Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.А. МАЛЯВКО

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ОСНОВЕ ТЕХНОЛОГИЙ OpenMP, MPI, CUDA

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2015

УДК 004.424.5.032.24(075.8)
М 219

Рецензенты:

канд. техн. наук, доцент *А.В. Гунько*
канд. техн. наук, доцент *С.П. Ильиных*

Работа подготовлена на кафедре вычислительной техники НГТУ
для студентов старших курсов специальностей 230000
«Информатика и вычислительная техника»

Малявко А.А.

М 219 Параллельное программирование на основе технологий OpenMP, MPI, CUDA: учеб. пособие / А.А. Малявко. – Новосибирск: Изд-во НГТУ, 2015. – 116 с.

ISBN 978-5-7782-2614-2

Пособие содержит: краткое введение в архитектуру параллельных вычислительных систем; введение в проблематику параллельного программирования для параллельных систем различных классов: «множественный поток команд, множественный поток данных» (МКМД) с общей памятью, МКМД с распределенной памятью и «одиночный поток команд, множественный поток данных» (ОКМД); сведения о технологии OpenMP и ее использовании при разработке параллельной программы для вычислительной системы с общей памятью, об архитектуре NVIDIA CUDA и ее использовании при разработке параллельной программы для графического процессора, об интерфейсе передачи сообщений MPI, а также о библиотеке MPE и их использовании при разработке параллельных программ для вычислительных систем с распределенной памятью.

Предназначено для студентов старших курсов (бакалавриат, магистратура), обучающихся по укрупненной группе направлений и специальностей 230000 «Информатика и вычислительная техника». Может использоваться студентами, магистрантами и аспирантами других специальностей при изучении родственных дисциплин, а также преподавателями смежных дисциплин.

УДК 004.424.5.032.24(075.8)

ISBN 978-5-7782-2614-2

Введение

В пособии содержится краткое введение в проблематику параллельного программирования для параллельных систем различных классов (с общей памятью, с распределенной памятью, графические процессоры) и основные сведения о наиболее популярных технологиях, используемых при параллельном программировании.

Первый раздел включает в себя краткие сведения об архитектурах многопроцессорных систем, необходимые для понимания существа параллельных вычислений.

Во втором разделе рассматриваются: понятие параллелизма, основные виды параллелизма, описывается общая техника распараллеливания программ, а также специфические проблемы, сопровождающие разработку и отладку параллельных программ.

Третий раздел посвящен детальному изложению технологии параллельного программирования OpenMP для систем с общей памятью.

В четвертом разделе рассматривается архитектура CUDA как совокупность программно-аппаратных средств, обеспечивающих неграфические параллельные вычисления на графических процессорах производства компании NVIDIA.

Пятый раздел содержит подробное описание интерфейса передачи сообщений MPI, являющегося в настоящее время общепризнанным стандартом де-факто для программирования параллельных вычислительных систем с распределенной памятью.

Материал разделов 3–5, содержащих описание конкретных технологий параллельного программирования, сопровождается контрольными вопросами.

1. Архитектура параллельных вычислительных систем

Применение параллельных вычислительных систем (ПВС) и суперкомпьютеров является стратегическим направлением развития вычислительной техники [1]. Это вызвано не только принципиальным ограничением максимально возможного быстродействия обычных последовательных ЭВМ, но и практически постоянным наличием вычислительных задач, для решения которых возможностей существующих средств вычислительной техники всегда оказывается недостаточно.

Выход из создавшегося положения напрашивался сам собой. Если один компьютер не справляется с решением задачи за нужное время, то попробуем взять два, три, десять компьютеров и заставим их *одновременно* работать над различными частями общей задачи, надеясь получить соответствующее ускорение.

Объединение компьютеров в единую систему повлекло за собой множество следствий. Чтобы обеспечить отдельные компьютеры работой, необходимо исходную задачу разделить на фрагменты, которые можно выполнять *независимо* друг от друга. Так стали возникать специальные численные методы, допускающие возможность подобного разделения. Чтобы описать способ одновременного выполнения разных фрагментов задачи на разных компьютерах, потребовались специальные языки программирования, специальные операционные системы и т. д. Постепенно такие слова, как «одновременный», «независимый» и похожие на них стали заменяться одним словом *«параллельный».* Всё это синонимы, если иметь в виду описание каких-то процессов, действий, фактов, состояний, не связанных друг с другом. Ничего другого слова «параллелизм» и «параллельный» в областях, относящихся к компьютерам, не означают.

Существует множество способов организации параллельно работающих вычислительных систем [1, 2]. Параллельность вычислений, когда в один и тот же момент выполняется одновременно несколько операций обработки данных, осуществляется в основном за счет введения избыточности функциональных устройств (многопроцессорности) [3]. В этом случае можно достичь ускорения процесса решения

вычислительной задачи, если осуществить разделение применяемого алгоритма на информационно независимые части и организовать выполнение каждой части вычислений на разных функциональных устройствах (процессорах, ядрах, арифметико-логических устройствах и т. д.). Подобный подход позволяет выполнять необходимые вычисления с меньшими затратами времени по сравнению с последовательной организацией вычислений. Возможность получения максимального ускорения ограничивается (по крайней мере, в принципе) только числом имеющихся процессоров и количеством параллельно выполняющихся частей в вычислениях.

Рассматривая проблемы организации параллельных вычислений, следует различать следующие известные режимы выполнения независимых частей программы:

- многозадачный режим (режим разделения времени), при котором для выполнения нескольких процессов используется единственный процессор. Этот режим является псевдопараллельным: активным (исполняемым) может быть один единственный процесс, а все остальные процессы находятся в состоянии ожидания своей очереди. Применение режима разделения времени может повысить эффективность организации вычислений (например, если один из процессов не может выполняться из-за ожидания вводимых данных, то процессор будет задействован для выполнения другого, готового к исполнению процесса). Кроме того, в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и др.), и как результат этот режим может быть использован при начальной подготовке параллельных программ;
- параллельные вычисления, когда в один и тот же момент может выполняться несколько команд обработки данных. Такой режим вычислений может быть обеспечен не только при использовании нескольких процессоров, но и при помощи конвейерных и векторных обрабатывающих устройств;
- распределенные вычисления – термин, который обычно применяют для указания параллельной обработки данных, когда используется несколько обрабатывающих устройств, достаточно удаленных друг от друга, в которых передача данных по линиям связи приводит к существенным временным задержкам. Как результат эффективная обработка данных при таком способе организации вычислений возможна только для параллельных алгоритмов с низкой интенсивностью потоков межпроцессорных передач данных. Перечисленные условия являются характерными, например, при организации вычислений в много-

машинных вычислительных комплексах, образуемых объединением нескольких отдельных ЭВМ с помощью каналов связи локальных или глобальных информационных сетей.

При разработке параллельных алгоритмов решения задач принципиальным моментом является анализ эффективности использования параллелизма [4], состоящий обычно в оценке получаемого *ускорения* процесса вычисления (сокращения времени решения задачи). Формирование подобных оценок ускорения может осуществляться применительно к выбранному вычислительному алгоритму (*оценка эффективности распараллеливания конкретного алгоритма*). Другой важный подход может состоять в построении оценок максимально возможного ускорения процесса получения решения задачи конкретного типа (*оценка эффективности параллельного способа решения задачи*).

Собственно параллелизм предполагает [5] наличие нескольких (n) устройств для обработки данных и алгоритм, позволяющий производить на каждом независимую часть вычислений. В конце обработки частичные данные собираются вместе для получения окончательного результата. В этом случае (пренебрегая накладными расходами на получение и сохранение данных) можно получить ускорение процесса в n раз. Далеко не каждый алгоритм может быть успешно распараллелен таким способом (естественным условием распараллеливания является вычисление независимых частей выходных данных по одинаковым, или сходным, процедурам; итерационность или рекурсивность вызывают наибольшие проблемы при распараллеливании).

Архитектура параллельных компьютеров с самого начала их создания и применения развивалась в самых различных направлениях. Большое разнообразие вычислительных систем породило естественное желание ввести для них какую-то классификацию. Эта классификация должна однозначно относить ту или иную вычислительную систему к некоторому классу, который, в свою очередь, должен достаточно полно ее характеризовать. Таких попыток предпринималось множество. Одна из первых классификаций, ссылки на которую наиболее часто встречаются в литературе, была предложена М. Флинном в конце 60-х годов прошлого века. Она базируется на понятиях двух потоков: команд и данных. На основе числа этих потоков выделяется четыре класса архитектур.

1. ОКОД (SISD – Single Instruction Single Data) – единственный поток команд и единственный поток данных. По сути – это классическая машина фон Неймана. К этому классу относятся все однопроцессорные системы.

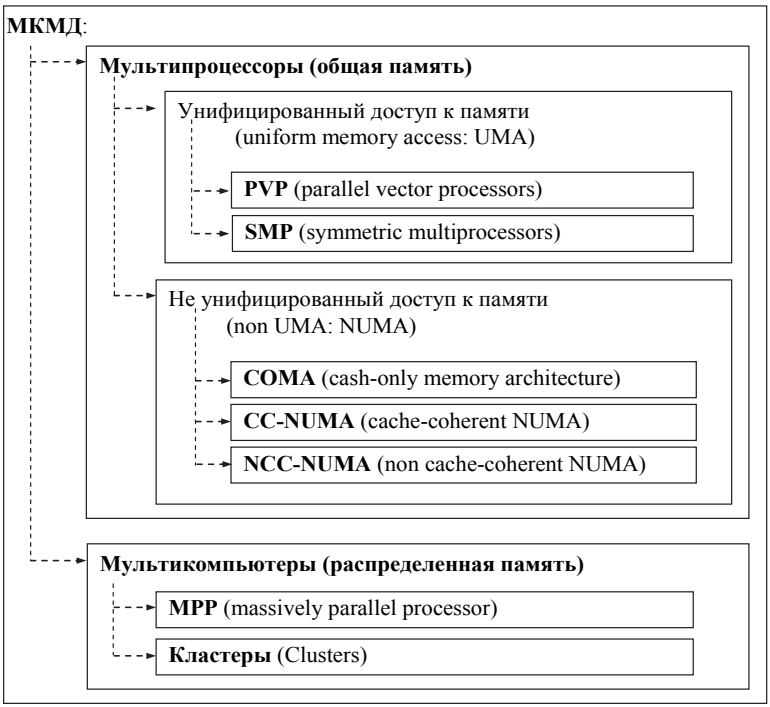
2. ОКМД (SIMD – Single Instruction Multiple Data) – единственный поток команд и множественный поток данных. Типичными представителями являются матричные компьютеры, в которых все процессорные элементы выполняют одну и ту же программу, применяемую к своим (различным для каждого процессора) локальным данным. Иногда к этому классу относят и векторно-конвейерные компьютеры, если каждый элемент вектора рассматривать как отдельный элемент потока данных. В настоящее время этот класс пополнился графическими процессорами, позволившими существенно увеличить достижимый порог производительности при сравнительно невысокой стоимости.

3. МКОД (MISD – Multiple Instruction Single Date) – множественный поток команд и единственный поток данных. Автор классификации не смог привести ни одного примера реально существующей системы, работающей на этом принципе. Представителями такой архитектуры можно считать высоконадежные вычислительные системы с дублированием процессов обработки одних и тех же данных на нескольких вычислительных устройствах с последующим мажоритарным выбором результатов решения задачи.

4. МКМД (MIMD – Multiple Instruction Multiple Date) – множественный поток команд и множественный поток данных. Разные вычислительные устройства обрабатывают собственные данные в соответствии с собственной программой каждое. К этому классу относятся практически все современные суперкомпьютеры. Состав и структура этого класса показаны на рисунке.

Компьютеры с общей памятью (или мультипроцессоры) состоят из нескольких (как правило, одинаковых) процессоров, имеющих равноприоритетный доступ к общей памяти с единым адресным пространством.

К этому классу относится большое количество векторно-конвейерных компьютеров, в свое время находившихся на верхушке списка самых высокопроизводительных систем. Другим типичным примером такой архитектуры являются компьютеры класса SMP (Symmetric Multi Processors), включающие в себя несколько процессоров, но одну память, комплект устройств ввода/вывода и операционную систему (другим классификатором таких систем является термин UMA – uniform memory access). Достоинством компьютеров с общей памятью является (относительная) простота программирования параллельных задач (нет необходимости заниматься организацией пересылок сообщений между процессорами с целью обмена данными), минусом – недостаточная масштабируемость (при увеличении числа процессоров



Структура класса МКМД

резко возрастает конкуренция за доступ к общим ресурсам, в первую очередь к памяти, что ограничивает суммарную производительность системы). Реальные SMP-системы содержат обычно не более 32 процессоров, для дальнейшего наращивания вычислительных мощностей подобных систем используется NUMA-технология (non uniform memory access).

При фактическом различии механизмов доступа к собственной памяти и памяти других процессоров в NUMA-системах реализуется виртуальная адресация, позволяющая пользовательским программам рассматривать всю (физически) распределенную между процессорами память как единое адресное пространство. Недостатками NUMA-компьютеров являются значительная разница времени обращения к собственной (локальной) памяти данного процессора и к памяти сторонних процессоров, а также проблема когерентности кэша (cache coherence problem) – в случае сохранения процессором П1 некоего значения в ячейке N1.

При последующей попытке прочтения данных из той же ячейки N1 процессором П2 последний получит значение, которое может не совпадать с истинным значением переменной, если кэш процессора П1 еще не ‘сброшен’ в память (о чем процессор П2 «знать» не обязан). Для решения проблемы когерентности (соответствия, одинаковости) кэша предложена и реализована архитектура ccNUMA (cache coherent NUMA), позволяющая (прозрачными для пользователя средствами) достигать соответствия кэшей процессоров (что требует дополнительных ресурсов и соответственно снижает производительность).

Для всех видов как векторно-конвейерных, так и *UMA-архитектур увеличение количества процессоров связано с непропорционально большим ростом стоимости механизмов обеспечения «прозрачности» доступа к любому участку «общей» памяти. В силу этого ни одна система с общей памятью в настоящее время уже не входит в список 500 суперкомпьютеров (хотя ранее, до 2003-го года, входили и даже составляли до половины всего списка).

Другим стремительно развивающимся направлением развития суперкомпьютеров являются системы с распределенной памятью. Идея построения вычислительных систем данного класса очень проста: берется какое-то количество вычислительных узлов, которые объединяются друг с другом некоторой коммуникационной средой. Каждый вычислительный узел имеет один или несколько процессоров и свою собственную локальную память, разделяемую этими процессорами. Распределенность памяти означает то, что каждый процессор имеет непосредственный доступ только к локальной памяти своего узла. Доступ к данным, расположенным в памяти других узлов, выполняется дольше и другими более сложными способами. В последнее время в качестве узлов все чаще и чаще используют полнофункциональные компьютеры, содержащие, например, и собственные внешние устройства. Коммуникационная среда может специально проектироваться для данной вычислительной системы либо быть стандартной сетью, доступной на рынке.

Преимуществ у такой схемы организации параллельных компьютеров много. В частности, покупатель может достаточно точно подобрать конфигурацию в зависимости от имеющегося бюджета и своих потребностей в вычислительной мощности. Соотношение цена/производительность у систем с распределенной памятью ниже, чем у компьютеров других классов. И главное, такая схема дает возможность практически неограниченно наращивать число процессоров в системе и увеличивать ее производительность. Большое число подключаемых

процессоров даже определило специальное название для систем данного класса – компьютеры с массовым параллелизмом, или массивно-параллельные компьютеры.

Основным преимуществом таких систем является масштабируемость (в зависимости от класса решаемых задач и бюджета может быть реализована система с числом узлов от нескольких десятков до нескольких тысяч). Различными производителями было создано большое количество массивно-параллельных суперкомпьютеров, в том числе Cray серии T3, IBM SP, Intel Paragon и другие. Однако сверхвысокая стоимость промышленных массивно-параллельных компьютеров не позволяла применить их в любой области, нуждающейся в системах высокой производительности. Это привело к развитию вычислительных кластеров.

Технологической основой развития кластеризации стали широкодоступные и относительно недорогие микропроцессоры и коммуникационные (сетевые) технологии, появившиеся в свободной продаже в 1990-х годах. Вычислительный кластер представляет собой совокупность вычислительных узлов (от десятков до сотен тысяч), управляющего компьютера и файл-сервера.

Вычислительные узлы и управляющий компьютер обычно объединяют как минимум две, обычно независимые, сети: управляющая сеть (служит целям управления вычислительными узлами) и (часто более производительная) коммуникационная сеть (непосредственный обмен данными между исполняемыми на узлах процессами). Управляющий узел обычно имеет выход в Интернет для доступа удаленных пользователей к ресурсам кластера. Файл-сервер (в небольших кластерах его функции выполняет управляющий компьютер) выполняет функции хранения программ пользователя. Администрирование кластера осуществляется с управляющего узла (или посредством удаленного доступа), пользователи имеют доступ к ресурсам кластера исключительно через управляющий компьютер (в соответствии с присвоенными администратором правами).

В настоящее время из списка 500 самых высокопроизводительных систем мира свыше 400 – это кластеры, остальные относятся к классу массивно-параллельных систем, при этом доля последних падает с появлением каждой следующей версии списка.

Дополнительные сведения об архитектуре параллельных вычислительных систем и методах разработки параллельных программ для них можно найти в работах [1–9].

2. Параллельные вычисления

Для того чтобы разрабатывать параллельные программы, необходимо иметь четкое представление о том, что такое параллелизм в вычислениях. Представим предельный случай отсутствия параллелизма: это так называемая Машина Тьюринга (МТ) – абстрактная математическая модель, с использованием которой были получены очень важные результаты в области вычислимости.

Машина Тьюринга представляет собой совокупность управляющего устройства, головки считывания/записи и бесконечную ленту для хранения обрабатываемых данных, разделенную на ячейки (рис. 2.1).

Рис. 2.1. Машина Тьюринга

Лента бесконечна в обе стороны, однако было доказано, что полубесконечная лента эквивалентна бесконечной.

В каждой ячейке может быть либо пусто, либо в ней хранится один из символов s_i алфавита МТ (сейчас представляет особый интерес случай, когда алфавит включает в себя два символа, например, двоичные цифры 0 и 1).

Управляющее устройство имеет некоторое множество состояний $\{q_0, q_1, \ldots\}$ и набор правил вида

$$\{q_{tj}, s_{tj}\} -> \{q_{tj+1}, s_{tj+1}, m\},$$

где q_{tj} q_{tj} и q_{tj+1} – состояния в моменты времени t_j и t_{j+1} соответственно; s_{tj} – символ в текущей (находящейся под головкой) ячейке до момента t_j (стирается машиной из ячейки); s_{tj+1} – символ в ячейке, начиная с момента t_{j+1} (пишется головкой при выполнении действия по данному правилу); m – перемещение (на одну ячейку влево, на одну ячейку вправо или остаться на месте).

Машина Тьюринга представляет собой автомат, который способен выполнить любой алгоритм (вычислить результат решения задачи), для чего этот алгоритм надо представить в виде соответствующей системы правил. Легко можно видеть, что процесс вычислений машиной Тьюринга является строго последовательной цепочкой преобразования состояний одиночных ячеек ленты, т. е. не содержит параллелизма ни в каком виде.

Другим предельным случаем можно считать достижение максимально возможного параллелизма вычислений путем полностью аппаратной реализации алгоритма решения задачи. Пусть, например, требуется вычислить значение

$$r = a + bc.$$

В принципе можно представить себе специализированное устройство решения этой задачи, в котором для вычисления значения каждого *i*-го двоичного разряда результата *r* используется схема, построенная на логических элементах И, ИЛИ и НЕ, реализующая логическую функцию

$$\mathbf{r_i = F_i(a_0, a_1, a_2, \ldots a_n, b_0, b_1, b_2, \ldots b_n, c_0, c_1, c_2, \ldots c_n).}$$

Приведем такие функции для самых младших значений *i*:

$\mathbf{r_0 = (a_0 \,\&\, \sim(b_0 \,\&\, c_0)) \mid (\sim(a_0) \,\&\, (b_0 \,\&\, c_0))}$;

$\mathbf{r_1 = ((a_1 \,\&\, \sim(b_1 \,\&\, c_1)) \mid (\sim(a_1) \,\&\, (b_1 \,\&\, c_1)) \,\&\, \sim((a_0 \,\&\, b_0) \mid (a_0 \,\&\, c_0) \mid (b_0 \,\&\, c_0))}$
$\mathbf{\mid (\ldots)}$

r2 = …

…

Фрагмент принципиальной схемы устройства, реализующего эти функции, показан на рис 2.2.

Очевидно, что такой способ реализации вычислений способен обеспечить максимально возможную скорость преобразования исходных данных в результаты решения задачи, но не может быть широко применен на практике вследствие чрезвычайно высоких затрат на создание очень сложных и при этом уникальных устройств для каждой конкретной задачи.

Реальный параллелизм есть совместное применение нескольких программируемых универсальных (т. е. способных выполнить любой алгоритм) вычислительных устройств для решения одной задачи. Каждое такое устройство (процессор) способно выполнять операции преобразования данных определенной сложности, о которых принято говорить,

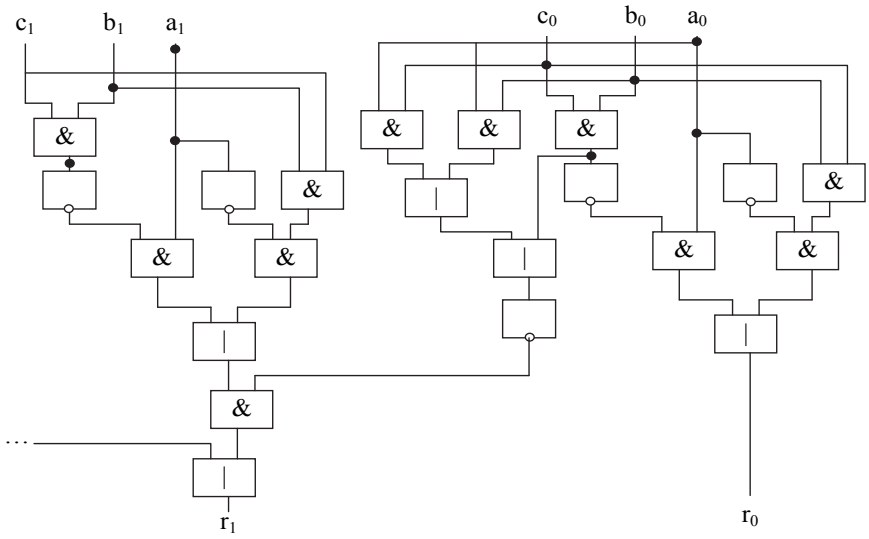


Рис. 2.2. Фрагмент схемы вычислительного устройства

что они являются наименьшими гранулами или зернами с точки зрения распределения общего объема вычислительной работы. Достижимый уровень распараллеливания, а следовательно, получения эффекта ускорения решения задачи по сравнению с вычислениями на одном программируемом универсальном устройстве зависит от очень многих факторов, главными из которых являются:

– степень внутреннего или собственного параллелизма конкретного алгоритма решения задачи (может существовать несколько алгоритмов с различной степенью параллелизма);

– структурные характеристики параллельной вычислительной системы (ОКМД или МКМД, общая или распределенная память, наличие/отсутствие кэшей памяти, …);

– технические характерики вычислительной системы (их очень много и их перечень зависит от класса системы, устройства памяти, коммуникационной среды, …).

2.1. Внутренний параллелизм алгоритма

Внутренний параллелизм, безусловно, является первичной характеристикой алгоритма при разработке параллельного варианта программы решения задачи. Для его выявления и анализа обычно предполагают, что вычислительная система обладает «идеальными» структурными и техническими характеристиками:

– имеется бесконечно много (точнее, столько, сколько нужно) параллельно работающих универсальных процессоров;

– все они работают синхронно и выполняют любую операцию за один и тот же интервал времени;

– система имеет бесконечно большую оперативную память;

– все передачи данных между процессорами и памятью, а также между самими процессорами осуществляются мгновенно и без конфликтов.

В качестве примера способа определения внутреннего параллелизма рассмотрим последовательность действий для вычисления значений корней полного квадратного уравнения $ax^2 + bx + c = 0$, которая определяется известными формулами:

$$x_1 = \left(-b + \sqrt{(b2 - 4ac)}\right) / 2a,$$

$$x_2 = \left(-b - \sqrt{(b2 - 4ac)}\right) / 2a.$$

Эта последовательность с учетом обычной гранулярности операций может выглядеть так:

1 $a2 \leftarrow 2a$ //вычисление промежуточной переменной $a2$
2 $a4 \leftarrow 4a$ //вычисление промежуточной переменной $a4$
3 $b\_neg \leftarrow -b$ //вычисление промежуточной переменной b_neg
4 $bb \leftarrow bb$ //вычисление промежуточной переменной bb
5 $ac4 \leftarrow a4c$ //вычисление промежуточной переменной $ac4$
6 $p\_sqr \leftarrow bb - ac4$ //вычисление подкоренного выражения p_sqr
7 $sq \leftarrow sqrt(p\_sqr)$ // извлечение квадратного корня
8 $w1 \leftarrow b\_neg + sq$ //вычисление промежуточной переменной $w1$
9 $w2 \leftarrow b\_neg - sq$ //вычисление промежуточной переменной $w2$
10 $root\_1 \leftarrow w1 / a2$ //вычисление первого корня уравнения
11 $root\_1 \leftarrow w2 / a2$ //вычисление второго корня уравнения

Поставим в соответствие каждому действию одну вершину графа и свяжем две вершины ориентированной дугой, если второе действие использует результат, вырабатываемый первым действием. Затем расположим на верхнем ярусе те и только те вершины, которые не имеют входящих дуг, т. е. зависят только от исходных данных. На всех следующих ярусах расположим те вершины, в которые входят такие дуги, хотя бы одна из которых выходит из какой-либо вершины предыдущего яруса. В результате получим так называемую ярусно-параллельную форму (ЯПФ) алгоритма (рис. 2.3).

