Нижегородский государственный университет им. Н.И. Лобачевского
Национальный исследовательский университет

Программа повышение конкурентоспособности
ННГУ им. Н.И. Лобачевского

Стратегическая инициатива 7 «Достижение лидирующих позиций в области суперкомпьютерных технологий и высокопроизводительных вычислений»

Основная образовательная программа
01.03.03 – Механика и математическое моделирование

Учебное пособие по дисциплине
Численное моделирование и вычислительный эксперимент

Игумнов Л.А., Воробцов И.В., Литвинчук С.Ю.

ПОДГОТОВКА НАУЧНО-ИССЛЕДОВАТЕЛЬСКИХ ПРИЛОЖЕНИЙ К ИСПОЛЬЗОВАНИЮ НА ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ СИСТЕМАХ

Нижний Новгород
2014 год

УДК 004.9
ББК 30.2

И26 **Игумнов Л.А., Воробцов И.В., Литвинчук С.Ю. Подготовка научно-исследовательских приложений к использованию на высокопроизводительных системах** – Н. Новгород, Нижегородский госуниверситет, 2014. – 104 с.

ISBN

В работе рассмотрены основы разработки параллельных приложений для эффективного выполнения на кластере. Представлены базовые понятие параллельных вычислений и существующих архитектур, а так же классическая схема распараллеливания, при которой MPI используется для распределения работы между множеством вычислительных узлов, а OpenMP используется для реализации параллелизма в рамках каждого узла кластера. Кроме того, дается подробный обзор рассматриваемых моделей программирования и приводится пример подготовки приложения для запуска на кластере.

Учебное пособие предназначено для аспирантов и студентов, и поможет не только начать создание приложений для высокопроизводительных систем, но и успешно портировать уже существующие разработки.

Ответственный редактор Л.А. Игумнов

ISBN

ББК 30.2

СОДЕРЖАНИЕ

Введение

В последнее время использование высокопроизводительных систем для решения научно-исследовательских задач стремительно растёт. Для эффективного использования имеющегося аппаратного потенциала необходима адаптация существующего ПО для выполнения на высокопроизводительных системах. Эта задача может оказаться весьма нетривиальной, учитывая тот факт, что потребуется изменять и перерабатывать исходный код, переход на другую ОС, при стремлении сохранить численные результаты. В данном методическом пособии будут рассмотрены основы параллельного программирования с использованием наиболее распространенных на сегодняшний день стандартов OpenMP и MPI как для систем с общей, так и для систем с распределенной памятью. Кроме того будет приведен пример подготовки существующего приложения, написанного на языке программирования Фортран, для последующего запуска на системе с распределенной памятью – кластере ННГУ.

Параллельные вычисления

Параллельные вычисления – способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). При этом в понятие включена не только совокупность вопросов параллелизма в программировании, но и создание эффективно действующих аппаратных реализаций.

Основная идея параллельных вычислений основана на том, что большинство существующих задач может быть разбито на набор меньших задач, которые могут быть решены одновременно. В отличие от последовательных алгоритмов, параллельные вычисления чаще всего требуют определенной координации действий, связанных с работой над общими данными. При этом параллельные вычисления можно разделить на несколько уровней:

1. **Инструкционный параллелизм**

   Любое приложение, по сути, представляет собой набор инструкций, выполняемых процессором. При определенных условиях их порядок можно менять, распределять по группам, которые будут выполняться параллельно, без изменения результатов работы всего приложения. Данный приём

известен как параллелизм на уровне инструкций, и реализован на аппаратном уровне.

Все современные процессоры имеют многоступенчатый конвейер команд и каждой ступени конвейера соответствует определённое действие, выполняемое процессором в этой инструкции на этом этапе. Процессор с K ступенями конвейера может иметь одновременно до K различных инструкций на разном уровне законченности.

В дополнение к конвейеризации, процессоры способны выполнять несколько инструкций одновременно при помощи суперскалярности. В этом случае инструкции сгруппированы вместе для параллельного выполнения, при условии отсутствия зависимости по данным.

Такой вид параллелизма практически неконтролируем разработчиком и по умолчанию используется всеми существующими приложениями.

2. Параллелизм по данным

Основная идея заключается в выполнении одной операции не над скаляром – одним элементом массива, а над несколькими элементами массива данных - вектором. Данный тип параллелизма обеспечивается как средствами разработки, так и аппаратными возможностями. Векторизация чаще всего выполняется уже на этапе компиляции – перевода исходного текста программы в машинные команды. Различные процессоры поддерживают работу с данными в рамках одной инструкции, но длина вектора может изменяться, в зависимости от используемых инструкций и их и поддержки в процессоре.

Роль разработчика сводится к заданию настроек векторной оптимизации компилятору, использованию директив для компиляции, и других способов реализации векторизации.

3. Параллелизм по задачам

В данном типе параллелизма вычислительная задача разбивается на несколько относительно самостоятельных подзадач, а каждый процессор загружается своей собственной подзадачей. В современных процессорах, параллелизм по задачам подразумевает загрузку всех его ядер.

Существует множество различных моделей параллельного программирования, дающих возможность реализовать параллелизм по задачам.

Наиболее большее распространение получил стандарт OpenMP, долгое время позволяющий реализовывать только данным тип параллелизма. Однако, относительно недавно он был расширен набором специальных возможностей, дающих возможность эффективно работать с параллелизмом и по данным.

4. **Межузловой параллелизм**

   Ещё одним уровнем параллелизма можно рассматривать распределённые системы, представляющие собой набор независимых узлов (компьютеров), соединенных сетью, с программным обеспечением, обеспечивающим их совместное функционирование. Важно отметить, что все узлы автономны, а пользователи думают, что имеют дело с единой системой. Кроме того, для пользователей скрыты различия между компьютерами и способы связи между ними.

   С точки зрения разработчика, необходимо определенным образом обеспечить разбиение работы между доступными узлами, реализовав тем самым параллелизм. Для решения этой задачи используется MPI, а параллельность внутри каждого узла достигается через использование OpenMP и векторизации.

Необходимость параллельных вычислений

Количество задач, для решения которых необходимо использовать параллельные вычисления, стремительно растет и обусловлен рядом факторов:

1. Возможность изучать явления, которые являются либо слишком сложными для исследования аналитическими методами, либо слишком дорогостоящими или опасными для экспериментального изучения.
2. Быстрый рост сложности объектов моделирования (усложнение и увеличение систем).
3. Возникновение необходимости решения задач, для которых необходимо проведение анализа сложного поведения.
4. Необходимость управления сложными промышленными и технологическими процессами в режиме реального времени и в условиях неопределенности.
5. Рост числа задач, для решения которых необходимо обрабатывать гигантские объемы информации.

Использование численных моделей и кластерных систем позволяет значительно уменьшить стоимость процесса научного и технологического поиска. Кластерные системы широко используются во всём мире.

Создать программу, для выполнения которой будут задействованы все ресурсы суперкомпьютера, не всегда возможно. В самом деле, при разработке параллельной программы для распределенной системы мало разбить программу на параллельные потоки. Для эффективного использования ресурсов необходимо обеспечить равномерную загрузку каждого из узлов кластера, что в свою очередь означает, что все потоки программы должны выполнить примерно одинаковый объем вычислений.

