Министерство образования и науки Российской Федерации
ФГБОУ ВПО «Удмуртский государственный университет»
Математический факультет

С. П. Копысов, А. К. Новиков

ПРОМЕЖУТОЧНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Учебное пособие

Ижевск
Издательство «Удмуртский университет»
2012

УДК 519.6, 004.4
ББК 32.973
К 65

Рекомендовано к изданию Учебно-методическим советом УдГУ

Рецензент: д. ф.-м.н., проф. М. В. Якобовский

С. П. Копысов, А. К. Новиков
К 65 Промежуточное программное обеспечение параллельных вычислений.
Ижевск: Изд-во «Удмуртский университет». 2012. 140 с.

Учебное пособие предназначено для студентов математического факультета, изучающих дисциплины, связанные с параллельными вычислениями: «Промежуточное программное обеспечение», «Параллельные алгоритмы», «Методы декомпозиции», «Программное обеспечение многопроцессорных вычислительных систем». Пособие охватывает разделы курса «Промежуточное программное обеспечение», посвященные программному обеспечению параллельных вычислений, содержит фрагменты программ и примеры заданий.

УДК 519.6, 004.4
ББК 32.973

© С. П. Копысов, А. К. Новиков, 2012
© Изд-во «Удмуртский университет», 2012

Содержание

Предисловие **6**

1 Промежуточное программное обеспечение высокопроизводительных вычислений **8**

1.1 Классификация . . . 9
1.2 Реализуемые модели параллелизма . . . 11
1.2.1 Модель обмен сообщениями . . . 16
1.2.2 Модель «общая память» . . . 17
1.2.3 Модель «параллелизм данных» . . . 17
1.3 Предъявляемые требования . . . 18
1.3.1 Инвариантность к языкам программирования . . . 18
1.3.2 Платформонезависимость . . . 19
1.3.3 Адаптация к различным архитектурам аппаратных средств . . . 20
1.3.4 Использование различных моделей распараллеливания 21
1.3.5 Объектно-ориентированный подход . . . 21
1.3.6 Компонентный подход . . . 23
1.3.7 Инкапсуляция и интеграция существующих программных систем . . . 24
1.3.8 Система управления . . . 24
1.3.9 Особенности программирования для многоядерных вычислительных систем . . . 25
1.4 Пример распараллеливания матрично-векторного произведения . . . 30
1.4.1 Параллельное вычисление скалярных произведений . 32
1.4.2 Параллельное вычисление линейных комбинаций . . 33
1.4.3 Оценка времени параллельного выполнения . . . 33
1.4.4 Произведение разреженной матрицы на вектор . . . 35
1.4.5 Объектно-ориентированное программирование в SpMV 41
1.4.6 Последовательная реализация SpMV на C++ . . . 42

2 Технология разработки масштабируемых параллельных программ на MPI **46**

2.1 Модель передачи сообщений MPI . . . 47
2.2 Обзор интерфейса передачи сообщениями MPI . . . 50
2.3 Особенности стандарта MPI-2 . . . 53
2.4 Оптимизация программ в модели обмена сообщениями . . . 55

2.4.1 Привязка MPI-процессов к ресурсам вычислительной системы . . . 56
2.4.2 Оптимизация коллективных коммуникаций . . . 58
2.5 Пример реализации SpMV на MPI . . . 61
2.6 Объектно-ориентированные интерфейсы MPI . . . 64
2.7 Интерфейсы передачи объектных сообщений . . . 65
2.7.1 TPO++ . . . 66
2.7.2 Charm++ . . . 68

3 Многопоточные вычисления на OpenMP 70
3.1 Модель параллельной программы . . . 70
3.2 Директивы и функции . . . 70
3.3 Параллельные циклы . . . 72
3.4 OpenMP в объектно-ориентированных программах . . . 72
3.5 Привязки потоков в OpenMP . . . 74
3.6 SpMV в модели общей памяти . . . 75

4 Массивно-параллельные вычисления в технологии CUDA 78
4.1 Архитектура GPU . . . 78
4.2 Технология CUDA . . . 81
4.2.1 Структура программы на CUDA . . . 82
4.2.2 Потоки в CUDA . . . 83
4.2.3 Типы памяти в CUDA . . . 84
4.3 Технология CUDA для нескольких GPU . . . 85
4.4 Поддержка программирования на C++ в CUDA . . . 86
4.5 Пример SpMV в технологии CUDA . . . 88

5 Использование гибридных технологий на гетерогенных кластерах 93
5.1 Технология OpenCL . . . 93
5.1.1 Архитектура OpenCL . . . 94
5.1.2 Модель платформы . . . 95
5.1.3 Модель исполнения . . . 96
5.1.4 Модель памяти . . . 99
5.1.5 Модель программирования . . . 100
5.1.6 Среда программирования OpenCL . . . 102
5.1.7 Пример использования технологии . . . 103
5.2 Гибридная модель MPI/OpenMP . . . 107

5.2.1 Возможности повышения производительности в гибридной модели MPI/OpenMP 107
5.2.2 Подходы к построению гибридной модели MPI/OpenMP 109
5.2.3 Особенности привязки процессов/потоков в гибридной модели MPI/OpenMP 110
5.2.4 Пример SpMV для гибридной модели MPI/OpenMP . 111
5.3 Гибридные модели MPI/CUDA и OpenMP/CUDA 117
5.3.1 Примеры программного кода SpMV 119
5.4 Гибридная модель MPI/CORBA 122
5.4.1 Технология CORBA 122
5.4.2 Метод интеграции CORBA и MPI 125
5.4.3 Построения интегрированной прикладной программной системы 127

Практические задания 132

Список сокращений 134

Предметный указатель 135

Список литературы 136

Предисловие

Настоящее издание является частью учебно-методических разработок, проводимых на кафедре «Вычислительная механика». Цель издания — привести общие сведения о промежуточном программном обеспечении, на относительно простых примерах ознакомить студентов с парадигмами программирования для параллельных вычислительных систем, подготовить базу для выполнения практических заданий по курсам «Промежуточное программное обеспечение», «Программное обеспечение многопроцессорных вычислительных систем», «Параллельные алгоритмы», «Методы декомпозиции».

Повышение сложности математических моделей требует высокой производительности аппаратных и программных вычислительных средств. Одним из способов повышения производительности программного обеспечения (ПО) является использование различных моделей распараллеливания. Для этого создается промежуточное программное обеспечение (ППО), которое реализует те или иные технологии распараллеливания. В книге представлен анализ некоторых из них, а также определен ряд требований к ППО.

Для правильной ориентации в растущем и усложняющемся рынке промежуточного ПО требуется, во-первых, понимание действующих стандартов. Во-вторых, необходимо разбираться с категориями существующего промежуточного ПО и понять назначение каждой категории. Полезно познакомиться с существующими свободно распространяемым ПО и научится правильно их использовать. В-третьих, в настоящее время везде используются много- и мульти-ядерные вычислительные системы, для которых целесообразно применение комбинации ППО для построения эффективных параллельных алгоритмов решения задач.

Особенностью данного пособия является то, что изложение материала строится на примере оценки параллельного ускорения и эффективности различных подходов к параллельной реализации матрично-векторного произведения (SpMV), используемого промежуточного программного обеспечения, зависимости способа хранения разряженных матриц и имеющихся возможностей различных современных вычислительных систем.

В большинстве рассматриваемых в книге примеров программ авторы будут стремиться к использованию объектно-ориентированного языка C++. Это связано с тем, что в C++ имеется возможность применения различных парадигм программирования, с другой стороны можно не использовать определённые языковые средства в соответствии с предпочтениями разработчика, например, не использовать макросы, шаблоны и т.д.

Составляя учебное пособие, авторы преследовали цель облегчить изучение предмета и подготовку к экзамену, а также познакомить преподавателей и всех заинтересованных лиц с конкретным содержанием читаемого курса, используя вариант электронного учебного пособия. Представление учебного материала в гипертекстовой форме существенно изменяет структуру и расширяет возможности электронного текста и позволяет определить дополнительные смысловые пространства и связи. Предполагается, что читателю доступен выход в Интернет. По ссылкам в библиографии можно найти практически весь список литературы для более глубокого изучения, установить рассматриваемое свободно-распространяемое промежуточное программное обеспечение и приступить к реализации практических заданий, взяв за основу шаблоны предлагаемых алгоритмов и программ.

На основании изучения этой дисциплины студент должен уметь применять промежуточное ПО и методы параллельного программирования в своей практической деятельности, знать основные принципы организации параллельных программных систем, иметь представление об основных тенденциях развития современных вычислительных технологий.

Курс является составной частью цикла специальных дисциплин, определяющих подготовку студентов в области современных вычислительных и информационных технологий.

Книга рассчитана на студентов старших курсов физико-математических факультетов. Она будет полезна также и аспирантам соответствующих специальностей. По-видимому, она будет интересна и специалистам, в первую очередь, гибридизацией технологий и теми требованиями, которые предъявляются к промежуточному программному обеспечению при создании масштабируемых программных комплексов для решении сложных индустриальных задач.

Авторы надеются, что данное руководство окажется полезным в учебном процессе и с благодарностью примут замечания и пожелания читателей.

Авторы глубоко признательны М.В. Якобовскому за конструктивные и доброжелательные дискуссии. Многие вопросы, затронутые в книге, активно обсуждались с В.Н. Рычковым, Л.Е. Тонковым и Н.С. Недожогиным. Авторы приносят им свою искреннюю благодарность.

1 Промежуточное программное обеспечение высокопроизводительных вычислений

Под параллельной вычислительной системой (ВС) будем понимать множество элементов аппаратного (hardware), промежуточного программного (middleware) и прикладного программного (software) обеспечения с набором связей между ними [1]. Все эти элементы объединены с целью проведения высокопроизводительных параллельных вычислений.

Взаимодействие элементов в ВС строится следующим образом: с элементами аппаратного обеспечения непосредственно взаимодействуют элементы промежуточного ПО, которые, учитывая все особенности архитектуры ЭВМ, предоставляют в совокупности программные средства управления ВС (программную модель ВС). Прикладные программы software через программную модель, обеспеченную middleware, реализуют те или иные алгоритмы для решения прикладной задачи, которая является целью данной ВС. Необходимо отметить, что промежуточное программное обеспечение (ППО) может быть многоуровневым, когда элементы объединяются в некоторые подсистемы, которые находятся в иерархической зависимости, например: операционная система – коммуникационная среда – система распределенных компонентов.

Термин «промежуточное программное обеспечение» является довольно устоявшимся, но в то же время его нередко используют для обозначения разных понятий. С самой общей точки зрения, ППО является типом программного обеспечения, предоставляющим API (Application Programming Interface — интерфейс прикладного программирования) между приложением и ресурсами, необходимыми ему для нормального функционирования, и которое помогает справляться с неоднородностью и распределённостью. Исходя из этого, промежуточным может быть названо любое ПО, позволяющее упростить процесс взаимодействия приложений друг с другом или с ресурсами [2].

В параллельной вычислительной системе может иметься один или несколько программных компонентов на одном или нескольких вычислительных узлах. Чтобы система предстала в виде единой вычислительной системы, эти компоненты должны поддерживать взаимные коммуникации. Теоретически, компоненты могли бы общаться с использованием примитивов, обеспечиваемых непосредственно сетевой операционной системой. На практике это слишком сложно, поскольку программистам приложений пришлось бы синхронизовать коммуникации между распределенными компонентами, выполняемыми параллельно, преобразовывать структуры данных уровня приложения в потоки байтов, которые можно пере-

давать на основе сетевых транспортных протоколов, и справляться с неоднородностью представления данных на разных компьютерных архитектурах. Основная роль промежуточного программного обеспечения состоит в упрощении этих задач и обеспечении уместных абстракций, используемых прикладными программистами при построении взаимодействующих компонентов параллельных/распределенных вычислительных систем.

ППО выполняет две основные функции. Первая — облегчение доступа приложений к ресурсам. Такая функциональность важна прежде всего для разработчиков, поскольку позволяет им большую часть времени уделять созданию логики, а не разработке механизмов доступа к ресурсам. Вторая функция ППО — ускорение процессов взаимодействия. Действительно, ПО, специально спроектированное для обеспечения взаимодействия, как правило, обладает лучшей производительностью, по сравнению с неспециализированным решением, созданным разработчиком.

Определим ППО, как специальный уровень прикладного ПО, который расположен между параллельной вычислительной частью приложения и коммуникационным уровнем операционной системы, и связывает приложение с сетевыми протоколами и инструментарием операционных систем. Промежуточное ПО сглаживает различия аппаратно-программных платформ.

1.1 Классификация

Все средства ППО можно разделить на две большие группы: ППО первой из них обеспечивает взаимодействие между активным приложением и пассивным ресурсом, второй — между активными приложениями. Под активным будем подразумевать приложение, реализующее некоторую логику, а роль пассивного ресурса может выполнять, например, сервер базы данных. Эту группу можно разбить на две основные подгруппы: ППО для работы с базами данных, и мониторы транзакций. Эти виды ППО остаются за рамками данной книги т.к. не связаны с высокопроизводительными вычислениями.

Промежуточное ПО, отнесенное по данной классификации ко второй группе (обеспечивающее взаимодействие между активными приложениями), может быть условно разбито на три основных типа [3]: ППО удаленного вызова процедур (RPC — Remots Procedure Call) (см. также раздел 5.4), ППО передачи сообщений (MOM — Message Orientet middleware) (раздел 2) и ППО брокеров объектных запросов (ORB — Object Request Broker) (см. раздел 5.4).

Средства RPC появились в начале 80-х годов, а их главным предна-

значением было обеспечение возможности выделения части создаваемого приложения для выполнения на удаленной машине. При использовании RPC разработчик организует вызов удаленного метода программы так, как если бы код этой функции находился в локальной машине.

Использование RPC накладывает определенные ограничения на тип связи между приложениями. Дело в том, что в RPC применяется синхронный механизм взаимодействия: запрашивающее приложение выдает запрос и ждет ответа. На время ожидания приложение оказывается заблокированным Распространение Java вызвало к жизни аналог RPC для Java-приложений — RMI (Remote Method Invocation).

Следующим шагом в разработке ППО для взаимодействия между активными приложениями стали системы передачи сообщений. В основе систем передачи сообщений лежит технология очередей сообщений: приложения обмениваются информацией не непосредственно друг с другом, а используя специальные буферы (очереди). В случае необходимости обмена данными программа пересылает их в принадлежащую ей очередь и продолжает функционирование.

Технология ORB выполняет функции интеллектуального посредника, т. е. принимает запросы от клиента (клиентского приложения), осуществляет поиск и активизацию удаленных объектов, которые принципиально могут ответить на запрос, и передает ответ объектам запрашивающего приложения. ППО ORB, RPC и MOM, скрывает от пользователя процесс доступа к удаленным объектам. Запрашивающий объект должен знать имя активизируемого объекта и передать ему некоторые параметры (как правило, это информация об интерфейсе вызываемого объекта – своего рода API для ORB). Интерес к ORB связан с тем, что это ППО поддерживает объектную модель, ставшую де-факто стандартом при разработке больших программных систем. В настоящее время существуют стандарт CORBA `http://citforum.ru/database/articles/corba.shtml` и технология .NET Remoting корпорации Microsoft.

Приведенное выше ППО характеризуется двумя различными парадигмами программирования, которые иногда пересекаются — параллельное и распределенное программирование. Это два базовых подхода к достижению параллельного выполнения составляющих программного обеспечения. Методы параллельного программирования позволяют распределить работу программы между несколькими процессорами в рамках одной физической или одной виртуальной ВС. Программа, содержащая параллелизм, выполняется на одной и той же физической или виртуальной ВС.

Методы распределенного программирования позволяют распределить работу программы между несколькими процессами, причем процессы мо-

гут существовать на одной ВС, или на разных. Распределенное программирование используется для реализации параллелизма особенно если речь идет о больших и сложных программных системах. Однако не все распределенные программы по умолчанию включают параллелизм. Части распределенной программы могут выполняться по различным запросам и в различные периоды времени. При чистом параллелизме одновременно выполняемые части являются компонентами одной и той же программы. Части распределенных приложений могут реализовываться как отдельные программы. Конечно же, существуют гибридные приложения, которые являются и параллельными, и распределенными одновременно.

Вычислительная среда параллельных/распределенных приложений может включать в себя множество различных операционных систем, аппаратных платформ, коммуникационных протоколов. Общие прикладные интерфейсы ППО API позволяют реализовать взаимодействие между составными частями приложения, не вдаваясь в архитектуру вычислительной среды. Вместе с тем, изменения в инфраструктуре такой среды не требуют изменений программного кода приложения, если они не затрагивают ППО API. Кроме того, промежуточное ПО отвечает за возможность обмена разнородной информацией.

Разные типы ППО обслуживают приложения с разными требованиями к межмодульным коммуникациям. Так например, объектная ориентированность прикладных разработок и построение приложений из готовых компонентов стимулирует развитие объектных решений промежуточного слоя.

1.2 Реализуемые модели параллелизма

Параллельное программирование представляет дополнительные сложности — необходимо явно управлять работой тысяч процессоров, координировать миллионы межпроцессорных взаимодействий. Для того решить задачу на параллельной ВС, необходимо распределить вычисления между процессорами системы так, чтобы каждый процессор был занят решением части задачи. Кроме того, желательно, чтобы как можно меньший объем данных пересылался между процессорами, поскольку коммуникации намного более медленные операции, чем вычисления. Часто, возникают конфликты между степенью распараллеливания и объемом коммуникаций, то есть, чем между большим числом процессоров распределена задача, тем больший объем данных необходимо пересылать между ними. Среда параллельного программирования должна обеспечивать адекватное управление распределением и коммуникациями данных.

В параллельных вычислениях предполагают, что несколько вычислительных процессов и/или потоков могут работать одновременно т.е. параллельно. При потоковом параллелизме все потоки (поток — это последовательная программа) работают над общей памятью и не нуждаются в специальных средствах передачи данных, но должны иметь средства синхронизации. Процессный параллелизм реализуется в многопроцессорной ВС имеющей каналы для обмена данными. Процесс — именованный программный шаблон, с которого можно сделать несколько копий программы — экземпляров процессов, каждый из которых имеет свою собственную память. Обмен данными и синхронизация между экземплярами процессов происходит посредством обмена сообщениями.

Для организации высокопроизводительных вычислений рассмотрим модель архитектуры, модель вычислений, модель программирования ВС.

Модель архитектуры, которая базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. После запуска на вычислительной системе, параллельная программа представляет собой множество потоков, каждый из которых выполняет программный код, параметризованный относительно идентификатора процесса.

Модель вычислений является следующим, более высоким уровнем абстракции и представляет ещё более формальную модель соответствующей модели архитектуры. Она позволяет строить функции стоимости, отражающие времена, необходимое для выполнения алгоритма на ресурсах ВС, заданной модели архитектуры. Такая модель вычислений обеспечивает аналитический метод для проектирования и оценки параллельных алгоритмов.

Модель программирования имеет еще более высокий уровень абстракции и описывает параллельную вычислительную систему с точки зрения семантики языка программирования или программного окружения.

Упрощенная классификация схем функционирования модели архитектуры была предложена М. Флинном (M.J. Flynn) [4]. Согласно этой классификации различаются схемы.

- SIMD (Single-Instruction/Multiple-Data — архитектура с одним потоком команд и многими потоками данных). В архитектурах подобного рода сохраняется один поток команд, включающий векторные команды.

- MIMD (Multiple-Instruction/ Multiple-Data — архитектура со множеством потоков команд и множеством потоков данных). Этот класс предполагает, что в ВС есть несколько устройств обработки команд,

объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

- SISD (Single Instruction/Single Data) — одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины.

- MISD (Multiple Instruction/Single Data) — множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных.

Позже эти схемы были расширены до SPMD (Single-Program/ Multiple-Data — одна программа, несколько потоков данных) и MPMD (Multiple-Programs/Multiple-Data — множество программ, множество потоков данных) соответственно. Схема SPMD (из класса SIMD) позволяет нескольким процессорам выполнять одну и ту же инструкцию или программу при условии, что каждый процессор получает доступ к различным данным. SPMD имеет более высокий уровень абстракции и является уже моделью параллельных вычислений, которая предполагает наличие нескольких взаимодействующих процессов выполнения программы. Также можно говорить, что SPMD является и моделью параллельного программирования.

Схема MPMD (из класса MIMD) позволяет работать нескольким процессорам, причем все они выполняют различные программы или инструкции и пользуются собственными данными. Таким образом, в одной схеме все процессоры выполняют одну и ту же программу или инструкцию, а в другой все процессоры выполняют различные программы или инструкции.

Конечно же, возможны гибриды этих моделей, в которых процессоры могут быть разделены на группы, из которых одни образуют SPMD-модель, а другие — MPMD-модель вычислений. При использовании схемы SPMD все процессоры просто выполняют одни и те же операции, но с различными данными. Например, мы можем разбить одну задачу на группы и назначить для каждой группы отдельный процессор. В этом случае каждый процессор при решении задачи будет применять одинаковые правила, обрабатывая при этом различные части этой задачи. Когда все процессоры выполнят свои участки работы, получаем решение всей задачи. Если же применяется схема MPMD, все процессоры выполняют различные виды работы, и, хотя при этом все они вместе пытаются решить одну проблему, каждому из них выделяется свою часть этой задачи.

Параллельная модель программирования определяет представление программиста о параллельной ВС, определяя, тем самым возможности каким

образом разработчик может запрограммировать алгоритм. При этом существуют много различных параллельных моделей программирования для одной и той же архитектуры. Например, для одной из наиболее распространенных MIMD-архитектур может быть реализована модель вычислений как SPMD, так и MPMD с помощью моделей программирования передачи сообщений (MPI), а в рамках одного вычислительного узла и с помощью OpenMP.

Существует несколько критериев, по которым можно разделить модели параллельного программирования:

- уровень параллелизма, который используется при параллельном выполнении (уровень инструкций, уровень команд, уровень функций, уровень объектов, уровень компонентов или параллельных циклов);
- неявная или явная, определённая пользователем спецификация параллелизма;
- каким образом определены параллельные части программы;
- способ выполнения параллельных подзадач (SPMD, синхронный или асинхронный);
- способы коммуникации между вычислительными подзадачами для обмена информацией (явная коммуникация или разделяемые переменные);
- механизмы синхронизации для обеспечения вычислений и связи между параллельными подзадачами.

Модели программирования могут содержать различные варианты удовлетворяющие тем или иным приведенным критериям. В зависимости от модели программирования параллельные вычисления могут быть определены на разных уровнях:

- последовательностью инструкций, выполняющих арифметические или логические операции;
- последовательность операторов, где каждый оператор может включать несколько инструкций;
- вызов функции или метода, который как правило состоит из нескольких операторов.

Много параллельных моделей программирования обеспечивают понятие параллельных циклов; итерации в параллельных циклах независимы друг из друга и поэтому могут быть выполнены параллельно.

На этом этапе необходимо определить наиболее подходящую модель программной реализации параллельных вычислений (задачи и каналы, пересылка сообщений с использованием стандартных библиотечных утилит, разделяемая память с использованием блокировок и семафоров и др.). В модели разделяемой памяти с использованием блокировок и семафоров задачи используют общее адресное пространство, в котором они читают и записывают данные асинхронно. Механизм блокировок и семафоров используется для контроля доступа к общей памяти. Другие примеры моделей параллельного программирования включают в себя: алгоритмические скелетоны; компоненты; распределённые объекты; потоковая обработка; удаленный вызов процедур; рабочие процессы (workflows) и др.

Ниже будут проанализированы различные системы промежуточного ПО и сформулирован ряд требований, выдвигаемых к программной модели высокопроизводительной ВС.

Рассмотрим только основные модели параллельного программирования [4].

- Параллелизм на уровне инструкций или параллелизм на уровне команд. В этой модели программу можно рассматривать как поток инструкций выполняемых процессором. Инструкции возможно распределить по группам, которые будут выполняться параллельно, без изменения результата работы всей программы.

- Параллелизм задач (`Task Parallel`). Модель программирования, которая подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Тем самым увеличивается общая эффективность программы, основанной на параллелизме задач.

- Параллелизм данных (`Data Parallel`). В этой модели единственная программа задает распределение данных между всеми процессорами компьютера и операции над ними. Распределяемыми данными обычно являются массивы. Как правило, языки программирования, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы, вырезки из массивов. Распараллеливание операций над массивами, циклов обработки массивов позволяет увеличить производительность програм-

мы. Компилятор отвечает за генерацию кода, осуществляющего распределение элементов массивов и вычислений между процессорами. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти. Программы с параллелизмом данных могут быть оттранслированы и исполнены как на MIMD, так и на SIMD ВС.

- Обмен сообщениями (`Message Passing`). В этой модели возможно различные программы, написанные на традиционном последовательном языке программирования исполняются процессорами компьютера. Каждая программа имеет доступ к памяти исполняющего её процессора. Программы обмениваются между собой данными, используя подпрограммы приема/передачи данных некоторой коммуникационной системы. Программы, использующие обмен сообщениями, могут выполняться на MIMD-архитектуре ВС.

- Общей памяти (`Shared Memory`). В этой модели все процессы совместно используют общее адресное пространство. Процессы асинхронно обращаются к общей памяти как с запросами на чтение, так и с запросами на запись, что создает проблемы при выборе момента, когда можно будет поместить данные в память, когда можно будет удалить их. Для управления доступом к общей памяти используются стандартные механизмы синхронизации: семафоры и блокировки процессов.

1.2.1 Модель обмен сообщениями

На сегодняшний день модель обмен сообщениями (message passing) является наиболее широко используемой моделью параллельного программирования. Программы этой модели создают множество процессов, с каждым из которых ассоциированы локальные данные. Каждый процесс идентифицируется уникальным именем. Процессы взаимодействуют, посылая и получая сообщения. В этом отношение модель обмен сообщениями является разновидностью модели процесс/канал и отличается только механизмом, используемым при передаче данных.

Модель обмен сообщениями не накладывает ограничений ни на динамическое создание процессов, ни на выполнение нескольких процессов одним процессором, ни на использование разных программ для разных процессов. Просто, формальные описания систем обмена сообщениями не рассматривают вопросы, связанные с манипулированием процессами, Однако, при реализации таких систем приходится принимать какое-либо ре-

шение в этом отношении. На практике сложилось так, что большинство систем обмена сообщениями при запуске параллельной программы создает фиксированное число идентичных процессов и не позволяет создавать и разрушать процессы в течение работы программы (см. раздел 2).

В системах передачи сообщений каждый процесс выполняет программный код, параметризованный относительно идентификатора процесса и коммуникатора. С использованием этих параметров может реализовываться как SPMD, так и MPMD модель вычислений.

Фактически стандартизованный механизм для построения параллельных программ в модели обмена сообщениями реализован в MPI (раздел 2).

1.2.2 Модель «общая память»

В модели программирования с общей памяти все процессы совместно используют общее адресное пространство, к которому они асинхронно обращаются с запросами на чтение и запись. В таких моделях для управления доступом к общей памяти используются всевозможные механизмы синхронизации типа семафоров и блокировок процессов. Преимущество этой модели, с точки зрения программирования, состоит в том, что понятие собственности данных (монопольного владения данными) отсутствует, следовательно, не нужно явно задавать обмен данными между производителями и потребителями. Эта модель, с одной стороны, упрощает разработку программы, но, с другой стороны, затрудняет понимание и управление локальностью данных, написание детерминированных программ. В основном, эта модель используется при программировании для архитектур с общедоступной памятью. Модель общая память поддерживает SPMD и MPMD модели программирования. Стандарт для программирования в модели общей памяти система OpenMP (см. раздел 3).

1.2.3 Модель «параллелизм данных»

Модель параллелизм данных также является часто используемой моделью параллельного программирования. Название модели происходит оттого, что она эксплуатирует параллелизм, который заключается в применении одной и той же операции к множеству элементов структуры данных. Поскольку операции над каждым элементом данных можно рассматривать как независимые процессы, то степень детализации таких вычислений очень велика. Программа, имеющая параллелизм данных, состоит из последовательностей подобных операций (сравнительно небольших подпрограмм, что следует из мелкозернистости модели). В этом случае, локальные копии данных существуют лишь на время выполнения указанной выше

операции, в отличие от модели обмена сообщениями, где данные инкапсулируются в процессы, существующие до завершения работы приложения. Поэтому, говоря о локализации данных в модели параллелизм данных, необходимо рассматривать время жизни данных.

Следовательно, компиляторы языков с параллелизмом данных часто требуют, чтобы программист предоставил информацию относительно того, как данные должны быть распределены между процессорами, другими словами, как данные будут распределены между задачами.

По определению, данная модель параллелизма предполагает SPMD модель программирования. Паралleизм данных — основная модель определяющая дизайн CUDA (см. раздел 4), OpenCL (см. раздел 5.1).

Для распараллеливания может быть использовано несколько программных моделей (см. разделы 5.2,5.3,5.4.2), каждая из которых соответствует определенному подмножеству требований. Заранее может быть трудно определить, какая именно модель лучше подходит. Еще больше сложностей возникает в связи с потребностью комбинировать различные модели.

1.3 Предъявляемые требования

Анализ существующего промежуточного программного обеспечения позволяет выделить ряд основных свойств технологий высокопроизводительных вычислений. Возможно, этот список неполный. Несомненно, он будет расширяться в связи с развитием вычислительной техники и технологий программирования.

1.3.1 Инвариантность к языкам программирования

Инвариантность к языкам программирования можно считать основным признаком технологии программирования, когда прикладная программа может быть написана на любом удобном для разработчика языке. Во всех системах языконезависимость достигается двумя способами: путем реализации интерфейса системы на различных языках программирования и путем введения специальных высокоуровневых описаний, которые затем транслируются на различные языки.

1. Реализация интерфейса системы на различных языках программирования. Такой подход применяется, например, в технологии MPI. Интерфейс MPI реализовывается только на языках C, Fortran. Недостаток такого подхода: несовершенный интерфейс. В связи с развитием языков программирования, в частности, с появлением объектно-ориентированных языков, возникла необходимость записать интер-

фейс MPI совершенно иным способом. Поэтому сначала появляются интерфейсы MPI++, OOMPI, в которых сама промежуточная среда представляется в виде объектов, затем создаются описания параллельных объектов TPO++ (см. раздел 2, интерфейсы для работы с компонентами [5].

2. Введение специальных высокоуровневых описаний. Так, в технологии CORBA [6] центральное место занимает язык описания интерфейсов IDL, который эволюционирует вместе с самой системой. Для того чтобы расширить языковую поддержку CORBA, необходимо реализовать транслятор, переводящий IDL-описания в текст на целевом языке. IDL-описания используются не только для получения исходного кода программы, Разные типы ППО обслуживают приложения с разными требованиями к межмодульным коммуникациям. Объектная ориентированность прикладных разработок и построение приложений из готовых компонентов стимулирует развитие объектных решений промежуточного слоя. Кроме этого, выделение языка как части системы позволяет CORBA не отставать от развития языков программирования.

1.3.2 Платформонезависимость

Платформонезависимость также можно считать основным признаком технологии вычислений. Как показал анализ, независимость от компьютерной платформы может достигаться на уровнях исходного кода, промежуточной среды, бинарного кода.

1. Переносимость на уровне исходного кода: в данном случае все прикладные программы переносимы, но для этого требуется их перекомпиляция на новой платформе. Взаимодействовать друг с другом компоненты разных платформ не могут. Перенос обеспечивается единым интерфейсом на общем языке программирования. Процесс усложняется или вообще невозможен, если используются библиотеки, связанные с той или иной платформой. MPI (Message Passing Interface) `http://www.mpi-forum.org/` — пример системы ППО, платформонезависимой на уровне исходного кода.

2. Переносимость на уровне промежуточной среды: переносимость на уровне исходного кода и, кроме того, программы, работающие на разных платформах, могут взаимодействовать друг с другом. Функционирование многокомпонентной системы обеспечивается единым интерфейсом, причем использование общего языка программирования

не обязательно. В качестве примеров можно привести надстройку над различными системами MPI для разных платформ, реализующую общую промежуточную среду, и технологию CORBA.

3. Переносимость на уровне бинарного кода: переносимы откомпилированные прикладные программы. Промежуточная система, реализующая такой подход, включает в себя некую виртуальную машину для выполнения программ. Такого рода перенос возможен между различными версиями ОС Windows посредством языконезависимой технологии COM и между ОС, оснащенными Java-системой. В данном случае узким местом является виртуальная машина: необходимо, чтобы она была реализована для широкого класса ОС, была достаточно производительной и использовала наработанное ПО для данной ОС.

Наиболее приемлемыми являются системы, платформонезависимые во втором смысле, такие как CORBA, OpenCL. Их преимущество — эффективное взаимодействие с платформой, т.к. доступ к ней осуществляется без посредников. Платформа понимается широко: операционная система, созданное на ее основе программное обеспечение, аппаратные средства. В данном случае может оказаться полезной некоторая зависимость от платформы. С другой стороны, очевидной становится проблема управления программными компонентами в гетерогенной аппаратно-программной среде. Чтобы решить эту проблему, можно расширить функциональность CORBA Application Server путем внедрения компиляторов, интерпретаторов и библиотек к ним, что позволит управлять исходным кодом компонентов.

1.3.3 Адаптация к различным архитектурам аппаратных средств

Промежуточное программное обеспечение позволяет абстрагироваться от аппаратных средств, поэтому его введение чревато неэффективным использованием компьютерных ресурсов. Важно, чтобы на первом месте стоял не стандартный программный интерфейс, а максимальная производительность.

ППО MPI (см. раздел 2) включает описания процессоров и их различных топологий, оптимизирует вычисления для виртуальных топологий, задаваемых программно. Это позволяет более эффективно отобразить прикладную задачу на сеть вычислительных узлов.

ППО OpenCL (см. раздел 5.1) для написания программ, связанных с параллельными вычислениями на различных графических (GPU) и центральных процессорах (CPU). В OpenCL входят язык программирования,

который базируется на стандарте C99, и интерфейс программирования приложений (API).