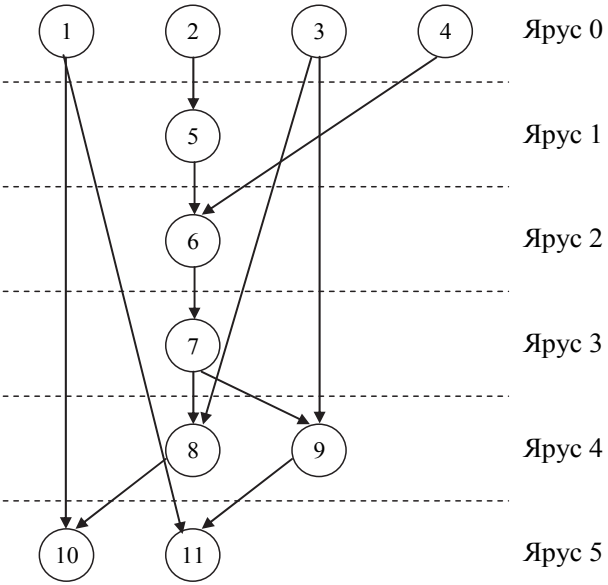


Рис. 2.3. Ярусно-параллельная форма алгоритма

Построив ЯПФ алгоритма, можно оценить степень внутреннего параллелизма и определить максимально возможное ускорение вычислений на идеальной вычислительной системе по сравнению с решением задачи на одном процессоре. Для этого используются ширина и высота ЯПФ.

Ширина ярусно-параллельной формы – характеристика алгоритма, показывающая максимально возможное количество параллельных ветвей. Для решения задачи распараллеливания могут быть использованы (в скобках указаны значения для рассмотренного примера):

- максимальная ширина (4);
- минимальная ширина (1);
- средняя ширина (1.83).

Высота ЯПФ (количество ее ярусов) – характеристика алгоритма, позволяющая оценить минимально возможное время его исполнения. Для данного примера высота ЯПФ равна 6. Отношение общего количества операций к высоте показывает максимально возможное ускорение исполнения параллельной программы по отношению к последовательной (для данного примера оно равно 1.83).

В реальной практике параллельного программирования для решения действительно больших задач ярусно-параллельные формы не используются вследствие слишком больших размеров графа и соответственно затрат компьютерного времени на его обработку. Действительно, если суперкомпьютеру с производительностью, например 1 петафлоп (10^{15} операций с плавающей точкой в секунду), нужен один час, для того чтобы решить задачу, то он должен выполнить $3.6 \cdot 10^{18}$ операций. Это и есть количество вершин графа ЯПФ. Количество дуг графа можно оценить величиной порядка $7 \cdot 10^{18}$ исходя из того, что для гранулярности на уровне процессорных операций в каждую вершину графа входит от одной до двух дуг. Для построения и анализа такого графа на каждую вершину и каждую дугу нужно затратить вряд ли менее чем 10 операций. Таким образом, получается, что на обработку ЯПФ нужно не менее чем $10(3.6+7)10^{18} \cong 10^{20}$ операций, что значительно превышает затраты на решение собственно самой задачи. В силу этого при практическом параллельном программировании обычно рассматриваются только циклические участки программы как кандидаты на распараллеливание вычислительного процесса.

2.2. Распараллеливание циклов

Известно, что более 90 % (как правило, значительно более, т. е. 99.999…%) из времени решения задачи выполняется всего 10 % (и меньше) текста программы (так называемое соотношение 90:10). Эти 10 % текста – циклы. Именно этими участками текста программы и следует заниматься при ее распараллеливании.

Распараллеливание циклов подразумевает большую степень гранулярности, чем пооперационное распараллеливание. Обычно гранулой при таком подходе считается одна итерация распараллеливаемого цикла.

Пусть в программе есть цикл, между итерациями которого нет информационной зависимости, например:

```
for( i = 0; i < N; i += 1)
    arr[i] = выражение, не содержащее arr[ k ] (k != i);
```

Итерации этого цикла могут быть выполнены в любом порядке, в том числе, одновременно друг с другом.

Для параллельной системы из *n* вычислительных элементов (процессоров) может быть реализовано:

Блочное распределение по *N / n* итераций (если *N* не делится нацело на *n*, то загрузка процессоров будет неодинаковой). При этом разные вычислительные узлы будут выполнять следующие фрагменты программы (в этих фрагментах через N0, N1, … Nk обозначены соответствующие количества итераций):

```
for( i = 0; i < N0; i += 1)          //доля процессора 0, N0 = N / n
    arr[i] = f( i, ...);
for( i = N0; i < N1; i += 1)         //доля процессора 1, N0 = 2*N / n
    arr[i] = f( i, ...);
...
for( i = Nk; i < N; i += 1)          //доля процессора n-1 = k, ...
    arr[i] = f( i, ...);
```

Циклическое распределение тоже по *N / n* итераций:

```
for( i = 0; i < N; i += n)           //доля процессора 0
    arr[i] = f( i, ...);
for( i = 1; i < N; i += n)           //доля процессора 1
    arr[i] = f( i, ...);
...
for( i = n-2; i < N; i += n)         //доля процессора n-1
    arr[i] = f( i, ...);
```

Блочно-циклическое распределение:

```
for ( k = 0; k < N; k += m)
    for( i = k * n; i < (k+1) * n; i += 1)          //доля процессора 0
        arr[i] = f( i, ...);
for ( k = 0; k < N; k += m)
    for( i = (k+1) * n; i < (k+2)* n; i += 1)       //доля процессора 1
        arr[i] = f( i, ...);
```

Для больших задач типичной является ситуация, когда основной объем вычислений определяется совокупностью вложенных друг в друга циклов (гнездом циклов):

```
for( i = 0; i < N; i += 1)
   for( j = 0; j < M; j += 1)
...
        for( k = U - 1; k >=0 ; k -= 1)
            < тело цикла >
```

Пространством итераций гнезда тесно вложенных циклов называют множество целочисленных векторов { [0, 0, U-1, …] , [0, 0, U-2], … }, координаты которых задаются значениями индексных переменных всех циклов данного гнезда.

Задача распараллеливания в этом случае сводится к разбиению этого множества векторов на подмножества таким образом, чтобы итерации этих подмножеств могли быть выполнены параллельно.

Существуют следующие методы разбиения: координат, гиперплоскостей, параллелепипедов, пирамид и более сложные.

Метод координат заключается в том, что пространство итераций фрагмента разбивается на гиперплоскости, перпендикулярные одной из координатных осей.

Пусть, например, есть гнездо циклов:

```
for( i = 1; i < N; i += 1)
   for( j = 0; j < M; j +=1)
       a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j ];
```

Разбиение пространства итераций по измерению *i* (гиперплоскостями, перпендикулярными оси *i*) приводит к разрыву информационных зависимостей. Следовательно, такое распараллеливание неприемлемо. Однако возможно применение метода координат с разбиением пространства итераций гиперплоскостями, перпендикулярными оси *j*.

Все такие операции назначаются для выполнения на одном процессоре.

При использовании метода гиперплоскостей пространство итераций размерности *n* разбивается на подпространства размерности от 1 до $n - 1$ таким образом, чтобы все операции, соответствующие точкам одного подпространства, могли выполняться параллельно. Каждое такое подпространство есть гиперплоскость, не обязательно перпендикулярная какой-либо оси координат.

Например, для такого гнезда:

```
for( i = 1; i < N; i += 1)
   for( j = 1; j < M; j +=1)
        a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j-1 ];
```

метод координат не годится.

Однако существуют удовлетворяющие условию $i + j = \text{const}$ гиперплоскости, проходящие через не содержащие информационных зависимостей вершины. При этом параллельно работающим процессорам следует назначить вычисления тела цикла для каждой из гиперплоскостей, для которых $i + j = \text{const}$.

Метод параллелепипедов является логическим развитием двух предыдущих методов и заключается в разбиении пространства итераций на *n*-мерные параллелепипеды с использованием ортогональных гиперплоскостей, весь объем каждого из которых назначается отдельному процессору.

Возможны и другие более сложные методы распределения пространства итераций при соблюдении основного принципа: параллельно выполняемые итерации не должны информационно зависеть друг от друга.

2.3. Преобразования циклов

Возможны ситуации, когда гнездо циклов исходной последовательной программы не может быть распараллелено никаким из описанных методов. Тогда может оказаться необходимым применить одно или несколько эквивалентных преобразований циклов для изменения характера информционных зависимостей между итерациями. К таким преобразованиям относятся следующие.

Перестановка циклов: внешний и внутренний циклы меняются местами (если это возможно). Например, фрагмент

```
for( i = 1; i < N; i += 1)
   for( j = 1; j < M; j +=1)
        a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j-1 ];
```

заменяется на

```
for( j = 1; j < M; j +=1)
   for( i = 1; i < N; i += 1)
        a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j-1 ];
```

Распределение цикла: один цикл расщепляется на два или более отдельных циклов. В этом примере то же самое гнездо циклов переписано так, что один внутренний цикл заменен на два, итерации выполняются в другой последовательности, но общее пространство итераций осталось неизменным:

```
for( i = 1; i < N; i += 1) {
   for( j = 1; j < M ; j += 2)
       a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j-1 ];
   for( j = 2 ; j < M; j += 2)
       a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j-1 ];
}
```

Слияние циклов: два цикла объединяются в один. В этом примере исходное гнездо двух циклов превращается в один цикл (общее количество итераций осталось таким же):

```
for( int i = 1, j = 1; (i < N) && (j < M); j += 1, i += j/M, j = (j>=M? 1 : j))
       a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j-1 ];
```

Развертка цикла: тело цикла выписывается два (или большее количество) раза при соответствующих изменениях способов модификации индексных переменных. Общий объем вычислений остается тем же самым, но количество итераций уменьшается:

```
for( i = 1; i < N; i += 1)
   for( j = 1; j < M; j +=2){
       a[ i ][ j ] = a[ i-1 ][ j ] + a[ i ][ j-1 ];
       a[ i ][ j+1 ] = a[ i-1 ][ j+1 ] + a[ i ][ j ];
   }
```

Расширение скаляра: одиночная переменная, значение которой вычисляется предыдущими итерациями и используется в последующих итерациях, заменяется элементом массива. Для примера рассмотрим фрагмент программы умножения матриц:

```
for( i = 0; i < N; i += 1)
   for( j = 0; j < N; j += 1 ){
       sum = 0.0;
       for( k = 0; k < N; k += 1)
           sum += a[ i ][ k ] * b[ k ][ j ];
       c[ i ][ j ] = sum;
   }
```

Используем целевой элемент вместо промежуточной переменной:

```
for( i = 0; i < N; i += 1)
   for( j = 0; j < N; j += 1 ){
       c[ i ][ j ] = 0.0;
       for( k = 0; k < N; k += 1)
            c[ i ][ j ] += a[i][k] * b[k][j];
   }
```

В результате этого преобразования исчезла зависимость итераций внешних циклов друг от друга и соответственно появилась возможность распараллелить их.

Разделение на полосы: итерации одного цикла разделяются на два вложенных цикла.

```
for( i = 0; i < N; i += 1)
   for( j = 0; j < N; j += 1 ){
       c[ i ][ j ] = 0.0;
       for( k = 0; k < N; k += 1)
            c[ i ][ j ] += a[i][k] * b[k][j];
   }

for( index = 0; index < N; index += step)    //предполагается, что
   for( i = index; i < index + step; i += 1)   //N делится на step
       for( j = 0; j < N; j += 1 ){
           c[ i ][ j ] = 0.0;
           for( k = 0; k < N; k += 1)
                c[ i ][ j ] += a[i][k] * b[k][j];
       }
```

Обычно разделение на полосы используется как часть разделения на блоки и развертки и сжатия.

Разделение на блоки: область итераций разбивается на прямоугольные блоки. Цель такого преобразования – согласование с размером области основной памяти, перемещаемой в кэш при обращении к любому ее элементу.

Пусть нужно из матрицы *a* построить транспонированную матрицу *b*:

```
for( i = 0; i < N; i += 1)
   for( j = 0; j < N; j += 1)
       b[ i ][ j ] = a [ j ][ i ];
```

При каждом обращении к *a*[*j*][*i*] в кэш вытаскивается много элементов строки матрицы *a*, которые до поры до времени в нем не нужны.

```
for( i = 0; i < N; i += step)
   for( j = 0; j < N; j += step)
      for( ii = 0; ii < min(N, i + step -1); ii += 1)
         for( jj = 0; jj < min(N, j + step – 1); jj += 1)
            b[ ii ][ jj ] = a [ jj ][ ii ];
```

Теперь два внутренних цикла работают с фрагментом матрицы размером step * step. Если по одному запросу к основной памяти в кэш переносится step элементов, то программа будет работать значительно быстрее.

Развертка и сжатие: комбинируются перестановка циклов, разделение на полосы и развертка.

Перекос цикла: выполняется модификация границ изменения индексов циклов таким образом, чтобы выявить параллельность фронта волны изменения элементов массивов. Например, при решении дифференциальных уравнений в частных производных обычно используется цикл

```
for( i = 0; i < N; i += 1)
   for( j = 0; j < N; j += 1)
   a[ i ][ j ] = ( a[ i-1 ][ j ] + a[ i ][ j-1 ] + a[ i+1 ][ j ] + a[ i ][ j+1 ]) / 4;
```

После такой модификации

```
for( i = 0; i < N; i += 1)
   for( j = i; j < i + N; j += 1)
   a[ i ][ j-i ] = ( a[ i-1 ][ j-i ] + a[ i ][ j-i-1 ] + a[ i+1 ][ j-i ] +
   + a[ i ][ j-i+1 ]) / 4;
```

явно обнаружится возможность распараллеливания по столбцам, для чего надо будет просто переставить циклы по *i* и по *j*.

2.4. Проблемы разработки параллельных программ

В рамках параллельной модели программирования существуют различные подходы к разработке, ориентированные на разные архитектуры высокопроизводительных вычислительных систем и различные инструментальные средства. К ним относятся следующие.

1. Модель общей памяти, в которой процессы (потоки) обращаются к общей памяти, имея общее адресное пространство и выполняя операции считывания/записи. Управление доступом к памяти осуществляется с помощью разных механизмов, таких, например, как семафоры. В рамках этой модели не требуется описывать обмен данными между задачами в явном виде. Это упрощает программирование. Вместе с тем особое внимание приходится уделять соблюдению детерминизма таким явлениям, как «гонки за данными», блокировки и т. д.

2. Модель передачи сообщений характеризуемая тем, что программа порождает несколько задач. Каждой задаче присваивается свой уникальный идентификатор. Взаимодействие задач осуществляется посредством отправки и приема сообщений. Новые задачи могут создаваться во время выполнения параллельной программы, несколько задач могут выполняться на одном процессоре.

3. Модель параллелизма данных, для которой характерно следующее:

- одна операция применяется к множеству элементов структуры данных. Программа содержит последовательность таких операций;
- гранулярность вычислений невелика;
- программист должен указать транслятору, как данные следует распределить между задачами.

При разработке параллельных программ обычно возникает целый ряд специфических проблем. Перечислим некоторые из них:

- необходимость управления работой множества процессов;
- организация межпроцессных взаимодействий;
- вероятность возникновения тупиковых ситуаций или взаимных блокировок (блокировки возникают, если один процесс или поток ожидает получения доступа к некоторому ресурсу, захваченному другим процессом, не предоставляя ему, в свою очередь, доступа к своему ресурсу);
- нелокальный и динамический характер ошибок, существенным образом усложняющий отладку программы;
- утрата детерминизма (предсказуемого поведения программы в процессе ее исполнения). Параллельная программа может давать разные результаты при повторных запусках даже без модификации кода и исходных данных. Это является следствием так называемых «гонок за данными» – одновременного и асинхронного доступа разных процессов или потоков к общим переменным;

- необходимость заботиться о масштабируемости программы (масштабируемость – это свойство вычислительной системы, компьютерной сети или процесса/процессов, которое заключается в сохранении способности эффективно работать при возрастании количества узлов);
- необходимость заботиться о сбалансированной загрузке вычислительных узлов.

Наиболее специфическими проблемами, возникающими при разработке параллельных программ в модели с общей памятью, являются:

- гонки за данными;
- блокировки;
- их разновидность – активные блокировки.

Гонки за данными. Они являются следствием таких информационных зависимостей, когда несколько потоков модифицируют содержимое одной и той же области памяти (переменной). Наличие гонок за данными далеко не всегда является очевидным. Пусть, например, два (или больше) потока время от времени выполняют казалось бы, очень простую операцию увеличения на единицу значения общей переменной:

count += 1;

Эта операция, представляющаяся неделимой, на самом деле выполняется как минимум тремя действиями.

1. Чтение из памяти в регистр процессора значения переменной *count*.
2. Увеличение этого значения на 1.
3. Запись результата в память.

Пусть переменная *count* имеет значение 0. Если процесс P_1 выполнил первое действие и был прерван процессом P_2 (в момент прерывания все регистры процессора для первого процесса сохраняются), который выполнил все эти три действия и записал в память значение 1, то после возвращения в процесс P_1 он также вычислит и запишет в переменную *count* значение 1. Однако два процесса добавляли по единице и переменная *count* должна иметь значение 2. Эта ситуация и называется «гонками за данными». В итоге параллельная программа, скорее всего, будет выдавать совсем не те результаты, которые выдала бы последовательная программа. Очевидно, что программа, в которой возможны такие ситуации, может вести себя по-разному в зависимости от временных соотношений моментов доступа разных процессов к общим данным.

Имеются два способа борьбы с гонками за данными:

- использование локальных для процесса, а не разделяемых переменных. Однако полностью без разделяемых переменных в параллельных программах обойтись невозможно, если их процессы действительно выполняют вычисления, направленные на решение одной задачи;
- управление доступом к разделяемым переменным с помощью различных средств синхронизации таким образом, чтобы цепочка действий «чтение – модификация – запись» одного процесса не могла быть прервана другим процессом. Средства синхронизации могут быть реализованы с помощью семафоров, критических секций, мьютексов и других механизмов.

Блокировки

Блокировка (тупик) возникает, если поток ожидает выполнение условия, которое не может быть выполнено. Обычно возникновение тупиковой ситуации является следствием конкуренции потоков за ресурс, который удерживается одним из них. Условия возникновения тупика:

- доступ к ресурсу эксклюзивен (возможен только одним потоком);
- поток может удерживать ресурс, запрашивая другой;
- ни один из конкурирующих потоков не может освободить запрашиваемый ресурс.

Активной блокировкой называют ситуацию, когда поток не производит вычислений, но и не блокируется. Потоки пытаются преодолеть помеху, создаваемую другим потоком.

Независимо от используемой модели параллельных вычислений выделяют следующие основные этапы разработки параллельного алгоритма.

1. *Декомпозиция.* На этом этапе выполняются анализ задачи и оценка возможности распараллеливания. Задача и связанные с ней данные разделяются на более мелкие части – подзадачи (процессы или потоки) и фрагменты структур данных. Особенности архитектуры конкретной вычислительной системы на данном этапе могут не учитываться.

2. *Проектирование взаимодействий между процессами.* Определяются коммуникации (источник, приемник, данные), необходимые для рассылки исходных значений, промежуточных результатов выполнения процессов, а также коммуникации, необходимые для управления их работой. Выбираются методы осуществления коммуникаций: точка-точка, широковещательная рассылка, сбор из всех в один процесс, ….

3. *Укрупнение.* Подзадачи объединяются в более крупные блоки, если это позволяет повысить эффективность алгоритма и снизить трудоемкость разработки.

4. *Планирование вычислений.* Распределение процессов высилительными узлами. Основной критерий выбора способа размещения подзадач – эффективное использование процессоров с минимальными затратами времени на обмены данными.

Опишем технологию выполнения этих этапов подробнее.

Декомпозиция может осуществляться либо по данным, либо функционально. На этом этапе следует стремиться к максимально возможной степени параллелизма.

Декомпозиция по данным выполняется так. Вначале сегментируются данные путем разбиения на фрагменты приблизительно одинакового размера. С фрагментами данных связываются операции их обработки, из которых и формируются процессы. На основании этого впоследствии будут определены необходимые пересылки данных. Пересечение частей, на которые разбивается задача, должно быть сведено к минимуму, что позволяет избежать дублирования вычислений. Схема разбиения в дальнейшем может уточняться. Если необходимо уменьшение числа обменов, то допускается увеличение степени перекрытия вычислений, осуществляемых разными процессами.

Сначала анализируются структуры данных наибольшего размера либо те, обращение к которым происходит чаще всего. На разных стадиях расчета могут использоваться разные структуры данных, поэтому могут использоваться как статические, так и динамические схемы декомпозиции этих структур. Обычно для декомпозиции по данным применяется один из видов дихотомии.

Рекурсивная дихотомия

Используется для разбиения области на подобласти, которым соответствует примерно одинаковая трудоемкость вычислений, а коммуникации сведены к минимуму. Область сначала разбивается на две части вдоль одного измерения. Разбиение повторяется рекурсивно в каждой новой подобласти столько раз, сколько потребуется для получения необходимого числа процессов.

Рекурсивная координатная дихотомия

Может применяться к нерегулярным сеткам. Деление выполняется на каждом шаге вдоль того измерения, по которому сетка имеет наибольшую протяженность.

Функциональная декомпозиция

Вначале сегментируется вычислительный алгоритм, а затем уже под эту схему подгоняется схема декомпозиции данных. Метод функциональной декомпозиции может оказаться полезным в ситуации, где нет структур данных, которые, очевидно, могли бы быть распараллелены. Эффективность декомпозиции обеспечивается выполнением следующих рекомендаций:

- количество процессов после декомпозиции должно примерно на порядок превосходить количество процессоров;
- следует избегать лишних вычислений и пересылок данных;
- процессы должны быть примерно одинакового размера;
- в идеале сегментация должна быть такой, чтобы с увеличением объема задачи количество процессов также возрастало (при сохранении постоянным размера одного процесса).

Размер процесса определяется зернистостью (гранулярностью). Мерой зернистости является количество операций в одном блоке. Выделяют три степени зернистости.

1. Мелкозернистый параллелизм: на уровне команд (единицы машинных команд на блок, количество параллельно выполняемых процессов может быть очень велико, но если каждая пара процессов нуждается во взаимодействии друг с другом, то потери времени на организацию коммуникаций могут превысить выигрыш от распараллеливания).

2. Крупнозернистый параллелизм: на уровне процедур или итераций внешнего цикла в гнезде циклов. Размер блока может составлять тысячи и десятки тысяч машинных команд. Требования к коммуникациям в этом случае обычно меньше, чем в случае параллелизма на уровне команд.

Проектирование коммуникаций. Можно выделить следующие основные типы:

- локальные – каждый процесс взаимодействует (передает или принимает данные) с небольшим набором других процессов;
- глобальные – каждый процесс взаимодействует со всеми остальными процессами параллельной программы;
- структурированные – коммуникации образуют регулярную структуру (например, с топологией решетки);
- неструктурированные – процессы связаны согласно некоторому произвольному графу;
- статические – схема организации коммуникаций не изменяется с течением времени выполнения процессов;

- динамические – схема коммуникаций изменяется в процессе выполнения программы;
- синхронные – отправители и получатели данных координируют моменты возобновления вычислений после выполнения обменов;
- асинхронные – вычисления и обмен данными не координируется.

Рекомендации по проектированию коммуникаций:

- количество коммуникаций для разных процессов должно быть примерно одинаковым, иначе приложение становится плохо масштабируемым;
- там, где это возможно, следует использовать локальные коммуникации;
- коммуникации должны быть, по возможности, параллельными.

Укрупнение. На этапе укрупнения (агломерации) учитывается архитектура и технические характеристики вычислительной системы и алгоритмы процессов, полученные на первых двух этапах, объединяются таким образом, чтобы их число соответствовало числу вычислительных узлов. При выполнении агломерации следует соблюдать следующие требования:

- должны быть снижены накладные расходы на коммуникации;
- если при укрупнении приходится дублировать вычисления или данные, это не должно приводить к потере производительности и масштабируемости программы;
- укрупненные процессы должны иметь примерно одинаковую трудоемкость;
- должна быть сохранена масштабируемость;
- должна быть сохранена возможность параллельного выполнения;
- должна быть снижена стоимость трудоемкости разработки.

Планирование вычислений. Этот этап обычно выполняется операционной системой либо специальными компонентами программного обеспечения.

Результаты выполнения этих этапов далее используются при кодировании параллельной программы с использованием одной из существующих систем поддержки параллельных вычислений или технологий параллельного программирования. Таких систем или технологий существует очень много.

В настоящем пособии опишем три из них: OpenMP, CUDA и MPI.

3. Технология параллельного программирования систем с общей памятью OpenMP

Технология OpenMP [10, 11] в настоящее время является одним из наиболее популярных средств параллельного программирования для компьютеров с общей памятью, базирующейся на традиционных последовательных языках программирования и использовании специальных комментариев. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Технология OpenMP нацелена на то, чтобы пользователь имел один и тот же вариант программы как для параллельного, так и для последовательного режима выполнения.