Рассмотрим частный случай, когда при решении некоторой параметрической задачи для разных значений параметров время поиска решения может значительно различаться. Тогда мы получим значительный перекос загрузки узлов кластера. В действительности практически любая вычислительная задача выполняется в кластере неравномерно.

Несмотря на это, использование кластерных систем всегда более эффективно для обслуживания вычислительных потребностей большого количества пользователей, чем использование эквивалентного количества однопроцессорных рабочих станций, так как в этом случае с помощью системы управления

заданиями легче обеспечить равномерную и более эффективную загрузку вычислительных ресурсов.

Классификация параллельных архитектур

Под архитектурой вычислительной системы понимаются абстрактное представление ЭВМ с точки зрения программиста. Полное описание архитектуры системы включает в себя:

- основные форматы представления данных;
- способы адресации данных в программе;
- состав аппаратных средств вычислительной машины, характеристики этих средств, принципы организации вычислительного процесса.

Структуру вычислительной системы можно определить как совокупность аппаратных средств ЭВМ с указанием основных связей между ними.

Имеется много различных классификаций вычислительных систем. Рассмотрим наиболее часто используемые классификации.

Классификация по Флинну

Наибольшее распространение получила классификация вычислительных систем, предложенная в 1966 г. профессором Стэндфордского университета М.Д.Флином (M.J.Flynn) - классификация Флина. Эта классификация охватывает только два классификационных признака – тип потока команд и тип потока данных.

В одиночном потоке команд в один момент времени может выполняться только одна команда. В этом случае эта единственная команда определяет в данный момент времени работу всех или, по крайней мере, многих устройств вычислительной системы.

Во множественном потоке команд в один момент времени может выполняться много команд. В этом случае каждая из таких команд определяет в данный момент времени работу только одного или лишь нескольких (но не всех) устройств вычислительной системы.

В одиночном потоке последовательно выполняются отдельные команды, во множественном потоке – группы команд.

Одиночный поток данных обязательно предполагает наличие в вычислительной системе только одного устройства оперативной памяти и одного

процессора. Однако при этом процессор может быть как угодно сложным, так что процесс обработки каждой единицы информации в потоке может требовать выполнения многих команд.

Множественный поток данных состоит из многих зависимых или независимых одиночных потоков данных.

Все разнообразие архитектур ЭВМ в этой таксономии сводится к четырем классам:

- ОКОД – Вычислительная система с одиночным потоком команд и одиночным потоком данных (SISD, Single Instruction stream over a Single Data stream). Представляет собой классическую однопроцессорную ЭВМ фоннеймановской архитектуры.
- ОКМД – Вычислительная система с одиночным потоком команд и множественным потоком данных (SIMD, Single Instruction, Multiple Data). Типичными представителями SIMD являются векторные архитектуры.
- МКОД – Вычислительная система с множественным потоком команд и одиночным потоком данных (MISD, Multiple Instruction Single Data). Ряд исследователей относит к этому классу конвейерные ЭВМ, однако это не нашло окончательного признания, поэтому можно считать, что реальных систем – представителей данного класса не существует.
- МКМД – Вычислительная система с множественным потоком команд и множественным потоком данных (MIMD, Multiple Instruction Multiple Data). Данный класс содержит много процессоров, которые (как правило, асинхронно) выполняют разные команды над разными данными. Подавляющее большинство современных супер ЭВМ имеют архитектуру MIMD.

Рассмотренная классификация позволяет по принадлежности компьютера к классу SIMD или MIMD сделать понятным базовый принцип его работы. Часто этого бывает достаточно. Недостатком классификации является "переполненность" класс MIMD.

Классификация по типу строения оперативной памяти

По типу строения оперативной памяти системы разделяются на системы с общей (разделяемой) памятью, системы с распределенной памятью и системы с физически распределенной, а логически общедоступной памятью (гибридные системы).

В вычислительных системах с общей памятью (Common Memory Systems или Shared Memory Systems) значение, записанное в память одним из процессоров, напрямую доступно для другого процессора. Общая память обычно имеет высокую пропускную способность памяти и низкую латентность памяти при передаче информации между процессорами, но при условии, что не происходит одновременного обращения нескольких процессоров к одному и тому же элементу памяти. К общей памяти доступ разных процессоров системы осуществляется, как правило, за одинаковое время. Поэтому такая память называется еще UMA-памятью (Unified Memory Access) – памятью с одинаковым временем доступа. Система с такой памятью носит название вычислительной системы с одинаковым временем доступа к памяти. Системы с общей памятью называются также сильносвязанными вычислительными системами.

В вычислительных системах с распределенной памятью (Distributed Memory Systems) каждый процессор имеет свою локальную память с локальным адресным пространством. Для систем с распределенной памятью характерно наличие большого числа быстрых каналов, которые связывают отдельные части этой памяти с отдельными процессорами. Обмен информацией между частями распределенной памяти осуществляется обычно относительно медленно. Системы с распределенной памятью называются также слабосвязанными вычислительными системами.

Вычислительные системы с гибридной памятью - (Non-Uniform Memory Access Systems) имеют память, которая физически распределена по различным частям системы, но логически разделяема (образует единое адресное пространство). Такая память называется еще логически общей (разделяемой) памятью. В отличие от UMA-систем, в NUMA-системах время доступа к различным частям оперативной памяти различно.

Заметим, что память современных параллельных систем является многоуровневой, иерархической, что порождает проблему ее когерентности.

Классификация по степени однородности

По степени однородности различают однородные (гомогенные) и неоднородные (гетерогенные) вычислительные системы. Обычно при этом имеется в виду тип используемых процессоров.

В однородных вычислительных системах (гомогенных вычислительных системах) используются одинаковые процессоры, в неоднородных вычислительных системах (гетерогенных вычислительных системах) – процессоры различных типов. Вычислительная система, содержащая какой-либо специализированный вычислитель (например, Фурье-процессор), относится к классу неоднородных вычислительных систем.

В настоящее время большинство высокопроизводительных систем относятся к классу однородных систем с общей памятью или к классу однородных систем с распределенной памятью.

Рассмотренные классификационные признаки параллельных вычислительных систем не исчерпывают всех возможных их характеристик. Существует, например, еще разделение систем по степени согласованности режимов работы (синхронные и асинхронные вычислительные системы), по способу обработки (с пословной обработкой и ассоциативные вычислительные системы), по жесткости структуры (системы с фиксированной структурой и системы с перестраиваемой структурой), по управляющему потоку (системы потока команд и системы потока данных) и т.п.

Современные высокопроизводительные системы имеют, как правило, иерархическую структуру. Например, на верхнем уровне иерархии система относится к классу MIMD, каждый процессор которой представляет собой систему MIMD или систему SIMD.

Отметим также тенденцию к построению распределенных систем с программируемой структурой. В таких системах нет общего ресурса, отказ которого приводил бы к отказу системы в целом – средства управления, обработки и хранения информации распределены по составным частям системы. Такие системы обладают способностью автоматически реконфигурироваться в случае выхода из строя отдельных их частей. Средства реконфигурирования позволяют также программно перестроить систему с целью повышения эффективности решения на этой системе данной задачи или класса задач.