В последнее время в распределенные системы включаются элементы, позволяющие учитывать архитектуру ЭВМ. Кроме этого, в компонентные технологии включаются системы реального времени, позволяющие эффективно управлять компонентами, находящимися на разных узлах.

Анализ показал, что самыми перспективными являются технологии с гибкими расширяющимися интерфейсами. Это позволяет не только адаптировать многокомпонентные системы к возможностям ЭВМ, но и создавать программные средства для этого.

1.3.4 Использование различных моделей распараллеливания

Широкий набор описаний процессов и данных позволяет точно записать задачу в терминах данной технологии. Многие большие задачи подразумевают массивный параллелизм и требуют значительных вычислительных ресурсов. От того, насколько правильно описаны параллельные процессы и распределенные данные, будет зависеть производительность вычислительной системы.

Эволюция параллельных и распределенных систем, несмотря на то, что они использовались для решения определенных классов задач, показывает, что сегодня высокопроизводительная технология должна описывать и SIMD, и MISD, и MIMD модели. Сегодня наметилась тенденция на расширение описательных возможностей существующих стандартов (для MPI разрабатываются технологии распределенных объектов, для CORBA — дополнительные модули параллельной обработки). Такие системы должны строиться на основе компонентного подхода с широким использованием различных методов взаимодействия между процессами (обмены) и доступа к объектам (клиент-сервер).

1.3.5 Объектно-ориентированный подход

Объектно-ориентированный стиль моделирования и программирования систем в настоящее время является наиболее удобным и эффективным для реализации программных систем [7]. Объектно-ориентированный подход (ООП) основан на систематическом использовании моделей для языково-независимой разработки программной системы.

Модель содержит не все признаки и свойства представляемого ею предмета, а только те, которые существенны для разрабатываемой программной системы. Таким образом, модель есть формальная конструкция: формальный характер моделей позволяет определить формальные зависимо-

сти между ними и формальные операции над ними. Это упрощает как разработку и анализ моделей, так и их программную реализацию. В частности, формальный характер моделей позволяет получить формальную модель разрабатываемой программной системы как композицию формальных моделей её компонентов. Поэтому, объектно-ориентированный подход помогает справиться с такими проблемами, как:

- уменьшение сложности программного обеспечения;
- повышение надежности программного обеспечения;
- модификация отдельных компонентов программного обеспечения без изменения остальных его компонентов;
- повторное использование отдельных компонентов программного обеспечения.

В соответствии с объектно-ориентированной технологии разработка приложений происходит в три фазы: анализ, проектирование и реализация. Во время фазы анализа разрабатываются три модели: объектная, динамическая и функциональная. Фаза проектирования опирается на эти модели. Объектная модель определяет компоненты данных, уместные для конкретного приложения, их атрибуты и характерные для них операции. Динамическая модель определяет последовательность операций на основе диаграмм состояний. Функциональная модель использует диаграммы потоков данных для определения входных и выходных данных, источников и приемников данных, расположения мест передачи и приема данных. В сущности, объектная модель описывает данные, в то время как две другие модели описывают аспекты, связанные с параллельными процессами. Аспекты, включаемые в функциональную модель, связаны с передачей сообщений и параллельным выполнением (см. разделы 2.6, 3.4, 4.2, 5.1).

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования. Объектно-ориентированный подход является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

ООП оказал существенное влияние как на интерфейс, так и на структуру ВС. Эволюция параллельных и распределенных вычислений привела к оформлению объектной структуры высокопроизводительной системы, к описанию обобщенных объектных интерфейсов.

1.3.6 Компонентный подход

На основе ООП развивается компонентный подход, в котором этапы проектирования и программирования четко разграничены. Компонентом называют объект, в котором разделены интерфейс и реализация. Это позволяет реализовать ее одновременно на разных платформах. Прикладное обеспечение, созданное на основе такой системы, может функционировать на многих платформах. Как компонентная технология с самого начала проектировалась CORBA. Показательно, что для системы MPI в ходе эволюции были определены объектная структура, интерфейс работы с объектами, компонентная модель. Компонентный подход дает много преимуществ по разработке программ, которые могут быть автоматизированы, поэтому важно создать соответствующие средства разработки. В настоящее время наметилась тенденция к созданию интегрированных сред разработки, направленных не только на создание программ на каком-либо языке программирования, но и поддерживающих различные технологии программирования. Можно выделить ряд необходимых свойств среды разработки:

- наличие инструментов объектного моделирования (например, Rational Rose);
- поддержка компонентной технологии (например: CORBA [6], ICE [8]);
- тесная интеграция всех инструментов для повышения эффективности программирования семантики компонентов;
- возможность групповой работы над проектами, для чего создаются CVS-системы (Control Version System — система контроля версий);
- ведение библиотек компонентов для их повторного использования (например репозитории CORBA);
- наличие хранилища шаблонов и мастеров для автоматизации создания компонентов,
- открытость, т.е. возможность включения в среду разработки новых компонентов (компиляторов, скриптов, редакторов и др.).

Среду разработки, удовлетворяющую данным требованиям имеет смысл создавать на основе существующих инструментов, таких как некоторые компоненты CORBA (IDL-компилятор, репозитории объектов, сервис поддержки жизненного цикла и др.), языков управления сценариями Python, Perl, Tcl.

1.3.7 Инкапсуляция и интеграция существующих программных систем

Функциональные возможности и скорость разработки прикладных программных систем напрямую зависят от технологий, которые лежат в их основе. Если технологии имеют эффективные механизмы повторного использования программ (объектно-ориентированный и компонентный подход), то появляется возможность инкапсуляции и интеграции существующих вычислительных систем.

1. Инкапсуляция — это включение в состав многокомпонентной системы набора компонентов некоторой ВС. Это позволяет повторно использовать программные компоненты. В тех случаях, когда ВС, которую необходимо инкапсулировать, основана на другой технологии, создаются специальные механизмы, например, программы каркасы между компонентными системами COM-CORBA, Java-COM, CORBA-RMI. Инкапсуляцию существующих ВС может осуществить прикладной программист.

2. Интеграцией называется инкапсуляция с одновременным изменением архитектуры исходной системы. Интеграцией MPI в CORBA можно назвать проект Cobra [5], когда модифицируются важные составляющие CORBA. Это относится к области системного программирования. Перед тем как браться за интеграцию, необходимо оценить преимущества и недостатки видоизменения базовой ВС.

Создание ВС является комплексной задачей, для решения которой необходимо решить ряд проблем, не связанных с основными вычислительными целями. Поэтому для увеличения скорости разработки ВС полезно инкапсулировать существующие системы визуализации данных, CAD-системы, некоторые стандартные математические библиотеки.

Интеграция в ВС каких-либо систем имеет смысл, когда на базе этих систем разработаны необходимые библиотеки. Интеграция полезна также при появлении в базовой ВС новых моделей распараллеливания. Сборка интегрированной системы является эффективным способом создания проблемно-ориентированных сред, объединяющих вычислительные системы различных дисциплин в единый программный комплекс.

1.3.8 Система управления

Рассмотрение компонентных систем показало значение компонентной модели и системы реального времени для управления программными ком-

понентами. Мало спроектировать и запрограммировать многокомпонентную систему, необходимо их установить и отследить во время выполнения. Если процесс разработки компонента определяется описанными выше характеристиками ВС, то, что можно сказать о процессе его внедрения и использования? Этот процесс можно и нужно формализовать.

Для описания эксплуатации компонентов необходимо определить структуру компонента, которая задается компонентной моделью, и основные действия над ним, которые будут выполняться системой реального времени. Для реализации такого подхода используется следующий прием.

1. Определяется интерфейс компонента, включающий наиболее основные и общие свойства.

2. Определяется интерфейс сервиса, обслуживающего компоненты данного класса. Сюда включаются основные методы управления компонентами: отслеживание состояния, активация, деактивация и т.д.

3. Сервис реализуется и включается в компонентную систему. Создается средство интерактивного взаимодействия с сервисом.

Средства управления играют самую значительную роль в процессе использования компонентов, позволяя конфигурировать их в соответствии с архитектурой аппаратной среды.

Таким образом, в ходе рассмотрения существующих на данный момент ППО выделены необходимые характеристики высокопроизводительной программной системы и определены основные технические стандарты, на основе которых она может быть построена. Рассмотрим основные из них более подробно.

1.3.9 Особенности программирования для многоядерных вычислительных систем

Преимущество нынешних и будущих многоядерных процессоров по сравнению с обычными многопроцессорными системами состоит в том, что существенная часть работы, связанная с коммуникациями и синхронизацией, решается на аппаратном уровне, поэтому на программное обеспечение выпадает необходимость распараллеливания не на самом нижнем уровне грануляции (это в некотором смысле достаточное условие для распараллеливания приложений).

Переход на многоядерные системы подразумевает существенное изменение парадигмы программирования: создание параллельных потоков, порождение и обработка асинхронных событий и ряда других. Особенности

перехода к параллельным вычислениям на многоядерных системах заключаются в следующем.

1. При переходе с одноядерных процессоров на многоядерные системы приходится принимать во внимание проблему последовательного выполнения. В многоядерной системе выполнение считается последовательным, когда одно или более ядер в какой-то момент не могут выполнять код параллельно с другими ядрами. Эта ситуация может возникнуть по разным причинам: ситуации блокировки при доступе к ресурсам, необходимость синхронизации процессов или потоков на различных ядрах, поддержание когерентности кэш-памяти и неравномерность загрузки.

2. Блокировки возникают из-за невозможности одновременного доступа приложений на разных ядрах к таким ресурсам, как жёсткий диск, некоторые устройства ввода/вывода, прикладным данным в определённых ситуациях (например, в момент «сборки мусора»).

3. Очень часто параллельные процессы, выполняемые на разных ядрах, должны синхронизироваться в определённые моменты времени. Например, приложение на одном из ядер должно использовать промежуточные данные, которые получаются приложением (потоком, процессом) на другом ядре. Первое приложение не может продолжать работу до тех пор, пока не получит эти данные, то есть должно находиться в состоянии ожидания до их получения. В этой ситуации появляются накладные расходы на синхронизацию приложений (процессов, потоков), выполняемых на разных ядрах.

4. При переходе на многоядерную архитектуру возникает необходимость поддержания когерентности (согласованности) кэш-памяти для всех ядер при использовании разделяемой памяти. В каждый момент времени необходимо поддерживать согласованность основной памяти и кэш-памяти всех ядер, использующих эту разделяемую память при любых операциях чтения-записи. Как правило, эта проблема решается на аппаратном уровне.

Применение многоядерных процессоров позволяет получать более совершенные распределения процессов для задействования ядер с целью минимизации конфликтов в кэш-памяти, оперативной памяти, процессоре и т.д. Для этого пользователь должен иметь возможность определения плана размещения и выполнения задания на вычислительных узлах, процессорах и ядрах. Такую возможность предоставляет привязка процессов к

ресурсам вычислительной системы (вычислительным узлам, физическим и логическим процессорам, оперативной памяти) [9].

Привязка процессов к аппаратному обеспечению

Привязка позволяет сохранить местоположение данных и не допускает миграцию процесса далеко от кэша, который содержит его данные. Для предотвращения перемещения процесса или потока, будем связывать процесс с логическими процессорами. В том случае, когда привязка не используется, операционная система может переместить процесс или поток на другой процессор по причине разбалансировки нагрузки и тем самым переместить процесс далеко от его данных. Процесс привязки позволяет получать более устойчивые показатели производительности, потому, что перемещение процесса требует аннулирования кэша и может ухудшить производительность приложений.

В современных вычислительных системах, как правило, применяются многоядерные (до 12-ти ядер), многосокетовые (до 8-х сокетов) вычислительные узлы, использующие стандартные комплектующие для построения высокопроизводительных систем. Данные системы характеризуются многоуровневой организацией коммуникаций и многоуровневым доступом к оперативной памяти. С увеличением количества процессорных ядер в значительной степени возрастает и сложность платформы. При этом некоторые ресурсы дублируются, а некоторые используются совместно. В NUMA-архитектуре разделение данных между узлами может быть весьма затратным по причине удаленного доступа к памяти. Это также следует принимать во внимание при миграции потоков, особенно между разными узлами, так как в случае кэш-промахов потребуется доступ к памяти другой группы ядер.

Сложность состоит в распределении потоков и процессов между ядрами таким образом, чтобы оптимально задействовать возможности системной топологии и архитектуры. В большинстве случаев планировщики в операционных системах (ОС) общего назначения назначают потоки ядрам с наименьшей загрузкой, чтобы обеспечить сбалансированную нагрузку, либо ядрам с уже подготовленными данными в кэше (cache-warm), используя преимущества привязки к кэшу (cache affinity).

Как правило, на многосокетовой/многоядерной архитектуре параллельная программа запускается таким образом, что несколько процессов выполняются на одном вычислительном узле и связываются с множеством процессов на других вычислительных узлах по коммуникационной сети. Поэтому, алгоритмы коммуникаций (особенно коллективных) должны быть

построены и реализованы так, чтобы эффективно использовать такую топологию коммуникаций. Различные реализации MPI содержат свои механизмы оптимизации выполнения межузловых и внутри узловых коммуникаций (алгоритмы, функции, параметры запуска приложения, подробнее см. раздел 2.4). И здесь к факторам имеющим существенное значение можно отнести расположение процессов/ядер на вычислительных узлах. Для получения желаемого размещения приложения необходимо знать отображение ядер на используемой вычислительной системе.

Операционная система рассматривает ядра процессоров, как «процессоры» (processor на рисунках 1 и 2). По аналогии с логическими жесткими дисками назовем ядра процессоров логическими CPU. Данное понятие отличается от понятия виртуальные процессоры, число которых полагается равным числу аппаратно поддерживаемых потоков (как правило, два потока на каждое ядро процессора). Например, в процессорах Intel Core i3 и Core i7 используется технология гиперпоточности (Hyper-Threading Technology), в этом случае каждое физическое ядро процессора определяется операционной системой как два логических (виртуальных).

Нумерация ядер в ОС изменяется для разных систем и ее нельзя наследовать для достижения оптимальных результатов на всех вычислительных системах. Например, для двух систем, на которых проводилось тестирова-

processor	0	1	2	3	4	5	6	7
model name	Intel(R) Xeon(R) CPU X5365 @ 3.00GHz							
cpu MHz	3000.111							
cache size	4096 KB							
physical id	0	1	0	1	0	1	0	1
core id	0	0	2	2	1	1	3	3
cpu cores	4							

Рис. 1. Вычислительный узел МВС-100К

processor	0	1	2	3	4	5	6	7
model name	Intel(R) Xeon(R) CPU E5430 @ 2.66GHz							
cpu MHz	2660.073							
cache size	6144 KB							
physical id	0	0	1	1	0	0	1	1
core id	0	2	0	2	1	3	1	3
cpu cores	4							

Рис. 2. Вычислительный узел кластера X4

ние МВС-100К МСЦ РАН (каждый ВУ — два четырехядерных процессора Xeon 5365, 3GHz) и кластер X4 ИМ УрО РАН (каждый ВУ — два четырехядерных процессора Xeon 5420, 2.66GHz) приведены данные на рисунках 1 и 2. Строка «processor» содержит номера логических CPU, «core id» — идентификаторы ядра в физических (реальных) процессоров («physical id»). Соответствие логических CPU ядрам вычислительных узлов для них показано на рисунках 3,4. Представленные вычислительные узлы идентичны по числу и архитектуре процессоров, но на уровне операционной системы имеют разные номера ядер. В данном случае, процессы/потоки, выполняемые нулевым и первым логическими CPU на узле кластера X4

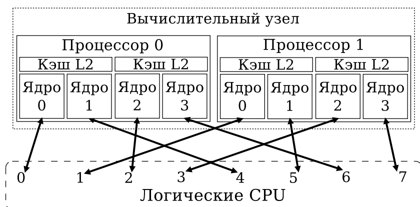

Рис. 3. Соответствие логических CPU ядрам вычислительных узлов МВС-100К

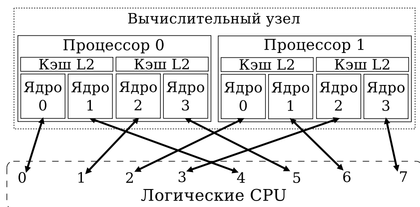

Рис. 4. Соответствие логических CPU ядрам вычислительных узлов кластера X4

будут выполнятся одним физическим процессором, конкурируя за один канал оперативной памяти, а на МВС-100К — двумя процессорами без конкуренции. Пары процессов/потоков, выполняемые на логических CPU с номерами 0 и 4, 1 и 5, 2 и 6, 3 и 7 на обеих вычислительных системах будут конкурировать за кэш-память второго уровня.

Эти особенности необходимо учитывать, как при программировании, так и при запуске программ на выполнение.

Рисунки 1 – 4 построены на основе информации из файла `cpuinfo`, размещаемого в каталоге `proc` корневого каталога Linux. Графические изображения аппаратной архитектуры вычислительного узла также могут быть построены при помощи утилиты `lstopo`.

Отметим, что при выполнении привязки процессов к ресурсам вычислительной системы следует учитывать комбинацию аппаратного обеспечения, системного и промежуточного программного обеспечения.

Высокоуровневые системные вызовы привязки

Как было отмечено ранее, операционная система определяет нумерацию ядер для вычислительных узлов (ВУ) с многоядерными процессорами. В операционной системе Linux пользователь может привязать процесс к логическому CPU, используя вызов функций `sched_setaffinity()`, `sched_getaffinity()` с соответствующим параметром `affinity_mask` из текущего процесса (своей программы). Эти функции могут отличаться типом параметров в зависимости от производителей Linux и версий библиотеки Glibc. Реализованы функции в библиотеке PLPA (The Portable Linux Processor Affinity `http://www.open-mpi.org/projects/plpa`).

Привязка процесса к определенному CPU (`cpu_affinity`) исключает его миграцию между процессорами, а также позволяет изменить для него

алгоритм работы планировщика задач и увеличить приоритет.

Готовая к выполнению задача (приложение) также может быть привязан к логическим CPU в Linux, при помощи команды `taskset`.

Отметим также, что для привязки процессов в других операционных системах также имеются системные вызовы: в операционной системе Windows — `SetThreadAffinityMask()`; в Solaris — `pbind()`.

Другие способы привязки параллельных процессов и потоков рассматриваются более подробно в разделах 2.4.1, 3.5.

Новая многоядерная архитектура требует смены программной парадигмы — перехода от параллельного стиля программирования к многоядерному. Таким образом, новое направление развития параллельных архитектур в реальности оборачивается переносом проблем, ранее известных лишь узкому кругу специалистов, в широкие массы прикладных программистов и вычислителей.

Рассмотрим возможные подходы построения и оптимизации параллельных программ для многоядерных вычислительных систем на основе широко используемого промежуточного программного обеспечения для одной из базовых задач вычислительной математики — матрично-векторного произведения.

Обратимся к задаче, необходимость распараллеливания которой нередко возникает при решении реальных вычислительных задач, например при решении итерационными методами систем линейных алгебраических уравнений (СЛАУ). В таких задачах наиболее ресурсоёмкой является операция умножения матрицы на вектор, поэтому именно её распараллеливание в наибольшей степени повышает скорость вычислений.

1.4 Пример распараллеливания матрично-векторного произведения

Матрицы и матричные операции широко используются при математическом моделировании разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчётов — среди областей приложений могут быть указаны вычислительная математика и механика и др. Одним из примеров применения матричных вычислений является решение систем линейных алгебраических уравнений (СЛАУ), которые, в свою очередь, занимают одну из основных ролей при использовании численных методов и математического моделирования [10]. Другие интересные задачи и примеры распараллеливания можно найти в [11].

Построение эффективных алгоритмов умножения разряженной матрицы на вектор (Sparse Matrix-Vector Multiplication — SpMV) затруднено тем, что для представления разряженных матриц используется сложная структура данных, и шаблон доступа к памяти не предсказуем. Способ вычисления произведения на любой архитектуре ВС зависит от формата представления разреженной матрицы, который, в свою очередь, связан со спецификой рассматриваемой задачи.

Вычислим произведение матрицы $\mathbf{A} \in \mathbb{R}^{N\times M}$ и вектора $\mathbf{b} \in \mathbb{R}^M$. (Здесь и далее жирным шрифтом будем выделять матрицы и вектора, в остальных случаях — скалярные величины.)

Последовательный алгоритм матрично-векторного произведения имеет вид:

$$c_i = \sum_{j=1}^{M} a_{ij} b_j, \quad i = 1, 2, \dots, N,$$

где $\mathbf{c} = (c_1, c_2, \dots, c_N) \in \mathbb{R}^N$, $\mathbf{A} = ((a_{ij})_{i=1,\dots,N, j=1,\dots,M})$, $\mathbf{b} = (b_1, b_2, \dots, b_M)$ и может быть реализован двумя различными способами в зависимости от переменной цикла.

В первом случае, умножение вычисляется как N скалярных произведений строк $(a_1, a_2, \dots, a_N)$ матрицы $\mathbf{A}$ и вектора $\mathbf{b}$:

$$\mathbf{A} \cdot \mathbf{b} = \begin{pmatrix} (\mathbf{a}_1, \mathbf{b}) \\ \vdots \\ (\mathbf{a}_N, \mathbf{b}) \end{pmatrix}$$

здесь скалярное произведение $(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{M} x_j y_j$ векторов $\mathbf{x}, \mathbf{y} \in \mathbb{R}^M$ с компонентами $\mathbf{x} = (x_1, x_2, \dots, x_M)$ и $\mathbf{y} = (y_1, y_2, \dots, y_M)$.

Соответствующий алгоритм на языке С можно представить как

```
1 for (i = 0; i < N; i++) c[i] = 0;
2 for (i = 0; i < N; i++)
3     for (j = 0; j < M; j++) c[i] = c[i] + A[i][j]*b[j];
```

Матрица $\mathbf{A} \in \mathbb{R}^{N\times M}$ представляется как двумерный массив, а соответствующие вектора как одномерные массивы. Для каждого внутреннего цикла $i = 0, 1, \dots, N-1$ имеется вложенный цикл содержащий одно скалярное произведение.

Во втором случае, произведение матрицы на вектор может быть записано как линейная комбинация столбцов $\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2, \dots, \tilde{\mathbf{a}}_M$ матрицы $\mathbf{A}$ с коэф-

фициентами $b_1, b_2, \ldots, b_M$

$$\mathbf{A} \cdot \mathbf{b} = \sum_{j=1}^{M} b_j \tilde{\mathbf{a}}_j.$$

Соответствующий программный код имеет вид:

```
1 for (i = 0; i < N; i++) c[i] = 0;
2 for (j = 0; j < M; i++)
3     for (i = 0; i < N; i++) c[i] = c[i] + A[i][j]*b[j];
```

Для последовательных вычислений обе эти программы одинаковы, т.к. нет зависимости от порядка переменных цикла. Для параллельных вычислений представление матрицы по строкам или столбцам приводит к двум различным алгоритмам:

- при строчно-ориентированном представлении матрицы $\mathbf{A}$, вычисляются N скалярных произведений $(\tilde{\mathbf{a}}_i, \mathbf{b}), i = 1, 2, \ldots, N$ строк матрицы $\mathbf{A}$ с вектором $\mathbf{b}$ параллельно. Каждый процессор из числа доступных процессоров ВС n_p обрабатывает N/n_p скалярных произведений;

- при столбцово-ориентированном представлении матрицы $\mathbf{A}$, вычисления линейных комбинаций $\sum_{j=1}^{M} b_j \tilde{\mathbf{a}}_j$ происходят на каждом процессоре над частью линейных комбинаций M/n_p векторов.

Рассмотри эти алгоритмы более подробно.

1.4.1 Параллельное вычисление скалярных произведений

Для реализации параллельного произведения на ВС с распределённой памятью необходимо распределить матрицу $\mathbf{A}$ и вектор $\mathbf{b}$, так чтобы вычисления скалярных произведений $(\mathbf{a}_i, \mathbf{b}), i = 1, 2, \ldots, N$ выполнялись над данными в памяти данного процессора, т.е. соответствующие строки $\mathbf{a}_i$ и вектора $\mathbf{b}$ хранились в памяти процессора вычисляющего процессора. Т.к. вектор $\mathbf{b}$ необходим для всех скалярных произведений, поэтому $\mathbf{b}$ просто дублируется.

При строчно-ориентированном распределении матрицы $\mathbf{A}$, процессор P_k, $k = 1, 2, \ldots, n_p$, хранит строки матрицы $\mathbf{a}_i$, $i = N/n_p \cdot (k-1), N/n_p \cdot (k-1) + 1, \ldots, N/n_p \cdot k$, в локальной памяти и вычисляет скалярные произведения $(\mathbf{a}_i, \mathbf{b})$. Для вычисления $(\mathbf{a}_i, \mathbf{b})$ нет необходимости в пересылке и приёме данных с других процессоров. Результат произведения вектор $\mathbf{c} = (c_1, c_2, \ldots, c_N)$ будет распределён по блокам.

Чаще всего в алгоритмах использующих параллельные скалярные произведения, требуется чтобы вектор **c** имел распределение по процессорам такое же как вектор **b**.

1.4.2 Параллельное вычисление линейных комбинаций

Для параллельных вычислений матрично-векторного умножения на ВС с распределённой памятью в форме линейных комбинаций, используется распределение матрицы **A** по столбцам. Каждый процессор вычисляет часть линейных комбинаций для соответствующих столбцов $\tilde{\mathbf{a}}_i$ $(i \in 1, 2, \ldots, M)$. Каждому процессору P_k распределяется часть столбцов $\tilde{\mathbf{a}}_i$ $(i = M/n_p \cdot (k-1) + 1, M/n_p \cdot (k-1) + 2, \ldots, k \cdot M/n_p)$ и вычисляется вектор размерности N как:

$$\mathbf{d}_k = \sum_{j=M/n_p \cdot (k-1)+1}^{M/n_p \cdot k},$$

где $k = 1, 2, \ldots, n_p$. Отметим, что на каждом процессоре хранится только часть вектора **b**. По окончанию параллельных вычислений вектора $\mathbf{d}_k$, хранимые в локальной памяти каждого процессора, должны быть просуммированы в результирующий вектор $\mathbf{c} = \sum_{k=1}^{n_p} \mathbf{d}_k$.

1.4.3 Оценка времени параллельного выполнения

Время необходимое для параллельного выполнения программы зависит:

- от размера входных данных и параметров алгоритма. Например, границы циклов или число итераций;
- числа используемых процессоров n_p;
- коммуникационных характеристик и, прежде всего, вида коммуникаций в ВС и ППО обеспечивающего коммуникации.

Рассмотрим простейшую параметрическую оценку модели производительности алгоритма матрично-векторного произведения, применительно к квадратной, заполненной матрице **A**. Выражение для времени работы и параллельного ускорения через параметры задачи (N) и характеристики архитектуры (n_p — число процессоров, α — время выполнения характерной арифметической операции, β — время инициализации пересылки, γ — характерное время пересылки одного слова данных). В данном случае

оценка характерна для модели вычислений SPMD и модели программирования MPI.

Определим время выполнения параллельных вычислений $t(n_p, N)$, как функцию, зависящую от размерности задачи N и числа используемых процессоров n_p. Будем полагать, что размерность задачи N пропорциональна числу процессоров $r = N/n_p$, и, что время выполнения характерной арифметической операции определяется через α.

В случае использования строчно-ориентированных блоков, каждый процессор хранит r строк $\mathbf{a}_i$ матрицы $\mathbf{A}$ таких, что $r \cdot (k-1) + 1 \leqslant i \leqslant r \cdot k$ и вычисляет

$$c_i = \sum_{j=1}^{N} a_{ij} \cdot b_j.$$

Для каждого блока размера r необходимо вычислить N перемножений и $N-1$ операций сложения, что приближенно можно оценить как вычислительные затраты, равные $2Nr\alpha$. Напомним, что вектор $\mathbf{b}$ дублируется на каждом процессоре. В случае если результирующий вектор $\mathbf{c}$ также должен быть на каждом процессоре, необходимо выполнить операцию сбора для каждого процессора $P_k, k = 1, 2, \ldots, n_p$ приняв $r = N/n_p$ элементов.

При использовании столбцово-ориентированной схемы каждый процессор P_k хранит r столбцов $\mathbf{a}_j$ матрицы $\mathbf{A}$ таких, что $r \cdot (k-1) + 1 \leqslant j \leqslant r \cdot k$, а также соответствующие элементы вектора $\mathbf{b}$. Процессор P_k также вычисляет часть суммы

$$d_{kj} = \sum_{l=r\cdot(k-1)+1}^{r\cdot k} a_{jl} b_l.$$

Для вычисления d_{kj} необходимо r перемножений и $r-1$ сложений. Таким образом, для вычисления всех N величин требуются вычислительные затраты равные $2Nr\alpha$. Для получения окончательного результата произведения каждый процессор суммирует величины $d_{1j}, d_{2j}, \ldots, d_{nj}$, здесь $r \cdot (k-1) + 1 \leqslant j \leqslant r \cdot k$, т.е. P_k принимает блоки размерностью r. Определим функцию затрат на коммуникации. Будем полагать, что пересылка r чисел с плавающей точкой на соседний процессор по коммуникационной сети требует затрат $\beta + r \cdot \gamma$, где β — время инициализации пересылки; $r = N/n_p$ — размер пересылаемых данных; γ — характерное время пересылки одного слова данных.

Отметим, что в обоих рассмотренных вариантах матрично-векторного произведения оценки время выполнения и коммуникаций имеют одинаковые затраты.

В случае, если процессоры соединены как линейный массив, то операция суммирования выполняется за n_p шагов на каждом из которых посылается сообщение размерности $r = N/n_p$ и затраты на коммуникацию составят $n_p(\beta + r \cdot \gamma)$. Для операций передачи данных от одного ВУ к остальным и операций передачи данных от всех ВУ ко всем узлам можно использовать и более сложные модели, отражающие специфику маршрутизации в используемой модели вычислений и программирования. Для рассматриваемого случая оценку времени выполнения можно записать как

$$t(N, n_p) = \frac{2N^2}{n_p}\alpha + n_p\left(\beta + \frac{N}{n_p} \cdot \gamma\right).$$

Отметим, что согласно этому выражению время вычислений сокращается с ростом числа процессоров n_p, а время коммуникаций увеличивается линейно с увеличением n_p. Таким образом, при заданном размере задачи N увеличение ускорения параллельного матрично-векторного произведения возможно при ограничении числа процессоров $n_p \leqslant n_p^*$. Определим оптимальное число процессоров n_p^* для заданного N из условия минимума функции времени

$$\left.\frac{\partial t(N, n_p)}{\partial n_p}\right|_{N=\text{const}} = -\frac{2N^2\alpha}{n_p^2} + \beta = 0.$$

Значение n_p, соответствующее минимуму $t(N, n_p)$ — это оптимальное число процессоров $n_p^* = N \cdot \sqrt{2\alpha/\beta}$, которое линейно зависит от размерности задачи N. Полученная оценка потребует уточнения в случае разреженной матрицы, хранящейся в сжатом формате.

1.4.4 Произведение разреженной матрицы на вектор

Для достижения максимального ускорения на некоторых архитектурах, необходимо рассматривать несколько форматов хранения матрицы коэффициентов.

Матрица, число ненулевых элементов которой гораздо меньше числа нулевых элементов, называется разряженной. При работе с такими матрицами имеет смысл хранить только ненулевые элементы, или ненулевые элементы и малое число нулей [12].

Пусть матрица $\mathbf{A}$ — произвольная разреженная матрица, размерности $N \times N$, и которая содержит Nnz ненулевых элементов.

Рассмотрим форматы представления разреженных матриц [12], т.е. матриц имеющих большое число нулевых элементов на примере матрицы $\mathbf{A}$,

имеющей следующий вид

$$\mathbf{A} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 4 & 2 \\ 1 & 0 & 1 & 5 \end{pmatrix}.$$

Отметим, что матрица $\mathbf{A}$ — квадратная, размерности 4×4 и имеет $Nnz = 8$ ненулевых элементов.

Формат CSR

CSR (Compressed Sparse Row Storage Format) — построчный формат хранения разряженных матриц. При использовании этого формата создаются три массива:

- AV — массив значений ненулевых элементов матрица $\mathbf{A}$. Значения элементов записываются упорядоченно, по строкам.
- ANC — i-ый элемент этого массива — номер столбца, в котором находится i-ый элемент массива AV в матрицы $\mathbf{A}$.
- ANL — i-ый элемент этого массива — номер элемента из массива AV, с которого начинается i-ая строка в матрице $\mathbf{A}$. Последний элемент всегда число ненулевых элементов.

Массивы AV и ANC имеют размерность Nnz, а массив ANL размерность $(N + 1)$.

На примере матрицы $\mathbf{A}$ представление её в формате CSR имеет вид:
$AV = \begin{pmatrix} 2 & 1 & 3 & 4 & 2 & 1 & 1 & 5 \end{pmatrix}$;
$ANC = \begin{pmatrix} 1 & 2 & 2 & 2 & 3 & 0 & 2 & 3 \end{pmatrix}$;
$ANL = \begin{pmatrix} 0 & 2 & 3 & 5 & 8 \end{pmatrix}$.

Последовательный алгоритм умножения матрицы $\mathbf{A}$ в формате CSR на вектор $\mathbf{b}$ размерности N может быть записан как

Алгоритм 1 Последовательный алгоритм матрично-векторного произведения в формате CSR

1: **for** $i = 0 \to N$ **do**
2:     **for** $j = ANL[i] \to ANL[i+1]$ **do**
3:         $res[i] := res[i] + AV[j] \cdot b[ANC[j]]$;
4:     **end for**
5: **end for**

Результат **Алгоритма 1** помещается в вектор res размерности N.