Параллелизм в OpenMP реализуется с помощью многопоточности. При запуске программы создается единственный «главный» (master) поток, который затем создает набор «подчиненных» (slave) потоков. Далее вычисления распределяются между всеми потоками. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (ядрами), причём количество процессоров/ядер не обязательно должно быть больше или равно количеству потоков. Другими словами, потоки параллельной программы могут обычным образом конкурировать между собой за время процессора/ядра.

Важным достоинством технологии OpenMP является возможность реализации так называемого инкрементального программирования, когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков. Таким образом, распараллеленные участки постепенно охватывают всё большую часть программы. Этот подход значительно облегчает процесс адаптации последовательных программ к параллельным компьютерам, а также отладку и оптимизацию программ.

3.1. Модель параллельной программы OpenMP

Модель исполнения параллельной программы, подготовленной с помощью технологии OpenMP, можно сформулировать следующим образом.

– Программа содержит набор последовательных и параллельных областей (или секций или регионов).

– В начальный момент времени создается главный поток, выполняющий впоследствии все последовательные области программы.

– При входе в параллельную область главным потоком выполняется операция fork, порождающая совокупность подчиненных потоков. Каждый поток имеет свой уникальный числовой идентификатор (главному потоку соответствует 0). При распараллеливании циклов все параллельные потоки исполняют один и тот же код, но с разными данными. В общем случае потоки могут исполнять различные фрагменты кода.

– При выходе из параллельной области всеми потоками выполняется операция *join.* Завершается выполнение всех потоков, кроме главного.

На рис. 3.1 проиллюстрировано создание и исполнение двух параллельных областей. В первой области все потоки приходят к моменту выхода из нее практически одновременно, во второй – в разные моменты времени (как правило, это более соответствует реальной картине).

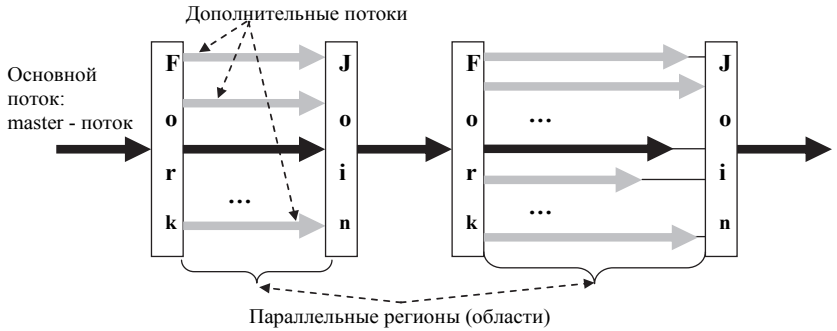


Рис. 3.1. Создание двух параллельных регионов

Обычно допускается возможность образования вложенных параллельных областей, когда поток, созданный как дополнительный, становится master-потоком для новой параллельной области, как показано на рис. 3.2.

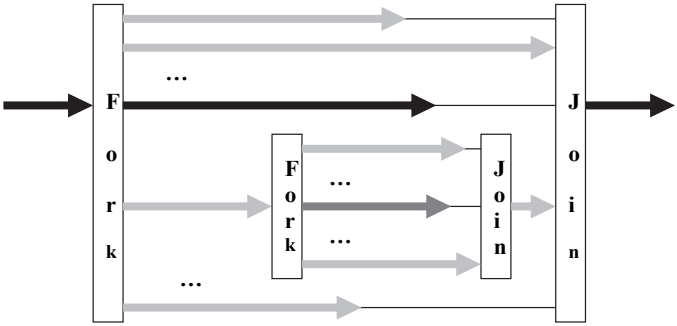


Рис. 3.2. Создание вложенных параллельных регионов

Под OpenMP понимается совокупность следующих компонентов.

– *Директивы компилятора* – используются для создания потоков, распределения работы между потоками и их синхронизации. Директивы включаются программистом в исходный текст программы.

– *Подпрограммы библиотеки времени выполнения* – используются для установки и определения атрибутов потоков. Вызовы этих подпрограмм включаются программистом в исходный текст.

– *Переменные окружения* – используются для управления поведением параллельной программы. Переменные окружения задаются для среды выполнения параллельной программы соответствующими средствами операционной системы.

Использование директив компилятора и подпрограмм библиотеки времени выполнения подчиняется правилам, которые различаются для разных языков программирования. Совокупность таких правил для одного языка программирования называется *привязкой к языку*. Технология OpenMP создана и поддерживается для языков C, C++ и Fortran. В данном пособии предполагается использование языка C/C++ в среде разработки Microsoft Visual Studio (начиная с версии 2005). Для того чтобы ее компилятор нужным образом реагировал на директивы OpenMP и строил параллельную программу, в командную строку его запуска должна быть включена опция /openmp (например, путем включения флажка «OpenMP support» в свойствах проекта на закладке Configuration properties\C/C++\Language).

Чтобы в программе на языке C/C++ стали доступны возможности технологии OpenMP, в нее нужно включить заголовочный файл omp.h:

```
#include <omp.h>
```

Если эту программу транслировать компилятором, не поддерживающим технологию OpenMP, то она будет построена в обычном последовательном (однопоточном) варианте, поскольку все директивы, задающие распараллеливание, оформляются в виде так называемых прагм (рекомендаций) и просто игнорируются такими компиляторами.

В программах на языке C/C++ все прагмы, имена функций и переменных окружения OpenMP начинаются со строки «omp». Формат директивы:

#pragma omp директива [опция_1[, опция_2, ...]]

Каждая директива вместе со всеми ее опциями обязательно должна занимать ровно одну строку текста. Действие некоторых директив распространяется только на один оператор (или блок операторов, заключенных в фигурные скобки), непосредственно следующий за директивой в тексте программы. Для таких директив будет указан

<структурный блок кода>.

3.2. Перечень директив OpenMP

3.2.1. Директива задания параллельно выполняемой секции:

#pragma omp parallel [опция_1[, опция_2, ...]]
<структурный блок кода>

С помощью опций этой директивы можно указать:

– требуемое количество потоков *n* (*num_threads(n)*);

– условие, при котором параллельная область действительно создается (*if(условие)*);

– список общих переменных для всех потоков данной секции (*shared(список переменных)*);

– список переменных, которые будут локальными в каждом потоке секции (*private(список переменных)*), причем их начальные значения не будут определены;

– список переменных, которые будут локальными в каждом потоке секции (*firstprivate(список переменных)*), причем в качестве их начальных значений будут установлены значения одноименных переменных из главного потока;

– способ назначения класса памяти *default(shared|none)* всем переменным потоков, которым класс не назначен явно с помощью опции *shared* (слово *none* означает, что класс памяти всех локальных пере-

менных должен быть задан явно); в реализациях для языка Fortran могут назначаться классы *private* и *firstprivate*;

– список переменных, объявленных директивой *threadprivate* (см. ниже), значения которых сохраняются в момент выхода из одной параллельной области и восстанавливаются в момент входа в другую;

– оператор сведения и список общих переменных *reduction(оператор : список переменных)*; для каждой указанной в списке переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора (для аддитивных операций ноль или его аналоги, для мультипликативных операций единица или её аналоги); над всеми локальными копиями каждой переменной после завершения параллельной секции будет выполнен заданный оператор сведения, результаты будут занесены в одноименные общие переменные; в качестве оператора можно указывать: +, –, *, &, |, ^, &&, ||.

3.2.2. Директива определения цикла, итерации которого нужно распределить между параллельно выполняемыми потоками:

#pragma omp for [опция_1[, опция_2, ...]]
<структурный блок кода>

С использованием опций директивы *for* можно указать:

– список переменных, которые будут локальными в каждом потоке секции (*private(список переменных)*), причем их начальные значения не будут определены;

– список переменных, которые будут локальными в каждом потоке секции (*firstprivate(список переменных)*), причем в качестве их начальных значений будут установлены значения одноименных переменных из главного потока;

– список переменных главного потока (*lastprivate(список переменных)*), которым будут присвоены значения, полученные при выполнении последней итерации цикла;

– оператор сведения и список общих переменных *reduction(оператор : список переменных)*; для каждой указанной в списке переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – ноль или его аналоги, для мультипликативных операций – единица или её аналоги); над всеми локальными копиями каждой переменной после завершения параллельной секции будет выполнен заданный оператор сведения, результаты будут занесены в одноимен-

ные общие переменные; в качестве оператора можно указывать: +, –, *, &, |, ^, &&, ||;

– способ распределения итераций цикла между потоками параллельной секции (*schedule(type[, chunk])*); параметр *chunk* этой опции определяет количество итераций на один поток (по умолчанию 1), параметр *type* указывает тип распределения и может иметь значения: *static* – статический, т. е. при компиляции, *dynamic* – динамический, т. е. при выполнении, *guided* – динамический с уменьшением количества итераций на поток от начального, определяемого автоматически, до заданного параметром *chunk*, *auto* – способ распределения выбирается компилятором или исполняющей системой, *runtime* – способ распределения задается специальной переменной окружения операционной системы;

– возможность (опция *ordered*) появления в теле цикла директивы *ordered*, которая требует исполнения охваченного ею блока операторов точно в той же последовательности, какая реализуется в последовательной версии данного цикла;

– отмену (*nowait*) неявной барьерной синхронизации потоков, достигших конца выполнения своей части итераций цикла (при отсутствии этой опции участок программы после цикла будет выполняться только тогда, когда все потоки выполнят все свои итерации);

– глубину *n* вложенных друг в друга циклов (*collapse(n)*), пространство итераций которых подлежит распределению между параллельными потоками (доступно только в реализации 2.5 технологии OpenMP);

Директивы *parallel* и *for* можно объединять в одну директиву, в которой можно указывать опции как директивы *parallel*, так и директивы *for*:

#pragma omp parallel for [опция_1[, опция_2, ...]]

3.2.3. Директива, указывающая на необходимость исполнения охватываемого ею участка кода (части тела цикла) в той последовательности, в которой он выполняется в чисто последовательном режиме

#pragma omp ordered
<структурный блок кода>

3.2.4. Директива задания области нециклического параллелизма (участки структурного блока кода, которые могут выполняться параллельно, выделяются с помощью описываемой далее директивы *section*):

#pragma omp sections [опция_1[, опция_2, ...]]
<структурный блок кода>

В качестве опций этой директивы можно указывать *private, first-private, lastprivate, reduction* и *nowait*, имеющие в точности такой же синтаксис и тот же смысл, что и у директивы *for.*

3.2.5. Директива определения участка нециклического кода для одного потока:

#pragma omp section

С помощью этой директивы внутри участка кода, охваченного директивой *sections*, выделяются отдельные фрагменты для исполнения параллельными потоками. Перед самым первым фрагментом участка нециклического кода директиву *section* можно не указывать.

3.2.6. Директива объявления списка локальных переменных потоков:

#pragma omp threadprivate(список переменных)

Эта директива позволяет сделать локальные копии для статических переменных языка C/C++ (и COMMON-блоков языка Фортран), которые по умолчанию являются общими.

3.2.7. Директива создания отдельной независимой задачи (начиная с версии 2.5 OpenMP):

#pragma omp task [опция_1[, опция_2, ...]]
<структурный блок кода>

Текущий поток создает в качестве задачи ассоциированный с директивой блок операторов. Эта задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией OpenMP.

Возможные опции:

– *if*, *default*, *private*, *firstprivate* и *shared* – имеют такой же синтаксис и смысл, как и в предыдущих директивах;

– *untied* – означает, что в случае откладывания задача может быть продолжена любым потоком из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившим её потоком;

3.2.8. Директива ожидания потоком завершения всех независимых задач, запущенных именно из данного потока:

#pragma omp taskwait

3.2.9. Директива, требующая исполнения охваченного ею участка кода в точности одним (любым) потоком:

#pragma omp single [опция_1[, опция_2, ...]]
<структурный блок кода>

Опции директивы позволяют указать:

списки переменных *private* и *firstprivate*, имеющие такой же синтаксис и семантику, как в ранее описанных директивах;

список переменных (*copyprivate(список переменных)*), значения которых после выполнения структурного блока, заданного директивой *single*, будут занесены во все одноименные локальные переменные (*private* и *firstprivate*), заданные для охватывающей параллельной секции; эта опция не может использоваться совместно с опцией *nowait*; переменные списка не должны быть перечислены в опциях *private* и *firstprivate* данной директивы *single*;

отмену (*nowait*) неявной барьерной синхронизации потоков, достигших точки, следующей за блоком операторов, охваченным директивой *single*; при отсутствии этой опции такая синхронизация выполняется.

Директива, требующая исполнения охваченного ею участка кода главным потоком программы:

#pragma omp master
<структурный блок кода>

Этот структурный блок будет выполнен только главным потоком программы. Остальные потоки просто пропускают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает.

Применимость всех опций в различных директивах OpenMP приведена в таблице.

3.2.11. Директива явной барьерной синхронизации:

#pragma omp barrier

Потоки, выполняющие текущую параллельную секцию, дойдя до этой директивы, останавливаются и ждут, пока все потоки не дойдут

до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые потоки завершили все порождённые ими задачи (директивы *task* и *taskwait*).

Применимость опций в директивах

Опции	Директивы					
	parallel	for	sections	single	parallel for	parallel sections
if	x				x	x
private	x	x	x	x	x	x
firstprivate	x	x	x	x	x	x
lastprivate		x	x		x	x
copyprivate				x		
copyin	x				x	x
shared	x				x	x
default	x				x	x
reduction	x	x	x		x	x
schedule		x			x	
ordered		x			x	
nowait		x	x	x		
collapse		x			x	

3.2.12. Директива, объявляющая критическую секцию – участок параллельной области программы, который единовременно может выполняться не более чем одним потоком:

#pragma omp critical [(<имя_критической_секции>)]
<структурный блок кода>

Если критическая секция уже выполняется каким-либо потоком, то все остальные, выполнившие эту директиву для секции с данным именем, будут заблокированы, пока вошедший в секцию поток не закончит выполнение этого блока кода. Как только это произойдет, один из заблокированных потоков войдет в критическую секцию. Если на входе в критическую секцию стояло несколько потоков, то случайным образом выбирается один из них, а остальные заблокированные продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и то-

же имя, рассматриваются как одна секция, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

3.2.13. Директива блокировки доступа к общей переменной из левой части оператора присваивания на время выполнения всех действий с этой переменной в данном операторе:

#pragma omp atomic
<оператор присваивания>

Атомарной (блокирующей выполнение остальных потоков) является только работа с переменной из левой части оператора присваивания, при этом вычисления в его правой части, использующие другие переменные, не обязаны быть атомарными.

3.2.13. Директива актуализации значений переменных потока:

#pragma omp flush [(список переменных)]

Значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущего потока, заносятся в основную память; следовательно, все изменения переменных, сделанные потоком во время работы, станут видимы остальным потокам; если какая-то информация хранится в буферах вывода, то буферы будут сброшены на внешние носители и т. п. Эти действия производятся только с данными выполнившего директиву потока, а данные, изменявшиеся другими потоками, не затрагиваются. До полного завершения этих действий никакие другие операции с участвующими в директиве *flush* переменными не могут выполняться. Поэтому выполнение директивы *flush* без списка переменных может повлечь значительные накладные расходы. Если в данный момент нужна гарантия согласованного представления не всех, а лишь некоторых переменных, то именно их следует явно перечислить в директиве списком.

Переменные окружения, значения которых могут использоваться в параллельной программе:

OMP_NUM_THREADS – количество потоков в новой параллельной области.

OMP_NESTED – возможность создания вложенных параллельных областей.

OMP_MAX_ACTIVE_LEVELS – максимальная глубина вложенности параллельных областей.

OMP_DYNAMIC – возможность динамического определения количества потоков (имеет больший приоритет, чем OMP_NUM_THREADS).

OMP_THREAD_LIMIT – максимальное количество потоков программы.

OMP_SCHEDULE – способ распределения итераций цикла для значения *runtime* опции *schedule* директивы *for*.

OMP_STACKSIZE – размер стека каждого потока.

OMP_WAIT_POLICY – выделять или нет кванты процессорного времени ждущим потокам.

3.3. Библиотека функций OpenMP

double omp_get_wtime(void); – возвращает текущее время.

double omp_get_wtick(void); – возвращает размер тика таймера.

int omp_get_thread_num(void); – возвращает номер потока.

int omp_get_max_threads(void); – возвращает максимально возможное количество потоков для следующей параллельной области.

int omp_get_thread_limit(void); – возвращает значение OMP_THREAD_LIMIT.

void omp_set_num_threads(int num); – устанавливает новое значение переменной OMP_NUM_THREADS.

int omp_get_num_procs(void); – возвращает количество процессоров/ядер.

int omp_get_dynamic(void); – возвращает значение OMP_DYNAMIC.

void omp_set_dynamic(int num); – устанавливает новое значение переменной OMP_DYNAMIC.

int omp_get_nested(void); – возвращает значение OMP_ NESTED.

void omp_set_nested(int nested); – устанавливает новое значение переменной OMP_ NESTED.

int omp_in_parallel(void); – возвращает 0, если функция вызвана из последовательной области, и 1, если из параллельной.

int omp_get_max_active_levels(void); – возвращает значение переменной OMP_MAX_ACTIVE_LEVELS

void omp_set_max_active_levels(int max);); – устанавливает новое значение OMP_MAX_ACTIVE_LEVELS

int omp_get_level(void); – возвращает глубину вложенности параллельных областей.

int omp_get_ancestor_thread_num(int level); – возвращает номер потока, породившего текущую параллельную область.

int omp_get_team_size(int level); – возвращает для заданного параметром level уровня вложенности параллельных областей количество потоков, порождённых одним родительским потоком.

int omp_get_active_level(void); – возвращает количество вложенных параллельных областей, обрабатываемых более чем одним потоком.

Следующая группа функций используется для синхронизации параллельно выполняющихся потоков с помощью так называемых замков (lock).

*void omp_init_lock(omp_lock_t *lock);* – создать (инициализировать) простой замок.

*void omp_init_nest_lock(omp_nest_lock_t *lock);* – создать замок с множественными захватами.

*void omp_destroy_lock(omp_lock_t *lock);* – уничтожить простой замок (перевести в неинициализированное состояние).

*void omp_destroy_nest_lock(omp_nest_lock_t *lock);* – уничтожить множественный замок (перевести в неинициализированное состояние).

*void omp_set_lock(omp_lock_t *lock);* – захватить простой замок (если замок уже захвачен другим потоком, то данный поток переводится в ждущее состояние).

*void omp_set_nest_lock(omp_nest_lock_t *lock);* – захватить множественный замок (если замок уже захвачен другим потоком, то данный поток переводится в ждущее состояние; если замок захвачен этим же потоком, то увеличивается счетчик захватов).

*void omp_unset_lock(omp_lock_t *lock);* – освободить простой замок (если есть потоки, ждущие его освобождения, то один из них, выбираемый случайным образом, захватывает этот замок и переходит в состояние выполнения).

*void omp_unset_nest_lock(omp_lock_t *lock);* – уменьшить на 1 количество захватов множественного замка (если количество захватов стало равно нулю и есть потоки, ждущие его освобождения, то один из них, выбираемый случайным образом, захватывает этот замок и переходит в состояние выполнения).

*int omp_test_lock(omp_lock_t *lock);* – попытаться захватить простой замок (если попытка не удалась, т. е. замок уже захвачен другим потоком, то возвращается 0, иначе возвращается 1).

*int omp_test_nest_lock(omp_lock_t *lock);* – попытаться захватить множественный замок (если попытка не удалась, т. е. замок уже захвачен другим потоком, то возвращается 0, иначе возвращается новое значение счетчика захватов).

Контрольные вопросы

1. Перечислите составные части технологии OpenMP.
2. С помощью какой директивы (директив) создаются новые параллельные области программы?
3. Что такое критическая секция программы?
4. Каким образом можно установить нужное количество потоков для создания очередной параллельной области?
5. Как обеспечить выполнение фрагмента параллельной области только главным потоком?
6. Какая опция директивы OpenMP *for* используется для указания способа распределения итераций цикла между потоками параллельной области?
7. Что такое *deadlock*? Каким правилам нужно следовать, чтобы избежать возможности попадания параллельной программы в *deadlock*?
8. Что такое сведение данных? Какие опции и в каких директивах используются для выполнения сведения?
9. Что делает директива OpenMP *threadprivate*?
10. Как обеспечить выполнение фрагмента параллельной области потоком с максимальным номером в данной параллельной области?
11. Для чего используется опция *firstprivate*? Чем она отличается от опций *private* и *lastprivate*?
12. Что такое вложенная параллельная область программы? В каких случаях ее нельзя создать?
13. Что такое неявная барьерная синхронизация? С помощью каких средств ее можно отменить?
14. Для чего используются директивы OpenMP *sections* и *section*? Что делает каждая из этих директив?
15. Перечислите все средства синхронизации потоков в OpenMP.
16. Перечислите возможные способы распределения итераций цикла между потоками.
17. Что делает директива OpenMP *atomic*?
18. В чем состоит различие между общими и локальными переменными потока?
19. С помощью каких средств можно ограничить глубину вложенности параллельных областей программы?
20. От чего зависит равномерность загрузки процессоров/ядер системы с общей памятью?
21. Каким образом функция, вызываемая из параллельной программы, может выяснить, в последовательной или параллельной области она выполняется?

22. Как обеспечить выполнение фрагмента параллельной области в точности одним потоком?

23. Какое значение будет иметь переменная *count* в результате выполнения фрагмента параллельной программы:

```
int count = 0;
#pragma omp parallel for
for(int i = 0; i<10; i++){
 count++;
}
```

24. Могут ли два потока одновременно захватить один множественный замок?

25. Для чего может использоваться опция *reduction* в некоторых директивах OpenMP?

26. Каково назначение директивы OpenMP *single*?

27. Каким ограничениям должен удовлетворять цикл, для того чтобы к нему можно было применить директиву OpenMP *for*?

28. Чем отличается простой замок от множественного?

29. Перечислите известные вам механизмы синхронизации потоков в OpenMP.

30. Перечислите способы распределения итераций цикла между потоками параллельной области.

4. CUDA – неграфические вычисления на графическом процессоре

CUDA (Compute Unified Device Architecture) [12–14] представляет собой совокупность аппаратного (графический процессор или GPU) и программного обеспечения, предоставляющего возможность подготовки и исполнения программ с очень высокой степенью параллелизма.

4.1. Архитектура CUDA

GPU есть специализированное вычислительное устройство, которое является сопроцессором к основному процессору компьютера (CPU), обладает собственной памятью и возможностью параллельного выполнения огромного количества (тысячи и десятки тысяч) отдельных нитей (потоков управления) обработки данных согласно модели ОКМД. Логическая структура связанных двухъядерного центрального и графического процессоров показана на рис. 4.1.

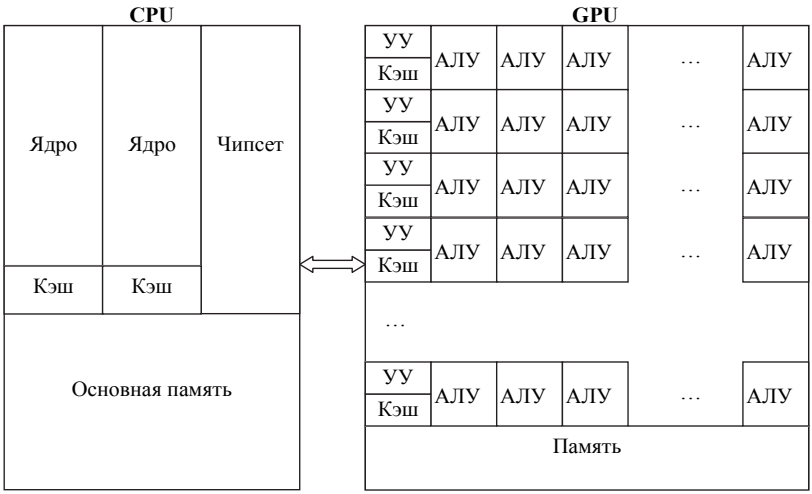


Рис. 4.1. Двухъядерный CPU и подключенный к нему GPU

Технология OpenMP, рассмотренная в разделе 3, и технология CUDA вполне совместимы и могут использоваться в одной параллельной программе для достижения максимально возможного ускорения.

Между потоками, выполняемыми на CPU, и потоками, выполняемыми графическим процессором, есть принципиальные различия:

- нити, выполняемые на GPU, обладают крайне низкой «стоимостью» – их создание и управление ими требует минимальных ресурсов (в отличие от потоков CPU);
- для эффективной утилизации возможностей GPU нужно использовать многие тысячи отдельных нитей (для CPU обычно бывает трудно организовывать более чем 10…20 потоков).

Приложения, использующие возможности CUDA для параллельной обработки данных, взаимодействуют с GPU через программные интерфейсы (рис. 4.2), называемые CUDA-runtime или CUDA-driver (обычно интерфейсы CUDA-runtime и CUDA-driver одновременно не используются, хотя это в принципе возможно).