Основные понятия

Для того чтобы описывать параллельные вычисления, необходимо определить ряд понятий.

Масштабируемость – это такое свойство параллельной системы, при котором её производительность пропорциональна числу содержащихся в ней процессоров. Масштабируемость зависит от возможностей коммуникационных сетей, а так же от параллельного алгоритма: алгоритм, проверенный и хорошо работающий на вычислительной системе с малым числом процессоров может плохо работать (не давать ожидаемого ускорения) на системе с большим числом процессоров

Балансировка нагрузки применяется для оптимизации выполнения параллельных вычислений с помощью параллельной вычислительной системы, она предполагает равномерную нагрузку процессоров многопроцессорной ЭВМ. При появлении новых задач программное обеспечение, реализующее балансировку, должно принять решение о том, на каком процессоре (вычислительном узле) следует выполнять вычисления, связанные с этой новой задачей. Балансировка нагрузки также предполагает перенос части вычислений с наиболее загруженных процессоров на менее загруженные.

Гранулярность параллельного алгоритма – это отношение времени вычислений ко времени накладных расходов. Если уровень гранулярности слишком высок, то производительность страдает от издержек на коммуникацию. Если гранулярность слишком низкая, то в приложении появится дисбаланс нагрузки. Целью является определение правильного уровня гранулярности (обычно, чем выше, тем лучше) для параллельных задач, избегая дисбаланса нагрузки и издержек на коммуникацию, для достижения наилучшей производительности.

Ускорение реализации алгоритма на вычислительной системе, состоящей из одинаковых процессоров – отношение времени выполнения алгоритма на одном процессоре (на одном ядре процессора) ко времени параллельного выполнения. Ускорение зависит от выбора параллельного алгоритма и от того, насколько этот алгоритм адекватен архитектуре вычислительной системы.

Реальная производительность вычислительной системы – количество операций, реально выполняемых в среднем за единицу времени.

Пиковая производительность вычислительной системы – максимальное количество операций, которое может быть выполнено системой за единицу времени.

Из определений вытекает, что реальная и пиковая производительности системы есть суммы соответственно реальных и пиковых производительностей, составляющих систему процессоров. Пиковая производительность одного процессора вычисляется как произведение n×f×k, где n – количество операций с плавающей запятой, выполняемых за один такт, f – тактовая частота процессора, k – количество ядер в процессоре.

Пиковую производительность еще называют теоретической производительностью. Такое название подчеркивает, что на реальной программе производительность не только не превысит, но никогда и не достигнет этого порога.

Кластер – группа компьютеров, объединенных в локальную вычислительную сеть и способных работать в качестве единого вычислительного ресурса.

Кластер предполагает более высокую надежность и эффективность, нежели просто локальная вычислительную сеть. Кластер использует типовые аппаратные и программные решения и поэтому имеет существенно более низкую стоимость в сравнении с другими типами параллельных вычислительных систем.

Поток – создаваемый операционной системой объект внутри процесса (процесс – выполняемое приложение с собственным виртуальным адресным пространством), который выполняет инструкции программы. Процесс может иметь один или несколько потоков, выполняемых в контексте данного процесса. Потоки (нити) позволяют осуществлять параллельное выполнение процессов и одновременное выполнение одним процессом различных частей программы на различных процессорах.

Архитектура используемой высокопроизводительной системы

Аппаратное обеспечение

В НИИМ Нижегородского государственного университета им. Н.И.Лобачевского установлена система высокопроизводительных вычислений, состоящая из:

- подсистемы управления и мониторинга;
- инструментальной подсистемы;
- вычислительной подсистемы.

Подсистема управления и мониторинга представляет собой выделенный сервер, на котором установлено программное обеспечение управления аппаратными средствами и питанием вычислительной системы, а также программное обеспечение мониторинга различных компонент и параметров системы.

Инструментальная подсистема представляет выделенный сервер, предназначенные для подготовки задач, постановки их на расчёт на вычислительной системе и обработки результатов расчётов.

Вычислительная подсистема предоставляет свои ресурсы вычислительным задачам и состоит из 20 вычислительных узлов. Каждый вычислительный узел состоит из двух восьмиядерных процессоров с архитектурой x86-64 Intel(R) Xeon(R) CPU E5-2670 производительностью не менее 2,6 ГГц и оперативной памятью архитектуры DDR3 общим объёмом не менее 128 ГБ. Каждый вычислительный узел содержит интегрированный контроллер архитектуры QDR InfiniBand и раздельные интерфейсы для подключения к коммуникационной, управляющей и сервисной сети ВК.

Управляющая и сервисная сетевые среды Gigabit Ethernet создана на базе 48 портового коммутатора архитектуры GE, которая объединяет в локальную сеть все вычислительные узлы.

Программное обеспечение

Программное обеспечение системы высокопроизводительных вычислений реализовано на базе операционной системы (ОС) CentOS 6.5 (Final) с ядром 2.6.32-431. В состав ПО универсальных вычислительных модулей входят:

- бездисковый образ операционной системы;
- клиент запуска задач системы пакетной обработки заданий Slurm v2.6.4;
- клиент сетевой информационной службы NIS;
- клиент сетевой службы времени NTP;
- клиент сетевой файловой системы NFS;
- встроенное в ОС CentOS 6.5 ПО для коммуникационной среды InfiniBand (аналог OFED).

Один из вычислительных узлов реализуют функции административного сервера. В состав ПО административного сервера входит следующее специализированное системное ПО:

- программный пакет Opensm v.3.3.13 – сетевой менеджер для коммуникационной среды InfiniBand;
- ПО для коммуникационной среды InfiniBand (аналог OFED);
- сервер системы пакетной обработки заданий Slurm v2.6.4;
- клиент сетевой файловой системы NFS;
- сервер сетевой информационной службы NIS;
- сервер сетевой службы времени NTP;
- сервер системы динамической конфигурации DHCP;
- средства управления и мониторинга аппаратно-программных компонентов.

Так же на одном из вычислительных узлов реализованы функции инструментального и файлового сервера:

- программный пакет Opensm v.3.3.15 – сетевой менеджер для коммуникационной среды InfiniBand;
- ПО для коммуникационной среды InfiniBand (аналог OFED);
- клиент запуска задач системы пакетной обработки заданий Slurm v2.6.4;
- клиент сетевой информационной службы NIS;
- серверная часть сетевой файловой системы NFS;
- системы программирования для языков C, C++ и Фортран для компиляторов GNU - gcc, g++, gfortran v.4.4.7;

- системы программирования для языков C, C++ и Фортран для компиляторов intel - icc, ifort v.14.0;
- реализации библиотеки MPI MVAPICH2-1.9, OpenMPI 1.6.5, Intel MPI 4.1.3
- средства управления и администрирования.

Производительность системы

Производительность одного вычислительного модуля на тесте Linpack составила 300,5 Гфлопс. Его пиковая производительность составляет 2.6 GHz * 16 cores * 8 (ops/cycle) = 332.8 Гфлопс. Производительность всей системы на том же тесте 5833 Gflops, что составляет 87,6% относительно пиковой производительности в 6656 Gflops.