Рассмотрим параллельный вариант SpMV также в формате CSR. Пусть на процессоре P_k хранятся k-ые части массивов AV, ANC, ANL и заданного вектора. Массив ANL разбивается на части, размером N/n_p, а AV и ANC на части, размером $(ANL[(k+1)\cdot\frac{N}{n_p}] - ANL[k\cdot\frac{N}{n_p}])$, где k — номер процессора P_k. Тогда параллельный алгоритм умножения матрицы $\mathbf{A}$ в формате CSR, на вектор $\mathbf{b}$ будет выглядеть следующим образом:

Алгоритм 2 Параллельный алгоритм матрично-векторного произведения в формате CSR

1: $res_k[i] := 0$; на P_k, $k = 0, 1, \ldots, (n_p - 1)$; $i = \frac{k\cdot N}{n_p}, \frac{k\cdot N}{n_p}+1, \ldots, \frac{(k+1)\cdot N}{n_p}-1$;

2: $res_k[i] := res_k[i] + AV_k[j]\cdot b[ANC_k[j]]$, где $k = 0, 1, \ldots, (n_p - 1)$;
$i = \frac{k\cdot N}{n_p}, \frac{k\cdot N}{n_p}+1, \ldots, \frac{(k+1)\cdot N}{n_p}-1$; $j = ANL_k[i], ANL_k[i]+1, \ldots, ANL_k[i+1]$;

3: $P_k(res_k) \longrightarrow P_0(res_k)$, $\quad k = 1, 2, \ldots, (n_p - 1)$;

4: $res[i] := res[i] + res_k[i]$ на P_0, $i = 0, 1, \ldots, (N - 1)$; $k = 0, 1, \ldots, (n_p - 1)$.

На первом шаге **Алгоритма 2** обнуляем i-ые компоненты вектора res на каждом процессоре, на втором — умножаем части матрицы и вектора, а на третьем — пересылаем вектора результатов на нулевой процессор, и на последнем шаге складываем локальные векторы в вектор res.

Формат COO

COO (Coordinate Storage Format) — координатный формат хранения. При использовании этого формата создаются три массива:

- AV — массив значений ненулевых элементов матрицы $\mathbf{A}$. Значения элементов могут записываться в любом порядке.
- ANC — i-ый элемент этого массива является номером столбца, в котором находится i-ый элемент массива AV в матрицы $\mathbf{A}$.
- ANR — i-ый элемент этого массива является номером строки, в которой находится i-ый элемент массива AV в матрицы $\mathbf{A}$.

Все массивы имеют размерность Nnz. Вернёмся к примеру матрицы $\mathbf{A}$:

$AV = \begin{pmatrix}2 & 1 & 3 & 4 & 2 & 1 & 1 & 5\end{pmatrix}$;

$ANC = \begin{pmatrix}1 & 2 & 2 & 2 & 3 & 0 & 2 & 3\end{pmatrix}$;

$ANR = \begin{pmatrix}0 & 0 & 1 & 2 & 2 & 3 & 3 & 3\end{pmatrix}$.

Последовательный и параллельный алгоритмы умножения матрицы $\mathbf{A}$ в формате COO, на вектор $\mathbf{b}$ размерности N представлены в **Алгоритме 3**

Алгоритм 3 Последовательный алгоритм матрично-векторного произведения в формате COO

1: **for** $i = 0 \to Nnz$ **do**
2: $\quad res[ANR[i]] := res[ANR[i]] + AV[i] \cdot b[ANC[i]];$
3: **end for**

и **Алгоритме 4** соответственно. Результаты записываются в вектор res размерности N.

Пусть на k-ом процессоре хранятся части массивов AV, ANC, ANR, размерности Nnz/n_p и вектор правых частей. Тогда параллельный алгоритм умножения матрицы $\mathbf{A}$ в формате COO, на вектор $\mathbf{b}$ будет выглядеть следующим образом:

Алгоритм 4 Параллельный алгоритм матрично-векторного произведения в формате COO

1: $res_k[i] := 0;$ на P_k, $k = 0, 1, \ldots, (n_p - 1); \; i = 0, 1, \ldots, (N - 1);$
2: $res_k[ANR_k[i]] := res_k[ANR_k[i]] + AV_k[i] \cdot b[ANC_k[i]],$
здесь $k = 0, 1, \ldots, (n_p - 1); \; i = 0, 1, \ldots, \frac{Nnz}{n_p};$
3: $P_k(res_k) \longrightarrow P_0(res_k), \quad k = 1, 2, \ldots, (n_p - 1)$
4: $res[i] := res[i] + res_k[i]$ на P_0, $i = 0, 1, \ldots, (N - 1); \; k = 0, 1, \ldots, (n_p - 1).$

При доступе к компонентам вектора res, на втором шаге **Алгоритма 4** применяется косвенная индексация, в которой индекс i — локальный.

Формат ELL

ELL (ELLPack Storage Format) — строчный формат хранения. Исходная матрица преобразуется следующим образом: все ненулевые элементы записываются в строках по очереди слева направо. Находим строку, в которой расположено наибольшее число ненулевых элементов $LNnz$. Остальные строки дополняются нулями до размера этой строки.

Представление разряженной матрицы включает два массива AVe и $ANCe$ размерности $N \cdot LNnz$:

- AVe — массив значений элементов (включая нули);
- $ANCe$ — массив номеров столбцов элементов. Если на i-ом месте массива AVe находится 0, то $ANCe[i] = -1$.

На примере матрицы **A**:

$$\mathbf{A} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 4 & 2 \\ 1 & 0 & 1 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 1 & \\ 3 & & \\ 4 & 2 & \\ 1 & 1 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 1 & 0 \\ 3 & 0 & 0 \\ 4 & 2 & 0 \\ 1 & 1 & 5 \end{pmatrix}$$

соответственно вектора имеют вид

$AVe = \begin{pmatrix} 2 & 1 & 0 & 3 & 0 & 0 & 4 & 1 & 0 & 1 & 1 & 5 \end{pmatrix}$;

$ANCe = \begin{pmatrix} 1 & 2 & -1 & 2 & -1 & -1 & 2 & 3 & -1 & 0 & 2 & 3 \end{pmatrix}$.

Последовательный алгоритм умножения матрицы **A**, хранящейся в формате ELL на вектор **b** размерности N. Результат **Алгоритма 5** записывается в вектор res размерности N:

Алгоритм 5 Последовательный алгоритм матрично-векторного произведения в формате ELL

1: **for** $i = 0 \rightarrow N \cdot LNnz$ **do**
2: **if** $AVe[i] \neq 0$ **then**
3: $res[i/LNnz] := res[i/LNnz] + AVe[i] \cdot b[ANCe[i]]$;
4: **end if**
5: **end for**

Пусть на k-ом процессоре хранятся части массивов $AVe, ANCe$, размерности $(N \cdot LNnz)/n_p$ и вектор. Тогда параллельный алгоритм умножения матрицы **A** в формате ELL на вектор **b** выглядит следующим образом:

Алгоритм 6 Параллельный алгоритм матрично-векторного произведения в формате ELL

1: $res_k[i] := 0;$ на P_k, $i = 0, \ldots, (N-1)$; $k = 0, \ldots, (n_p - 1)$;
2: $res_k[i/LNnz + k \cdot LNnz] := res_k[i/LNnz + k \cdot LNnz] +$
$+ AVe_k[i] \cdot b[ANCe_k[i]], \quad i = 0, 1, \ldots, \frac{N \cdot LNnz}{n_p}; \; k = 0, 1, \ldots, (n_p - 1)$;
3: $P_k(res_k) \longrightarrow P_0(res_k), \; k = 1, 2, \ldots, (n_p - 1)$;
4: $res[i] := res[i] + res_k[i]$ на P_0, $i = 0, 1, \ldots, (N-1)$; $k = 0, 1, \ldots, (n_p - 1)$.

Отметим, что на втором шаге **Алгоритма 6** используется локальная нумерация при доступе к компонентам вектора res.

Формат BCSR

BCSR (Block-Based Compressed Row Storage Format) [12] — блочный формат хранения по строкам. Матрица **A** разбивается на блоки заранее из-

вестного размера. Рассмотрим разбиение матрицы на блоки размера 2×2. При этом создаются три массива AVB, $ANCB$, $ANLB$, элементы которых:

- AVB — содержат значения элементов матрицы $\mathbf{A}$. Значения записываются упорядоченно, по блокам. При этом встречаются не только ненулевые элементы.
- $ANCB$ — являются номерами столбцов левого верхнего элемента каждого блока.
- $ANLB$ — являются номерами элементов из массива $ANCB$, с которых начинаются новые строки блоков.

Массив AVB имеет размерность $Nnzb \geq Nnz$, массив $ANCB$ размерность Nb равную количеству блоков, массив $ANLB$ размерность, равную $N/2+1$. Тогда матрица $\mathbf{A}$ будет иметь следующий вид:

$$\mathbf{A} = \left(\begin{array}{cc|cc} 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 0 \\ \hline 0 & 0 & 4 & 2 \\ 1 & 0 & 1 & 5 \end{array}\right)$$

и её представление как:
$AVB = \begin{pmatrix} 0 & 2 & 0 & 0 & 1 & 0 & 3 & 0 & 0 & 0 & 1 & 0 & 4 & 2 & 1 & 5 \end{pmatrix}$;
$ANCB = \begin{pmatrix} 0 & 2 & 0 & 2 \end{pmatrix}$;
$ANLB = \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}$.

Последовательный алгоритм умножения матрицы $\mathbf{A}$ в формате BCSR, на вектор $\mathbf{b}$ размерности N имеет вид:

Алгоритм 7 Последовательный алгоритм матрично-векторного произведения в формате BCSR

1: **for** $i = 0 \to \frac{N}{2}$ **do**
2:     **for** $j = ANLB[i] \to ANLB[i+1]$ **do**
3:         $res[2 \cdot i] := res[2 \cdot i] + AVB[j] \cdot b[ANCB[j]] + {} + AVB[j+1] \cdot b[ANCB[j]+1];$
4:         $res[2 \cdot i+1] := res[2 \cdot i+1] + AVB[j+2] \cdot b[ANCB[j]] + {} + AVB[j+3] \cdot b[ANCB[j]+1];$
5:     **end for**
6: **end for**

Результат **Алгоритма 7** записывается в вектор res размерности N. Пусть на k-ом процессоре P_k хранятся части массивов AVB, $ANCB$, $ANLB$

Алгоритм 8 Параллельный алгоритм матрично-векторного произведения в формате BCSR

1: $res_k[i] := 0$; на P_k, $i = \frac{k\cdot N}{n_p}, \frac{k\cdot N}{n_p}+1, \ldots, \frac{(k+1)\cdot N}{n_p}-1$; $k = 0, 1, \ldots, (n_p-1)$;

2: $res_k[2\cdot i] := res_k[2\cdot i] + AVB_k[j]\cdot b[ANCB_k[j]] +$
$+AVB_k[j+1]\cdot b[ANCB_k[j]+1]$;
$res_k[2\cdot i+1] := res_k[2\cdot i+1] + AVB_k[j+2]\cdot b[ANCB_k[j]] +$
$+AVB_k[j+3]\cdot b[ANCB_k[j]+1]$;
где $k = 0, 1, \ldots, (n_p-1)$; $j = ANLB_k[i], ANLB_k[i]+1, \ldots, ANLB_k[i+1]$;
$i = \frac{k\cdot N}{2\cdot n_p}, \frac{k\cdot N}{2\cdot n_p}+1, \ldots, \frac{(k+1)\cdot N}{2\cdot n_p}-1$;

3: $P_k(res_k) \longrightarrow P_0(res_k), \quad k = 1, 2, \ldots, (n_p-1)$.

4: $res[i] := res[i] + res_k[i]$ на P_0, $i = 0, 1, \ldots, (N-1)$; $k = 0, 1, \ldots, (n_p-1)$.

и вектор правых частей. Массив $ANLB$ разбивается на части, размером $\frac{N}{2\cdot n_p}$, массив $ANCB$ на части размером $(ANLB[(k+1)\cdot\frac{N}{n_p}] - ANLB[k\cdot\frac{N}{n_p}])$, а массив AVB на части в четыре раза большие, чем $ANCB$.

Тогда параллельный алгоритм умножения матрицы $\mathbf{A}$ в формате BCSR на вектор $\mathbf{b}$ будет иметь вид **Алгоритма 8**. В отличие от предыдущих параллельных алгоритмов, на втором шаге вычисляются две компоненты вектора res_k, поэтому диапазон индекса i изменяется в два раза.

Введенное ранее, оптимальное число процессоров n_p^* для сжатых форматов хранения зависит от числа ненулевых элементов матрицы, и например для CSR формата составит $n_p^* = \sqrt{(2\cdot Nnz\cdot\alpha)/\beta}$.

1.4.5 Объектно-ориентированное программирование в SpMV

Большинство далее рассматриваемых программных кодов будут использовать язык C++. Это связано прежде всего с тем, что на C++ имеется возможность использования различных парадигм программирования, а с другой стороны можно не использовать определённые языковые средства в соответствии с предпочтениями разработчика. Например, не использовать макросы, шаблоны и т.д. Хорошим введением в тему данного раздела может послужить работа С.С. Гайсаряна [7].

Объектно-ориентированное программирование (ООП) — парадигма программирования, основанная на представлении предметной области в виде системы взаимосвязанных абстрактных объектов и их реализаций. Практически все решаемые вычислительные задачи состоят из тесно взаимосвязанных объектов с динамическими свойствами, поэтому ООП зачастую лучше подходит для решения прикладных задач, связанных с параллель-

ными вычислениями. Более того, структурированный ООП код легче модифицировать, заменяя внутренние части в соответствии с изменениями в прикладных вычислительных моделях.

Для многих вычислительных задач программирование с ООП является важным инструментом, поскольку обеспечивает модульность и целостность данных, облегчает распространение и повторное использование кода.

Однако, не все вычислительные задачи необходимо выражать через объекты и их отношения — в некоторых случаях ориентация на объекты приводит к лишнему усложнению. Например, код произведения матрицы на вектор в разделе 1.4.6 можно было бы написать без использования специального класса матриц. Кроме того, между объектами возможно сложное взаимодействие, которое приведет к еще большему усложнению программной реализации или нежелательным издержкам.

Язык C++ поддерживает несколько парадигм в подходе к разработке библиотек программ, которые делают средства параллельного программирования легко доступными. Помимо библиотек системного уровня, для поддержки параллелизма в C++ могут применяться промежуточное программное обеспечение, как MPI, OpenMP и CORBA, которое будет подробно рассмотрено в следующих разделах.

1.4.6 Последовательная реализация SpMV на C++

Программная реализация SpMV в рамках объектно-ориентированного подхода C++ может строиться на основе как пользовательских классов (или шаблонов классов) для матрицы **A** и векторов **b** и **c**, так и с использованием контейнеров из библиотеки стандартных шаблонов (Standard Templates Library (STL)) `http://www.sgi.com/tech/stl/`. Контейнеры — это объекты, хранящие в себе другие объекты. Для хранения матрицы в CSR формате могут использоваться последовательные контейнеры `vector` или ассоциативный контейнер `map`. Произведение матрицы на вектор может быть реализовано несколькими способами: переопределением оператора `*`, в виде функции члена класса «матрица», реализуемой в виде отдельной функции.

При выборе реализации необходимо учитывать, что *наследование* приводит к дополнительным зависимостям по данным, которые ограничивают возможности распараллеливания программного кода. Ограничимся рассмотрением варианта, в котором для хранения матриц в формате CSR используется класс `SpM` (см. **Листинг 1**), а произведение матрицы на вектор реализуется в виде метода (функции члена класса) `SpMV` данного класса. Для хранения матрицы в классе `SpM` применяются контейнеры `vector`.

Листинг 1. Объявление класса SpM

```
1  // Файл spm.h
2  #include <vector>
3
4  class SpM {
5    int N; // Число строк матрицы
6    int Nnz; // Число ненулевых элементов матрицы
7    vector <double> AV; //Значения элементов матрицы
8    vector <int> ANC; //Столбцовые индексы элементов матрицы
9    vector <int> ANL; //Начальные позиции строк в AV и ANC
10 public:
11   SpM (int , int ); // конструктор
12  ~SpM () {}; // деструктор
13 /* Произведение матрицы на вектор */
14 void SpMV(vector <double> &, vector <double> &, int , int );
15   int getN (void) {return N;};
16   int getNnz (void) {return Nnz;};
17 vector <double>& getAV (void) {return AV;};
18 vector <int>&    getANC (void) {return ANC;};
19 vector <int>&    getANL (void) {return ANL;};
20 };
```

При объявлении объекта класса `SpM` в пользовательской программе конструктор `SpM::SpM` (**Листинг 2**) создаст структуру данных для хранения матрицы размерности $fN \times fN$ с $fNNZ$ ненулевыми элементами. Определение метода `SpMV` показано в **Листинге 3**.

Листинг 2. Конструктор класса SpM

```
1 // Файл spm.h
2 SpM::SpM (int fN, int fNNZ)
3 {
4    N = fN;   Nnz = fNNZ;
5    AV.resize(fNNZ); ANC.resize(fNNZ); ANL.resize(fN+1);
6 }
```

Границы внешнего цикла `ib` и `ie` позволят в дальнейшем (см. раздел 3.6) применить метод `SpM::SpMV` к строчному блоку матрицы $\mathbf{A}$, при создании параллельных версий программы. Введенные дополнительно переменные `jb`, `je` и `sum` уменьшают частоту обращений к контейнерам `ANL` и `c`.

Листинг 3. Произведение матрицы, хранящейся в CSR формате на вектор

```
// Файл spm.h
void SpM::SpMV (vector <double> &b, vector <double> &c,
                                    int ib, int ie)
{
    for(int i=ib; i<ie; i++)
    {
        double sum=0;
        int jb=ANL[i]; int je=ANL[i+1];
        for(int j=jb;j<je;j++)
            sum+= AV[j]*b[ANC[j]];
        c[i]=sum;
    }
}
```

Рассмотрим пример программы, вычисляющей произведение $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ с использованием класса `SpM`, представленного **Листингом 4**.

В данном примере полагается, что $N = 8$, $Nnz = 64$. Матрица $\mathbf{A} = \{a_{ij}\} \in \mathbb{R}^{N\times N}$, здесь $a_{ij} = k$, где $k = 0, 1, \ldots, Nnz$. Векторы $\mathbf{b} \in \mathbb{R}^N$ и $\mathbf{c} \in \mathbb{R}^N$, где $\mathbf{b} = \{b_i\} \,|\, b_i = 1$ и $\mathbf{c} = \{c_i\} \,|\, c_i = 1$.

Листинг 4. Пример программы, использующей метод SpM::SpMV

```
// Файл main.cpp
#include <vector>
#include <iostream>
#include "spm.h"

using namespace std;
const int N=8, Nnz=64;
void filling(SpM &);

int main(int argc, char *argv[])
{
/* Инициализация векторов b,c и матрицы A */
vector <double> b(N,1), c(N,1);
SpM A(N,Nnz);
    filling(A); // Заполнение A

    A.SpMV(b,c,0,N); // Произведение c = A * b

```

```
19 double bc=0; // Проверка результата
20 for(int i=0; i<N; i++) bc += b[i]*c[i];
21 cout << "(b,c)=" << bc << " =" << Nnz*(Nnz-1)/2 << endl;
22 return 0;
23 }
```

Векторы **b** и **c** также реализованы в виде последовательных контейнеров `vector`, элементы векторов при создании инициируются единицами. Матрица **A** создается как объект класса `SpM` с заданной размерностью и числом ненулевых элементов. Элементы матрицы инициируются в функции `filling`. Собственно, матрично-векторное произведение $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ реализуется в вызове метода `SpMV` класса `SpM`, применяемого к объекту `A`.

В программе **Листинг 4** строки 21–23 используются для контроля достоверности результата. Функция `filling` (см. **Листинг 5**), реализует выражение $a_{ij} = k$, где $k = 0, 1, \ldots, Nnz$ для матрицы **A** в формате CSR.

Листинг 5. Функция для заполнения матрицы

```
1  // Файл main.cpp
2  void filling(SpM& A)
3  {
4  vector <double>& AV =A.getAV();
5  vector <int>&    ANC=A.getANC();
6  vector <int>&    ANL=A.getANL();
7  for(int k=0; k<A.getNnz(); k++) AV[k]=k;
8  for(int i=0, k=0; i<N; i++)
9   {
10   for(int j=0; j<A.getN(); j++, k++) ANC[k]=j;
11         ANL[i+1] = k;
12  }
13 }
```

В примере с SpMV (см. раздел 1.4.3) удаётся оценить затраты на коммуникацию приняв модель архитектуры ВС в виде линейного массива, однако, ещё один вклад во многом определяющий успех параллельного SpMV, зависит от выбора подходящего промежуточного программного обеспечения, обеспечивающего коммуникации. Далее будут рассмотрены основные системы ППО и на примере реализации SpMV показана их эффективность.

2 Технология разработки масштабируемых параллельных программ на MPI

В настоящее время парадигма передачи сообщений является очень распространенной. Универсальность этой парадигмы основана на модели параллельных процессов, которые взаимодействуют посредством обмена сообщениями. Обмен сообщениями может осуществляться по сетевым протоколам, в общей памяти и используя прямой доступ к удаленной памяти (RDMA — remote direct memory access). Одним из первых интерфейсов передачи сообщениями стал интерфейс PVM (Parallel Virtual Machine — параллельная виртуальная машина. Отличительной особенностью PVM (`http://www.portablecomponentsforall.com/edu/pvm-ru/`) являются поддержка гетерогенности вычислительных узлов, коммуникационных сетей и приложений, и динамически настраиваемый процессорный пул (множество, используемых процессоров может изменяться во время исполнения программы). PVM — это фактический стандарт параллельного программирования на гетерогенных кластерах.

На основе PVM разработана технология MPI [13] (Message Passing Interface — интерфейс передачи сообщений), изначально ориентированная на вычислительные системы с массовым параллелизмом, которые характеризуются аппаратной однородностью.

Программы, написанные с применением MPI, могут выполняться на многопроцессорных системах с общей или разделяемой памятью, на сетях рабочих станций, на однопроцессорных машинах с одно- и многоядерными процессорами. Исходный код MPI-программ переносим на любые платформы, поддерживающие модель MPI, но исполняемые коды программ под разными платформами чаще всего взаимодействовать не могут.

Интерфейс MPI стандартизован (стандарт MPI-1 принят в 1993 г., в 1997 г. был разработан проект стандарта MPI-2). MPI — это самое распростаненное middleware для параллельных вычислений. Существуют множество реализаций, придерживающихся этого стандарта, таких как MPICH, Intel MPI, HP MPI, Open MPI и т.п. Так, реализация Open MPI является наследником LAM/MPI и поддерживается консорциумом партнеров из различных областей науки, разработчиков и производителей. OpenMPI (`http://www.open-mpi.org/`) — открытая бесплатная реализация технологии MPI-2. Она может использоваться для проведения параллельных расчётов на различных вычислительных кластерах и отдельных компьютерах.

2.1 Модель передачи сообщений MPI

В модели передачи сообщений MPI, параллельная программа представляет собой множество процессов, каждый из которых имеет собственное локальное адресное пространство (модель распределенной памяти). Взаимодействие процессов — обмен данными и синхронизация — осуществляется посредством передачи сообщений (модель явной коммуникации).

Достоинствами MPI являются:

- использование на вычислительных системах с общей и распределенной памятью;
- широкая распространённость свободнораспространяемых реализаций;
- простота интерфейса обмена сообщениями;
- наличие разных схем и режимов передачи данных (точечные, коллективные, с блокировкой и без, явная буферизация и т.д);
- возможность введения новых типов пересылаемых данных и объединение данных при их упаковке;
- наличие распределенных вычислительных операций;
- управление группами взаимодействующих процессов;
- удобная функциональность измерения времени выполнения процессов.

Кратко рассмотрим отмеченные достоинства MPI. Большинство реализаций MPI построены на концепции абстрактного устройства (ADI — Abstract Device Interface), обеспечивающего переход к различным аппаратным платформам при выборе, соответствующего ADI (p4 — для систем с передачей сообщений и p2 — для систем с разделяемой памятью). Таким образом, обеспечивается использование MPI на вычислительных системах с общей и распределенной памятью.

Практически на каждом современном суперкомпьютере или однородном кластере установлена одна или несколько реализаций MPI, как правило, свободнораспространяемых.

Интерфейс обмена сообщениями прост — функции MPI, используют высокоуровневую информацию. Для осуществления передачи данных пользователь должен указать номер процесса получателя, пересылаемые данные, их тип и коммуникатор (коммуникационный объект, а для пользователя — почти синоним выбранной группы процессов), в рамках которого осуществляется передача данных.

Наличие разных схем и режимов передачи данных не только обеспечивает обмены для разного числа процессов (парные («точка-точка») и коллективные функции), но и позволяет совмещать пересылку данных с вычислениями (неблокирующие функции, буферизованный режим и режим по готовности), и выполнять неявную синхронизацию процессов (коллективные функции и функции с блокировкой).

Производные (новые коммуникационные) типы и упаковка данных позволяют уменьшить число сообщений за счет объединения данных, в том числе в случае данных разного типа и размера.

Применение распределенных вычислительных операций упрощает поиск глобальных экстремумов в данных, распределенных по процессам, суммирование таких данных и т.д.

Функциональность работы с группами и коммуникаторами позволяет MPI организовывать различные группы процессов, исходя из номеров процессов (чётные, нечетные, с заданным и произвольным шагом по номерам процессов). Полученные группы процессов могут взаимодействовать уже посредством коллективных функций, а не функций «точка-точка», т.е. более оптимально и, вместе с тем, не загромождая программу «коммуникационным» кодом.

Измерение времени выполнения отдельного процесса в MPI состоит в обрамлении участка программного кода вызовами функции `MPI_Wtime()`. Разность возвращенных значений — искомое время в секундах. Чтобы получить время исполнения всей параллельной программы предварительно все процессы синхронизируются вызовом `MPI_Barrier(MPI_COMM_WORLD)`, а после измерения времени (по представленной выше процедуре) берется максимальное значение времени выполнения, например, с помощью функции `MPI_Reduce(&t,&tmax,1,MPI_DOUBLE,MPI_MAX,root,MPI_COMM_WORLD)`, здесь `t` — время, полученное в каждом процессе, `tmax` — максимальное значение времени, помещаемое в процесс с номером `root`.

Функция `MPI_Wtick()` возвратит минимальный измеримый отрезок времени и позволит оценить достоверность замера времени.

Наряду с достоинствами, у интерфейса MPI имеются и недостатки отметим основные из них:

- ограниченность базовых типов данных одномерными массивами, отсутствие механизма для передачи данных типа класс или объект класса;

- необходимость детального управления распределением данных между процессами (процессы отправитель и получатель, адресация отправляемого и принимаемого участка памяти, объем и тип данных)

— это приводит к высокой трудоемкости разработки программ и снижает читабельность кода;

- сложность написания программ, способных выполняться при произвольном числе процессов и произвольных размерах данных (специальная обработка массивов, имеющих размер не кратный числу процессов);
- сложность выражения многоуровневого параллелизма программы;
- гранулярность больших сообщений часто приводит к изменению латентности;
- перекрытие коммуникаций;
- глобальные (коллективные) операции с возвратом результата во все процессы часто являются дорогостоящими, а на многоядерных системах — неоправданно дорогостоящими.

На этапе разработки параллельной программы необходимо учитывать отмеченные недостатки MPI. Так, ограниченность базовых типов данных компенсируется возможностью введения новых типов пересылаемых данных и объединения данных при их упаковке. Но при этом, следует учесть, что введение новых типов (создание типа и его регистрация) — процедура достаточно затратная и ее не следует выполнять в многократно вызываемых функциях (методах). Кроме того, многомерные однотипные данные можно линеаризовать, что немного ухудшает читабельность кода, но упрощает его оптимизацию.

Отсутствие механизма для передачи данных типа класс или объект класса компенсируется SPMD стилем программирования. Наличие одного кода для взаимодействующих процессов предполагает объявление класса и объектов в каждом из процессов по умолчанию. Но необходимо быть внимательным при объявлении объектов и переменных в ветвях кода, соответствующих отдельным процессам (ветвление MPI-программ, связано с номерами процессов).

При разработке параллельной программы изначально следует ориентироваться на произвольное число процессов и размер массивов. Последовательность вызовов функций «точка-точка» должна обеспечивать обмен без взаимных блокировок (deadlock-ов) за счет функций передачи данных без блокировки или функции `MPI_Sendrecv`, при взаимных и кольцевых обменах. Посредством различных процедур выравнивания данных можно перейти к обменам данными фиксированной длины. Переменного числа

процессов в ходе счета можно достичь при избыточном (MPI-1) или динамическом (MPI-2) выделении процессов.

Многоуровневый параллелизм реализуется в MPI-1 за счет статического выделения нескольких процессов на каждую подзадачу, с выделением ведущих процессов на каждом уровне, и многоуровневой организации обменов (при помощи функций семейств `MPI_Group_` и `MPI_Comm_` строится иерархия коммуникаторов — однократно!). В MPI-2 необходимые процессы выделяются динамически.

Передачи большого объема данных по возможности следует избегать. В рамках MPI-2 данные могут быть введены в процессы или записаны из них параллельными функциями ввода/вывода. В MPI-1 данные можно разделить и разослать частями, используя, например, иерархию коммуникаторов. Другой подход — совмещение вычислений и коммуникаций при использовании функций передачи данных без блокировки.

Глобальные (коллективные) операции могут быть заменены на функции из оптимизированных библиотек. Так же `MPI_Allreduce` заменяется последовательным вызовом `MPI_Reduce` и `MPI_Bcast` или процедурой, построенной на функциях «точка-точка». К тому же, разработчики основных реализаций MPI совершенствуют свое программное обеспечение.

При использовании MPI происходит дублирование в MPI-процессах следующих данных: пользовательских массивов, которые требуются для вычислений, распределенных между разными процессами; используемых в пользовательской программе буферов для обмена данными между процессами; системных буферов, используемых библиотекой MPI. Все эти свойственные MPI затраты дополнительной памяти становятся очень критичными для многоядерных узлов, поскольку имеется явная тенденция к сокращению объема оперативной памяти, приходящейся на одно ядро — объём памяти и быстродействие процессора всегда сбалансированы, исходя из потребностей приложений, а быстродействие процессора будет повышаться за счет увеличения числа ядер в нём.

2.2 Обзор интерфейса передачи сообщениями MPI

Парадигма MPI является развитием идеи последовательного программирования для параллельных вычислений. Несколько задач последовательного программирования рассматриваются вместе, для их кооперации используются средства коммуникации. Таким образом, MPI имеет интерфейс передачи сообщений, но не имеет интерфейсов общего адресного пространства или непосредственного обращения одного процессора к памяти другого.

Эффективность MPI достигается также за счет отсутствия ненужной работы по пересылке лишней информации с каждым сообщением, или кодированию и декодированию заголовков сообщений. MPI был разработан так, чтобы поддерживать одновременное выполнение вычислений и коммуникаций. Это достигается использованием неблокируемых коммуникационных вызовов, которые разделяют инициацию коммуникации и ее завершение. Для уменьшения времени передачи сообщений в MPI используются и другие приемы. Например, встроенная буферизация позволяет избежать задержек при отправке данных — управление в передающую ветвь возвращается немедленно, даже если ветвь-получатель еще не подготовилась к приему. MPI использует многопоточность (multi-threading), вынося большую часть своей работы в потоки (threads) с низким приоритетом. Буферизация и многопоточность сводят к минимуму негативное влияние на производительность прикладной программы простоев, неизбежных при пересылках. На передачу данных типа «один-всем» затрачивается время, пропорциональное не числу участвующих ветвей, а логарифму этого числа.

В MPI используется абстрактное понятие *области коммуникаций/связи (communication domain)*. Для каждой области связи вводится ее описатель — коммуникатор. Области связи автоматически создаются и уничтожаются вместе с коммуникаторами. Именно с коммуникаторами пользователь имеет дело, вызывая функции пересылки данных, а также подавляющую часть вспомогательных функций. Одной области связи могут соответствовать несколько коммуникаторов. В MPI-1 коммуникаторы являются «несообщающимися сосудами»: если данные отправлены через один коммуникатор, ветвь-получатель сможет принять их только через этот же самый коммуникатор, но ни через какой-либо другой.

Основными функциональными возможностями MPI являются.

- *Передача сообщений «точка-точка».* Основной механизм коммуникаций в MPI — передача данных между парой процессов (функции передачи — `MPI_Send`, приема — `MPI_Recv` и их гибрид — `MPI_Sendrecv`). При этом, необходимо обеспечить парность вызовов функций `MPI_Send(...,dest,...,comm)`,`MPI_Recv(...,source,...,comm,...)` в рамках одного коммуникатора `comm`, здесь `MPI_Send` вызывается в процессе с номером `source`, а `MPI_Recv` в процессе `dest`. Корректность принятого сообщения в MPI обеспечивается соответствием типа данных и размера сообщений в передающем и принимающем процессах. Существует два вида функций для коммуникаций «точка-точка» (блокирующие и неблокирующие) и четыре режима передачи сообщений: стандартный, синхронный, буферизованный и по ожиданию.

- *Коллективные операции.* Под термином «коллективные» в MPI подразумеваются три группы функций: функции коллективного обмена данными, синхронизации (или барьеры) и функции поддержки распределенных операций. Вызов коллективной функции является корректным, в том случае, если он произведен из всех процессов соответствующей области связи, и именно с этим коммуникатором в качестве аргумента (хотя для одной области связи может иметься несколько коммуникаторов, подставлять их вместо друг друга нельзя). В этом и заключается коллективность: либо функция вызывается всем коллективом процессов, либо никем.