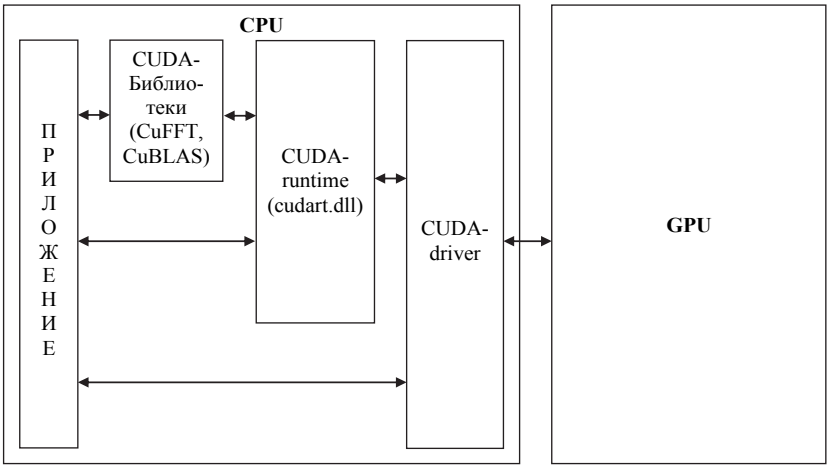


Рис. 4.2. Состав программного обеспечения CUDA

Программы для CUDA пишутся на «расширенном» языке C, при этом их параллельная часть (так называемые «ядра») выполняется на GPU, а обычная последовательная часть – на CPU. Компоненты архитектуры CUDA, принимающие участие в подготовке и исполнении приложения, автоматически осуществляют разделение частей и управление их запуском.

С точки зрения разработчика программы для CUDA, графический процессор представляет собой одно устройство управления и огромное количество арифметико-логических устройств (АЛУ), каждое из которых обладает собственной регистровой памятью (распределением регистров между АЛУ управляет программное обеспечение CUDA) и имеет доступ к нескольким уровням памяти различного объема и быстродействия. Все арифметико-логические устройства в любой данный момент времени исполняют одну команду, выбранную и декодированную устройством управления. Реально устройств управления несколько (их количество зависит от модели графического процессора). Совокупность из одного устройства управления связанных с ним арифметико-логических устройств, регистров, быстрой разделяемой памяти, доступной из каждого АЛУ, и более медленной локальной памяти каждого АЛУ, образуют так называемый мультипроцессор. Все мультипроцессоры имеют доступ к еще более медленным: кэшу текстур, кэшу констант и глобальной памяти устройства, через которую осуществляется взаимодействие с основной памятью компьютера. Внутренние элементы памяти мультипроцессора недоступны из CPU. Структура памяти графического процессора и возможность доступа к памяти разного уровня из программы центрального процессора показаны на рис. 4.3.

Отображение одной или нескольких параллельных областей программы на совокупность мультипроцессоров осуществляет программное обеспечение CUDA, которое является «прозрачным» для программиста. Ему нужно знать только о том, что с логической точки зрения параллельная область программы («ядро» в терминологии CUDA) представляет собой сетку (grid) блоков (block) потоков (thread). Блок потоков – это набор потоков, выполняемых одним мультипроцессором. Потоки одного блока могут взаимодействовать через разделяемую память и синхронизироваться между собой. Сетка и (независимо от нее) каждый из блоков представляют собой одно-, дву- или трехмерную структуру блоков и потоков соответственно. Количество размерностей и размеры каждой из них задает программист. Каждый поток после запуска имеет доступ к переменным (структурам), хранящим его собственные координаты внутри блока и координаты охватывающего блока внутри сетки.

Архитектура CUDA предусматривает широкий набор возможностей и поддерживается в видеокартах нескольких линеек (GEforce, Quadro, NVS, Tesla, ION). Разные модели графических процессоров, функционирующих в таких видеокартах, имеют каждая собственный

набор параметров (например, количество регистров, объем разделяемой памяти мультипроцессора) и поддерживают разные наборы возможностей, сгруппированные в версии архитектуры от 1.0 до 5.5 (по состоянию на 2014 год).

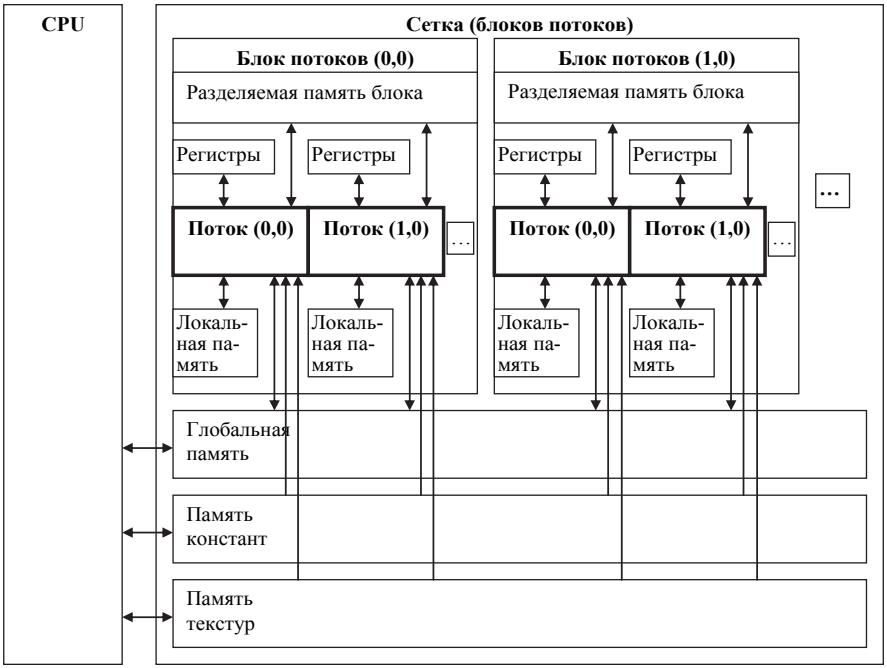


Рис.4.3. Структура памяти графического процессора

Для каждого устройства производитель указывает поддерживаемую им версию CUDA, по номеру версии можно определить значения параметров и доступные возможности. Соответствие поддерживаемых возможностей версии CUDA показано в табл. 4.1.

В табл. 4.2 приведено соответствие технических характеристик, на которые нужно ориентироваться при разработке параллельных программ, версии CUDA, реализованной в конкретной видеокарте. Для того чтобы узнать, какую версию поддерживает подключенная к компьютеру видеокарта, нужно воспользоваться функцией

cudaGetDeviceProperties(...);

Таблица 4.1

<table>
<tr><th rowspan="2">Поддерживаемые возможности
(не указанные здесь возможности
поддерживаются во всех версиях)</th><th colspan="7">Версия CUDA</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.5</th></tr>
<tr><td>Целочисленные атомарные функции, работающие над 32-разрядными словами в глобальной памяти</td><td rowspan="2">Нет</td><td colspan="6" rowspan="2">Да</td></tr>
<tr><td>Атомарные обменные операции над 32-битными числами с плавающей точкой в глобальной памяти</td></tr>
<tr><td>Целочисленные атомарные функции, работающие над 32-разрядными словами в разделяемой памяти</td><td colspan="3" rowspan="4">Нет</td><td colspan="4" rowspan="4">Да</td></tr>
<tr><td>Атомарные обменные операции над 32-битными числами с плавающей точкой в разделяемой памяти</td></tr>
<tr><td>Целочисленные атомарные функции, работающие над 64-разрядными словами в глобальной памяти</td></tr>
<tr><td>Warp-функции голосования</td></tr>
<tr><td>Операции с плавающей точкой двойной точности</td><td colspan="4" rowspan="8">Нет</td><td colspan="3" rowspan="8">Да</td></tr>
<tr><td>Целочисленные атомарные-функции, работающие над 64-разрядными словами в разделяемой памяти</td></tr>
<tr><td>Атомарные операции типа сложения над 32-битными числами с плавающей точкой в глобальной и разделяемой памяти</td></tr>
<tr><td>Функция _ballot()</td></tr>
<tr><td>Набор функций _threadfence_system()</td></tr>
<tr><td>Функции _syncthreads_count(), syncthreads_and(), _syncthreads_or()</td></tr>
<tr><td>Поверхностные функции</td></tr>
<tr><td>3-мерная сетка блоков потоков</td></tr>
<tr><td>Функции перемешивания warp’ов</td><td colspan="5">Нет</td><td colspan="2">Да</td></tr>
<tr><td>Сдвиги воронкой</td><td colspan="6" rowspan="2">Нет</td><td rowspan="2">Да</td></tr>
<tr><td>Динамический параллелизм</td></tr>
</table>

Таблица 4.2

Технические спецификации	Версия						
	1.0	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**
Максимальная размерность сетки блоков потоков	2				3		
Максимальные *x*-, *y*- или *z*- размерности сетки блоков	65535					2^{31}-1	
Максимальная размерность блока потоков	3						
Максимальные *x*- или *y*-размерности блока	512				1024		
Максимальная *z*-размерность блока	64						
Максимальное количество потоков в блоке	512				1024		
Размер warp'а (количество потоков)	32						
Максимальное количество резидентных блоков в мультипроцессоре	8					16	
Максимальное количество резидентных warp'ов в мультипроцессоре	24		32		48	64	
Максимальное количество резидентных потоков в мультипроцессоре	768		1024		1536	2048	
Количество 32-битных регистров в мультипроцессоре	8 K		16 K		32 K	64 K	
Максимальное количество 32-битных регистров на поток	128				63		255
Максимальный объем разделяемой памяти на мультипроцессор	16 KB				48 KB		
Количество банков разделяемой памяти	16				32		
Объем локальной памяти на поток	16 KB				512 KB		
Максимальное количество инструкций на ядро	2 миллиона				512 миллионов		

Формат ее аргументов и возвращаемых результатов можно найти в руководстве по программированию для CUDA [15].

4.2. Технология программирования CUDA

Рекомендуемая разработчиками архитектуры технология параллельного программирования состоит в следующем.

1. Спланировать разбиение обрабатываемых данных на фрагменты, целиком помещающиеся в разделяемую память графического процессора.

2. Обеспечить загрузку данных в глобальную память GPU.

3. Запустить ядро в программе каждого блока.

3.1. Выполнить перемещение нужных фрагментов данных из глобальной памяти в локальную память блока.

3.2. Выполнить требуемые вычисления.

3.3. Скопировать сформированные результаты обработки из разделяемой памяти в глобальную память устройства.

4. Перенести полученные результаты из памяти графического процессора в основную память компьютера, выполнить их окончательную обработку или вывести на внешний носитель.

Расширенная среда программирования для языка С включает в себя:

– расширения синтаксиса для написания кода для GPU, реализуемые дополнительным препроцессором nvcc;

– Run-time библиотеки:

- общую часть – встроенные векторные типы данных и набор функций, исполняемых и на CPU, и на GPU;
- CPU-компонент, для доступа и управления одним или несколькими GPU (драйвер);
- GPU-компонент, предоставляющий функции, специфические только для GPU.

Расширения синтаксиса языка включают в себя следующее.

1. Дополнительные виды функций:

	Исполняется на	Вызывается из
__host__ float HostFunc()	CPU	CPU
__global__ void KernelFunc()	GPU	CPU
__device__ float DeviceFunc()	GPU	GPU

Особенности дополнительных видов функций:

функция, объявленная как __global__, является функцией-ядром, она не может возвращать никакого значения (всегда void);

функция вида __device__ и выполняется на GPU и запускается из функций, исполняемых на GPU, поэтому нельзя получить указатель на такую функцию.

В функциях, исполняемых на GPU:

− недопустима рекурсия (однако начиная с версии 2.0, рекурсия поддерживается в функциях вида __device__);

− не может быть статических переменных внутри функций;

− количество аргументов не может быть переменным.

2. Дополнительные виды переменных:

	Тип памяти	Область видимости	Время жизни
__device__ __shared__ int	разделяемая	блок	блок
__device__ int	глобальная	сетка	ядро
__device__ __constant__ int	кэш констант	сетка	ядро

Ключевое слово *__device__* необязательно, если используется *__shared__* или *__constant__* . Локальные переменные без идентификатора вида хранятся в регистрах, за исключением больших структур или массивов, действительно располагающихся в локальной памяти. Тип памяти указателя адаптируется к типу присвоенного ему выражения.

3. Встроенные векторные типы объявляются так:

[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]

Буква *u* в этих объявлениях необязательна, является сокращением от слова *unsigned* и в случае ее указания означает, что все поля данного типа являются беззнаковыми целыми.

Доступ к полям этих типов осуществляется по именам: *x, y, z, w,* например:

```
uint4 param;
int y = param.y;
int abc = param.w;
```

Тип *dim3* является синонимом *uint3*. Обычно тип *dim3* используется для задания параметров сетки.

4. Ядро (собственно программа для GPU) и его запуск.

При запуске ядру должны быть переданы обязательные параметры конфигурации сетки. Для этого в программе для CPU описываются и создаются следующие функции и переменные:

__global__ void KernelFunc(...); – прототип функции, являющейся ядром.

dim3 DimGrid(100, 50); – переменная CPU, определяющая двумерность сетки блоков потоков и ее размеры 100 × 50, т. е. 5000 блоков.

dim3 DimBlock(4, 8, 8); – переменная CPU, определяющая трехмерность совокупности потоков в каждом блоке и размеры этой совокупности 4 × 8 × 8, т. е. 256 потоков на блок.

size_t SharedMemBytes = 64; – переменная CPU, определяющая размер разделяемой памяти в 64 байта, выделяемой каждому потоку.

После этих объявлений можно запустить ядро, для этого тоже реализован специальный синтаксис:

KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);

Опциональные *SharedMemBytes* байт:

– выделяются в дополнение к статически объявленным разделяемым переменным в ядре;

– отображаются на любую переменную вида: *extern __shared__ float DynamicSharedMem[]*.

Запуск ядра асинхронен, сразу после вызова функции управление немедленно возвращается к следующему оператору программы для CPU.

5. Встроенные переменные, создаваемые и инициализируемые автоматически при запуске ядра:

dim3 gridDim; – размеры сетки в блоках (*gridDim.w* не используется).

dim3 blockDim; – размеры одного блока в потоках.

dim3 blockIdx; – индекс блока внутри сетки.

dim3 threadIdx; – индекс потока внутри блока.

Эти переменные доступны только для кода параллельных ветвей ядра, исполняемого на GPU.

Общая часть Run-time библиотек содержит большое количество функций, необходимых (или опциональных) при написании параллельной программы. Таких функций слишком много, для того чтобы приводить их описание в настоящей работе, поэтому здесь приводится описание только тех функций, которые действительно необходимы. Все библиотечные функции можно разделить на несколько групп.

1. Управление потоками CPU и GPU, синхронизация.
2. Обработка ошибок.
3. Управление устройством.
4. Управление событиями.

5. Управление глобальной памятью.
6. Интерфейс с OpenGL и Direct3D версий 9, 10 и 11.
7. Интерфейс с видеоадаптером.
8. Управление текстурами и поверхностями.
9. Интерфейс с драйвером CUDA.

При разработке любой параллельной программы невозможно обойтись без использования функций управления глобальной памятью GPU. Таких функций более 40, из них наиболее употребительны следующие:

*cudaError_t cudaMalloc (void **devPtr, size_t size);*

выделяет блок глобальной памяти устройства размером *size* и заносит адрес этого блока в указатель *devPtr*. Возвращает либо код успешного завершения *cudaSuccess*, либо код ошибки, если блок памяти распределить не удалось.

*cudaError_t cudaFree (void *devPtr);*

возвращает ранее выделенный блок глобальной памяти в пул свободной памяти. Возвращает либо код успешного завершения, либо код ошибки (например, если этот блок ранее не выделялся).

*cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind);*

копирует блок памяти размером *count* байт, расположенный по адресу *src*, в память по адресу *dst*. Один из этих блоков размещается в основной памяти компьютера, второй – в глобальной памяти графического процессора, какой из них где – определяется значением аргумента *kind* (допустимы *cudaMemcpyHostToDevice* и *cudaMemcpyDeviceToHost*).

Контрольные вопросы

1. Что такое ядро в терминологии CUDA?
2. Как задается размерность и количество потоков ядра, выполняемого графическим процессором?
3. Перечислите виды и характеристики памяти, доступной из программы GPU.
4. Как указать требуемое размещение переменной в памяти GPU (регистровой, разделяемой, глобальной, памяти текстур, …)?
5. Перечислите виды и характеристики памяти, доступной из программы CPU при использовании архитектуры CUDA.

6. Перечислите встроенные векторные типы данных расширения языка C и объясните смысл их наименований.

7. Как в программе для CPU обеспечить синхронизацию с программой для GPU?

8. Если к одному CPU подключено несколько видеокарт NVidia архитектуры CUDA, то каким образом можно обеспечить их одновременную загрузку?

9. Перечислите виды и характеристики памяти GPU, доступной из программы основного процессора.

10. Какие ограничения наложены на функции, которые должны выполняться на GPU?

11. Что такое мультипроцессор в терминологии CUDA?

12. Какие ограничения наложены на функции, выполняемые в графическом процессоре?

13. Как определить версию и технические характеристики графического процессора NVIDIA?

14. Перечислите и охарактеризуйте группы библиотечных функций Run-time библиотеки CUDA.

15. Что такое сетка, блок, поток в терминологии CUDA?

16. Что такое тип данных dim3?

17. Какова максимальная размерность сетки в блоках?

18. Перечислите и охарактеризуйте функции компонентов программного обеспечения CUDA.

19. Какую модель параллелизма реализует архитектура CUDA?

20. Что делает препроцессор nvcc?

21. Можно ли получить указатель на функцию, выполняемую графическим процессором?

22. С помощью какого расширения синтаксиса запускается ядро CUDA?

23. Перечислите и охарактеризуйте дополнительные виды функций, существующие в расширении синтаксиса языка C для CUDA.

24. Какие виды памяти доступны из программ для GPU?

25. Как синхронизируются процессы в CPU и в GPU?

26. Можно ли одновременно использовать технологии OpenMP и CUDA?

27. Какие имена полей используются во встроенных векторных типах?

5. Интерфейс передачи сообщений MPI

Аббревиатура MPI расшифровывается так: Message passing interface (интерфейс передачи сообщений). Этот интерфейс в настоящее время в параллельном программировании для вычислительных систем с распределенной памятью является стандартом де-факто [16, 17], поддерживается Аргоннской национальной лабораторией в США и развивается форумом разработчиков MPIForum [18]. Известны и популярны две основные версии стандарта – 1.1 и 2.0, для которых существует довольно большое количество реализаций интерфейса MPI для различных платформ (Windows, Linux, …) как свободно распространяемых, так и коммерческих. В настоящем пособии рассматривается свободно распространяемая реализация MPICH, созданная Аргоннской национальной лабораторией и работающая на ОС Windows NT (и всех ее последующих вариантах) и на большинстве UNIX-систем. В этой реализации поддерживается вся функциональность версии 1.1 стандарта MPI и некоторые возможности версии 2.0.

Реализация интерфейса MPICH (буквы CH являются сокращением от слова chameleon, которым разработчики обозначили многоплатформенность) включает в себя следующие составные части:

– **Spmd** – демон (в терминах *NIX), службу (в терминах Windows). Постоянно работает на каждом узле сети, на котором возможен запуск/выполнение параллельных MPI-программ. Обслуживает запросы на запуск/останов этих программ и запросы от их ветвей на взаимодействие. Здесь будет использоваться термин «служба».

– **Библиотека функций MPI** – обеспечивает интерфейс между ветвью параллельной программы и службой spmd. Для использования функций MPI к программе нужно подключать заголовочные файлы (.h) и файлы собственно библиотеки (.a, .o, .so для *NIX, .lib для Windows).

– **Утилиты**, с помощью которых осуществляется запуск параллельных программ, настройка среды и собственно MPI. Для запуска используется утилита MPIRun (в Windows – MPIexec, с графической облочкой – WMPIexec). Для настройки MPI и окружающей среды используются утилиты MPIConfig и MPIRegister.

– **Библиотека MPE** – обеспечивает сбор, обработку и визуализацию результатов обработки статистики взаимодействий параллельных ветвей. Используется для отладки и оптимизации параллельных программ.

Для того чтобы исполнять параллельные программы, написанные в стандарте MPI, на компьютерах локальной сети, на каждом из них нужно:

– выполнить процедуру установки пакета (например, запустить файл mpich2-1.3.2p1-win-ia32.msi);

– обеспечить возможность регистрации пользователя с одним и тем же логином/паролем (некоторые операции, выполняемые службой, требуют аутентификации и должны выполняться от имени одного и того же пользователя на разных узлах сети);

– убедиться, что брандмауэр компьютера не блокирует службу spmd и программы запуска MPI-приложений MpiExec.exe и WMpiExec.exe, в противном случае – добавить эти программы в список исключений блокировки (в некоторых версиях ОС возможно потребуется обеспечить исключение блокировки брандмауэром всех запускаемых параллельных программ);

– убедиться, что на закладке «Дополнительно» брандмауэра выключена опция «Подключение по локальной сети»;

– создать на диске одного из компьютеров сети общий (share) ресурс с полными правами доступа для пользователя с логином из предыдущего пункта (ветви параллельной программы на каждом узле при запуске ищутся по одному и тому же пути, поэтому при отсутствии общего ресурса программу придется вручную копировать на каждый компьютер, обеспечивая тождество путей к ней во всей сети);

– запустить утилиту MPIRegister.exe и зарегистрировать пользователя с этим логином.

После этого следует запустить утилиту MpiConfig.exe и определить с ее помощью список компьютеров, доступных для исполнения параллельных программ. Каждую такую программу нужно скопировать на общий ресурс.

Для того чтобы с помощью среды MS Visual Studio готовить к запуску параллельные программы, нужно запустить MS Visual Studio. В пункте меню «Options/Tools/Projects and Solutions/VC++ Directories» выбрать в поле «Show directories for:» пункт «Include files», щелкнуть по иконке с каталогом, потом по иконке «...» и выбрать путь «…/Program

files/MPICH2/include». В пункте меню «Options/Tools/Projects and Solutions/VC++ Directories» выбрать в поле «Show directories for:» пункт «Library files», щелкнуть по иконке с каталогом, потом по иконке "..." и выбрать путь «…/Program files/MPICH2/lib». Щелкнуть по кнопке «Ok». Закрыть MS Visual Studio.

5.1. Основные сведения о стандартах MPI-1 и MPI-2

Отличительная особенность программ, написанных с использованием стандарта MPI-1, состоит в том, что в них допускается только статическое распараллеливание, т. е. количество параллельных процессов (ветвей) во время запуска и исполнения программы фиксировано.

Стандарт MPI-1 включает в себя следующие группы функций.

1. Инициализации, завершения, определения окружения.
2. Передача сообщений типа «точка-точка».
3. Коллективные операции взаимодействия.
4. Создание и использование производных типов данных.
5. Управление группами ветвей и коммуникаторами.
6. Управление виртуальными топологиями.

Стандарт MPI-2 помимо функциональности стандарта MPI-1 включает в себя следующие дополнительные группы операций.

1. Улучшенный внешний интерфейс для разработчика.
2. Динамическое порождение ветвей.
3. Удаленный доступ к памяти (односторонних взаимодействий).
4. Расширенные коллективные операции.
5. Параллельный ввод/вывод.

Основными понятиями, на которых базируется вся функциональность интерфейса MPI, являются понятия коммуникатора и группы параллельных ветвей. Любые взаимодействия между ветвями параллельной программы реализуются с обязательным указанием коммуникатора.

Коммуникатором называется указатель (дескриптор) на внутреннюю структуру библиотеки MPI, хранящую сведения о коллективе параллельных ветвей. В MPI понятия коллектива ветвей и группы ветвей различаются. Группа ветвей как объект может существовать самостоятельно, коллектив – это всегда коммуникатор. Имена коммуникаторов обычно начинаются с префикса MPI_COMM_.

Коммуникатор коллектива, который включает в себя все ветви приложения, создается автоматически при инициализации библиотеки и называется MPI_COMM_WORLD. В этот момент создаются также:

– коммуникатор MPI_COMM_SELF. В этот коллектив входит единственная ветвь, вызвавшая функцию инициализации;

– пустой коммуникатор MPI_COMM_NULL, не содержащий ни одной ветви. Использование этого коммуникатора в любой функции MPI приведет к аварийному завершению ветви, вызвавшей функцию, и аварийному завершению всего приложения.

В программе можно явно создавать сколько угодно новых коммуникаторов, указывая нужные коллективы параллельных ветвей (предварительно нужно создавать соответствующие группы ветвей MPI для указания их в функциях создания коммуникаторов). Любая ветвь может быть членом произвольного количества коммуникаторов.

Нужно помнить, что каждый коммуникатор, как и любой другой объект, создаваемый средствами библиотеки, расходует ресурсы, в частности, оперативную память. В настоящем пособии рассматриваются только интракоммуникаторы. Существуют интеркоммуникаторы, используемые для организации взаимодействия родительских групп ветвей и порождаемых ими потомков. Такие коммуникаторы не будут рассматриваться в этой работе.

Для приема/передачи сообщений важным является понятие буфера данных. Во многих функциях MPI должен быть указан буфер, содержащий передаваемые данные или предназначенный для принимаемых от других ветвей данных. Каждый буфер всегда определяется тремя параметрами:

– адрес (указатель на первый байт буфера);

– длина (размер буфера в единицах длины типа, заданного третьим параметром);

– тип (в MPI определен набор типов данных, указываемых с помощью символических имен; существует возможность создания собственных, или производных типов, указываемых с помощью имен переменных, получающих в качестве значений дескрипторы пользовательских типов).