Модель программирования

Для реализации межузлового параллелизма на высокопроизводительной системе используется модель программирования, основанная на применении MPI (Message Passing Interface - интерфейс передачи данных) - наиболее распространенной на сегодняшний день модели при работе на системах с распределенной памятью. Она позволяет распределять вычислительную нагрузку и организовать информационное взаимодействие (передачу данных) между процессорами.

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов, при этом они могут выполняться как на разных процессорах, так и на одном. Все процессы порождаются один раз, при этом каждый процесс работает в своем адресном пространстве, что решает проблему с доступом к общим данным. Каждый процесс является копией одного и того же программного кода (модель SPMP - Single Programm Multiple Processes), который разрабатывается на алгоритмическом языке Fortran с использованием той или иной реализации библиотеки MPI.

Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ. Процессы программы последовательно пронумерованы от 0 до n-1, где n есть общее количество процессов. Номер процесса называется рангом процесса.

Для того чтобы выполнять различные вычисления на разных процессорах, нужно подставлять разные данные для программы, а так же с помощью средств идентификации процесса, организовать различия в вычислениях в зависимости от используемого программой процессора. В итоге становится возможным загружать ту или иную подзадачу в зависимости от “номера” процессора. При этом происходит декомпозиция исходной задачи. Обычная техника состоит в представлении каждой подзадачи в виде отдельной структурной единицы (функции, модуля), на всех процессорах запускается одна и та же программа “загрузчик”, которая, в зависимости от “номера” процессора загружает ту или иную подзадачу.

Информационное взаимодействие между процессами реализовано с использованием различных операций приема и передачи данных.

Для реализации параллелизма внутри каждого узла применяется стандарт OpenMP. Этот открытый стандарт, описывающий совокупность директив компилятора, библиотечных процедур и переменных окружения, предназначен для программирования многопоточных приложений на многопроцессорных системах с общей памятью. В случае использования установленной высокопроизводительной системы, таковой является каждый отдельный узел.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» поток создает набор подчиненных потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков).

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

Основы параллельного программирования на MPI

В вычислительных системах с распределенной памятью узлы работают независимо друг от друга. Для того чтобы реализовать параллельные вычисления, нужно иметь возможность распределять вычислительную нагрузку между узлами и эффективно организовать передачу данных между ними. Для решения всех перечисленных задач применяется интерфейс передачи данных MPI - Message Passing Interface.

В общем случае, для распределения вычислений между узлами необходимо:

- проанализировать алгоритм решения задачи
- выделить информационно независимые фрагменты вычислений
- выполнить программную реализацию
- разместить полученные части программы на разных узлах.

MPI использует более простой подход. Реализуется одна программа и эта единственная программа запускается одновременно на выполнение на всех имеющихся в системе узлах. Естественно, что нам не нужно выполнять одну и ту же программу много раз - целью является реализация парарллелизма. Для этого в приложение нужно подставлять разные данные на разных узлах, или, с помощью специальных средств MPI, идентифицировать узел, на котором выполняется программа, и производить различные вычисления в зависимости от используемого узла.

Для взаимодействия между узлами с точки зрения разработчика достаточно наличия операции приема и передачи данных. На аппаратном уровне, необходимо иметь возможность коммуникации между узлами по специальным каналам. MPI предоставляет огромное количество различных операций приема/передачи данных, что заложено уже в само название.

MPI - это стандарт, которому должны удовлетворять средства организации передачи сообщений. Кроме того, под MPI понимают программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта. Так, в соответствии со стандартом, эти программные средства должны быть организованы в виде библиотек и быть доступны для языков C и Фортран. Как правило, аббревиатура MPI используется для обозначения стандарта, а сочетание "библиотека MPI" указывает на ту

или иную программную реализацию стандарта. Часто для краткости обозначение MPI применяется и для библиотек MPI.

Основные преимущества MPI:

- позволяет в значительной степени снизить зависимость от аппаратной реализации, то есть решает проблему переносимости параллельных программ между разными компьютерными системами. Приложение, разработанное на языке C или Фортран, с использованием библиотеки MPI, будет работать на разных вычислительных платформах;
- повышает эффективность параллельных вычислений из-за наличия множества реализаций библиотек MPI, максимально оптимизированных для используемого оборудования;
- уменьшает сложность разработки параллельных приложений и позволяет использовать MPI совместно с другими технологиями параллельного программирования, реализующими параллелизм на других уровнях.

Основные понятия и определения

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы. Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (модель SPMP). Данный программный код, представленный в виде исполняемой программы, должен быть доступен в момент запуска параллельной программы на всех используемых процессорах. Исходный программный код для исполняемой программы разрабатывается на алгоритмических языках C или Фортран с использованием той или иной реализации библиотеки MPI. Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-

программ и в ходе вычислений меняться не может (в стандарте MPI-2 предусматривается возможность динамического изменения количества процессов). Все процессы программы последовательно перенумерованы от 0 до p-1, где p есть общее количество процессов. Номер процесса именуется рангом процесса.

Основу MPI составляют операции передачи сообщений. Среди предусмотренных в составе MPI функций различаются парные (point-to-point) операции между двумя процессами и коллективные (collective) коммуникационные действия для одновременного взаимодействия нескольких процессов. Для выполнения парных операций могут использоваться разные режимы передачи, среди которых синхронный, блокирующий и др. Как уже отмечалось ранее, стандарт MPI предусматривает необходимость реализации большинства основных коллективных операций передачи данных.

Процессы параллельной программы объединяются в группы. Под коммуникатором в MPI понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (контекст), используемых при выполнении операций передачи данных.

Как правило, парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов 3коммуникатора. Как результат, указание используемого коммуникатора является обязательным для операций передачи данных в MPI. В ходе вычислений могут создаваться новые и удаляться существующие группы процессов и коммуникаторы. Один и тот же процесс может принадлежать разным группам и коммуникаторам. Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD. При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор (intercommunicator).

При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях MPI необходимо указывать тип пересылаемых данных. MPI содержит большой набор базовых типов данных, во многом совпадающих с типами данных в языках C и Фортран. Кроме того, в MPI имеются возможности для создания новых производных типов данных для более точного и краткого описания содержимого пересылаемых сообщений.

Базовые функции

В начале любого приложения, использующего MPI, должна быть вызвана функция инициализации среды выполнения.

C:

*int MPI_Init (int *agrc, char ***argv);*

Фортран:

subroutine MPI_INIT(ierr)

integer ierr

Параметрами функции на языке C являются аргументы функции main, полученные из командной строки, на Фортране – код ошибки. Функция, как и все другие функции MPI, возвращает код ошибки в случае использования в языке C. При успешном выполнении функции возвращаемый код равен MPI_SUCCESS. Другие значения кода завершения свидетельствуют об обнаружении тех или иных ошибочных ситуаций в ходе выполнения функций. Для выяснения типа обнаруженной ошибки используются предопределенные именованные константы, среди которых:

- MPI_ERR_BUFFER – неправильный указатель на буфер,
- MPI_ERR_COMM – неправильный коммуникатор,
- MPI_ERR_RANK – неправильный ранг процесса,

Полный список констант для проверки кода завершения содержится в файле mpi.h (C) и mpif.h (Фортран).

В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы-приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе. Кроме этого, создается предопределенный коммуникатор MPI_COMM_SELF, описывающий свою область связи для каждого отдельного процесса.