- *Группы процессов и коммуникаторы.* Группа — это подмножество процессов MPI. Вместе с тем, группа является не самодостаточным понятием, а лишь более развитым инструментом создания новых коммуникаторов. Один процесс может быть членом нескольких групп. Изначально не существует группы из всех процессов, соответствующей коммуникатору `MPI_COMM_WORLD`, она создается вызовом `MPI_Comm_group(MPI_COMM_WORLD,group)`. Согласно концепции MPI, после создания группу нельзя дополнить или уменьшить — можно создать только новую группу под требуемое множество процессов на базе уже существующей. При помощи механизма групп можно создать коммуникаторы для коллективных операциях, выполняемых на процессах с достаточно сложной графовой структурой связей (например, связаны процессы с номерами: 17,21,32,44,67). В более простых случаях можно использовать функции `MPI_Comm_split`.

Наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий, является одной из наиболее привлекательных сторон MPI. Причем, гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа «точка-точка» состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором, т.е. соответствующие функции должны быть вызваны в каждом из процессов.

В MPI содержатся коллективные функции следующих типов:

- барьерная синхронизация процессов коммуникатора (`MPI_Barrier`);

- рассылка одних и тех же данных от одного процесса всем (`MPI_Bcast`);

- сбор данных одним процессом от остальных (MPI_Gather);
- разделение данных одного процесса между всеми процессами (MPI_Scatter);
- вариация сбора данных, при которой все процессы получают результат (MPI_Allgather);
- обмен данными «каждый с каждым» (MPI_Alltoall);
- редукционные операции (MPI_Reduce);
- комбинированные операции редукции и разделения (MPI_Reduce_scatter);
- частичные редукционные операции (MPI_Scan).

Для большинства коллективных коммуникационных функций предусмотрен вариант с переменным размером сообщений (MPI_Gatherv, MPI_Scatterv, MPI_Allgatherv,MPI_Alltoallv).

Отличительные особенности коллективных операций.

- Коллективные коммуникации не взаимодействуют с коммуникациями типа «точка-точка».
- Коллективные коммуникации выполняются в режиме с локальной блокировкой. Возврат из функции MPI в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
- Сообщения не имеют идентификаторов.

Лучший алгоритм для коллективных коммуникаций зависит от размера сообщения, числа процессов и их привязки к процессорам вычислительной системы.

2.3 Особенности стандарта MPI-2

Существенно расширяются функциональные возможности в стандарте MPI-2 (http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf). Отметим наиболее значимые из них.

- Создание и управление процессами направлено на исключение только статической модели процесса в MPI.

- Односторонняя передача данных, определяет коммуникационные операции, которые могут быть выполнены одним процессом. Они включают операции с общей памятью (поместить/извлечь), и операции удаленного накопления.

- Расширение коллективных операций. Расширяет семантику коллективных операций MPI-1. Здесь также добавляются более удобные методы построения интеркоммуникаторов и две новых коллективных операции.

- Введение внешних интерфейсов. Включает обобщенные запросы, подпрограммы, которые декодируют скрытые объекты MPI и потоки.

- Параллельный ввод-вывод.

- Объектно-ориентированный интерфейс для C++ и ФОРТРАН 90.

В стандарт MPI-2 введена возможность динамического порождения процессов и управления ими. Данная модель процессов допускает создание и совместное завершение процессов после запуска приложения MPI. Существует механизм установки соединения между вновь созданными процессами и существующим приложением MPI. Имеется механизм установки соединения между двумя существующими приложениями MPI, даже если одно из них не запускает другое.

Порождение новых процессов происходит из уже запущенных процессов через запуск программ (подобно функциям `exec()` стандарта POSIX.1) с помощью функций `MPI_Comm_spawn`, которая запускает заданное количество копий одной программы, или с помощью `MPI_Comm_spawn_multiple`, которая запускает несколько разных программ. Запущенные с помощью `MPI_Comm_spawn` процессы не принадлежат группе (`MPI_COMM_WORLD`) и выполняются в отдельной среде. Порожденные процессы имеют свои ранги, которые могут совпадать с рангами группы, в которой выполнялся родительский процесс.

Обмен сообщениями между процессами родительской и дочерней групп происходит с использованием так называемого интеркоммуникатора, который возвращается процессу-родителю функцией `MPI_Comm_spawn`. Дочерние процессы могут получить интеркоммуникатор группы родителя с помощью функции `MPI_Comm_get_parent`. Значение интеркоммуникатора используется функциями обмена сообщениями `MPI_Send()`, `MPI_Recv()` и другими.

2.4 Оптимизация программ в модели обмена сообщениями

Программист имеет возможность оптимизировать передачу сообщений в своем программном коде. Оптимизация передачи сообщений сводится: к уменьшению числа сообщений, совмещению вычислений и передачи сообщений (скрытие передачи сообщений за вычислениями), оптимизации алгоритмов передачи сообщений. Отметим наиболее существенные на наш взгляд способы оптимизации MPI-программ.

- *Уменьшение числа сообщений.*
  - *Введение новых типов и передача упакованных данных.* Для передачи многомерных массивов и разнородных данных (структуры, объединения) в MPI вводятся *производные типы* данных, коммуникационные типы на основе базовых типов MPI. Этот механизм может применяться и к разнотипным, и к разноразмерным массивам данных с целью уменьшения числа сообщений. В этом случае несколько массивов передается как одна структура и соответственно число сообщений уменьшается в несколько раз. При этом, латентность нескольких сообщений заменяется латентностью одного. Другой доступный в MPI механизм объединения сообщений — упаковка пересылаемых данных (функция `MPI_Pack`). Данные последовательно упаковываются в одно сообщение в передающем процессе. После получения распаковываются в обратном порядке в принимающем процессе. В обоих случаях требуются дополнительные операции: создание и регистрация нового типа, упаковка и распаковка данных, которые могут оказаться более затратными, чем передача необъединённых сообщений. Поэтому создание и регистрация нового типа не должны выполняться внутри часто вызываемых функций и методов.
  - *Выделение подмножества коммуникационных процессов и создание новых вычислительных процессов.* Изначально создается множество процессов MPI равное по мощности множеству вычислительных узлов (физических процессоров). Эти процессы образуют коммуникационное подмножество процессов, между элементами которого происходят передачи сообщений. Для выполнения вычислений в каждом таком процессе создается множество вычислительных процессов (MPI-2) или нитей (OpenMP).

- *Совмещение вычислений и обменов.* Совмещение вычислений и обменов осуществляется за счет применения функций передачи данных без блокировки (`MPI_Isend`, `MPI_Irecv`), режима передачи сообщений с буферизацией. Предполагает декомпозицию данных на внутренние и общие/внешние. Инициируется передача внешних данных. После инициализации передачи сообщения управление передается обратно в вызвавший процесс, в котором выполняются операции над внутренними данными. Подтверждается прием внешних данных (`MPI_Wait`, `MPI_Test`). Вычисления продолжаются над полученными данными.

- *Оптимизации алгоритмов передачи сообщений.*

  – *Многоуровневая передача сообщений.* Для того, чтобы уменьшить конкуренцию процессов за коммуникационные ресурсы процессы группируются по уровням (уровень вычислительных узлов, для которого создается новый коммуникатор `comm2` и уровень процессоров/ядер — коммуникатор `comm1`). Например, вызов `MPI_Allreduce(...,MPI_COMM_WORLD)` можно заменить на:

```
1 MPI_Reduce(... ,      comm1); // внутри выч. узла
2 MPI_Allreduce(... ,   comm2); // между узлами
3 MPI_Bcast(... ,       comm1); // внутри узла
```

  Многоуровневая передача сообщений требует *привязку процессов* к ресурсам вычислительной системы, т.к. при построении иерархии связей необходимо знать на каком процессоре выполняется данный процесс.

  – *Оптимизация коллективных коммуникаций.* Выбираются коммуникационные алгоритмы наиболее оптимальные в данной реализации MPI, как правило, для некоторого размера сообщений.

2.4.1 Привязка MPI-процессов к ресурсам вычислительной системы

Промежуточное обеспечение MPI не поддерживает полного связывания с CPU, поэтому необходимы системы позволяющие связывать процессы при запуске MPI-приложения. Рассмотрим существующие способы привязки, отличающиеся вариантами задания и взаимодействием с промежуточным программным обеспечением (различные MPI-реализации, планировщики).

Промежуточное программное обеспечение MPICH2 с менеджером ресурсов SLURM

Например, на кластере X4 ИМ УрО РАН задания MPICH2 запускаются менеджером ресурсов SLURM (Simple Linux Utility for Resource Management `https://computing.llnl.gov/linux/slurm/slurm.html`). Запуск приложения осуществляется входящей в SLURM командой `srun`:

```
srun -n64 -w n1,n2 --cpu_bind=map_cpu:0,1,4,5,2,3,6,7 ./a.out
```

Привязка задается значением `cpu_bind` для узлов с именами `n1`, `n2`. Кроме `map_cpu` и `mask_cpu` с явным указанием номеров логических CPU (ядер) `cpu_bind` принимает значения: `sockets`, `cores`, `threads`, `rank`.

Привязка процессов в MVAPICH

В MVAPICH (`http://mvapich.cse.ohio-state.edu/`) привязка процессов выполняется с использованием PLPA (раздел 1.3.9). Привязка может быть определена установкой переменных окружения `MV2_CPU_MAPPING`. Кроме того, привязка процессов в MVAPICH поддерживается в менеджере ресурсов SLURM.

Приложение запускается скриптом `mpirun`:

```
mpirun -np 64 ... параметры привязки ./a.out
```

Привязка в MVAPICH задается парой параметров: `VIADEV_USE_AFFINITY` и `VIADEV_CPU_MAPPING`. Параметр `VIADEV_USE_AFFINITY=1` инициирует привязку MPI-процессов. Распределение задается `VIADEV_CPU_MAPPING`, в котором указываются номера логических процессоров (ядер).
Например `VIADEV_CPU_MAPPING=0,2,4,6,1,3,5,7`.

Для проверки правильности задания привязки в обоих случаях (MPICH2 и MVAPICH) можно вызывать функцию `sched_getaffinity` в каждом MPI-процессе (необходимо включить заголовочный файл `sched.h`).

Привязка процессов в Open MPI

Отметим, что Open MPI (`http://www.open-mpi.org/`) является одной из реализаций MPI, которая содержит собственный механизм привязки процессов. В этом случае, приложение запускается с собственными параметрами привязки Open MPI:

```
mpirun -np 64 ... параметр привязки ./a.out
```

где параметр привязки принимает значение:
`bynode` — привязка к вычислительным узлам по кольцевой схеме;
`byslot` — привязка к процессорам по кольцевой схеме;
`loadbalance` — равномерное распределение процессов;
`slot-list 0,4,1,5,2,6,3,7` — явное соответствие процессов логическим процессорам.

Маска привязки может быть записана в файл и задействована с опцией `rankfile имя файла`. Файл содержит последовательность строк вида: `rank номер процесса=имя узла slot=номер слота:номер ядра`. Строка `rank 2=node2 slot=0:1` означает, что второй процесс, соответствующий коммуникатору `MPI_COMM_WORLD` будет выполняться на узле `node2`, слоте (физическом процессоре) с номером ноль, на первом ядре.

При запуске Open MPI заданий в SLURM, ресурсы для заданий выделяются командой `salloc` (SLURM), а запуск осуществляется `mpirun` с обычным набором аргументов. Список вычислительных узлов и процессов, которые будут выполняться на каждом узле, передается из SLURM.

2.4.2 Оптимизация коллективных коммуникаций

Главное отличие коллективных операций от операций типа «точка-точка» состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. В следствии этого, при запуске процессов MPI на всех процессорных ядрах вычислительной системы использование коллективных операций с коммуникатором `MPI_COMM_WORLD` необходимо передавать сообщения не только по сети (между вычислительными узлами), но и внутри узла (между ядрами процессоров). Кроме того, как правило, каждое процессорное ядро не обеспечено своим аппаратным сетевым каналом, что приводит к конкуренции за сетевые ресурсы системы на передающей и принимающей сторонах. Поэтому, алгоритмы коллективных коммуникаций должны быть строится так, чтобы эффективно использовать иерархию связей процессов. Рассмотрим варианты оптимизации коллективных операций, предлагаемые наиболее популярных реализациях стандарта MPI.

Коллективные коммуникации MPICH2

В MPICH2 (`http://www.mcs.anl.gov/research/projects/mpich2/`) реализуется ADI-3 с коммуникационным устройством `ch3` (третья версия интерфейса `"channel"`), начиная с MPICH2 версии 1.1, по умолчанию применяется метод `Nemesis`. Этот метод использует общую память для передачи

сообщений между процессами одного узла и сеть для процессов с разных узлов.

В методе `Nemesis` коммуникации внутри узла осуществляются при помощи свободных от блокировок очередей (lock-free queues) в общей памяти. Вводится два типа очередей свободные (free queue) и получения или получающие (receive queue). Передающий процесс извлекает элемент из свободной очереди, заполняет его сообщением и помещает его в очередь получения процесса-получателя. Процесс-получатель извлекает элемент из очереди получения, обрабатывает сообщение и помещает элемент обратно в свободную очередь процесса-отправителя. Перечисленные операции происходят в общей памяти, для этого один из процессов каждого узла выделяет область общей памяти для очередей всех процессов (свободных и получения сообщения). Другие процессы отображают эту область общей памяти в свое адресное пространство. Каждый процесс хранит массив указателей на очереди (свободные и получения сообщений) всех остальных процессов. В связи с тем, что область не может быть отображена в некоторые адреса всех процессов, указатели в эту область реализуются как смещения, а не как абсолютные указатели. Здесь под «очередью» понимается структура данных, имеющая начало и конец, при этом извлечение элементов выполняется из начала очереди, а новые элементы добавляются в конец очереди. Примененный в `Nemesis` механизм очередей, при котором каждый процесс имеет свою свободную очередь, обеспечивает хорошую масштабируемость.

Низкая латентность (задержка, связанная с инициализацией процесса передачи) обеспечивается введением дополнительного указателя на начало очереди (shadow head pointer) который хранится в строке кэша, отличной от строки, в которой хранятся начало и конец очереди. Таким образом, уменьшается число неудачных обращений к кэшу второго уровня при доступе к началу и концу очереди. Другим фактором уменьшающим латентность является применение в `Nemesis` буферов с флагом заполнения (fastbox). Сообщения помещаются в очередь только после проверки флага заполнения. Один буфер выделяется для пары процессов на узле.

Кроме того, в MPICH2 имеется несколько алгоритмов для каждой коллективной коммуникации. Например, в `MPI_Bcast` существуют три способа рассылки сообщения всем процессам коммуникатора: бинарное дерево; рассылка частей сообщения и последующий сбор; кольцевая рассылка. Для рассматриваемой функции алгоритмы выбираются следующим образом: для сообщения размером 12 Кб — бинарное дерево (вне зависимости от размера коммуникатора); для коммуникатора, соответствующего боле чем восьми процессам и размера сообщения менее 512 Кб — рассылка частей и последующий сбор; для данных больше 512 Кб — кольцевая рассылка.

Оптимизации коллективных коммуникаций, имеются также в таких реализациях, как MPICH-MX, MPICH-GM, и OpenMPI.

Коллективные коммуникации OpenMPI

Данная реализация MPI имеет отдельную компоненту `coll` настраиваемых коллективных коммуникаций, которая может быть использована при запуске заданий

Компонент коллективных коммуникаций (`coll`) работает со следующими подключаемыми модулями.

- *Основной компонент* (`basic`), который предоставляет выполнение всех внешних и внутренних коммуникаторов. Большинство коллективных функций используют простой(линейный) алгоритм, хотя некоторые применяют как линейный так и биномиальные алгоритм без сегментации. Это компонента взята из LAM MPI.

- *Настраиваемый компонент* (`tuned`), использует несколько алгоритмов для внешних (inter) коммуникаторов. Эти алгоритмы применяют подпрограммы коммуникаций «точка-точка». Алгоритм выбора в данной компоненте может быть выполнен одним из следующих трех способов: компиляцией определенных функций выбора настраиваемых на соединения с высокой пропускной способностью и низкой латентностью; путём определяемых пользователем флагов к командной строке; использованием схем с групповым кодированием которое может быть использовано для настройки определенных систем.

- *Иерархический компонент* (`hierarch`), основанный на использовании коллективных функций на иерархических системах (кластеры с большими SMP узлами). Этот компонент может использовать локальную информацию об испытаниях применения эффективных алгоритмов для некоторого подмножества процессоров в коммуникаторе. Например, если кластер имеет несколько SMP узлов соединенных сетевым интерфейсом MX, и запускается функция `MPI_Bcast`, интерфейс MX будет использован только между узлами, а на узлах для рассылки данных будет использована модель общей памяти.

- *Компонент для общей памяти* (`sm`), включает различные коллективные коммуникации на системах с общей памятью.

Подключение необходимых модулей Open MPI в командной строке происходит при добавлении следующей опции `--mca имя величина`. Так, строка

`mpirun -np 8 --mca coll_hierarch_detection_alg 2 ./a.out` определяет двухуровневую иерархию связей при запуске восьми процессов `a.out`.

2.5 Пример реализации SpMV на MPI

При реализации матрично-векторного произведения используем алгоритм, в котором главный процесс управляет работой других процессов (подчиненных). Главный процесс (master) выполняет только коммуникационные и управляющие операции, а подчиненные осуществляют вычислительные операции и обмениваются данными с главным процессом. Для наглядности текст программы разбит на три части: общую часть (**Листинг 6**), код главного процесса (**Листинг 7**) и код подчиненного процесса (**Листинг 8**). В данном примере единицей вычислительной работы, которая распределяется по процессам, является скалярное произведение строки матрицы **A** на вектор **b**.

Листинг 6. Код общей части программы

```
1  #include <mpi.h>
2  #include <math.h>
3  int main(int argc, char *argv[])
4  {
5  MPI_Status status;
6  MPI_Comm world = MPI_COMM_WORLD;
7  double AV[NNz], b[N], c[N], AVs[N], res;
8  int myid, master=0, nprocs, i, nsent, sender;
9  int ANC[NNz], ANL[N+1], ANCs[N], residx, row;
10 MPI_Init( &argc, &argv );
11 MPI_Comm_rank( world, &myid );
12 MPI_Comm_size( world, &nprocs );
13 ... // инициализация A и b в процессе master
14 // рассылка b и ANL каждому подчиненному процессу
15 MPI_Bcast (b, N, MPI_DOUBLE, master, world);
16 MPI_Bcast (ANL, N+1, MPI_INT, master, world);
17 if ( myid == master )
18 {
19 ... // код главного процесса
20 }
21 else
22 {
23 ... // код подчиненного процесса
```

```
24 }
25 MPI_Finalize();
26 return 0;
27 }
```

В общей части программы вводятся: матрица **A**, хранящаяся в формате CSR (см. раздел 1.4.4), вектор **b**, вектор результата **c**, определяется число процессов (не меньше двух). Код главного процесса представлен в **Листинге 7**. Сначала, главный процесс передает массив `b` и массив `ANL` в каждый подчиненный процесс, затем пересылает одну строку матрицы **A** в каждый подчиненный процесс. Главный процесс, получая результат от очередного подчиненного процесса, передает ему новую работу. Цикл заканчивается, когда все строки будут распределены и получены результаты. При передаче данных из главного процесса в параметре `tag` указывается номер передаваемой строки. После вычисления произведения номер строки вместе с результатом отправляется в главный процесс.

Листинг 7. Код главного процесса

```
1 // посылка строки каждому подчиненному процессу
2 ib=ANL[0];
3 int ii = min(nprocs+1, rows);
4 for(i=0; i< ii; i++)
5 { ie=ANL[i+1];
6     MPI_Send(AV+ib, ie-ib, MPI_DOUBLE, i, i, world);
7     MPI_Send(ANC+ib, ie-ib, MPI_DOUBLE, i, i, world);
8   ib=ie;
9 }
10 nsent=i-1; //число посланных строк
11
12 for(i=0;i<N;i++) // прием результатов
13 { MPI_Recv(&res,1,MPI_DOUBLE,MPI_ANY_SOURCE,
14                    MPI_ANY_TAG, world,&status);
15   sender=status.MPI_SOURCE; // номер процесса-отправителя
16   residx=status.MPI_TAG; // номер строчного индекса
17     c[residx] = res; // записываем результат умножения
18  if (nsent < rows) // посылка следующей строки
19  { nsent++;
20   MPI_Send(AV+ANL[nsent],ANL[nsent+1]-ANL[nsent],
21                              MPI_DOUBLE, sender, nsent, world);
22   MPI_Send(ANC+ANL[nsent],ANL[nsent+1]-ANL[nsent],
23                              MPI_INT, sender, nsent, world);
```

```
24  }
25  else // посылка признака конца работы
26      MPI_Send(MPI_BOTTOM,0,MPI_DOUBLE,sender,0,world);
27 }
```

В общей части программы каждый подчиненный процесс получает массив `b` и массив `ANL`, хранящий начальные позиции строк для `AV` и `ANC`. Далее выполняется цикл, в котором подчиненный процесс: получает очередную строку матрицы **A**, вычисляет скалярное произведение строки и вектора **A**, посылает результат главному процессу, получает новую строку и так далее. Подчиненные процессы посылают результаты в главный процесс, а параметр `MPI_ANY_TAG` в операции приема главного процесса указывает, что результаты принимаются в любой последовательности.

Листинг 8. Код подчиненного процесса

```
1  // выход, если процессов больше чем число строк матрицы
2  if(nprocs < N+1)
3  {// прием строки матрицы
4    while(1)
5    {
6      MPI_Probe(master, world, &status);
7        row = status.MPI_TAG; // номер полученной строки
8      if (row == 0) break;      // конец работы
9      else
10     { // получение строки
11         ib=ANL[row]; ie=ANL[row+1];
12       MPI_Recv(AVs,ie-ib,MPI_DOUBLE,master,MPI_ANY_TAG,
13                               world,&status);
14       MPI_Recv(ANCs,ie-ib,MPI_INT,master,MPI_ANY_TAG,
15                               world,&status);
16 // скалярное произведение векторов
17     for(i=ib,res=0; i<ie; i++) res+=AVs[i-ib]*b[ANCs[i]];
18 // передача результата головному процессу
19       MPI_Send (&res,1,MPI_DOUBLE,master,row,world);
20     }
21   }
22 }
```

Переменная `status` обеспечивает прием атрибутов сообщения. В функциях MPI атрибуты сообщения помещаются в аргумент типа `MPI_Status`, применительно к C/C++ — это структура с тремя полями `MPI_SOURCE`, `MPI_TAG`

и MPI_ERROR. Поле MPI_SOURCE содержит номер процесса, который послал сообщение, по этому адресу главный процесс будет пересылать новую работу. В поле status.MPI_TAG из аргумента tag, соответствующие функции MPI_Send передается номер обработанной строки, что обеспечивает размещение полученного результата. После того как главный процесс разослал все строки матрицы **A**, на запросы подчиненных процессов он отвечает сообщением с аргументом tag равным нулю (см. **Листинг 7**).

При небольшом числе процессов функциональность главного процесса возможно совместить с вычислениями подчиненного процесса. Это может быть сделано, как на уровне программного кода, добавлением скалярного произведения в процесс master, так и при помощи привязки процессов к ядрам вычислительной системы (см. раздел 2.4.1), запуская на ядре процесс master и подчиненный процесс.

2.6 Объектно-ориентированные интерфейсы MPI

Первым этапом введения объектной ориентированности в параллельные вычисления на MPI является построение модели классов самой системы MPI. Существует много попыток ввести поддержку C++ в MPI. Рассмотрим реализации MPI++ (http://www.netlib.org/mpi/oon-ski.ps) и OOMPI (https://www.osl.iu.edu/download/research/oompi/oompi.pdf). Эти библиотеки инкапсулируют функциональность MPI в иерархию классов для обеспечения простого, переносимого и интуитивно понятного интерфейса. При помощи наследования пользователи могут создавать свои прикладные классы. В данных системах предложена модель, состоящая из объектов «коммуникатор», «группа коммуникаторов», «сообщение» и некоторых других.

В **Листинге 9** представлен пример передачи сообщения в OOMPI с использованием «портов» OOMPI и коммуникаций, основанных на потоках C++.

Такой подход позволяет создавать достаточно простые, гибкие и расширяемые прикладные программы, но проблема разрушения объектных моделей при использовании сообщений остаётся. Чтобы избежать этого, разработчикам необходимо создавать значительное количество кода, связанного с трансляцией пересылаемых объектов в сообщения и маршалингом вызываемых методов объектов. Более подробно об этих библиотеках см. в [5]. Продолжением этих подходов стали привязки (bindings) к языку C++ в стандарте MPI-2. Наличие объектного интерфейса не означает передачу объектов, поэтому разработчиками ПО предпринимались попытки решения этой проблемы.

Листинг 9. Пример передачи сообщения в OOMPI

```
#include <iostream>
#include "oompi.h"
using namespace std;

int main(int argc, char *argv[])
{
int msg;
OOMPI_COMM_WORLD.Init(argc, argv);
int rank = OOMPI_COMM_WORLD.Rank();
int size = OOMPI_COMM_WORLD.Size();
OOMPI_Port to =   OOMPI_COMM_WORLD[rank + 1];
OOMPI_Port from = OOMPI_COMM_WORLD[rank - 1];

if (!rank) { msg = rank; to << msg;}
else if (rank == 1) from >> msg;

OOMPI_COMM_WORLD.Finalize();
return 0;
}
```

2.7 Интерфейсы передачи объектных сообщений

Решения проблемы передачи объектных сообщений предложены в системах TPO++ (`http://tpo.sourceforge.net/doc/tpo_ug.pdf`) и Charm++ (`http://charm.cs.illinois.edu/manuals/pdf/charm++.pdf`). Данные системы включают наборы классов, которые могут сопоставляться с какими-либо объектами, участвующими в межпроцессорных взаимодействиях. Путем наследования от этих классов, реализующих между собой объектные обмены на основе раннего связывания, можно создавать объекты для конкретной прикладной задачи. Объектная модель поддерживается полностью, но она достаточно бедная, поэтому предполагается, что пользователи будут расширять ее посредством наследования. К сожалению, наследование не удовлетворяет полностью требованиям гибкости программного обеспечения: при расширении системы требуется ее полная перекомпиляция.

Главным нововведением является процедура маршалинга [5], состоящая во взаимодействии с удаленным объектом. Эта процедура хорошо формализуется и может быть автоматизирована, но в данных системах

ее нужно заново определять для новых объектов. Кроме этого, не реализована общая модель клиент-сервер. Клиент-серверный подход подразумевает параллельную обработку серверным объектом действий, вызываемых несколькими клиентами одновременно. Потоки (нити, threads) — основа для реализации модели клиент-сервер и распределенных объектов, когда необходим динамический запуск дополнительных потоков команд на одном процессоре.

2.7.1 TPO++

Проект TPO++ — это объектно-ориентированная библиотека передачи сообщений, написанная на C++ поверх MPI. Ее главные особенности – простая передача объектов, сохранение типов (type-safety), MPI-совместимость и интегрирование с библиотекой STL (Standard Template Library - стандартная библиотека шаблонов).

Следующий пример показывает как передается и принимается контейнер vector библиотеки STL при помощи TPO++ (см. **Листинг 10**).

Листинг 10. Пример передачи STL контейнера в TPO++

```
1 #include <tpo++.h>
2 int main(int argc, char *argv[])
3 {
4 TPO::init(&argc, &argv);
5    if(TPO::CommWorld.rank()==0){
6          vector<double> VD(10);
7          fill(VD.begin(), VD.end(), 3.14);
8         TPO::CommWorld.send(VD.begin(), VD.end(), 1);
9    }
10   else {
11         vector<double> VD(10);
12        TPO::CommWorld.recv(VD.begin(), VD.end());
13   }
14 }
```

В этом примере вектор VD пересылается из процесса номер ноль в процесс номер один.

Разработчики TPO++ ставили перед собой следующие цели.

- *Объектная ориентация и сохранение типов.* Главная цель объектно-ориентированной библиотеки классов, поддерживающей передачу сообщений(message-passing) — это тесная интеграция с объектно-ориентированными концепциями, в особенности передача объектов в

простой, эффективной и безопасной (type-safety) манере. Решение не должно нарушать существующие возможности ООП, такие как инкапсуляция и наследование.

- *Интеграция с* C++. Реализация на C++ должна принимать во внимание все новые возможности языка, такие как использование исключений для обработки событий и даже, что по мнению авторов более важно, интеграцию со стандартной библиотекой шаблонов STL (поддерживаются STL контейнеры и соглашения об интерфейсах).

- *MPI совместимость.* Разработка должна соответствовать MPI интерфейсу, соглашению имен и семантически быть настолько близкой, насколько это возможно без нарушения объектно-ориентированных концепций. Это помогает перестраивать C программы и легко переносить существующий C и C++ код.

- *Эффективность.* Реализация должна не слишком отличаться от MPI в терминах передачи сообщений и эффективности памяти. Поэтому она должна быть легкой, насколько это возможно, позволяя C++ компилятору статически убирать почти все интерфейсы верхнего уровня. Коммуникации должны быть реализованы через MPI вызовы и, если это возможно, не использовать дополнительные буферы, которые сохраняют память, и позволять низкоуровневой MPI реализации оптимизировать коммуникации, например, если сетевое оборудование способно получать и посылать рассеянные блоки памяти автоматически.

Поставленные цели достигаются с помощью техники трейтов (trait). Техника трейтов позволяет писать общий код, определяя класс не по типу, как в простых шаблонах, а по частным характеристикам типа. Основываясь на таких характеристиках компилятор может генерировать специфический код. Главное решение заключается в том, чтобы не задавать эти характеристики как новый параметр в обычных шаблонах, который может раздражать пользователя, обычно не интересующегося деталями реализации, а вместо этого определить новый шаблон. Этот шаблон — трейт шаблон — может использоваться для внутреннего накопления специфической информации об актуальных параметрах шаблона. Любые коммуникационные методы верхнего уровня, дающие информацию о типе, можно однозначно заменить последовательностью базовых типов. Важно отметить, что большинство преобразований статично, то есть может быть выполнено во время компиляции, поэтому можно быть уверенным в сохранении эффективности нижележащего MPI кода.

Данной информации достаточно для запаковки данных в коммуникационный буфер или для создания MPI-типа для передачи. Реализация TPO++ адресуется только к однородным архитектурам. Никакая информация о типах не отображается в коммуникациях. Блоки памяти, полученные преобразованиями, передаются напрямую. Информация о типе, полученная во время приведения может использоваться для быстрого построения MPI-типов данных. Если бы оптимизация для тривиальных конструкторов была опущена, то это также обобщило бы реализацию на гетерогенную архитектуру.

Хотя в этой библиотеке реализована достаточно эффективная передача данных объекта, в ней все-таки существуют много нерешенных проблем. Первая – это сложность пересылки динамически определенных пользовательских типов данных. Для этого требуется в каждом таком типе определять маршалинг его данных. Вторая, наиболее важная для распределенных систем проблема, это удаленный вызов методов объектов. В TPO++ он не реализован. Все методы объекта вызываются только на процессоре, на котором находится сам объект. Для того чтобы вызвать его метод на другом процессоре, нужно переслать туда все его данные. В библиотеке TPO++ нет понятия распределенного объекта, что необходимо для реализации полностью объектно-ориентированной распределенной параллельной системы.

2.7.2 Charm++

В системе Charm++ для процедуры маршалинга предлагается модель актера (Actor) [5], включающая классы актера и заместителя, которые могут сопоставляться с какими либо объектами, участвующими в межпроцессорных взаимодействиях. Путем наследования от этих классов, реализующих между собой маршалинг на основе раннего связывания, можно создавать объекты для конкретной прикладной задачи. Charm++ построена на базе языка программирования C++. Подобно C++, Charm++ объекты могут содержать приватные данные и публичные методы. Различие заключается в том, что эти методы могут вызываться с удаленных процессоров асинхронно. Асинхронный вызов метода означает, что вызвавший его не ждет, когда метод действительно выполнится и вернет результат. Поэтому Charm++ методы, названные entry методами, не имеют возвращаемых значений. Так как реальный Charm++ объект, у которого вызывается на выполнение метод, может находится на удаленном процессоре, возможно, в другом адресном пространстве, способ адресации C++ к объекту, такой как указатель C++, не действителен в Charm++. Вместо этого существуют

два способа адресации к удаленному объекту. Первый — каждый объект имеет уникальный дескриптор, названный `ChareID`. Это структура, уникально определяющая адрес объекта на параллельной машине. Отметим, что она одинакова для любых типов объекта (его класса). Второй способ — это ссылка на объект через его прокси.

Для каждого объектного типа существует его прокси (фактически, генерируемый интерфейсным компилятором), который содержит ссылку, например, `ChareID`, на реальный объект. Кроме того, методы прокси класса сообщаются с методами реального класса и действуют как опережающие. То есть, когда метод вызывается на прокси удаленного объекта, он переправляет этот вызов реальному объекту. Прокси возвращает результат созданием нового удаленного объекта. Все объекты, создаваемые и управляемые удаленно, в Charm++ имеют прокси. Прокси для каждого объекта в Charm++ имеют некоторые различия из-за особенностей ими поддерживаемых, но основной синтаксис и семантика почти всегда одинаковы, и это позволяет вызывать методы на удаленных объектах с помощью их прокси.

Единственные методы, которые могут вызываться с других процессоров в Charm++ – это асинхронные `entry` методы, применяющиеся для удаленного вызова методов объектов `chare`. Для этого, ядро должно знать о типах chare в программе, о методах, которые могут запускаться с удаленных процессоров, их аргументах и т.д. Поэтому при старте программы эти пользовательские объекты нужно зарегистрировать в runtime системе Charm Kernel, который назначает каждому из них уникальный идентификатор. При вызове методов на удаленном объекте эти идентификаторы должны быть указаны системе. Регистрация пользовательских объектов и поддержание этих идентификаторов может быть очень громоздким. Поэтому, в Charm++ добавлен интерфейсный транслятор. Он генерирует определения прокси объектов. Кроме того, интерфейсный транслятор позволяет расширять базовую функциональность ядра Charm++ введением потоков и future-объектов пользовательского уровня. Эти потоковые entry методы могут блокироваться в ожидании данных, выполнением синхронного вызова методов удаленного объекта, которые возвращают данные в виде сообщения.