Перечень базовых (предопределенных) типов данных MPI:

Тип данных MPI	Тип данных языка C	Тип данных языка C++	Тип данных языка Fortran
MPI_CHAR	signed char	signed char	Character(1)
MPI_WCHAR	wchar_t	wchar_t	Character(1)
MPI_SHORT	signed short int	signed short int	Integer * 2

Окончание таблицы

Тип данных MPI	Тип данных языка C	Тип данных языка C++	Тип данных языка Fortran
MPI_INT	signed int	signed int	Integer * 4
MPI_LONG	signed long int	signed long int	Integer * 4
MPI_UNSIGNED_CHAR	unsigned char	unsigned char	Character
MPI_UNSIGNED_SHORT	unsigned short int	unsigned short int	Integer * 2
MPI_UNSIGNED	unsigned int	unsigned int	Integer * 4
MPI_UNSIGNED_LONG	unsigned long int	unsigned long int	Integer * 4
MPI_BOOL (MPI_LOGICAL)	signed int	bool	Logical
MPI_FLOAT	float	float	Real * 4
MPI_DOUBLE	double	double	Double precision (Real * 8)
MPI_LONG_DOUBLE	long double	long double	Double precision (Real * 8)
MPI_COMPLEX	-	Complex<float>	Complex
MPI_DOUBLE_COMPLEX	-	Complex<double>	Double complex
MPI_LONG_DOUBLE_COMPLEX	-	Complex<long double>	-
MPI_BYTE (8-битный байт)	-	-	-
MPI_PACKED	-	-	-

Важную роль в MPI играет понятие тэга. Тэгом называется целочисленное значение, используемое для установления соответствия внутри потоков сообщений, передаваемых от одной ветви другой. С помощью тэгов решается такая проблема: если ветвь *A* асинхронно (без синхронизации) отправляет два (или более) сообщения одинакового размера, адресованных ветви *B*, то в общем случае нет гарантии, что сообщение, отправленное первым, будет первым и доставлено.

По самым разным причинам может оказаться, что второе сообщение будет доставлено раньше первого. Если оно будет обработано вет-

вью *B* как первое, то результаты работы программы в целом будут неверны. Указание разных значений тэгов для этих сообщений как на передающей, так и на приемной стороне позволяет избежать путаницы при доставке сообщений.

Рекомендуется тэги объявлять с помощью оператора препроцессора #define, выбирая семантически значимые имена для целочисленных констант, например:

```
#define FIRST_VECTOR 100
#define COLUMN 101
```

Для сообщений, обмен которыми выполняется внутри циклических участков программы, в качестве тэга рекомендуется использовать индекс соответствующего цикла.

Подавляющее большинство функций MPI возвращают целое значение, равное MPI_SUCCESS, если выполнение функции завершилось без ошибок, и код ошибки в противном случае. Коды ошибок зависят от реализации библиотеки и содержатся в справочных руководствах.

Далее рассматриваются основные группы функций и их краткое описание.

5.2. Функции инициализации, завершения, определения окружения

*int MPI_Init(int *argc, char **argv);* – до вызова этой функции не может быть вызвана никакая другая функция библиотеки MPI (за исключением функции *MPI_initialized*).

int MPI_Finalize(); – завершение: после вызова этой функции больше нельзя вызывать какие бы то ни было функции MPI, кроме *MPI_initialized* (однако в некоторых реализациях допускается повторная инициализация). Для нормального завершения параллельной программы эту функцию должны вызвать все ее ветви.

*int MPI_Comm_size(MPI_Comm comm, int *size);* – получение количества ветвей в коллективе коммуникатора *comm.*

*int MPI_Comm_rank(MPI_Comm comm, int *rank);* – получение индекса (номера) данной ветви в коммуникаторе *comm.*

*int MPI_Get_processor_name(char *name, int *resultlen);* – получение имени узла сети/кластера, на котором выполняется данная ветвь.

double MPI_Wtime(); – получение текущего времени, которое отсчитывается от момента запуска программы.

double MPI_Wtick(); – получение величины тика таймера узла в секундах.

*int MPI_Buffer_attach(void *buffer, int size);* – передача библиотеке MPI буфера памяти для некоторых операций обмена сообщениями.

*int MPI_Buffer_detach(void *buffer_addr, int size);* – извещение библиотеки MPI о том, что указанный буфер памяти использовать больше нельзя. Возврат из функции будет выполнен только тогда, когда все буферизованные операции передачи данных, связанные с этим буфером, будут завершены, т. е. когда этот буфер будет полностью освобожден.

5.3. Обмен сообщениями типа «точка–точка»

Эта группа функций обеспечивает передачу/прием сообщений между парами ветвей одного коммуникатора. Передача/прием могут быть блокирующими (выполнение ветви останавливается вплоть до момента завершения обмена данными) и неблокирующими (ветвь получает управление сразу после того, как библиотека обработала аргументы функции).

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);* – передача сообщения с блокировкой. Здесь и далее три аргумента *buf*, *count* и *datatype* полностью определяют буфер с данными. Аргументы *dest* и *tag* однозначно определяют индекс ветви-получателя сообщения в коммуникаторе *comm* и порядковый номер сообщения.

*int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);* – прием сообщения, отправляемого функцией *MPI_Send*, с блокировкой. Смысл аргументов функции точно такой же, как и у функции *MPI_Send* (аргумент *source* кроме очевидного номера ветви-отправителя в коммуникаторе *comm* может иметь значение MPI_ANY_SOURCE, т. е. отправитель – любая ветвь коммуникатора), однако здесь есть дополнительный аргумент *status*, являющийся указателем на структуру, поля которой позволяют получить дополнительные сведения о принятом сообщении (их перечень определен исходя из нужд функций без блокировки):

– int cancelled – 0, если прием завершился успешно, не 0 в противном случае;

– int count – размер буфера для принимаемых данных (не фактическое количество принятых данных);

– int MPI_ERROR – код ошибки, если прием завершился с ошибкой (может не совпадать с тем, что возвращает функция *MPI_Recv*);

– int MPI_SOURCE – идентификатор ветви-отправителя данных;

– int MPI_TAG – тэг принятого сообщения.

Если ни одна ветвь не отправит в данную ветвь сообщения с указанным тэгом, то ветвь-получатель никогда не получит управления, т. е. параллельная программа просто зависнет.

Операции *MPI_Send* и *MPI_Recv* являются основными для обменов типа «ветвь-ветвь». Однако их использование в реальных программах сопряжено с известными трудностями и в большинстве случаев приводит к невысокой производительности.

Поэтому в MPI дополнительно поддерживаются следующие разновидности операций передачи.

– Стандартная передача. Считается завершенной, как только сообщение отправлено, независимо от того, получено оно или нет. Передача сообщения может начинаться, даже если принимающая ветвь еще не вызвала функцию приема.

– Синхронная передача. Не завершается до тех пор, пока не будет завершена приемка сообщения.

– Буферизированная передача. Вызов функции завершается сразу же, потому что сообщение при этом копируется в системный буфер и там ожидает пересылки.

– Передача по готовности. Собственно передача начинается только в том случае, если начата приемка сообщения. В этот момент без ожидания окончания приема выполняется возврат в ветвь.

Кроме того, каждая из этих четырех разновидностей передачи может быть выполнена в блокирующей или неблокирующей форме (блокирующий прием / передача приостанавливает ветвь на время приема / передачи сообщения). Таким образом, всего существует восемь разновидностей операций передачи, включая уже описанную функцию *MPI_Send*. В MPI принято соглашение об именах процедур, позволяющее легко определять тип используемой операции по имени функции, которое строится по следующему правилу:

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

где

– *I* (Immediate) – обозначает неблокирующую операцию;

– *R* (Ready) – передача по готовности;

– *S* (Synchronous) – синхронный;

– *B* (Buffer)– буферизированный.

(только первая буква после подчеркивания является прописной, все остальные – строчные, как например *MPI_Ibsend*).

Аргументы всех блокирующих (без первой после символа подчеркивания буквы *I*) функций точно такие же, как у функции *MPI_Send*. Добавочным аргументом неблокирующих функций *MPI_I*send* является дескриптор так называемой квитанции, который в программе должен быть объявлен так:

MPI_Request request; (имя переменной может отличаться от имени *request*). Квитанции используются библиотекой MPI для синхронизации работы ветви и процессов доставки сообщения.

Кроме блокирующей функции приема сообщения *MPI_Recv* существует ее неблокирующий вариант *MPI_Irecv* с добавочным аргументом-квитанцией.

Запущенную ранее неблокирующую операцию приема/передачи в любой момент до ее завершения можно отменить, вызвав функцию

*int MPI_Cancel(MPI_Request *request);*

Статус принимаемого сообщения можно получить, не читая из MPI самого сообщения. Это делается либо с помощью блокирующей функции:

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

либо с помощью неблокирующей функции:

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

Проверить, завершилась ли одиночная неблокирующая операция, можно путем вызова функции

*int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);* – эта функция проверяет состояние операции, связанной с квитанцией *request* и формирует значение флага *false*, если операция еще не завершилась, и значение *true*, если операция завершилась (в этот момент становятся актуальными поля структуры *status*).

Дождаться завершения одной операции можно, вызвав функцию

*int MPI_Wait(MPI_Request *request, MPI_Status *status);* – вместо адреса структуры *status* может быть указано значение MPI_STATUS_IGNORE. Тогда определить, нормально ли завершилась операция, можно только по коду завершения, возвращаемому функцией *MPI_Test* или *MPI_Wait*. Эта функция возвращает управление в ветвь только тогда, когда завершается неблокирующая операция MPI, в вызове которой был указан данный *request*. В этот момент формируются / модифицируются поля структуры *status* (если она указана).

Если было запущено более одной неблокирующей операции приема/передачи, то с помощью функции:

*int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status);* можно узнать, не завершилась ли хотя бы одна из них (любая, ее порядковый номер будет записан в аргумент *index*, если аргумент *flag* будет после вызова иметь значение true).

Проверить завершение произвольного количества (но не обязательно всех) операций приема/передачи, связанных с квитанциями из массива *array_of_requests*, можно с помощью функции

*int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[]);* Аргумент *incount* (как и *count* в предыдущей функции) определяет размер массивов *array_of_requests, array_of_indices* и *array_of_statuses.* Аргумент *outcount* предназначен для возврата количества завершившихся операций, индексы соотвествующих квитанций будут записаны в массив *array_of_indices*.

Проверить факт завершения всех операций приема/передачи, квитанции которых содержатся во втором аргументе, можно с помощью функции

*int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[]);*

Проверить, отменена или нет операция приема/передачи, можно с помощью функции, устанавливающей *flag*:

*int MPI_Test_cancelled(MPI_Status *status, int *flag);*

Ожидать завершения любой ранее запущенной операции передачи/приема, квитанции на которые указаны в массиве, передаваемом в качестве второго аргумента, можно с помощью функции

*int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status array_of_statuses[]);* – индекс завершившейся операции заносится в аргумент *index*.

Ожидать завершения всех ранее запущенных операций передач/приемов, квитанции на которые указаны в массиве, передаваемом в качестве второго аргумента, можно с помощью функции:

int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[]);

Ожидать завершения любого количества ранее запущенных операций передачи/приема, квитанции на которые указаны в массиве,

передаваемом в качестве второго аргумента, можно с помощью функции

*int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[]);*

Все функции *MPI_Wait** и *MPI_Test** автоматически освобождают квитанции завершившихся неблокирующих операций приема/передачи и устанавливают соответствующие дескрипторы в состояние MPI_REQUEST_NULL. Использовать такую квитанцию впоследствии нельзя, ее надо заново инициализировать.

Есть функция, с помощью которой можно узнать, завершилась ли операция приема/передачи без деструкции соответствующей квитанции:

*int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status);*

Квитанция, связанная с любой неблокирующей операцией приема/передачи, может быть явно «освобождена» без ожидания завершения этой операции:

*int MPI_Request_free(MPI_Request *request) ;*

Параметр request устанавливается в значение MPI_REQUEST_ NULL. Связанная с квитанцией операция не прерывается, однако проверить ее завершение с помощью *MPI_Wait* или *MPI_Test* уже нельзя.

Любая блокирующая операция приема/передачи может быть инициирована в так называемом «отложенном» режиме. Это означает, что библиотека получает и запоминает аргументы этой операции, но начинает выполнять ее только при вызове одной из функций *MPI_Start* или *MPI_Startall*. Функции инициирования выглядят так:

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

*int MPI_Rsend_init (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);*

*int MPI_Ssend_init (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);*

*int MPI_Bsend_init (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);*

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

Во всех функциях **send_init* добавлен (по сравнению с соответствующей функцией **send*) последний аргумент *MPI_Request *request.*

В функции *MPI_Recv_init* этот аргумент появился взамен аргумента *MPI_Status *status*.

Отложенные операции приема/передачи можно запустить на выполнение с помощью функций

*int MPI_Start(MPI_Request *request);*

int MPI_Startall(int count, MPI_Request array_of_requests[]);

В MPI есть группа функций, совмещающих операции приема и передачи. Они достаточно часто применяются при программировании «каскадных» или «линейных» схем, когда необходимо осуществлять однотипный обмен данными между ветвями. Примером является функция:

int MPI_Sendrecv (void sendbuffer, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag, void* recvbuffer, int recvcount, MPI_Datatype recvdatatype, int src, int recvtag, MPI_Comm comm, MPI_Status* status);* – эта функция копирует данные из массива *sendbuffer* ветви с идентификатором *src* в буфер *recvbuffer* ветви с идентификатором *dest*.

Другая полезная функция:

int MPI_Sendrecv_replace (void buffer, int count, MPI_Datatype datatype, int dest, int sendtag, int src, int recvtag,MPI_Comm comm, MPI_Status* status);* – данные также передаются из ветви src в ветвь dest, но используется только один буфер (вначале сообщение из него передается, потом в него же принимается).

5.4. Коллективные операции взаимодействия процессов

Набор операций типа «точка–точка» в принципе является достаточным для программирования любых алгоритмов. Однако во многих случаях бывает удобнее использовать так называемые «коллективные операции», которые, как правило, будут выполняться быстрее. Например, часто возникает потребность разослать значение переменной или массива из одного процессора всем остальным. Конечно, такую рассылку можно реализовать с использованием операций Send/Recv, однако гораздо удобнее воспользоваться специальной коллективной операцией.

Главное отличие коллективных операций от операций типа «точка–точка» состоит в том, что в них всегда участвуют все ветви указанного коммуникатора. Несоблюдение этого правила приводит либо к аварийному завершению программы, либо к еще более неприятному ее зависанию.

Отличительные особенности коллективных операций:

– коллективные операции не взаимодействуют с операциями типа «точка–точка»;

– коллективные операции всегда выполняются в режиме с блокировкой;

– возврат из функции коллективного взаимодействия в каждой ветви происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие ветви завершили операцию;

– количество получаемых данных должно быть в точности равно количеству посланных данных;

– типы элементов посылаемых и получаемых сообщений должны совпадать;

– сообщения не имеют идентификаторов (тэгов).

Барьерная синхронизация всех ветвей коммуникатора:

int MPI_Barrier(MPI_Comm comm); – исполнение программы продолжится только тогда, когда все ветви коммуникатора *comm* вызовут эту функцию.

Рассылка одного и того же сообщения от одной ветви всем остальным ветвям данного коммуникатора:

int MPI_Bcast(void buffer, int count, MPI_Datatype datatype, int sender, MPI_Comm comm);* – эта функция в ветви с номером sender работает как *MPI_Send*, а во всех остальных ветвях – как *MPI_Recv*. Поэтому перед вызовом этой функции не надо выяснять с помощью условного оператора, является ветвь *sender'ом* или нет. После возврата из этой функции массив *buffer* во всех ветвях будет содержать одинаковые значения в первых *count* элементах типа *datatype*.

Рассылка частей сообщений от одной ветви всем остальным ветвям данного коммуникатора:

int MPI_Scatter (void sendbuffer, int sendcount, MPI_Datatype senddatatype, void* recvbuffer, int recvcount,MPI_Datatype recvdatatype, int sender, MPI_Comm comm);* – эта функция тоже в ветви с номером sender работает как *MPI_Send*, а во всех остальных ветвях – как *MPI_Recv*. Важно помнить, что должно выполняться условие *recvcount*sizeof(recvdatatype) >= sendcount*sizeof(senddatatype)*, иначе программа будет завершена аварийно. После возврата из этой функции массив *recvbuffer* будет содержать:

– в ветви 0: первые *sendcount* (0 – *sendcount*-1) элементов буфера *sendbuffer* передающей ветви;

– в ветви 1: *sendcount* элементов с номерами от *sendcount* до 2**sendcount*-1 буфера *sendbuffer* передающей ветви.

– …

Векторный вариант рассылки:

*int MPI_Scatterv(void *sendbuf, int *sendcnts, int *displs, MPI_Datatype senddatatype, void *recvbuf, int recvcnt, MPI_Datatype recvdatatype, int sender, MPI_Comm comm);* – отличается от предыдущей функции тем, что количества элементов данных, рассылаемых по ветвям, управляются содержимым массива *sendcnts* (при этом для всех *i* должно быть *recvcnt >= sendcnts[i]*, иначе параллельная программа аварийно завершается).

Сбор частей сообщений из всех ветвей коммуникатора в одну ветвь:

*int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype senddatatype, void *recvbuf, int recvcnt, MPI_Datatype recvdatatype, int receiver, MPI_Comm comm);* – выполняемые действия в точности противоположны по отношению к *MPI_Scatter*. В буфер *recvbuf* ветви *receiver* складываются порции данных из всех остальных ветвей в строгом соответствии с их номерами. Значение аргумента *recvcnt* может быть больше или равно *sendcnt*, но не наоборот. При *recvcnt* < *sendcnt* программа будет завершена аварийно.

Векторный вариант сбора частей сообщений от всех ветвей данного коммуникатора в одну ветвь:

int MPI_Gatherv(void sendbuf, int sendcount, MPI_Datatype senddatatype, void* recvbuf, int *recvcnts, int *displs, MPI_Datatype recvdatatype, int receiver, MPI_Comm comm);* – Как обычно, для каждой принимаемой порции должно удовлетворяться условие: *recvcnts[i]*sizeof(recvdatatype) >= sendcount*sizeof(senddatatype)*, иначе – аварийное завершение.

Функция *MPI_Allgather* выполняется так же, как *MPI_Gather*, но получателями являются все ветви группы данного коммуникатора. Данные, отправленные ветвью *i* из своего буфера *sendbuf*, помещаются в *i*-ю порцию буфера *recvbuf* каждой ветви. После завершения операции содержимое буферов приема *recvbuf* у всех ветвей будет одинаковым.

int MPI_Allgather(void sendbuf, int sendcount, MPI_Datatype senddatatype, void* recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm);*

Функция *MPI_Allgatherv* является аналогом функции *MPI_Gatherv*, но сборка данных выполняется всеми ветвями коммуникатора. Поэтому в списке аргументов отсутствует параметр *receiver*.

int MPI_Allgatherv(void sendbuf, int sendcount, MPI_Datatype senddatatype, void* recvbuf, int *recvcnts, int *displs, MPI_Datatype recvdatatype, MPI_Comm comm);*

Функция *MPI_Alltoall* совмещает в себе операции *Scatter* и *Gather* и является, по сути, расширением операции *Allgather*, когда каждая ветвь посылает различные данные разным получателям. Ветвь *i* посылает *j*-й блок своего буфера *sendbuf* ветви *j*, которая помещает его в *i*-й блок своего буфера *recvbuf*. Количество посланных данных должно быть равно количеству полученных данных для каждой пары ветвей.

*int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm);*

Функция *MPI_Alltoallv* является векторным вариантом функции *MPI_Alltoall*, позволяющим гибко управлять местоположением и размерами порций сообщений, передаваемых между ветвями.

*int MPI_Alltoallv(void *sendbuf, int *sendcnts, int *senddispls, MPI_Datatype senddatatype, void *recvbuf, int *recvcnts, int *recvdispls, MPI_Datatype recvdatatype, MPI_Comm comm);*

Еще более гибкой является функция *MPI_Alltoallw*, дополнительно позволяющая передавать/принимать данные разных типов:

*int MPI_Alltoallw(void *sendbuf, int *sendcnts, int *senddispls, MPI_Datatype *senddatatypes, void *recvbuf, int *recvcnts, int *recvdispls, MPI_Datatype *recvdatatypes, MPI_Comm comm);*

Операцией редукции называется операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора. В частности, если компоненты вектора расположены в адресных пространствах ветвей, выполняющихся на различных процессорах, то в этом случае говорят о глобальной (параллельной) редукции.

Например, пусть в адресном пространстве ветвей некоторой группы имеются собственные копии переменной var (необязательно имеющие одно и то же значение). Тогда применение к этой переменной операции вычисления глобальной суммы или, другими словами, операции редукции SUM возвратит одно значение, которое будет содержать сумму всех локальных значений этой переменной.

Использование операций редукции является одним из основных средств организации распределенных вычислений. В функциях редукции MPI могут быть использованы следующие предопределенные операции:

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

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

Для языка C:

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

Функция *MPI_Reduce*:

int MPI_Reduce(void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int receiver, MPI_Comm comm);* – над всеми эле-

ментами буфера *sendbuf* всех ветвей коммуникатора выполняется операция *op*, результаты ее собираются в буфере *recvbuf* ветви *receiver*. В качестве операции *op* можно использовать либо одну из предопределенных операций, либо операцию, сконструированную пользователем. Все предопределенные операции являются ассоциативными и коммутативными. Сконструированная пользователем операция, по крайней мере, должна быть ассоциативной. Порядок редукции определяется номерами ветвей в коммуникаторе. Тип *datatype* элементов должен быть совместим с операцией *op*.

Похожа на *MPI_Reduce* функция *MPI_Allreduce*, только результаты собираются не в одной ветви, а во всех ветвях:

int MPI_AllReduce(void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);*

Функция *MPI_Reduce_scatter* отличается от *MPI_Allreduce* тем, что результат операции разрезается на непересекающиеся части по числу ветвей в группе, *i*-я часть посылается *i*-й ветви в ее буфер приема. Длины этих частей задает третий параметр, являющийся массивом.

*int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int *recvcnts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)*

Функция *MPI_Scan* выполняет префиксную включающую редукцию. Параметры такие же, как в *MPI_Allreduce*, но получаемые каждой ветвью результаты отличаются друг от друга. Операция пересылает в буфер приема *i*-й ветви редукцию значений из входных буферов ветвей с номерами 0, ... , *i* включительно.

int MPI_Scan(void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);*

Функция *MPI_Exscan* выполняет префиксную исключающую редукцию (собственные значения каждой ветви не участвуют в редукции):

int MPI_Exscan(void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);*

Определение собственных операций для функций MPI_Reduce, MPI_Reduce_scatter, MPI_Allreduce и MPI_Scan:

*int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op);*

Описание типа собственной функции выглядит следующим образом:

*typedef void (MPI_User_function)(void *a, void *b, int *len, MPI_Datatype *dtype);*

Эта функция должна выполнять обработку данных из векторов *a* и *b* следующим образом:

b[i] = a[i] <пользовательская операция> b[i] для i = 0, ..., len-1.

Удаление ранее определенной собственной функции

*int MPI_Op_free(MPI_Op *op);*

В результате вызова переменная *op* получает значение MPI_OP_NULL.

5.5. Производные типы и упаковка/распаковка данных

Все рассмотренные ранее коммуникационные операции позволяют посылать или получать последовательность элементов одного типа, занимающих смежные области памяти. При разработке параллельных программ часто возникает потребность передавать данные разных типов (например, структуры) или данные, расположенные в несмежных областях памяти ветви-отправителя (например, части массивов, не образующие непрерывную последовательность элементов).

Для эффективной пересылки данных в таких случаях MPI предоставляет два механизма:

– возможность создания производных типов данных для использования в коммуникационных операциях вместо предопределенных типов MPI;

– пересылку упакованных данных (процесс-отправитель упаковывает пересылаемые данные перед их отправкой, а процесс-получатель распаковывает их после получения).

Производные типы данных стандарта MPI не являются в полном смысле типами данных, как это понимается в языках программирования. Они не могут использоваться ни в каких других операциях, кроме операций передачи/приема сообщений. Производные типы данных MPI следует понимать просто как описатели расположения в памяти элементов базовых типов. Производный тип MPI представляет собой скрытый (opaque) объект, который специфицирует две вещи:

– последовательность базовых типов;

– последовательность их смещений.

Упорядоченный набор этих пар называется отображением (картой) типа:

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

Значения смещений в карте типа не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию.

Стандартный сценарий определения и использования производных типов включает следующие шаги.

– Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью функций-конструкторов.

– Новый производный тип регистрируется вызовом функции *MPI_Type_commit.*

– После регистрации новый производный тип можно использовать в коммуникационных операциях и при конструировании других типов. Предопределенные типы MPI считаются зарегистрированными.

– С производными типами можно выполнять некоторые дополнительные операции (в частности, узнавать их размер, протяженность и некоторые другие характеристики).

– Когда производный тип становится ненужным, он уничтожается функцией *MPI_Type_free.*

Некоторые из перечисленных ниже функций-конструкторов появились только в стандарте MPI-2, однако для сохранения единства в изложении материала сведения о них приводятся здесь.

Самый простой конструктор типа *MPI_Type_contiguous* создает новый тип, элементы которого состоят из указанного числа элементов базового типа, занимающих смежные области памяти:

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

Следующий конструктор создает тип, элемент которого представляет собой несколько равноудаленных друг от друга блоков из одинакового числа смежных элементов базового типа:

*int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);* – эта функция создает тип *newtype*, элемент которого состоит из *count* блоков, каждый из которых содержит одинаковое число *blocklength* элементов типа *oldtype*. Шаг *stride* между началом блока и началом следующего блока всюду одинаков и кратен протяженности представления базового типа.

