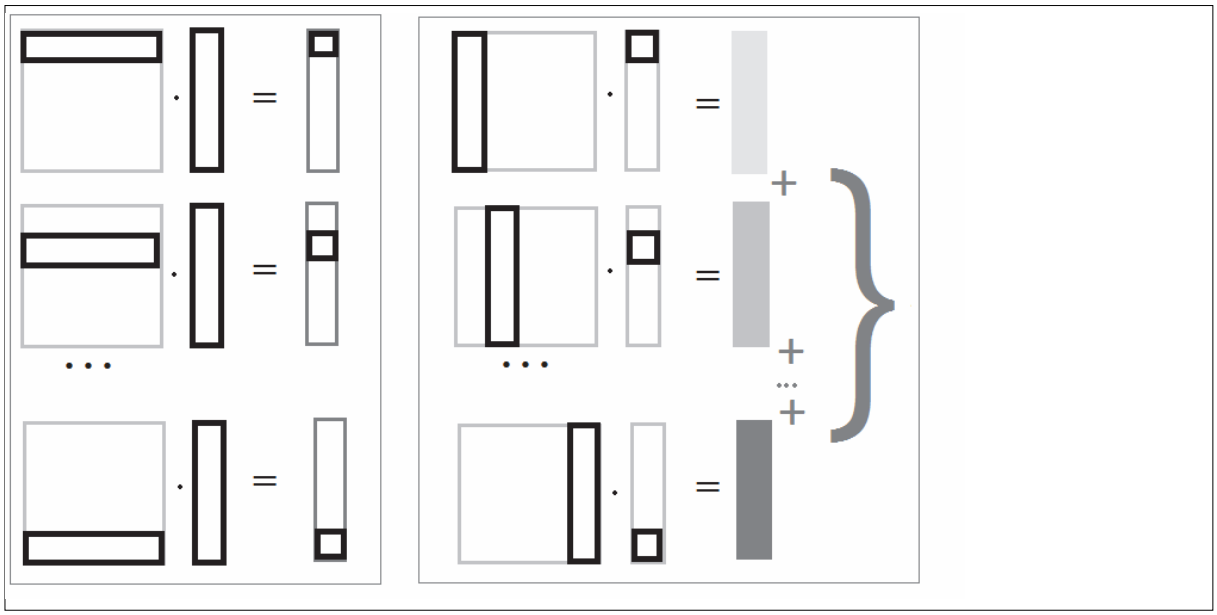

Рис. 6. Варианты распараллеливания задачи умножения матрицы на вектор.
Слева – способ распараллеливания по строкам; справа - способ распараллеливания по столбцам.

производится умножение. Именно таким образом в данном способе производится выделение независимых частей. Теперь нужно установить, какие данные будут нужны каждому процессу для такой организации вычислений. Понятно, что если распределять вычисления на n процессов, то каждому процессу для вычислений будет нужна соответствующая строка и вектор-столбец (см. рис. 6).

Также стоит отметить, что способ распараллеливания по строкам предполагает и обобщение на случай, если участвующих процессов $p < n$. В этом случае разделим количество строк на p. Если $n = pk$, где k – некоторое целое число, то каждый процесс может рассчитывать произведение k строк матрицы на вектор и работа будет распределена равномерно. После того, как все процессы вычислили свою часть работы, нужно организовать сбор данных, поскольку у каждого процесса находится только часть ответа. Это можно сделать при помощи функции `MPI_Gather()`.

Следует отметить, что размер матрицы может оказаться не кратным количеству процессов, и тогда строки матрицы не могут быть разделены поровну между процессами. В этих ситуациях можно отступить от требования равномерности загрузки процессов и для получения более простой вычислительной схемы принять правило, что размещение данных на процессах осуществляется только построчно (т.е. элементы одной строки матрицы не могут быть разделены между несколькими процессами). Неодинаковое количество строк приводит к разной вычислительной нагрузке процессов, и, тем самым, завершение вычислений (общая длительность решения задачи) будет определяться временем работы наиболее загруженного процесса. При этом процессы, выполнившие свою часть вычислений, будут простаивать, ожидая своих более загруженных коллег.

СПОСОБ 2

Идея второго способа реализации заключается в следующем. Если умножать столбцы матрицы на соответствующие координаты вектора, а потом сложить результаты, то получится нужный результирующий вектор. Таким образом, выделяются следующие части задачи, которые могут выполняться независимо: умножение столбца

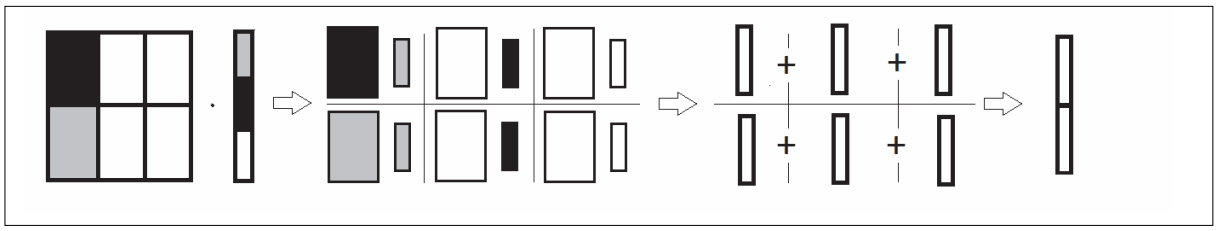

Рис. 7. Блочное распараллеливание алгоритма умножения матрицы на вектор. Разделение на блоки, распределение работы по процессам, вычисление результата.

матрицы на соответствующую координату вектора. Главное отличие в том, что для проведения этой операции не нужно передавать целый вектор каждому процессу, в случае распараллеливания на n процессов достаточно передать i-му процессу i-ю координату вектора и i-й столбец матрицы (см. рис. 6). Аналогично первому способу можно предусмотреть вариант распараллеливания, когда $n=pk$, где p – число процессов, k – натуральное число. В этом случае можно обеспечить наилучшую загрузку процессов.

СПОСОБ 3

Последний способ реализации, который будет нами рассмотрен, заключается в комбинировании первого и второго методов. Матрицу можно разделить на более мелкие прямоугольные блоки, вектор разделить на компоненты, соответствующие размерам блоков (см. рис. 7). Тогда каждый процесс будет вычислять произведение блока на компоненту вектора. В результате нужно будет просуммировать результаты по каждой компоненте и собрать из них результирующий вектор. Более подробно, пусть матрица представляется в следующем виде

$$A=\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1p} \\ A_{21} & a_{22} & \dots & a_{2p} \\ \dots & \dots & & \dots \\ A_{s1} & A_{s2} & \dots & A_{sp} \end{pmatrix},$$ где каждый блок $$A_{ij}=\begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \dots & a_{i_0 j_{l-1}} \\ a_{i_1 j_0} & a_{i_1 j_1} & \dots & a_{i_1 j_{l-1}} \\ \dots & \dots & & \dots \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & \dots & a_{i_{k-1} j_{l-1}} \end{pmatrix}$$ имеет размер $k \times l$.

Причем, считаем, что n делится нацело на p и s.

$$i_v = ik + v, 0 \le v \le k, k = \frac{n}{s},$$

$$j_u = jl + u, 0 \le u \le l, l = \frac{n}{q}.$$

Вектор при этом разделяется на части следующим образом: $b = \begin{pmatrix} B_1 & B_2 & \dots & B_p \end{pmatrix}$, где $B_i = \begin{pmatrix} b_{i_0} & b_{i_1} & \dots & b_{i_{l-1}} \end{pmatrix}$. При вычислении каждый процесс будет производить умножение прямоугольной матрицы размера $k \times l$ на вектор длины l.

Блочная реализация является обобщением двух предыдущих алгоритмов: при $p = 1$ данные разбиваются на блоки, состоящие из строк, при $s = 1$ — из столбцов. Таким образом, максимально необходимое количество процессов определяется величиной n^2. Если такое количество процессов недоступно, что особенно актуально в случае больших матриц, нужно подбирать такие параметры p и s, чтобы все блоки были одного

размера для равномерной загрузки вычислительных узлов. В книге [6] приводятся интересные исследования ускорения в зависимости от числа узлов и размера матрицы.

Неравномерность загрузки процессов снижает эффективность использования, и *проблема балансировки* относится к числу важнейших проблем параллельного программирования. Поэтому при написании алгоритма следует учитывать загрузку процессов и подбирать число узлов так, чтобы загрузка была равномерной.

Реализация

Рассмотрим одну из возможных реализаций этой задачи, для начала самый простой случай — реализацию распараллеливания по строкам. Функция **int** `main()` будет иметь следующий вид:

```
1   int main()
2   {
3     double* Matrix;
4     double* Vector;
5     double* Result;
6     int Size;                    // выделение памяти
7     double* ProcRows;
8     double* ProcResult;
9     int RowNum;
10
11    MPI_Init(NULL, NULL);
12    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
13    MPI_Comm_rank(MPI_COMM_WORLD, &Rank);
14
15    if (Rank == 0) {
16      printf("Parallel matrix-vector multiplication program\n");
17    }
18    Initialization(Matrix, Vector, Result, ProcRows, ProcResult, Size, RowNum);
19    DataDistribution(Matrix, ProcRows, Vector, Size, RowNum);
20    Calculation(ProcRows, Vector, ProcResult, Size, RowNum);
21    ResultReplication(ProcResult, Result, Size, RowNum);
22    TestOutput(Matrix, Vector, ProcRows, Size, RowNum, Result);
23    if (Rank == 0) {
24      delete[] Matrix; }
25    delete[] Vector;
26    delete[] Result;             //удаление выделенной памяти
27    delete[] ProcRows;
28    delete[] ProcResult;
29    MPI_Finalize();
30    return 0;
31   }
```

Основные вычисления осуществляются в строках 18 – 21.

Функция `Initialization` в строке 18 — инициализация основных величин и структур данных. Она может быть реализована множеством различных способов. В случае, когда нам просто необходима некая иллюстрация алгоритма, можно использовать случайное заполнение матрицы и вектора. В более прикладных задачах размер и компоненты матрицы и вектора можно считывать из файла. Эти процедуры обычно выполняются одним процессом, например, root-процессом.

Следует обратит внимание, что память в функции **int** `main()` выделяется только на процессе `root`, и это происходит до инициализации MPI. Матрица, таким образом, будет храниться только на `root`, а остальным узлам будут передаваться только нужные для вычисления строки (в зависимости от общего числа процессов) и сам вектор. Поэтому, при запуске инициализации необходимо выделить память для приема вектора, для приема строк матрицы, с которыми будет проводиться работа, а так же для

результата, который будет получать процесс. Поскольку число процессов, которые будут использоваться в вычислениях, задается пользователем и заранее неизвестно, память следует выделять динамически.

Остановимся более подробно на функции передачи данных `DataDistribution`, вызываемой в строке 19. При исследовании механизма взаимодействия между процессами было указано, что обмен данными будет строиться по следующей схеме: `root` должен разослать всем процессам вектор и часть строк матрицы для вычисления. Для этого можно использовать две функции MPI: `MPI_Bcast()` для передачи вектора и `MPI_Scatterv()` для рассылки каждому вычислительному узлу тех строк матрицы, которые ему положены. Для того, чтобы корректно определить, какие строки нужно переслать, функции **void** `MPI_Scatterv()` нужно будет получить в качестве аргументов массив данных, указатель на место, с которого нужно прочитать данные, размер массива данных и их тип. В качестве выходных параметров нужен буфер для записи этих данных, размер и тип данных. Так же в аргументах необходимо указать номер отправителя (обычно 0), и коммуникатор, в этом случае `MPI_COMM_WORLD`.

```
1   void DataDistribution(double* Matrix, double* ProcRows, double* Vector,
2                         int Size, int RowNum) {
3     int* SendNum;         // число элементов, посылаемых процессу
4     int* SendInd;         // индекс первого элемента
5     int  RestRows=Size;   // количество не распределенных строк
6
7
8     MPI_Bcast(Vector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
9
10    SendInd = new int [ProcNum]; // выделение памяти для вспомогательных массивов
11    SendNum = new int [ProcNum];
12
13    // для каждого процесса определяются индекс первого элемента строк к пересылке
14    // и число посылаемых элементов (на случай некратного числа процессов)
15    RowNum = (Size/ProcNum);
16    SendNum[0] = RowNum*Size;
17    SendInd[0] = 0;
18    for (int i=1; i<ProcNum; i++) {
19      RestRows -= RowNum;
20      RowNum = RestRows/(ProcNum-i);
21      SendNum[i] = RowNum*Size;
22      SendInd[i] = SendInd[i-1]+SendNum[i-1];
23    }
24
25    MPI_Scatterv(Matrix , SendNum, SendInd, MPI_DOUBLE, ProcRows,
26      pSendNum[Rank], MPI_DOUBLE, 0, MPI_COMM_WORLD); //отправить-принять строки
27
28    delete[] SendNum;     // удаление временных массивов
29    delete[] SendInd;
30  }
```

Функция `Calculation(`*`ProcRows`*`, `*`Vector`*`, `*`ProcResult`*`, `*`Size`*`, `*`RowNum`*`)` вычисляет вектор длины *`ProcResult`*, который получается умножением сток матрицы *`ProcRows`* на *`Vector`*. В случае числа процессов, равного числу строк в матрице, *`ProcResult`*`=1`. После выполнения функции `Calculation()` нужно собрать из частей *`ProcResult`*, вычисленных на узлах, вектор *`Result`*.

Следующая вслед за `Calculation()` функция сбора результата `ResultReplication(ProcResult, Result, Size, RowNum)` организована аналогично функции `DataDistribution()`. Сначала следует определить размеры векторов, которые будут возвращаться, потом собрать их. Для ясности код функции приведен целиком.

```
1   void ResultReplication(double* ProcResult, double* Result, int Size,
    int RowNum)
2   {
3     int  i;
4     int* pReceiveNum;  // число элементов, посылаемых процессом
5     int* pReceiveInd;  // индекс элемента в векторе-результате
6     int  RestRows=Size; // количество не распределенных строк
7
8     pReceiveNum = new int [ProcNum]; //выделение памяти для временных величин
9     pReceiveInd = new int [ProcNum];
10
11    pReceiveInd[0] = 0;
12    pReceiveNum[0] = Size/ProcNum;
13    for (i=1; i<ProcNum; i++) {
14      RestRows -= pReceiveNum[i-1];
15      pReceiveNum[i] = RestRows/(ProcNum-i);
16      pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
17    }
18
19    MPI_Allgatherv(ProcResult, pReceiveNum[Rank], MPI_DOUBLE, Result,
20      pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD); // сбор результата
21
22    delete[] pReceiveNum; //удаление временных массивов
23    delete[] pReceiveInd;
24  }
```

Сбор данных производится при помощи функции `MPI_Allgatherv`, которая осуществляет передачу данных от каждого процесса всем (см. раздел о коллективных передачах данных). Таким образом, вектор `Result` собирается на каждом из процессов.

Для отслеживания правильности работы программы вызывается функция `TestOutput(Matrix, Vector, ProcRows, Size, RowNum, Result)`, которая будет выводить на экран или в файл начальные матрицу и вектор, работающие в программе,а также вектор-результат. При помощи этой функции можно проверить корректность работы программы и правильность вычисления.

Вопрос отладки параллельных программ стоит особенно остро. Основное отличие от последовательной организации в том, что при расчетах происходит передача и прием данных. Нужно внимательно следить, какие данные направляются на вход и какие приходят к процессу, поскольку обмены могут быть достаточно сложны. Для этого на этапе отладки можно писать функции тестового вывода для каждого из процессов. Особенно внимательно нужно следить за размерами буферов приема-передачи данных и не забывать вовремя удалять не нужную более динамическую память.

Организовать программу для реализации второго способа распараллеливания по столбцам можно по той же схеме. Отметим, однако, некоторые тонкости реализации в этом случае. На процессы будут передаваться столбцы, а информация о них в памяти не будет записана непрерывно. Можно решить эту проблему, организовав, например, производный тип данных — «столбец метрицы». Другим вариантом является применение некоторой хитрости на этапе инициализации: записывать столбцы матрицы по строкам. Тогда они будут лежать в памяти непрерывно и функцию раздачи данных `DataDistribution` можно будет поменять лишь незначительно. Другое, более существенное отличие будет в том, что `MPI_Bcast()` нужно заменить на `MPI_Scatterv()`, потому что каждый процесс будет иметь дело со своей частью вектора. Функция `MPI_Scatterv()`, распределяющая вектор (и все параметры для нее) будет записываться аналогично функции, распределяющей столбцы.

Для сбора данных можно использовать `MPI_Reduce()` с операцией `MPI_SUM`. Операция в функции `MPI_Reduce` к массивам применяется покомпонентно, поэтому в итоге вектор результата будет получен на процессе *root*.

Способ реализации по столбцам более выгоден с точки зрения выделения памяти на узлах. За счет того, что каждому процессу передается только часть вектора, а не весь, достигается экономия памяти.

Задача умножения матрицы на вектор часто рассматривается в литературе как модельный пример, дополнительно можно изучить руководства [6], [8], [13], [14], [15] где можно найти разбор и других примеров из линейной алгебры.

4.2. Решение систем ОДУ

Множество задач по моделированию систем сводится к решению систем линейных дифференциальных уравнений или уравнений в частных производных. Явное решение этих уравнений зачастую оказывается возможным только в простейших случаях; в случаях более сложных исследователю приходится использовать численные методы. Когда в задаче много параметров, численное решение уравнений является самой трудоемкой в вычислительном плане частью алгоритма. Применяя параллельные методы, можно добиться значительного ускорения при решении такого рода задач.

Для сравнения приведем последовательную и параллельную реализацию одного из алгоритмов численного решения систем обыкновенных дифференциальных уравнений (ОДУ). О существующих методах численного решения систем ОДУ можно узнать подробнее в [16, 17, 33].

Рассмотрим самый простой метод решения систем ОДУ – метод Эйлера. Пусть требуется найти решение задачи Коши для системы обыкновенных дифференциальных уравнений первого порядка следующего вида:

$$\begin{aligned} &u_1' = f_1(x,u_1,u_2,...u_n), u_1(x_0) = u_1^0 \\ &u_2' = f_2(x,u_1,u_2,...u_n), u_2(x_0) = u_2^0 \\ &... \\ &u_n' = f_n(x,u_1,u_2,...u_n), u_n(x_0) = u_n^0 \end{aligned} \qquad (2)$$

Введем векторные обозначения

$$U = \begin{pmatrix} u_1 \\ u_2 \\ ... \\ u_n \end{pmatrix}, \quad U' = \begin{pmatrix} u_1' \\ u_2' \\ ... \\ u_n' \end{pmatrix}, \quad F(x,u) = \begin{pmatrix} f_1(x,u_1,u_2,...u_n) \\ f_2(x,u_1,u_2,...u_n) \\ ... \\ f_n(x,u_1,u_2,...u_n) \end{pmatrix}, \quad U^0 = \begin{pmatrix} u_1^0 \\ u_2^0 \\ ... \\ u_n^0 \end{pmatrix}$$

В этих обозначениях задача Коши примет вид:

$$U = F(x,U), U(x_0) = U^0$$

Кратко напомним метод Эйлера. Предположим, что известно значение v_i в точке x_i (в частности, в начале расчетов известно значение в точке x_0). Разложим решение в окрестности точки x_i по формуле Тейлора:

$$u(x_i + h) = u(x_i) + u'(x_i)h + ... + \frac{u^{(k)}(x_i)}{k!}h^k + o(h^k)$$

Оставим в этой формуле слагаемые до второго порядка по h и, учитывая, что, из уравнений (2), $u'(x_i) = f(x_i, u_i)$, получим формулу численной схемы Эйлера:

$$v_{i+1} = v_i + hf(x_i, v_i)$$

Здесь считаем, что v_{i+1} - значение приближенного решения v в точке x_{i+1}. Поскольку разложение в формуле Тейлора проводилось до членов второго порядка, эта формула на каждом шаге интегрирования будет давать ошибку порядка h^2. Существует большое количество методов более высокого порядка точности, один из них будет приведен при разборе задач молекулярной динамики в разделе.

Для системы уравнений (2) расчётные формулы метода Эйлера будут выглядеть следующим образом:

$V^{i+1} = V^i + hF(x_i, V^i)$,

где V^{i+1} – приближенное значение вектора точных решений U на $i+1$ шаге метода.

Отметим, что переход от одной итерации к другой может осуществляться только последовательно, потому что для выполнения следующего шага алгоритма необходимы результаты выполнения алгоритма на предыдущем шаге. Поэтому распараллеливание может производиться только внутри одного шага алгоритма и будет эффективно в случае вычислительно-трудоемкой функции $F(x,u)$. В случае простой функции накладные расходы на пересылку данных на каждом шаге могут существенно понижать эффективность распараллеливания.

Рассмотрим вариант параллельной реализации задачи на p процессах. Можно разделить все уравнения системы на блоки размера n/p, и каждый процесс будет проводить вычисления только тех компонент вектора решения, которые относятся к его блоку. Рассмотрим описанную схему более подробно.

1. Уравнения системы распределяются по p процессам;
2. Параллельно вычисляются значения соответствующих функций f_i и компоненты вектора V^1
   на процессе p_1 вычисляются значения $f_1(x_0, U^0), ..., f_p(x_0, U^0)$ и компоненты $v_1^1, ... v_p^1$;
   на процессе p_2 вычисляются значения $f_{p+1}(x_0, U^0), ..., f_{2p}(x_0, U^0)$ и компоненты $v_{p+1}^1, ... v_{2p}^1$;
   …
   на процессе p_N вычисляются значения $f_{(N-1)p+1}(x_0, U^0), ..., f_{Np}(x_0, U^0)$ и компоненты $v_{(N-1)p+1}^1, ... v_{Np}^1$;
3. Каждый из процессов посылает каждому из остальных процессов вычисленные компоненты вектора V^1;
4. Аналогично каждый процесс вычисляет свою часть $f_i(x_1, V^1)$ и компонент V^2, действуя по схеме п.2-3;
5. По той же схеме вычисляются остальные компоненты.

В этой задаче становится особенно важной проблема равномерной загрузки процессов, поскольку к следующему шагу можно приступать только после завершения предыдущего всеми вычислительными узлами. Так как функции f_i могут иметь различную вычислительную сложность, способ распараллеливания, описанный выше, может привести к плохой балансировке процессов, то есть части задачи могут оказаться неэквива-

лентными по вычислительной сложности, и процессы, которым достались более короткие вычислительные куски, будут простаивать в ожидании данных от других процессов.

В силу невысокой точности в вычислительной практике метод Эйлера используется редко. Однако рассмотренная схема распараллеливания с небольшими изменениями переносится и на другие методы интегрирования систем ОДУ более высокого порядка.

Стоит отметить, что, в зависимости от задачи, можно разрабатывать и применять различные способы распараллеливания. Например, если система дифференциальных уравнения разбивается на две невзаимосвязанные между собой части, можно эффективно распределить вычисления на два вычислительных узла, в этом случае расчет будет происходить без дополнительных передач данных внутри шагов численной схемы.

4.3. Решение задачи Дирихле для уравнения Пуассона.

Рассмотрим часто встречающееся в практике уравнение в частных производных – уравнение Пуассона и задачу Дирихле для него. Эта задача является хорошей иллюстрацией возможности применения параллельных вычислений и часто рассматривается в руководствах, [6, 18].

$$\begin{cases} u_{xx}(x,y)+u_{yy}(x,y)=f(x,y), & (x,y)\in D \\ u(x,y)=g(x,y), & (x,y)\in \Gamma_D \end{cases}$$

Пусть для простоты $D=[0,1]\times[0,1]$, Γ_D — граница области D. Одним из наиболее распространенных подходов численного решения дифференциальных уравнений является метод конечных разностей (метод сеток) [16-17]. Следуя этому подходу, область D решения представляется в виде дискретной сетки узлов. На рисунке 8 представлена сетка с шагом h, по каждой стороне квадрата $h=\dfrac{1}{m+1}$.