Последней вызываемой функцией обязательно должна являться функция финализации выполнения работы окружением:

C:

int MPI_Finalize (void);

Фортран:
MPI_FINALIZE(IERROR)
INTEGER IERROR

Функция закрывает все MPI-процессы и ликвидирует все области связи.

Таким образом, структура параллельной программы, разработанная с использованием MPI, обычно имеет следующий вид:

C:
```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
 <код без MPI функций>
 MPI_Init ( &agrc, &argv );
 < код с использованием MPI функций>
 MPI_Finalize();
 < код без MPI функций>
 return 0;
}
```

Фортран:
```
program test
include 'mpif.h'
integer ierr
<код без MPI функций>
call MPI_INIT ( ierr )
< код с MPI функциями>
call MPI_FINALIZE ( ierr )
< код без MPI функций>
end
```

В файлах mpi.h и mpif.h содержатся определения именованных констант, прототипов функций и типов данных библиотеки MPI. Функции MPI_Init и MPI_Finalize являются обязательными и должны быть выполнены только один

раз каждым процессом параллельной программы. Кроме того, перед вызовом MPI_Init может быть использована функция MPI_Initialized для определения того, был ли ранее выполнен вызов MPI_Init.

Рассмотренные примеры дают представление о том, как формируются имена функций в MPI. Они начинаются с префикса MPI, далее следует одно или несколько слов названия, первое слово в имени функции начинается с заглавного символа (синтаксис языка C), слова разделяются знаком нижнего подчеркивания. Сами названия функций MPI поясняют назначение выполняемых действий. Схематично формат вызова можно показать так:

C:

rc = MPI_Xxxxx(parameter, ...)

Фортран:

CALL MPI_XXXXX(parameter,..., ierr)

call mpi_xxxxx(parameter,..., ierr)

Определение количества процессов в области связи коммуникатора comm осуществляется при помощи функции:

C:

*int MPI_Comm_size (MPI_Comm comm, int *size).*

Фортран:

MPI_COMM_SIZE(COMM, SIZE, IERROR)

INTEGER COMM, SIZE, IERROR

Аргумент size возвращает число процессов в области связи коммуникатора comm. До момента создания явным образом групп и связанных с ними коммуникаторов возможными значениями параметра COMM могут быть MPI_COMM_WORLD и MPI_COMM_SELF, которые создаются автоматически при инициализации MPI.

Для определения номера (ранга) процесса используется функция:

C:

*int MPI_Comm_rank (MPI_Comm comm, int *rank).*

Фортран:

MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER COMM, RANK, IERROR

Номера процессов лежат в диапазоне от 0 до size-1, а значение size может быть определено с помощью функции MPI_Comm_size. Обычно вызов функций MPI_Comm_size и MPI_Comm_rank выполняется сразу после инициализации:

C:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
 int ntasks, rank;
 < код без MPI функций>
 MPI_Init ( &agrc, &argv );
 MPI_Comm_size ( MPI_COMM_WORLD, & ntasks);
 MPI_Comm_rank ( MPI_COMM_WORLD, &rank);
 < код с MPI функциями>
 MPI_Finalize();
 < код без MPI функций>
 return 0;
}
```

Фортран:

```
program test
include 'mpif.h'
integer ierr, ntasks, rank,rc
<код без MPI функций>
call MPI_INIT ( ierr )
if (ierr .ne. MPI_SUCCESS) then
    print *,'Ошибка инициализации MPI'
    call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
  end if
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
< код с MPI функциями>
call MPI_FINALIZE ( ierr )
```

< код без MPI функций>

end

Коммуникатор MPI_COMM_WORLD создается по умолчанию и представляет все процессы выполняемой параллельной программы, а ранг, получаемый при помощи функции MPI_Comm_rank, является рангом процесса, выполнившего вызов этой функции. Таким образом, переменная *rank* будет принимать различные значения в разных процессах. Для примера на языке Фортран показано, как можно обработать ошибку при инициализации MPI.

В минимальный набор базовых функций MPI, необходимо включить также две функции передачи и приема сообщений.

Для передачи сообщения процесс-отправитель должен выполнить функцию:

C:

*int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm),*

Фортран:

MPI_SEND(BUF, COUNT, TYPE, DEST, TAG, COMM, IERROR)

<type> BUF(*)

INTEGER COUNT, TYPE, DEST, TAG, COMM, IERROR

Аргументами функции являются:

buf – адрес начала буфера памяти, в котором располагаются пересылаемые данные,

count – количество пересылаемых элементов данных в сообщении,

type – тип данных пересылаемых элементов,

dest – ранг процесса-получателя, которому отправляется сообщение,

tag – идентификатор сообщения,

comm – коммуникатор области связи.

Функция MPI_Send выполняет посылку count элементов типа type сообщения с идентификатором tag процессу dest в области связи коммуникатора comm. Переменная buf - это, как правило, массив или скалярная переменная. В последнем случае значение count = 1.

Для указания типа пересылаемых данных в MPI имеется ряд базовых типов, список которых приведен в таблице ниже.

C		Фортран	
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_FLOAT	float	MPI_REAL	real
MPI_INT	int	MPI_INTEGER	integer
MPI_C_COMPLEX	float _Complex	MPI_COMPLEX	complex
MPI_C_BOOL	_Bool	MPI_LOGICAL	logical
MPI_LONG	long int	MPI_DOUBLE_COMPLEX	double complex
MPI_LONG_DOUBLE	long double		
MPI_SHORT	short		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long		
MPI_UNSIGNED_SHORT	unsigned short		
MPI_PACKED			

Отправляемое сообщение определяется через указание буфера памяти, в котором это сообщение располагается. Аргументы buf, count, type передаются почти во всех функциях передачи данных. Процессы, между которыми выполняется передача данных, в обязательном порядке должны принадлежать коммуникатору, указываемому в функции MPI_Send. Параметр tag используется только при необходимости идентификации передаваемых сообщений. Если этого не требуется, можно использовать произвольное целое число в качестве входного параметра функции.

Сразу после выполнения функции MPI_Send процесс-отправитель может повторно использовать буфер памяти, в котором располагалось отправляемое сообщение. Тем не менее, в момент завершения функции MPI_Send состояние пересылаемого сообщения может быть различным. Сообщение может располагаться в процессе-отправителе, может находиться в процессе передачи, может храниться в процессе-получателе или может быть принято процессом-получателем при помощи функции MPI_Recv. Завершение функции MPI_Send означает лишь то, что операция передачи начала выполняться и пересылка сообщения будет в какой-то момент времени выполнена.

Для приема сообщения процесс-получатель должен выполнить функцию:

C:

*int MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status),*

Фортран:

MPI_RECV(BUF, COUNT, TYPE, SOURCE, TAG, COMM, STATUS, IERROR)

<type> BUF(*)

INTEGER COUNT, TYPE, SOURCE, TAG, COMM,

STATUS(MPI_STATUS_SIZE), IERROR

Разница в аргументах по сравнению с функцией MPI_Send очевидна – большинство параметров теперь используется для приема сообщения:

buf, count, type – буфер памяти для приема сообщения

source - ранг процесса, от которого должен быть выполнен прием сообщения

tag - тег сообщения, которое должно быть принято для процесса

comm - коммуникатор, в рамках которого выполняется передача данных

- status – указатель на структуру данных с информацией о результате выполнения операции приема данных.

Функция выполняет прием count элементов типа type сообщения с идентификатором tag от процесса source в области связи коммуникатора comm. При этом буфер памяти должен быть достаточным для приема сообщения, а тип элементов передаваемого и принимаемого сообщения должны совпадать. В случае если памяти недостаточно, часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения. Если необходимо принять сообщения от любого процесса-отправителя, для параметра source можно указать значение MPI_ANY_SOURCE. Аналогично, при необходимости приема сообщения с любым тегом для соответствующего параметра tag может быть указано значение MPI_ANY_TAG.

Новый по сравнению с функцией отправки параметр status позволяет определить ряд характеристик принятого сообщения:

status.MPI_SOURCE – ранг процесса-отправителя принятого сообщения,

status.MPI_TAG - тег принятого сообщения.

Описанные функции приема/передачи сообщений реализуют стандартный режим работы с блокировкой, то есть подразумевают выход из них только

после полного окончания операции. Таким образом, вызывающий процесс блокируется, пока операция не будет завершена. Для функции посылки сообщения это означает, что все пересылаемые данные помещены в буфер. Для функции приема сообщения блокируется выполнение других операций, пока все данные из буфера не будут помещены в адресное пространство принимающего процесса.

Рассмотренный набор функций оказывается достаточным для разработки параллельных программ. Приводимая ниже программа является стандартным начальным примером для языка С. Аналогичный код может быть написан на языке Фортран.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
    int ntasks, prank, rrank;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &prank);
    if (prank == 0 )
    {
        // выполняется только процессом с рангом 0
        printf ("\n Hello from process %3d", prank);
        for ( int i=1; i< ntasks; i++ )
        {
            MPI_Recv(&rrank, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("\n Hello from process %3d", rrank);
        }
    }
    else    // сообщение отправляется всеми процессами,
        // кроме процесса с рангом 0
        MPI_Send(&prank,1,MPI_INT,0,0,MPI_COMM_WORLD);
MPI_Finalize();
 return 0;
```