3 Многопоточные вычисления на OpenMP

Стандарт OpenMP (`http://openmp.org/wp/`) был разработан как API, ориентированный на написание портируемых многопоточных приложений. Сначала он был основан на языке Fortran, но позднее включил в себя и C/C++. Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP, ccNUMA и др.) в модели общей памяти (shared memory model).

В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

3.1 Модель параллельной программы

OpenMP использует модель программирования для общей памяти и неявные коммуникации [14].

OpenMP представляет собой многофункциональный набор прагм, с помощью которых можно распараллелить циклы, фрагменты кода, вызовы функций/подпрограмм.

Программа, использующая OpenMP, делится на последовательные и параллельные участки. В момент запуска порождается основной поток, в рамках которого будут выполнены все последовательные участки кода. Когда работа программы доходит до параллельной области, основной поток порождает необходимое количество дополнительных, которые запускаются параллельно на различных процессорах/ядрах. По окончании параллельной секции, основной поток ожидает завершения остальных, и дальнейшие действия выполняет только он. Конечно, такая организация работы требует сложной логики синхронизации потоков выполнения и разграничения доступа к переменным.

OpenMP прост в использовании и включает лишь две группы программных конструкций: директивы `#pragma omp` и функции исполняющей среды OpenMP.

3.2 Директивы и функции

Директивы OpenMP начинаются с `#pragma omp` и построены на основе директивы `#pragma`, которая позволяет передавать компилятору различные инструкции. Как и любые другие директивы `pragma`, они игнорируются компилятором, не поддерживающим OpenMP.

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API-функции для поддержки некоторых типов синхронизации. Чтобы задействовать функции

библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл `omp.h`.

Для создания параллельного приложения необходимо просто добавить в код директивы OpenMP и, если нужно, воспользоваться функциями библиотеки периода выполнения.

Директивы `pragma` имеют следующий формат:

`#pragma omp <директива> [раздел [ [,] раздел]...]`

OpenMP поддерживает директивы `parallel`, `for`, `parallel for`, `section`, `sections`, `single`, `master`, `critical`, `flush`, `ordered` и `atomic`, которые определяют или механизмы разделения работы, или конструкции синхронизации.

Разрабатывая параллельные программы с OpenMP, необходимо понимать, какие данные являются общими (shared), а какие частными (private) от этого зависит не только производительность, но и корректная работа программы.

Общие переменные доступны всем потокам из группы, поэтому изменения таких переменных в одном потоке видимы другим потокам в параллельной области. Что касается частных переменных, то каждый поток из группы располагает их отдельными экземплярами, поэтому изменения таких переменных в одном потоке никак не сказываются на их экземплярах, принадлежащих другим потокам.

При одновременном выполнении нескольких потоков часто возникает необходимость их синхронизации. OpenMP поддерживает несколько типов синхронизации, помогающих во многих ситуациях.

Один из типов — неявная барьерная синхронизация, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что, пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.

Второй тип — явная барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Для этого включите в код директиву `#pragma omp barrier`. В качестве барьеров можно использовать критические секции.

OpenMP включает и функции, предназначенные для синхронизации кода. В OpenMP два типа блокировок: простые и вкладываемые (nestable); блокировки обоих типов могут находиться в одном из трех состояний — неинициализированном, заблокированном и разблокированном.

3.3 Параллельные циклы

Как правило, OpenMP используется для распараллеливания циклов, но OpenMP поддерживает параллелизм и на уровне функций. Этот механизм называется секциями OpenMP (`OpenMP sections`).

Весь свой потенциал OpenMP демонстрирует именно при организации параллельного выполнения циклов. Если в приложении есть длительные циклы без зависимостей, OpenMP — идеальное решение.

По умолчанию в OpenMP для планирования параллельного выполнения циклов `for` применяется алгоритм, называемый статическим планированием (static scheduling). Это означает, что все потоки из группы выполняют одинаковое число итераций цикла. Если N — число итераций цикла, а N_{th} — число потоков в группе, каждый поток выполнит N/N_{th} итераций. Однако OpenMP поддерживает и другие механизмы планирования, оптимальные в разных ситуациях: динамическое планирование (dynamic scheduling), планирование в период выполнения (runtime scheduling) и управляемое планирование (guided scheduling).

В OpenMP существует ряд ограничений на параллельные циклы:

- переменная цикла должна иметь тип `int`. Беззнаковые целые числа, такие как `DWORD`, не работают;

- цикл должен являться базовым блоком и не может использовать `goto` и `break` (за исключением оператора `exit`, который завершает все приложение);

- инкрементная часть цикла `for` должна являться либо целочисленным сложением, либо целочисленным вычитанием, и должна практически совпадать со значением инварианта цикла;

- если используется операция сравнения `<` или `<=`, переменная цикла должна увеличиваться при каждой итерации, а при использовании операции `>` или `>=` переменная цикла должна уменьшаться.

3.4 OpenMP в объектно-ориентированных программах

В C и C++ программах часто встречаются циклы, которые не могут быть распараллелены с использованием конструкции цикла-worksharing, потому что они не соответствуют OpenMP спецификации.

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного

цикла выполнит. Нельзя использовать дополнительный выход из параллельного цикла. Размер блока итераций, указанный в опции `schedule`, не должен изменяться в рамках цикла.

Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций.

Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл.

Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она определена общей, то она неявно становиться локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции `lastprivate`.

При распараллеливании цикла необходимо убедиться в том, что итерации данного цикла не имеют информационных зависимостей. Если цикл не содержит зависимостей, его итерации можно выполнять в любом порядке, в том числе параллельно. Соблюдение этого важного требования компилятор не проверяет, вся ответственность лежит на программисте. Если дать указание компилятору распараллелить цикл, содержащий зависимости, компилятор это сделает, но результат работы программы может оказаться некорректным.

Возможны различные подходы для работы с циклами в C++:

- в первом подходе создается распараллеленный цикл с дополнительным циклом, в котором `iterator` указателя сохранен в массиве. Обработка этого массива может быть распараллелена с использованием конструкции цикла-worksharing;

- второй подход используется в компиляторах Intel для организации очереди задачи Intel worksharing конструкция. Для каждого значения цикла индексируют переменную, тело цикла установлено в очередь работы, которая обрабатывается параллельно всеми потоками;

- в третьем подходе цикл помещается в параллельный регион и тело цикла в единственную worksharing-конструкцию, и неявный барьер опускается, определяя нет ли ожидания;

- в четвертом подходе значение увеличенного `iterator` назначено в критическом интервале на частный `iterator`, и частный `iterator` используется, чтобы вызвать вычисляющуюся функцию.

3.5 Привязки потоков в OpenMP

Технология OpenMP реализована во многих версиях компиляторов для эффективного использования возможностей многопроцессорных систем, а также процессоров использующих технологию многопоточности.

OpenMP — мощная технология распараллеливания приложений. Она позволяет реализовать параллельное выполнение как циклов, так и функциональных блоков кода. Она легко интегрируется в существующие приложения и включается одним параметром компилятора. OpenMP позволяет более полно использовать вычислительную мощь многоядерных процессоров.

В настоящее время в стандарте OpenMP не существует средств прямого определения привязки потока к процессору. Однако известен метод для задания привязки потоков через переменные среды `KMP_AFFINITY` только для компиляторов Intel C/C ++. Данная переменная по существу разрешает определять три возможные реализации: должны ли потоки быть привязаны к отдельным ядрам или должны перемещаться среди имеющихся ядер процессора; будет ли учитываться маска привязки потоков задаваемая по умолчанию; будут ли последовательные потоки располагаться на соседних ядрах (для оптимизации эффектов кэша памяти).

Возможность контроля привязки потоков к процессорам на уровне компилятора реализована PathScale Compiler (`http://www.pathscale.com`), так же через переменную окружения `PSC_OMP_AFFINITY`, которая должна иметь значение `TRUE`.

Основным подходом к привязкам для потоков OpenMP остается вызов системных функций PLPA (раздел 1.3.9) из текущего потока.

Выделим также основные достоинства OpenMP:

- относительная простота применения. Директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области;

- нет необходимости создавать две программы: для одноядерной архитектуры и отдельно для многоядерной (разделить эти варианты можно например с помощью заглушек — `stubs`);

- теоретически, лучше использовать для МВС с общей памятью;

- эффективен линейный и мелкозернистый параллелизм;

- технология совместима с компиляторами, не поддерживающими OpenMP.

Отметим и главные недостатки OpenMP:

- OpenMP рассчитан на мультипроцессоры и DSM-системы (системы с распределенной памятью, на которых смоделирована общая память) и изначально не ориентирован на кластеры;
- на DSM-системах большого размера (64 и выше), например SGI Altix, эффективность OpenMP-программ невысока, что заставляет программистов использовать MPI;
- организация взаимодействия потоков через общие переменные, а не через передачу сообщений, часто приводит к трудно обнаруживаемым ошибкам (race condition — условия гонок), а необходимые для поиска таких ошибок средства отладки — либо отсутствуют вообще, либо мало доступны;
- для крупнозернистого параллелизма часто требуется стратегия распараллеливания, подобная стратегии MPI, а также внешняя синхронизация, что иногда бывает накладно и трудоёмко реализовать средствами OpenMP.

Создание как обычных потоков, так и параллельных регионов OpenMP имеет свою цену. Чтобы применение OpenMP стало эффективным, выигрыш в скорости, обеспечиваемый параллельным регионом, должен превосходить издержки на создание группы потоков.

OpenMP-директивы `#pragma` просты в использовании, но не позволяют получать детальные сведения об ошибках. Если создаётся критически важное приложение, которое должно определять ошибки и корректно восстанавливать нормальную работу, от OpenMP, пожалуй, следует отказаться.

3.6 SpMV в модели общей памяти

Программная реализация **Алгоритма 2** при помощи OpenMP возможна, как на уровне функции, обращающейся к методу `SpM::SpMV` (**Листинг 4**), так и в самом методе (**Листинг 3**).

В первом случае метод `SpMV` вызывается в каждом потоке OpenMP (см. **Листинг 11**), а границы внешнего цикла данного метода `ib` и `ie` зависят от номера потока `myid`. Для двух — четырёх потоков такой вариант распараллеливания может быть также реализован с помощью директивы `#pragma omp sections`.

Листинг 11. Вызов метода SpMV в нескольких потоках OpenMP

```
1  // Файл mainomp.cpp
2  #include <omp.h>
3  #include <iostream>
4  #include "spm.h"
5
6  using namespace std;
7  const int N=8, NNZ=64;
8
9  void filling(SpM &);
10
11 int main(int argc, char *argv[])
12 { /* Инициализация векторов b,c и матрицы A */
13 vector <double> b(N,1), c(N,1);
14 SpM A(N,NNZ);
15     filling(A); // Заполнение A
16  int NThs=1, myid=0;
17 #ifdef _OPENMP
18     NThs=omp_get_num_procs(); // Число потоков OpenMP
19 #endif
20 /*  Произведение c = A * b           */
21  #pragma omp parallel num_threads(NThs)
22  {
23 #ifdef _OPENMP
24     myid = omp_get_thread_num(); // номер потока
25 #endif
26    int  ib = (N*myid)/NThs;
27    int  ie = (myid+1-N) ? ((N*(myid+1))/NThs) : N;
28     A.SpMV(b,c,ib,ie);
29  cout << "myid="<< myid << endl;
30  }
31 // Проверка результата
32 return 0;
33 }
```

Число потоков в данном примере задаётся равным числу ядер процессоров вычислительной системы при помощи функции `omp_get_num_procs()` и опции `num_threads(NThs)` директивы `omp parallel`. Код для проверки результата может быть взят из последовательной программы (**Листинге 4**) без изменений.

Второй вариант основан на применении директивы `#pragma omp for` в самом методе `SpM::SpMV`, так в коде, показанном в **Листинг 12** применен сокращенный вариант записи `#pragma omp parallel for`. Для распределения вычислительной нагрузки использован статический режим с размером `size` блока строк матрицы. Кроме указанных изменений в файл `spm.h` необходимо также добавить строку `#include <omp.h>`.

Листинг 12. Распараллеливание цикла внутри метода SpMV

```
// Файл spmomp.h
void SpM::SpMV (vector <double> &b, vector <double> &c,
                                   int ib, int ie)
{
#ifdef _OPENMP
int NThs=omp_get_max_threads(); //Число потоков OpenMP
int size=(ie-ib)/NThs;
cout << "NThs=" << NThs << endl;
 #pragma omp parallel for schedule(static,size)
#endif
    for(int i=ib; i<ie; i++)
    {
        double sum=0;
        int jb=ANL[i]; int je=ANL[i+1];
        for(int j=jb;j<je;j++)
            sum+= AV[j]*b[ANC[j]];
        c[i]=sum;
    }
}
```

Как видно, из приведенных примеров OpenMP позволяет достаточно просто получить параллельный исходный код из уже существующего последовательного кода. При создании исполняемого кода компилятору `g++` необходимо всего лишь указать ключ `-fopenmp`. Например, запуск командной строки `g++ -O2 -fopenmp mainomp.cpp -o mainomp` создаст исполняемый файл `mainomp`, из файла `mainomp.cpp`, код которого приведен в **Листинге 11**. Макроопределениями `_OPENMP` выделены обращения к функциям OpenMP периода выполнения, чем обеспечивается возможность компиляции как параллельной, так и последовательной программы (достаточно исключить ключ `-fopenmp` из командной строки).

4 Массивно-параллельные вычисления в технологии CUDA

Современные вычислительные узлы, как правило, содержат кроме многоядерного центрального процессора высокопараллельные графические процессоры (GPU — Graphics Processing Units), которые могут обеспечить существенное увеличение производительности при вычислениях общего назначения. При использовании видеоускорителей для проведения расчетов в прикладных задачах, говорят о вычислениях общего назначения на GPU (GPGPU — General-Purpose computing on GPU). Гетерогенность такой вычислительной системы учитывается за счет разбиения кода программ на часть, исполняемую на центральном процессоре, и часть, выполняемую на графических процессорах. В этом случае, реализуется модель параллелизм данных (Data Parallel), когда часть программы, выполняемая на CPU, задает распределение данных между процессорами GPU. Другая часть, выполняемая на GPU, осуществляет массивно-параллельные вычисления с мелкозернистым, практически атомарным параллелизмом.

Графические процессоры аппаратно предназначены для интенсивного выполнения вычислений и являются мощными SIMD процессорами. Поэтому программы с интенсивным обращением к памяти или сложной логикой будут выполняться на них неэффективно, т.к. GPU обладает слабыми средствами кэширования обращения к памяти (к тому же их требуется настраивать вручную) и «не переносит» ветвлений в программе, особенно если потоки одного блока расходятся по разным веткам.

4.1 Архитектура GPU

В вычислениях общего назначения на GPU из всех систем графического ускорителя (видеокарты) наиболее существенное значение имеют графический процессор и видеопамять. Современный графический процессор NVIDIA включает в себя от одного до шестнадцати потоковых мультипроцессоров. Вместе с тем, каждый потоковый мультипроцессор содержит несколько десятков шейдерных процессоров, которые также называются процессорные ядра CUDA (CUDA cores). Таким образом, графический ускоритель может содержать сотни процессорных ядер CUDA, например, видеокарта GeForce GTX 590 содержит два графических процессора GF110, в которых суммарно 1024 процессорных ядра CUDA.

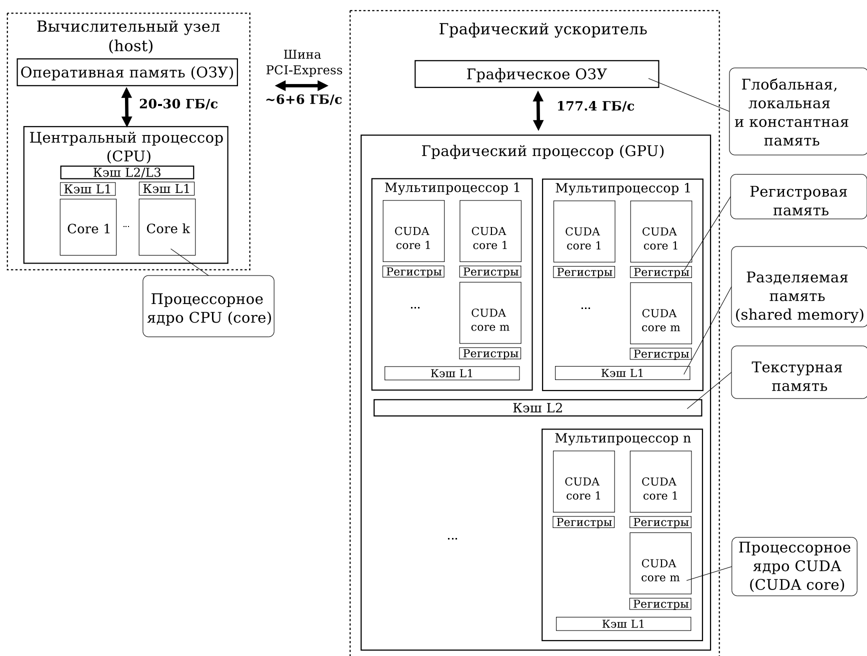

Рис. 5. Вычислительный узел с графическим ускорителем

В многоядерном CPU число процессорных ядер $k < m$, где m — число ядер CUDA в одном мультипроцессоре (см. рисунок 5). Вместе с тем, как правило, число CPU в вычислительном узле не превышает четырех, в то время как число мультипроцессоров n в одном графическом процессоре может достигать шестнадцати. Общее число ядер CUDA в современном графическом ускорителе может достигать $2 \cdot n \cdot m >> k$.

Разделяемая память принадлежит мультипроцессору, регистровая — процессору, текстурный и константный кэши — находятся в памяти устройства. Унифицированный кэш L2 в процессорах Fermi заменяет собой текстурный кэш, а также различные буферы и используется как для записи, так и чтения данных. Система памяти CUDA будет рассмотрена подробнее в разделе 4.2.3, а здесь следует отметить существенное отличие в пропускной способности CPU — оперативная память, оперативная память — графическое ОЗУ (более известна как GDDR), графическое ОЗУ — графический процессор (рисунок 5). Узким местом является связь оперативная память — графическое ОЗУ.

Процессорное ядро CUDA, или унифицированный шейдерный процессор, представляет собой скалярный процессор общего назначения для обработки целочисленных данных и данных с плавающей запятой. Шейдерные процессоры являются унифицированными в том смысле, что способны выполнять графические вычисления и вычисления общего назначения. Следует отметить, что процессорные ядра CUDA работают на удвоенной частоте графического процессора.

Традиционно в процессорах существуют два типа инструкций: векторные и скалярные. В случае векторных инструкций данные (операнды) представляются в виде n-мерных векторов, при этом над большим массивом данных проводится всего одна операция. Скалярные инструкции осуществляются над парой чисел. Очевидно, что векторная обработка увеличивает скорость обработки данных за счет того, что обработка набора данных (вектора) выполняется одной командой. В то же время векторная архитектура менее эффективно использует вычислительные ресурсы, особенно при обработке сложных смешанных шейдеров, сочетающих векторные и скалярные инструкции. Кроме того, сложно добиться эффективной обработки скалярных операций с помощью векторных исполнительных модулей. Именно поэтому в унифицированных процессорах используются скалярные исполнительные блоки.

Мультипроцессоры также могут включать от четырех до восьми специальных блока SFU (Special Function Unit), которые используются для математических расчетов. Эти блоки способны выполнять такие математические операции, как синус, косинус, квадратный корень и т.п. Блок SFU способен выполнять одну математическую операцию на поток за такт.

Размещение потоков по мультипроцессорам осуществляет планировщик GigaThread Engine, который создает группы из 32 потоков (warp-ы), и распределяет их по разным мультипроцессорам. В свою очередь, в каждом мультипроцессоре имеются два планировщика для warp-ов (Warp Scheduler), которые работают параллельно, что позволяет выбирать одновременно два warp-а для выполнения на мультипроцессоре. Планировщики передают по одной инструкции от каждого warp-а группе из шестнадцати процессорных ядер CUDA или четырех SFU.

Современный мультипроцессор может выполнять до восьми потоковых блоков, 24/32/48 warp-ов, или 768/1024/1536 потоков. Число потоков в блоке — 512 (для устройств с Compute capability 1.0–1.3) и 1024 (для устройств с Compute capability 2.x). Так, например, на видеокарте GeForce GTX 560 Ti (устройство с Compute capability 2.1) установлен один графический процессор GF110, который содержит восемь мультипроцессоров, поэтому на данном ускорителе могут выполняться параллельно $8 \times 1536 = 12288$ пото-

ков CUDA. Понятие Compute capability введено компанией NVIDIA, чтобы классифицировать, применительно к CUDA, вычислительные возможности графических ускорителей.

4.2 Технология CUDA

Предложенная nVidia технология CUDA (Compute Unified Device Architecture) `http://www.nvidia.ru/object/what_is_cuda_new_ru.html` предназначена для разработки GPGPU-приложений, выполняемых на графических вычислительных устройствах. Полагается, что GPU (называемый device или устройство) является массивно-параллельным сопроцессором к CPU (называемый host), обладает собственной памятью и возможностью запуска значительного числа потоков. Устройства, обладающие вычислительными возможностями, начиная с версии 1.3 (Compute capability 1.3) аппаратно поддерживают вычисления с двойной точностью.

Программа на CUDA задействует и CPU, и GPU. Последовательная часть кода выполняется на CPU, а массивно-параллельные вычисления выполняются множеством одновременно выполняемых потоков (threads) на GPU. Потоки, выполняемые на GPU характеризуются существенно меньшими затратами на создание, управление и удаление, чем потоки на CPU. Поэтому для эффективной загрузки GPU требуются тысячи потоков.

В CUDA вводятся расширения языка C, которые включают [15–17]:

- спецификаторы функций (см. таблицу 1), показывающих где будет выполняться функция и откуда она может быть вызвана;
- спецификаторы переменных, определяющие тип памяти, используемый для данных переменных;
- директиву запуска ядра, задающую иерархию нитей;
- встроенные переменные, содержащие информацию о текущей нити;
- CUDA host API, включающий в себя функции, которые могут быть использованы только CPU (управления GPU и выполнением кода; работу с памятью, текстурами и т.д.), и дополнительные типы данных.

В CUDA вводится понятие ядро (kernel) — функция, которая выполняется на GPU и запускается только с CPU. Для этой функции используется спецификатор `__global__`, и она должна возвращать значение типа `void`. На функции, выполняемые на GPU (`__device__` и `__global__`) накладываются ограничения: нельзя брать их адрес (за исключением `__global__`

); не поддерживается: рекурсия, статические переменные внутри функций и переменное число входных аргументов. Размещение переменных в па-

Таблица 1. Спецификаторы функций в CUDA

Спецификатор	Место вызова	Место выполнения
`__device__`	GPU	GPU
`__global__`	CPU	GPU
`__host__`	CPU	CPU

мяти GPU определяется спецификаторами `__device__` , `__constant__` и `__shared__`. Данные спецификаторы не могут применяться к полям структур и объединений. Соответствующие этим спецификаторам переменные могут использоваться только в пределах одного файла (их нельзя объявлять как extern). Запись в переменные типа `__constant__` выполняется только CPU при помощи специальных функций. Переменные класса памяти `__shared__` не могут инициализироваться при объявлении.

Кроме того, код устройства поддерживает следующие инструменты C++: полиморфизм, параметры по умолчанию, перегрузка операторов, пространства имен, шаблоны функций, классы для устройств с вычислительными возможностями 2.0.

4.2.1 Структура программы на CUDA

Программа на CUDA состоит из двух частей — host-части, выполняемой на CPU и device-части программы (ядро), выполняемой на GPU. Кроме собственных вычислений и операций ввода-вывода, host-часть программы управляет GPU, реализуя следующую последовательность операций.

- Выделение памяти на GPU.
- Копирование данных из памяти CPU в выделенную память GPU.
- Запуск ядра или последовательно нескольких ядер выполняется как `имя ядра <<<параметры запуска>>>(список аргументов)`. Параметрами запуска образуют следующий список переменных (значений): первая переменная (или значение) типа `dim3` задает размерность и размер сетки (в блоках); вторая переменная (или значение) типа `dim3` задает размерность и размер блока (в потоках); необязательная третья переменная типа `size_t` задает объем разделяемой памяти, который должен быть выделен динамически каждому блоку дополнительно к статически выделенной разделяемой памяти (если не задано,

полагается 0); четвертая переменная (значение) типа `cudaStream_t` задает поток (CUDA stream), в котором произойдет вызов ядра, по умолчанию 0.

- Копирование результатов из памяти GPU в память CPU.
- Освобождение выделенной памяти GPU.

Все операции с памятью GPU осуществляются с помощью специальных функций (`cudaMalloc` — выделение области памяти, `cudaMemcpy` — копирование данных, `cudaFree` — освобождение памяти). Направление копирования задается аргументом функции `cudaMemcpy` (`cudaMemcpyHostToDevice` или `cudaMemcpyDeviceToHost`).

В исходном коде device-части определяются операции над данными, выполняемые отдельно взятым потоком.

4.2.2 Потоки в CUDA

CUDA использует большое число параллельно выполняемых потоков, поэтому каждому потоку ставится в соответствие один элемент обрабатываемых данных (элемент матрицы или компонента вектора). Потоки организованы в следующую иерархию: сетка (grid) — блок (block) — поток (thread). Сетка соответствует всем потокам, выполняющим данное ядро (одномерный или двумерный массив блоков). Блок — одномерный, двумерный или трехмерный массив потоков. Все блоки, образующие сетку имеют одинаковую размерность и размеры. Адрес блока в сетке определяется индексом блока, адрес потока - индекс потока в блоке. Потоки разбиваются на группы по 32 потока — warp-ы. Только потоки из одного warp-а выполняются физически одновременно. Разбиение на warp-ы происходит для каждого блока отдельно, поэтому все нити одного warp-а всегда принадлежат одному блоку. Потоки могут взаимодействовать только в пределах блока.

Для определения номера потока ядро использует встроенные переменные `threadIdx` и `blockIdx`. Каждая из этих переменных — целочисленный массив из трех элементов. Размеры сетки и блока могут быть получены через встроенные переменные `gridDim` и `blockDim`.

Существует только два механизма взаимодействия потоков: разделяемая (shared) память и синхронизация. Каждому блоку выделяется определенный объем быстрой разделяемой памяти, которую могут совместно использовать все потоки данного блока. Для того, чтобы избежать конфликтов при одновременном обращении к разделяемой памяти, применяется синхронизация потоков данного блока. Барьерная синхронизация в

CUDA осуществляется при помощи функции `__syncthreads()`, которая блокирует потоки блока до тех пор, пока все потоки не вызовут эту функцию.

В CUDA реализуются следующие уровни параллелизма: параллелизм на уровне потоков (каждый поток — независимая нить исполнения); параллелизм на уровне данных (по потокам внутри блока или по блокам внутри ядра); параллелизм на уровне задач (блоки исполняются независимо друг от друга).

4.2.3 Типы памяти в CUDA

Правильное использование памяти является ключём к эффективности программ в CUDA. Наиболее быстрая часть памяти — регистровая память мультипроцессоров GPU, в которой размещаются локальные данные потока. Объем регистровой памяти — 8192 или 32768 32-битовых регистра на каждом мультипроцессоре GPU. Поток получает в монопольное пользование несколько регистров, которые доступны ему во время выполнения ядра. Доступа к регистрам других потоков нет. Если регистров не хватает — данные помещаются в локальную память.

Разделяемая (shared) память — 48 КБ на каждом мультипроцессоре. Доступна всем потокам блока на чтение и запись, по латентности близка к регистровой памяти.

Константная память — 64 КБ, выделяется в оперативной памяти графического ускорителя, кэшируется по 8 КБ на каждый мультипроцессор и имеет время доступа к данным много меньше чем к глобальной памяти, при наличии данных в кэше, доступна на чтение всем потокам сетки. Запись в нее осуществляется специальными функциями CPU.

Текстурная память — кэшируемая часть оперативной памяти графического ускорителя. Каждый мультипроцессор имеет доступ к текстурной памяти через текстурное устройство, которое выполняет различные режимы адресации и фильтрацию данных. Основным преимуществом текстуры является текстурный кэш. Текстурная память относится к блок-линейной памяти, работа с которой состоит из выделения участка памяти (функция `cudaMallocArray`), привязки его к текстурной ссылке (функция `cudaBindTextureToArray`), чтения из текстуры (функции `tex1D`, `tex2D` или `tex3D`). С текстурной ссылкой можно связать и обычную линейную память (`cudaBindTexture` или `cudaBindTexture2D`). Основным отличием текстур является возможность кэширования данных в двух измерениях.

Локальная память — область оперативной памяти графического ускорителя, к которой имеет доступ только один потоковый процессор, используется для хранения локальных данных.

Глобальная память — область оперативной памяти графического ускорителя, выделяемая при помощи функций CPU. Максимальный объём памяти, доступный всем мультипроцессорам, размер составляет от 256 МБ до 3 ГБ на видеокартах GeForce (3–6 ГБ на ускорителях Tesla). Обладает высокой пропускной способностью, более 100 ГБ/с, но большой латентностью. Поддерживаются обобщённые инструкции `load` и `store`, и обычные указатели на память. Локальная и глобальная области памяти не кэшируются и характеризуются большой латентностью (400–600 тактов).

При программировании следует учитывать, что размещение данных в разделяемой памяти позволяет уменьшить время выполнения программы в полтора раза, а использование текстурной памяти — в два раза.

4.3 Технология CUDA для нескольких GPU

Технология CUDA позволяет использовать одновременно несколько графических ускорителей, установленных в рамках одного вычислительного узла. Каждый GPU имеет свой уникальный идентификатор, с помощью которого можно запускать выполнение задачи на выбранном устройстве. До последнего времени, для одновременной работы нескольких GPU необходимо было создать на CPU определенное число параллельных потоков, чтобы каждый поток работал с собственным графическим ускорителем. В CUDA 4.0 один CPU поток имеет доступ ко всем GPU (до CUDA 4.0 было ограничение на один GPU), и появилась возможность координации работы между несколькими GPU с помощью функций `cudaStreamWaitEvent()`, `cudaDeviceSynchronize()`.

Известны также несколько расширений языков программирования, которые поддерживают параллелизм на нескольких GPU. В них предоставляется единый интерфейс для абстракций более высокого уровня начиная от общей шины до сетевых соединений. Вводятся дополнительные слои, обеспечивающие основные функции описания архитектуры и модели программирования графического устройства в то время, как основные коммуникации и механизмы планирование полностью скрываются от пользователя. Некоторые из предлагаемых подходов можно найти по ссылке `http://gpu.parallel.ru/stream.html`.

Разрабатываются также коммуникационные библиотеки glMPI, DCGN, cudaMPI (`http://www.cs.uaf.edu/sw/cudaMPI/`) и другие, которые обеспечивают MPI-подобный интерфейс для передачи данных, хранящихся на графических ускорителях вычислительных кластеров с распределенной памятью. Особенности использования ППО MPI для организации обменов между GPU будут рассмотрены в разделе 5.3.

4.4 Поддержка программирования на C++ в CUDA

Как было отмечено ранее, в CUDA часть программы, выполняемая на CPU, поддерживает полную функциональность C++, в то время как, та часть программы, которая выполняется на GPU обеспечивает ограниченную функциональность: полиморфизм; параметры по умолчанию; перегрузка операторов; пространства имен; шаблоны функций; классы для устройств с вычислительными возможностями (compute capability) 2.0. Ограничения функциональности следует учитывать при разработке нового программного обеспечения.

Для интеграции CUDA и существующего объектно-ориентированного программного обеспечения создаются новые среды разработки, такие как CuPP (`http://www.plm.eecs.uni-kassel.de/plm/index.php?id=cupp`) или библиотека шаблонов Thrust (`http://code.google.com/p/thrust/`).

Среда разработки CuPP состоит из пяти взаимосвязанных частей, некоторые из которых заменяют существующую функциональность CUDA, в то время, как другие предоставляют новые функциональные возможности.

1. Управление устройством. Осуществляется явно, связывая поток с устройством, как это делается в CUDA. Вместо этого, разработчик вынужден создать дескриптор устройства (`cupp::device`), который передается во все функции CuPP при помощи устройства, например, вызовы ядра и распределение памяти.

2. Управление памятью. Доступны две разных концепции управления памятью. Одна из них идентична предлагаемым CUDA, не считая того, что исключения «выбрасываются», когда происходит ошибка, а не возвращается код ошибки. Для облегчения разработки этого подхода предоставляется совместимый с библиотекой boost общий указатель для глобальной памяти (`http://www.boost.org/doc`). Второй тип управления памятью использует класс `cupp::memory1d`. Объекты этого класса представляют линейный блок глобальной памяти. Память распределяется, когда объект создается и освобождается когда объект удаляется. Данные могут перемещаться в память из любой структуры данных, поддерживающей итераторы.

3. C++ вызов ядра. Вызов CuPP ядра осуществляется посредством функтора C++ (`cupp::kernel`), который добавляет вызов по ссылке, близкий по семантике к вызовам ядра в CUDA. Это может использоваться, чтобы поместить в ядро структуры данных, такие как `cupp::vector`. Таким образом, устройство сможет изменять эти структуры.

4. Поддержка для классов. Используя методику названную «преобразование типа» общие классы C++ могут быть легко переданы в память устройства и из неё.

5. Структуры данных. В настоящее время предлагается только wrapper (обертка) `std::vector`, обеспечивающий автоматическое управление памятью. Этот класс также реализует функцию под названием «ленивое» (lazy) копирование памяти, чтобы минимизировать любые пересылки между памятью устройства и host-а. В настоящее время, другие структуры данных не предоставляются, но могут быть добавлены.