Введем обозначения $u_{i,j}=u(x_i,y_j)$. Частные производные можно аппроксимировать следующими разностными формулами с точностью второго порядка (существуют и другие способы аппроксимации производных второго порядка, [16]):

$$u_{xx}(x_i,y_j)=\frac{u_{i-1,j}+u_{i+1,j}-2u_{i,j}}{h^2}$$

$$u_{yy}(x,y)=\frac{u_{i,j-1}+u_{i,j-1}-2u_{i,j}}{h^2}$$

В этом случае уравнение Пуассона перепишется относительно сеточных функций $u_{i,j}$ в следующем виде:

$$u_{i,j}=\frac{1}{4}(u_{i-1,j}+u_{i+1,j}+u_{i,j-1}+u_{i,j-1}-h^2 f_{i,j}),\ \text{где}\ f_{i,j}=f(x_i,y_j) \qquad (3)$$

Разностные уравнения, записанные выше, позволяют определять значение для всех $u_{i,j}$ по известным значениям функции $u(x,y)$ в соседних узлах. Заметим, что система уравнений (3) представляет собой систему m^2 линейных алгебраических уравнений относительно неизвестных $u_{i,j}\ (i,j\in 1,\ldots \mathrm{m})$ во внутренних узлах сетки (значения на границе определены из условия задачи Дирихле). Матрица этой системы будет иметь ленточную структуру, а потому её можно решать соответствующими методами, к которым можно применять параллельные вычисления. Мы рассмотрим метод простой итерации решения системы. Для этого мы строим из уравнения (3) итерационную формулу следующим образом:

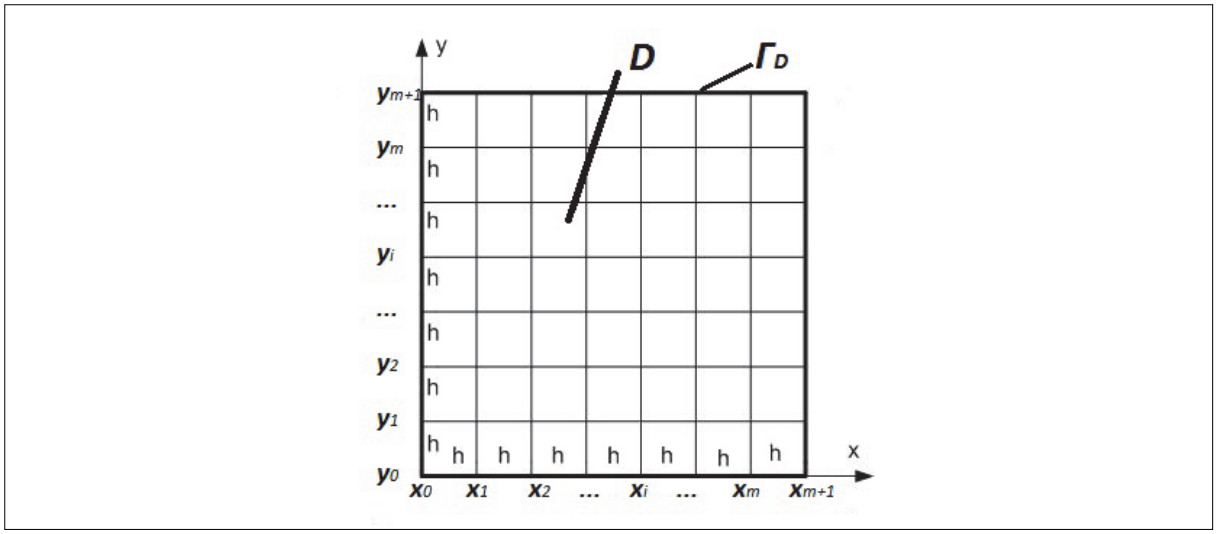

Рис. 8. Покрытие области D равномерной сеткой с шагом *h*.

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j-1}^{k} - h^2 f_{i,j})$$

Выполнение итераций обычно продолжается до тех пор, пока получаемые в результате итераций изменения значений $u_{i,j}$ не станут меньше требуемой точности вычислений, то есть

$$\max_{i,j\in[1,m]} \left| u_{i,j}^{k+1} - u_{i,j}^{k} \right| \le \varepsilon$$

Значения $u_{i,j}^{k}$ на каждом шаге итерации могут быть вычислены параллельно. Рассмотрим подробно одну из возможных схем распараллеливания метода простой итерации для задачи Пуассона. Пусть для вычисления используется N процессов, будем считать, что $m = pN$ ($m+1$– количество элементарных ячеек, на которые разбили область моделирования, рис. 8).

1. Разделим область моделирования *D* по линиям сетки на подобласти D_k следующего вида: в область D_k будут входить узлы (см. рис. 9).

   (kp,1), (kp+1, 1), ... ,((k+1)p, 1),
   (kp,2), (kp+1, 2), ... ,((k+1)p, 2),
   …,
   (kp,m), (kp+1, m), ... ,((k+1)p, m)
2. Включим дополнительно в каждую подобласть два ряда узлов по *m* в каждом, а приграничные области D_1 и D_N одним аналогичным рядом дополнительных узлов (см. рис 9). При этом собственная область процесса с номером i будет включать в себя узлы с номерами *(ip,j), (ip+1, j), ... ,((i+1)p, j),* $1 \le j \le m$. Граничные столбцы для этого процесса будут иметь номера *(ip-1,j)* и *((i+1)p, j)* $1 \le j \le m$.
   Распределим по процессам расширенные подобласти и положим счетчик итераций равным нулю.
3. Зададим для каждого процесса параллельно (для каждой расширенной подобласти) некоторое начальное приближение к решению. Это необходимо, чтобы инициировать итерационный процесс. (Здесь мы предполагаем, что начальное приближение будет достаточно хорошим, то есть итерационный процесс с таким начальным приближением действительно будет сходиться к решению задачи. На самом деле это вопрос отдельного исследования в каждом конкретном случае.)

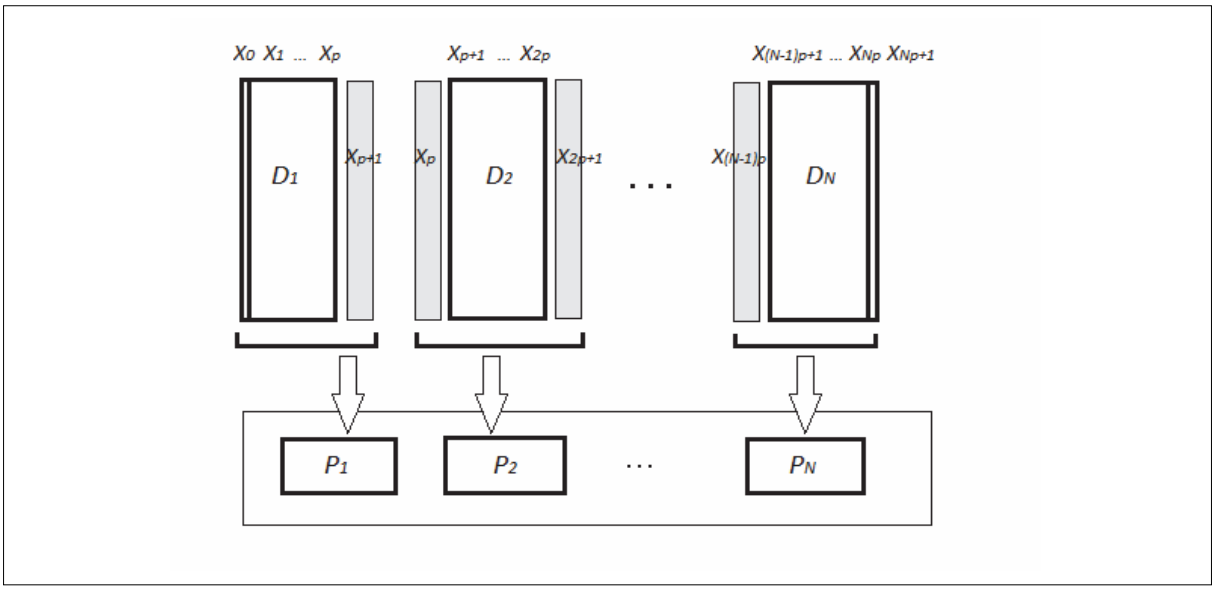

Рис. 9. Распределение работы между процессами. Серым выделены дополнительные столбцы узлов сетки для расчета задачи. $D_1 \ldots D_N$ – подобласти, $P_1 \ldots P_N$ – процессы.

4. На всех процессах параллельно вычислим следующее приближение к решению (проведем один шаг итерации):
   на процессе с номером 1 вычислим $u_{i,j}^{r+1}, i \in [1, p], j \in [1, m]$
   …,
   на процессе с номером N вычислим $u_{i,j}^{r+1}, i \in [(N-1)p+1, Np = m], j \in [1, m]$
5. Произведем обмен данными в дополнительных рядах между процессами, рассчитывающими смежные подобласти, а именно
   процесс с номером i посылает процессу с номером $i+1$ значения $u_{i,j}^{r+1}$ в ряде узлов *(ip, .)*
   принимает от процесса с номером $i+1$ значения $u_{i,j}^{r+1}$ в ряде узлов *(ip+1, .)* и записывает их в дополнительный ряд *(ip+1, .)*
   посылает процессу с номером *i-1* значения $u_{i,j}^{r+1}$ в ряде узлов *((i-1)p+1, .)*
   принимает от процесса с номером *i-1* значения $u_{i,j}^{r+1}$ в ряде узлов *((i-1)p+1, .)*
   и записывает их в дополнительный ряд *((i-1)p+1, .)*

   Процесс с номером 1 осуществляет передачу и прием данных от процесса 2, процесс с номером N осуществляет прием и передачу данных процессу *N-1*.
   Важно, что граничные столбцы, которые передаются, требуются только для расчета строк в основной области. То есть для каждого процесса при расчетах они выполняют вспомогательную роль, обеспечивая необходимые данные для расчета в основной области. После того, как значения в основной области подсчитаны для всех процессов, происходит передача и прием новых данных в качестве дополнительных граничных столбцов.
6. Параллельно на всех процессах вычисляются величины для проверки точности полученного приближения, а именно, процесс с номером i вычисляет величину

$$\varepsilon_i = \max_{i \in [(i-1)p+1, ip], j \in [1, m]} \left| u_{i,j}^{r+1} - u_{i,j}^{r} \right|$$

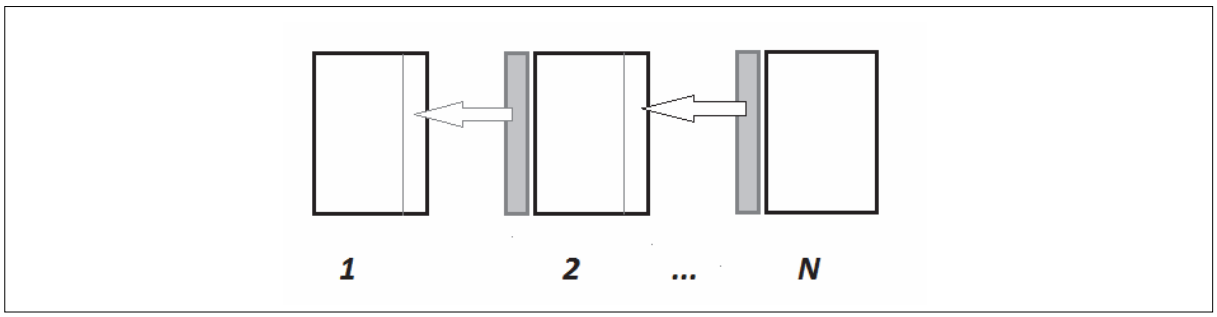

Рис. 10. Организация передачи данных, передача данных предыдущему процессу.

7. Передаем вычисленные значения процессу *root*. *Root* вычисляет максимум из полученных величин и сравнивает его с заданной точностью. Если выполняется неравенство $\max_{i==1,\ldots N} \varepsilon_i < \varepsilon$, обеспечивающее заданную точность, то нужно собрать все вычисленные $u_{i,j}^{r+1}$ на процессе *root* и закончить вычисления.

Пункт 5 алгоритма можно реализовывать разными способами. Один из способов предполагает неблокирующие передачи данных. Мы остановимся подробнее на использовании блокирующих обменов данными, поскольку применение блокирующих функций более прозрачно и надежно. Самый простой вариант организации обмена при помощи блокирующих функций состоит в организации двух последовательных операций. Во время первой процессы с номерами i, где $2 \le i \le N$, производят передачу граничного столбца предыдущему процессу с номером i-1, а затем процессы с номерами $1 \le i \le N\text{-}1$ принимают данные от следующего процесса с номером на 1 больше (см. рис. 10).

Передача данных осуществляется при помощи функций `MPI_Send` и `MPI_Recv`. При такой организации обмена данными все процессы одновременно обращаются к операции `MPI_Send` а затем дожидаются, когда соседний слева процесс начнет прием данных. Пока нужный процесс не начнет прием данных, передающий процесс находится в ожидании и не может начать прием, поскольку операция блокирующая. При такой схеме прием начнется с процесса 1, затем, после того, как произойдет прием данных от процесса 2, процесс 2 сможет окончить передачу данных и перейти к приему данных от процесса 3 и т.д. Таким образом, все пересылки данных происходят по очереди.

Можно организовать передачу данных иначе, чтобы избежать последовательного выполнения операций обмена данными. Для этого нужно изменить порядок очередности выполнения приема и передачи следующим образом: процессы с четными номерами передают данные предыдущему процессу, в то время как процессы с нечетными номерами принимают данные от следующего. Таким образом, можно организовать операции передачи граничных столбцов всего за два последовательных шага. На первом шаге все процессы с нечетными номерами выполняют передачи данных соседним процессам, а процессы с четными номерами осуществляют прием этих данных. На втором шаге наоборот – четные процессы выполняют передачу, нечетные процессы – прием.

Хорошим способом организации передачи данных является использование функции `MPI_Sendrecv`, которая может обеспечить передачу данных следующему и прием данных от предыдущего процесса. Реализация `MPI_Sendrecv` обеспечивает корректную работу на крайних процессорах, когда не нужно выполнять одну из операций приема или передачи.

Пункт 7 алгоритма можно организовать при помощи функции `MPI_Reduce` с использованием операции `MPI_MAX`.

Можно производить разделение рабочей области D не на столбцы, а на строки или блоки. При разбиении по строкам алгоритм аналогичный, при разбиении на блоки нужно будет поменять алгоритм обмена данными, поскольку у каждой области будет 4 соседних, соответственно нужно будет передавать 4 граничных строки. Очевидно, что при блочном разбиении увеличивается количество обменов данными, поэтому такая схема будет оправданной, если число узлов сетки будет достаточно большим.

При блочном представлении сетки может быть реализован также и волновой метод выполнения расчетов, подробнее о нем можно узнать, например в [6].

5. Молекулярная динамика

5.1. Постановка задачи

В качестве классической иллюстрации эффективного применения параллельных вычислений можно привести пример программы для моделирования молекулярной динамики. В последнее время задача молекулярной динамики становится все более популярной, благодаря достаточной простоте формулировки и большим научным возможностям. Моделирование определенных веществ или небольших биологических систем на атомном уровне с помощью молекулярной динамики позволяет сделать некоторые выводы или предположения о различных макроскопических свойствах этих систем. Благодаря появлению суперкомпьютеров и возможности применения параллельных алгоритмов к задачам молекулярной динамики стало возможным рассчитывать системы значительных размерностей, поскольку именно они представляют наибольший интерес.

Первые работы по молекулярной динамике были написаны в конце 50-х – начале 60-х годов, расчет одного шага траектории занимал около минуты, а системы содержали не более 1000 частиц [19-21]. В настоящее время вычислительные и алгоритмические методы и компьютерные технологии продолжают развиваться, и становится возможным рассчитывать большие системы за меньшее время [22].

Молекулярной динамикой называют метод, в котором временная эволюция системы взаимодействующих частиц (атомов, молекул или более сложных агрегатов) отслеживается интегрированием их уравнений движения. Поведение отдельной частицы описывается классическими уравнениями движения и имеет вид:

$$m_k \ddot{r}_k = F_k$$

где k – номер частицы, m – масса, $\ddot{r}_k$ – радиус-вектор, F_k – полная сила, действующая на частицу. Таким образом, в основании молекулярной динамики лежит классическое представление о динамике системы. Для учёта квантовых эффектов иногда употребляются гибридные методы молекулярной динамики и квантово-механических расчётов. Мы, однако, ограничимся рассмотрением принципов классической молекулярной динамики.

Молекулярная динамика строит *классические* модели молекулярных систем. Это позволяет использовать макроскопические аналогии и говорить, например, что ковалентные связи представляются жёсткими стержнями (или пружинами) между шариками-атомами. Такое соответствие между реальными молекулами и механистическими моделями определяется конкретной *молекулярно-динамической моделью*. Так, различные модели могут по-разному интерпретировать одну и ту же молекулу, и, как следствие, давать различные особенности её поведения. Кроме того, для различных типов молекулярных комплексов (белков, жиров, нуклеиновых кислот) обычно применяют различные модели, более точно моделирующие конкретный тип молекул. Такие модели ещё называют *силовыми полями*, и существующие пакеты для молекулярно-динамических расчётов могут поддерживать как одно, так и несколько различных силовых полей. Се-

годня наиболее распространены силовые поля AMBER(*Assisted Model Building with Energy Refinement*) и CHARMM (*Chemistry at HARvard Macromolecular Mechanics*).

В силу того, что предметом исследования молекулярной динамики обычно является некоторый молекулярный комплекс, находящийся в окружении растворителя, можно выделить два вклада во взаимодействие: взаимодействия со всеми атомами комплекса, и силы взаимодействия с молекулами окружающей среды (растворителя). Потенциальную энергию можно представлять в виде суммы вкладов от различных типов взаимодействия между частицами системы. Для различных целей могут использоваться различные виды потенциального вклада в равнодействующую.

Типичная задача молекулярной динамики состоит в рассмотрении системы из N частиц в замкнутой области. Для того чтобы получить движение системы нужно численно проинтегрировать уравнения движения, то есть решить систему $3N$ дифференциальных уравнений второго порядка, которую можно свести к $6N$ дифференциальным уравнениям первого порядка. Кроме того, чтобы с помощью конечного числа частиц смоделировать бесконечное пространство используются периодические граничные условия. Этот метод заключается в том, что частица, вышедшая при динамике за пределы рассматриваемой области, появляется в этой же области с противоположной стороны, так же, как она вошла бы в соседнюю область.

При этом действие частицы у одной границы области распространяется так же на частицы у противоположной границы области (см. рис. 11). Другими словами, рассматриваются не только сами частицы, но и их образы, находящиеся за границами моделируемой ячейки. Математически данный подход равносилен рассмотрению задачи на торе.

Моделирование молекулярно-динамической системы представляет собой численное решение задачи Коши для дифференциальных уравнений Ньютона. Теория решения этой задачи разработана достаточно хорошо, и сегодня существуют различные алгоритмы, позволяющиепреодолеть эту проблему. Они отличаются, в основном, точностью расчета, трудоемкостью вычислений и устойчивастью.

В молекулярной динамике широко применяется метод Верле [23]. Этот метод предназначен для решения систем дифференциальных уравнений второго порядка, каковыми являются уравнения Ньютона. Принцип работы алгоритма заключается в том, что, зная координаты, ускорения и координаты системы на предыдущем шаге $r(t-\delta t)$, можно вычислить положение частиц в момент времени по формуле:

$$r(t+\delta t) = 2r(t) - r(t-\delta t) + \delta t^2 a(t) \qquad (1)$$

Отметим одно важное обстоятельство. Данный метод является разновидностью методов *Адамса* интегрирования диференциальных уравнений. Преимущество данного класса методов состоит в том, что они являются *симплектическими*, что проявляется в том, что они достаточно хорошо сохраняют интегралы движения, [24].

Стоит отметить, что в этом выражении отсутствуют скорости. Они взаимоуничтожаются при сложении двух выражений формулы Тейлора для $r(t+\delta t)$ и $r(t-\delta t)$. Скорости не требуются для вычисления траектории, но они важны для вычисления кинетической энергии, а значит и для полной энергии системы. Их можно получить из выражения

$$v(t) = \frac{r(t+\delta t) - r(t-\delta t)}{2\delta t} \qquad (2)$$

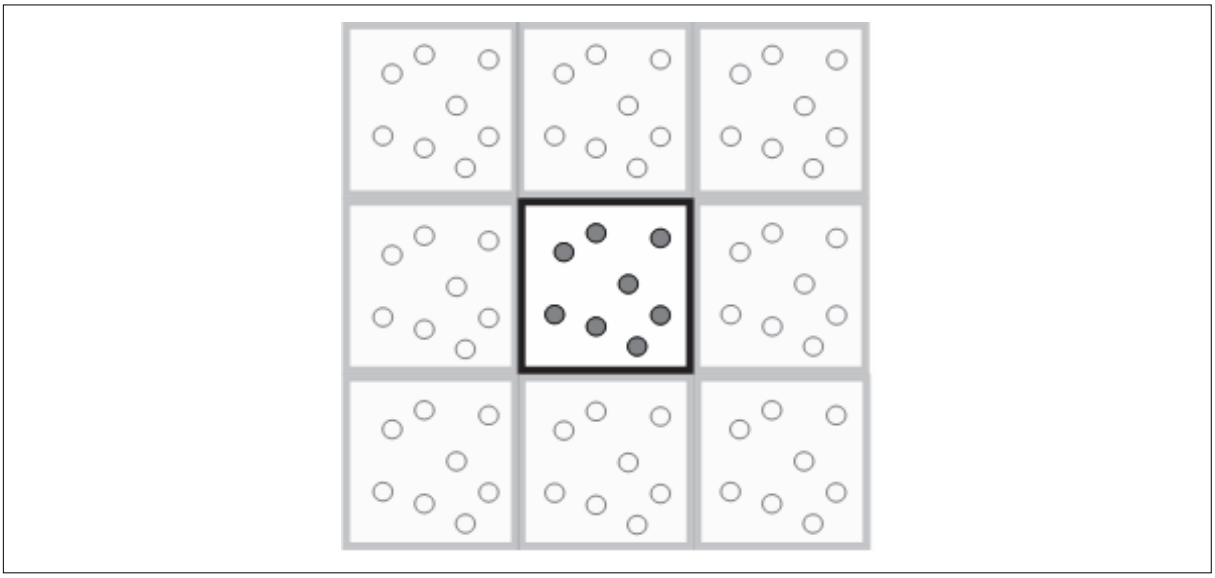

Рис. 11. Периодические граничные условия для двумерного случая. Центральная ячейка – область моделирования, остальные ячейки – образы.

Тогда как равенство (1) имеет четвертый порядок точности, выражение для скоростей (2) имеет точность лишь второго порядка. Ускорения на текущем шаге вычисляются из формулы закона Ньютона. Используются также и различные модификации метода Верле, о них можно подробнее прочитать в [23]. Мы, следуя [25], будем использовать модификацию *LeapFrog*. В этом случае интергирование проводится в два шага. На первом шаге вычисляются

$$v(t + \frac{\delta t}{2}) = v(t) + \frac{\delta t}{2} a(t)$$

$$r(t + \delta t) = r(t) + \delta t v(t + \frac{\delta t}{2})$$

На втором шаге вычисляется величина

$$v(t + \delta t) = v(t + \frac{\delta t}{2}) + \frac{\delta t}{2} a(t + \delta t)$$

Сравнение численых алгоритмов для решения задач молекулярной динамики – тема отдельного исследования. Поскольку в большинстве задач используется именно этот метод интегрирования, мы также рассмотрим именно его.

5.2. Возможности применения параллельных вычислений к молекулярной динамике

Для исследования задач молекулярной динамики в настоящее время создано много готовых решений. Существуют программные пакеты, которые позволяют проводить достаточно успешное моделирование различных систем. Но использование готовых решений обладает так же рядом недостатков.

Так, если исследователь не имеет значительного опыта в использовании численных методов, он может по ошибке выбрать неверные параметры моделирования. Напротив, при написании собственной программы автор осмысливает используемые алгоритмы, стараясь подобрать параметры счёта наилучшим образом.

В случае если пользователь достаточно опытен, можно обратить внимание на следующие моменты. Во-первых, программа, нацеленная на конкретную модель, как правило, значительно быстрее той, которая имеет большое число параметров и подключаемых возможностей. Вторым моментом является гибкость собственной программы по сравнению с чужим продуктом. Так, автор программы всегда сможет добавить в неё вычисление некоторой нестандартной величины, в то время как изменение сторонней программы может потребовать значительных усилий или вообще оказаться невозможным. И, наконец, чем больше программа, тем больше вероятность того, что в ней есть ошибки. В этом отношении, написанная собственными руками небольшая программа всегда будет проще в проверке, нежели продукт из тысяч строк кода. Вообще говоря, готовое программное решение не обязательно должно правильно работать на всех возможных входных данных.

Подводя итог вышесказанному, можно заключить, что написание собственных программ моделирования является оправданным и полезным. Использовать готовые продукты следует лишь тщательно разобравшись в принципах их работы и лишь тогда, когда затраты на написание программы могут оказаться неоправданно большими.

Остановимся подробнее на возможностях применения методов параллельного программирования к задачам молекулярной динамики. Системы для молекулярной динамики с короткодействующими силами являются системами, наиболее удобными для применения параллельных вычислений, поскольку можно выделить части системы, которые между собой не взаимодействуют, и рассчитывать их параллельно.

Другим способом получения более высокой производительности является представление данных в форме вектора, и применение векторного подхода к распараллеливанию с последующим использованием графических процессоров (*GPU*). Ограничением такого способа является то, что данные должны быть организованы в относительно длинные векторы, причём так, что все элементы могут быть обработаны независимо, то есть, расчет одного элемента не будет влиять на расчет остальных. Такое условие может быть выполнено далеко не всегда. По этой причине, несмотря на то, что в настоящее время в программных пакетах начинают появляться возможности для применения этого метода, он будет эффективным лишь для ограниченного класса задач.

Конечно, эти решения являются лишь наиболее часто встречающимися, существуют и другие. Ценой их эффективного использования является увеличение сложности алгоритмов и программного обеспечения. В отличие от "простых" компьютеров, где оптимизирующие компиляторы для типовой программы могут помочь добиться приемлемой производительности, проблемы параллельной и векторной обработки не всегда могут быть решены автоматически, потому что не всегда очевидно, при условии, что это вообще возможно, как организовать процесс автоматизации. Кроме собственно написания рабочей параллельной программы, очень важна эффективность организации вычислений. Параллельные вычисления с большими накладными расходами (межпроцессорное взаимодействие, прием-передача данных) не дадут существенного выигрыша в скорости, векторизация вычислений, реализованная неэффективно (векторы слишком короткие, или перестановка данных к требуемой форме осуществлена со значительными затратами по времени) тоже не принесет желаемого ускорения.