}

Каждый процесс определяет свой ранг, после чего действия в программе разделяются. Все процессы, кроме процесса с рангом 0, передают значение своего ранга нулевому процессу. Процесс с рангом 0 сначала печатает значение своего ранга, а далее последовательно принимает сообщения с рангами процессов и также печатает их значения. Порядок приема сообщений заранее не определен и зависит от условий выполнения параллельной программы. Он так же может меняться от запуска к запуску. Один из возможных вариантов печати процесса 0 для параллельной программы из четырех процессов может выглядеть так:

Hello from process 0
Hello from process 3
Hello from process 2
Hello from process 1

Нужно иметь ввиду, что такая невоспроизводимость результатов от запуска к запуску усложняет разработку, тестирование и отладку приложения. Кроме того, нужно заботиться о сохранении численных результатов и особенно внимательно относиться к операциям суммирования, которые могут быть выполнены в произвольном порядке, что приводит к различным численным результатам. Для рассматриваемого простого примера можно восстановить постоянство получаемых результатов при помощи задания ранга процесса-отправителя в операции приема сообщения:

MPI_Recv(&rrank, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status).

Таким образом, явно указывается порядок приема сообщений, и печать происходит в порядке возрастания рангов процессов, что, в определенных случаях, может замедлять выполнение параллельного приложения.

Разрабатываемая с использованием MPI программа порождает все процессы и должна определять вычисления, выполняемые в них. Для разделения выполняемого кода между процессами обычно используется подход, рассмотренный в примере. Сначала, с помощью функции MPI_Comm_rank определяется ранг процесса, а затем, в зависимости от ранга, выполняются различные участки кода. Подобный поход усложняет понимание и разработку MPI-

программ, поэтому необходимо выносить код для разных процессов в отдельные функции. Схема MPI программы в этом случае будет выглядеть так:

```
MPI_Comm_rank(MPI_COMM_WORLD, &prank);
if (prank == 0 ) process0();
else if (prank == 1 ) process1();
else if (prank == 2 ) process2();
```

Часто необходимо выделить отдельный код только для процесса с рангом 0. В этом случае общая структура принимает более простой вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, & prank);
if (prank == 0 )
    MasterProcess();
else
    WorkerProcesses();
```

При разработке параллельных приложений возникает необходимость определять время их выполнения для оценки получаемого ускорения. Стандартные средства для измерения времени обычно зависят от аппаратной платформы, операционной системы, языка программирования и других факторов. В MPI включены специальные функции для измерения времени, не имеющие зависимостей от среды выполнения.

Функция отсчета времени имеет следующий вид:

C:

```
double MPI_Wtime(void),
```

Фортран:

```
DOUBLE PRECISION MPI_WTIME()
```

Функция возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом (точки отсчета). Гарантируется, что эта точка отсчета не будет изменена в течение жизни процесса. Для хронометража участка программы вызов функции делается в начале и конце участка и определяется разница между показаниями таймера.

Возможный пример применения функции MPI_Wtime:

```
double t1, t2, t;
t1 = MPI_Wtime();
...
```

t2 = MPI_Wtime();

t = t2 – t1;

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция MPI_Wtick, возвращающая разрешение таймера (минимальное значение кванта времени):

C:

double MPI_Wtick(void)

Фортран:

DOUBLE PRECISION MPI_WTICK()

Приведенных материалов достаточно для начала разработки приложений на системах с распределенной памятью и понимания их функционирования. Далее более детально будут рассмотрены более мощные возможности MPI.

Функция MPI_Sendrecv объединяет функционал приема и отправки сообщений.

C:

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

Фортран:

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST,
             SENDTAG,  RECVBUF,  RECVCOUNT,  RECVTYPE,
             SOURCE, RECVTAG, COMM, STATUS, IERROR)
<TYPE> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT,
RECVTYPE,SOURCE, RECV TAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

Аргументами функции являются:

sbuf – отправляемое сообщение;

scount – число элементов в отправляемом сообщении;

stype – тип элементов в отправляемом сообщении;

dest – номер процесса-получателя;

stag – идентификатор отправляемого сообщения;

rbuf – принимаемое сообщение;

rcount – максимальное число элементов в принимаемом сообщении;

rtype – тип элементов в принимаемом сообщении;

source – номер процесса-отправителя;

rtag – идентификатор принимаемого сообщения;

comm – коммуникатор группы;

status – параметры принятого сообщения.

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

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

C:

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

Фортран:

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

Аргументами функции являются:

source – номер процесса-отправителя или MPI_ANY_SOURCE;

tag – идентификатор принимаемого сообщения или MPI_ANY_TAG;

comm – коммуникатор группы;

status – параметры принимаемого сообщения.

Функция MPI_Probe получает параметры принимаемого сообщения с блокировкой процесса. Возврата из функции не происходит до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для приема. Параметры принимаемого сообщения опреде-

ляются обычным образом с помощью параметра status. Функция MPI_Probe отмечает только факт получения сообщения, но реально его не принимает.

Для определения количества уже полученных сообщений используется функция MPI_Get_Count.

C:

```
int MPI_Get_Count (  MPI_Status *status,  MPI_Datatype datatype,
                     int *count );
```

Фортран:

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

Аргументами функции являются:

status – параметры сообщения;

datatype – тип элементов;

count – число элементов в сообщении.

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

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

```
#include <stdio.h>
#include "mpi.h"
int main (int argc, char *argv[])
{
    int i;
    int size, me;
    int sum;
    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
     MPI_Comm_rank (MPI_COMM_WORLD, &me);
     sum = 0;
     for (i=me*10+1; i<=(me+1)*10; i++)
```

```
            sum = sum + i;
        if (me == 0)
        {
            if (size > 1)
            {
                printf ("Процесс 0, сумма = %d\n", sum);
                for (i=1; i<size; i++)
                {
                    MPI_Recv (&sum, 1, MPI_INT, i,
                                1, MPI_COMM_WORLD, &status);
                    printf ("Процесс %d, сумма = %d\n", i, sum);
                }
            }
         }
        else
            MPI_Send (&sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        MPI_Finalize ();
        return (0);
}
```

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

Асинхронные сообщения

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

могательных функций программа может контролировать процесс приема/отправки и таким образом координировать свои дальнейшие действия.

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

Для инициирования процесса асинхронной отправки сообщения используется функция MPI_Isend.

C:

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

Фортран:

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG,
          COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Аргументами функции являются:

buf – отправляемое сообщение;

count – число элементов в отправляемом сообщении;

datatype – тип элементов;

dest – номер процесса-получателя;

msgtag – идентификатор сообщения;

comm – коммуникатор группы;

request – идентификатор асинхронной передачи.

Функция MPI_Isend осуществляет отправку сообщения аналогично функции MPI_Send, однако возврат из функции происходит сразу после инициирования процесса отправки без ожидания обработки всего сообщения, находящегося в буфере buf. Это означает, что нельзя повторно использовать данный

буфер для других целей без получения дополнительной информации о завершении инициированной ранее отправки. Завершение процесса отправки сообщения можно определить с помощью параметра request в функциях MPI_Wait и MPI_Test. Только после успешной отправки сообщения буфер buf может быть использован повторно без опасения испортить передаваемые сообщения.

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

Для инициирования процесса асинхронного приема сообщения используется функция MPI_Irecv.

C:

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

Фортран:

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE,
                TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM
INTEGER REQUEST, IERROR
```

Аргументами функции являются:

buf – принимаемое сообщение;

count – максимальное число элементов в принимаемом сообщении;

datatype – тип элементов;

source – номер процесса-отправителя;

msgtag – идентификатор сообщения;

comm – коммуникатор группы;

request – идентификатор асинхронной передачи.

Функция MPI_Irecv осуществляет прием сообщения аналогично функции MPI_Recv, однако возврат из функции происходит сразу после инициирования процесса приема без ожидания получения сообщения в буфере buf. За-

вершение процесса приема сообщения можно определить с помощью параметра request в функциях MPI_Wait и MPI_Test.

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

C:

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

Фортран:

MPI_WAIT(REQUEST, STATUS, IERROR)

INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Аргументами функции являются:

request – идентификатор асинхронной передачи;

status – параметры переданного сообщения.

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

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

```
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"
int main (int argc, char *argv[])
{
     int size, me, left, right;
    int rbuffer[10], sbuffer[10];
    double time;
    MPI_Request rrequest, srequest;
```

```
MPI_Status status;
 MPI_Init (&argc, &argv);
 MPI_Comm_size (MPI_COMM_WORLD, &size);
 MPI_Comm_rank (MPI_COMM_WORLD, &me);
 right = (me + 1) % size;
 left = me – 1;
 if (left < 0)
        left = size – 1;
 time = MPI_Wtime();
 printf ("Мой номер %d, время до паузы %.0f\n",  me, time);
 sleep (5);
if (me == 0)
        sleep(5);
MPI_Irecv (rbuffer, 10, MPI_INT, left, 123,
                MPI_COMM_WORLD, &rrequest);
MPI_Isend (sbuffer, 10, MPI_INT, right, 123,
                MPI_COMM_WORLD, &srequest);
 time = MPI_Wtime();
printf ("Мой номер %d, время после паузы %.0f\n", me, time);
MPI_Wait (&rrequest, &status);
time = MPI_Wtime ();
 printf ("Мой номер %d, время запуска %.0f\n", me, time);
MPI_Finalize ();
return (0);
}
```

В этом примере используется рассмотренная ранее функция MPI_Wtime, которая возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. В данной программе функция MPI_Wtime используется для определения моментов времени, когда программа проходит различные этапы.

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

- До первого вывода сообщения на экран все процессы доходят в одно и то же время.

- До второго вывода сообщения на экран одновременно доходят все процессы кроме нулевого. Нулевой процесс задерживается на 5 секунд.
- До третьего вывода сообщения на экран одновременно доходят все процессы кроме первого и нулевого. Первый и нулевой процессы задерживаются на 5 секунд.

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

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

C:

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

Фортран:

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS,
ARRAY_OF_STATUSES, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,), IERROR*

Аргументами функции являются:

count – число идентификаторов асинхронной передачи;

requests – идентификаторы асинхронной передачи;

statuses – параметры переданных сообщений.

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

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

C:

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

Фортран:

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS,
INDEX, STATUS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*)
INTEGER INDEX, STATUS(MPI_STATUS_SIZE), IERROR

Аргументами функции являются:

count – число идентификаторов асинхронной передачи;

requests – идентификаторы асинхронной передачи;

index – номер завершенной передачи;

status – параметры переданного сообщения.

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

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

C:

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

Фортран:

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
ARRAY_OF_INDICES, ARRAY_OF_STATUSES,
IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT
INTEGER ARRAY_OF_INDICES(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,), IERROR*

Аргументами функции являются:

incount – число идентификаторов асинхронной передачи;

requests – идентификаторы асинхронной передачи;

outcount – число завершенных асинхронных передач;

indexes – массив номеров завершенных передач;

statuses – параметры переданных сообщений в завершенных передачах.

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

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

C:

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

Фортран:

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)

LOGICAL FLAG

INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Аргументами функции являются:

request – идентификатор асинхронной передачи;

flag – признак завершения передачи;

status – параметры переданного сообщения.

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

C:

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

Фортран:

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG,
ARRAY_OF_STATUSES, IERROR)
LOGICAL FLAG
INTEGER COUNT, ARRAY_OF_REQUESTS(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,), IERROR*

Аргументами функции являются:
count – число идентификаторов асинхронной передачи;
requests – идентификаторы асинхронной передачи;
flag – признак завершения передачи;
statuses – параметры переданных сообщений.

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

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

Фортран:
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG,
STATUS, IERROR)
LOGICAL FLAG
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX
INTEGER STATUS(MPI_STATUS_SIZE), IERROR

Аргументами функции являются:
count – число идентификаторов асинхронной передачи;
requests – идентификаторы асинхронной передачи;
index – номер завершенной передачи;
flag – признак завершения передачи;
status – параметры переданного сообщения.