Начиная с версии 4.0, в CUDA включена библиотека Thrust, которая является библиотекой шаблонов C++ для CUDA и разработана на основе стандартной библиотеки шаблонов (STL). Библиотека Thrust позволяет реализовать высокопроизводительные параллельные приложения с минимальными затратами на программирование, благодаря интерфейсу высокого уровня, который полностью совместим с языком C в CUDA. Как следствие, Thrust может быть использован в быстром прототипировании приложений CUDA, когда производительность труда программиста важна также, как и надежность.

В библиотеке Thrust имеются два контейнера последовательного типа `host_vector` и `device_vector`, которые хранятся соответственно в памяти CPU и в памяти графического ускорителя. Оператор присваивания может быть использован для копирования `host_vector` в `device_vector` или наоборот при инициализации. Например, если в программе объявлены вектор `thrust::host_vector<int> h_vec(1<<24,1)`, тогда операция `thrust::device_vector<int> d_vec=h_vec` приведет к заполнению объекта `d_vec`, хранящегося на GPU единицами. Однако, поскольку подобное копирование требует вызова функции `cudaMemcpy`, они должны использоваться разумно.

Кроме базовых алгоритмов STL, таких как: поиск, сортировка, операции над множествами и других, в библиотеке Thrust реализованы алгоритмы префиксной редукции (scan), редукции (reduce) — некоторые аналоги функций `MPI_Scan` и `MPI_Reduce` соответственно.

На основе этих контейнеров, алгоритмов могут быть построены более сложные алгоритмы с кратким, читабельным исходным кодом.

4.5 Пример SpMV в технологии CUDA

В качестве примера использования технологии CUDA, рассмотрим программный код матрично-векторного произведения, который представлен в виде: host-функции `SpMV`, написанной на C++, функции «ядра» `kernelSpMV` и device-функций для работы с текстурной памятью `fetch_AV` и `fetch_v`.

Аргументами функции SpMV (см. **Листинг 13**) являются: maxDimGrid — максимальное количество блоков на GPU; `c` — вектор результата; `b` — вектор, на который умножается матрица; `d_ANL` — массив номеров элементов из `AV` с которых начинается новая строка матрицы (хранится в глобальной памяти GPU, device-копия массива `ANL`); `N` — размерность матрицы; `Nnz` — количество ненулевых элементов матрицы. Макроопределением `DSz` определяется размер типа `double` в байтах (`#define DSz sizeof(double)`).

Листинг 13. Функция SpMV, реализующая на GPU **Алгоритм 2**

```
1  void SpMV (const int maxDimGrid, double* c,
2                     const double *b, const int *d_ANL,
3                     const int N, const int Nnz)
4  {
5   const int dimBlock = 128;  //число нитей в блоке
6  //среднее число ненулевых элементов в строке
7   int nnz_per_row = Nnz / N;
8  //число нитей для вычисления одной координаты вектора
9   int thr_per_vec;
10
11         if      (nnz_per_row <= 2)  thr_per_vec=2;
12         else if (nnz_per_row <= 4)  thr_per_vec=4;
13         else if (nnz_per_row <= 8)  thr_per_vec=8;
14         else if (nnz_per_row <= 16) thr_per_vec=16;
15         else                        thr_per_vec=32;
16
17 // число векторов в блоке
18  const int vecs_per_bl = dimBlock / thr_per_vec;
19 //вычисление числа блоков DimGrid
20  size_t tmp = (dim+vecs_per_bl-1)/vecs_per_bl;
21  const size_t DimGrid = std::min<int>(maxDimGrid,tmp);
22 //"привязка текстуры"
23     cudaBindTexture( 0, tex_b, b, DSz*N );
24 //вызов <<ядра>> SpMV
25  kernelSpMV <<<DimGrid,dimBlock>>> (c, d_ANL, N,
26                          vecs_per_bl, thr_per_vec);
```

```
27 //"отвязка текстуры"
28      cudaUnbindTexture(tex_b);
29 }
```

Соответственно, аргументами «ядра» kernelSpMV (**Листинг 14**) являются: d_res — вектор результата, d_ANL — массив номеров элементов, с которых начинается новая строка матрицы (device-копия массива ANL); N — размер матрицы; vecs_per_bl — количество векторов в блоке; thrs_per_vec — количество нитей задействованных для вычислений одного вектора (может принимать значения 2,4,8,16,32).

Листинг 14. Функция «ядро» для умножения матрицы на вектор

```
1  __global__ void kernelSpMV ( double *d_res,
2          const int *d_ANL,    const int N,
3          const int vecs_per_bl, const int thrs_per_vec )
4  {
5    __shared__ double sdata[144];
6    __shared__ int    ptrs[64][2];
7        const int    tidx = threadIdx.x;
8              int    row, jj;
9
10 //количество нитей в блоке
11     const int thrs_per_bl = vecs_per_bl * thrs_per_vec;
12 //глобальный номер нити
13     const int i = blockIdx.x * thrs_per_bl + tidx;
14 //номер нити внутри вектора
15     const int thr_lane = tidx & (thrs_per_vec - 1);
16 //глобальный номер вектора
17     const int vec_id = i / thrs_per_vec;
18 //номер вектора внутри блока
19     const int vec_lane = tidx / thrs_per_vec;
20 //общее количество векторов
21     const int num_vecs = vecs_per_bl * gridDim.x;
22
23 for(row=vec_id; row < N; row += num_vecs)
24 {
25  if(thr_lane<2)
26       ptrs[vec_lane][thr_lane]=d_ANL[row+thr_lane];
27
28  const int row_beg=ptrs[vec_lane][0]; //row_beg=ANL[row]
29  const int row_end=ptrs[vec_lane][1]; //row_end=ANL[row+1]
```

```
30
31    double sum = 0;
32
33   if(thrs_per_vec == 32 && row_end - row_beg > 32)
34   {
35 //выравнивание доступа к памяти
36          jj=row_beg-(row_beg & (thrs_per_vec-1))+thr_lane;
37 //накапливание локальной суммы
38      if((jj >= row_beg) && (jj <row_end))
39       sum +=fetch_AV(jj)*fetch_v(tex1Dfetch(tex_ANC,jj).x);
40 //накапливание локальной суммы
41      for(jj+=thrs_per_vec; jj<row_end; jj+=thrs_per_vec)
42       sum +=fetch_AV(jj)*fetch_v(tex1Dfetch(tex_ANC,jj).x);
43    }
44    else
45    {
46 //накапливание локальной суммы
47      for(jj=row_beg+thr_lane; jj<row_end; jj+=thrs_per_vec)
48       sum +=fetch_AV(jj)*fetch_v(tex1Dfetch(tex_ANC,jj).x);
49    }
50 //запись локальной суммы в разделяемую память
51       sdata[tidx] = sum;
52
53 // сбор локальных сумм по строкам
54   if(THRS_PER_VEC >16)sdata[tidx]=sum=sum+sdata[tidx+16];
55   if(THRS_PER_VEC > 8)sdata[tidx]=sum=sum+sdata[tidx+ 8];
56   if(THRS_PER_VEC > 4)sdata[tidx]=sum=sum+sdata[tidx+ 4];
57   if(THRS_PER_VEC > 2)sdata[tidx]=sum=sum+sdata[tidx+ 2];
58   if(THRS_PER_VEC > 1)sdata[tidx]=sum=sum+sdata[tidx+ 1];
59
60 // запись результата в нить с номером ноль
61         if (thr_lane == 0) d_res[row] = sdata[tidx];
62   }
63 }
```

В рассматриваемом примере используются ссылки на текстуры: `tex_AV`, `tex_ANC`, `tex_x`, их объявления представлены в **Листинге 15**. Вместе с тем, привязка текстуры к данным в функции `SpMV` осуществлена только для `tex_x`. Привязки ссылок `tex_AV` и `tex_ANC` выполняются в функции, которая вызывает `SpMV`, во избежания многократного обращения к функциям привязки при вызовах `SpMV`.

Листинг 15. Объявление ссылок на текстуры

```
//ссылка на текстуры значений матрицы
texture <int2, 1, cudaReadModeElementType> tex_AV;
//ссылка на текстуру номеров столбцов
texture <uint1, 1, cudaReadModeElementType> tex_ANC;
//ссылка на текстуру значений вектора
texture <int2, 1, cudaReadModeElementType> tex_b;
```

При вычислении локальных сумм (переменная `sum` в **Листинге 14**) используется обращение к текстурной памяти. Доступ к элементам матрицы и вектора, хранящимся в текстурной памяти происходит при помощи функций `fetch_AV` и `fetch_v` (**Листинг 16**).

Листинг 16. Функции доступа к данным, хранящимся в текстурной памяти

```
// функция возвращает значение i-го элемента массива AV,
//   привязаного к текстуре tex_AV
__inline__ __device__ double fetch_AV(const int& i)
{
//взятие компоненты из текстуры
 int2 v = tex1Dfetch(tex_AV, i);
//преобразование числа в double
 return __hiloint2double(v.y,v.x);
}

// функция возвращает значение i-го элемента массива x,
//   привязаного к текстуре tex_x
__inline__ __device__ double fetch_v(const int& i)
{
//взятие компоненты из текстуры
 int2 v = tex1Dfetch(tex_x, i);
//преобразование числа в double
 return __hiloint2double(v.y, v.x);
}
```

При определении времени выполнения в CUDA используется механизм событий (events). Завершение всех операций данного события обеспечивается функцией `cudaEventSynchronize`. Время выполнения рассматривается, как время, прошедшее между двумя событиями и возвращается функцией `cudaEventElapsedTime`.

Отметим, что аргумент, соответствующий времени выполнения, является указателем на тип `float` и возвращает значение в миллисекундах. В рассмотренном далее примере (**Листинг 17**) создаются два события `start` и `stop`, а время передается в переменную `t`.

Листинг 17. Определение времени выполнения функции SpMV

```
cudaEvent_t start,stop;
float t=0.0;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
cudaEventSynchronize(start);
    SpMV(maxDimGrid, Ax, x, d_ANL, dim, Nnz);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&t,start,stop);
```

Представленные технологии ориентированы на определенные аппаратные и программные архитектуры (распределенная и общая память, SIMD). Эта специализация технологий параллельного программирования является одновременно преимуществом и недостатком. Далее будут рассмотрены гибридные технологии, универсальность которых строится на основе достоинств каждой технологии.

Создание приложений для гетерогенных параллельных вычислительных платформ является сложным процессом, поскольку классические подходы к программированию для многоядерных CPU и GPU значительно различаются. Модели программирования CPU, хотя в большинстве случаев и основаны на стандартах, но обычно они предполагают наличие общего адресного пространства и не учитывают возможность векторных операций.

Модели программирования GPU общего назначения, включая сложные иерархии памяти и векторные операции, традиционно остаются платформозависимыми. Эти ограничения затрудняют доступ разработчиков к имеющимся программным кодам для CPU, GPU и других типов процессоров. В настоящее время необходимо предоставить возможность разработчикам ПО эффективно использовать все преимущества гетерогенных вычислительных платформ — высокопроизводительных вычислительных кластеров, которые включают разнообразные параллельные CPU, GPU и другие процессоры.

5 Использование гибридных технологий на гетерогенных кластерах

Рассмотренные ранее модели параллельного программирования изначально ориентированны на определенную аппаратную архитектуру.

Модель обмена сообщениями, в первую очередь предназначена для вычислительных модулей, связанных сетевым интерфейсом, модель общей памяти — для распараллеливания внутри вычислительных модулей, в рамках единого адресного пространства, модель параллелизм данных — для специализированных устройств с большим числом упрощенных вычислительных элементов. В то же время, современные вычислительные системы, состоят из многопроцессорных модулей с мультиядерными процессорами, кроме того модули могут включать графические ускорители, что предполагает использование более общих подходов к программированию. Одним из таких подходов является программная «универсализация» вычислительных устройств, примером которой является OpenCL.

Другой подход, связанный с объединением прикладных программ, предназначенных для гетерогенных многопроцессорных вычислительных систем, требует применения одновременно нескольких парадигм, реализованных в виде коммуникационного промежуточного программного обеспечения. В данной главе будут рассмотрены методы совместного использования промежуточного программного обеспечения MPI, широко применяемого в параллельных научных расчетах, и ППО OpenCL, CUDA, OpenMP, а также CORBA, предназначенного для разработки распределенных объектно-ориентированных приложений. Это дает возможность сборки интегрированных комплексов программ для междисциплинарных расчетов на гетерогенных многопроцессорных системах с повторным использованием накопленного прикладного программного обеспечения.

5.1 Технология OpenCL

OpenCL (Open Computing Language) `http://www.khronos.org/opencl/` — открытый стандарт для универсального параллельного программирования различных типов процессоров, таких как CPU, GPU и другие. Стандарт предоставляет программистам переносимый и эффективный доступ к гетерогенным вычислительным платформам. OpenCL поддерживает широкий круг приложений: от встроенного и клиентского программного обеспечения до высокопроизводительных решений, благодаря низкоуровневой, высокопроизводительной, переносимой абстракции.

Стандарт OpenCL описывает API для управления процессом парал-

лельных вычислений среди гетерогенных процессоров; так же кросс-платформенный язык программирования с детально описанным вычислительным окружением. Стандарт OpenCL (версия 1.1):

- поддерживает модели параллельного программирования, основанные на параллелизме данных и модели, основанные на параллелизме задач;
- использует подмножество ISO C99 с расширениями для поддержки параллелизма;
- поддерживает стандарт арифметики чисел с плавающей точкой IEEE 754 (`http://softelectro.ru/ieee754.html`);
- определяет профили конфигураций для переносных и встраиваемых устройств.

5.1.1 Архитектура OpenCL

OpenCL [18–20] — это окружение для параллельного программирования, которое включает язык программирования, API, библиотеки и систему запуска, поддерживающую разработку программного обеспечения. Созданные на OpenCL программы используют ресурсы вычислительной системы следующим образом:

- определяются доступные устройства в гетерогенной системы и выбираются подходящие;
- создается последовательность инструкций (kernel), которые будут выполняться на выбранных устройствах;
- подготавливаются входные данные для вычислений, с использованием объектов памяти OpenCL (memory objects);
- выполняется kernel, обрабатывающий подготовленные данные на выбранном ранее устройстве;
- собираются результаты вычислений.

Для описания основных идей OpenCL вводится следующая иерархия моделей [19]:

- *модель платформы* дает высокоуровневое описание гетерогенной системы;

- *модель исполнения* описывает абстрактное представление того, как потоки инструкций выполняются в гетерогенной системе;
- *модель памяти* описывает набор областей памяти и манипулирование ими во время проведения вычислений;
- *модель программирования* описывает варианты переноса абстрактного алгоритма на гетерогенные вычислительные ресурсы.

5.1.2 Модель платформы

Платформы OpenCL (см. рисунок 6) состоит из главного устройства (`host`) и соединенных с ним одного или более OpenCL-устройств (далее device). Центральным элементом модели платформы является понятие host — главное устройство, которое управляет OpenCL-вычислениями и осуществляет все взаимодействия с пользователем. Платформа OpenCL всегда содержит только один host.

OpenCL-устройством может быть CPU, GPU, DSP (сигнальный микропроцессор) или любой другой процессор, архитектура которого поддерживается разработчиками OpenCL. Каждое OpenCL-устройство состоит из одного или более *вычислительных устройств* (compute units), которые разделяются на один или более *процессорных элементов* (processing elements). Вычисления на OpenCL-устройстве выполняются процессорными элементами.

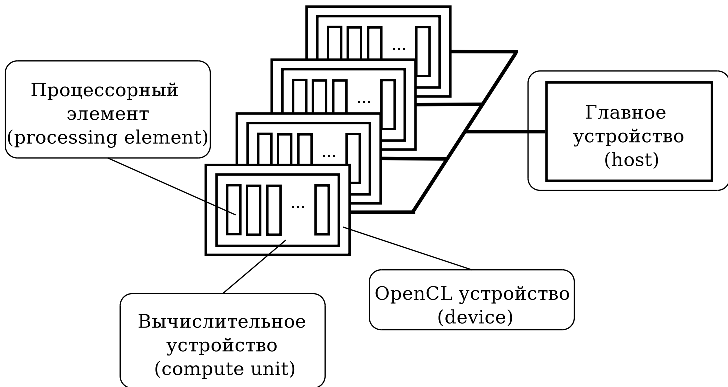

Рис. 6. Модель платформы OpenCL

5.1.3 Модель исполнения

Модель исполнения описывает выполнение потоков инструкций в гетерогенной системе. OpenCL приложение состоит из двух отдельных частей: главной программы — host program и набора из одного или нескольких ядер (kernel). Главная программа выполняется на главном устройстве. OpenCL не определяет, как главная программа работает, а только, как она взаимодействует с объектами, определенными в OpenCL.

Ядра выполняются на OpenCL-устройствах. Они выполняют основную вычислительную работу OpenCL-приложения. Можно сказать, что ядра осуществляют отображение входных объектов памяти в выходные объекты памяти. OpenCL определяет два типа ядер:

- *ядра OpenCL* (OpenCL kernels) — функции, написанные на языке программирования OpenCL C (подмножество языка ISO C 99) и скомпилированные компилятором OpenCL. Все реализации OpenCL должны поддерживать ядра OpenCL;

- *собственные ядра* (native kernels) — функции, созданные вне OpenCL и доступные в OpenCL при помощи указателей на функцию. Выполнение native kernel относится к необязательным функциональным возможностям OpenCL и определяется реализацией.

Важным моментом для понимания того, как работает OpenCL является выполнение ядра на OpenCL-устройстве. Ядро создается в главной программе и затем с помощью специальной команды ставится в очередь на выполнение в одном из OpenCL-устройств. Во время выполнения указанной команды, OpenCL Runtime System создает целочисленное индексное пространство, каждый элемент этого пространства называется глобальный идентификатор (global ID). Экземпляр ядра выполняется для каждой точки в полученном индексном пространстве. Каждый экземпляр выполняемого ядра называется work-item (далее поток) и идентифицируется по его координатам в пространстве индексов. Эти координаты являются глобальными идентификаторами (ID) для потока.

Каждый поток выполняет один и тот же код, но конкретный путь исполнения (ветвления и т.п.) и данные, с которыми он работает, могут быть различными. Потоки организуются в группы (work-groups). Группы предоставляют более грубое разбиение в пространстве индексов. Каждой группе приписывается групповой ID с такой же размерностью, которая использовалась для адресации отдельных элементов. Каждому элементу сопоставляется уникальный, в рамках группы, локальный ID. Таким образом, потоки OpenCL могут быть адресованы как по глобальному идентификатору,

так и по комбинации идентификатора группы и локального ID. Потоки в группе исполняются параллельно на *процессорных элементах* одного *вычислительного устройства*. Это гарантируется стандартом, в то время как совершенно не гарантируется, что несколько потоков из разных групп будут выполнены параллельно. Об этом важном свойстве необходимо всегда помнить при разработке OpenCL-программ.

Таблица 2. Соответствие понятий OpenCL и CUDA

OpenCL	Эквивалент в CUDA
kernel	kernel
host	host
NDRange	Grid
Work Group	Block
Work Item	Thread

В таблице 2 приведены эквиваленты понятий OpenCL и CUDA, используемых при описании структуры программ и иерархии потоков.

Пространство индексов в OpenCL называется NDRange и может быть n-мерным, где $n = 1, 2, 3$. Внутри OpenCL-программы, NDRange определяется, как целочисленный массив длины n указанием размера индексного пространства в каждой измерение. Глобальные и локальные ID каждого потока являются n-мерными кортежами. В простейшем случае, компонентами глобального ID являются значения в диапазоне от нуля до числа элементов в этом измерении минус один.

Выбор размерности NDRange определяется удобством для конкретного алгоритма: в случае работы с трехмерными моделями удобно индексировать по трехмерным координатам, в случае работы с изображениями или двумерными сетками удобнее, когда размерность индексов два.

Контекст и очереди команд в модели исполнения

Другим важным понятием модели вычислений является контекст, определение которого (при помощи вызова специальных функций OpenCL API) является первой задачей OpenCL-приложения.

Главная программа определяет контекст для исполнения ядер. Контекст включает в себя следующие ресурсы:

- devices — набор устройств OpenCL (далее устройства), которые использует главное устройство;

- kernels (ядра) — функции OpenCL, которые запускаются на устройствах;
- program Objects (программные объекты) — исходные коды и исполняемые файлы ядер;
- memory Objects (объекты памяти) — набор объектов памяти, видимых устройствам. Объекты памяти содержат значения, которые могут обрабатываться экземплярами ядра.

Взаимодействие между главным устройством и OpenCL-устройством происходит посредством команд, помещенных в очередь команд. Команды ожидают в очереди своего выполнения на устройстве. Очередь команд создается главной программой и сопоставляется одному OpenCL-устройству после того, как будет определен контекст. Команды отвечают за выполнение ядер, управление памятью и синхронизацию выполнения команд в очереди. OpenCL поддерживает три типа команд:

- команды выполнения ядра выполняют ядро на *процессорных элементах* OpenCL-устройства;
- команды памяти перемещают данные между host-ом и различными объектами памяти, или из них и между ними;
- команды синхронизации устанавливают ограничения на порядок выполнения команд.

Команды могут выполняться последовательно или не по порядку. В случае последовательного выполнения команды запускаются на исполнение в том порядке, в котором они расположены в очереди и завершаются так же по порядку. При исполнении не по порядку команды отправляются на исполнение по порядку, но не ждут завершения предыдущей команды перед началом исполнения. В этом случае программист должен явно использовать команды синхронизации. Второй вариант организации очередей поддерживается не всеми платформами, о чем необходимо помнить.

Команды выполнения ядра и команды памяти, помещенные в очередь создают объекты события, используются, чтобы контролировать выполнение команд и чтобы управлять взаимодействием между главным и другими устройствами.

В OpenCL существуют две области синхронизации: потоки в одной группе; команды, помещенные в очередь (очереди) команд в одном контексте.

Синхронизация между потоками в одной группе осуществляется, используя барьер группы. Все потоки некоторой группы должны выполнить барьер, прежде чем любой из них продолжит выполнение за барьером. Отметим, что барьер группы должен быть установлен всем потокам группы, выполняющей данное ядро, или ни одному из них. Не существует механизма для синхронизации между группами потоков.

Точками синхронизации между командами в очереди команд являются.

- Барьер очереди команд. Барьер очереди команд гарантирует, что все ранее помещенные в очередь команды завершат выполнение и в результате любые обновления объектов памяти станут видны командам, поставленным в очередь позднее, прежде чем они начнут выполнение. Данный барьер может использоваться только чтобы синхронизировать команды в одной очереди команд.

- Ожидание события. Все OpenCL API функции, ставящие команды в очередь, возвращают событие, которое идентифицирует команду и обновляет объекты памяти. Последующим командам, ожидающим этого события, гарантируется, что обновления данных объектов памяти произойдут до начала выполнения команд.

С одним контекстом можно связать несколько очередей команд. Эти очереди исполняются, конкурируя между собой и независимо, без каких-либо явных способов синхронизации между ними. Использование очереди команд, позволяет добиться большой универсальности и гибкости при использовании OpenCL. Современные GPU имеют собственный планировщик, который решает, что, когда и на каких вычислительных блоках исполнять. Использование очереди не затрудняет работу планировщика, который имеет собственную очередь команд.

5.1.4 Модель памяти

В OpenCL определяется два типа объектов памяти: объекты буферы и объекты изображения. Объект буфер, как следует из названия, это просто непрерывный блок памяти, доступный для ядра. Программист может отображать структуры данных на этот буфер и имеет доступ к буферу с помощью указателей. Этим обеспечивается гибкость, позволяющая определить практически любую структуру данных по желанию разработчика (с учетом ограничений по языку программирования для ядер OpenCL).

Тип объект памяти изображение предназначен для решения задач обработки изображений. Формат хранения изображений может быть оптими-

Таблица 3. Способы выделения и возможности доступа к памяти

Область памяти	Часть программы			
	host		kernel	
	выделение	доступ	выделение	доступ
global	динамическое	чтение/запись	нет	чтение/запись
constant	динамическое	чтение/запись	статическое	чтение
local	динамическое	нет	статическое	чтение/запись
private	нет	нет	статическое	чтение/запись

зирован под конкретное OpenCL устройство. Объект памяти изображение является скрытым объектом OpenCL, для работы с которым применяются соответствующие функции, а содержание объекта изображение скрыто от ядра программы.

Каждый поток имеет доступ к четырем различным областям памяти.

- Глобальная память предоставляет доступ на чтение и запись элементам всех групп. Каждый поток может писать и читать из любой части объекта памяти. Запись и чтение глобальной памяти может кэшироваться в зависимости от возможностей устройства.

- Константная память — область глобальной памяти, которая остается постоянной во время исполнения ядра. Главная программа выделяет и инициализирует объекты памяти, расположенные в константной памяти.

- Локальная память — область памяти, локальная для группы. Эта область памяти может использоваться, чтобы создавать переменные, разделяемые всей группой. Она может быть реализована как отдельная память на OpenCL-устройстве. Этот вид памяти может быть также размечена как область в глобальной памяти.

- Частная память — область памяти, принадлежащая потоку. Переменные, определенные в частной памяти одного потока, не видны другим.

Способы выделения и возможности доступа к различным областям памяти приведены в таблице 3.

5.1.5 Модель программирования

Модель исполнения OpenCL поддерживает две программные модели: параллелизм данных (Data Parallel) и параллелизм заданий (Task Parallel),

Таблица 4. Соответствие типов памяти OpenCL и CUDA

Тип памяти в OpenCL	Эквивалент в CUDA
global	global
constant	constant
local	shared
private	local

так же поддерживаются гибридные модели. Основной программной моделью OpenCL является параллелизм данных.

Программная модель с параллелизмом данных. Данная модель определяет вычисления, как последовательность инструкций, применяемых к множеству элементов объекта памяти. Пространство индексов, ассоциированное с моделью исполнения OpenCL, определяет потоки и то, каким образом данные распределяются между ними. В модели параллелизма данных OpenCL не требуется строгое соответствие между потоком и элементом в объекте памяти, с которым работает ядро.

OpenCL предоставляет иерархическую модель параллелизма данных. Существует два способа определить иерархическое деление. В явной модели программист определяет общее число элементов, которые должны исполняться параллельно, и так же каким образом эти элементы будут распределены по группам. В неявной модели программист только определяет общее число элементов, которые должны исполняться параллельно, а разделение по рабочим группам выполняется автоматически.

Программная модель с параллелизмом заданий. В этой модели каждый экземпляр ядра исполняется независимо от пространства индексов. Логически, это эквивалентно исполнению ядра на вычислительном устройстве группой, состоящей из одного элемента. В такой модели пользователи выражают параллелизм следующими способами:

- используются векторные типы данных, реализованные в устройстве;
- в очередь устанавливается множество заданий;
- устанавливаются в очередь собственные ядра, использующие программную модель, ортогональную к OpenCL.

Поддержка двух моделей программирования в технологии OpenCL придаёт большую универсальность.

Для современных GPU и процессоров Cell хорошо подходит первая модель. Но не все алгоритмы можно эффективно реализовать в рамках такой

модели, а так же есть вероятность появления устройства, архитектура которого будет неудобна для использования первой модели. В таком случае вторая модель позволяет описать специфичные для другой архитектуры приложения.

5.1.6 Среда программирования OpenCL

Среда программирования OpenCL позволяет приложениям использовать host и одно или несколько OpenCL-устройств, как одну гетерогенную параллельную компьютерную систему. Среда программирования состоит из следующих компонентов:

- API платформы OpenCL позволяет головной программе обнаруживать OpenCL-устройства, опрашивать их свойства и создавать контекст;
- среда исполнения OpenCL позволяет головной программе управлять контекстом после его создания;
- язык программирования OpenCL служит для написания программного кода для ядер, основан на расширенном подмножестве стандарта ISO C99.

Программирование главной программы OpenCL осуществляется на C или C++ и не предполагает ограничений на язык программирования, характерных для ядра. В главной программе OpenCL приложения, при помощи функций, указанных в скобках, выполняются следующие шаги.

1. Выбирается OpenCL устройство (`clGetPlatformIDs`, `clGetDeviceIDs`).
2. Для исполнения программы на устройстве создается контекст, которым инициализируется выбранное устройство (`clCreateContext`);
3. на основе исходных кодов или бинарных файлов и контекста создается программа (`clCreateProgramWithSource`).
4. Собирается программа (`clBuildProgram`).
5. Создается ядро (`clCreateKernel`).
6. Определяется очередь команд на основе ID устройства и контекста (`clCreateCommandQueue`).
7. Для входных и выходных данных создаются объекты памяти (`clCreateBuffer` и `clSetKernelArg`).

8. Записываются данные из области памяти на хосте в память устройства (clSetKernelArg).

9. В очередь команд помещаются команды:

   9.1 исполнения созданного ядра (clEnqueueNDRangeKernel);

   9.2 считывания результатов из устройства (clEnqueueReadBuffer).

10. Освобождаются ресурсы (clReleaseMemObject, clReleaseKernel, clReleaseCommandQueue,clReleaseProgram, clReleaseContext).

В результате создания эффективного, учитывающего аппаратные особенности платформы программного интерфейса, OpenCL способен сформировать базовый уровень параллельного вычислительного кода независимых от аппаратной платформы программных инструментов, промежуточного ПО и других видов приложений.

5.1.7 Пример использования технологии

В качестве примера использования технологии OpenCL рассмотрим пример произведения неразрежённой матрицы **A** на вектор **b**. Программа состоит из двух частей, выполняемых соответственно на CPU (см. **Листинг 18**) и GPU (см. **Листинг 19**). В комментариях к программному коду (см. **Листинг 18**) отмечены шаги создания пользовательского приложения, приведенные выше. Исходный код ядра, т.е. код, выполняемый на GPU (**Листинг 19**), считывается из файла mvcl.cl, находящегося в том же каталоге, что и файл mvcl.c, содержащий главную функцию (**Листинг 18**).