Поскольку классификация мультипроцессорных систем достаточно сложна, существует множество факторов, которые нужно принимать в расчет при разработке. Отдельные процессы выполняют один и тот же набор операций в течение работы или они могут действовать независимо друг от друга, выполняя различные задачи; каждый процесс имеет собственную память или для всех существует общая память, или доступны обе возможности; передача данных между процессорами реализована через сети связи

или через общую память; топология сетей связи. Некоторые из этих факторов могут оказывать влияние на то, как следует организовывать вычисления.

Рассмотрим наиболее часто встречающуюся ситуацию, когда имеется несколько процессов с независимой памятью. Такой вычислительный комплекс можно организовать, например, соединив несколько компьютеров по локальной сети. В этом случае для эффективности реализации параллельных вычислений необходимо свести к минимуму передачи данных между процессорами. Постараемся изложить один из наиболее эффективных алгоритмы распараллеливания молекулярной динамики с учетом этих требований.

Существуют различные пути распределения вычислений молекулярной динамики на несколько процессоров. Можно разделить вычисления потенциала на независимые части, или рассматриваемые атомы разделить на группы, динамика для которых считается независимо, или разделить пространство на ячейки, атомы в которых будут рассчитываться параллельно. Если попытаться распределить только вычисление потенциала, то информация обо всей моделируемой системе должна быть доступна каждому процессу. Такой подход является наиболее просто реализуемым, и его применение уже может привести к достаточному ускорению работы в ряде задач. Поэтому после того, как произведен расчет системы для следующего момента времени, необходимо, чтобы вся информация о системе опять была доступна всем процессам. Таким способом можно распараллеливать достаточно небольшие системы, так же такой подход можно применять при использовании вычислительных комплексов с общей памятью. Пример эффективного подхода применения параллельных методов для вычислительного комплекса с общей памятью приведен в статье [26].

Второй способ предполагает, что каждый процесс занимается расчетами траектории одного атома, не зависимо от его текущего положения в пространстве. Первая работа была написана в 1989 году [27]. Такой подход будет эффективен при расчетах с дальнодействующими потенциалами. Но если у потенциала радиус действия невелик, как это чаще всего случается, то наиболее эффективным будет использование третьей схемы.

Можно разделить область моделирования в пространстве на подобласти, каждый процесс будет рассчитывать траекторию для своей подобласти [28]. Все атомы, находящиеся внутри области, рассчитываются одним процессом; для атомов, находящихся на границе, нужны дополнительные данные об атомах из соседней области для расчета движения (см. рис. 12).

Как только атом совершает перемещение в соседнюю область, все связанные с ним величины передаются от одного процесса другому. Если предположить, что для трехмерной задачи расчет ведется по крайне мере 10^4 или более частиц для подобласти, и потенциал взаимодействия имеет короткий радиус действия, то большая часть взаимодействий будет происходить для атомов внутри области, и лишь малая часть атомов в приграничной области будут взаимодействовать с соседними, относящимися к другой подобласти. По сравнению с общим числом атомов рассчитываемой системы сравнительно небольшое число атомов будет совершать перемещения между областями, поэтому получается сэкономить на передачах между процессорами, что позволяет уменьшить время расчета. При таком подходе обычно наблюдается линейный рост времени работы программы при увеличении числа частиц и при уменьшении числа процессоров.

В дополнение к инициализации, вычислениям взаимодействия и интегрированию уравнений движения, важно уточнить, какой процессор рассчитывает какую область, определить, какие атомы участвуют в передаче и грамотно провести передачу данных. В данной задаче передача данных производится с несколькими целями:

- для вычисления взаимодействий атомов в приграничной области (передача данных о приграничных атомах соседних областей)

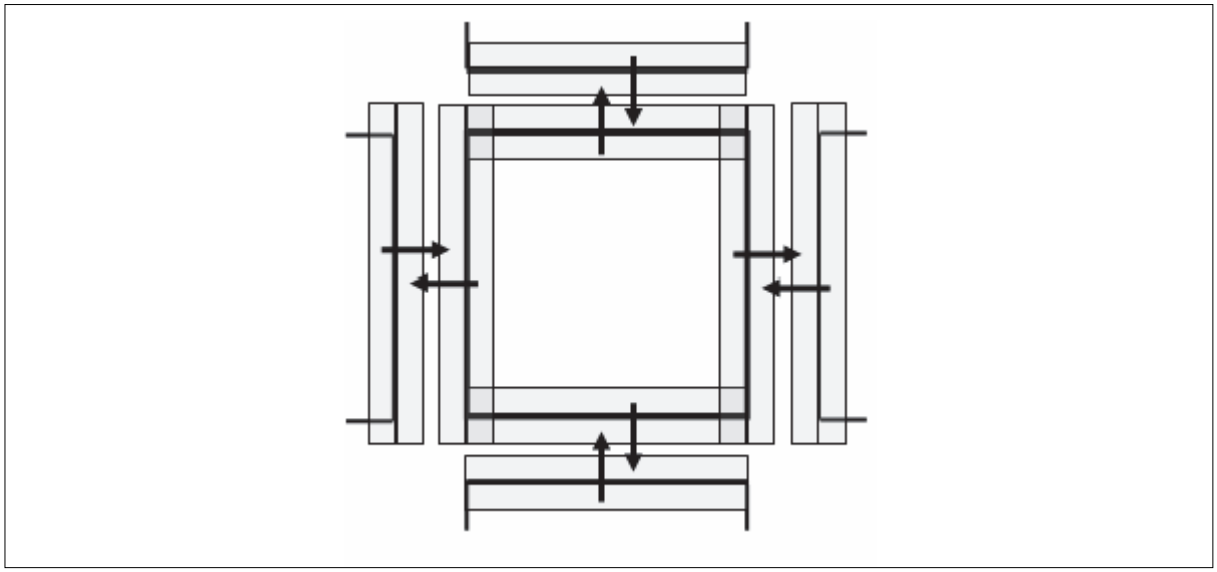

Рис. 12. Часть области моделирования (для двумерного случая). Представленная квадратная область обрабатывается одним процессом, затененные места обозначают те области, атомы которых взаимодействуют с атомами, рассчитывающимися соседними процессами. Стрелками обозначен обмен данными.

- при перемещении атома в соседнюю область (передача всех данных, касающихся этого атома)
- при вычислении дополнительных величин, например, энергии системы, вычисления проводятся отдельно для каждой области, а потом должны быть объединены для получения итогового результата.

Можно проиллюстрировать схему следующим образом:

1. Разбиение пространства на ячейки. Если радиус действия потенциала a, то удобно разбить на ячейки размера $a \times a$ Действие производится до начала параллельной части, выполняется на одном процессоре.
2. Разделение работы между сопроцессами. На этом шаге каждый сопроцесс получает данные о частицах, с которыми ему придется работать.
3. Определение частиц, которые находятся в граничных ячейках.
4. Расчет частиц в граничных ячейках с частицами внутри ячейки.
5. Обмен информацией о частицах в граничных ячейках с соседними сопроцессами.
6. Расчет частиц в граничных ячейках с частицами граничных ячеек других процессов (данные о них были получены на предыдущем шаге).
7. Расчет частиц в остальных ячейках подобласти.
8. Пересчет координат частиц.
9. Обмен данными о частицах, сменивших свою подобласть.
10. Привязка частиц к новым ячейкам.

Несмотря на то, что схема является достаточно ясной, некоторые детали должны быть оговорены подробнее.

5.3. Организация вычислений

Сначала нужно распределить исходные данные по процессорам таким образом, чтобы для каждого были определены соответствующие области, памяти были доступны данные об атомах, находящихся внутри области, а так же данные атомов соседних об-

ластей, необходимые для вычисления взаимодействия. Так же для того, чтобы начать вычисления необходимо получить начальное состояние системы. Это может быть реализовано различными способами, в зависимости от задачи. Если начальное состояние как-то задано, то *root*-процессор может считывать данные из файла, а затем рассылать каждому исполняющему процессору свою часть данных (для этого можно использовать функцию `MPI_Scatter()`).

Если начальное положение можно задать случайно или с использованием какой-нибудь функции, это можно сделать параллельно. В таком случае для получения начального состояния каждый процессор исполняет функцию инициализации, учитывая, какие частицы находятся в его подобласти. После того, как каждый процессор провел инициализацию своей части данных, root-процесс должен собрать с исполнителей информацию о скоростях и центрах масс, чтобы убедиться, что система задана корректно и центр масс имеет нулевую скорость, и, наконец, передать данные об успешной инициализации всем процессам. Соответственно, исполнительные процессора выполняют инициализацию, расчет центра масс и скоростей, и передают эту информацию root-процессу, ожидая сообщения об успешной инициализации чтобы продолжать работу. Как уже упоминалось ранее, root-процесс выполняет еще и работу исполнителей, он не простаивает во время вычислений.

Далее для ускорения вычислений и прозрачной организации передачи данных для расчета взаимодействий на границе будет удобно составить список соседей. Суть составления списка соседей заключается в том, чтобы не перебирать на каждом шаге интегрирования всевозможные пары молекул, вычисляя между ними расстояния, а сохранять в списке те, которые находятся на удалении не более . Список может обновляться каждые M шагов интегрирования (), с учетом того, что молекулы за это время не проходят расстояние большее $\Delta r_L = r_L - r_c$, где некоторая константа, зависящая от диаметра частицы. Обычно считают , где σ – диаметр молекулы. Δr_L берется обычно равным [26]. Такой метод немного снижает точность, но позволяет выиграть до 20% времени вычислений. Существуют различные подходы к реализации метода составления списка соседей. Можно хранить номера только взаимодействующих между собой молекул, тогда размер памяти для списка соседей выбирается исходя из примерного числа взаимодействующих молекул [29]. Другой подход, [25], основан на хранении информации о каждой паре, такой подход требует дополнительных затрат памяти, но позволяет сэкономить до 5% вычислительного времени.

При реализации следует обратить внимание на то, что ячейки и соседи определяются отдельно для каждой области пространства. Так же для границы каждой области определяется слой ближайших соседей, которые должны полностью окружать область (затененные области на рисунке 12). Периодические граничные условия будут удовлетворяться при передаче данных между процессорами, это будет подробно рассказано позже на примере. За основу указанных здесь алгоритмических и программных решений взят пример из книги Раппапорта [25].

По сравнению с последовательной организацией программы нет больших отличий в вычислениях взаимодействий между частицами. Для приграничных частиц существуют разные случаи: ячейка на грани, на ребре или в углу, для каждого случая предусматривается взаимодействие с требуемыми образами. Данные о нужных соседних частицах передаются на каждый вычисляющий процесс. Периодические граничные условия аналогично не будут влиять на работу этой функции.

5.4. Организация программы

Приведенная здесь программа написана на языке C++. Перечислим основные структуры, которые используются в программе.

Для проведения вычислений потребуется векторная алгебра, для чего удобно ввести структуру вектора (трехмерного) и написать функции для работы с векторами. Мы вводим структуру вектора следующим образом:

```
typedef struct {
      double v[DIM]; // DIM - размерность задачи, в нашем случае - 3.
}
Vector;
```

Определяются так же сопутствующие функции для использования векторной алгебры: сложение векторов, умножение вектора на число, скалярное произведение. Можно их прописать как функции, можно использовать перегрузку операторов, мы предпочли второй вариант. На основе структуры вектора вводится структура данных для задачи молекулярной динамики, то есть структурируются данные о переменных, использующихся в задаче. Каждая частица характеризуется радиус-вектором, скоростью и ускорением. Поэтому удобно ввести структуру `Mol`, которая включает в себя эти характеристики. Для составления списка соседей для каждой частицы вводится порядковый номер `int id`. При работе программы для каждой молекулы он остается неизменным.

```
typedef struct {
  Vector r, rv, ra;  // радиус-вектор, скорость и ускорение частицы
  int id;            // порядковый номер (для списка соседей)
} Mol;
```

Для составления списка соседей удобно воспользоваться структурой двумерного целочисленного вектора, поскольку список соседей состоит из списка пар взаимодействующих частиц. Структура определяется аналогично структуре `Vector`.

Все используемые структуры перечислены.

Программа начинается как обычно с инициализации MPI и запуска параллельных процессов. После этого на каждом процессе проводится инициализация системы. В нашем случае данные считываются из файла процессом root, пространство моделирования разбивается на блоки в зависимости от желаемого числа процессов, данные рассылаются по исполняющим процессам для дальнейших вычислений.

Основной функцией является проведение шага интегрирования системы. Поговорим о ней подробнее.

Интегрирование производится, как уже было сказано, с использованием популярной модификации метода Верле, *LeapFrog*. Единственное изменение в функции, отвечающей за шаг интегрирования, по сравнению с обычной последовательной реализацией, следующее: вместо общего числа атомов используется число атомов в ячейке, `nMolMe`. Поскольку *LeapFrog* состоит из двух шагов, причем новые координаты вычисляются на первом шаге, передача данных об атомах, вышедших за границу происходит между шагами метода. То есть, скорости для частицы, вышедшей в новую область, рассчитываются уже на новом процессоре. Атомы, данные о которых были скопированы для корректного расчета взаимодействия, не учитываются. Как уже было упомянуто, при вы-

числениях взаимодействий не будет существенных отличий по сравнению с обычной непараллельной реализацией программы.

Ниже приведен код этой функции с комментариями.

```
1   void SingleStep () {
2         ++stepCount;                    //номер шага увеличивается на 1
3         timeNow = stepCount * deltaT;   // делается шаг по времени
4         LeapfrogStep(1);                // первая часть схемы интегрирования
5         if (nebrNow > 0)                // есть ли вышедшие за границу, если есть,
6               DoParlMove();             // то сделать передачу координат
7         DoParlCopy ();                  // сделать копию соседей
8         if (nebrNow > 0) {              // если были вышедшие за границу, то
9               nebrNow = 0;              // теперь их больше нет
10              if (rank == 0) displ = 0.;
11              BuildNebrList();          //создать список соседей
12        }
13        ComputeForces();    // вычислить взаимодействия (копии соседей получены)
14        LeapfrogStep(2);    // вторая часть схемы интегрирования
15  ComputeDisplacement();    // вычисляется общее максимальное смещение
16  if (rank == 0){
17        if (displ > rDatum)
18              nebrNow = 1; //если смещение > заданной величины,
19                           //надо будет обновить список соседей.
20        MPI_Bcast(&nebrNow, 1, MPI_INT, 0, MPI_COMM_WORLD );
21                                    //передать значение nebrNow всем
22  }
23  else
24        MPI_Bcast(&nebrNow, 1, MPI_INT, 0, MPI_COMM_WORLD );
25                                    // получить значение nebrNow
26        /*Можно добавить код, параллельно вычисляющий дополнительные параметры,
27        энергию, давление,  центр масс. Подробнее в [25]*/
28  }
```

Операции коммуникации между процессорами здесь осуществляются при помощи текущей величины переменной **int** *nebrNow*, которая равна 1, если для этого процессора определены частицы, вышедшие за границу области, и равна нулю, если таких частиц нет. Величина **int** *nebrNow* полагается равной 0 или 1 в зависимости от того, какое максимальное смещение было совершена атомами за ход, то есть, если общее смещение достаточно большое, то частицы, оказавшиеся за границами области, есть и нужно сделать перерасчет списка соседей. Следует отметить, что передачи между процессами могут происходить не на каждом шаге, и пересчет списка соседей не будет производиться, если общее смешение всех молекул на шаге было не достаточным. Максимальное смещение вычисляется при помощи функции `ComputeDisplacement()` (строка 15), которая так же реализуется параллельной. Организовать ее можно по следующему принципу: каждый процесс подсчитываем общее смещение в своей области, потом вычисленные значения передаются на root-процессор, который суммирует полученные данные.

Функции `DoParlMove()` и `DoParlCopy()` (строки 6 и 7) отвечают за передачу данных между исполнительными процессорами, когда *nebrNow* = 1, то есть частица переходит из области, рассчитываемой одним процессором в область, рассчитываемую другим.

Еще раз подчеркнем важную деталь: при составлении параллельной программы существенно соблюдать соответствие между отправленными и полученными сообщениями. Если данные отправлены каким-то процессором, то каким-то процессом (или какими-то процессами) они должны быть получены. Существенно, что операции по приему-передаче сообщений так же способствуют синхронизации. Когда разрабатывается программа с возможностью работы в параллель на нескольких процессах, очень

важно следить за синхронизацией их работы, иначе могут появляться ошибки, которые трудно диагностировать.

Теперь остановимся подробнее на организации функций, ответственных за передачу и прием данных о частицах, вышедших за границы области моделирования для фиксированного процесса и функций, копирующих данные о соседях для расчета взаимодействий.

Функции, осуществляющие обмен данными о вышедших за границы области частицах, учитывают периодические граничные условия. Один атом может участвовать в нескольких перемещениях. Так же нужно учитывать случай, когда область является из-за периодических граничных условий своим же соседом, (то есть, например, частица, вышедшая за границу области справа должна появиться в этой же области слева). После того, как выполнены все перемещения, происходит перезапись данных о частицах в массивах так, чтобы устранить пробелы. Сначала несколько слов об используемых обозначениях.

Введем следующее обозначение для сокращения записи функции:

```
#define OutsideProc
   (subDir == 0 && mol[n].r.v[dir] < subRegionLo.v[dir] ||
   subDir == 1 && mol[n].r.v[dir] >= subRegionHi.v[dir]))
```

где, напомним, *mol*[n].*r*.*v*[dir] является компонентой с номером dir вектора *mol*[n].r.

OutsideProc – условие на то, что частица вышла за границу области, рассчитываемую процессором. Векторные величины *subRegionLo* и *subRegionHi* – содержат информацию о границе области, рассчитываемой процессом. Эти величины инициализируются при разделении общей работы между процессами в зависимости от общего количества процессов.

На рисунке 13 точки с координатами *subRegionLo* и *subRegionHi* обозначены A и B. Система трехмерная и подобласть расчета имеет форму прямоугольного параллелепипеда, поэтому можно считать, что есть шесть направлений, соответствующие шести граням, за которые может выйти частица при моделировании. Для удобства направления разделены на две группы, первая – с величиной *subDir* = 0, соответствует граням 1, 2 и 3, вторая, – с величиной *subDir* = 1, соответствует граням 4, 5 и 6. Тогда условие OutsideProc можно пояснить следующим образом: «если рассматривается первое направление и частица вышла за границу со стороны граней 1, 2 или 3, или рассматривается второе направление и частица вышла за границу со стороны граней 4, 5 или 6».

Еще несколько переменных, не упомянутых ранее:

```
int**   trPtr        массив номеров атомов, которые следует переместить.
int     nOut[2][DIM] массив из шести (в трехмерном случае) элементов, туда будет
                     записано количество  атомов, которые следует переместить в
                     каждом из направлений.
double* trBuff       размер буфера для передачи данных между процессорами
int     trBuffMax    размер буфера trBuff
Vector  procNebrLo,  номера процессоров, которые рассчитывают соседние области;
        procNebrHi   векторы организованы для удобства организации передачи данных.
```

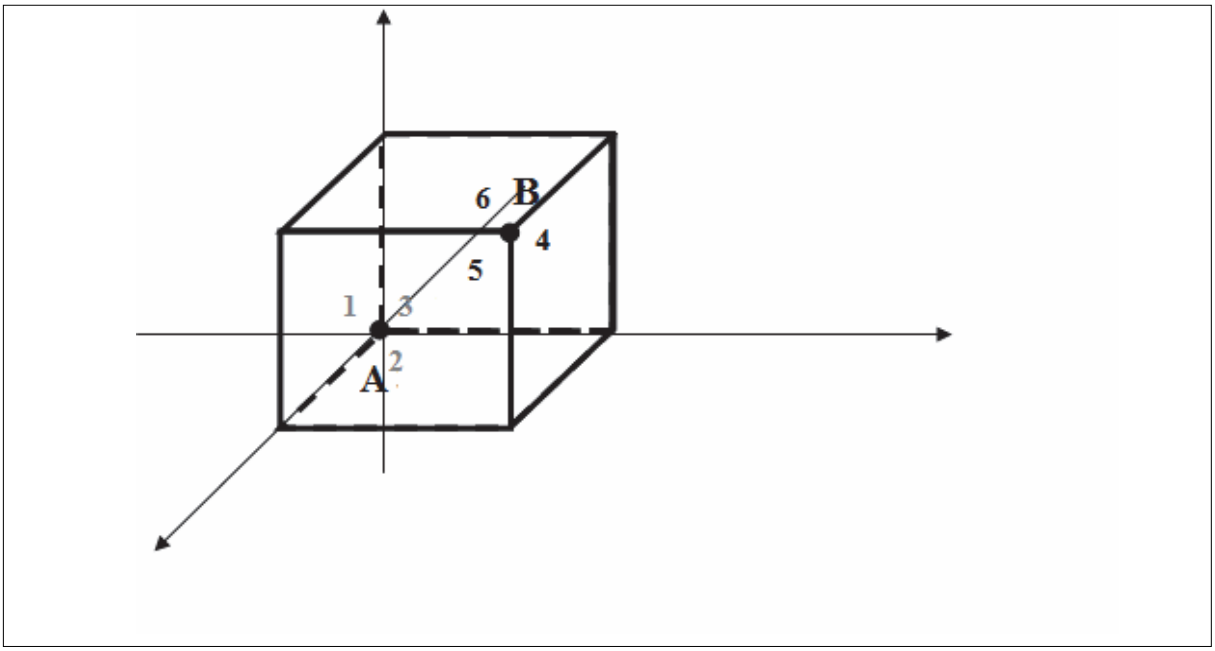

Рис. 13. Нумерация граней для организации передачи данных между процессами. Точке A соответствуют грани 1,2 и 3; точке B соответствуют грани 4, 5 и 6.

Процессы с номерами *procNebrLo* рассчитывают области, соседние по направлению *subDir* = 0, есть, процессор с номером *procNebrLo*.v[0] является соседним со стороны грани 1, и т.д; процессора с номерами *procNebrHi* рассчитывают области, соседние по направлению *subDir* = 1, то есть процессор с номером *procNebrHi*.v[0] является соседним со стороны грани 4.

Приведем список функций, которые будут использоваться в дальнейшем с кратким описанием.

```
MsgSendRecv(int from, int to, int id); передать сообщение от процессора с номером
                                       from к процессору с номером to
                                       (эквивалент MPI_Sendrecv)
PackMovedData();
UnpackMovedData(nIn);
PackData();                            упаковать данные для отправки
UnpackData();                          распаковать полученные данные
```

С учетом этих обозначений приведем код функции, отвечающей за определение и передачу вышедших за границу частиц.

```
#define NWORD_COPY  (DIM + 1)
void DoParlMove ()
{
  int dir, sDir, nt, n, nIn; // счетчики: размерность, направление, число частиц
                             // к перемещению,
  for (dir = 0; dir < DIM; dir++) {        //цикл по размерности
    for (sDir = 0; sDir < 2; sDir++) {     //цикл по направлениям
      nt = 0;                          // положили число частиц к перемещению нулем
      for (n = 0; n < nMolMe; n++) {   //цикл по всем частицам в области
        if (mol[n].id >= 0) {
          if (OutsideProc (0.)) { // если выполнено условие выхода за границу,то
            trPtr[sDir][trBuffMax * dir + nt] = n; //записать номер частицы
            ++nt;                            // число частиц к перемещению стало на
                                             // 1 больше
```

```
            if (NWORD_MOVE * nt > DIM * trBuffMax) { //если слишком много
                                                    //частиц надо
              errCode = ERR_COPY_BUFF_FULL;  // переместить, то ошибка записи в
                                             // буфер
              --nt;                         // уменьшить число частиц на одну.
            }
          }
        }
      }
      nOut[sDir][dir] = nt;       // записали количество атомов к перемещению в
                                  // соответствующем направлении
    }
    for (sDir = 0; sDir < 2; sDir++) {   //цикл по направлениям
      nt = nOut[sDir][dir];       // положили nt равным числу атомов
                                  // к перемещению в этом
                                  // направлении
      PackMovedData(dir, sDir, &trPtr[sDir][trBuffMax * dir], nt);
                          // подготовили данные о нужных атомах к перемещению
      if (procArraySize.v[dir] > 1) {
        PackData();    // упаковать данные к передаче
        if (sDir == 1) // перемещаем в зависимости от направления
          MsgSendRecv(procNebrLo.v[dir], procNebrHi.v[dir],
                                                  140 + 2*dir + 1);
        else
          MsgSendRecv(procNebrHi.v[dir], procNebrLo.v[dir],
                                                  140 + 2*dir);
        UnpackData();  // распаковать полученные данные
      }
      else
         nIn = nt;
      if (nMolMe + nIn > nMolMeMax) {
        errCode = ERR_TOO_MANY_MOVES;
        nIn = 0;
      }
      UnpackMovedData(nIn);
    }
  }
  RepackMolArray(); //перегруппировать атомы таким образом, чтобы не было проме-
жутков
                    // в записи массива.
}
```

Функция `MsgSendRecv(from, to, id)` реализуется функцией библиотеки MPI `MPI_Sendrecv (buffSend, buffWords, MPI_DOUBLE, to, id, buffRecv, BUFF_LEN, MPI_DOUBLE, from, id, MPI_COMM_WORLD, &mpiStatus)`. О функции `MPI_Sendrecv ()` рассказано подробнее в разделе 3.3.2, а переобозначение здесь введено для того, чтобы сделать код более читаемым.