Конструктор типа *MPI_Type_hvector* расширяет возможности конструктора *MPI_Type_vector*, позволяя задавать произвольный шаг между началами блоков в байтах:

*int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype);*

Конструктор типа *MPI_Type_indexed* является более универсальным конструктором по сравнению с *MPI_Type_vector*, так как элемен-

ты создаваемого типа состоят из произвольных по длине блоков с произвольным смещением блоков от начала размещения элемента. Смещения задаются в элементах базового типа:

*int MPI_Type_create_indexed(int count , int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);* – эта функция создает тип *newtype*, каждый элемент которого состоит из count блоков, где *i*-й блок содержит *array_of_blocklengths[i]* элементов базового типа и смещен от начала размещения элемента нового типа на *array_of_displacements[i]* элементов базового типа.

Конструктор типа *MPI_Type_create_hindexed* идентичен конструктору *MPI_Type_indexed* за исключением того, что смещения измеряются в байтах:

*int MPI_create_hindexed(int count , int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);* – элемент нового типа состоит из *count* блоков, где *i*-й блок содержит *array_of_blocklengths[i]* элементов старого типа и смещен от начала размещения элемента нового типа на array_of_*displacements[i]* байт.

Конструктор типа *MPI_Type_create_indexed_block* похож на конструктор *MPI_Type_indexed* за исключением того, что все блоки одинаковы:

*int MPI_Type_create_indexed_block(int count , int blocklength, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);*

Конструктор типа *MPI_Type_create_struct* – самый универсальный из всех конструкторов. Создаваемый им тип является структурой, состоящей из произвольного числа блоков, каждый из которых может содержать произвольное число элементов одного из базовых типов и может быть смещен на произвольное число байтов от начала размещения структуры:

*int MPI_Type_create_struct(int count , int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);* – эта функция создает тип *newtype*, элемент которого состоит из count блоков, где *i*-й блок содержит *array_of_blocklengths[i]* элементов типа *array_of_types[i]*. Смещение *i*-го блока от начала размещения элемента нового типа измеряется в байтах и задается в *array_of_displacements[i*].

Конструктор типа данных «субмассив»:

int MPI_Type_create_subarray(int ndims, int array_of_sizes, int* array_of_subsizes, int* array_of_starts, int order, MPI_Datatype oldtype,*

*MPI_Datatype *newtype);* – конструктор типа для субмассива создает тип данных MPI, описывающий *n*-мерный субмассив *n*-мерного масива. Субмассив может находиться в любом месте полного массива и может быть любого ненулевого размера вплоть до размера полного массива. Этот конструктор позволяет создавать типы файлов для доступа к массивам, разбитым между процессами по блокам через один файл, содержащий глобальный массив.

И, наконец, конструктор распределенного массива поддерживает распределения данных, сходные с HPF (High Performance Fortran). Кроме этого в отличие от HPF порядок хранения может быть задан как для массивов Си, так и для ФОРТРАНА:

int MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype newtype);*

Функция *MPI_Type_commit* регистрирует созданный производный тип. Только после регистрации новый тип может использоваться в коммуникационных операциях:

*int MPI_Type_commit(MPI_Datatype *datatype);*

Любой зарегистрированный тип данных можно продублировать с помощью функции

*int MPI_Type_dup(MPI_Datatype datatype, MPI_Datatype *newtype);*

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах: протяженность типа определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как адрес последней ячейки данных – адрес первой ячейки данных + длина последней ячейки данных. Протяженность может быть получена с помощью функций *MPI_Type_get_extent*, а если к типу применялись операции изменения нижней и/или верхней границы, то *MPI_Type_get_true_extent*.

Размер типа определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа. Размер типа данных MPI может быть получен с помощью функции MPI_Type_size.

Для простых типов протяженность и размер совпадают.

*int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent););* – функция возвращает нижнюю границу и протяженность типа.

*int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent);* – возвращает истинные нижнюю границу и протя-

женность даже в том случае, если для данного типа изменялась нижняя граница.

*int MPI_Type_size(MPI_Datatype datatype, int *size);* – возвращает в переменной *size* «чистый» размер элемента некоторого типа (за вычетом пустых промежутков), т. е. количество байт данных, которые будут фактически передаваться при коммуникациях между ветвями.

*int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype);* – эта функция возвращает в *newtype* дескриптор нового типа данных, идентичного *oldtype*, за исключением того, что нижняя граница типа данных установлена в *lb*, а верхняя – в *lb* + *extent*. Любые предыдущие маркеры *lb* и *ub* стираются, и в позиции, указанные аргументами *lb* и *extent*, помещается новая пара маркеров. Это влияет на поведение типа данных при коммуникациях с *count*>1 и при создании новых порожденных типов данных.

*int MPI_Type_set_name(MPI_Datatype type, char *type_name);* – присваивает символическое имя производному типу данных (удобно для отладки).

*int MPI_Type_get_name(MPI_Data_type type, char *type_name, int *resultlen);* – возвращает имя производного типа данных.

*int MPI_Type_free(MPI_Datatype *datatype);* – уничтожает производный тип. Эта функция устанавливает дескриптор типа в состояние MPI_DATATYPE_NULL, что не повлияет ни на выполняющиеся в данный момент коммуникационные операции с этим типом данных, ни на производные типы, которые ранее были определены через уничтоженный тип.

Передача/прием упакованных данных.

Функция *MPI_Pack* упаковывает элементы предопределенного или производного типа, помещая их побайтное представление в выходной буфер:

int MPI_Pack(void inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm);* – функция *MPI_Pack* упаковывает *incount* элементов типа *datatype* из области памяти с начальным адресом *inbuf*. Результат упаковки помещается в выходной буфер с начальным адресом *outbuf* и размером *outsize* байт. Параметр *position* указывает текущую позицию в байтах, начиная с которой будут размещаться упакованные данные. После возврата из функции значение *position* будет увеличено на число упакованных байт, указывая на первый свободный байт в выходном буфере. Параметр

comm при последующей посылке упакованного сообщения будет использован как коммуникатор.

Функция *MPI_Pack_size* позволяет определить размер буфера, необходимый для упаковки заданного количества данных типа *datatype*:

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

Функция *MPI_Unpack* извлекает заданное число элементов некоторого типа из побайтного представления элементов во входном массиве:

int MPI_Unpack(void inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm);* – эта функция извлекает *outcount* элементов типа *datatype* из побайтного представления элементов в массиве *inbuf*, начиная с адреса *position*, и помещает их в область памяти с начальным адресом *outbuf*. После возврата из функции параметр *position* будет увеличен на размер распакованного сообщения.

5.6. Управление группами ветвей и коммуникаторами

Коммуникаторы определяют области действия любых операций обмена данными в MPI. Коммуникаторы разделяются на два вида: интра-коммуникаторы (внутригрупповые коммуникаторы), предназначенные для операций в пределах отдельной группы процессов, и интер-коммуникаторы (межгрупповые коммуникаторы), предназначенные для обменов между двумя группами процессов. В большинстве случаев используются интракоммуникаторы, интеркоммуникаторы здесь рассматриваться не будут.

Типичной проблемой, которую может решить использование коммуникаторов, является проблема недопущения «пересечения» обменов по тегам. В самом деле, если программист использует какую-то библиотеку параллельных методов, то он часто не знает, какие теги использует эта библиотека при передаче сообщений. В таком случае существует опасность, что тег, выбранный программистом, совпадет с одним из тегов, используемых библиотекой.

Чтобы этого не произошло, программист может создать собственный коммуникатор и выполнять операции обмена в его рамках. Несмотря на то что этот коммуникатор будет содержать те же ветви программы и такие же теги, что и коммуникатор библиотеки, сообщения, передаваемые в рамках этих двух разных коммуникаторов, не перепутаются.

Управление группами

С понятием коммуникатора тесно связано понятие группы ветвей. Под группой понимают упорядоченное множество ветвей программы. Каждой ветви в группе соответствует уникальный номер – ранг. Группа – отдельное понятие MPI, и операции с группами могут выполняться отдельно от операций с коммуникаторами, но операции обмена для указания области действия всегда используют коммуникаторы, а не группы. С группами ветвей в MPI допустимы следующие действия:

– объединение групп;
– пересечение групп;
– образование разности групп.

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

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

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

Размер группы (количество ветвей в ней) можно получить с помощью функции.

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

Номер ветви в группе можно определить, вызвав функцию

*int MPI_Group_rank(MPI_Group group, int *rank);*

Если есть две группы ветвей (неважно, каким образом они были образованы), то с помощью функции

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

можно установить соответствие между номерами ветвей в этих группах.

Две группы ветвей можно сравнить:

*int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result);* – здесь *result* – переменная, получающая значение в результате сравнения:

– MPI_IDENT, если все элементы двух групп одинаковы и следуют друг за другом в одинаковом порядке;

– MPI_SIMILAR, если все элементы двух групп одинаковы, но порядок следования различен;

– MPI_UNEQUAL – во всех остальных случаях.

Однажды созданная группа не может быть изменена. Она может быть только уничтожена (освобождена). Для конструирования новых групп можно использовать следующие функции.

Создание новой группы из явно указанных ветвей старой группы:

*int MPI_Group_incl(MPI_Group oldgroup, int n, int *ranks, MPI_Group *newgroup);* – эта функция создает новую группу *newgroup*, которая состоит из ветвей существующей группы, перечисленных в массиве *ranks*. Ветвь с номером *i* в новой группе есть ветвь с номером *ranks[i]* в существующей группе. Каждый элемент в массиве *ranks* должен иметь корректный номер ветви из группы *oldgroup*, и среди этих элементов не должно быть совпадающих, иначе вся программа завершится аварийно.

Создание новой группы из таких ветвей старой группы, которые не принадлежат явно указанному перечню:

*int MPI_Group_excl(MPI_Group oldgroup, int n, int *ranks, MPI_Group *newgroup);* эта функция создает новую группу *newgroup*, которая состоит из всех ветвей существующей группы, за исключением ветвей, перечисленных в массиве *ranks*. Каждый элемент в массиве *ranks* должен иметь корректный номер ветви из группы *oldgroup*, и среди этих элементов не должно быть совпадающих, иначе вся программа завершится аварийно.

Есть две функции, являющиеся обобщением функций *MPI_Group_incl* и *MPI_Group_excl*:

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

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

У этих функций одномерные массивы номеров ranks заменены двумерными массивами триплетов, каждый из которых имеет вид:

<нижняя граница> <верхняя граница> <шаг>

Границы и шаг должны определять допустимые номера ветвей, причем все номера ветвей новой группы, определяемые с их использованием, должны быть различными, иначе вся программа завершится аварийно. Последовательность, определяемая триплетом, может быть пустой (например, если верхняя граница меньше нижней). Эти функции удобно применять в программах для суперкомпьютеров с очень большим (десятки и сотни тысяч) количеством ядер. Пример массива ranges:

int rngs[][] = {{2, 32768, 16},{0, 131072, 256}};

Здесь имеются два триплета:

– 2, 32768, 16 – порождает последовательность номеров 2, 18, 34, 50, 66, … 32738, 32754 (всего 2047 номеров)

– 0, 131072, 256 – порождает последовательность 0, 256, 512, … 130816, 131072 (всего 512 номеров)

Следующие три функции имеют одинаковые наборы аргументов и создают новую группу как результат заданной теоретико-множественной операции над множествами ветвей двух групп.

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

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

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

Уничтожение группы выполняется путем вызова функции

*int MPI_Group_free(MPI_Group *group).*

Управление коммуникаторами

Создание коммуникатора – операция коллективная (и должна вызываться всеми процессами коммуникатора).

Функция *MPI_Comm_dup* копирует уже существующий коммуникатор:

*int MPI_Comm_dup(MPI_Comm oldcomm, MPI_comm *newcomm);* – в результате вызова этой функции будет создан новый коммуникатор *newcomm*, включающий в себя те же ветви, что и коммуникатор *oldcomm*.

Для создания нового коммуникатора служит функция *MPI_Comm_create*:

*int MPI_comm_create(MPI_Comm oldcomm, MPI_Group group, MPI_Comm *newcomm);* – этот вызов создает новый коммуникатор *newcomm*, который будет включать в себя ветви группы *group* коммуникатора *oldcomm*. Это коллективная операция и она должна вызываться всеми ветвями родительского коммуникатора, даже если ветвь не входит в группу нового коммуникатора. Такие ветви в качестве нового коммуникатора получат MPI_COMM_NULL.

Функция расщепления коммуникатора:

*int MPI_Comm_split(MPI_Comm oldcomm, int color, int key, MPI_Comm *newcomm);* – эта функция расщепляет группу, связанную с родительским коммуникатором, на непересекающиеся подгруппы по

одной на каждое значение признака подгруппы *color*. Значение *color* должно быть неотрицательным. Каждая подгруппа будет содержать ветви с одним и тем же значением признака *color*. Параметр *key* управляет упорядочиванием внутри новых групп: меньшему значению *key* соответствует меньшее значение идентификатора ветви. В случае равенства параметра *key* для нескольких ветвей упорядочивание выполняется в соответствии с порядком в родительской группе. Каждая ветвь получит свой новый коммуникатор.

Функция сравнения двух коммуникаторов:

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

Функция уничтожения коммуникатора:

*int MPI_Comm_free(MPI_Comm *comm);*

5.7. Управление виртуальными топологиями

Топология – механизм MPI, который позволяет устанавливать дополнительную систему адресации для процессов. Топология в MPI является логической и никак не связана с топологией физической среды передачи данных. Введение в MPI поддержки топологий связано с тем, что большинство прикладных алгоритмов устроено таким образом, когда ветви должны быть упорядоченными в соответствии с некоторой топологией. В MPI поддерживаются два вида топологий – прямоугольная решетка произвольной размерности (декартова топология) и граф.

Декартовы топологии или решетки часто применяются при решении прикладных задач. Известно много алгоритмов, в которых используются «сетки». Один из наиболее широко представленных классов таких задач – сеточные методы решения дифференциальных уравнений в частных производных. Именно для упрощения программирования «сеточных» задач в MPI и были введены функции управления топологиями.

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

*int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims, int *periods, int reorder, MPI_Comm grid_comm);* – операция создания топологии является коллективной операцией, ее должны выполнить все процессы коммуникатора *oldcomm*. При этом если какие-то ветви не попадают в новую группу, то для них возвращается результат

MPI_COMM_NULL. В случае, когда размеры заказываемой сетки больше имеющегося в группе числа ветвей, функция завершается аварийно. Значение параметра *reorder=false* означает, что идентификаторы всех ветвей в новой группе будут такими же, как в старой группе, при значении *true* MPI будет пытаться перенумеровать их с целью оптимизации коммуникаций. С помощью этой функции можно создавать топологии с произвольным числом измерений, причем по каждому измерению в отдельности можно накладывать периодические граничные условия. Таким образом, для одномерной топологии мы можем получить или линейную структуру, или кольцо, в зависимости от того, какие граничные условия будут наложены. Для двумерной топологии – соответственно либо прямоугольник, либо цилиндр, либо тор и т. д.

Перед вызовом функции MPI_Cart_create полезно определить оптимальное распределение ветвей программы по предполагаемой решетке с помощью вызова функции

*int MPI_Dims_create(int nnodes, int ndims, int *dims);* – перед вызовом функции в массив dims должны быть занесены целые неотрицательные числа. Если элементу массива *dims[i]* присвоено положительное число, то для этой размерности решетки вычисление не производится (число ветвей вдоль этого направления считается заданным). Вычисляются только те компоненты *dims[i]*, для которых перед обращением к функции были присвоены значения 0. Функция стремится создать максимально равномерное распределение ветвей вдоль направлений, выстраивая их по убыванию.

Для определения того, связана ли (и какая) топология с коммуникатором, служит функция

*int MPI_Topo_test(MPI_Comm comm, int *status);* – функция *MPI_Topo_test* возвращает через переменную *status* топологию коммуникатора *comm*. Возможные значения:

- MPI_CART – декартова топология;
- MPI_GRAPH – топология графа;
- MPI_UNDEFINED – топология не задана.

Функция опроса числа измерений декартовой топологии:

*int MPI_Cartdim_get(MPI_Comm comm, int *ndims);*– результат, возвращаемый этой функцией, может быть использован в качестве параметра для вызова функции *MPI_Cart_get*, которая служит для получения более детальной информации.

*int MPI_Cart_get(MPI_Comm comm, int ndims, int *dims, int *periods, int *coords);*

Для определения декартовых координат ветви по ее рангу (номеру) можно воспользоваться функцией

*int MPI_Cart_coords(MPI_Comm comm,int rank,int maxdims,int *coords);*

Функция *MPI_Cart_rank* возвращает ранг ветви по ее декартовым координатам:

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

Для совокупностей ветвей, организованных в гиперкуб, могут выполняться обмены особого рода – сдвиги. Существует два вида сдвигов:

– циклический сдвиг на *k* элементов вдоль ребра решетки. Данные от ветви с номером *n* пересылаются ветви с номером (*n*+*k*) % *dim*, где *dim* – размерность измерения, вдоль которого производится сдвиг;

– линейный сдвиг на *k* позиций вдоль ребра решетки. Данные от ветви с номером *n* пересылаются ветви с номером (*n*+*k*) (если такая существует).

Собственно передачи/приемы данных выполняются функцией *MPI_Sendrecv*, но вычисление номеров ветвей приемников/ получателей целесообразно выполнять с помощью вызова функции

int MPI_Cart_shift(MPI_Comm comm, int direction,int displ, int source, int* dst);*

Часто используемая операция – выделение в декартовой топологии подпространств меньшей размерности и связывание с ними отдельных коммуникаторов. Функция выделения подпространства в декартовой топологии:

*int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm);*

Создание коммуникаторов с произвольной топологией выполняется функцией

*int MPI_Graph_create(MPI_Comm oldcomm, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph);* – эта функция передает дескриптор новому коммуникатору, к которому присоединяется информация о графовой топологии. Если размер декартовой решетки *nnodes* меньше, чем размер группы коммуникатора, то некоторым ветвям возвращается значение MPI_COMM_NULL, по аналогии с *MPI_Cart_split* и *MPI_Comm_split*. Вызов будет неверным, если он определяет граф большего размера, чем размер группы исходного коммуникатора.

Структуру графа определяют три параметра: nnodes, index и edges. Способ определения аргументов nnodes, index, и edges иллюстрируется следующим простым примером.

Пусть есть четыре ветви, имеющие следующих соседей: ветвь 0 – 1 и 3, ветвь 1 – 0, ветвь 2 – 3 и ветвь 3 – 0 и 2. Тогда исходными аргументами функции должны быть:

nnodes = 4, *index* = {2, 3, 4, 6}, *edges* = {1, 3, 0, 3, 0, 2}

Заметим, что не имеет никакого смысла создавать виртуальную топологию полного графа. Возможность передавать данные между любой парой ветвей программы существует изначально, как только разработчик подключает к программе библиотеку MPI. Виртуальные топологии – это средство упростить формирование координат «соседей» ветви для указания их в вызовах функций приема/передачи данных и не более того.

Для коммуникаторов с графовой топологией есть функции получения сведений, аналогичные функциям *MPI_Cartdims_get* и *MPI_Cart_get*. Размеры массивов вершин и ребер графа могут быть получены с помощью вызова функции

*int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges);* – информация, получаемая в результате вызова этой функции, может быть использована для корректного определения размера векторов index и edges в последующем вызове функции

*int MPI_Graph_get(MPI_Comm comm, int nnodes, int nedges, int *index, int *edges);*

Информацию о смежных ветвях любой вершины графа можно получить с помощью двух функций. Первая из них является вспомогательной и возвращает количество «соседей» ветви с заданным рангом:

*int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors);*

После получения количества соседей можно получить полный перечень этих соседей с помощью функции:

int MPI_Graph_*neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors);*

При необходимости использования других функций можно использовать руководства по программированию для MPI и справочные материалы.

5.8. Динамическое порождение ветвей

Модель процессов стандарта MPI-2 позволяет выполнять создание и совместное завершение процессов после запуска приложения MPI. Она обеспечивает механизм установки соединения между вновь со-

зданными ветвями и ветвями уже существующего приложения MPI. Более того, она предлагает механизм установки соединения между двумя существующими приложениями MPI, даже если ни одно из них не запускает другого. Модель процессов MPI-2 удовлетворяет следующим требованиям.

– Модель процессов MPI-2 должна реализовываться в подавляющем большинстве современных параллельных сред – от тесно связанных многопроцессорных систем до гетерогенных сетей рабочих станций.

– MPI не должен принимать на себя обязанности операционной системы. Вместо этого он должен предлагать прозрачный интерфейс между приложением и системным программным обеспечением.

– MPI должен продолжать гарантировать детерминизм коммуникации, т. е. управление процессами не должно вводить неизбежных условий возникновения «гонок».

– Программы MPI-1 должны работать под управлением MPI-2. Другими словами, статическая модель процессов MPI-1 должна быть специальным частным случаем динамической модели MPI-2.

Модель управления процессами MPI-2 решает эти вопросы двумя способами. Во-первых, стандарт MPI главным образом сохраняет библиотеку коммуникации. Он не управляет параллельной средой, в которой выполняются параллельные программы, хотя он и предлагает минимальный интерфейс между приложением и внешними менеджерами ресурсов и процессов.

Во-вторых, MPI-2 не изменяет концепции коммуникатора. Когда коммуникатор создан, он ведет себя точно так же, как определенный в MPI-1. Коммуникатор никогда не изменяется после создания и всегда создается с использованием определенных коллективных операций.

Запуск новой совокупности параллельных ветвей осуществляется вызовом функции (однако если возможно, то, в силу соображений производительности, нужно запускать все ветви сразу, как единое приложение MPI):

*int MPI_Comm_spawn(char *command, char **argv, int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int *array_of_errcodes);* – это коллективная операция, в которой должны участвовать все ветви коммуникатора *comm* (все аргументы до *root* включительно используются только в ветви *root*). Она пытается запустить *maxprocs* одинаковых копий программы *MPI*, определяемой *command*, устанавливая с ними соединение и возвращая интеркомму-

никатор *intercomm*. Порожденные ветви называются потомками, а ветви, их породившие, родителями. Потомки имеют свой собственный MPI_COMM_WORLD, отдельный от родителей. Этот вызов не завершается, пока во всех потомках не будет вызвана *Mpi_Init*. Подобным образом *Mpi_Init* в потомках не завершается, пока все родители не вызовут *MPI_Comm_spawn*. В этом смысле *MPI_Comm_spawn* в родителях и MPI_INIT в потомках формируют коллективную операцию над объединением родительских и дочерних ветвей. Интеркоммуникатор, возвращаемый *MPI_Comm_spawn*, содержит родительские ветви в локальной группе и ветви-потомки в удаленной группе. Порядок ветвей в локальной и удаленной группах такой же, как и порядок группы *comm* для родителей и MPI_COMM_WORLD для потомков. Этот интеркоммуникатор может быть получен в потомке через функцию *MPI_Comm_get_parent*.

Хотя этого механизма достаточно для большинства случаев, он не позволяет порождать новые группы ветвей из нескольких исполняемых файлов или из одного файла с разными наборами аргументов. Следующая функция порождает ветви из нескольких исполняемых файлов или одного файла с разными наборами аргументов, устанавливает с ними связь и помещает их в один MPI_COMM_WORLD:

*int MPI_Comm_spawn_multiple (int count, char **array_of_commands, char ***array_of_argv, int *array_of_maxprocs, MPI_Info *array_of_info, int root, MPI_Comm comm, MPI_Comm *intercomm, int *array_of_errcodes);*

Любая ветвь может получить коммуникатор, включающий ее родителей с помощью вызова:

*int MPI_Comm_get_parent (MPI_Comm *parent);*

5.9. Удаленный доступ к памяти (односторонние взаимодействия)

Удаленный доступ к памяти (RMA) расширяет механизмы взаимодействий MPI, позволяя одной ветви определять все коммуникационные параметры как для посылающей стороны, так и для получающей.

Во время любого из рассмотренных ранее коллективного MPI-взаимодействия взаимосвязанно выполняются:

– передача данных от отправителя к получателю;
– синхронизация отправителя и получателя.

В отличие от коллективных взаимодействий функции RMA спроектированы таким образом, что эти операции разделяются, передача

данных в память / из памяти одной ветви выполняется без ее участия по запросу другой ветви. Именно поэтому такие передачи и называются односторонними взаимодействиями.

Инициатором RMA-взаимодействия называется ветвь, которая запрашивает доступ к памяти другой ветви, и адресатом – ветвь, к памяти которой выполняется обращение. Инициатор и адресат могут быть одной и той же ветвью, тогда операции передачи данных используются просто для их перемещения в пределах памяти этой ветви.

Тем не менее коллективность имеется и в односторонних взаимодействиях. Прежде всего коллективными являются: операция создания «окна» для удаленного доступа к памяти некоторой ветви параллельной программы, операция удаления (уничтожения) такого окна и операции синхронизации.

Собственно удаленный доступ к окну памяти реализуется с использованием одной из трех функций:

1. *MPI_Put* передает данные из памяти инициатора в память адресата.

2. *MPI_Get* передает данные из памяти адресата в память инициатора.

3. *MPI_Accumulate* модифицирует данные в памяти адресата (например, добавляя к ним значения, посланные из памяти инициатора).

Эти три операции являются неблокирующими: т. е. вызов функции инициирует взаимодействие, после чего передача данных может продолжаться после возврата из вызова. Выполнение передачи завершается как в инициаторе, так и в адресате, когда инициатором выдан последующий синхронизационный вызов к участвующему оконному объекту.