Листинг 18. Пример главной функции OpenCL приложения

```
1  // Файл mvcl.c
2  #define PROGRAM_FILE "mvcl.cl"
3  #define KERNEL_FUNC "mv_kernel"
4  #define DSz sizeof(double)
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/types.h>
8  #include <CL/cl.h>
9
10 int main() {
11 cl_platform_id platform; // платформа OpenCL
12 cl_device_id device; // устройство OpenCL
```

```
13 cl_context context; // контекст OpenCL
14 cl_command_queue queue; // очередь команд
15 cl_kernel kernel; //
16 cl_mem A_buff, b_buff, c_buff;//буферы под данные
17 cl_program program; // программа OpenCL
18 cl_int i, err; // индекс и индикатор ошибки
19 FILE *program_handle;
20 char *program_buffer, *program_log;
21 size_t program_size, log_size;
22 size_t wus_k=4;// число WU на ядро
23 double A[16],b[4],c[4]={0.0, 0.0, 0.0, 0.0},cgpu[4];
24 /* Инициализация A и b, вычисление c=A*b на CPU */
25 for(i=0;i<16;i++) A[i]=i*2.0;// Инициализация матрицы
26 for(i=0; i<4; i++)
27 { b[i] = i * 3.0; // Инициализация вектора b
28 // c = A*b, вычисляемое на CPU
29 c[0] += A[i]   * b[i]; c[1] += A[i+4]  * b[i];
30 c[2] += A[i+8] * b[i]; c[3] += A[i+12] * b[i];
31 }
32 /* Шаги 1-2. Определение платформы и устройства.*/
33 clGetPlatformIDs(1, &platform, NULL);
34 clGetDeviceIDs(platform,CL_DEVICE_TYPE_GPU,
35                              1,&device,NULL);
36 context = clCreateContext(NULL,
37                              1,&device,NULL,NULL,&err);
38 /* Считывание исходного кода ядра из PROGRAM_FILE */
39 program_handle = fopen(PROGRAM_FILE, "r");
40 fseek(program_handle, 0, SEEK_END);
41 program_size = ftell(program_handle);
42 rewind(program_handle);
43 program_buffer = (char*)malloc(program_size + 1);
44 program_buffer[program_size] = '\0';
45 fread(program_buffer, sizeof(char), program_size,
46 program_handle);
47 fclose(program_handle);
48 /* Шаги 3-6. Создание программы. */
49 program = clCreateProgramWithSource(context, 1,
50 (const char**)&program_buffer, &program_size, &err);
51 free(program_buffer);
52 clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

```
53 kernel = clCreateKernel(program, KERNEL_FUNC, &err);
54 queue = clCreateCommandQueue(context, device, 0, &err);
55 /* Шаги 7 и 8. Выделение памяти и передача
56                           аргументов в ядро. */
57 A_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
58 CL_MEM_COPY_HOST_PTR, 16*DSz, A, &err);
59 b_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
60 CL_MEM_COPY_HOST_PTR, 4*DSz, b, &err);
61 c_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
62 4*DSz, NULL, &err);
63 clSetKernelArg(kernel, 0, sizeof(cl_mem), &A_buff);
64 clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_buff);
65 clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_buff);
66 /* Шаг 9.1. Вычисление c = A * b на GPU */
67 clEnqueueNDRangeKernel(queue,kernel,1,NULL,&wus_k,
68                               NULL,0,NULL,NULL);
69    /* Шаг 9.2. Возвращение вектора c на CPU. */
70 clEnqueueReadBuffer(queue,c_buff,CL_TRUE,0,4*DSz,
71                               cgpu,0,NULL,NULL);
72 if((cgpu[0]==c[0])&&(cgpu[1]==c[1])&&(cgpu[2]==c[2])
73      &&(cgpu[3]==c[3])) printf("A*b was successful.\n");
74 else                   printf("A*b was unsuccessful.\n");
75  /* Шаг 10. Освобождение занятых ресурсов */
76  clReleaseMemObject(A_buff); clReleaseMemObject(b_buff);
77  clReleaseMemObject(c_buff);
78  clReleaseKernel(kernel); clReleaseCommandQueue(queue);
79  clReleaseProgram(program);  clReleaseContext(context);
80 return 0;
81 }
```

Отметим, что при вычислении произведения $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ на CPU (см. **Листинг 18**), применяется так называемое разворачивание цикла.

Для того, чтобы использовать в коде ядра переменные типа `double`, в **Листинге 19** применена прагма `OPENCL EXTENSION cl_khr_fp64 : enable`, которая подключает расширение `cl_khr_fp64`. Возможность подключения этого расширения для используемого устройства определяется при помощи функции `clGetDeviceInfo`.

Благодаря тому, что матрица **A** хранится строками фиксированной длины ($N = 4$), в ядре `mv_kernel` для вычисления произведения строки i матрицы **A** на вектор **b** используется встроенная функция `dot` из кате-

гории геометрических функций OpenCL, а также применяется векторный тип данных OpenCL — double4. Очевидно, что при хранении матрицы в формате CSR потребуется изменить ядро.

Листинг 19. OpenCL ядро mv_kernel

```
1 // Файл mvcl.cl
2 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
3
4 __kernel void mv_kernel(__global double4* matrix,
5         __global double4* vector,__global double* result)
6 {
7     int i = get_global_id(0);
8 result[i] = dot(matrix[i], vector[0]);
9 }
```

Из представленного в **Листингах 18** и **19** программного кода собирается приложение mvcl, при помощи следующей командной строки

```
gcc -O2 mvcl.c -o mvcl -lOpenCL
```

В рассматриваемом случае полагается, что заголовочные файлы OpenCL и библиотека (для NVIDIA — это libOpenCL.so) находятся в стандартных каталогах (/usr/include/CL и /usr/lib).

Разработка оптимальных с точки зрения производительности OpenCL функций для GPU и CPU требует знания аппаратных особенностей платформы исполнения. Поэтому программы, созданные при помощи OpenCL, будут переносимыми в большом диапазоне аппаратных архитектур, но производительность при таком переносе не сохранится. Как видно из примера, требуется большое число строк программного кода, что является платой за возможность выполнения кода практически на любом процессоре, предназначенном для параллельных вычислений. Тем не менее, программы на OpenCL показывают достойную производительность, по сравнению с конкурентами, и могут успешно использоваться для параллельных вычислений общего назначения.

Большее распространение в настоящее время получили гибридные подходы такие как «обмен сообщениями – общая память», «обмен сообщениями – параллелизм данных», позволяющие сочетать достоинства каждой модели, далее остановимся на них подробнее. Вместе с тем, очевиден недостаток гибридных подходов — пользователю необходимо использовать две разные парадигмы параллельного программирования и разное ППО. А вот преимущества гибридного подхода по сравнению с MPI и целесообразность

его использования на многоядерных кластерах — это вопрос, по которому единого мнения в настоящее время нет.

5.2 Гибридная модель MPI/OpenMP

Успешное внедрение OpenMP на мультипроцессорах, появление SMP-кластеров (узлами которых являются мультиядерные процессоры) привело к тому, что все шире начал использоваться гибридный подход «обмен сообщениями – общая память», когда программа представляет собой систему взаимодействующих MPI-процессов, а внутри MPI-процесса (он считается мастер-потоком) могут создаваться новые потоки OpenMP. Такой подход упрощает программирование, когда в программе есть два уровня параллелизма — параллелизм между подзадачами и параллелизм внутри подзадач. Такая ситуация возникает, например, при использовании методов декомпозиции области при решении вычислительных задач.

Программировать на MPI подзадачи для подобластей гораздо сложнее, чем их взаимодействие, поскольку распараллеливание подзадачи связано с распределением элементов массивов и витков циклов между процессами. Организация же взаимодействия подобластей таких сложностей не вызывает, поскольку сводится к обмену между ними граничными значениями.

5.2.1 Возможности повышения производительности в гибридной модели MPI/OpenMP

Основные возможности повышения производительности в гибридной модели MPI/OpenMP относительно MPI следующие.

- *Перекрытие MPI внутри вычислительного узла.* Лучше использовать OpenMP на узлах избегая перекрытия связанные с вызовом MPI-библиотек. В этом случае гибридная модель OpenMP/MPI будет показывать лучшую производительность по сравнению с чистым MPI. Однако есть и осложняющие факторы.
  - Использование OpenMP может ввести дополнительные перекрытия, которых не было в MPI-версии (т.е. синхронизация, последовательные секции, неправильное разделение).
  - Может потребоваться больше синхронизаций, чем при чистом MPI.
  - В гибридном случае должна использоваться MPI реализация не поддерживающая потоки.

  - Неявная синхронизация точка-точка должна быть заменена более дорогостоящими барьерами.
  - Жесткость в перекрытии внутренних потоковых коммуникаций с сообщениями между узлами.

- *Соревнование за сеть.* На одном вычислительном узле с n_p процессорами будет наблюдаться ситуация когда все n_p MPI-процессов (при чистом MPI) будут посылать сообщения другим узлам в произвольный момент времени. Это может привести к борьбе за сетевые ресурсы. В этом случае лучше послать одно сообщение большей длины.

- *Плохое использование MPI.* При использовании MPI в приложениях не для кластеров можно достичь значительных преимуществ в гибридной модели. Хорошее применение коллективных коммуникаций должно минимизировать внутриузловые сообщения, т.е. сокращение в узлах, затем через узлы. Гибридная модель обеспечивает это естественно, т.е. OpenMP сокращает внутри узла, а MPI через узлы.

- *Плохая масштабируемость MPI.* В случае плохой масштабируемости MPI-версии приложения, гибридная модель может показывать значительно лучшие результаты. OpenMP масштабируется лучше MPI в случае, если есть алгоритмические особенности, т.е. случае, где балансировка нагрузки порой может вызывать трудности.

- *Дублирование данных.* Некоторые MPI-приложения используют дублирование данных, и все процессоры имеют копию основной структуры данных. Для чистого MPI-кода требуется одна копия данных на процессор. В гибридной модели будет необходима одна копия на узел. Структура данных может быть общей для нескольких потоков на процессоре, поэтому можно получить существенное повышение производительности на данных большого размера за счет уменьшения обмена данными.

- *Ограничение на число MPI-процессоров.* Некоторые MPI-приложения не могут быть запущены на произвольном числе процессоров, т.е. требуется число процессоров по степени два. Некоторые SMP узлы имеют число процессоров не по степени два. В гибридной модели эти ограничения могут быть просто снято, и не потребуется переписывание MPI-кода.

- *Ограничение числа MPI-процессоров.* Применение MPI не может быть адекватным при числе процессоров больше ста тысяч и возможно

только на специально выпускаемых больших ВС. Гибридная модель будет уменьшать число MPI-процессоров.

- *Балансировка вычислительной нагрузки.* В гибридной модели: работа разделяется равномерно между MPI-процессорами; когда процесс перегружен работой, уменьшается число OpenMP-потоков; требуется гибкий менеджер ресурсов; лучше используется для ВС с общей памятью, чем для кластеров.

5.2.2 Подходы к построению гибридной модели MPI/OpenMP

Очевидным способом создания гибридных MPI/OpenMP программ является расширение программного кода MPI-программ средствами OpenMP.

Можно выделить два основных подхода. В первом подходе все коммуникации осуществляются из MPI-процесса (последовательной части программного кода OpenMP). Другой подход построения гибридной модели более общий, в нем реализуются любые виды MPI коммуникаций внутри параллельной области OpenMP. Отметим, что в обоих подходах могут применяться следующие способы распараллеливания.

- Параллельное выполнение ветвей циклов внутри MPI-процесса при помощи директивы `#pragma omp for`.

- Одновременное выполнение нескольких функций или областей кода внутри MPI-процесса при помощи директивы `#pragma omp sections` или явного подхода с использованием номера потока.

Как отмечено выше, особенностью первого подхода является участие в в коммуникациях между вычислительными узлами только главного потока OpenMP, он же MPI-процесс. Для этого, может выполняться разделение данных на внутренние и интерфейсные, после чего они обрабатываются соответственно MPI-процессом или OpenMP потоками [22].

В рамках второго подхода реализуются дополнительные возможности распараллеливания.

- Параллельные обмены данными. Невзаимосвязанные пересылки данных выполняются параллельно в разных потоках (по разным коммуникаторам). Прием и передача осуществляются из двух разных потоков OpenMP.

- Совмещение передачи данных и вычислений. Один или несколько потоков OpenMP из параллельной области осуществляют вызовы коммуникационных функций MPI, в то время, как оставшиеся выполняют необходимые вычисления.

Обращаться к функциям MPI из параллельной области OpenMP позволяет стандарт MPI-2, в котором среда потоков инициализируется при помощи функции `MPI_Init_thread`. Третий аргумент данной функции — желательный уровень поддержки потоков, последний аргумент — предоставляемый. Уровень `MPI_THREAD_FUNNELED` соответствует вызову функций MPI только из одного потока, а уровень `MPI_THREAD_MULTIPLE` обеспечивает вызов функций MPI из всех потоков без ограничений.

5.2.3 Особенности привязки процессов/потоков в гибридной модели MPI/OpenMP

Привязка позволяет определить какое ядро будет выполнять потоки. Установка маски привязки применимо как в случае чистого MPI-приложения, так и для гибридного MPI/OpenMP, но особенно значимо влияние привязки в гибридной модели параллельного программирования.

При использовании гибридной MPI/OpenMP-модели, OpenMP-потоки создаются как часть MPI-процесса. Если привязка потоков не установлена явно, они все наследуют привязку MPI-процесса. Это подразумевает, что по умолчанию, все OpenMP-потоки будут выполняться на тех же логических процессорах (ядрах), что и MPI-процесс. При привязке каждого MPI-процесса к отдельному ядру потоки OpenMP не смогут запустится на других ядрах.

Например, в случае программы, запущенной с двумя MPI-процессами на 2-х ядрах, и порождением в каждом процессе еще четырёх OpenMP-потоков, загружены будут только два ядра — по четыре потока на каждом ядре.

При запуске гибридного MPI/OpenMP-приложения число запускаемых OpenMP-потоков задается переменной окружения `OMP_NUM_THREADS`. Поэтому для обеспечения миграции OpenMP-потоков на свободные ядра вычислительного узла необходимо использовать привязки MPI-процессов к сокетам или вычислительному узлу. Применительно к MPICH2 и SLURM — это флаг `--cpu_bind=sockets` или сочетание флагов `--cpus-per-task` и `--cpu_bind`. Во втором случае, значение `ncpus` полагается равным числу OpenMP-потоков, флаг `--cpu_bind` принимает значения `cores` или `threads`. Для Open MPI — это параметры привязки `bynode` и `byslot`.

При программировании кластеров, состоящих из многопроцессорных узлов, часто используется гибридный подход MPI/OpenMP, когда взаимодействие между узлами программируется с использованием MPI, а реализация задач на отдельных узлах осуществляется с помощью OpenMP. Это неизбежно приводит к усложнению не только проектирования параллельной программы, но и к усложнению разработки приложения в целом.

5.2.4 Пример SpMV для гибридной модели MPI/OpenMP

Рассмотрим применение MPI/OpenMP подхода, опираясь на предложенную ранее (см. **Листинг 11**) OpenMP-версию метода `SpM::SpMV`. Полагается, что N_p — число MPI процессов, N_{th} — число потоков OpenMP и число процессоров (ядер) — n_p соотносятся как $n_p = N_p \cdot N_{th}$.

При реализации данного примера необходимо внести ряд изменений в класс `SpM` (**Листинг 1**). Объект для хранения матрицы $\mathbf{A}$ создается конструктором по умолчанию, с последующим изменением размеров контейнеров при помощи метода `SpM::resize`. Тело метода `SpM::resize` совпадает с телом конструктора `SpM::SpM(int, int)`. Для того, чтобы скрыть работу с размером контейнера `c`, в метод `SpM::SpMV` перед пятой строкой **Листинг 12** добавляется строка `c.resize(ie);`.

Программный код (**Листинг 20**) выполняет инициализацию матрицы $\mathbf{A}$ и вектора $\mathbf{b}$, декомпозицию матрицы, рассылку ее блоков $\mathbf{A}_i$ и вектора $\mathbf{b}$ по процессам MPI, вычисление произведения $\mathbf{c}_i = \mathbf{A}_i\mathbf{b}$ в каждом процессе и объединения вектора результата $\mathbf{c} = \bigcup_{i=0}^{N_p-1} \mathbf{c}_i$.

В каждом процессе MPI создаются объекты для хранения матрицы $\mathbf{A}$ и ее блоков, и векторов $\mathbf{b}$ и $\mathbf{c}$. Операции разделения матрицы, рассылки $\mathbf{A}_i$ и вектора $\mathbf{b}$, объединения вектора и проверки результата выносятся в отдельные функции, упрощая сопровождение программного кода и повышая его читабельность.

Листинг 20. Пример MPI/OpenMP реализации SpMV

```
1 // Файл mpiomp.cpp
2 #include <vector>
3 #include <iostream>
4 #include <mpi.h>
5 #include "spm.h"
6
7 using namespace std;
8
9 void filling(SpM& , int, int);
```

```
10 void filling(vector<double>&, int, double);
11 void decomp(int,int,SpM &,vector <int>&,vector <int>&,
12             vector <int>&,vector <int>&,int &,
13                         int &,int &,int &, MPI_Comm &);
14 void distrib(int, SpM&, SpM&, int, int,vector <double>&,
15             int, vector <int>&,vector <int>&,
16             vector <int>&, vector <int>&, MPI_Comm &);
17 void gather(int,int,int,vector <double>&, vector <double>&,
18               vector <int>&,vector <int>&, MPI_Comm &);
19 void ctrl(int, int, vector <double>&, vector <double>&);
20
21 int main(int argc, char *argv[])
22 {
23 MPI_Comm  world = MPI_COMM_WORLD;
24 int Np, myid, Ni, NNZi, N, NNZ;
25 vector <int> displs,rcnts,rcnts1,displs1;
26 SpM A, A_i;
27 vector <double> b, c, c_i;
28
29 MPI_Init(&argc,&argv);
30 MPI_Comm_size (world, &Np);
31 MPI_Comm_rank (world, &myid);
32
33  if(!myid)
34  {
35    N=8, NNZ=64;
36     filling(A,N,NNZ); // Заполнение A
37     filling(b,N,1); // Заполнение b
38  }
39 // Декомпозиция матрицы A
40  decomp(myid,Np,A,displs,rcnts,displs1,rcnts1,
41                              Ni,NNZi,N,NNZ,world);
42 // Рассылка блоков Ai и вектора b по процессам
43  distrib(myid,A,A_i,Ni,NNZi,b,N,displs,rcnts,
44                         displs1,rcnts1,world);
45 // Произведение c_i = A_i * b
46   A_i.SpMV(b,c_i,0,Ni);
47
48 // Объединение вектора c
49   gather(myid,Np,N,c_i,c,displs,rcnts,world);
```

```
50
51 // Проверка результата
52    ctrl(myid,NNZ,b,c);
53
54 MPI_Finalize();
55 return 0;
56 }
```

Произведение блока $\mathbf{A}_i$, состоящего примерно из N/N_p строк матрицы, на вектор $\mathbf{b}$ вычисляется в каждом MPI-процессе. Диапазон строк матрицы $\mathbf{A}$, входящих в $\mathbf{A}_i$ и обрабатываемых процессом `myid` задается переменными `ib` и `ie`. Результат произведения в каждом MPI-процессе помещается в вектор $\mathbf{c}_i$. Объединение векторов $\mathbf{c}_i$ выполняется на нулевом процессе, в этом же процессе происходит проверка результата.

Потоки OpenMP используются внутри метода `SpM::SpMV` для параллельного вычисления $\mathbf{c}_i = \mathbf{A}_i \cdot \mathbf{b}$. Число потоков N_{th} задается в самом методе при помощи функции `omp_get_max_threads` (см. **Листинг 12**).

Контейнеры векторного типа `rcnts`, `displs`, `rcnts1`, `displs1`, вводятся для рассылки $\mathbf{A}_i$ по процессам MPI и объединения вектора . Объект для хранения матрицы $\mathbf{A}$ создается конструктором по умолчанию, с последующим изменением размеров контейнеров при помощи функции `SpM::resize`.

Для задания значений матрицы и вектора применяется переопределение функции `filling` (см. **Листинг 21**).

Листинг 21. Функция для задания значений матрицы и вектора

```
1 void filling(SpM& A, int N, int NNZ)
2 {
3 A.resize(N,NNZ);
4 vector <double>& AV =A.getAV();
5 vector <int>&    ANC=A.getANC();
6 vector <int>&    ANL=A.getANL();
7 for(int i=0; i<NNZ; i++) AV[i]=i;
8    for(int i=0, k=0; i<N; i++)
9    {
10      for(int j=0; j<N; j++, k++) ANC[k]=j;
11           ANL[i+1] = k;
12   }
13 }
14
15 void filling(vector<double>& b, int N, double bval)
```

```
16 {
17     b.resize(N);
18   for(int  i=0;  i<N;  i++)  b[i]=bval;
19 }
```

В функции `decomp` при помощи вспомогательных контейнеров векторного типа `rcnts`, `displs`, `rcnts1`, `displs1` формируется структура данных для рассылки блоков матрицы, после чего в процессы MPI рассылаются размеры этих блоков N_i и NNZ_i. Разделение **A** на блоки происходит при рассылке данных.

Листинг 22. Декомпозиция матрицы **A** на блоки $\mathbf{A}_i$

```
1  void  decomp(int  myid,int  Np,  SpM& A,vector <int>& displs,
2                vector <int>& rcnts,  vector <int>& displs1,
3                vector <int>& rcnts1,  int &Ni,  int &NNZi,
4
5                int &N,  int &NNZ,  MPI_Comm& world)
6  {
7  vector <int>   sbuf(4*Np),  rbuf(4);
8
9   rcnts.resize(Np);   displs.resize(Np+1);
10 rcnts1.resize(Np);  displs1.resize(Np+1);
11
12 if(!myid)
13 {
14 vector <int>& ANL=A.getANL();
15 int N   = A.getN();
16 int NNZ= A.getNNZ();
17   for  (int  i=0;  i<Np;  i++)
18   {
19      displs[i]  =  (i*N)/Np;
20       rcnts[i]  =  (i+1-Np)  ?  (N/Np)  :  (N-((N*(Np-1))/Np));
21   }
22
23   displs[Np]=N;  displs1[0]=0;
24
25   for  (int  i=0,  k=0,  ii=0;  i<Np;  i++)
26   {
27      ii=ANL[displs[i+1]]-ANL[displs[i]];
28      sbuf[4*i]=rcnts[i];  sbuf[4*i+1]=ii;
29      rcnts1[i]=ii;  displs1[i]=k;
```

```
30      rcnts[i]++;   k+=ii;
31      sbuf[4*i+2]=N;   sbuf[4*i+3]=NNZ;
32   }
33
34 }
35   MPI_Scatter(&sbuf[4*myid],4,MPI_INT,&rbuf[0],4,
36                                   MPI_INT,0,world);
37   Ni=rbuf[0]; NNZi=rbuf[1]; N=rbuf[2]; NNZ=rbuf[3];
38 }
```

Рассылка матрицы **A**, хранящейся в CSR формате, осуществляется в функции `distrib` при помощи коллективной функции `MPI_Scatterv`, что позволяет применять данную функцию в случаях, когда N и Nnz не кратны n_p. В нулевом процессе в функцию `MPI_Scatterv` подаются **A**, начальные позиции и размеры блоков, а во всех процессах принимаются уже соответствующие блоки $\mathbf{A}_i$. Структура CSR формата представляется тремя векторными контейнерами `AV`,`ANC`,`ANL`, которые рассылаются отдельно, тремя вызовами `MPI_Scatterv`. Во всех MPI-процессах, за исключением нулевого значения из `ANLi`, переводятся в диапазон $0, 1, \ldots, Nnz_{i+1}$ (**Листинг 23**, строки 23–24). Вектор **b** рассылается функцией `MPI_Bcast`.

Листинг 23. Рассылка блоков $\mathbf{A}_i$ и вектора **b**

```
1  void distrib(int myid, SpM& A, SpM& Ai,int Ni, int NNZi,
2           vector <double>& b1, int N, vector <int>& displs,
3           vector <int>& rcnts, vector <int>& displs1,
4           vector <int>& rcnts1, MPI_Comm& world)
5  {
6  vector <double>& AV =A.getAV();
7  vector <int>&    ANC=A.getANC();
8  vector <int>&    ANL=A.getANL();
9
10 Ai.resize(Ni,NNZi);
11 vector <double>& AVi =Ai.getAV();
12 vector <int>&    ANCi=Ai.getANC();
13 vector <int>&    ANLi=Ai.getANL();
14
15 MPI_Scatterv(&AV[0], &rcnts1[0], &displs1[0], MPI_DOUBLE,
16                      &AVi[0],    NNZi, MPI_DOUBLE, 0, world);
17 MPI_Scatterv(&ANC[0],&rcnts1[0], &displs1[0], MPI_INT,
18                      &ANCi[0],   NNZi, MPI_INT, 0, world);
19 MPI_Scatterv(&ANL[0],&rcnts[0],&displs[0], MPI_INT,
```

```
20                          &ANLi[0],   Ni+1, MPI_INT, 0, world);
21 if(myid)
22 {
23     int j=ANLi[0];
24     for (int i=0; i<Ni+1; i++)ANLi[i]-=j;
25     b1.resize(N);
26 }
27 MPI_Bcast(&b1[0], N, MPI_DOUBLE, 0, world);
28 }
```

Объединение вектора **c** осуществляется функцией `MPI_Gatherv`, результат помещается в MPI-процесс с номером ноль. Для этого начальные позиции векторов $\mathbf{c}_i$ в векторе **c** и их размерности помещаются в контейнеры `displs` и `rcnts` (см. **Листинг 24**).

Листинг 24. Объединение вектора **c**

```
1 void gather(int myid, int Np, int N, vector <double>& c_i,
2             vector <double>& c, vector <int>& displs,
3             vector <int>& rcnts, MPI_Comm& world)
4 {
5 c.resize(N);
6 int Ni = c_i.size();
7
8 if(!myid) // Процесс с номером ноль
9 for (int i=0; i<Np; i++)
10 {
11  displs[i] = (i*N)/Np;
12 rcnts[i] = (i+1-Np) ? N/Np : (N-((N*(Np-1))/Np));
13 }
14  MPI_Gatherv( &c_i[0], Ni, MPI_DOUBLE, &c[0], &rcnts[0],
15                       &displs[0], MPI_DOUBLE, 0, world);
16 }
```

Проверка результата вынесена в функцию `ctrl` (см. **Листинг 25**).

Листинг 25. Функция проверки результата

```
1 void ctrl(int myid, int NNZ, vector <double>& b,
2                             vector <double>& c)
3 {
4   if(!myid) // Процесс с номером ноль
5   {
```

```
6       int N = b.size();
7       double bc=0;
8       for(int i=0; i<N; i++) bc += b[i]*c[i];
9   cout<<"(b,c)="<<bc<<"\tsum(a_ij)="<<NNZ*(NNZ-1)/2<<endl;
10    }
11 }
```

Приведенный пример показывает, как сделать расчетные данные доступными всем процессам MPI и для этого потребуется коллективный обмен сообщениями. Вместе с тем, декомпозиция матрицы позволяет избавиться от хранения избыточной информации, например, объект матрица `SpM A_i(Ni,NNZi)` требует примерно в N_p раз меньший объем памяти, чем объект `SpM A(N,NNZ)`.

Гибридный MPI/OpenMP подход позволяет выполнять программный код (**Листинг 20**) на многопроцессорных вычислительных системах, кластерах и многоядерных серверах или персональных компьютерах.

5.3 Гибридные модели MPI/CUDA и OpenMP/CUDA

Большинство графических ускорителей имеет объём памяти до 3-х Гбайт, что подразумевает использование систем с несколькими ускорителями на CUDA (multiGPU) или даже кластера таких систем. В этом случае, целесообразным является применение комбинации технологий промежуточного программного обеспечения [22]:

- MPI для программирования обмена данными между узлами кластерной ВС;
- OpenMP для создания многопоточного окружения на каждом узле кластера ВС, необходимого для MultiGPU систем;
- CUDA для непосредственного вычисления.

В подобном гетерогенном кластере, главной задачей будет выделение унифицированных вычислительных ядер CUDA, независимых от данных, а за пересылки и распределение данных должны отвечать технологии MPI и OpenMP.

Если рассматривается возможность увеличения производительности с помощью использования технологии MPI+CUDA, то необходимыми являются следующие условия:

- обладает ли алгоритм двумя уровнями параллелизма. Если алгоритм имеет возможность распараллеливания на втором уровне, то

это необходимо учитывать при проектировании параллельного алгоритма;

- на втором уровне распараллеливания должна содержатся интенсивная вычислительная часть, которая может быть обработана на GPU. Если это небольшой объём вычислений, то не следует ожидать значительного повышения производительности по сравнению с применением ППО CUDA.

Главной задачей при построении параллельных алгоритмов для MPI/CUDA становится эффективное разделение расчётных данных задачи между блоками и потоками. Таким образом, как и при использовании MPI, необходимо использовать геометрический параллелизм, когда каждому блоку назначается свой сектор исходных данных, а каждый поток выполняет вычисления только с одним элементом данных. Для каждого ускорителя требуется создать свой поток, инициализировать CUDA-контекст, а потом еще заниматься синхронизацией.

В результате подход, основанный на MPI/CUDA, ставший популярным решением, оказывается, масштабируется как на систему с несколькими ускорителями, так и на целый кластер. Единственным недостатком такого решения — сложно эффективно использовать центральный процессор.

В ППО CUDA 4.0 реализована возможность работать из одного CPU-потока с разными устройствами. К одному GPU теперь можно обращаться из разных потоков. Также стоит упомянуть про заявленную официальную «интеграцию с MPI». В ближайшее время будет выпущена специальная модификация OpenMPI trunk, MVAPICH2-GPU адаптированная для работы с CUDA. В частности, с помощью операций `Send/Receive` можно будет копировать данные напрямую в видеопамять.

Узким местом модели MPI/CUDA является совместное использование компиляторов mpicc/mpiCC и nvcc. Часть программного кода, написанного на CUDA, необходимо скомпилировать и собрать в библиотеку при помощи nvcc. Далее необходимо подключить эту библиотеку при сборке исполняемого файла с использованием mpiCC. В случае с OpenMP, такой проблемы не возникает потому, что библиотека OpenMP входит в пакет компилятора, например gcc/g++, и вызывается как его опция, а mpicc/mpiCC и nvcc являются надстройками над тем же gcc/g++ и не являются взаимосвязанными программными продуктами.

Для кластеров с общей памятью и архитектурой (CPU+GPU), а также современных настольных компьютеров предпочтительнее применять модель «общая память — параллелизм данных». В этом случае, каждому

GPU ставится в соответствие поток OpenMP. Рассмотренные далее примеры основаны на программном коде, приведенном в (**Листинге 13**).

5.3.1 Примеры программного кода SpMV

Логика программ MPI/CUDA (**Листинг 26**) и OpenMP/CUDA (**Листинг 28**) практически не отличается. Процессу MPI или потоку OpenMP при помощи функции, возвращающей номер текущего процесса или потока назначается GPU. В **Листинге 26** переменная `myid` соответствует номеру процесса MPI, а в **Листинге 28** номеру процесса OpenMP.

Листинг 26. Фрагмент кода, выполняемый каждым MPI-процессом

```
1  #define DSz sizeof(double)
2  #define ISz sizeof(int)
3  int N_GPU; // число доступных GPU
4  cudaGetDeviceCount(&N_GPU);
5  cudaDeviceProp deviceProp;
6
7  MPI_Comm_rank(world, &myid);
8  cudaSetDevice(myid);
9  cudaGetDevicePropeties (&deviceProp, myid);
10 /* Разделение матрицы на блоки
11                 и копирование данных на GPU */
12     ...
13 /* Привязка текстур к массивам */
14
15   cudaBindTexture ( 0, tex_AV , d_AV, DSz*N );
16   cudaBindTexture ( 0, tex_ANC, d_ANC,DSz*N );
17 /* Умножение блока матрицы A на вектор b  */
18 SpMV(deviceProp.MaxGridSize[0], d_c, d_b, Ni,
19            ANL[Num_line[myid+1]-ANL[Num_line[myid]]);
20 /* Копирование результата на CPU */
21 cudaMemcpy(c+Num_line[myid],d_c,Ni,
22                   cudaMemcpyDeviceToHost);
23 /* Отвязка текстур  */
24   cudaUnbindTexture(tex_AV);
25   cudaUnbindTexture(tex_ANC);
```

В host-части (MPI или OpenMP) программы матрица **A** разделяется на блоки, состоящие из N_i строк **A** (см. **Листинг 27**), полученные блоки $\mathbf{A}_i$ копируются в память GPU. Число строк в блоке хранится в переменной `Ni`.

Далее блоки $\mathbf{A}_i$ умножаются на копии вектора $\mathbf{b}$, каждый в своём GPU, при помощи функции `SpMV` из раздела 4.5 (см. **Листинг 13**). Вектор $\mathbf{c}$ собирается при копировании элементов массива `d_c` из памяти GPU в оперативную память. Привязка текстур к массивам `d_AV`, `d_ANC` происходит внутри кода, выполняемого каждым процессом MPI (**Листинг 26**) или потоком OpenMP (**Листинг 28**).

Разделение матрицы на блоки и копирование данных на GPU вынесены в отдельный фрагменте кода (см. **Листинг 27**). При копировании в память GPU элементов массива `ANL` происходит отображение данных в массив меньшей размерности `d_ANL`, при помощи массива смещений номеров строк `Num_line`.

Листинг 27. Разделение матрицы на блоки и копирование данных на GPU

```
int *Num_line = new int [N_GPU+1];
int Ni=(myid—N_GPU+1) ? N/N_GPU : (N—(N*(N_GPU—1)/N_GPU));
/*              Смещение номеров строк            */
for (i=myid; i<N_GPU; i++) Num_line[i+1]+=Ni;

/* Выделение памяти GPU */
  double *d_AV, *d_res, *d_b;
  int    *d_ANC,*d_ANL;

  cudaMalloc((void **)&d_AV,   DSz*NNZ);
  cudaMalloc((void **)&d_ANC,  ISz*NNZ);
  cudaMalloc((void **)&d_ANL,  DSz*(Ni+1));
  cudaMalloc((void **)&d_c,    DSz*Ni);
  cudaMalloc((void **)&d_b,    DSz*N);
/* Копирование данных на GPU */
  cudaMemcpy(d_b,   b,   DSz*Nnz, cudaMemcpyHostToDevice);
  cudaMemcpy(d_AV,  AV,  DSz*Nnz, cudaMemcpyHostToDevice);
  cudaMemcpy(d_ANC,ANC,DSz*Nnz, cudaMemcpyHostToDevice);
  cudaMemcpy(d_ANL,ANL+Num_line[myid], DSz*(Ni+1),
                                  cudaMemcpyHostToDevice);
```

Приведенный ниже фрагмент кода (**Листинг 28**) может выполняться не только в составе OpenMP/CUDA приложения, но и вызываться в MPI процессе как часть MPI/OpenMP/CUDA приложения.

В случае MPI/OpenMP/CUDA приложения необходимо удалить строки 3–5 из **Листинга 26** и заменить строки 8–24 на код из **Листинга 28**.

В этом случае, разделение матрицы на блоки будет двухуровневым. На первом уровне матрица будет разделена на блоки $\mathbf{A}_i$, которые рас-

пределятся между MPI-процессами, на втором уровне OpenMP выполнит распределение строк, из которых состоят блоки $\mathbf{A}_i$ и векторов между GPU.

Листинг 28. Фрагмент кода, выполняемый потоком OpenMP в OpenMP/CUDA приложении