Функции `PackData()` и `UnpackData()` состоят из двух этапов: надо запаковать (соответственно распаковать) данные типа `int` о списке соседей, а так же данные типа `double` о координатах. Приведем явно код функции о упаковке и распаковке массива типа `int`, функции для типа `double` будут аналогичными.

```
1   void PackInt (int *w, int nw)
2   {
3     int n;
4     if (buffWords + nw >= BUFF_LEN) { // если не хватает размера буфера
5       errCode = ERR_MSG_BUFF_FULL;    // вывести сообщение об ошибке
6     }
7     else {
8       for (n = 0; n < nw; n++)
9         buffSend[buffWords + n] = (int) w[n]; //заполнить буфер отправки
10      buffWords += nw;
```

```
11    }
12  }
13
14  void UnpackInt (int *w, int nw)
15  {
16    int n;
17    for (n = 0; n < nw; n++)
18      w[n] = (int) buffRecv[buffWords + n];   // распаковать из буфера приема
19    buffWords += nw;
20  }
```

Функции PackMovedData и UnpackMovedData отвечают за упаковку для передачи данных о молекулах, которые переместились в область расчета другого процесса. Данные о молекулах записываются в массив буфера передачи данных подряд: координата, скорость и ускорение, таким образом из трех векторов получается массив длины девять. Так происходит для всех частиц. Потом при получении данных из массива собираются обратно структуры векторов.

Можно поступить иначе: создать производный тип для вектора, на основе него создать производный тип для структуры молекулы, и для обмена данными о молекулах пользоваться полученными типами.

Функция копирования данных с соседних процессоров строится аналогично. По той же схеме производится определение приграничных частиц, данные о которых нужно скопировать. В этом случае нужно будет поправить условие выхода из приграничной зоны. Построить новое условие можно на основе условия `OutsideProc`, описанного выше. Для этого можно считать границы области расширенными на требуемую величину.

В случае использования системы нескольких процессоров, у которых совместный доступ к общей памяти, существует два подхода. Один из них использует ту же схему для передачи сообщений, описанную выше. В зависимости от операционной системы и компьютера, вполне возможно, что такой алгоритм будет работать быстрее, чем если бы процессам приходилось общаться по сети, потому что существуют специализированные функции для внутренней связи. Альтернативным является использование подхода, основанного на потоках (threads). В настоящее время подход с использованием потоков становится все более популярным в связи с широкой доступностью графических процессоров, и известные программные решения адаптируются к этой парадигме параллельных вычислений. Но общая эффективность использования потоков зависит от архитектуры процессора и характера моделируемой проблемы, и вполне возможно (и достаточно часто случается), что работа не будет масштабироваться так же эффективно с увеличением числа потоков, как с увеличением числа процессов MPI. Если доступна эффективная вычислительная система с несколькими полноценными узлами, то использование MPI может принести больший выигрыш во времени.

6. Метод Монте-Карло

В прикладных задачах физики конденсированных сред крайне часто возникает необходимость искать величины, характерные для реальной физической среды. С физической точки зрения таким величинам соответствуют термодинамические средние искомых величин по некоторому ансамблю систем. Такая трактовка задачи соответствует представлению о реальной системе как находящейся в термодинамическом равновесии, то есть, описывающей некоторую траекторию в фазовом пространстве. Математически, такая задача состоит в нахождении интеграла от искомой величины по этой траектории, причём величина берётся с некоторой весовой функцией. Далее к системе применяется *эргодическая гипотеза*, постулирующая возможность замены средних по динамической траектории системы средним по всему фазовому пространству. Вопрос о том, в каких

случаях эта гипотеза верна, а в каких – нет, изучается достаточно активно. Мы же будем считать её верной, поскольку пока, насколько нам известно, нет физических систем, где бы эта гипотеза не выполнялась.

Таким образом, задача сводится к нахождению интеграла определённого вида по фазовому пространству системы. Иногда её удаётся решить аналитически, однако обычно это возможно лишь для простых систем. Для более сложных имеет смысл применить численные методы, позволяющие найти интеграл с заданной точностью, но явное вычисление интеграла возможно далеко не всегда.

Так, с ростом количества степеней свободы сложность вычисления (число вычислительных операций) возрастает экспоненциально и, нахождение такого интеграла для системы со 100 степенями свободы превращается в крайне сложную задачу. А что делать, если количество степеней свободы составляет несколько миллионов?

В таких случаях часто возможно лишь найти некоторую оценку искомой величины. В этом вопросе на помощь приходят вероятностные алгоритмы, основанные на методе Монте-Карло. Идея последнего состоит в том, чтобы некоторым образом выбирать точки пространства и на основании знаний о поведении функции в них делать какие-либо выводы о глобальном поведении. Как применяется данный метод для поиска площади фигур, было показано в примере 3. В данном разделе мы рассмотрим более сложные задачи, для которых применим алгоритм Монте-Карло.

6.1. Алгоритм Метрополиса

Исходя из того, что мы рассматриваем физические задачи, можно несколько уточнить вид интеграла, значение которого представляет для нас интерес. Часто вероятность системы находиться в состоянии с заданной энергией $\boldsymbol{H}$ описывается распределением Гиббса и равна $\alpha e^{-\beta H}$, где α – нормировочная константа, а $\beta=1/k_BT$. Исходя из этого, интересующее нас термодинамическое среднее $<\boldsymbol{A}>$ некоторой величины $\boldsymbol{A}$ будет вычисляться по формуле

$$< A >= \alpha \int_{M^n} A e^{-\beta H} d^n x,$$

где интегрирование ведётся по n-мерному фазовому пространству M^n, причём n достаточно большое. Будем теперь, в соответствии с методом Монте-Карло, случайно выбирать точки и считать в них значения подынтегрального выражения, после чего можно оценить значение интеграла средним арифметическим. Поскольку данный алгоритм вероятностный, то получаемый ответ будет приближаться к реальному значению интеграла лишь в пределе бесконечного числа шагов. На практике, естественно, число точек далеко не бесконечно. Более того, уже для системы из 100 точек, расположенных на прямой (каждая точка имеет одну степень свободы), в этом вопросе возникают существенные проблемы: даже если рассматривать по два значения на каждую степень свободы, фазовое пространство будет состоять из 2^{200} точек. Даже за сто лет при счёте на всех компьютерах мира не удастся даже близко приблизиться к тому, чтобы учесть их все. Таким образом, сегодняшние компьютеры позволяют рассмотреть крайне малое число точек фазового пространства, а потому можно ожидать, что результат будет существенно зависеть от того, какие именно точки мы будем брать. В этом отношении, для того, чтобы получить ответ, хоть сколько-нибудь близкий к реальности, следует более внимательно отнестись к вопросу выбора точек фазового пространства. В литературе способ выбора точек называют *сэмплингом* (sampling).

Идея метода Метрополиса основана на особенности весовой функции рассматриваемых интегралов, $\alpha e^{-\beta H}$, которая убывает очень быстро при увеличении энергии. Так, если поверхность потенциальной энергии системы имеет вид достаточно глубокой

ямы в окрестности некоторой точки x_0 фазового пространства, а в других точках более высокие значения энергии, то вклад одной точки вблизи ямы может оказаться важнее всех остальных точек вне ямы. Точки с меньшим значением энергии в этом смысле «важнее» точек с большей энергией, а потому нам следует выбирать их чаще. Следующим важным наблюдением является тот факт, что если мы будем выбирать точки, в которых значение энергии равно $\boldsymbol{H}$ с вероятностью $\alpha e^{-\beta H}$, то для большого числа точек среднее арифметическое значений $\boldsymbol{A}$ в этих точках будет приближать искомый интеграл. Получаемое таким образом приближение, вообще говоря, лучше приближения, получаемого рассмотрением равномерной сетки на фазовом пространстве. Описанный метод часто даёт достаточно хорошее приближение термодинамических средних величин, хотя и не всегда. Следует помнить, что ответ, получаемый этим методом, приближается к верному лишь в пределе бесконечного числа точек. Поскольку в реальных приложениях количество точек крайне мало, вопросы сэмплинга играют крайне важную роль. В этом отношении, получение физически корректных результатов превращается в некоторое искусство. При этом получить гарантию того, что полученный ответ не только разумен, но и верен, получается далеко не всегда.

Перейдём теперь к описанию алгоритма Метрополиса. Как было отмечено выше, с целью получения достаточно хорошего приближения необходимо построить набор точек фазового пространства так, чтобы точки с энергией $\boldsymbol{H}$ встречались бы в этом наборе с вероятностью $\alpha e^{-\beta H}$. Тогда термодинамическое среднее будет вычисляться как среднее арифметическое от значений величины $\boldsymbol{A}$ в точках построенного набора. Один из вариантов построения такого набора состоит в построении *Марковской цепи*, или случайного блуждания, в фазовом пространстве системы. Марковским случайным процессом, или Марковской цепью, называют процесс, который, как говорят, «не имеет памяти». Это значит, что его значение в каждый дискретный момент времени *t* зависит только от предыдущего в момент времени *t-1* и никак не зависит от того, что было до этого шага. Для более глубокого ознакомления с вопросом можно изучить [24]. Другими словами, начиная из какой-то определённой точки, мы начинаем делать шаги в фазовом пространстве. Так, если мы находимся в точке с энергией $\boldsymbol{H_0}$, сначала случайным образом выбирается следующая, соседняя с первой, точка с энергией $\boldsymbol{H_1}$, а затем с вероятностью

$$\min\{e^{\beta[H_0-H_1]},1\}$$

происходит переход в неё (если энергия нового состояния меньше энергии старого, то переход происходит всегда). Если переход произошёл, то следующая точка выбирается из окрестности новой точки. В противном случае перехода не происходит, и процедура повторяется (рис. 14).

Будем далее обозначать через *x*[H,A] состояние *x* с энергией $\boldsymbol{H}$ и значением $\boldsymbol{A}$ величины, среднее от которой нас интересует. Тогда сформулированную выше идею алгоритма можно реализовать следующим образом:

```
выбирается начальное состояние x0[H0,A0];
номер шага          i = 0;
общее число шагов   n = N;
среднее значение   <A>= 0;
пока (i < n)
{
   выбирается пробное состояние t[Ht,At] в окрестности xi[Hi,Ai];
   генерируется случайное число p с равномерным распределением на отрезке [0;1];
   если (p < exp(βHi - βHt))     xi+1[Hi+1,Ai+1] = t[Ht,At],
   иначе                         xi+1[Hi+1,Ai+1] = xi[Hi,Ai];
   увеличиваем среднее           <A> = <A> + Ai+1/n;
   переходим к следующему шагу i++;
}
```

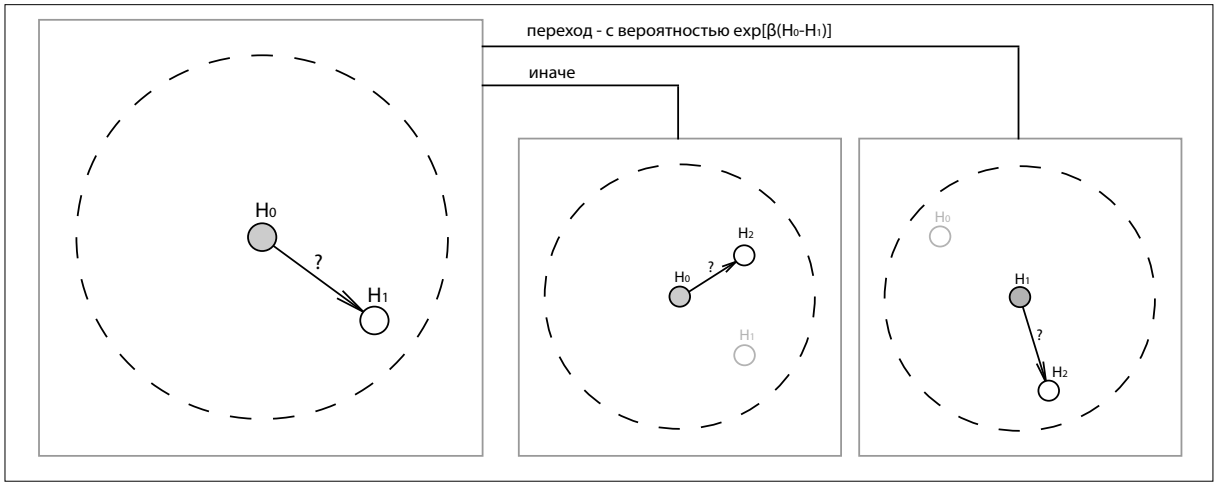

Рис. 14. Схема построения Марковской цепи

Отметим также, что иногда осреднение начинают не с первого шага, а по прошествии некоторого их количества. Такой приём применяется как мера предосторожности, если, например, начальное состояние имеет очень большую энергию. В этом случае оно может сильно повлиять на вычисление среднего значения, внеся нежелательные искажения.

6.1.1. Особенности построения цепи

В изложенной схеме есть один тонкий момент: какие состояния называть соседними? Другими словами, в какие точки мы сможем попасть из заданной за один шаг? Обычно правильный выбор такой окрестности неоднозначен, определяется исследователем и может значительно влиять на сходимость метода. Так, если выбрать окрестность достаточно маленькой, а начальную точку – вблизи локального минимума, полученное случайное блуждание сделает большое число шагов вблизи этого минимума. Полученный в результате набор точек даст информацию лишь об очень малой окрестности фазового пространства, а ответ не будет иметь отношения к реальному значению искомой величины. Напротив, выбор слишком большого шага может привести к большим прыжкам температуры между последовательными состояниями случайного блуждания. По этой причине будет приниматься лишь малая их часть, что также приведёт к плохой выборке точек для усреднения и замедлению сходимости. Принято считать, что отношение принятых шагов ко всем должно быть примерно 0.3 [24].

Выбор способа построения цепи Маркова зависит от задачи и подхода к реализации. Пожалуй, самым разумным способом определения оптимальности способа построения цепи является явная проверка. Для этого следует начинать блуждания из различных точек и делать разное количество шагов. Если получаемые в итоге результаты практически не зависят от начальной точки, и для их получения делается наименьшее по сравнению с другими методами число шагов, то такой способ построения можно назвать оптимальным. Для различных задач оптимальные параметры блуждания, вообще говоря, различны.

Существует критерий, который позволяет проверить корректность построенного блуждания. Рассмотрим два произвольных состояния x и y. Процесс называется *Марковским*, если для него выполнено условие *детального баланса*. Забудем на миг про вероятность перехода $\min\{e^{\beta[H_0-H_1]},1\}$, и будем считать, что после выбора новой точки фазового пространства мы всегда переходим в неё (вероятность равна 1). Тогда условие детального баланса заключается в том, что, что если на шаге совершается переход из x в

y, причём точка y выбирается среди остальных точек окрестности с вероятностью p, то на следующем шаге возможен обратный переход из y в x, причём вероятность выбора точки x тоже равна p. Таким образом, можно воспринимать это условие как обратимость конструируемого процесса. Условие детального баланса обычно гарантирует правильность выбора параметров блуждания. Обратное, вообще говоря, неверно: существуют некоторые примеры блужданий, для которых условие детального баланса (в явном виде) не выполняется. Однако оно может выполняться в несколько иной форме.

6.1.2. Пример задачи N тел

Следующий пример является важным, поскольку очень широкий класс задач сводится некоторым образом к рассматриваемой системе. Рассмотрим в качестве примера газ из N частиц, одинаковых материальных точек массы m, попарно взаимодействующих между собой, причём потенциал этого взаимодействия U зависит только от расстояния между этими точками. Кроме того, пусть эти точки находятся в кубическом ящике фиксированного объёма и не выходят за его границы. Наконец, зафиксируем температуру T и будем считать её известной (другими словами, нам известна β). По определению температуры как средней кинетической энергии молекул можно считать, что нам известна кинетическая энергия всего газа. Применим метод Монте-Карло для поиска термодинамических средних данной системы. Тогда для сформулированной задачи интеграл

$$< A >= \alpha \int_{M^n} A e^{-\beta H} d^n x$$

перепишется в виде

$$< A >= \alpha \int_{\mathbf{R}^{6N}} A \exp\left[-\beta \left(\sum_{i=1}^{N} \frac{p_i^{\,2}}{2m} + \sum_{0<i<j<N+1} U(r_{ij}) \right) \right] d^{3N} x d^{3N} p$$

Но поскольку нам интересно лишь среднее по каноническому NVT-ансамблю, вместо этого интеграла нас интересует интеграл по подпространству, определяемому условием

$$\sum_{i=1}^{N} \frac{p_i^{\,2}}{2m} = NkT$$

Тогда искомый интеграл будет иметь вид

$$< A >= \alpha_1 \int_{L^{3N}} A \exp\left[- \sum_{0<i<j<N+1} U(r_{ij}) \right] d^{3N} x$$

где L – ребро кубического ящика, а $\alpha_1 = \alpha e^{-N}$. Для подсчёта этого интеграла уже можно явно применять описанный ранее метод для получения значений термодинамических средних.

Первым шагом является выбор начального распределения. С эти обычно проблем не возникает: используя генератор случайных чисел можно без труда получать различные начальные состояния, то есть, координаты всех частиц.

Следующий шаг – выбор пробного состояния – уже зависит от способа построения цепи Маркова. Возникает вопрос, как, имея конфигурацию N точек, следует строить новую конфигурацию, лежащую в фазовом пространстве в некоторой окрестности исходной. Ответ на этот вопрос неоднозначен и зависит от нескольких факторов.

Самым первым, что приходит в голову, является небольшое смещение всех частиц на случайное расстояние не больше некоторого параметра ε. Если ε мало, новая конфигурация будет слабо отличаться от исходной. С увеличением параметра это отличие будет возрастать, пока, наконец, мы не придём к выбору произвольной точки фазового пространства в качестве шага.

Вторым методом является такое построение случайного блуждания, при котором следующая конфигурация получается из исходной изменением положения лишь одной из частиц. С точки зрения перемещения по фазовому пространству данный метод менее эффективен, чем первый (на один шаг первого метода приходится минимум N шагов второго), однако у него есть и свои преимущества, главное из которых состоит в следующем. Поскольку при построении цепи нам необходимо знать разность энергий $\boldsymbol{H_0} - \boldsymbol{H_1}$, на каждом шаге подразумевается вычисление суммы

$$\sum_{0<i<j<N+1} U(r_{ij})$$

Сложность вычисления этого выражения с ростом N возрастает как N^2, что существенно замедляет расчёты для больших N. При смещении всех частиц на каждом шаге необходимо заново пересчитывать всю энергию, в то время как второй способ — смещение лишь одной частицы, помогает избежать этой проблемы. В этом случае для того, чтобы знать разность энергий достаточно посчитать изменение энергии, обусловленное смещением одной частицы, на что требуется порядка N операций. Таким образом, второй метод часто оказывается практичнее с точки зрения скорости вычислений, в то время как в случае простых функций энергии можно использовать любой метод. Кроме того, сдвиг лишь одной частицы на шаге также значительно уменьшает и количество передаваемых по сети данных, что также может сыграть решающую роль при параллельной реализации. По этой причине в большинстве случаев используется именно этот метод.

С точки зрения программной реализации первый метод вполне ясен: используя генератор случайных чисел можно получить случайные сдвиги частиц. При этом следует всегда помнить о детальном балансе. Приведём простой пример: разрешим частице сдвигаться на векторы из диапазона [0;1]x[0;1]x[0;1]. В этом случае нарушается условие детального баланса, частица не может с той же вероятностью попасть на следующем шаге в предыдущее положение. Более того, понятна и возникающая в этой ситуации ошибка: разрешая всем частицам двигаться лишь в положительном направлении по всем трём осям, мы добавляем некую силу, препятствующую их движению в обратном направлении. Понятно, что полученный результат будет неверным в силу неправильного построения цепи.

Реализация второго метода построения блуждания, при котором за шаг сдвигается лишь одна частица, отличается от первого лишь предварительным выбором частицы сдвига. Этот выбор обычно осуществляется случайным образом, хотя возможны и другие варианты, [24], например, поочерёдный выбор частиц {1, 2, 3, ..., N, 1, 2,...}.

6.1.3. Отжиг

Следующим примером использования алгоритма Метрополиса является его приложение к задаче поиска локального минимума, [30]. Пусть имеется функция $f(x_1, x_2, x_3, ..., x_N)$, для которой необходимо найти глобальный минимум, то есть, набор параметров, при котором данная функция принимает наименьшее значение. Возникает вопрос, почему бы не воспользоваться другими методами поиска минимумов (градиентный спуск, метод сопряжённых градиентов, [16]). Ответ весьма прост, все эти методы, во-первых, находят лишь локальный минимум, а, во-вторых, используются тогда, когда подсчёт функции происходит достаточно быстро. Напротив, поиск глобального минимума функции многих переменных является существенно более сложной задачей. Еще больше проблема усугубляется, если каждый подсчёт функции занимает значительное время.

Для решения таких задач часто применяют вероятностные алгоритмы: метод отжига или так называемые генетические алгоритмы. Остановимся подробнее на первом из них, в то время как со втором классом алгоритмов можно познакомиться по книге [31].

Суть метода отжига состоит в построении Марковской цепи в пространстве параметров задачи $(x_1, x_2, x_3, ..., x_N)$, где роль энергии играет функция $f(x_1, x_2, x_3, ..., x_N)$. Температура в данном представлении просто является некоторым параметром, у которого, нет физического смысла (как, впрочем, и у самой функции f). Далее, после того, как блуждание совершит некоторое количество шагов, температура понижается, и блуждание вновь продолжается. Когда температура достигает нуля, система может оказаться в состоянии, соответствующем глобальному минимуму. В данном алгоритме всё зависит от выбранного набора параметров, что является предметом отдельного обсуждения. Более подробную информацию читатель может найти, например, в статье [30].

Приведём интересный пример использования данного метода в молекулярном моделировании. Рассмотрим две достаточно крупные молекулы (например, белки). После этого мы хотим решить задачу *докинга* (*docking*), то есть, найти такое относительное положение молекул, при котором их энергия была бы минимальной. Иногда накладываются дополнительные условия, например, вклад площади соприкосновения белков в качестве учёта гидрофобных взаимодействий. Принято считать, что в ряде задач такие положения имеет важный биологический смысл. При этом белки считаются твёрдыми телами, а потому имеется не так много параметров: по три координаты угла поворота для каждой молекулы, итого 12 параметров. Основная сложность здесь заключается в том, что зависимость потенциала взаимодействия от этих параметров имеет весьма сложный вид. Кроме того, часто неизвестна точная конформация белков, откуда возникает необходимость рассмотрения случая возможной деформации белков во время докинга, что может добавить ещё несколько степеней свободы. В итоге задача поиска минимума становится достаточно сложной и иногда её решают именно методом отжига.

В заключение хотелось бы ещё раз подчеркнуть тот факт, что выбор способа построения цепи Маркова в различных задачах является специфичным по отношению к задаче. Кроме того, вычислительная система, на которой производится вычисление, может сыграть важную роль в этом вопросе. Так, в работе [32] применяется очень интересная схема построения Марковской цепи, которая гармонично вписывается в схему работы SIMD-процессора. В этом отношении, подбор необходимых параметров для конкретных задач сам по себе является некоторой задачей, которую необходимо решить, прежде чем приступить к исследованию первоначальной проблемы.

6.2. Параллельная реализация

Как было отмечено выше, алгоритмы Монте-Карло используются для того, чтобы, исходя из некоторого набора точек пространства, извлечь некоторую информацию об интегралах по всему пространству. При этом число точек, содержащееся в таких наборах, ничтожно мало, а потому сама возможность получить осмысленный результат кажется некоторым чудом. По этой причине, результаты, полученные подобными методами, следует проверять очень тщательно, по возможности сверяясь с полученными ранее данными.

Казалось бы, имеет смысл отбросить эти методы, коль скоро они не дают никаких гарантий. Но проблема заключается в том, что других методов исследования некоторых задач просто нет, и применение стохастических методов – это попытка сделать хоть что-то, и надеяться получить разумный результат. Причина подобного бессилия, с одной стороны, состоит в отсутствии других плодотворных методов исследования, и, с другой, в слабости вычислительной техники.

Первую проблему применимости стохастических методов следует решать в каждом конкретном случае по-своему, в то время как решение второй, технической, сразу поможет в решении достаточно широкого круга задач. Одной из попыток решения последней является использование параллельных реализаций программ, позволяющих в разы (а для суперкомпьютеров и в тысячи раз) ускорить вычисления.

6.2.1. Варианты распараллеливания

Поскольку нас в данном разделе интересует лишь метод Монте-Карло (алгоритм Метрополиса), рассмотрим, как вообще можно произвести распараллеливание подобной программы на примере задачи N точек, сформулированной выше. Алгоритм состоит из следующих фрагментов:

1. Генерация состояния на следующем шаге. В случае если следующая конфигурация получается сдвигом всех частиц сразу, потребуется порядка N операций. Если за шаг сдвигается только одна частица, то число операций постоянно.
2. Вычисление энергии системы. В случае если на шаге смещаются все частицы, необходимо сделать N^2 операций. Во втором случае – порядка N.
3. Сравнение энергий.
4. Принятие/непринятие новой конфигурации.