Коллективная операция инициализации позволяет каждой ветви из группы интракоммуникатора определить «окно» в своей памяти, которое становится доступным для других ветвей этой группы. Вызов возвращает объект со скрытой структурой, представляющий группу процессов, которые являются обладателями и имеют доступ к окну и к его атрибутам.

int MPI*_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win);* – этот вызов возвращает оконный объект, который может использоваться всеми ветвями коммуникатора *comm* для выполнения RMA-операций. Каждая ветвь определяет окно в своей памяти, которое она предоставляет для дистанционно-

го доступа всем ветвям из группы *comm*. Окно состоит из *size* байт, начинающихся с адреса *base*. Если значение *size* равно нулю, то окно из памяти этой ветви недоступно другим ветвям. Для упрощения адресной арифметики в будущих RMA-операциях задается аргумент *disp_unit*, определяющий коэффициент масштабирования. Обычным выбором для значения *disp_unit* являются либо 1 (нет масштабирования), либо *sizeof(type)* для окон, которые состоят из массива элементов типа *type*. В последнем случае в RMA-вызовах можно будет использовать индексы элементов массивов, причем правильное масштабирование к байтовому смещению обеспечивается даже в неоднородной среде. Параметр *info* предоставляет подсказку исполняющей системе для выполнения оптимизации относительно ожидаемой структуры использования окна. Разные ветви в группе коммуникационного взаимодействия *comm* могут определять окна-адресаты, полностью различающиеся по расположению, размеру, по единицам смещения и параметру (аргументу) *info*. Одна и та же область памяти может использоваться одновременно в нескольких разных окнах. Однако следует помнить, что перекрывающиеся окна являются источником трудно обнаруживаемых ошибок. Окно можно создать в любой части памяти ветви. На некоторых платформах эффективность окон будет выше, если они создаются в памяти, которая распределена вызовом *MPI_Alloc_mem*. Эффективность также обычно улучшается, когда границы окна выравниваются по «естественным» границам (слово, двойное слово, строка кэша, страничный блок и т. д.).

Ранее созданное окно может быть уничтожено с помощью коллективной операции:

*int MPI_Win_free(MPI_Win *win);* – освобождает оконный объект и заносит в переменную *win* пустой дескриптор (значение MPI_WIN_NULL). Эта функция может вызываться ветвью только после того, как она завершила свое участие в RMA-взаимодействиях с оконным объектом *win*:

– ветвь вызвала *MPI_Win_fence* или *MPI_Win_wait*, чтобы выполнить согласование с предыдущим вызовом *MPI_Win_post*;

– ветвь вызвала *MPI_Win_complete*, чтобы выполнить согласование с предыдущим вызовом *MPI_Win_start*;

– или ветвь выполнила вызов *MPI_Win_unlock*, чтобы выполнить согласование с предыдущим вызовом *MPI_Win_lock*.

Функция *MPI_Win_free* внутри использует барьерную синхронизацию: никакая из ветвей не получит возврат из нее, пока все ветви из группы данного окна *win* не вызовут эту функцию.

Операция передачи данных из памяти инициатора в память адресата:

*int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win);* – из памяти ветви-инициатора, начиная с адреса *origin_addr*, передаются *origin_count* следующих друг за другом записей типа *origin_datatype*,в память адресата, определяемого парой *win* и *target_rank*. Данные записываются в буфер адресата по адресу *target_addr*, равному *win_base* + *target_disp* * *disp_unit*, где *win_base* и *disp_unit* – базовый адрес и единица смещения, определенные при инициализации оконного объекта *win* ветвью-адресатом. Передача данных происходит так же, как если бы инициатор выполнил операцию *MPI_Send* с аргументами *origin_addr, origin_count, origin_datatype, target rank, tag, comm*, и процесс-получатель выполнил операцию *MPI_Recv* с аргументами *target_addr, target_datatype, source, tag, comm*, где *target_addr* – это адрес буфера получателя, вычисленный, как указано в предыдущем абзаце, а *comm* – это коммуникатор для группы ветвей *win*.

Операция передачи данных из памяти адресата в память инициатора:

*int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win);* – все аргументы точно такие же. Передача данных выполняется в обратном направлении: из памяти адресата в память инициатора.

Модификация данных в памяти адресата с использованием заданной операции и данных из памяти инициатора:

*int MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win);* – все аргументы такие же, как и у двух предыдущих функций, добавлен только аргумент *op*, используемый точно так же, как в операции *MPI_Reduce* для «свертки» значений данных из буферов ветви-инициатора и ветви-адресата.

Синхронизация при удаленном доступе к памяти

RMA-взаимодействия могут выполняться в двух режимах:

– с активным адресатом, когда данные перемещаются из памяти одной ветви в память другой, и обе явно участвуют в коммуникациях. Такая модель похожа на обычную передачу сообщений, за исключением того, что все аргументы передачи данных обеспечиваются одной ветвью, а вторая участвует только в синхронизации;

– с пассивным адресатом, когда данные перемещаются из памяти одной ветви в память другой, и только инициатор явно участвует в передаче. При этом два или более инициатора могут взаимодействовать, имея доступ к одним и тем же данным в окне ветви-адресата. Ветвь, владеющая адресным окном, может отличаться от взаимодействующих ветвей, в этом случае она явно не участвует в коммуникациях. Этот режим близок к модели общей (*shared*) памяти, где общие данные могут быть доступны всем ветвям, невзирая на их расположение.

И в том и в другом режиме должна обеспечиваться синхронизация действий ветвей, обеспечивающая единственность периода доступа к любой порции данных в любом окне. Под периодом доступа какой-либо ветви к какому-либо буферу памяти понимается промежуток времени, в течение которого только эта ветвь имеет право читать/изменять данные в этом буфере. Нарушение этого принципа обычно влечет за собой трудно обнаруживаемые ошибки в программе. MPI обеспечивает три механизма синхронизации.

1. Коллективная синхронизация *MPI_Win_fence*. Она обеспечивает простую модель синхронизации, которая часто используется при параллельных вычислениях: а именно слабосинхронную модель (loosely synchronous model), при которой вычислительные фазы перемежаются с фазами комуникаций. Такой механизм более всего пригоден для слабосинхронных алгоритмов, когда граф взаимодействующих процессов меняется очень часто либо когда каждый процесс взаимодействует со многими другими. Этот вид синхронизации используется для коммуникаций с активным адресатом. Периоды доступа начинаются и заканчиваются вызовами функции *MPI_Win_fence*.

2. Четыре функции *MPI_Win_start*, *MPI_Win_complete*, *MPI_Win_post* и *MPI_Win_wait* могут использоваться, чтобы свести затраты на синхронизацию к минимуму. Синхронизируются только пары взаимодействующих процессов, и это происходит только тогда, когда синхронизация необходима, чтобы корректно упорядочить RMA-обращения к окну (с учетом локальных обращений к этому же окну).

3. И последний механизм – блокировки обеспечиваются двумя функциями: *MPI_Win_lock* и *MPI_Win_unlock*. Синхронизация с блокировками полезна для таких приложений, которые эмулируют модель с общей памятью через MPI-вызовы. Типичный пример – модель «доска объявлений», где ветви программы могут читать или обновлять различные части «доски объявлений» в случайные промежутки времени.

Функция коллективной «заборной» синхронизации:

int MPI_Win_fence(int assert, MPI_Win win); – здесь assert задает контекст вызова для возможной внутренней оптимизации коммуникационных процессов MPI путем комбинирования флагов (допустимо отсутствие флагов, т. е. значение 0):

– MPI_MODE_NOSTORE – локальная ветвь не меняла данные окна с момента последней синхронизации. Отменяет необходимость синхронизации кэшей.

– MPI_MODE_NOPUT – данные локального окна не будут изменены вызовами *MPI_Put* и *MPI_Accumulate* до вызова *MPI_Win_wait*; позволяет избежать синхронизации кэшей в момент вызова *MPI_Win_wait*.

– MPI_MODE_NOPRECEDE – до этого вызова не было других вызовов RMA (имеется в виду – после последнего вызова *MPI_Win_fence*). Если одна из ветвей окна устанавливает этот флаг, все остальные ветви группы должны тоже установить его.

– MPI_MODE_NOSUCCEED – после этого вызова не будет других вызовов RMA (до следующей коллективной синхронизации). Если одна из ветвей окна устанавливает этот флаг, то все остальные должны тоже установить его.

Вызов функции *MPI_Win_fence* подразумевает выполнение барьерной синхронизации всех ветвей, входящих в группу окна *win*.

Общая синхронизация с активным адресатом. Начало периода доступа к окну из группы ветвей-инициаторов:

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win); – здесь assert – задает контекст вызова для возможной внутренней оптимизации коммуникационных процессов MPI:

– MPI_MODE_NOCHECK – соответствующий вызов *MPI_Win_post* уже завершился на всех удаленных процессах до момента вызова *MPI_Win_start*. Этот флаг может быть указан в *MPI_Win_start* тогда и только тогда, когда он указан в соответствующем вызове *MPI_Win_post*.

– Значение assert=0 может быть использовано в любой паре связанных вызовов *MPI_Win_start* и *MPI_Win_post*.

Вызов этой функции начинает период RMA-доступа ветвей группы *group* к данным окна *win* с активным адресатом. Каждый адресат должен вызвать функцию *MPI_Win_post* с соответствующими (такими же) аргументами. До тех пор, пока это не будет сделано, любое RMA-обращение к окну-адресату будет задержано.

Завершение периода доступа к окну активного адресата:

int MPI_Win_complete(MPI_Win win); – вызов этой функции завершает период RMA-доступа к окну *win*, начатый вызовом *MPI_Win_start*. Все коммуникационные RMA-вызовы к окну *win*, созданные во время этого периода, завершатся в инициаторе к моменту, когда произойдет возврат из вызова этой функции. *MPI_Win_complete* заставляет завершиться предшествующие RMA-вызовы в инициаторе, но не в адресате. Вызов *MPI_put* или *MPI_accumulate* может еще не выполниться у адресата, в то время как он уже выполнился у инициатора.

Общая синхронизация с активным адресатом. Начало периода доступа к локальному окну в ветви-адресате:

int MPI_Win_post(MPI_Group group, int assert, MPI_Win win); – assert задает контекст вызова для возможной внутренней оптимизации коммуникационных процессов MPI:

– MPI_MODE_NOCHECK – соответствующие функции *MPI_Win_start* еще не были вызваны ни одной из ветвей, намеревающейся осуществить доступ к данным до момента вызова *MPI_Win_post*;

– MPI_MODE_NOSTORE – локальная ветвь не изменила данные окна с момента последней синхронизации. Отменяет необходимость синхронизации кэшей;

– MPI_MODE_NOPUT – данные локального окна не будут изменены вызовами *MPI_Put* и *MPI_Accumulate* до вызова *MPI_Win_wait*;

Начинает период предоставления RMA-доступа для локального окна, связанного с *win*. Только ветви из группы *group* будут иметь доступ к данным окна при помощи RMA-вызовов во время этого периода. Каждая ветвь в группе должна обеспечить соответствующий вызов *MPI_Win_start*. Функция *MPI_Win_post* не блокирует вызывающую ее ветвь программы.

Завершение периода доступа к локальному окну:

int MPI_Win_wait(MPI_Win win); – завершает период предоставления RMA-доступа к окну *win*, начатый вызовом *MPI_Win_post*. Этот вызов соответствует вызовам *MPI_Win_complete*, созданным каждым инициатором, которые имели доступ к окну. Вызов *MPI_Win_wait* будет блокировать ветвь, пока не завершатся все соответствующие вызовы *MPI_Win_complete*. Это гарантирует, что все инициаторы закончили свой RMA-доступ к локальному окну.

Неблокирующая версия функции *MPI_Win_wait*:

*int MPI_Win_test(MPI_Win win, int *flag);* – этот вызов является неблокирующей версией *MPI_Win_wait*. Он возвращает *flag = true*, если

из вызова *MPI_Win_wait* может быть выполнен возврат, и *flag = false* в противном случае. Эффект от возвращения *MPI_Win_test* с *flag = true* такой же, как эффект от возвращения *MPI_Win_wait*. Если же возвращен *flag = false*, тогда у вызова нет видимого эффекта. *MPI_Win_test* должен вызываться только там, где можно вызвать *MPI_Win_wait*. Как только произойдет возврат с кодом *flag = true* для некоторого оконного объекта, *MPI_Win_test* больше не должен вызываться для этого окна, пока оно не будет снова предоставлено для доступа.

Захват окна одной ветвью-инициатором:

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win); – здесь *lock_type* – тип захвата:

– MPI_LOCK_EXCLUSIVE – только для ветви с номером *rank*;

– MPI_LOCK_SHARED – возможен доступ из любой ветви;

assert – задает контекст вызова для возможной внутренней оптимизации коммуникационных процессов MPI;

– MPI_MODE_NOCHECK – ни одна из ветвей не удерживает окно или не будет обращаться к окну во время удерживания блокировки. Это бывает полезно в тех случаях, когда взаимное исключение обеспечивается другими способами, но синхронизация все равно необходима.

Освобождение захваченного окна:

int MPI_Win_unlock(int rank, MPI_Win win); – вызов этой функции завершает период RMA-доступа, начатый вызовом *MPI_Win_lock* (...,*win*). RMA-операции, вызванные во время этого периода, завершатся как в инициаторе, так и в адресате к моменту возврата из этого вызова. Блокировки используются, чтобы защитить обращения к заблокированному окну-адресату, на которое действуют RMA-вызовы, выданные между вызовами *lock* и *unlock*, и чтобы защитить локальные *load*/*store* обращения к заблокированному локальному окну, выполняемые между вызовами *lock* и *unlock*. Обращения, защищенные при помощи эксклюзивной блокировки, не будут пересекаться в пределах окна с другими обращениями к этому же окну, которое защищено блокировкой. Обращения, которые защищены совместной блокировкой, не будут пересекаться в пределах окна с обращениями, защищенными с помощью эксклюзивной блокировки, к одному и тому же окну.

Расширенные коллективные операции.

В стандарте MPI-1 определены коллективные коммуникации для интракоммуникаторов. Стандарт MPI-2 вводит расширения многих коллективных процедур MPI-1 для интеркоммуникаторов и дополни-

тельные процедуры для создания интеркоммуникаторов. Все расширения прозрачны по смыслу и имеют те же самые списки аргументов, что и соответствующие им функции стандарта MPI-1, поэтому детали этих расширений здесь не описываются. При возникновении неясностей в их применении следует обращаться к технической документации по используемой реализации библиотеки MPI.

5.10. Параллельный ввод/вывод

Вначале – определения нескольких понятий, играющих существенную роль в параллельном вводе/выводе.

MPI-файл – это упорядоченная последовательность типизированных блоков данных. Библиотека MPI поддерживает произвольный или последовательный доступ к любому цельному набору этих блоков. Файл открывается коллективно группой ветвей.

Смещение файла – это абсолютное положение байта, относящегося к началу этого файла. Смещение определяет место, с которого начинается текущий вид файла. Существует также понятие типового (ударение на первом слоге) смещения, связанного с применяемым в операциях ввода/вывода типом данных.

Е-тип (элементарный тип данных) – это единица доступа к данным и позиционирования. Е-типом может быть любой определенный в MPI или производный тип данных. Производные е-типы могут быть созданы при помощи любой из подпрограмм создания типов данных MPI, обеспечивающих, чтобы получающиеся типовые смещения были неотрицательными и монотонно неубывающими. Доступ к данным идет в единицах е-типов, считывается или записывается всегда целый блок данных какого-либо из е-типов. Смещения выражаются как количества е-типов; указатели на файл указывают на начала е-типов. В зависимости от контекста термин «е-тип» будет использоваться для обозначения одного из трех аспектов элементарного типа данных:

– непосредственно типа MPI;

– блока данных соответствующего типа;

– размера этого типа.

Файловый тип – это базис для разбиения файла, он определяет шаблон доступа к файлу. Файловый тип – это обычный е-тип или производный тип данных MPI, состоящий из нескольких элементов одного и того же е-типа. Размер любой «дыры» в файловом типе должен быть кратным размеру этого е-типа.

Вид файла определяет текущий набор данных, видимый и доступный из открытого файла как упорядоченный набор е-типов. Каждая ветвь имеет свой вид файла, определенный тремя параметрами: смещением, е-типом и файловым типом. Шаблон, описанный в файловом типе, повторяется, начиная со смещения, чтобы определить вид. Шаблон повторения такой, какой был бы создан функцией MPI_Type_contiguous, если бы в нее были переданы файловый тип и сколь угодно большое число. Виды могут меняться пользователем во время выполнения программы. По умолчанию вид – это линейный поток байтов (смещение равно нулю, е-тип и файловый тип есть MPI_BYTE).

Типовое смещение (или просто смещение) – это позиция в файле относительно текущего вида, представленная как число е-типов.

Размер MPI файла измеряется в байтах от начала файла. Только что созданный файл имеет нулевой размер. Использование размера как абсолютного смещения дает позицию байта, следующего сразу за последним байтом файла. Для любого вида конец файла – это смещение первого е-типа, доступного в данном виде, начинающегося после последнего байта в файле.

Указатель на файл – это постоянное смещение, устанавливаемое MPI. Индивидуальные файловые указатели – файловые указатели, локальные для каждой ветви, открывающей файл.

Дескриптор файла – это закрытый объект, создаваемый функцией *MPI_File_open* и уничтожаемый MPI_*File_close*. Все операции над открытым файлом работают с файлом через его дескриптор.

Обычная последовательность операций каждой ветви при работе с файлами состоит в следующем:

- определение необходимых переменных и типов данных;
- открытие файла;
- установка вида файла;
- запись/чтение;
- для неблокирующих операций – ожидание их завершения;
- закрытие файла.

Открытие файла осуществляется с помощью вызова функции

*int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh);* – эта функция открывает указанный файл и возвращает указатель на него (*fh*) актуальный до закрытия файла. Поддерживаются следующие типы доступа (задаваемые в amode, получаемом при применении операции | (ИЛИ) к следующим целым константам):

- MPI_MODE_RDONLY – только чтение,
- MPI_MODE_RDWR – чтение и запись,

– MPI_MODE_WRONLY – только запись,
– MPI_MODE_CREATE – создавать файл, если он не существует,
– MPI_MODE_EXCL – ошибка, если создаваемый файл уже существует,
– MPI_MODE_DELETE_ON_CLOSE – удалять файл при закрытии,
– MPI_MODE_UNIQUE_OPEN – файл не будет параллельно открыт где-либо еще,
– MPI_MODE_SEQUENTIAL – файл будет доступен лишь последовательно,
– MPI_MODE_APPEND – установить начальную позицию всех файловых указателей на конец файла.

Закрытие файла:

*int MPI_File_close(MPI_File *fh);*

Удаление файла:

*int MPI_File_delete(char *filename, MPI_Info info);* – аргумент *info* может быть использован, чтобы предоставить информацию относительно специфики файловой системы. Константа MPI_INFO_NULL соответствует нулевому *info* и может быть использована в тех случаях, когда дополнительная информация не нужна.

Изменение размера файла:

int MPI_File_set_size(MPI_File fh, MPI_Offset size); – это коллективная операция, т. е. все ветви должны вызывать эту функцию и устанавливать одинаковое значение нового размера.

Резервирование памяти под файл:

int MPI_File_preallocate(MPI_File fh, MPI_Offset size); – коллективная операция, все ветви в группе должны устанавливать одно и то же значение *size*. Области файла, записанные ранее, не изменяются. На новые области файла, располагаемые в памяти, эта функция производит тот же эффект, как и запись неопределенных данных. Если *size* больше, чем текущий размер файла, размер файла увеличивается до *size*. Если *size* меньше либо равен текущему размеру файла, размер файла не изменяется.

Получение размера файла:

*int MPI_File_get_size(MPI_File fh, MPI_Offset *size);*

Изменение вида файла выполняется путем вызова функции

*int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info);* – изменяет вид данных файла для каждой ветви. Начало вида устанавливается в *disp* (ненулевое значение *disp* может использоваться, например, для пропуска заголовка файла при чтении только данных), тип данных

устанавливается в *etype*, распределение данных по ветвям в *filetype*, название этого вида представления данных – строка *datarep*. Кроме того, функция сбрасывает все индивидуальные и общие файловые указатели в 0. *MPI_File_set_view* – коллективная операция, значения *datarep* и размеры е-типов в представлении данных должны совпадать во всех ветвях в группе, а значения *disp*, *filetype* и *info* могут быть различны. Типы данных, передаваемые в *etype* и *filetype*, должны быть согласованы (размер *filetype* должен делиться на размер *etype*).

Текущий вид файла может быть получен с помощью функции

*int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep);* – возвращает вид файла. Текущее значение смещения возвращается в *disp*. *etype* и *filetype* – это новые типы данных с картами типов, равными картам типов текущих е-типа и файлового типа соответственно. Название представления данных возвращается в *datarep*. Строка *datarep* должна быть достаточно велика, чтобы вместить возвращаемую строку названия представления данных. Длина этой строки ограничена значением MPI_MAX_DATAREP_STRING.

Доступ к данным файла может осуществляться:

– по явным смещениям;
– по индивидуальным указателям;
– по общим указателям.

При этом могут использоваться разные способы синхронизации:

– блокирующие;
– неблокирующие.

И, наконец, операции чтения/записи могут быть:

– индивидуальные;
– коллективные.

Соответствие между функциями доступа и способами позиционирования файлов, синхронизации и координации ветвей приведено в следующей таблице.

Позиционирование	Синхронизация	Координация ветвей	
		Неколлективные	Коллективные
Явные смещения	Блокирующие	*MPI_File_read_at*	*MPI_File_read_at_all*
		MPI_File_write_at	*MPI_File_write_at_all*
	Неблокирующие	*MPI_File_iread_at*	*MPI_File_read_at_all_begin*
			MPI_File_read_at_all_end
		MPI_File_iwrite_at	*MPI_File_write_at_all_begin*
			MPI_File_write_at_all_end

Окончание таблицы

Позиционирование	Синхронизация	Координация ветвей	
		Неколлективные	Коллективные
Индивидуальные указатели	Блокирующие	*MPI_File_read*	*MPI_File_read_all*
		MPI_File_write	*MPI_File_write_all*
	Неблокирующие	*MPI_File_iread*	*MPI_File_read_at_all_begin*
			MPI_File_read_at_all_end
		MPI_File_iwrite	*MPI_File_write_at_all_begin*
			MPI_File_write_at_all_end
Общие указатели	Блокирующие	*MPI_File_read_shared*	*MPI_File_read_ordered*
		MPI_File_write_shared	*MPI_File_write_ordered*
	Неблокирующие	*MPI_File_iread_shared*	*MPI_File_read_ordered_begin*
			MPI_File_read_ordered_end
		MPI_File_iwrite_shared	*MPI_File_write_ordered_begin*
			MPI_File_write_ordered_end

Доступ к данным с явным указанием смещения

Если при открытии файла был определен тип доступа MPI_MODE_SEQUENTIAL, то вызов любой из функций, использующих явное смещение, приведет к аварийному завершению всей параллельной программы.

Блокирующее чтение данных из файла с явным указанием смещения выполняется функцией:

*int MPI_File_read_at(MPI_File fh, MPI_Offset offset,void *buf, int count, MPI_Datatype datatype, MPI_Status *status);* – аргументы *fh* и *offset* определяют местоположение читаемых данных, аргументы *buf*, *count* и *datatype* – буфер в памяти, в который они должны быть прочитаны. Возврат из функции выполняется после завершения операции, структура *status* в этот момент содержит сведения о том, как завершилась операция.

Коллективная версия этой функции, которую должны вызывать все ветви группы:

*int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);* – все аргументы имеют точно такой же смысл, но разные ветви могут задавать разные их значения (в том числе, разные смещения).

Неблокирующая функция чтения данных выглядит так:

*int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);* – вместо аргумента *status* здесь указывается квитанция *request*, используемая точно так же, как и в функциях коммуникаций.

Совершенно аналогичны функции, используемые для записи данных в файл (и их аргументы имеют точно такое же назначение):

*int MPI_File_write_at(MPI_File fh, MPI_Offset offset,void *buf, int count, MPI_Datatype datatype, MPI_Status *status);*

*int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);*

*int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);*

Доступ к данным с индивидуальными файловыми указателями

Интерфейс *MPI* поддерживает по одному индивидуальному файловому указателю в каждой ветви на каждый дескриптор файла. Текущее значение этого указателя неявно определяет смещение для соответствующих функций доступа к данным. Эти функции используют и обновляют только индивидуальные файловые указатели, общие файловые указатели не используются и не обновляются. Вызовы функций с использованием индивидуальных файловых указателей имеют ту же семантику, что и доступ к данным через функции с указанием точного смещения со следующим изменением: фактическое смещение каждой операции определяется текущим значением индивидуального файлового указателя.

После того как была инициирована операция с файловым указателем, индивидуальный файловый указатель изменяется так, чтобы указывать на следующий е-тип после последнего, к которому был осуществлен доступ. Файловый указатель изменяется относительно текущего вида файла.

Если при открытии файла был определен тип доступа MPI_MODE_SEQUENTIAL, то вызов любой из таких функций приведет к аварийному завершению параллельной программы.

Блокирующее чтение:

*int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);* – все аргументы имеют тот же смысл, что у функции *MPI_File_read_at* отсутствует аргумент *offset*, вместо значения которого используется индивидуальный файловый указатель.

Коллективная версия блокирующего чтения:

*int MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);* – все аргументы имеют точно такой же смысл, но разные ветви могут задавать разные их значения.

Неблокирующее чтение:

*int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);* – как обычно, для неблокирующей операции аргумент *status* для получения сведений о завершении заменяется на квитанцию *request*.