```
#pragma omp parallel num_threads(N_GPU)
{
    cudaDeviceProp deviceProp;
    int myid = omp_get_thread_num();
    cudaSetDevice(myid);
    cudaGetDevicePropeties (&deviceProp, myid);
/* Разделение матрицы на блоки
                и копирование данных на GPU */
    ...
/* Привязка текстур к массивам */
  cudaBindTexture ( 0, tex_AV , d_AV,  DSz*N ) ;
  cudaBindTexture ( 0, tex_ANC, d_ANC, DSz*N ) ;
/* Умножение блока матрицы A на вектор b  */
    SpMV(deviceProp.MaxGridSize[0], d_c, d_b, Ni,
                ANL[Num_line[myid+1]-ANL[Num_line[myid]]);
/* Копирование результата на CPU */
  cudaMemcpy(c+Num_line[myid], d_c, Ni,
                              cudaMemcpyDeviceToHost);
/* Отвязка текстур  */
  cudaUnbindTexture(tex_AV);
  cudaUnbindTexture(tex_ANC);
}
```

Полученное параллельное приложение можно выполнять на системах, состоящих из множества multiGPU вычислительных узлов, используя достоинства каждой технологии.

Использованием MPI/CUDA достигается ускорение параллельных алгоритмов, которые имеют определенные характеристики параллелизма. В таких случаях повышение производительности возможно более значительное, чем на кластерах с использованием только MPI. Можно отметить, что подход MPI/CUDA позволяет создавать высокопроизводительные вычислительные кластеры при низких затратах.

5.4 Гибридная модель MPI/CORBA

При разработке интегрированных комплексов программ для междисциплинарных расчетов приходится объединять коды различных прикладных программ, которые зачастую разработаны независимо, имеют свои требования к вычислительным ресурсам и используют разные коммуникационные парадигмы. В этом случае, проблема их совместного использования решается не только на уровне компиляторов языков программирования. Необходимо обеспечить взаимодействие используемых систем промежуточного программного обеспечения. Многие вычислительные программы используют параллельное коммуникационное программное обеспечение MPI. Известно, что объединение прикладных программ, основанных на промежуточном программном обеспечении MPI, становится проблематичным, если в качестве коммуникаций между ними также используется MPI. Это связано с тем, что в рамках комплекса нескольких программ, как правило, используется распределенная парадигма коммуникаций. Поэтому при построении интегрированной программной системы целесообразнее применять CORBA.

5.4.1 Технология CORBA

ППО CORBA (`http://www.corba.org/`) создана для поддержки разработки и развёртывания сложных объектно-ориентированных прикладных систем.

CORBA является механизмом в программном обеспечении для осуществления интеграции изолированных систем, который даёт возможность программам, написанным на разных языках программирования, работающих в разных узлах вычислительной системы, взаимодействовать друг с другом так же просто, как если бы они находились в адресном пространстве одного процесса.

Технология CORBA (Common Object Request Broker Architecture — общая архитектура брокера объектный запросов) [6] — стандарт и промежуточное программное обеспечение для разработки распределенных объектно-ориентированных программ на различных языках программирования, как правило, объектно-ориентированных (C++, Java и др.). Независимость от языков программирования достигается путем предварительного описания объектов на специальном языке IDL (Interface Definition Language — язык описания интерфейса) и компиляции этих описаний в код на языке программирования, на котором впоследствии объекты должны быть реализованы. При разработке программ, кроме традиционных объектов языка программирования, возникают объекты специального вида:

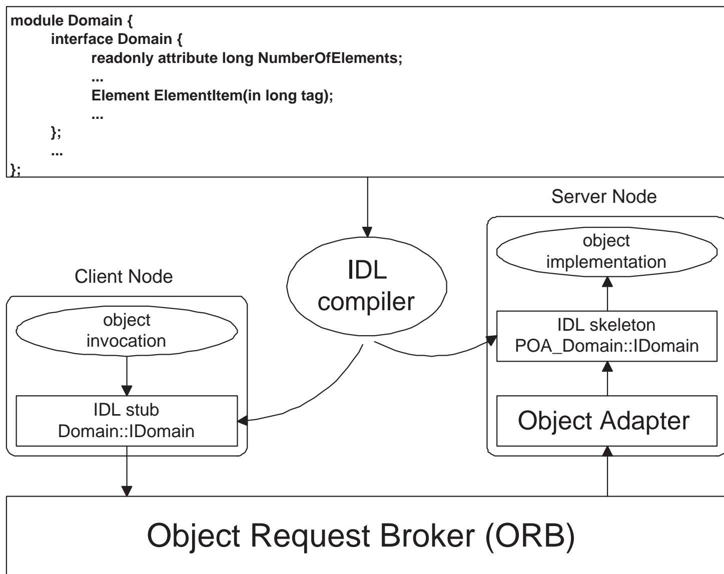

Рис. 7. CORBA программа: разработка и выполнение

- сервант — удаленный объект,
- стаб (локальный стаб) и скелетон (удаленный стаб) — объекты на стороне клиентского и серверного процессов соответственно, обеспечивающие удаленное обращение к серванту.

Удаленный вызов метода серванта осуществляется через стаб и скелетон, аналогично RPC (рисунок 7). Ядро CORBA имеет стандартизованный объектно-ориентированный интерфейс и может быть расширено для обеспечения новых возможностей данного промежуточного программного обеспечения.

Параллельное выполнение процессов в CORBA чаще всего реализуется на основе односторонних вызовов `oneway` (вызовов без уведомления) [21]. Для этого определяются парные CORBA объекты: основной объект и обработчик события. Методы основного объекта, вызываемые асинхронно, содержат параметр-ссылку на обработчик, не возвращают результат и имеют флаг `oneway`; обработчик содержит парные методы, входными параметрами которых являются выходные параметры соответствующих методов

основного объекта. Набор активированных основных объектов после выполнения того или иного параллельного метода возвращает результат, вызывая парный метод обработчика и передавая ему в качестве входных параметров результаты своей работы. Объект-обработчик предназначен для использования в клиентской части программы, в которой одновременно (асинхронно) вызывается один и тот же метод у набора основных объектов. Далее, периодически прерываясь на обработку результата, полученного от какого-либо процесса, клиентский код может выполняться до тех пор, пока не понадобятся результаты параллельного метода. В итоге все объекты-процессы функционируют параллельно, асинхронно клиентскому приложению.

AMI (`http://www.cs.wustl.edu/~schmidt/TAO.html`) — Asynchronous Method Invocation - асинхронный вызов методов описывает два способа асинхронного вызова методов: обратный вызов (callback) и опрос (polling). В модели обратного вызова клиентское приложение асинхронно вызывает тот или иной метод CORBA объекта и передает ему в качестве дополнительного входного параметра ссылку на обработчик события, являющийся CORBA сервантом, который уже активирован на клиентской стороне. По окончании выполнения операции автоматически вызывается парный метод обработчика, которому в качестве входных параметров передаются результаты операции. По сути, это автоматизированный аналог изложенного выше подхода. В данном случае нет необходимости вводить вспомогательные CORBA объекты. IDL компилятор дополнительно к коду основного CORBA объекта генерирует код обработчика его методов. ORB обеспечивает взаимосвязь серверного объекта и обработчика событий, а также оптимизирует обмены.

Результатом асинхронного вызова метода в механизме опроса (polling) является вспомогательный объект – запрошенный объект (не является сервантом CORBA). Когда основному процессу требуется результат, он обращается к запрошенному объекту, который опрашивает параллельный объект на предмет завершения выполнения метода. В таком случае нет необходимости дробить весь вычислительный процесс на код до асинхронного вызова, код обработки событий, связанных с завершением выполнения одной из нитей, и код заключительной обработки.

В обоих случаях IDL-компилятор генерирует коды основного и вспомогательного объекта и готовые к использованию клиентские стабы с асинхронными вызовами. Асинхронные вызовы обозначаются `sendc_/sendp_` для callback/polling методов соответственно. В этом случае пользователю нет необходимости разрабатывать вспомогательные интерфейсы и объекты, кроме этого, AMI обеспечивает методы обработки ошибок. Первая мо-

дель разрушает структуру объектов, т.к. часть кода переносится в методы обработки сообщения о завершении асинхронного вызова. Вторая модель менее производительна — требует частого обращения к удаленному объекту, — тогда как в первой модели обращение по сети происходит только один раз по факту выполнения метода.

Еще одним методом повышения производительности промежуточного программного обеспечения CORBA является совместное использование с MPI. Это метод позволяет одновременно вовлечь в процесс вычислений компьютеры различных архитектур, обеспечивает стандартные интерфейсы программирования, ориентирован на объединение прикладных программ с использованием как параллельной, так и распределенной парадигмы коммуникаций. В IDL-описании компонента отражается его возможное поведение либо путем использования специальных интерфейсов, либо путем расширения синтаксиса IDL. Это позволяет значительно упростить разработку параллельных программ еще и потому, что описания интерфейсов CORBA могут быть использованы специально вводимыми компонентами (сервисами), отрабатывающими различные методы распараллеливания. Реализовать SIMD-технологии можно путем использования промежуточного ПО MPI в составе CORBA.

5.4.2 Метод интеграции CORBA и MPI

Рассмотрим задачу совместного использования CORBA и MPI на примере промежуточного программного обеспечения TAO и MPICH [23]. Обычно, проблема совместного использования программ решается на уровне компиляторов языков программирования, путем одновременного подключения их модулей, здесь же, кроме этого, необходимо решить вопросы взаимодействия двух систем промежуточного программного обеспечения и декомпозиции MPI кода. Определим основные шаги, которые необходимо пройти при интеграции систем.

1. Инициализация промежуточного программного обеспечения. Возможные решения: инициализация MPI в рамках CORBA и инициализация CORBA в рамках MPI. Первый вариант нецелесообразен, поскольку сам процесс инициализации MPI не стандартизован. Второй вариант заключается в использовании системы запуска MPI, при этом входные параметры приложения пересылаются по MPI и доступны далее при инициализации CORBA. Кроме этого, во время инициализации CORBA доступна информация о месте процесса в MPI коммуникаторе.

2. Создание объектов CORBA. На этом шаге определятся способ работы с объектами CORBA: все процессы подключаются к одному или

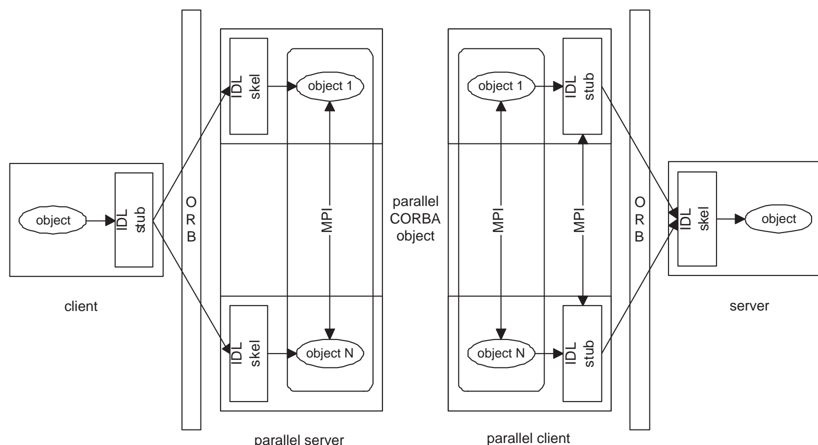

Рис. 8. Интеграция MPI и CORBA

нескольким удаленным CORBA объектам (набор клиентов) или во всех процессах создается CORBA объект (набор серверов) — рисунок 8. В первом случае, в процессах описывается смешанный код: взаимодействие с другими процессами MPI и обращение к удаленным объектам CORBA. Во втором случае, CORBA код отделяется от кода MPI: в процессе создаются CORBA-объекты и реализовывается процесс обработки запросов к ним, MPI-код помещается внутри тех или иных методов CORBA объектов. Этот способ позволяет явно описать параллельную модель в виде объектов и соответственно обеспечивает более гибкий механизм разработки интегрированного кода.

3. Организация обмена сообщениями через MPI. Следующим важным моментом при интеграции CORBA и MPI является организация обмена сообщениями через MPI. В плане обмена сообщениями через MPI, клиентский код ничем не отличается от обычной MPI-программы. В случае же серверного кода, необходим одновременный вызов методов, содержащих идентичные части MPI кода. Для этого можно использовать технологию CORBA AMI (Asynchronous Method Invocation — асинхронный вызов методов).

В итоге, наиболее приемлемая стратегия совместного использования CORBA и MPI заключается в следующем. Первоначально инициализировать систему CORBA внутри MPI процессов, далее в каждом процессе создать однопоточный объектный адаптер, активизировать в нем CORBA

объект и запустить обработку объектных запросов, а MPI код поместить внутри тех или иных методов CORBA объекта, которые следует вызывать асинхронно. Данный подход предназначен для проектирования параллельных распределенных объектно-ориентированных моделей, в том числе, с повторным использованием существующего прикладного программного обеспечения, реализованного с помощью MPI. Параллельным объектом назовем объект CORBA, методы которого вызываются асинхронно. SPMD-объектом назовем параллельный объект, методы которого содержат MPI код и вызываются скоординировано с методами других объектов в наборе.

При построении SPMD объектов с повторным использованием существующего прикладного программного обеспечения, необходимо пройти еще один этап.

4. Декомпозиция MPI программы. В MPI-программе переплетены все возможные модели поведения процессов, поэтому при декомпозиции программы возможно определение нескольких классов объектов, которые включают только один из вариантов. Это даст возможность избавиться от указанного недостатка разработки под MPI и сделает объектно-ориентированную модель более понятной, позволив четко определить отношения между объектами – клиент-сервер. Во время запуска в MPI-процессах можно выделить фрагменты кода, включающие обмены между процессами, и фрагменты, выполняемые независимо. Первые необходимо включить в те методы SPMD объекта, которые будут вызываться асинхронно коллективно. Вторые можно скрыть в методах, вызываемых как синхронно, так и асинхронно. SPMD объектами задаются не только параллельные процессы, но и распределенные данные, для которых определена стратегия параллельного чтения или записи.

5.4.3 Построения интегрированной прикладной программной системы

Предложенный подход был применен при интеграции MPI пакета линейной алгебры PETSc (The Portable, Extensible Toolkit for Scientific Computation `http://www.mcs.anl.gov/petsc/petsc-as/` — переносимые расширяемые инструментальные средства для научных расчетов) в параллельную распределенную объектно-ориентированную вычислительную среду для конечно-элементного анализа.

При решении систем линейных алгебраических уравнений, используются модули параллельных решателей систем линейных алгебраических уравнений с предобуславливателями и распределенные векторы и матри-

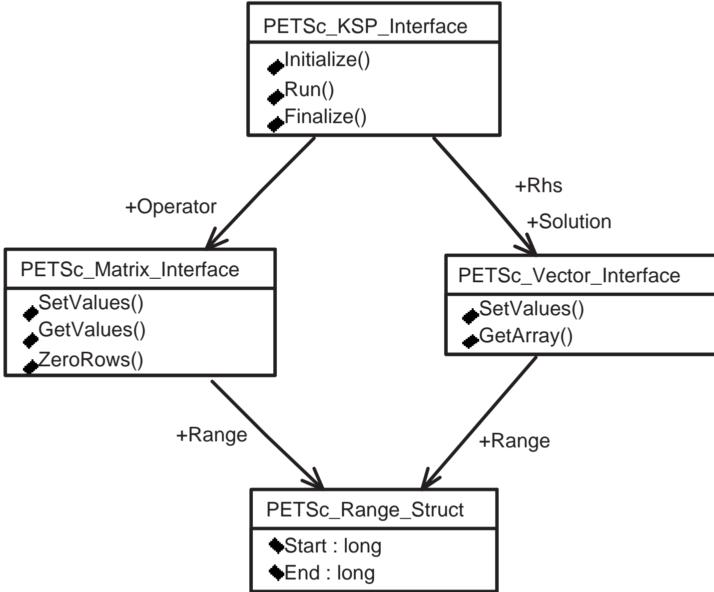

Рис. 9. Модель IDL описания одной нити решения над одной частью данных

цы с различными схемами хранения элементов. В прикладных программах имеется возможность выполнять как коллективные, так и отдельные операции над распределенными данными. Таким образом, параллельную распределенную объектно-ориентированную модель можно задать тремя классами SPMD объектов, интегрированными соответственно с модулями решателей, матриц и векторов.

Модель IDL описания одной нити решения над одной частью данных представлена на рисунке 9. Параллельные объекты `PETSc_KSP_Interface` включают методы создания, решения и удаления системы линейных алгебраических уравнений, задаваемой связанными с ними параллельными объектами частей матрицы `Operator` (интерфейс `PETSc_Matrix_Interface`), правой части `Rhs` и решения `Solution` (интерфейс `PETSc_Vector_Interface`). Объекты частей матрицы и вектора содержат методы доступа к наборам элементов и свои рамки `PETSc_Range_Struct`.

Рассмотрим реализацию объектов серверной стороны на языке C++. Объект `PETSc` выполняет инициализацию MPI и PETSc. Наборы CORBA

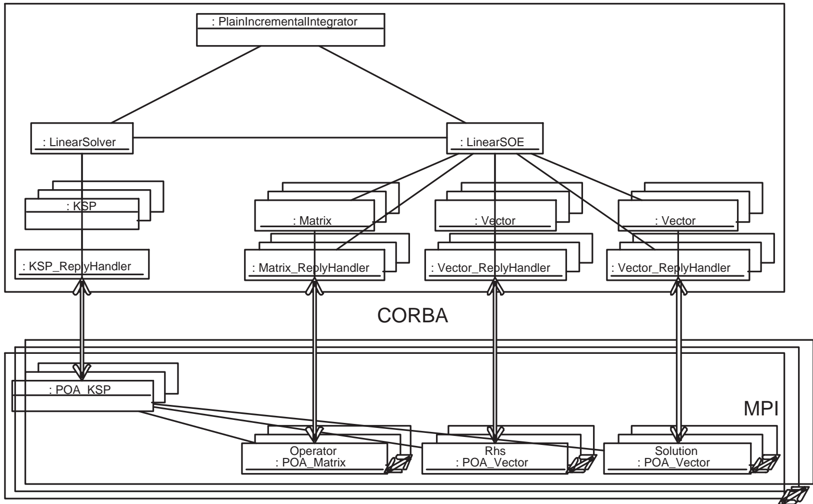

Рис. 10. Организация коммуникаций между объектами, инкапсулирующими PETSc и объектами численной модели

объектов PETSc_Matrix, PETSc_Vector и PETSc_KSP задают соответственно распределенную матрицу, распределенный вектор и параллельный решатель. Одна нить решения связана с решателем, одной частью матрицы и двух векторов (асинхронные вызовы с клиентской стороны отмечены половинчатыми стрелками).

Основным этапом интеграции PETSc является включение объектов клиентской стороны в подсистему решения систем линейных алгебраических уравнений, определенную в объектно-ориентированной модели метода конечных элементов. Класс параллельного решения PETSc_LinearSolver — клиент набора объектов PETSc_KSP, синхронизация которых осуществляется посредством одного общего обработчика обратного вызова обозначенного как PETSc_KSP_ReplyHandler. Класс распределенной системы уравнений PETSc_LinearSOE — клиент наборов объектов PETSc_KSP, PETSc_Matrix и двух наборов PETSc_Vector (для векторов правой части и решения системы линейных алгебраических уравнений).

Для каждого набора создается набор соответствующих обработчиков обратного вызова, это позволяет выполнять серии одновременных асинхронных вызовов. При добавлении блока в распределенную систему уравнений PETSc_LinearSOE, блок делится на части, предназначенные для от-

правки соответствующим частям распределенных матрицы и вектора правой части. Полученные фрагменты добавляемого в систему блока ставятся в очереди обработчиков обратного вызова для асинхронной отправки соответствующим удаленным объектам. Таким образом, блок отправляется одновременно на все части распределенной матрицы, которым он принадлежит, и, если в это время добавляется еще один блок, то в коллективную операцию включаются другие части распределенной матрицы, к занятым же образуется очередь.

Взаимодействия объектов продемонстрированы на рисунке 10. Серверные процессы обмениваются между собой по MPI. Управляющий клиентский процесс обменивается с серверными по CORBA, при этом запуск коллективных операций производится асинхронно с помощью AMI.

В настоящее время интересной заменой CORBA представляется технология Ice (Internet Communication Engine) `http://www.zeroc.com/ice.html`. Технология была создана под влиянием CORBA, сравнима по возможностям, но лишена ее недостатков. Ice [8] позиционируется как эффективная и масштабируемая, при этом лёгкая система для практического применения. Кроме того, Ice является объектно-ориентированным ППО, то есть предоставляет средства для разработки объектно-ориентированных распределенных приложений. Данные приложения реализуются в соответствии с известной моделью «клиент-сервер». Клиентами называются активные сущности, запрашивающие определенные сервисы у сервера. Серверами называются пассивные сущности, предоставляющие сервисы в ответ на запросы клиентов. Зачастую отдельные части распределенного приложения не являются «чистыми» клиентами или серверами, а сочетают в себе обе функции. Поэтому корректнее говорить о роли, которую играет та или иная часть приложения в контексте конкретного удаленного вызова. Ice поддерживает разработку приложений в гетерогенной среде, то есть отдельные части приложения (клиенты и серверы) могут быть реализованы на различных языках программирования и работать под управлением различных операционных систем на различных вычислительных платформах.

Архитектура Ice предоставляет разработчикам программного обеспечения следующие возможности. Ice полностью придерживается объектно-ориентированной парадигмы. Все операции, вызовы осуществляются посредством Ice-объектов. Поддерживаются синхронные и асинхронные вызовы методов и синхронная и асинхронная диспетчеризация, что позволяет разработчику самому определять модель взаимодействия удаленных объектов. Каждый Ice-объект может иметь несколько интерфейсов, через которые с ним осуществляется взаимодействие. ППО Ice не зависит от ап-

паратных особенностей платформы. Приложения, использующие Ice могут быть написаны на любом языке программирования (в данный момент поддерживаются: C++, C#, Java, Python, Ruby, PHP). API ППО Ice полностью кросплатформенный и может быть скомпилирован для любой операционной системы. Среда выполнения приложений Ice полностью многопоточна. Это позволяет разработчику создавать высокопроизводительные клиент-серверные приложения, не используя платформенный или языкозависимый инструмент для организации многопоточности. Ice распространяется под двойной лицензией. Под свободной GPL и коммерческой.

Таким образом, выбор любой из упомянутых выше технологий ППО неизбежно связан с определенными проблемами, будь то сложность технологии или низкая производительность. В любом из случаев разработчику приходится идти на некоторый компромисс.

В заключение необходимо отметить, что основным принципом при выборе того или иного типа промежуточного ПО должно быть его соответствие тем условиям, в которых осуществляется взаимодействие при параллельных вычислениях, а не модность или современность того или иного программного обеспечения. Кроме того, в ряде случаев целесообразно комбинирование различных типов ППО для достижения необходимой функциональности, тем более что многие из тех технологий, что были рассмотрены, предоставляют удобные интерфейсы для взаимодействия друг с другом.

Практические задания

Специальное внимание при выполнении практических заданий должно уделяться анализу работоспособности и эффективности различных реализаций SpMV на примере тестовых разреженных матриц, заимствованных из известного проекта MatrixMarket (`http://math.nist.gov/MatrixMarket/`). Все данные для проведения вычислительных экспериментов должны браться из MatrixMarket — репозитория тестовых данных, предназначенных для сравнительного изучения численных алгоритмов линейной алгебры. Проект насчитывает около пятисот видов разреженных матриц, получаемых при решении различных прикладных задач. Кроме того, данный ресурс включает программные инструменты для генерации тестовых матриц.

Для приобретения практического опыта использования внешних библиотек и вызова вычислительных процедур из соответствующих библиотек в заданиях предлагается использование ввода/вывода в формате разреженных матриц Matrix Market и сравнение самостоятельно выполненных параллельных реализаций алгоритма SpMV между собой и известными реализациями из популярных численных библиотек.

Раздел 1

Задание 1. Написать программу SpMV на языке C для матрицы хранимой по строкам и столбцам.

Задание 2. Написать программу SpMV на языке C++ с различными абстракциями.

Задание 3. Продемонстрировать работу программ SpMV на C, C++ на Windows и Linux системах с привязкой к произвольному ядру процессора.

Задание 4. Сравнить реализуемый SpMV с известной реализацией BLAS на C++ развиваемой NIST `http://math.nist.gov/spblas/`.

Задание 5. Инкапсулировать блок ввода/выводы матриц в формате Matrix Market `http://math.nist.gov/MatrixMarket/mmio-c.html`.

Задание 6. Реализовать SpMV для матриц хранимых в сжатом формате CSR, COO, ELL.

Задание 7. Оценить вычислительные затраты SpMV для матриц из коллекции Matrix Market.

Раздел 2

Задание 1. Реализовать параллельную версию SpMV при помощи MPI.

Задание 2. Вычислить ускорение и эффективность параллельного алгоритма SpMV на MPI для матриц из библиотеки Matrix Market.

Задание 3. Провести исследование масштабируемости алгоритма SpMV при увеличении размерности матрицы и числа процессоров и сравнить с аналитическими оценками из раздела 1.4.3.

Задание 4. Рассмотреть влияние различных вариантов выбора коллективных коммуникаций.

Раздел 3

Задание 1. Реализовать алгоритм SpMV на промежуточном обеспечении OpenMP.

Задание 2. Вычислить ускорение и эффективность параллельного алгоритма SpMV на OpenMP для матриц из библиотеки Matrix Market.

Задание 3. Сравнить параллельное ускорение и эффективность параллельного алгоритма SpMV на MPI, OpenMP.

Задание 4. Исследовать влияние различных вариантов привязок процессов к ядрам, сокетам.

Раздел 4

Задание 1. Реализовать алгоритм SpMV на промежуточном ПО CUDA.

Задание 2. Вычислить ускорение и эффективность параллельного алгоритма SpMV на CUDA для матриц из библиотеки Matrix Market.

Задание 3. Оценить влияние сжатого формата представления матриц на параллельное ускорение.

Задание 4. Сравнить параллельное ускорение и эффективность параллельного алгоритма SpMV на MPI, OpenMP, CUDA.

Задание 5. Сравнить влияние точности вычислений на графическом процессоре.

Раздел 5

Задание 1. Реализовать алгоритм SpMV на промежуточном обеспечении OpenCL и сравнить временные затраты при запуске CPU и GPU для разреженных и плотных матриц.

Задание 2. Реализовать алгоритм SpMV на промежуточном обеспечении MPI+OpenMP.

Задание 3. Сравнить параллельное ускорение и эффективность параллельного алгоритма SpMV на гибридном модели MPI+OpenMP с вариантами MPI, OpenMP, CUDA.

Задание 4. Исследовать влияние различных вариантов привязок процессов к ядрам, сокетам.

Список сокращений

ВС — вычислительная система;
МВС — многопроцессорная вычислительная система;
СЛАУ — система линейных алгебраических уравнений;
ПО — программное обеспечение;
ППО — промежуточное программное обеспечение;
ОС — операционная система;
ООП — объектно-ориентированное программирование;
AMI — асинхронный вызов методов;
API — программный интерфейс приложения;
CORBA — ППО объектных запросов;
CU — вычислительное устройство;
CPU — центральный процессор;
CUDA — ППО для вычислений на графическом процессоре;
CSR — построчный формат хранения разряженных матриц;
GPU — графический процессор;
GPGPU — вычисления общего назначения на GPU;
ICE — объектно-ориентированное ППО;
ID — идентификатор;
IDL — язык описания интерфейсов;
MIMD — архитектура со множеством потоков команд и множеством потоков данных;
MPMD — архитектура со множеством программ, множество потоков данных;
MPI — ППО обмена сообщениями;
OpenMP — ППО общей памяти;
OpenCL — ППО параллельного программирования для различных типов процессоров;
ORB — брокер объектных запросов;
PE — процессорный элемент;
SIMD — архитектура с одним потоком команд и многими потоками данных;
SMP — архитектура симметричной мультипроцессорной системы;
SpMV — матрично-векторное произведение;
SPMD — архитектура с одной программой и несколькими потоками данных;
RAM — оперативная память;
RPC — ППО удаленный вызов процедур.

Предметный указатель

Коммуникации
    коллективные, 58
    оценка, 33
    типы, 52
Модели параллелизма
    параллелизм данных, 93, 101
    параллелизм задач, 93, 101
Модель
    архитектуры, 12
        классификация, 12
    параллелизма
        обмен сообщениями, 16
        общая память, 17
        параллелизм данных, 17, 78
    программирования, 12
    программная
        вычислительной системы, 8
    вычислений, 12
Парадигма программирования
    клиент-сервер, 130
    компонентный поход, 22
    объектно-ориентированный подход, 21, 41
    параллельная, 10
    распределенная, 10
Привязка
    потоков, 73
    процессов, 57
Программное обеспечение
    функции, 9
    классификация, 9
    прикладное, 8
    промежуточное, 8
Промежуточное ПО
    адаптация к архитектуре, 20
    инвариантность, 18
    передачи сообщений, 10, 46
    платформонезависимость, 19
    система управления, 24
    требования, 18
    удаленного вызова процедур, 9

Список литературы

1. Рычков В. Н., Красноперов И. В., Копысов С. П. Параллельные распределенные вычисления и реализующие их технологии [Электронный ресурс] // Препринт ИПМ УрО РАН. — 2001. — 43 с. — Электрон. версия печат. публ. — Режим доступа: http://www.udman.ru/iam/ru/node/3978. - Дата доступа: 23.03.2012.

2. Эммерих В., Аояма М., Свентек Д. Влияние исследований на технологию промежуточного программного обеспечения [Электронный ресурс]. — Режим доступа: http://www.citforum.urc.ac.ru/SE/middleware/history/. - Дата доступа: 23.03.2012.

3. Олифер Н. А., Олифер В. Г. Сетевые операционные системы [Электронный ресурс]. — Режим доступа: http://citforum.ru/operating_systems/sos/glava_12.shtml . - Дата доступа: 23.03.2012.

4. Foster I. Designing and Building Parallel Programs, Second Edition [Electronic resource]. — Addison-Wesley, 1995. — The electronic version of print. publ. — Mode of access: http://www.mcs.anl.gov/~itf/dbpp/. - Date of access: 23.03.2012.

5. Рычков В. Н., Красноперов И. В., Копысов С. П. Промежуточное программное обеспечение для высокопроизводительных вычислений [Электронный ресурс] // Вычислительные методы и программирование. — 2001. — Т. 2, № 1. — С. 109–124. — Электрон. версия печат. публ. — Режим доступа: http://num-meth.srcc.msu.ru/zhurnal/tom_2001/pdf/art2_8.pdf. - Дата доступа: 23.03.2012.

6. Цимбал А. Технология CORBA для профессионалов. — С.-Петербург.: Питер, 2001. — 624 с.

7. Гайсарян С. С. Объектно-ориентированные технологии проектирования прикладных программных систем [Электронный ресурс]. — Режим доступа: http://citforum.ru/programming/oop_rsis/. - Дата доступа: 23.03.2012.

8. Сухорослов О. В. Промежуточное программное обеспечение Ice [Электронный ресурс] // Труды ИСА РАН. — 2008. — Т. 2. —

С. 33–67. — Электрон. версия печат. публ. — Режим доступа: http://www.isa.ru/proceedings/images/documents/2008-32/33-67.pdf. - Дата доступа: 23.03.2012.

9. Копысов С. П., Новиков А. К., Тонков Л. Е., Береснев Д. В. Методы привязки параллельных процессов и потоков к многоядерным узлам вычислительных систем [Электронный ресурс] // Вестник Удмурсткого университета. Математика. Механика. Компьютерные науки. — 2010. — № 1. — С. 123–132. — Электрон. версия печат. публ. — Режим доступа: http://vst.ics.org.ru/uploads/vestnik/1_2010/vu10111.pdf. - Дата доступа: 23.03.2012.

10. Баландин М. Ю., Шурина Э. П. Методы решения СЛАУ большой размерности [Электронный ресурс]: учеб.пособие.— Новосибирск: Изд-во НГТУ, 2000. — Электрон. версия печат. публ. — 70 с. — Режим доступа: http://fpmi.ami.nstu.ru/arhive/courses/umf/MethodsSLE.pdf. - Дата доступа: 23.03.2012.

11. Якобовский М. В., Кулькова Е. Ю. Решение задач на многопроцессорных вычислительных системах с разделяемой памятью [Электронный ресурс]: учеб. пособие.— М.: СТАНКИН, 2004. — Электрон. версия печат. публ. — 30 с. — Режим доступа: http://www.imamod.ru/~serge/arc/stud/Jackob_2.pdf. - Дата доступа: 23.03.2012.

12. Saad Y. Iterative Methods for Sparse Linear Systems, Second Edition [Electronic resource]. — SIAM, 2003. — 567 pp. — The electronic version of print. publ. — Mode of access: http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf. - Date of access: 23.03.2012.

13. Шпаковский Г. И., Серикова Н. В. Программирование для многопроцессорных систем в стандарте MPI [Электронный ресурс]: учеб.пособие.— Мн.: БГУ, 2002. — 323 с. — Электрон. версия печат. публ. — Режим доступа: http://www.cluster.bsu.by/download/book_PDF.pdf. - Дата доступа: 23.03.2012.

14. Антонов А. С. Параллельное программирование с использованием технологии OpenMP [Электронный ресурс]: учеб. пособие. — М.: Изд-во МГУ, 2009. — Электрон. версия печат. публ. — 77 с. — Режим доступа: http://parallel.ru/info/parallel/openmp/OpenMP.pdf. - Дата доступа: 23.03.2012.

15. Боресков А. В., Харламов А. А. Основы работы с технологией CUDA. — М.: ДМК Пресс, 2010. — 232 с.

16. Сандерс Д., Кэндрот Э. Технология CUDA в примерах. — М.: ДМК Пресс, 2011. — 232 с.

17. TESLA: Инструменты разработчика [Электронный ресурс]. — Режим доступа: `http://www.nvidia.ru/object/tesla_software_ru.html`. - Дата доступа: 23.03.2012.

18. Heterogeneous Computing with OpenCL / B. Gaster, L. Howes, D. R. Kaeli et al. — Morgan Kaufmann, 2011. — 296 pp.

19. OpenCL Programming Guide / A. Munshi, B. R. Gaster, T. G. Mattson at al. — Addison-Wesley, 2011. — 246 pp.

20. OpenCL для NVIDIA [Электронный ресурс]. — Режим доступа: `http://www.nvidia.ru/object/cuda_opencl_new_ru.html`. - Дата доступа: 23.03.2012.

21. Хьюз К., Хьюз Т. Параллельное и распределенное программирование на C++. — М.: Издат. дом «Вильямс», 2004. — 672 с.

22. Горобец А.В., Суков С.А., Железняков А.О., Богданов П.Б., Четверушкин Б.Н. Применение GPU в рамках гибридного двухуровневого распараллеливания MPI+OpenMP на гетерогенных вычислительных системах // Параллельные вычислительные технологии 2011: Междунар. науч. конф. МГУ. М., 2011. С. 452–460. — Электрон. версия печат. публ. — Режим доступа: `http://omega.sp.susu.ac.ru/books/conference/PaVT2011/short/152.pdf`. - Дата доступа: 23.03.2012.

23. Копысов С. П., Красноперов И. В., Рычков В. Н. Совместное использование систем промежуточного программного обеспечения CORBA и MPI // Программирование. — 2006. — Т. 32, № 5. — С. 51–61. — Электрон. версия печат. публ. — Режим доступа: `http://www.springerlink.com/content/r87123q445034811/`. - Дата доступа: 23.03.2012.

Для заметок

Учебное издание

Сергей Петрович Копысов
Александр Константинович Новиков
ПРОМЕЖУТОЧНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ
Учебное пособие

Авторская редакция

Подписано в печать 01.02.12. Формат $60 \times 84\,1/16$.
Печать офсетная. Усл.печ.л. 8,02.
Заказ № . Тираж 50 экз.

Издательство «Удмуртский университет»
426034, г. Ижевск, ул. Университетская, 1, корп. 4.