Из анализа видно, что если что-то и стоит распараллеливать в данной задаче, так это вычисление энергии и, при желании, формирование нового состояния, что встречается достаточно редко. Распараллеливание энергии, заданной в виде

$$\sum_{0<i<j<N+1} U(r_{ij})$$

можно осуществлять несколькими способами. Рассмотрим некоторые из них.

Первый способ можно назвать примитивным и, с другой стороны, самым простым с точки зрения реализации. Этот способ можно применять, если уже имеется рабочая последовательная (не распараллеленная) программа. Так, в самой простой реализации подсчёт энергии обычно реализован подобным кодом

```
...
U = 0;
for (int i = 0; i<N; ++i)
{
  for (int j = i+1; j<N; ++j)
  {
    U += PairEnergy(Mol[i],Mol[j]);
  }
}
...
```

Здесь функция `PairEnergy` реализует вычисление энергии между частицами i и j, которые в данном коде мы представили элементами массива частиц `Mol[N]`, и полученные результаты суммируются в переменной `U`. Итак, в простейшем случае MPI-реализации этот кусок кода перепишется в виде

```
class Conformation {
public:
   Particle* Mol;              // Массив частиц (координаты)
   double    U;                // Потенциальная энергия
   . . .                       // Прочие параметры

void CalculateEnergy();
};

class System {
public:
   Conformation   curr,        // Текущая конфигурация
                  next;        // Пробная конфигурация системы
```

```
   . . .                          // Прочие параметры
};


     // Добавляем в main() инициализацию MPI
...
int MyRank, Size;
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
MPI_Comm_size(MPI_COMM_WORLD, &Size);
...
     // Модифицируем вычисление энергии
void Conformation :: CalculateEnergy()
{
   U     = 0;
   Upart = 0;
   for (int i = N*MyRank/Size; i < N*(MyRank+1)/Size; ++i)
   {
      for (int j = i+1; j<N; ++j)
      {
          Upart += PairEnergy(Mol[i],Mol[j]);
      }
   }
   MPI_Reduce(&Upart, &U, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Представленная реализация имеет ряд преимуществ и недостатков. В качестве первого очевидного плюса является простота реализации: подобные преобразования достаточно просты и обычно не требуют значительной модификации программы. Вторым плюсом является минимальное количество передаваемых данных, что может оказаться критичным при моделировании больших систем, поскольку передачи данных, например, по обыкновенной локальной 100Mbit/s сети занимают сравнительно много времени, а потому их рекомендуется избегать.

Основным недостатком данного вида распараллеливания является малая гибкость: такая реализация может быть использована лишь для простых циклов. В случае более сложных изменений такое распараллеливание, вероятно, окажется непригодным. Вторым недостатком является некоторый «пустой» расход вычислительной мощи. Дело даже не в том, что здесь процессы вычисляют парную энергию разное количество раз. Этот недостаток достаточно легко устранить, переписав циклы в несколько другой форме, чтобы более равномерно перераспределить нагрузку. Более того, если Марковская цепь строится путём смещения только одной частицы, такой проблемы не возникает. В таком случае необходимо считать лишь энергию взаимодействия одной частицы с остальными, ввиду чего внутренний цикл исчезает. Проблема же состоит в том, что в данной реализации все процессы выполняют одну и ту же исходную последовательную задачу. Это означает, что все они получают на каждом шаге одно и то же, за исключением места, где вычисляется энергия. В этом смысле, вместо какого-то числа M операций всеми процессорами выполняется M*`Size`, причём M – достаточно большое число.

Данные рассуждения подводят нас ко второй идее параллелизации, которая состоит в следующем. Выделяется один процесс, называемый главным, или рутом (*root*), который и проводит все вычисления, связанные с построением цепи Маркова. При этом вся работа по подсчёту энергий и прочих величин возлагается на остальные узлы. Основным преимуществом данного метода является то, что никакая работа не выполняется дважды.

Не менее важно и то преимущество, что данный подход более гибкий и позволяет проще применять возникающие по ходу реализации идеи оптимизаций. Более того, сама

реализация в этом случае далеко не однозначна и определяется, в основном, предпочтениями автора. В качестве примера приведём схему варианта подобной реализации.

```
...
     // основная функция
if (MyRank == ROOT)   // Главный процесс
{
   Conformation curr,       // Текущая конфигурация
                next,       // Шаг цепи Маркова - пробная конфигурация
                            // ещё не принята)
                trycurr,    // Конфигурация в окрестности текущей (try current)
                trynext;    // Конфигурация в окрестности пробной (try next)

   curr.Init();              // Задаём начальную конфигурацию
   curr.CalculateEnergy();   // Вычисляем начальную энергию

   next = GenerateNextStep(curr); // Генерируем пробную конфигурацию около curr

   for (int step=0; step<MaxSteps; ++step)
   {
       StartEnergyCount(next); // Отправляем считаться энергию пробного состояния
                               // next
       CountAverages(curr);              // вычисляем промежуточные средние
       ...                               // возможно, что-то ещё
       trycurr = GenerateNextStep(curr); // генерируем два состояния на следующий
       trynext = GenerateNextStep(next); // шаг за next - выберем нужный позже
       EndEnergyCount(next);   // Забираем посчитанную энергию пробного состояния
                               // next

       if (exp((curr.U - next.U)/kT) > ((double)rand())/RAND_MAX)
       {                       // Шаг принят
          curr = next;
          next = trynext;
       }
       else                    // Шаг не принят
       {
          next = trycurr;
       }
   }
}
else
{      // остальные процессы лишь вычисляют энергию
   for (int step=0; step<MaxSteps; ++step)
   {
      GetData();
      CalculatePartEnergy();
      ReturnData();
   }
}
```

Здесь `ROOT` – определённая константа, соответствующая главному процессу. Класс представляет объект частицы со всеми необходимыми переменными, такими, как, например, координаты или энергия. Функция `Init(curr)` заполняет значениями (например, случайными) координаты конфигурации *curr*. Вызов `GenerateNextStep(curr)` возвращает в *next* новую конфигурацию, случайно выбранную из окрестности аргумента. Вычисление энергии состояний осуществляется вызовом последовательным двух функций. Так, `StartEnergyCount(next)` осуществляет отправку аргумента на остальные процессы, для которых значение *MyRank* отлично от `ROOT`, после чего эти процессы начинают вычисление энергии. Последние, соответственно, сначала получают передаваемую им информацию о системе функцией `GetData()`, после чего начинают вычислять свою часть энергии, определяемую в функции `CalculatePartEnergy()`. В это время ещё неизвестно, будет ли принята конфигурация *next*. Главный процесс сво-

боден, а потому мы можем заставить его сперва вычислить средние значения на текущий момент по формуле

$$\langle A \rangle_{n+1} = n\langle A \rangle_n + A_{n+1},$$

а затем сгенерировать следующие состояния «на все случаи жизни», то есть, одно (`trynext`) на случай, если конфигурация `next` будет принята, и одно (`trycurr`) на случай, если этого не произойдёт, и процесс Маркова останется в точке `curr`. Эти конфигурации выбираются, соответственно, в окрестностях `next` и `curr`. Помимо вычисления средних и следующих состояний, здесь также могут находиться алгоритмы термостатов и баростатов (см. ниже). После этого главный процесс вызывает `EndEnergyCount()`, сообщая остальным процессам о готовности получить посчитанные результаты. Остальные процессы отсылают необходимую информацию функцией `ReturnData()`. В результате мы получаем согласованную работу узлов, где построение цепи Маркова и вычисление энергии происходит параллельно.

Существенным недостатком данного метода является значительные передачи данных по сравнению с первым методом. Впрочем, если передача идёт через быстрые интерфейсы (InfiniBand, например), то эта проблема не столь существенна. В качестве альтернативного решения этой проблемы может быть предложено применение асинхронных передач данных (помимо более простых оптимизаций объёма передач, исходя из специфики задачи). По этой причине можно сказать, что другой сложностью реализации этого варианта является необходимость существенного изменения программы и углублённого изучения MPI.

Приведённые примеры представляют собой возможные варианты осуществления параллельных вычислений энергии для задачи N тел. В зависимости от сложности задачи схема распараллеливания может быть изменена. Так, например, если моделируется ансамбль ионов, энергия взаимодействия между которыми описывается комбинацией электростатических сил и потенциала твёрдых сфер (равен бесконечности, если радиусы частиц перекрываются, и нулю в противном случае), может оказаться разумным разделённое вычисление этих потенциалов: к примеру, три процесса для электростатики и только один для потенциала твёрдых сфер.

Чем сложнее задача, тем более изощрённой может оказаться параллельная реализация.

6.2.2. Генераторы случайных чисел

Крайне важным элементом случайных алгоритмов является генератор случайных чисел. Применимость всех стохастических методов основывается на том, что в нашем распоряжении имеется *идеальный* генератор случайных чисел, но при работе с реальными вычислительными системами это, к сожалению, не так. Числа, которые получаются при помощи некоторых численных методов, часто называют генераторами *псевдослучайных* чисел. Подобная псевдо случайность является следствием двух фактов. Во-первых, все численные генераторы имеют некоторый период, то есть, начинают в точности дублировать себя, начиная с некоторого момента. Принято считать, что за время работы программы число генераций не должно превышать периода, в противном случае может произойти ошибка (впрочем, может и не произойти, но лучше не рисковать). Во-вторых, у генераторов есть стартовая точка (*seed*), которая однозначно определяет генерируемую последовательность. Если в программе эта точка задана явно, то последовательность будет одна и та же, а программа, несмотря на алгоритм с использованием случайных чисел, будет работать идентично при каждом запуске. Наконец, в ряде случаев реальное распределение генератора часто оказывается не совсем таким, как требовалось разработчику для написания решения.

Для проверки «степени идеальности» генераторов сегодня существует множество программ и библиотек, позволяющих определить распределение чисел генерируемых заданным алгоритмом.

В C++ программист может воспользоваться рядом готовых генераторов случайных чисел, их реализация может зависеть от компилятора. Функции распределения всех генераторов подразумеваются равными константе на области определения. Так, в Windows-компиляторах (*Borland C++*, компиляторы *Microsoft Visual Studio*) обычно используется функция `rand()`, которая генерирует псевдослучайное целое число от 0 до `RAND_MAX(32767)`. Эта же функция поддерживается и linux-компиляторами *gcc*.

Для большинства версий компиляторов функция `rand()` реализует генератор с периодом 2^{32}=4 294 967 296. Для большого числа приложений такого должно хватать. Функция `srand()` имеет один параметр, определяющий стартовую точку генерируемой последовательности.

Помимо `rand()` компилятор *gcc* имеет дополнительные возможности генерации случайных чисел:

`drand48()` и `erand48()` возвращают значения типа `double` из промежутка [0.0,1.0);
`jrand48()` и `mrand48()` возвращают значения типа `long` из промежутка $[-2^{31},2^{31})$;
`lrand48()` и `nrand48()` возвращают значения типа `long` из промежутка $[0,2^{31})$.
Функции `seed48()` и `srand48()` определяют отправную точку этих генераторов.

Идея одного из методов генерации псевдослучайных чисел, которая используется в перечисленных функциях, состоит в рекуррентном вычислении последовательности $X_{n+1}=[a*X_n+b] \bmod 2^{48}$ для заданных значений a, b и X_0. Периоды этих генераторов составляют 2^{48}=281 474 976 710 656, и такого числа должно хватить для моделирования любой физической системы. Подробнее об алгоритмах генерации псевдослучайных чисел можно прочитать, например, в [33].

В случае обычной последовательной программы никаких проблем в использовании перечисленных функций обычно не возникает. Однако, имея дело с параллельной реализацией, следует всегда помнить о том, что именно ожидается от генератора случайных чисел. Если желаемый результат – получить одну последовательность на всех узлах, то нужно, например, задать единую отправную точку и использовать одну и ту же функцию для всех процессов. Если же каждый процесс должен иметь свою индивидуальную последовательность псевдослучайных чисел, то следует, например, вызвать на каждом процессе функцию `srand()`, передав ей в качестве аргумента ранг этого процесса.

6.3. Улучшения сэмплинга

Применение схемы Метрополиса даёт верные значения средних лишь в бесконечном пределе числа шагов. В этом смысле можно говорить, что средние величины, посчитанные для первых *S* шагов, сходятся к некоторой величине при неограниченном росте *S*. В ряде случаев такая сходимость происходит крайне медленно. В моделировании мы не можем ждать бесконечно долго, поэтому необходимы приёмы, позволяющие ускорить сходимость метода Монте-Карло в указанном выше смысле. Большинство существующих схем, улучшающих сходимость, основано на улучшении выборки точек фазового пространства. Более подробную информацию о предложенных ниже схемах можно найти в [24].

6.3.1. Расчёт нескольких систем

Как было отмечено выше, проблема сэмплинга в ряде случаев стоит крайне остро. Так, если система во время расчёта попала в локальный минимум потенциальной энергии, она может очень долго из него не выбираться, что приведёт к некорректному результату.

Однако, выражаясь достаточно грубо, правильным является учёт всех локальных минимумов сразу, или хотя бы значительной их части. По этой причине, "плохие" задачи для метода Монте-Карло, то есть, такие случаи, когда рельеф поверхности потенциальной энергии очень неровный и имеет большое число локальных минимумов, крайне сложно изучать обычной схемой Метрополиса (хотя такие случаи встречаются довольно часто). В таких случаях необходимо применять некоторые хитрости, которые позволяют учитывать большое количество точек фазового пространства.

Одним из таких методов является параллельный запуск сразу нескольких случайных блужданий из различных (и достаточно далёких) точек пространства. Такой подход обеспечивает возможность получения достаточно широкого класса принципиально отличных друг от друга конформаций, поскольку, скорее всего, все полученные траектории не будут ограничиваться фиксированной областью фазового пространства, как это может произойти в случае одного блуждания.

С точки реализации подобная схема может быть реализована как в рамках MPI, так и без его помощи. Необходимо лишь построить некоторое число Марковских цепей, начинающихся в различных точках, после чего усреднить результаты, взяв в качестве средней величины среднее арифметическое результатов, полученных в частных случаях. В этом смысле, применение MPI имеет то преимущество, что такие осреднения не придётся вести вручную, можно воспользоваться возможностями MPI. Преимуществом многократного запуска одной последовательной программы является то, то в случае отказа одного из вычислительных узлов будет потеряна лишь часть результатов, в то время как в MPI ситуация отказа одного из узлов автоматически приведет к потере информации со всех процессов.

Основным недостатком данного метода является независимость построенных блужданий. Так, если энергия одного из блужданий существенно больше остальных, и при этом, оно находится в окрестности минимума, осреднение такой цепи "на равных" с остальными, где энергия меньше, может привести к неверным результатам. Этот недостаток можно исправить, слегка изменив свойства моделируемых систем.

6.3.2. Parallel tempering

Достаточно новым методом сэмплинга алгоритма Метрополиса является применение схемы, в зарубежной литературе называемой Parallel Tempering. Он крайне хорошо зарекомендовал себя в моделировании систем с большим числом локальных минимумов. Идея метода состоит в следующем. Как и в предыдущем случае, происходит одновременное моделирование сразу нескольких систем, и на основании их работы делаются выводы о значениях средних величин, но отличие состоит в том, что в данном случае моделирование происходит при *различных* наборах параметров. Это может быть как температура, так и объём или число частиц.

Итак, предположим, что нас интересует поведение системы при температуре T. Будем параллельно моделировать K систем при температурах $T_1, T_2, \ldots, T_K$, где $T = T_1$ (то есть, остальные системы разогреты сильнее интересующей нас температуры). Различие температур не позволяет провести осреднение результатов впрямую, но вместо этого мы разрешим соседним по температуре системам меняться конфигурациями (точками фазового пространства, в которых они находятся в данный момент) с вероятностью

$$\min\{e^{(H_i-H_j)(\beta_i-\beta_j)},1\},$$

где, как и ранее, H_i и H_j - энергии состояний систем при T_i и T_j соответственно, а параметры $\beta_i = 1/kT_i$, $\beta_j = 1/kT_j$. Ещё раз подчеркнём, что обычно обмен рассматривают лишь между соседними температурами, то есть, когда $i = j+1$ или $i = j-1$.

При этом при вычислении осреднение происходит лишь по конфигурациям системы с температурой T, а остальные не вносят явно никакого вклада. Однако если число систем достаточно большое, то разумно вычислять средние сразу для нескольких температур $T_1, \ldots T_{K-M}$ для некоторого M. Считается, что при $M > 1$ получается достаточно хорошая сходимость.

Преимуществом подобной схемы по сравнению с описанными ранее является то, что построенный таким образом сэмплинг умеет измерять "глубину" локальных минимумов. В качестве измерителя служит температура, при которой блуждание может из этой ямы выбраться. Кроме того, система, в которой происходит осреднение (то есть, при температуре T), не застрянет в окрестности локального минимума с большим значением энергии, поскольку другие блуждания, обнаружив положение с меньшей энергией, "перекинут" усредняемую систему в ту точку, а застрявшая система начнёт постепенно нагреваться. Взятием в качестве максимальной температуры достаточно большого T_K, можно гарантировать подъём из любой ямы, созданной локальным минимумом.

Таким образом, описанный алгоритм имеет преимущество предыдущей системы и притом лишена её недостатка: происходит построение сразу нескольких блужданий, в различных местах фазового пространства, но теперь невозможно застревание в окрестности какого-либо из локальных минимумов с достаточно большой энергией. В результате, можно ожидать, что построенная марковская цепь значительно улучшит сходимость средних величин по ансамблю.

Отметим, что подобные схемы можно применять и при моделировании динамики.

Что касается реализации данного типа сэмплинга, для сложных систем без MPI тут не обойтись. Здесь возможны две реализации, которые мы проиллюстрируем ниже на примере уже знакомой нам системы N попарно взаимодействующих частиц.

ВАРИАНТ 1

Этот вариант наиболее прост в осуществлении, и предназначен для небольшого числа процессов. В разделе 6.2.1 первым был рассмотрен вариант распараллеливания вычисления энергии, при котором все процессы знают всё о вычисляемой системе. Как было отмечено, в этом случае распараллеливание осуществляется достаточно просто. Сделаем следующий шаг, который позволит учесть наличие нескольких систем. Для этого можно создать новый класс *Tempering*, объединяющий K экземпляров класса моделируемой системы *System*. При этом существенное изменение претерпит лишь функция шага.

```
class Tempering {
public:
  int      SystemsNumber;   // число систем, моделируемых параллельно
  System*  Syst;            // массив объектов систем
  double*  Temperatures;    // массив температур (необязателен)
  int*     index;           // вектор номеров систем в порядке возрастания
                            // температуры
  . . .                     // прочие параметры
};
```

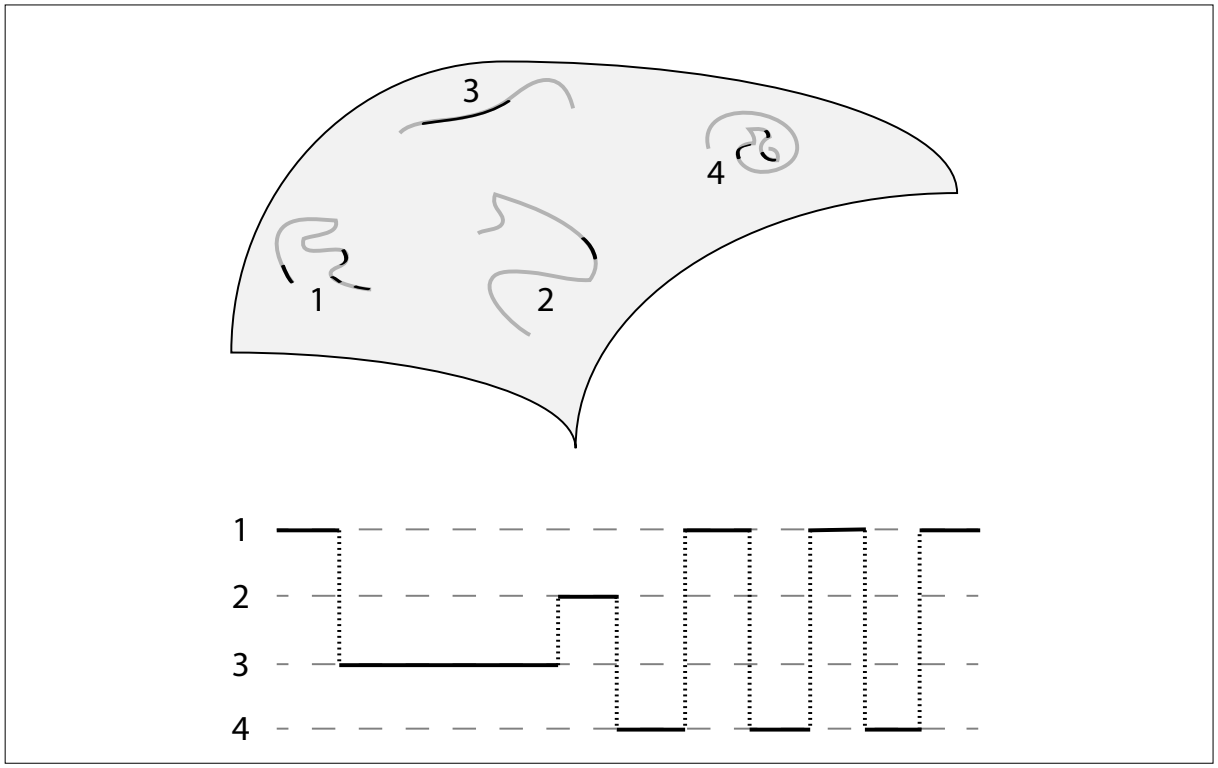

Рис. 15. Иллюстрация работы Parallel Tempering. На фазовом пространстве серыми кривыми показаны блуждания четырёх систем. Чёрные участки кривых обозначают состояния истинной марковской цепи, по которой ведётся осреднение.