Операции, осуществляющие передачу данных в противоположном направлении, выполняются с помощью вызовов функций:

*int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);*

*int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);*

*int MPI_File_iwrite(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);*

При использовании операций ввода/вывода с индивидуальными файловыми указателями возможны ситуации, когда требуется явно модифицировать указатель, т. е. прочитать/записать данные по нужному файловому смещению. Для этого библиотека MPI предоставляет специальные функции, позволяющие вычислять значения смещений и заносить их в файловый указатель

int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence); – изменяет индивидуальный файловый указатель относительно аргумента *whence*, который может иметь следующие значения:

– MPI_SEEK_SET: указатель устанавливается в *offset*.

– MPI_SEEK_CUR: указатель устанавливается в текущую позицию плюс *offset*.

– MPI_SEEK_END: указатель устанавливается в конец файла плюс *offset*.

Значение *offset* может быть отрицательным, что позволяет вести поиск в обратном направлении, однако отрицательные значения указателя приведут к аварийному завершению программы.

Функция:

*int MPI_File_get_position(MPI_File fh, MPI_Offset *offset);* – возвращает в аргументе *offset* текущую позицию индивидуального файлового указателя в единицах е-типа относительно текущего вида. Полученное смещение может быть использовано при дальнейших вызовах

MPI_File_seek (с использованием *whence* = MPI_SEEK_SET), для возврата к текущей позиции. Чтобы установить смещение файлового указателя, сначала необходимо преобразовать *offset* в абсолютную позицию байта, используя *MPI_File_get_byte_offset* (см. следующий абзац), а затем вызвать *MPI_File_set_view* с получившимся смещением.

Функция:

*int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp*); – преобразует смещение относительно вида в абсолютную позицию байта. Абсолютная позиция байта (от начала файла) с данным смещением *offset* относительно текущего вида *fh* возвращается в *disp*.

Доступ к данным с общими указателями

MPI поддерживает ровно один общий указатель для каждого файла, открытого с использованием коллективного вызова *MPI_file_open* (общий для процессов в группе коммуникатора). Текущее значение этого указателя неявно определяет смещение в соответствующих функциях доступа к данным. Эти функции используют и обновляют только общие файловые указатели, поддерживаемые MPI. Индивидуальные файловые указатели не используются и не обновляются. Операции с использованием общих файловых указателей имеют ту же семантику, что и доступ к данным через функции с указанием точного смещения со следующими изменениями:

– смещение определяется текущим значением общего файлового указателя, поддерживаемого MPI;

– эффект множественных вызовов функций с использованием общих файловых указателей определен так, чтобы вести себя, как если бы эти вызовы были преобразованы в последовательную форму (сериализованы);

– использование этих функций возможно только в том случае, когда все процессы используют один и тот же вид файла.

Для неколлективных подпрограмм, работающих с общими файловыми указателями, невозможно определить порядок сериализации. Для того чтобы обеспечить особый порядок выполнения, пользователю необходимо использовать другие средства синхронизации.

После того как инициирована операция с общим файловым указателем, общий файловый указатель обновляется, чтобы указывать на следующий за последним е-типом, к которому был осуществлен доступ, е-тип. Файловый указатель обновляется относительно текущего вида файла.

Неколлективные операции:

*int MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);* – читает файл, используя общий файловый указатель.

*int MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);* – неблокирующая версия чтения.

*int MPI_File_write_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);* – пишет данные в файл, используя общий файловый указатель.

*int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);* – неблокирующая версия записи.

К о л л е к т и в н ы е о п е р а ц и и

Семантика коллективного доступа с использованием общих файловых указателей состоит в том, что обращения к файлу будут идти в порядке, определяемом рангами ветвей в группе. Для каждой ветви место в файле, в котором будет осуществлен доступ к данным, будет позицией общего файлового указателя, после того как все ветви, имеющие меньший ранг в группе, чем данный, осуществят доступ к данным. Кроме того, чтобы последующие обращения той же ветви к файлу не перекрывались с коллективным обращением, вызов должен завершаться только после того, как все ветви в группе инициировали свои обращения. Когда вызов завершается, общий файловый указатель указывает на следующий доступный согласно виду файла, используемому всеми процессами, е-тип, следующий после последнего запрошенного е-типа.

*int MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);* – это коллективная версия функции *MPI_File_read_shared*.

*int MPI_File_write_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) ;* – это коллективная версия функции *MPI_File_write_shared*.

Как и для операций ввода/вывода с индивидуальными файловыми указателями библиотека MPI при использовании операций ввода/вывода с общими указателями предоставляет специальные функции, позволяющие вычислять значения смещений и модифицировать на их основе файловый указатель:

MPI_File_Seek_shared(MPI_Offset offset, int whence); – изменяет общий файловый указатель относительно аргумента *whence*, значения которого описывались при описании функции *MPI_File_seek. MPI_File_Seek_shared* – коллективная операция; все ветви в группе коммуникатора, ассоциированной с дескриптором файла *fh*, должны вызывать ее с одинаковыми значениями *offset* и *whence*. Аргумент *offset* может быть отрицательным, что позволяет осуществлять поиск в обратном направлении.

*int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset);* – возвращает в аргументе *offset* текущую позицию индивидуального файлового указателя в единицах е-типа относительно текущего вида.

Разделенные коллективные функции доступа к данным.

MPI предоставляет ограниченную форму «неблокирующих коллективных» операций ввода-вывода для всех типов доступа к данным – использование разделенных коллективных функций доступа к данным. Они называются «разделенными», потому что обычные коллективные разделены на две: начинающая функция и заканчивающая функция. Начинающая функция начинает операцию почти так же, как неблокирующее обращение к данным (например, *MPI_file_iread*). Заканчивающая функция завершает операцию почти так же, как тестирование или ожидание (например, *MPI_Wait*). Как и в случае неблокирующих операций, пользователь не должен использовать буфер, переданный начинающей функцией, пока она ожидает обработки; операция должна быть завершена заканчивающей функцией, прежде чем можно будет безопасно освобождать буферы, и т. д.

Разделенные коллективные операции доступа к данным над дескриптором файла *fh* подчиняются следующим семантическим правилам.

– В любой ветви MPI каждый дескриптор файла в любой момент может иметь не более одной активной разделенной коллективной операции.

– Начинающие вызовы коллективны для группы ветвей, участвовавших в коллективном открытии, и подчиняются правилам порядка выполнения для коллективных вызовов.

– Заканчивающие вызовы коллективны для группы ветвей, участвовавших в коллективном открытии, и подчиняются правилам порядка выполнения для коллективных вызовов. Каждый заканчивающий вызов соответствует предшествующему начинающему вызову той же коллективной операции. Когда делается заканчивающий вызов, ему

должен предшествовать ровно один начинающий вызов той же операции, не имеющий пары.

– Начинающий и заканчивающий вызовы предоставлены для того, чтобы позволить пользователю и библиотеке MPI оптимизировать коллективные операции.

– Разделенные коллективные операции неаналогичны соответствующим обычным коллективным операциям. Например, операция простого коллективного чтения *MPI_File_read_all* в одной ветви неаналогична паре *MPI_File_read_all_begin*/ *MPI_File_read_all_end* другой ветви.

– Разделенные коллективные операции должны определять буфер и в начинающей и в завершающей функции.

– Коллективные операции ввода-вывода над дескриптором файла недопустимы одновременно с разделенными коллективными операциями доступа над этим дескриптором (т. е. между началом и завершением доступа).

– В многопоточной реализации любые разделенные коллективные начинающая и завершающая операции, вызываемые ветвью, должны вызываться из одного потока.

Аргументы этих функций имеют те же значения, что и для эквивалентных коллективных версий (например, аргументы *MPI_File_read_all_begin* и *MPI_File_read_all_end* эквивалентны аргументам функции *MPI_File_read_all*). Начинающая функция (например, *MPI_File_read_all_begin*) начинает разделенную коллективную операцию, которая после завершения соответствующей оканчивающей подпрограммой (т. е. *MPI_File_read_all_end*) производит тот же результат, что и эквивалентная коллективная подпрограмма (т. е. *MPI_File_read_all*). В целях обеспечения семантики согласованности пара разделенных коллективных операций доступа к данным (например, *MPI_File_read_all_begin* и *MPI_File_read_all_end*) образует одно простое обращение к данным.

5.11. Использование библиотеки MPE для анализа процессов взаимодействия ветвей программы

Библиотека MPE содержит программу визуализации лог-файлов Jumpshot-4 и набор функций, позволяющих создавать эти лог-файлы и записывать в них «трассу» исполнения параллельной MPI-программы.

Приложение Jumpshot-4 позволяет представить работу параллельной программы в наиболее удобном для восприятия человека виде –

визуализировать. Другими словами, программа Jumpshot-4 – это средство для визуализации трассы параллельной программы. Трасса – это журнал, содержащий информацию о ходе выполнения программы. Делает это она при помощи так называемых лог-файлов. Лог-файл – это файл трассы отмеченных во времени событий. Формат лог-файла, используемого Jumpshot-4, называется SLOG2.

Основная визуальная компонента Jumpshot-4 – канва с двумя измерениями: по горизонтали откладывается время, а по вертикали – процессы параллельной программы; канву можно масштабировать и прокручивать в горизонтальном и вертикальном направлениях.

Таким образом, каждая точка на канве может быть идентифицирована по двум числам – отметке времени и ID-линии времени.

Графические объекты, которые содержатся в SLOG-2 файле, рисуются на канве этой координатной системы. Эти объекты бывают двух типов: простые и составные. Простые объекты, как правило, являются основными объектами SLOG-2 файла. Есть деление этих объектов, основанное на их топологической структуре.

Формат SLOG-2 поддерживает три вида топологий: состояние, стрелка и событие. Состояние и стрелка определяются двумя точками на канве – это соответственно координаты: отметка времени и идентификатор (ID) линии времени. Начальный ID линии времени для состояния совпадает с конечным ID (т. е. этот объект относится к одному процессу), а для стрелки начальный и конечный ID могут различаться. Событие состоит только из одной точки на канве, т. е. имеет только одну линию времени и одно значение времени. Можно сказать, что состояние – это промежуток времени между двумя событиями.

Составные объекты имеют более сложную структуру и конструируются из набора простых объектов.

Лог-файлы и библиотека MPE

Исходные данные для программы Jumpshot содержатся в лог-файлах, которые можно получить путем включения в параллельную программу вызовов функций библиотеки MPE (Multi-Processing Environment). Она содержит процедуры, которые облегчают написание, отладку и оценку эффективности MPI-программ. Рассмотрим структуру этой библиотеки чуть подробнее.

MPE-процедуры делятся на несколько категорий.

- Параллельная графика (Parallel X graphics). Эти процедуры обеспечивают доступ всем процессам к разделяемому X–дисплею. Они со-

здают удобный вывод для параллельных программ, позволяют чертить текст, прямоугольники, круги, линии и т. д.

- Регистрация (Logging). Библиотека MPE создает возможность легко получить лог-файл в каждом процессе и собрать их вместе по окончанию работы. Она также автоматически обрабатывает рассогласование и дрейф часов на множественных процессорах, если система не обеспечивает синхронизацию часов. Это библиотека для пользователя, который желает создать свои собственные события и программные состояния.
- Последовательные секции (Sequential Sections). Иногда секция кода, которая выполняется на ряде процессов, должна быть выполнена только по одному процессу за раз в порядке номеров этих процессов. MPE имеет функции для такой работы.
- Обработка ошибок (Error Handling). MPI имеет механизм, который позволяет пользователю управлять реакцией реализации на ошибки времени исполнения, включая возможность создать свой собственный обработчик ошибок.

Далее будут преимущественно рассмотрены средства регистрации (logging) для анализа эффективности параллельных программ. Анализ результатов регистрации производится после выполнения вычислений. Средства регистрации и анализа включают ряд профилирующих библиотек, утилитных программ и ряд графических средств.

Первая группа средств − профилирование. Библиотечные ключи обеспечивают собрание процедур, которые создают лог-файлы. Эти лог-файлы могут быть созданы вручную путем размещения в программе MPI обращений к MPE, или автоматически при установлении связи с соответствующими MPE–библиотеками (например, откомпилировав программу с библиотекой mpich2mpe.lib), или комбинацией этих двух методов. В настоящее время MPE предлагает следующие три профилирующие библиотеки.

1. Tracing Library (библиотека трассирования) – трассирует все MPI-вызовы. Каждый вызов предваряется строкой, которая содержит номер вызывающего процесса в MPI_COMM_WORLD и сопровождается другой строкой, указывающей, что вызов завершился. Большинство процедур SEND и RECIEVE также указывают значение count, tag и имена процессов, которые посылают или принимают данные.

2. Animation Libraries (анимационная библиотека) – простая форма программной анимации в реальном времени, которая требует процедур X–окна.

3. Logging Libraries (библиотека регистрации) – самые полезные и широко используемые профилирующие библиотеки в MPE. Они формируют базис для генерации лог-файлов из пользовательских программ. Имеется три различных формата лог-файлов, допустимых в MPE. Формат CLOG содержит совокупность событий с единым отметчиком времени. Формат ALOG больше не развивается и поддерживается для обеспечения совместимости с ранними программами. И наиболее мощным является формат SLOG (для Scalable Logfile), который может быть конвертирован из уже имеющегося CLOG-файла или получен прямо из выполняемой программы (для этого необходимо установить переменную среды MPE_LOG_FORMAT в SLOG).

Набор утилитных программ в MPE включает конверторы лог-форматов (например, clog2slog), печать лог-файлов (slog_print), оболочки средств визуализации лог-файлов, которые выбирают корректные графические средства для представления лог-файлов в соответствии с их расширениями.

Далее будут рассмотрены только библиотеки регистрации Logging Libraries. Оценки времени выполнения взаимодействий дают некоторое представление об эффективности программы. Но в большинстве случаев нужно подробно узнать, какова была последовательность событий, сколько времени было потрачено на каждую стадию вычисления и сколько времени занимает каждая отдельная операция передачи. Чтобы облегчить их восприятие, нужно представить их в графической форме.

Но для этого сначала нужно создать файлы событий со связанными временными отметками, затем исследовать их после окончания программы и только затем интерпретировать их графически на рабочей станции. Такие файлы ранее уже названы лог-файлами. Способность автоматически генерировать лог-файлы является важной компонентой всех средств для анализа эффективности параллельных программ.

Опишем некоторые простые инструментальные средства для создания лог-файлов и их просмотра. Библиотека для создания лог-файлов отделена от библиотеки обмена сообщениями MPI. Просмотр лог-файлов независим от их создания и поэтому могут использоваться различные инструментальные средства. Библиотека для создания лог-файлов MPE разработана таким образом, чтобы сосуществовать с любой MPI-реализацией и распространяется наряду с модельной версией MPI.

Для того чтобы создать файл регистрации, необходимо вызвать процедуру MPE_Log_event. Кроме того, каждый процесс должен вы-

звать процедуру MPE_Init_log, чтобы приготовиться к регистрации, и MPE_Finish_log, чтобы объединить файлы, сохраняемые локально при каждом процессе в единый лог-файл. MPE_Stop_log используется, чтобы приостановить регистрацию, хотя таймер продолжает работать. MPE_Start_log возобновляет регистрацию.

Программист выбирает любые неотрицательные целые числа, желательные для типов событий; система не придает никаких частных значений типам событий. Чтобы присвоить событию свой номер, нужно вызвать функцию MPE_Log_get_event_number. События рассматриваются, как не имеющие продолжительности. Чтобы измерить продолжительность состояния программы, необходимо, чтобы пара событий отметила начало и окончание состояния. Состояние определяется процедурой MPE_Describe_state, которая описывает начало и окончание типов событий. Процедура MPE_Describe_state также добавляет название состояния и его цвет на графическом представлении. Соответствующая процедура MPE_Describe_event обеспечивает описание события каждого типа.

Приведем прототипы описанных выше функций:

int MPE_Init_log (void)
int MPE_Start_log (void)
int MPE_Stop_log (void)
*int MPE_Finish_log (char *logfilename)*
int MPE_Log_get_event_number (void)
*int MPE_Describe_state (int start, int end, char *name, char *color)*
*int MPE_Describe_event (int event, char *name)*
*int MPE_Log_event (int event, int intdata char *chardata)*

Как уже было сказано, создать лог-файл можно и автоматически, откомпилировав проект с библиотекой mpich2mpe.lib (конечно, если при написании использовалась реализация MPICH2, аналогичная библиотека в MPICH, к сожалению, неизвестна). В этом случае в папке проекта создается файл *.clog2, который содержит всю трассу программы. В случае, если программист хочет какой-то «кусок» программы проанализировать отдельно или же просто создать свой лог-файл, имеет смысл использовать процедуры библиотеки MPE непосредственно в коде программы.

Рассмотрим, как это делается на примере простейшей программы (аналог «HelloWorld!»), которая печатает ранги всех процессов, на которых она выполнялась.

```
#include "stdafx.h"
#include "C:\Program Files\MPICH2\include\mpi.h"
#include "C:\Program Files\MPICH2\include\mpe.h"
int _tmain(int argc, char* argv[])
{
    int ProcsNum,ProcRank;
    int event1a,event1b;
    int event2a,event2b;
    MPI_Init(&argc,&argv);
    MPE_Init_log();
    event1a = MPE_Log_get_event_number();
    event1b = MPE_Log_get_event_number();
    event2a = MPE_Log_get_event_number();
    event2b = MPE_Log_get_event_number();
    MPE_Describe_state(event1a,event1b,"GetSize","magenta");
    MPE_Describe_state(event2a,event2b,"GetRank","green");
    MPE_Start_log();
    MPE_Log_event(event1a,0,"start size");
    MPI_Comm_size(MPI_COMM_WORLD,&ProcsNum);
    MPE_Log_event(event1b,0,"finish size");
    MPE_Log_event(event2a,0,"start rank");
    MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
    MPE_Log_event(event2b,0,"finish rank");
    printf("Hello from process %d\n",ProcRank);
    MPE_Finish_log("logfile1");
    MPI_Finalize();
    return 0;
}
```

Вначале идет подготовка к регистрации (MPE_Init_log), после которой начинается блок описаний. В данной программе всего две функции MPI – получение размера системы и ранга процесса; предположим, мы хотим описать их обе.

Для этого сначала определяются события. Поскольку каждое состояние (т. е. функция MPI) – это время между двумя событиями, то нам нужно определить четыре события. Для удобства лучше, чтобы названия событий, относящихся к одному состоянию, не сильно отличались (можно их назвать, например, event1a и event1b). Функция MPE_Log_get_event_number присваивает каждому событию уникальный номер.

Далее описываются состояния: в данном случае состояние с названием «GetSize» расположено между событиями event1a и event1b и на канве в Jumpshot-4 будет отображаться малиновым цветом. Со вторым – аналогично.

Регистрация начинается путем вызова функции MPE_Start_log. Считается, что события eventXa относятся к началу состояния, eventXb – к концу. Регистрируется определение размера системы (количества процессов). Для этого до и после соответствующей функции вызывается функция MPE_Log_event.

И, наконец, вызывается функция MPE_Finish_log, чтобы закончить регистрацию и закрыть лог-файл logfile.clog2.

Чтобы иметь возможность просмотреть созданный лог-файл в программе Jumpshot-4, нужно конвертировать его в logfile.slog2. Соответствующая утилита входит в состав приложения Jumpshot-4.

Контрольные вопросы

1. Какие виды виртуальных топологий можно реализовывать с использованием MPI?
2. Что такое барьерная синхронизация в MPI? Какие еще виды синхронизации существуют в стандарте MPI-1?
3. Перечислите основные группы функций стандарта MPI-1.
4. Может ли MPI-программа, выполняющаяся на многоядерном узле, использовать все его ядра? и если да, то каким образом?
5. Что такое пользовательская операция редукции?
6. Чем различаются размер и протяженность типов данных MPI?
7. Что такое буферизованная передача сообщений?
8. Что такое коммуникатор? Какие проблемы решаются с использованием этого понятия?
9. Сколько в MPI операций приема сообщений типа «точка-точка»? Перечислите и охарактеризуйте их.
10. Что такое коллективное взаимодействие ветвей программы?
11. Как путем вызова одной функции можно одновременно передать и принять блок данных?
12. Перечислите предопределенные операции редукции данных MPI.
13. Что такое виртуальная топология? Как она связана с физической топологией вычислительной сети/кластера/комплекса?
14. Для чего в MPI-программах можно использовать производные типы данных?

15. Каково назначение библиотеки MPE?

16. Что понимается под операциями сдвига и циклического сдвига данных? Какие функции обычно используются для их реализации?

17. Что такое исключающая редукция (MPI_Exscan), как она выполняется?

18. Может ли ветвь параллельной программы с помощью MPI передавать сообщения сама себе?

19. Какие операции рассылки и сбора данных реализованы в MPI?

20. Должны ли абсолютно все ветви параллельной программы принимать участие в коллективном взаимодействии?

21. Сколько конструкторов производных типов реализовано в MPI? Перечислите и охарактеризуйте их.

22. Чем различаются топологии «тор» и «решетка»?

23. Что такое вид файла? Для чего используется это понятие?

24. Какие способы синхронизации используются при удаленном доступе к памяти?

25. Что такое интеркоммуникатор? Чем это понятие отличается от понятия интракоммуникатора?

26. Что такое файловый тип?

27. Что такое «окно» при удаленном доступе памяти? Как создать окно?

28. Что такое синхронизация с блокировками при удаленном доступе к памяти? Какие еще существуют виды синхронизации?

29. Что такое смещение файла? Чем оно отличается от типового смещения?

30. Какие функции могут быть использованы для модификации состояния памяти «окна» другой ветви?

31. Перечислите основные группы функций библиотеки MPE.

32. Чем различаются индивидуальные и общие указатели файлов?

33. Что такое е-тип? Может ли в качестве е-типа использоваться базовый тип?

34. Что такое заборная синхронизация?

35. Какие типы доступа к файлам существуют в MPI?

36. С помощью каких функций выполняется создание новых групп ветвей?

37. Что такое синхронизация с пассивным адресатом?

38. Что такое доступ к файлу по явному смещению? Являются ли операции такого доступа коллективными?

39. Можно ли изменять вид файла в процессе выполнения программы?

40. С помощью каких функций выполняется захват/освобождение окна при удаленном доступе к памяти?

41. Перечислите блокирующие неколлективные функции доступа к данным файла с использованием общих указателей.

42. Для чего и как используются функции *MPI_Win_start*, *MPI_Win_complete*, *MPI_Win_post* и *MPI_Win_wait*?

43. Как выполняется синхронизация при неблокирующем коллективном доступе к данным файла?

44. Как одной операцией прочитать из файла несколько целых чисел, расположенных на одинаковом (ненулевом) расстоянии друг от друга?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Воеводин В.В.* Параллельные вычисления / В.В. Воеводин, Вл.В. Воеводин. – СПб.: БХВ-Петербург, 2004.

2. *Богачев К.Ю.* Основы параллельных вычислений / К.Ю. Богачев. – М.: Бином. Лаборатория знаний, 2010.

3. *Гергель В.П.* Теория и практика параллельных вычислений / В.П. Гегель. – М.: Бином. Лаборатория знаний, 2007.

4. *Гергель В.П.* Основы параллельных вычислений для многопроцессорных вычислительных систем / В.П. Гергель, Р.Г. Стронгин. – Н.Новгород.: Изд-во ННГУ, 2003.

5. *Баканов В.М.* Параллельные вычисления / В.М. Баканов. – М.: Изд-во МГУПИ, 2006.

6. *Малышкин В.Э.* Параллельное программирование мультикомпьютеров / В.Э. Малышкин, В.Д. Корнеев. – Новосибирск, 2006.

7. *Миллер Р*. Последовательные и параллельные алгоритмы: общий подход / Р. Миллер, Л. Боксер. – М.: Бином. Лаборатория знаний, 2006.

8. *Лупин С.А*. Технологии параллельного программирования. Серия: Высшее образование / С.А. Лупин, М.А. Посыпкин. – М.: Форум, Инфра-М, 2008.

9. *Антонов А.С.* Параллельное программирование с использованием технологии OpenMP / А.С. Антонов. – М.: Изд-во МГУ, 2009.

10. *Левин М.П.* Параллельное программирование с использованием OpenMP/ М.П. Левин. – М.: Бином. Лаборатория знаний, 2008.

11. Официальный сайт OpenMP Architecture Review Board – URL: http://openmp.org/wp/ (дата обращения 06.10.2014).

12. *Боресков А.В.* Основы CUDA. – URL: http://www.steps3d.narod.ru/tutorials/cuda-tutorial.html (дата обращения 06.10.2014).

13. *Фролов В.* Введение в технологию CUDA. – URL: http://cgm.computergraphics.ru/issues/issue16/cuda (дата обращения 06.10.2014).

14. *Осколков И.* NVIDIA CUDA – доступный билет в мир больших вычислений / И. Осколков. – URL: http://www.computerra.ru/interactive/423392/ (дата обращения 06.10.2014).

15. Официальный сайт http://www.nvidia.com/object/cuda_ home_ new.html (дата обращения 06.10.2014).

16. *Корнеев В.Д.* Параллельное программирование в MPI / В.Д. Корнеев. – Москва-Ижевск: Институт компьютерных исследований, 2003.

17. *Антонов А.С.* Параллельное программирование с использованием технологии MPI / А.С. Антонов. – М.: Изд-во МГУ, 2004.

18. Сайт форума разработчиков MPI – URL: http://www.mpi-forum.org/ (дата обращения 06.10.2014).

ОГЛАВЛЕНИЕ

Малявко Александр Антонович

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ
НА ОСНОВЕ ТЕХНОЛОГИЙ OpenMP, MPI, CUDA

Учебное пособие

Редактор *Л.Н. Ветчакова*
Выпускающий редактор *И.П. Брованова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 03.03.2015. Формат 60 × 84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 6,74. Печ. л. 7,25. Изд. № 264/14. Заказ № 376. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20