```
 // Процедура выполнения следующего шага будет иметь вид
void Tempering :: MakeStep()
{
  double p;
  int    q;
  for (int k = 0; k < SystemsNumber; ++k)
  {
      Syst[k].next = GenerateNextStep(Syst[k].curr);
      Syst[k].next.CalculateEnergy();
      p = ((double)rand())/RAND_MAX;
      if (exp((Syst[k].curr.U - Syst[k].next.U)/Syst[k].kT) > p)
      {
          Syst[k].curr = Syst[k].next;
      }
  }

 // Прыжки между системами
  for (int k = 0; k < SystemsNumber-1; ++k)
  {
      p = ((double)rand())/RAND_MAX;
      if (exp((Syst[index[k]].curr.U-Syst[index[k+1]].curr.U)*
                                          (1/Syst[k].kT-1/Syst[k+1].kT)) > p)
      {   // меняем температуры
          p            = Syst[k].kT;
          Syst[k].kT   = Syst[k+1].kT;
          Syst[k+1].kT = p;
          // меняем индексы
          q            = index[k];
          index[k]     = index[k+1];
          index[k+1]   = q;
      }
  }
}
```

Таким образом, основное изменение функции шага проявится лишь в том, что, в дополнение к случайным блужданиям внутри систем, здесь появилась возможность обмена конфигурациями между системами. На уровне реализации практичней менять местами температуры двух систем, нежели их конфигурации. При этом необходимо запомнить, в каком порядке идут системы по возрастанию температур, чтобы не запутаться при осреднении.

Как видно из приведённой нами функции, вычисление энергии происходит последовательно для каждой из систем. Предположим теперь, что в нашем распоряжении имеется 1000 процессоров, а интересующая система имеет всего 1000 частиц. Запуск описанной схемы на всех процессорах не принесёт значительной пользы по сравнению с 10 процессорами. В этом случае имеет смысл либо запустить много копий программы для широкого диапазона данных, либо переписать программу так, чтобы все энергии вычислялись параллельно и, главное, независимо друг от друга.

ВАРИАНТ 2

Второй вариант распараллеливания, хоть и более трудоёмкий, но и более универсальный, в том смысле, что может быть применён для большего количества различных вычислительных систем. По своей идеологии этот способ распараллеливания близок к примеру раздела 6.2.1. Основная мысль такой реализации состоит в выделении одного управляющего процесса, который будет распределять вычислительную нагрузку между узлами и следить выполнением работы.

Схема построения цепи Маркова при такой реализации будет выглядеть следующим образом

```
...
  if (MyRank == ROOT)                    // Главный процесс
  {
     Conformation
        trycurr[SystemsNumber],
        trynext[SystemsNumber];

     for (int i=0; i<SystemsNumber; ++i)
     {
        Tempr.Syst[i].curr.Init();
        Tempr.Syst[i].curr.CalculateEnergy();
        Tempr.Syst[i].next = GenerateNextStep(Tempr.Syst[i].curr);
     }
     for (int step=0; step<MaxSteps; ++step)
     {
           StartEnergyCount(Tempr);
           CountAverages(Tempr);
           ...
           for (int i = 0; i < SystemsNumber; ++i)
           {
             trycurr[i] = GenerateNextStep(Tempr.Syst[i].curr);
             trynext[i] = GenerateNextStep(Tempr.Syst[i].next);
           }
           EndEnergyCount(Tempr);
           for (int i = 0; i < SystemsNumber; ++i)
           {
             p = ((double)rand())/RAND_MAX;
            if (exp((Tempr.Syst[i].curr.U-Tempr.Syst[i].next.U)/Tempr.Syst[i].kT)) > p)
             {
               Tempr.Syst[i].curr = Tempr.Syst[i].next;
               Tempr.Syst[i].next = trynext[i];
             }
            else
             {
               Tempr.Syst[i].next = trycurr[i];
```

```
            }
          }

          /* Прыжки между системами
          ...
          */
      }
  }
  else
  {
     for (int step=0; step<MaxSteps; ++step)
     {
       GetData();
       CalculatePartEnergy();
       ReturnData();
     }
  }
...
```

При этом функции передачи будут иметь вид (напомним, что переменная *Size* содержит количество процессов в коммуникаторе MPI_COMM_WORLD)

```
void StartEnergyCount(Tempering arg)
{
   int threads_per_sys = (Size-1)/arg.SystemsNumber;  // процессов на систему
   for (int j=0; j<arg.SystemsNumber; ++j)
   {
      for (int i=j*threads_per_sys; i<(j+1)*threads_per_sys; ++i)
      {
          MPI_Send(&arg.Syst[j],size,type,i,tag,comm); // раздать работу
       }
   }
}

void EndEnergyCount(Tempering arg)
{
   int     threads_per_sys = (Size-1)/SystemsNumber;  // процессов на систему
   double  temp;
   for (int j=0; j<SystemsNumber; ++j)
   {
      arg.Syst[j].curr.Energy = 0;
      for (int i=j*threads_per_sys; i<(j+1)*threads_per_sys; ++i)
      {  // сбор результатов
          MPI_Recv(&temp,1,MPI_DOUBLE,i,tag,comm);
          arg.Syst[j].curr.Energy += temp;
      }
   }
}
```

Рассылка данных, производимая первой функцией, зависит от способа реализации конкретной задачи. Так, в нашем коде мы подразумеваем раздачу всего объекта ряду процессов. Это можно сделать и более красиво, используя коммуникаторы и функцию broadcast. Так, следовало бы сначала разделить все процессы без одного (главного) на число групп, равное числу моделируемых систем, например, функцией *split* (под группой мы подразумевает коммуникатор). После этого к каждому из коммуникаторов присоединяется главный процесс, для того, чтобы он мог меняться сообщениями со всеми. Такая организация коммуникаторов поможет сделать передачи намного проще и понятнее с точки зрения реализации кода.

6.4. Граничные условия

Как и в случае молекулярной динамики, при моделировании ансамблей частиц возникает проблема выбора поведения системы на границе. Эта проблема обычно решается одним из двух способов: либо на границе ставится жёсткая стенка, либо вводятся периодические граничные условия.

Первый вариант имеет преимущество простоты в реализации и вычислениях. Однако в этом случае возникают граничные эффекты, поскольку на границах ячейки моделирования частицы имеют соседей лишь с одной стороны, что существенно отличает их от молекул в объёме. В результате на границах могут образоваться, например, более плотные структуры, которые влияют и на соседние молекулы. В результате влияние границ может распространиться на заметные расстояния, оставляя лишь малое количество частиц, на которые граничные эффекты не влияют. С точки зрения физики, именно последние представляют наибольший интерес. Поэтому, для их моделирования необходимо брать ячейку значительно большего размера, что при фиксированной плотности влечёт и значительное увеличение N. Естественно, возрастает и объём вычислений. Такой вариант границы удобен с точки зрения изучения граничных эффектов, но часто неприменим к исследованию свойств частиц в объёме.

Второй, более распространённый, вариант – это использование периодических граничных условий (pbc, periodic boundary condition). Он служит как раз для того, чтобы избежать граничных эффектов достаточно простым способом. Его суть заключается в том, чтобы проводить моделирование на трёхмерном торе. Говоря иначе, предполагается, что всё пространство состоит из одинаковых (в нашем случае кубических) ячеек с частицами, получающихся друг из друга сдвигом на векторы $L*(k,l,m)$, где L – ребро куба, в котором происходит моделирование, а k, l и m – произвольные целые числа. Всё, что происходит с исходном кубе, дублируется и во всех остальных. Понятно, что после такого предположения число частиц становится бесконечным, и надо как-то учитывать и образы. Можно выделить три основных подхода к их учёту.

Первый известен под названием *closest image convention (CIC)*. Он состоит в том, чтобы учитывать для каждой частицы A взаимодействие только с теми частицами и их образами, которые находятся в шаре с центром в A радиуса $R_c < L/2$, см. рис. 16. R_c называют *радиусом обрезания* (*cutoff*). Физически это соответствует учёту только ближайших образов (возможно, не всех частиц) и пренебрежению остальными. То есть, каждая частица считает, что на расстоянии больше чем R_c от неё ничего нет. Для короткодействующих потенциалов такой подход корректен при разумном выборе радиуса R_c.

Второй метод применяется для явного учёта периодичности. Необходимость учёта всех образов возникает лишь при наличии в системе дальнодействующих сил, главным образом электромагнитных. Тут возникает интересная ситуация. Учёт всех образов эквивалентен вычислению потенциала в виде суммы ряда

$$U = \frac{1}{2} \sum_{1 \le i,j \le N} \tilde{\sum_{\vec{k} \in \mathbf{Z}^3}} \frac{c_{ij}}{|\vec{r}_{ij} + \vec{k}L|^p},$$

где первая сумма идёт по всем упорядоченным парам частиц основной ячейки, а вторая учитывает образы. Волна над последней стоит в знак того, что член $i=j$ опускается, если вектор $k=0$. Случай p=1 соответствует зарядам, а p=3 – диполям. Таким образом, происходит учёт всех образов всех частиц.

Рассмотрим два примера. Сначала пусть в ячейке имеется лишь одна частица с зарядом q. В этом случае написанный ряд, очевидно, будет расходиться. Отсюда возникает мысль, что если мы надеемся посчитать сумму ряда, было бы неплохо, если суммарный заряд всей ячейки был нулевым. Рассмотрим теперь второй случай: добавим в ячейку вторую частицу с зарядом $-q$. Несмотря на то, что суммарный заряд ячейки

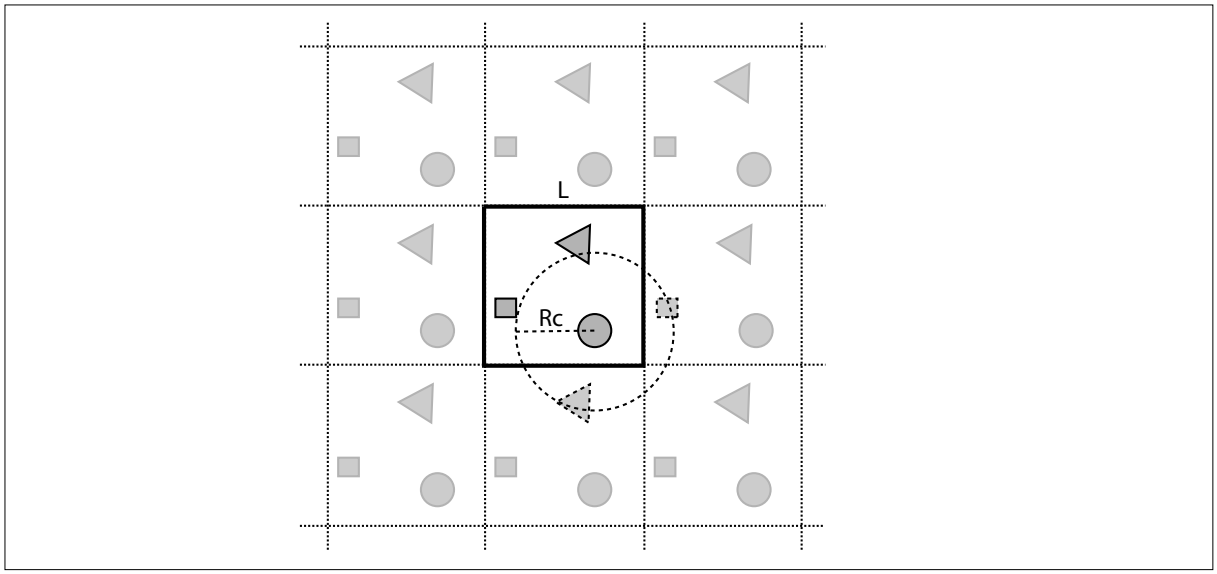

Рис. 16. Иллюстрация периодических граничных условий ближайшего образа. Пунктирная окружность обозначает радиус обрезания; фигуры с пунктирной границей есть ближайшие образы для частицы, обозначенной кругом.

ноль, есть какие-то члены со знаком «+», какие-то со знаком «–», однако ряд попрежнему расходится *абсолютно*. Это означает, что перестановка его членов играет решающую роль: меняя порядок суммирования можно получить в результате любое число! Возникает вопрос, какой порядок суммирования выбрать, чтобы получить разумный результат. Эта проблема достаточно подробно изучена в литературе. Существует ряд методов сложения данного ряда, среди которых основным считается *суммирование Эвальда* (Ewald), изначально предложенное для кристаллов. Помимо названного, существуют и другие разновидности. Для изучения этого вопроса на сегодняшний день имеется обширная литература, по которой читатель может более подробно ознакомиться с методами явного учёта образов. Отметим два источника, [24] и [34], в которых, помимо общих идей, приведены явные выражения сумм Эвальда для работы с зарядами и диполями.

Последним способом учёта периодических граничных условий, который будет рассмотрен в данной работе, является метод самосогласованного поля (*reaction field*). Этот метод является обобщением подхода среднего поля, который применялся в модели Изинга, [35], и для статистического описания поведения диполей во внешнем поле [36]. Идея данного метода состоит в следующем: рассмотрим частицу изотропной фазы (газа или жидкости). Подобно человеку, находящемуся в толпе, эта частица чётко различает лишь своих соседей, находящихся на небольшом расстоянии от неё. Если же человек, находящийся в толпе, попытается понять, что происходит на большом расстоянии от него, ему это не удастся: ближайшие ряды заслоняют ему обзор, и он сможет судить об этом лишь по косвенным признакам (например, звукам или запахам). По тому же принципу, мы предположим, что частица изотропной фазы воспринимает всё частицы, находящиеся от неё на достаточно большом расстоянии, как *непрерывную среду.* Среда эта непременно оказывает воздействие на частицу и сама ощущает воздействие последней. Математическое описание описанной схемы состоит во введении, как и в *CIC*, радиуса обрезания R_c для каждой частицы. Внутри этого радиуса все взаимодействия учитываются явно, в то время как всё, что лежит вне этого шара описывается неким набором макроскопических параметров, после чего делаются некоторые предположения о взаимосвязи частиц ячейки со средой. Более подробное изложение метода в случае, если

упомянутая взаимосвязь строится на основе уравнения Пуассона-Больцмана, можно найти в [37]. Для моделей с диполями может также применяться метод, основанный на формуле Клаузиуса-Моссотти, см. например [38].

Математически проблема граничных условий сводится к компактификации пространства, в котором существуют частицы. В этом отношении, представление ячейки как трёхмерного тора (периодические граничные условия) не единственно. Подобную проблему можно решать и другими способами, например, отождествляя куб с проективным пространством. Подобные методы, однако, менее распространены (подробнее с подобными методами можно ознакомиться, например, в [34]).

6.4.1. Граничные условия в динамике и Монте-Карло

В данном разделе мы уделим внимание вопросу применимости различных методов учёта периодических граничных условий к разным типам задач.

МОЛЕКУЛЯРНАЯ ДИНАМИКА

Основным отличием молекулярно-динамических расчётов является наличие времени как переменной интегрирования дифференциальных уравнений динамики системы. Вслед за временем появляются законы сохранения различных величин: энергии, момента и других. Такие величины называют интегралами движения системы. Именно то, насколько хорошо сохраняются эти величины, часто определяет степень аккуратности расчёта. Причины отсутствия «идеального» сохранения интегралов просты: практически всё численные схемы дискретны, а потому в их терминах можно лишь с некоторой точностью говорить о непрерывных объектах. Так, производные заменяются конечными разностями, да и само время, изначально непрерывное, заменяется некоторым набором точек.

Одно из последствий дискретного приближения связано с периодическими граничными условиями. Дело в том, что, даже пытаясь рассмотреть все образы частиц явно, как это делает суммирование Эвальда, мы не можем иметь дело с бесконечностью. Другими словами, у каждой частицы всегда будет существовать такой радиус R_c, вне которого никакие взаимодействия с ней не учитываются. Но у потенциалов Леннарда-Джонса или кулоновского взаимодействия такого радиуса нет. В этом смысле, происходит искусственное обрезание таких потенциалов (рис. 17).

Рассмотрим теперь последствия таких обрезаний. Пусть две частицы A и B движутся навстречу друг другу с расстояния, большего радиуса обрезания (влиянием остальных частиц пока пренебрежём). Тогда их потенциальная энергия равна нулю. Далее, поскольку время дискретно, то настанет такой момент t, что в этот момент расстояния между A и B больше R_c, а в момент $t+\delta t$ – меньше (δt – шаг интегрирования). Но, в силу принципа работы схем численного интегрирования, кинетическая энергия не почувствует никакого изменения, в то время как потенциальная совершит прыжок. Таким образом, входы и выходы частиц из сфер, по которым происходит обрезание потенциала, ведут к флуктуациям энергии. При этом можно ожидать и разрывного поведения других параметров. Естественно, подобные флуктуации являются эффектом численных методов, которого следует избегать. Величина таких флуктуаций определяется величиной прыжка, совершаемого потенциалом, и их количеством.

Поскольку все потенциалы убывают с расстоянием, можно пытаться уменьшить ошибку такого рода путём увеличения радиуса обрезания. При этом, для короткодействующих потенциалов (например, Леннарда-Джонса), убывание с расстоянием происходит достаточно быстро (намного быстрее, чем площадь сферы, пропорциональная числу частиц, которое за шаг может влезть в эту сферу извне), а потому такой путь может оказаться разумным для любого из трёх методов учёта периодичных условий. Кроме того, ошибки, возникающие из-за обрезания потенциалов часто можно немного уменьшить путём поправок, зависящих от потенциала [24].

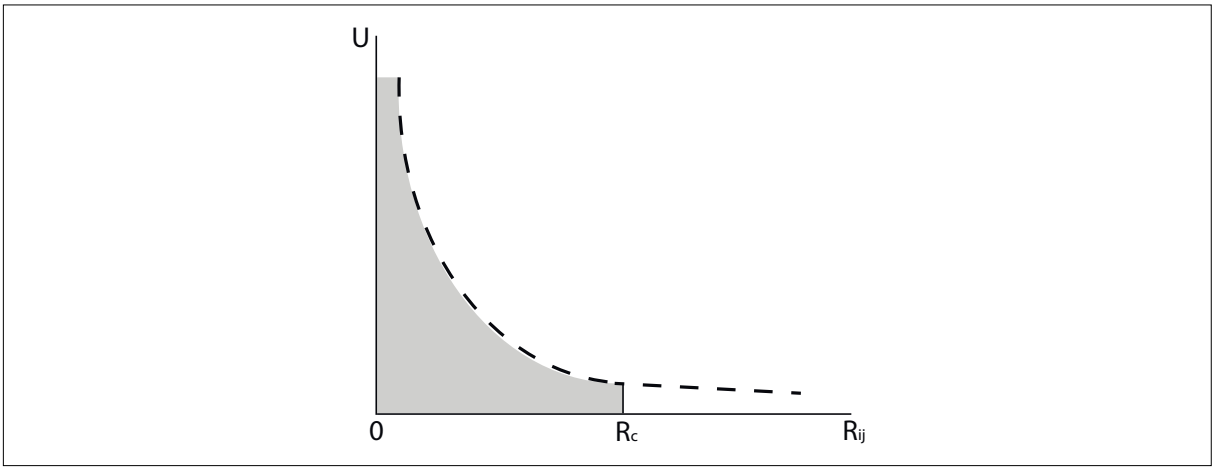

Рис. 17. Исходный (пунктир) и обрезанный по радиусу R_c (закрашенный) потенциалы.

Значительно хуже обстоят дела с дальнодействующими потенциалами. Здесь нельзя утверждать, что увеличение радиуса обрезания увеличит точность, поскольку, например, потенциал взаимодействия заряд-заряд убывает как первая степень расстояния, а площадь сферы растёт как квадрат радиуса. Поэтому, в этом случае можно ожидать и роста флуктуаций. Здесь остаётся лишь надеяться на то, что в равновесии подобные флуктуации уравновесят друг друга.

В этом смысле метод сумм Эвальда выгодно отличается от своих аналогов, поскольку известно, что он сходится, и притом достаточно быстро. По этой причине, система с таким учётом периодических граничных условий наиболее устойчива. Большинство исследователей, в том или ином виде занимающихся моделированием многочастичных задач, из всех методов периодических граничных условий отдают предпочтение именно суммированию Эвальда. Но, как и у любой другой системы, у этого метода есть свои недостатки. Его стоит применять лишь к системам, где имеется строгий (кристаллический) порядок, поскольку именно на него опирается данная схема. По этой причине, моделирование изотропных систем, где флуктуации представляют особой интерес, может привести к неверным результатам. К другим недостаткам этого метода можно отнести его вычислительную сложность (как, впрочем, и определённые трудности реализации) в сравнении с другими методами.

Подводя итог, скажем, что методы *CIC* и самосогласованного поля для дальнодействующих потенциалов могут давать грубые ошибки в поведении физических величин, и менее устойчивы по сравнению с суммами Эвальда. С другой стороны, несмотря на меньшую устойчивость, они могут оказаться более корректными в моделировании изотропных сред. В этом отношении, пока нет универсального метода, которым можно было бы моделировать любую систему. Так, утверждения «*метод А лучше метода В*» не имеют смысла. Правильной формулировкой будет «*метод А более устойчив, чем В, для данной задачи*» или «*метод А даёт более осмысленные результаты, чем В, для данной задачи*».

О сравнении различных методов в динамике см. работу [37] и ссылки в ней.

ТЕРМОСТАТЫ И БАРОСТАТЫ

Как было отмечено выше, флуктуации, возникающие в ходе счёта, являются нежелательным последствием некоторых неточностей модели. Была предпринята попытка минимизировать эти флуктуации, чтобы счёт был более аккуратным в том смысле, что если начать расчёт с определённого состояния, система будет уходить со своей идеаль-

ной траектории как можно медленнее. При этом, под идеальной траекторией мы подразумеваем точное решение уравнений динамики.

Но есть и другой способ, основанный на том, чтобы рассматривать флуктуации численных методов как тепловые. В некотором смысле, этот подход эквивалентен добавлению в уравнения Ньютона некоторой случайной силы (*уравнение Ланжевена*). После этого предположения, ошибки воспринимаются как вполне естественный физический эффект. При этом разумность этого предположения проверяется на практике для каждого конкретного примера. Как результат, энергия, температура и прочие величины могут испытывать значительные флуктуации на протяжении расчёта. Но поскольку нас часто интересует поведение системы в смысле некоторого ансамбля, например, *NVT* (фиксировано число частиц, объём и температура) или *NPT* (число частиц, давление и температура), возникает необходимость периодически возвращать систему на подпространство с фиксированным значением соответствующих величин, для чего создаются отдельные алгоритмы. Так, алгоритмы, которые приводят систему к заданной температуре T_0, называют термостатами. Алгоритмы, проделывающие подобное с давлением, называют баростатами.

В качестве простейшего термостата рассмотрим термостат *Берендсена* (*v-rescale*) на примере задачи *N* частиц. Так, пусть изначально имеется распределение скоростей частиц, такое, что частица с номером *i* имеет скорость v_i. Про эти скорости, вообще говоря, известно два факта. Во-первых, средняя кинетическая энергия равна температуре с некоторым коэффициентом

$$\sum_{i=1}^{N} \frac{p_i^{\,2}}{2m} = NkT$$

здесь $p_i=mv_i$ – импульс частицы. Во вторых, согласно статистической физике, скорости имеют распределение Больцмана, то есть, плотность вероятности того, что частица имеет значение *v*, равна

$$p(v) = \alpha e^{-\frac{mv_i^2}{2kT}},$$

где α – нормировочный коэффициент. Через некоторое число шагов численная схема начинает давать ошибки, что приводит к тому, что средняя температура меняет своё значение (в идеальном случае, то есть, на точных решениях уравнений Ньютона, этого не происходит) и становится равной T_1. Согласно алгоритму Берендсена, производится перенормировка скоростей так, что

$$v_i` = v_i \sqrt{\frac{T}{T_1}},$$

где через v_i обозначены новые скорости. В результате такой замены, средняя кинетическая энергия системы уменьшится в T_1/T раз, и средняя кинетическая энергия системы вновь будет принимать исходное значение. Основная проблема данного алгоритма заключается в том, что в ряде систем он не сохраняет распределение Больцмана, а потому его следует применять с осторожностью.

В качестве примера баростата можно привести следующий алгоритм. Рассмотрим вновь систему *N* частиц, предположив дополнительно их малую плотность. Цель такого предположения в том, чтобы для некоторой константы α имело место приближение идеального газа

$$PV = \alpha RT.$$

Если моделируется *NPT*-ансамбль, необходимо следить, чтобы значение плотности лежало в окрестности некоторого начального значения. Аналогично случаю с температурой, через некоторое время давление системы начнёт изменяться и примет значение

P_1. Идея баростата заключается в том, что можно менять размеры ячейки моделирования так, что новый объём ячейки был бы равен

$$V` = V\frac{P_1}{P},$$

где $V`$ – новый объём. Естественно, объём ячейки не должен меняться слишком быстро, это может привести к слишком большим флуктуациям.

Отметим, что применение подобных методов в дополнение к неустойчивому (например, к *CIC*), может сделать всю систему более устойчивой. По этой причине, данные методы часто используются при молекулярно-динамических расчётах больших систем.

Подробнее о баростатах и термостатах можно прочитать в [24].

МОНТЕ-КАРЛО

В отличие от молекулярной динамики, в расчётах методом Монте-Карло не присутствует время, а потому гладкость потенциалов не мешает данному методу моделирования. Это сужает круг проблем, которые могут возникнуть в ходе моделирования. Так, для учёта периодических граничных условий может быть выбран любой метод без каких-либо опасений за устойчивость численной схемы. Это предоставляет большую свободу в выборе метода, подходящего для данной задачи по структурному критерию. Так, принято считать, что для моделирования систем с дальним порядком (например, кристаллов) более оправданно использование метода Эвальда, в то время как для изотропных неупорядоченных сред (жидкостей) лучше использовать непрерывные подходы, такие как метод самосопряжённого поля.

6.4.2. Применимость методов

Подводя итог разделов о Монте-Карло, отметим, что, помимо метода Метрополиса существуют и другие алгоритмы построения Марковской цепи. Часто они строятся с применением других вероятностей приёма или отмены шага, исходя из постановки решаемой задачи.

Применяя любой алгоритм к конкретной системе, следует помнить, что это – своего рода «трюки», предназначенные для обхода той или иной проблемы. Часто они разрабатывались для конкретной задачи, после чего их стали применять и к другим проблемам. По этой причине следует производить проверку того, применим ли *данный* алгоритм к *данной* задаче. Неумелое или неосмысленное применение алгоритма к неподходящей системе сродни ошибке в эксперименте. Следует также сказать, что подобная "неприменимость" является довольно распространённым явлением, и сегодня известно довольно большое число примеров расчётов с неверным выбором параметров.

К сожалению, сегодня существуют исследователи, использующие подобные алгоритмы, которые не только не задумываются над тем, совместим ли алгоритм с задачей, но и не знают принципов их работы. Данный феномен является следствием большого числа исследований, проводимых в готовых продуктах, например, для молекулярно-динамических расчётов. Поэтому следует всегда заботиться о тщательности проводимых расчётов и, по возможности, проводить одни и те же расчёты, используя различные алгоритмы, после чего сравнивать результаты.

Компьютерное моделирование в целом и динамики многочастичных систем в частности, представляют собой новую научную методологию. Вместо того чтобы следовать традиционной теоретической практике построения шаг за шагом предположения и приближения, эта современная альтернатива предполагает решение исходной задачи сразу во всех подробностях. К сожалению, явления в основном квантово-механической природы по-прежнему представляют концептуальные и технические трудности, но что касается классических задач, подход развивается настолько быстро, насколько развива-

ются компьютерные технологии. Для этого класса задач пределы того, что может быть достигнуто остаются далеко за горизонтом.

Теоретические прорывы включают в себя как новые понятия, так и математический аппарат, с которым их можно развивать. Большинство основных теоретических достижений двадцатого века базируются на математических основах, разработанных в ходе прошлого века, если не раньше. Неизвестно, помогут ли пока не развитые математические инструменты и новые понятия получить информацию, которую в наше время можно получить лишь путем компьютерного моделирования, или приближенное решение является единственным, что можно получить при решении задачи. Станет ли компьютерное моделирование неотъемлемой частью теоретической науки, или же оно будет продолжать существовать также самостоятельно, так же неизвестно, но в настоящее время численные эксперименты играют заметную роль в исследованиях. В более общем смысле, численные эксперименты уже в некоторых областях заменяют до некоторой степени эксперименты реальные, и можно прогнозировать дальнейшее распространение такой практики.

7. Установка и настройка MPI

В предыдущем изложении мы всегда говорили об MPI как о библиотеке (в частности, к C++). Однако это не совсем точно: на самом деле, MPI – это стандарт, то есть набор функций с аргументами и описанием того, что они должны осуществлять. Этот стандарт не содержит описания принципов работы и алгоритмов этих функций, поэтому их реализация остаётся проблемой тех людей, которые пишут библиотеку в соответствии со стандартом.

Любая группа, создающая собственную библиотеку MPI, ориентируется, прежде всего, на компьютерную систему, которая имеется у неё в распоряжении. Ясно, что если две различные группы имеют различные компьютеры и системы их коммуникации, то и реализованные ими библиотеки MPI будут написаны с приоритетами для различных аспектов работы. Так, если компьютерная система состоит из большого числа одноядерных узлов (компьютеров), упор будет сделан на оптимизацию передачи данных между ними. Напротив, если узлов немного, то траты больших усилий на ускорение передачи сообщений могут оказаться неоправданными.

По этой причине, на сегодняшний день существует достаточно большое число различных реализаций MPI, отличающихся друг от друга в некоторых деталях, а именно, в нацеленности на определённый тип вычислительных систем. Выделим три основных класса реализаций MPI: MPICH, OpenMPI и Intel MPI.

MPICH является свободно распространяемой кроссплатформенной реализацией MPI. Суффикс CH в его названии появился от «Chameleon», названия библиотеки для параллельного программирования, разработанной человеком по имени William Gropp, одного из основателей MPICH. На сегодняшний день эта библиотека имеет множество различных модификаций для различных типов систем. Так, например, реализация MVAPICH, разработанная в Ohio State University, предназначена для вычислительных систем с InfiniBand.

OpenMPI является продуктом слияния трёх проектов:

FT-MPI созданного организацией *University of Tennessee*,

LA-MPI написанного *Los Alamos National Laboratory*,

LAM/MPI, *продукта Indiana University*.

Основная мысль этого проекта заключалась в объединении трёх кустарных продуктов в один полноценный, который превосходил бы по скорости остальные проекты этой области.

OpenMPI также является открытой кроссплатформенной версией MPI. Принято считать, что OpenMPI работает быстрее MPICH, хотя это лишь мнение, поскольку много зависит от реализации и систем, на которых производятся вычисления.

Intel MPI, как следует из названия, является продуктом компании Intel. Данная библиотека является кроссплатформенной, хотя и не бесплатна. По причине последнего, она менее распространена среди рядовых пользователей. Данная реализация ориентирована на системы, построенные на процессорах Intel. Ясно, что никто так хорошо не знает эту архитектуру, как сами разработчики Intel, поэтому можно ждать от этой реализации наилучшую производительность на Intel-системах.

Вопрос выбора реализации в общем случае крайне сложен. Он зависит как от системы, на которой планируется проводить вычисления, так и от задачи и её конкретной реализации. В этой связи выбор MPI обычно осуществляется на основании прямого сравнения скорости вычислений для различных реализаций MPI.

7.1. Установка в Windows

В данной главе рассматривается пошаговая установка программного обеспечения, позволяющего как производить параллельные вычисления на различных машинах, так и эмулировать параллельные процессы на персональном компьютере (возможно даже одноядерном) для отладки приложений. Процесс установки будет отличаться в зависимости от операционной системы и версии MPI.

Рассмотрим подробнее процесс установки MPI для Windows (XP, Vista, 7). Будем рассматривать реализацию MPICH2. Двойка в названии — это номер того стандарта MPI, который реализован в библиотеке. MPICH2 соответствует стандарту MPI 2.0. MPICH — одна из самых известных и часто использующихся реализаций MPI, созданная в Арагонской национальной лаборатории (США). Версии этой свободно распространяемой библиотеки существуют для всех популярных операционных систем, в том числе для Windows.

MPICH для Windows состоит из следующих компонентов:

а) Менеджер процессов smpd.exe (Process manager service for MPICH2 applications). Менеджер процессов ведёт список вычислительных узлов системы, и запускает на этих узлах MPI-программы, предоставляя им необходимую информацию для работы и обмена сообщениями.
б) Заголовочные файлы (.h) и библиотеки стадии компиляции (.lib), необходимые для разработки MPI-программ.
в) Библиотеки времени выполнения (.dll), необходимые для работы MPI-программ.
г) Дополнительные утилиты (.exe), необходимые для настройки MPICH и запуска MPI-программ.

Менеджер процессов является основным компонентом, который должен быть установлен и настроен на всех компьютерах сети (библиотеки времени выполнения можно, в крайнем случае, скопировать вместе с MPI-программой). Остальные файлы требуются для разработки MPI-программ и настройки некоторого «головного» компьютера, с которого будет производиться их запуск.

Особо отметим, что перед запуском MPI-программа автоматически не копируется на вычислительные узлы кластера. Менеджер процессов передаёт узлам путь к исполняемому файлу программы в том же виде, в котором пользователь указал этот путь программе Mpiexec. Это означает, что если, например, запустить программу C:\prog.exe, то все менеджеры процессов на вычислительных узлах будут пытаться запустить файл C:\prog.exe. Если хотя бы на одном из узлов такого файла не окажется, произойдёт ошибка запуска MPI-программы.

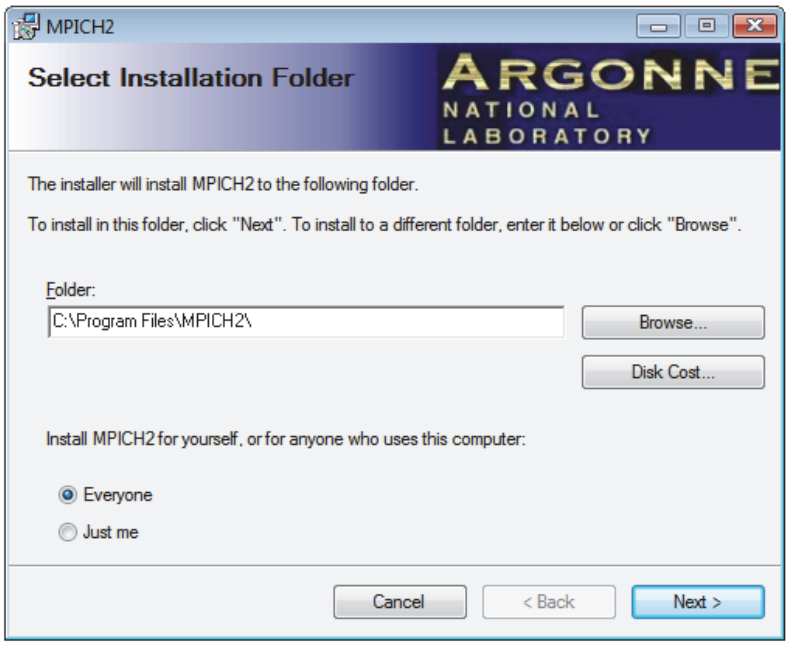

Рис. 18. MPICH2, путь установки.

Поэтому нужно следить, чтобы на каждой машине была копия программы по одному и тому же адресу. Это можно сделать, запуская перед исполнением программы bat-файл, который будет копировать программу на вычислительные узлы. Другим решением является создание общего сетевого диска. При локальном запуске на одной машине все решает операционная система, и не нужно дополнительно заботиться о памяти. В этом руководстве мы не будем рассматривать создание общего сетевого диска.

Теперь остановимся на собственно установке. Мы во многом опираемся на тьюториал [39]. Сначала нужно получить дистрибутив MPICH2, соответствующий машине, на которой будут производиться вычисления. Для 32-х и 64-х разрядных систем дистрибутивы отличаются, и программа не будет работать, если будет выбрана неподходящая реализация. Дальнейшие действия по установке выполнялись в случае 32-х разрядной системы Windows7 с дистрибутивом *mpich2-1.4.1p1-win-ia32.msi*

Выполняется установка программы под руководством установщика. В процессе установки потребуется ввести пароль для smpd.exe. По умолчанию там прописан пароль “behappy”, мы поменяем его на «mpi». Следует использовать исключительно латинские буквы. Когда один менеджер процессов обращается к другому менеджеру процессов, он передаёт ему свой пароль, поэтому при установке на несколько компьютеров нужно обязательно указывать один и тот же пароль.

По умолчанию, установка производится в папку *C:\Program Files\MPICH2.*

В поле «Install MPICH2 for yourself, or for anyone who uses this computer» нужно выбрать вариант «Everyone».

Скорее всего, Брандмауэр спросит о доступе новой программы в сеть. Доступ следует разрешить и для частных, и для общественных сетей.

Теперь, скорее всего, MPICH2 установлен правильно. Но, прежде чем переходить к настройке, обязательно следует проверить две вещи: запущена ли служба «MPICH2 Process Manager», и разрешён ли этой службе доступ в сеть.

Нажмите Пуск → Панель управления → Администрирование → Службы. Вы должны увидеть «MPICH2 Process Manager» в списке служб (рисунок 20). Если служба в списке отсутствует, то нужно переустановить программу от имени администратора.

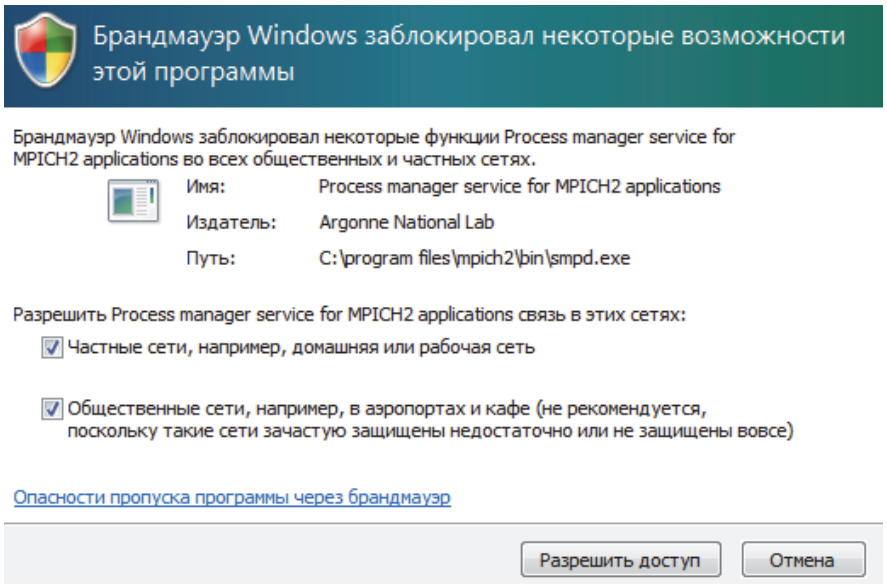

Рис. 19. Брандмауэр Windows, разрешение доступа в сеть для *smpd.exe*

Теперь нужно проверить, разрешён ли доступ в сеть для MPICH. Зайдите в Пуск → Панель управления → Брандмауэр Windows. Там нажмите «Разрешение запуска программы через брандмауэр Windows». В списке разрешённых программ должны быть два приложения: «Process launcher for MPICH2 applications» и «Process manager service for MPICH2 applications». Если какой-то из них в списке нет, можно добавить вручную. Для этого нужно нажать кнопку «Добавить программу...», и добавить C:\program files\mpich2\bin\mpiexec.exe, если отсутствует «Process launcher for MPICH2 applications», и C:\program files\mpich2\bin\smpd.exe, если отсутствует «Process manager service for MPICH2 applications»

Теперь перейдем к настройке. Для корректной работы следует создать отдельную учетную запись. На каждом компьютере, который будет использоваться для расчетов, нужно создать запись с одинаковым именем и паролем. Пароль для учетной записи нужен обязательно, он так же должен совпадать с паролем, который был указан для smpd. При создании имени следует использовать только латинские буквы. Учетная запись создается через панель управления:

```
Панель управления → Учетные записи пользователей → Управление другой учетной
записью → Создание учетной записи.
```

Учетная запись должна быть с правами администратора. В примере мы создали учетную запись *mpi* с правами администратора и паролем «*mpi*». Новую учетную запись следует создать, даже если планируется использовать только эмулятор параллельных процессов.

Следует так же отключить контроль учетных записей UAC, для этого надо нажать на кнопку Изменение параметров контроля учетных записей. Для Windows Vista следует отключить контроль учетных записей, для Windows 7 перевести ползунок в самое низкое положение «никогда не уведомлять».

Для корректной работы следует так же изменить переменную PATH:

```
Компьютер → Свойства → Дополнительно → Переменные среды → переменная path
изменить, добавив в конец через точку с запятой C:\Program Files\MPICH2\bin
```

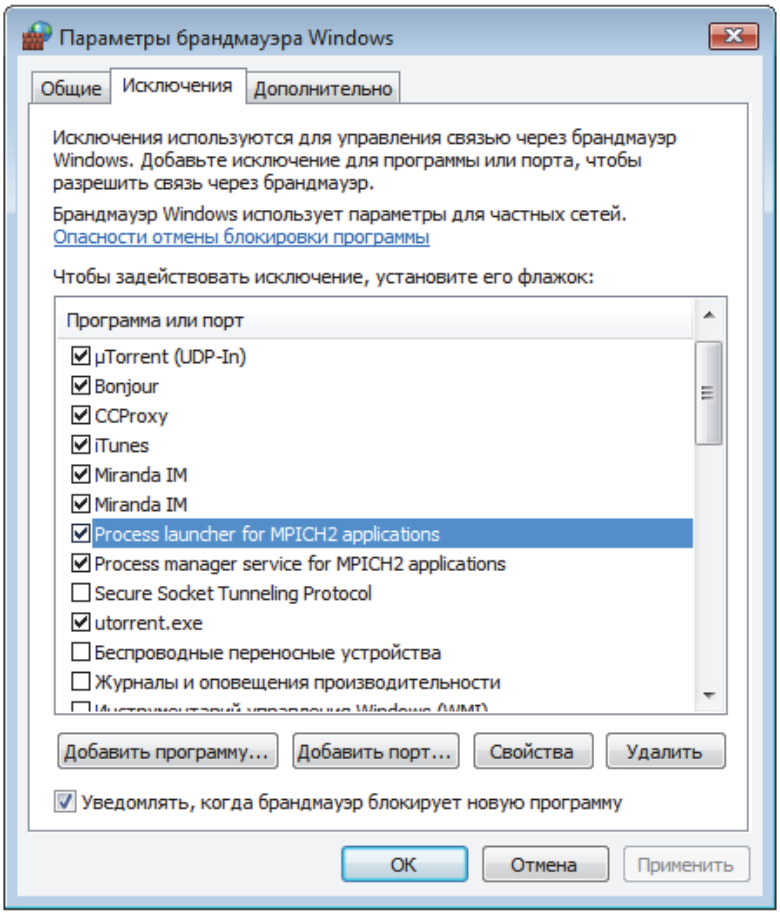

Рис. 20. Программы smpd.exe и mpiexec.exe в списке исключений брандмауэра

Теперь нужно настроить среду разработки, в нашем случае это Visual Studio 2008. Создаем новый проект, использующий библиотеки MPI. Теперь нужно провести следующие действия:

```
Меню Project → Properties → Linker → General → Additional library  directories
добавить C:\Program Files\MPICH2\lib

Меню Project → Properties → Linker → Input → Additional dependencies
добавить  mpi.lib

Меню Project → Properties → C/C++ → General → Additional include directories
добавить C:\Program Files\MPICH2\include
```

Для большего удобства можно настроить копирование исполняемого файла в директорию с более простым путем. Практика показывает, что лучше не допускать пробелов в пути к исполняемому файлу. Настроим копирование исполняемого файла например в папку *C:\prmpi*. Для этого опять следует изменить свойства проекта.

Меню Project – Properties - Build event - Post-build event - command line ввести в поле команду, аналогичную следующей: *copy "C:\Users\MyUser\Documents\Visual Studio 2008\Projects\mpi\Debug\mpi.exe" "C:\prmpi"*. Теперь после компиляции исполняемый файл будет переписываться в папку по адресу *C:\prmpi*.

Теперь проект можно компилировать, все библиотеки должны работать корректно.

Для запуска программы будем пользоваться командной строкой. Открываем ее Программы – Стандартные – Командная строка. Для запуска программы в режиме эмуляции процессов нужно написать следующие команды:

```
>cd C:\prmpi
>mpiexec –n 4 prmpi.exe
```

Первая команда соответствует переходу в директорию, в которой лежит исполняемый файл.

После того, как написана команда mpiexec, менеджер процессов попросит ввести пароль. Нужно ввести тот пароль, который был использован при установке (напомним, что он должен совпадать с паролем от учетной записи), в нашем случае «*mpi*». В течении одной сессии (несколько подряд запусков программы, не закрывая командную строку) пароль нужно будет ввести только один раз. При закрытии и повторном открытии командной строки может потребоваться ввести пароль еще раз.

Отметим, что использовать MPI с Windows для расчетов на нескольких компьютерах не очень удобно, поскольку могут потребоваться дополнительные неочевидные настройки сети в случае разных версий Windows на машинах. Даже если все версии одинаковые, понадобятся дополнительные настройки а так же организация общего сетевого ресурса на одном из компьютеров, о чем можно прочитать в [39]. Установив MPICH2 на один компьютер можно работать в режиме эмуляции процессов для отладки написанных программ. В случае, когда требуется работать с несколькими компьютерами, будет удобнее использовать Linux для настройки корректной работы.

Запуск MPI-программы:

1. Пользователь с помощью программы Mpiexec указывает имя исполняемого файла MPI-программы и требуемое число процессов.
2. Mpiexec передаёт сведения о запуске менеджеру процессов smpd.exe, который располагает сведениями о вычислительных узлах.
3. Менеджер процессов «головного» компьютера обращается к вычислительным узлам по списку, передавая запущенным на них менеджерам процессов указания по запуску MPI-программы.
4. Менеджеры процессов запускают на вычислительных узлах несколько копий MPI-программы, передавая программам необходимую информацию для связи друг с другом.

7.2. Установка OpenMPI на Linux

В данном разделе мы рассмотрим вопросы, касающиеся установки MPI на различные системы на базе debian linux. Обратим внимание, что, как уже отмечалось во введении, MPI предназначен, прежде всего, для работы с кластерами из нескольких компьютеров, соединённых локальной сетью. Другими словами, когда речь идёт о MPI подразумеваются два аспекта: что передача данных осуществляется посредством отправки сообщений, и что передача эта происходит по некоторому сетевому протоколу (обычно, сравнительно медленно).

Принимая во внимание это обстоятельство мы рассмотрим распространённый случай, когда необходимо установить MPI на несколько компьютеров (возможно, многоядерных), соединённых локальной сетью.

Итак, будем считать, что в нашем распоряжении имеется хотя бы 2 компьютера с установленным linux, соединённых локальной сетью. Ввиду простоты в установке, мы изложим процесс создания openMPI-кластера компьютеров с установленным linux на

ядре debian (*buntu-системы) (мы использовали *Ubuntu server*). Необходимо также иметь интернет соединение, поскольку все программы устанавливаются из интернета.

Все действия, которые производятся далее, осуществляются без оглядки на безопасность. В этом смысле, их имеет смысл применять лишь тогда, когда вы полностью доверяете пользователям этой системы и уверены, что никто из них не будет пытаться навредить системе.

Приготовления

Первое, что необходимо сделать, это выбрать один компьютер в качестве главного (*master node*). С него будет происходить запуск MPI-приложений. Остальные компьютеры применяются только для вычисления и их часто называют вспомогательными (*«рабы» – slave nodes*). Так, мы будем называть их соответственно slave1, slave2, …, slaveN.

Кроме того, мы будем считать, что у каждого из этих компьютеров есть статический IP-адрес, который мы будем обозначать через 192.168.1.K, где K=0 для главного узла, и K=i для slave-узла с номером i. Для удобства пропишем соответствующие адреса в файл `/etc/hosts`, дописав в конец этого файла строки вида

```
<ip-адрес>  <имя узла>
```

Например, для первого узла эта запись имеет вид

```
192.168.1.1  slave1
```

И так для всех узлов.

Далее, на всех компьютерах необходимо создать пользователя с одним и тем же именем, например, mpiuser (желательно, чтобы этот пользователь был единственным). Это делается командой

```
> sudo adduser mpiuser
```

В результате, кроме всего прочего, должна создаться папка `/home/mpiuser`.

NFS

Поскольку ввод и вывод программ обычно происходят соответственно из и в файл, необходимо создать общую для всех узлов папку. Это делается при помощи протокола NFS (*Network File System*). С его помощью мы сделаем папку главного узла доступной всем остальным.

Главный узел

Установим на главный узел NFS-Server командой

```
> apt-get install nfs-kernel-server nfs-common portmap
```

Разрешим полный доступ к папке, которую собираемся сделать доступной

```
> chmod 777 /home/mpiuser
```

Откроем файл настройки NFS `/etc/exports` и добавим в него строчку

```
/home/mpiuser/    192.168.1.0/24(rw,insecure,async,no_subtree_check)
```

Сначала указана папка, которую мы хотим сделать доступной, а затем IP-адреса клиентских машин (в нашем случае это slave-узлы; запись `192.168.1.0/24` означает все адреса в промежутке `192.168.1.0 - 192.168.1.255`). В скобках указаны параметры, описывающие степень доступности данной папки клиентам.

Теперь нужно перезапустить сервер, чтобы изменения вступили в силу

```
> /etc/init.d/nfs-kernel-server restart
```

Клиентские узлы

Установим необходимые файлы

```
> apt-get install nfs-common portmap
```

Это практически всё, что нужно сделать. Осталось лишь смонтировать необходимую папку. Для этого допишем в конец файла `/etc/fstab` следующую строку

```
192.168.1.0:/home/mpiuser /home/mpiuser nfs4 _netdev,auto 0 0
```

Она позволяет автоматически подключать папку при каждом включении компьютера. Осталось только написать

```
> mount /home/mpiuser
```

Опять же оговоримся: здесь всё делается очень небезопасно, и если компьютер будет использоваться достаточно большим количеством человек (больше трёх), то рекомендуется подробнее ознакомиться с настройками NFS.

После этих операций файл, который записывается в папку /home/mpiuser появляется на всех узлах.

OpenMPI

Теперь, когда у компьютеров есть общая папка, можно приступить к следующему шагу, а именно, установить openmpi на все узлы. Для этого следует загрузить и установить не обходимые пакеты. Сперва убедимся, что на узлах установлен компилятор g++

```
> apt-get install gcc g++
```

Теперь установим пакеты openmpi

```
> apt-get install openmpi-bin openmpi-common openmpi-doc libopenmpi-dev
```

В случае, если необходимо произвести установку в другой каталог, это можно сделать и вручную, предварительно скачав необходимые файлы и ознакомившись с правилами установки.

OpenSSH

Процедура общения между узлами осуществляется посредством стандартного для linux протокола ssh (*secure shell*), поэтому необходимо убедиться, что на всех узлах установлен ssh-сервер:

```
> apt-get install openssh-server
```

Поскольку управление осуществляется с главного узла, на него необходимо установить ssh-клиент:

```
> apt-get install openssh-client
```

Теперь необходимо сделать так, чтобы главный узел мог беспрепятственно подключаться к slave-узлам. Для этого необходимо из-под пользователя mpiuser на главном узле создать пару файлов-ключей командой

```
> ssh-keygen –t dsa
```

пароль можно выбрать любой, а в качестве директории ключа указать “/home/mpiuser/.ssh/id_dsa”. После этого будет создан файл /home/mpiuser/.ssh/id_dsa.pub, копию которого необходимо разместить в файле /home/mpiuser/.ssh/authorized_keys

```
> cp /home/mpiuser/.ssh/id_dsa.pub /home/mpiuser/.ssh/authorized_keys
```

Поскольку мы сделали папку /home/mpiuser/ главного узла общедоступной, файл /home/mpiuser/.ssh/authorized_keys должен был появиться на всех узлах (если же для общего доступа открыта другая папка, необходимо скопировать этот файл вручную на все узлы). Теперь все узлы смогут распознать главный узел по созданному нами ключу. Осталось лишь поправить разрешения для ключей:

```
> chmod 700 /home/mpiuser/.ssh
> chmod 600 /home/mpiuser/.ssh/authorized_keys
```

После этих операций при подключении главного узла к вспомогательным по протоколу ssh имя пользователя должно определяться автоматически. Проверить это можно, подключившись, например, к первому узлу командой

```
> ssh slave1
```

При этом нужно ввести пароль, с которым создавались ключи.

Для того чтобы обойти ввод пароля, воспользуемся ssh-агентом:

```
> eval ssh-agent
> ssh-add ~/.ssh/id_dsa
```

Теперь подключение

```
> ssh slave1
```

должно осуществляться без пароля. Все эти операции необходимо производить из-под пользователя `mpiuser`.

Настройка MPI

К настоящему моменту у нас имеется система из нескольких компьютеров, из которых один является главным и умеет управлять другими через протокол `ssh`. Однако главный узел пока не знает ничего о том, как физически устроена построенная нами система и как мы хотим распределять её ресурсы. Такая настройка производится путём редактирования так называемого host-файла. Его название и расположение, вообще говоря, произвольно, но вы выберем для этой цели файл `/home/mpiuser/.mpi_hostfile` . Содержание этого файла имеет следующий вид

```
# mpi hostfile
localhost
localhost
slave1
slave1
slave2
slave3
```

Количество упоминаний каждого узла соответствует количеству процессов, которое будет на нём запущенно. Так, в данном случае на главном в первом вспомогательном узлах будет запущено по 2 процесса, в то время как на втором и третьем узлах – по одному. Порядок данных записей определяет тот порядок, в котором будут загружаться узлы. При этом запуском и распределением задач по узлам занимается операционная система.

Компиляция и запуск программ

Компиляция MPI программ происходит так же, как и обычным компилятором `g++/gcc`, но вместо последних следует использовать компиляторы `mpic++` и `mpicc`. Например,

```
mpic++ test.cpp –o test –funroll-loops –O3
```

Для запуска используется команда

```
mpirun –np 6 --hostfile=/home/mpiuser/.mpi_hostfile  ./test
```

где 6 – число процессов, далее – пусть к host-файлу, и, наконец, запускаемый файл.
Проверить работоспособность MPI можно, скомпилировав и запустив программу примера 1.

Прочие настройки

Приведённые в данном разделе настройки осуществляют настройку MPI в случае, когда ожидается, что системой будет пользоваться небольшое число людей. В случае если пользователей будет достаточно много, мы рекомендуем пересмотреть политику безопасности, поскольку неопытный пользователь без труда сможет (и это, как правило, обязательно случается) сбить настройки, что, в лучшем случае, приведёт ко временной неработоспособности системы.

8. Вопросы оптимизации MPI программ

Написание качественных программ, использующих MPI, требует некоторого опыта. Дело в том, что программы, написанные по-разному, будут по-разному работать на различных вычислительных системах. В этом смысле, при выборе реализации программы следует учитывать информацию об архитектуре и топологии тех систем, на которых планируется вести расчёт.

Изначально MPI разрабатывался как система передачи данных между компьютерами при помощи локальной сети. Оптимизация передачи данных является крайне важным вопросом оптимизации MPI-приложений. С изменением архитектур менялся и MPI, но параметры сетей по-прежнему оказывают влияние на работу программ. В вопросах, касающихся передачи данных по некоторому механизму, обычно говорят о двух параметрах: латентности (*latency*) и скорости передачи (*bandwidth*). Первая представляет собой то время, которое проходит с момента вызова функции передачи данных до начала передачи. Этот период можно представлять себе как время, необходимое для установления связи для передачи. Вторым параметром является скорость передачи, то есть, количество байт, передаваемое за секунду (считая с момента, когда передача началась). Комбинация этих параметров для конкретного механизма достаточно сильно зависит от того, в какой системе он используется.

На рисунке 21 представлена информация о параметрах передачи данных для суперкомпьютера Cray XT5, использующего технологию *HyperTransport*. Приведено сравнение параметров передачи между процессами одного узла (*on-node*) и процессами разных узлов (*off-node*). Основное внимание хотелось бы обратить на то, что задержка перед передачей данных в рамках узла значительно ниже. Такая тенденция характерна практически для всех вычислительных систем. Общее время передачи пакета можно оценить по формуле

время передачи = латентность + размер сообщения / скорость передачи

Если сообщений *N*, то эта формула примет вид

*время передачи = N*латентность +общий размер сообщений / скорость передачи*

Таким образом, для того, чтобы минимизировать время передачи в рамках одного механизма передачи, необходимо

- передавать как можно меньше сообщений;
- передавать как можно меньше данных.

ПРЕДПОЧТИТЕЛЬНОЕ ИСПОЛЬЗОВАНИЕ ЛОКАЛЬНЫХ ПЕРЕДАЧ

Значительные значения латентности передачи данных между узлами в сравнении с передачами внутри узла подразумевают то, что передача большого числа пакетов между узлами во время работы программы приведёт к большим временным издержкам. И это верно для суперкомпьютера, сеть которого проектировали и оптимизировали профессионалы. В случае простой локальной сети (100Мбит/с) латентность может оказаться значительно выше.

Поскольку передачи данных внутри узла происходят значительно быстрее, можно попытаться получить некоторую выгоду от преимущественного использования более быстрого механизма передачи. Так, при написании программ, использующих некоторое число независимых коммуникаторов, имеет смысл составлять коммуникаторы из процессов одного узла. Например, в случае *parallel tempering* каждая независимая система может рассчитываться на отдельном узле, что уменьшит затраты времени на передачу данных.

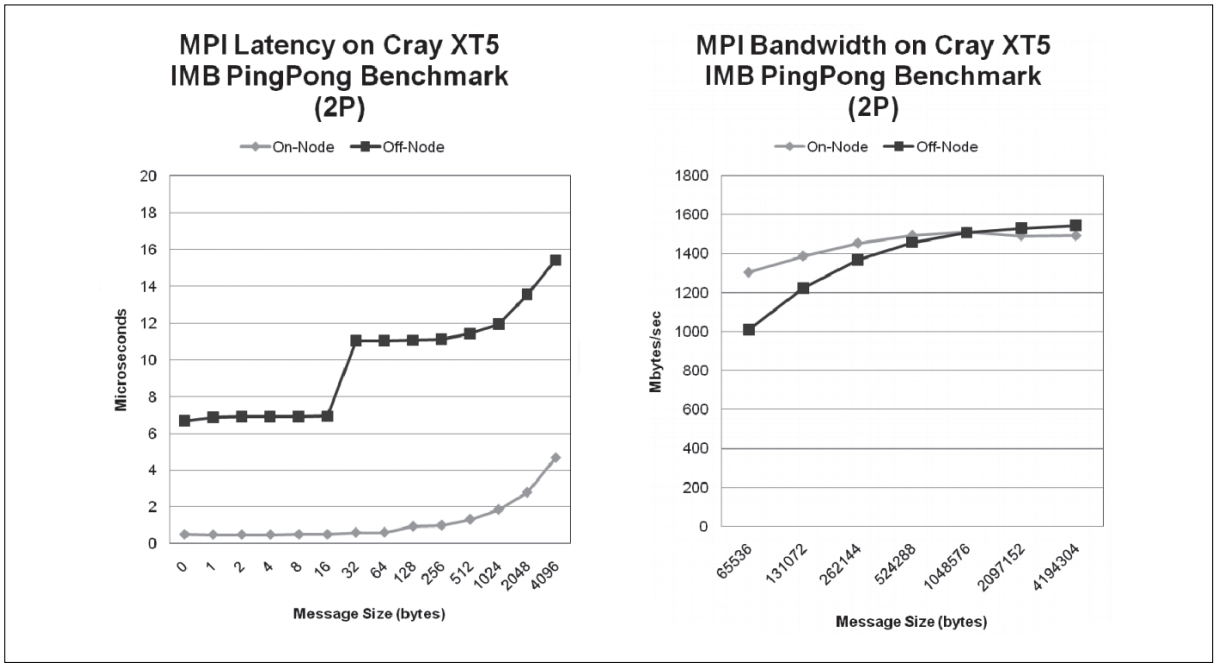

Рис. 21. Сравнение параметров задержки и скорости передачи данных между процессами одного узла (on-node) и между узлами (off-node). Взято из [40].

Осуществить эту идею не всегда просто. В случае небольшого кластера, собранного самостоятельно, ранги процессов в коммуникаторе `MPI_COMM_WORLD` обычно совпадают с тем порядком, в котором записаны узлы в конфигурационном файле MPI (тот, что при установке мы назвали `.mpi_hostfile`). Однако на более сложных машинах, где есть очередь задач, изначально неясно, на каких узлах будет выполняться программа. В этом случае данная проблема должна быть решена для конкретной вычислительной системы. Например, в ряде случаев система постановки задач выделяет для каждой задачи отдельные узлы, то есть, две разные задачи не могут выполняться на одном узле. В этом случае можно ожидать, что ранги процессов в `MPI_COMM_WORLD` представляют собой последовательное перечисление узлов с учётом кратностей. Так, сначала идут все процессы первого узла, затем все процессы второго и так далее. В этом случае достаточно лишь знать, как устроен каждый узел: сколько на нём процессоров и сколько ядер содержит каждый из них.

Несмотря на то, что технологические улучшения сетей значительно понижают латентность сети и увеличивают скорость передачи, обмены данными между процессами в рамках узла по-прежнему остаются более выгодными.

Таким образом, минимизация количества передач между различными узлами является эффективным методом борьбы с высокой латентностью сети, соединяющей узлы.

ПРОТОКОЛЫ ПЕРЕДАЧ СООБЩЕНИЙ

Данный раздел посвящён более подробному изложению принципов работы функции `MPI_Send()`. Как было замечено при её описании, данная функция является блокирующей, то есть, она не возвращает результата (а, значит, программа останавливается на этом месте), пока отправляемое сообщение не было скопировано в буфер, после чего отправитель может безопасно редактировать отправленные переменные. При этом буфер, который мы неявно подразумевали ранее, находился у процесса-адресата и задавался аргументом функции `MPI_Recv()`. Однако MPI предоставляет другую возможность, а именно, создать временный буфер у процесса-получателя и скопировать в него сообщение. Отличие этого случая состоит в том, что функция `MPI_Send()` может вернуть управление *до*

того, как адресат вызовет функцию `MPI_Recv()`. Такая буферизация требует дополнительной памяти и выполнения дополнительных операций. По этой причине её нельзя использовать повсеместно: передача громоздких сообщений потребует лишней операции копирования и наличия буфера большого объёма, что можно значительно понизить скорость работы программы. Для работы с буферизацией в MPI имеются механизмы, позволяющие вручную управлять протоколом передачи сообщений [11].

Функция `MPI_Send()` соответствует *стандартному* протоколу передачи сообщений. Это означает, что выбор, буферизовать сообщения или передавать их сразу, предоставлен MPI. Соответственно, `MPI_Send()` может вернуть управление до того, как адресат получит сообщение, или этого может не произойти, и функция будет ждать, пока адресат не получит сообщение. Обычно, короткие сообщения буферизуются, а длинные – нет.

Протокол *буферизации* соответствует способу передачи сообщений, при котором все сообщения буферизуются, если не вызвана функция получения процессом-получателем. Так, при вызове `MPI_Bsend()` (*buffer send*) процесс отправляет сообщение и забывает про него, возвращая управление основной функции. Размер буфера можно менять самостоятельно соответствующими функциями. Такой подход может дать прирост производительности программы.

Напротив, *синхронный* протокол подразумевает ожидание процессом-отправителем некоторого отклика от адресата. Так, вызванная функция `MPI_Ssend()`(*synchronous send*) посылает запрос на отправку сообщения и не возвращает до тех пор, пока процесс-получатель не разрешит отправку, и данные не будут отправлены.

Последний протокол заключается в отправке по готовности. Так, вызов функции `MPI_Rsend()`(*ready send*) производит отправку сообщения как можно скорее. При этом подразумевается, что получатель *уже вызвал* функцию `MPI_Recv()` и готов к получению. В противном случае возвращается ошибка, а результат не определён.

Отметим, что, несмотря на изобилие функций отправки, получение осуществляется одной функцией `MPI_Recv()` для всех протоколов. Кроме того, с точки зрения порядка следования аргументов все четыре функции отправки одинаковы (то есть, замена одной на другую не вызовет ошибки компилятора).

Механизм буферизации помогает процессу-получателю избежать задержек, обусловленных латентностью сети. В этом смысле, правильное использование протоколов может дать прирост в скорости работы программы. Отметим, что спецификация MPI [11] не запрещает совпадение реализаций функций отправки по готовности и стандартной отправки.

АСИНХРОННЫЕ ПЕРЕДАЧИ

Другим способом оптимизации программ является использование механизма асинхронных передач сообщений. Идея данного метода заключается в выполнении передач данных в две итерации, явно выделяя отправку и получение сообщения. Такая реализация подразумевает наложение передачи сообщения и выполнения основных команд, за счёт чего время простоя процессов уменьшается и возрастает производительность. Минусом использования неблокирующих передач является необходимость следить за тем, чтобы не изменить данные до того, как они полностью отправились для процесса-отправителя. В случае процесса-получателя следует быть внимательным, чтобы не начать обработку еще не пришедших данных. Таким образом, воспользоваться результатом неблокирующей коммуникационной операции или повторно использовать её параметры можно лишь после её полного завершения.

Чтобы начать асинхронную передачу, необходимо осуществить вызов функции-инициатора. На асинхронные передачи распространяются те же протоколы, что и на блокирующие, а потому есть не одна, а четыре функции-инициатора: `MPI_Isend()`,

MPI_Ibsend(), MPI_Issend(), MPI_Irsend(). Данные функции запускают процесс передачи данных, которые возвращают управление до того, как передаваемые данные будут скопированы в буфер. Аналогичным образом функция MPI_Irecv() запускает процесс получения данных и может вернуть до того, как данные будут получены. Эти функции передают командование передачей сообщения другим устройствам, отвечающим за коммуникацию между элементами вычислительной системы. После этого процессы продолжают свою деятельность, не меняя буферы отправки и получения (они всё ещё могут использоваться передающими механизмами), пока не будут вызваны функции завершения передачи. Упомянутые команды имеют следующий синтаксис

```
int MPI_Isend(
         (void*)       buf,      // адрес отправки данных                        input
          int          count,    // число элементов в отправляемом буфере       input
          MPI_Datatype datatype, // тип передаваемых данных                      input
          int          dest,     // ранг процесса-получателя                     input
          int          tag,      // метка сообщения                              input
          MPI_Comm     comm,     // коммуникатор                                 input
          MPI_Request* request   // идентификатор асинхронной передачи(int) output
          )
```

```
int MPI_Irecv(
         (void*)       buf,        // адрес приёма данных                    input
          int          count,      // число принимаемых элементов            input
          MPI_Datatype datatype,   // тип передаваемых данных                input
          int          dest,       // ранг процесса-отправителя              input
          int          tag,        // метка сообщения                        input
          MPI_Comm     comm,       // коммуникатор                           input
          MPI_Request* request     // идентификатор асинхронной
                                   // передачи(int)                         output
          )
```

Следует отметить небольшую особенность, отличающую отправку блокирующих сообщений от асинхронных. Эта особенность состоит в наличии дополнительного аргумента *request*, необходимого для того, чтобы после инициации как-то обращаться к протекающей асинхронной передаче. Это своего рода ссылка на передачу.

Что касается функций завершения отправки сообщения, их две. Первая из них, MPI_Wait(), возвращает управление в том случае, если сообщение было успешно отправлено в рамках указанного протокола, то есть, оказалось в буфере или у процесса-получателя. В случае процесса-получателя данная функция возвращает тогда, когда сообщение было получено. Отметим, что асинхронные отправки могут быть получены синхронным (блокирующим) получением и наоборот.

Помимо MPI_Wait() существует ещё одна функция завершения передачи сообщения MPI_Test(). Разница между ними состоит в том, что первая функция является блокирующей, а вторая нет. Так, используя вторую функцию, процесс может спросить, завершена ли передача, и действовать по-разному в зависимости от ответа. Первая функция не возвращает управление до тех пор, пока передача не будет завершена.

```
int MPI_Wait (
          MPI_Request* request,   // идентификатор асинхронной передачи    input
          MPI_Status*  status     // статус                               output
          );
```

```
int MPI_Test(
          MPI_Request* request,   // идентификатор асинхронной передачи      input
          int*         flag,      // завершена ли передача                   output
          MPI_Status*  status     // статус                                  output
          );
```

Обращение к MPI_Test возвращает *flag*=1, если операция, указанная в запросе *request* завершена. В этом случае объект *status* содержит информацию об операции. Иначе возвращается *flag*=0, *status* в этом случае не определён.

Обратим внимание, что завершение неблокирующей операции передачи не влечет за собой завершение операции приема и наоборот. Завершение операции передачи показывает лишь, что отправитель теперь может изменять содержимое ячеек буфера передачи (операция передачи сама не меняет содержание буфера), про принимающий процесс нельзя сделать никаких выводов, следует выяснять статус приема отдельно. Аналогично завершение операции приема показывает, что буфер приема содержит принятое сообщение, и процесс-получатель может обращаться к принятым данным, но это не означает, что операция посылки завершена (указывает лишь, что передача была инициирована).

ПРОЧИЕ СОВЕТЫ

Остановимся в данном разделе, прежде всего, на использовании групповых операциях. В случае, если в программе необходима групповая передача данных, рекомендуется использовать встроенные функции MPI, поскольку они хорошо оптимизированы и предоставляют широкие возможности обмена сообщениями. Лишь функцию ALL-TO-ALL следует избегать, поскольку сам механизм подобного обмена требует большого количества оправок сообщений.

Кроме того, следует избегать использования барьеров MPI_BARRIER, поскольку большинство функций коллективных передач данных является блокирующими.

В заключение отметим, что не следует переусердствовать с попытками оптимизировать передачи данных. Часто использование сложных механизмов сопряжено с ростом времени работы программы. Следует помнить, что оптимизированная программа – это, прежде всего, равномерное соотношения времён работы всех её частей. Часто нет смысла оптимизировать кусок кода, который выполняется десятую часть времени всей программы. Необходимо следить за отсутствием грубых ошибок, и осуществлять оптимизации лишь в том случае, когда это действительно необходимо (например, если это увеличит скорость расчёта в несколько раз).

Заключение

Мы рассмотрели вопросы, касающиеся применения MPI к моделированию задач физики конденсированных сред. В вопросе применения MPI мы ограничились лишь языком C++. При этом, мы пытались сделать так, чтобы примеры использования MPI были понятны и людям, не знающим C++, но, возможно, знакомым с другими языками программирования. Если читатель заинтересован в применении связки Fortran-MPI, он может обратиться к спецификации библиотеки MPI, [11].

Кроме того, этот же документ содержит и описание объекта MPI, содержащего все возможности MPI, завёрнутые в объектную оболочку, что соответствует распространённой парадигме программирования на C++. Однако при введении базовых функций мы использовали C-синтаксис MPI, поскольку он, на наш взгляд, более подходит для ознакомления с библиотекой читателей, не знакомых с C++. В этом отношении, исполь-

зование функций с «объектно-ориентированными» названиями вида `MPI::Datatype::Get_true_extent()` может оказаться пугающим и запутывающим. Если же читателю близок подобный подход и применение классов его не пугает, а, напротив, кажется привычным (как, скажем, в случае *Java*), то он может ознакомиться и использовать для параллелизации объект MPI.

Что же касается примеров, которые мы приводим, то, в случае MPI-программ, они приведены полностью, чтобы читатель мог скомпилировать и испытать их самостоятельно. С другой стороны, полных текстов программ Монте-Карло или молекулярной динамики мы не приводим. Это сделано потому, что мы старались не перегружать текст кодом, а делать акцент на идеях, которые используются в алгоритмах. Данная расстановка приоритетов сделана из тех соображений, что, во-первых, наличие кода и его объяснение существенно увеличило бы объём текста, а, во-вторых, мы хотели бы, чтобы читатель, ознакомившийся с представленным материалом относительно небольшого объёма, мог представить достаточно полную картину методов моделирования физических систем многих частиц. Нашей целью, таким образом, было создать некий каркас из идей, который после может быть самостоятельно достроен читателем, исходя из того, чем он планирует заниматься. Так, это могут быть математические аспекты применимости алгоритмов [16], более подробное знакомство с молекулярной динамикой [24-26], Монте-Карло [24], или со стохастическими и прочими алгоритмами в целом [33].

Наконец, для тех, кто заинтересуется параллельным программированием, мы приводим процесс установки MPI, а также минимум сведений, необходимых для того, чтобы начать полноценно использовать эту библиотеку. Естественно, мы не могли полностью включить её спецификацию. В этом отношении, мы рекомендуем подробнее ознакомиться с полным описанием MPI [11], в особенности, приведёнными там примерами.

Данная работа написана при поддержке РФФИ (гранты №10-01-00748 и №11-02-01462а), программы «Ведущие научные школы РФ» (грант НШ-3224.2010.1), и поддержке гранта Правительства РФ для господдержки научных исследований, проводимых под руководством ведущих ученых, в ФГБОУ ВПО "Московский государственный университет имени М.В.Ломоносова" по договору № 11.G34.31.0054.

Литература

1. *http://www.gromacs.org/*
2. *http://www.ks.uiuc.edu/Research/namd/*
3. *http://lammps.sandia.gov/*
4. Grand Challenges: High Performance Computing and Communications. The FY 1992 U.S. Research and Development Program // A Report by the Committee on Physical, Mathematical and Engineering Sciences, 64pp.
5. *Amdahl G.* Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities //AFIPS Conference Proceedings **30,** 483–485, 1967.
6. *Гергель В.П.* Теория и практика параллельных вычислений. Учебное пособие // М.: БИНОМ. Лаборатория знаний, 2007, 423 стр.
7. *Барский А.Б.* Архитектура параллельных вычислительных систем // ИНТУИТ.ру, 2006.
8. *Богачёв К.Ю.* Основы параллельного программирования // М.: БИНОМ. Лаборатория знаний, 2003, 342с.
9. *Pacheco P.* Parallel Programming with MPI // Morgan Kaufmann, 1996.
10. *Gropp W., Lusk E., Skjellum A.* Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation) // MIT Press, 1999.
11. *http://www.mpi-forum.org/docs/docs.html*
12. *http://www.mhpcc.edu/training/workshop/mpi/main.html*
13. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления // СПб.: БХВ-Петербург, 2002, 608 стр.
14. *Kumar V., Grama A., Gupta A., Karypis G.* Introduction to Parallel Computing // Benjamin Cummings/ Addison Wesley, Redwod City, 1994 (2nd edition, 2003).
15. *Dongarra J.J., Duff L.S., Sorensen D.C., Vorst H.A.V.* Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) // SIAM, Philadelphia, 1998.
16. *Бахвалов Н.С., Жидков Н.П., Кобельков Г.М.* Численные методы // М.: Наука, 1987, 630 стр.

17. *Самарский А.А.* Введение в теорию разностных схем // М.:Наука, 1971, 553 стр..
18. *Корнеев В.В.* Параллельное программирование в MPI // Новосибирск: ИВМиМГ СО РАН, 2002, 215 стр.
19. *Alder B.J., Waingwright T.E.* Phase Transition for a Hard Sphere System // J. Chem. Phys. **27**, 1208-1209, 1957.
20. *Gibson J.B., Goland A.N., Milgram M., Vineyard G.H.* Dynamics of Radiation Damage// Phys. Rev. **120**, 1229-1253, 1960.
21. *Rahman A.* Correlations in the Motion of Atoms in Liquid Argon// Phys. Rev. **136**, A405-A411, 1964.
22. *Shaw D.E. et al* Atomic-level characterization of the Structural Dynamics of Proteins // Science **330**, 341–346, 2010.
23. *Allen M.P., Tildesley D.J.* Computer Simulation of Liquids // Oxford: Clarendon Press, 2002.
24. *Frenkel D., Smit B.* Understanding molecular simulation. From Algorithms to Applications // Academic Press, 2002, 658pp.
25. *Rapaport D.C.* The Art of Molecular Dynamics Simulation // Cambridge University Press, 1995, 565pp.
26. *Мирный В., Фрэнер М.* Об одной программе моделирования молекулярной динамики газа с элементами распараллеливания алгоритма // Вычислительные технологии, **6**, 3, 2001.
27. *Raine A.R.C., Fincham D., Smith W.* Systolic loop methods for molecular dynamics simulation using multiple transputers // Compo Phys. Comm. **55**, 13, 1989.
28. *Rapaport D.C.* Multi-million particle molecular dynamics II: Design considerations for distributed processing // Compo Phys. Comm. **62**, 217, 1991.
29. *Haile J.M.* Molecular Dynamics Simulation. Elementary methods // N.Y.: John Wiley & Sons, 1992.
30. *Vanderbilt D., Louie S.G.* A Monte carlo simulated annealing approach to optimization over continuous variables // J.Comp.Phys. **56**, 2, 259-271, 1984.
31. *Coley D.A.* An Introduction to Genetic Algorithms for Scientists and Engineers // World Scientific Publishing Co., 1999, 227pp.
32. *Preis T. et al*. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model // J. Comp. Phys. **228**, 12, 4468-4477,2009.
33. *Press W.H. et al.* Numerical Recipes: The Art of Scientific Computing, 3rd edition // Cambridge University Press, 2007, 1262 pp.
34. *Weis J.-J., Levesque D.* Simple dipolar Fluids as Generic Models for Soft Matter // Adv. Polym. Sci. **185**, 163-225, 2005.
35. *Исихара А.* Статистическая физика // М.:МИР, 1973, 471 стр.
36. *Зоммерфельд А.* Электродинамика // М.:Издательство Иностранной Литературы, 1958, 505 стр.
37. *Tironi I.G. et al.* A generalized reaction field method for molecular dynamics simulations // J. Chem. Phys. **102**, 5451-5459, 1995.
38. *Bartke J., Hentschke R.* Phase behaviour of the Stockmayer fluid via molecular dynamics simulation // Phys. Rev. E **75**, 061503, 2007, 11pp.
39. *http://iproc.ru/programming/mpich-windows*
40. *http://www.nccs.gov/wp-content/uploads/2009/10/nccs_mpi-optimization_and_tips_print.pdf*

PROGRAMMING FOR PROBLEMS OF CONDENSED MATTER PHYSICS USING MPI

V.N. Blinov, A.A. Seveniuk

The Lomonosov Moscow State University

blinov.veniamin@gmail.com, kireeva.al@gmail.com

Received 01.06.2012

Recent development of computational systems has chosen a path of mutli-core architectures instead of clock rate increase. This choice dramatically changed the paradigms of scientific programming. Thus, the problem of microscopic description of a typical condensed matter physics system can be effectively implemented on a mutli-processor system. This text contains basic ideas modelling of such systems as well as the basics of MPI.