Функция MPI_Testany проверяет завершение передач из массива requests. Параметр flag по умолчанию равен нулю и принимает значение 1, если

хотя бы одна из указанных передач завершена. Параметр index возвращает номер элемента в массиве requests, содержащего идентификатор завершенной передачи.

Если могут быть завершены сразу несколько передач, то случайным образом выбирается одна из них.

C:

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

Фортран:

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
ARRAY_OF_INDICES, ARRAY_OF_STATUSES,
IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*)
INTEGER OUTCOUNT, ARRAY_OF_INDICES(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,), IERROR*

Аргументами функции являются:

incount – число идентификаторов асинхронной передачи;

requests – идентификаторы асинхронной передачи;

outcount – число завершенных асинхронных передач;

indexes – массив номеров завершенных передач;

statuses – параметры переданных сообщений в завершенных передачах.

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

Использование функции MPI_Test и ее производных позволяет разработчику организовывать дополнительную активность процессов без создания дополнительных потоков. Реагирующий на события потоковый планировщик

легко реализуется с помощью циклического вызова MPI_Test. Код такого планировщика упрощенно выглядит следующим образом:

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

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

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

C:

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

Фортран:

```
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

Аргументами функции являются:

source – номер процесса-отправителя или MPI_ANY_SOURCE;

msgtag – идентификатор принимаемого сообщения или MPI_ANY_TAG;

comm – коммуникатор группы;

flag – признак завершения передачи;

status – параметры принимаемого сообщения.

Функция MPI_Iprobe получает информацию о структуре принимаемого сообщения без блокировки процесса. Параметр flag по умолчанию равен нулю и принимает значение 1, если сообщение с подходящим идентификатором и номером процесса-отправителя доступно для приема. Параметры принимаемого сообщения определяются обычным образом с помощью параметра status.

Функция MPI_Iprobe отмечает только факт получения сообщения, но реально его не принимает.

Код планировщика с применением функции MPI_Iprobe может упрощенно выглядеть следующим образом

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

Объединение запросов

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

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

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

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

Для снижения накладных расходов при передаче сообщений между процессами несколько запросов на прием и отправку сообщений предварительно

объединяются вместе и затем одновременно выполняются. Для этого используются функции MPI_Send_init, MPI_Recv_init, MPI_Start и их производные.

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

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

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

Рассмотрим подробнее функции для объединения запросов на передачу сообщений.

C:

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

Фортран:

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG,
COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG
INTEGER COMM, REQUEST, IERROR

Аргументами функции являются:

buf – отправляемое сообщение;

count – число элементов в отправляемом сообщении;

datatype – тип элементов;

dest – номер процесса-получателя;

msgtag – идентификатор сообщения;

comm – коммуникатор группы;

request – идентификатор асинхронной передачи.

Фортран:

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

C:

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

Фортран:

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG,
COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG
INTEGER COMM, REQUEST, IERROR

Аргументами функции являются:

buf – принимаемое сообщение;

count – максимальное число элементов в принимаемом сообщении;

datatype – тип элементов;

source – номер процесса-отправителя;

msgtag – идентификатор сообщения;

comm – коммуникатор группы;

request – идентификатор асинхронной передачи.

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

C:

*int MPI_Start (MPI_Request *request);*

Фортран:

Аргументом функции является идентификатор асинхронной передачи request.

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

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

C:

*int MPI_Start_all (MPI_Request *requests);*

Фортран:

MPI_START(REQUEST, IERROR)

INTEGER REQUEST, IERROR

Аргументом функции являются идентификаторы асинхронной передачи requests.

Функция MPI_Start_all запускает все отложенные передачи с идентификаторами из массива requests, сформированные вызовами функций MPI_Send_init и MPI_Recv_init. В момент обращения к функции MPI_Start_all необходимо, чтобы указатели на идентификаторы асинхронной передачи указывали на полностью сформированные запросы.

Для отмены отложенной передачи данных используется функция MPI_Request_free.

C:

*int MPI_Request_free (MPI_Request *request);*

Фортран:

MPI_REQUEST_FREE(REQUEST, IERROR)

INTEGER REQUEST, IERROR

Аргументом функции является идентификатор асинхронной передачи request.

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

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

Приведенная ниже программа демонстрирует работу функций отложенного взаимодействия:

```
#include <stdio.h>
#include "mpi.h"
int main (int argc, char *argv[])
{
    int i, j;
    int me, size;
    int buffer[3];
    MPI_Request request;
    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &me);
    if (size > 1)
    {
        if (me == 0)
        {
            MPI_Send_init (buffer, 3, MPI_INT, 1,
                           1, MPI_COMM_WORLD, &request);
            for (i=0; i<4; i++)
            {
                for (j=0; j<=2; j++)
                    buffer[j] = j * (i + 1);
                MPI_Start (&request);
                MPI_Wait (&request, &status);
            }
        }
    if (me == 1)
     {
        for (i=0; i<4; i++)
```

```
            {
                MPI_Recv (buffer, 3, MPI_INT, 0,
                        1, MPI_COMM_WORLD, &status);
                printf ("Сообщение %d – %d, %d, %d\n",
                    i + 1, buffer[0], buffer[1], buffer[2]);
            }
        }
        }
    MPI_Finalize ();
    return (0);
    }
```

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

Данная программа демонстрирует алгоритм отказа от формирования запросов на отправку каждого типового сообщения.

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

Барьерная синхронизация

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

Функция MPI_Barrier предоставляет возможность синхронизации процессов без передачи сообщений.

C:

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

Фортран:

```
MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR
```

Аргументом функции является коммуникатор группы comm.

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

Приведенная ниже программа демонстрирует работу функции MPI_Barrier:

```
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"
int main (int argc, char *argv[])
{
    int i;
    int me, size;
    double buffer[3];
    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &me);
    buffer[0] = MPI_Wtime();
    sleep (me);
    buffer[1] = MPI_Wtime();
    MPI_Barrier (MPI_COMM_WORLD);
    buffer[2] = MPI_Wtime();
    if (me > 0)
        MPI_Send (buffer, 3, MPI_DOUBLE, 0,
                  1, MPI_COMM_WORLD);
    else
    {
        for (i=0; i<size; i++)
        {
            if (i > 0)
```

```
                MPI_Recv (buffer, 3, MPI_DOUBLE, i,
                        1, MPI_COMM_WORLD, &status);
            printf ("Процесс %d, время в точке 1 - %.0f, 2 – %.0f,
                3 – %.0f\n", i, buffer[0], buffer[1], buffer[2]);
        }
    }
    MPI_Finalize ();
    return (0);
}
```

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

- До первого вывода сообщения на экран.
- Перед вызовом функции MPI_Barrier.
- После вызова функции MPI_Barrier.

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

Группы процессов

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

Группа с коммуникатором MPI_GROUP_EMPTY – пустая группа, не содержащая ни одного процесса. Группа с коммуникатором MPI_GROUP_NULL – значение, используемое для ошибочной группы.

Новые группы можно создавать как на основе уже существующих групп, так и на основе коммуникаторов, но в операциях обмена могут использоваться только коммуникаторы. Базовая группа, из которой создаются все остальные группы процессов, связана с коммуникатором MPI_COMM_WORLD, в нее входят все процессы приложения. Операции над группами процессов являются локальными, в них вовлекается только вызвав-

ший процедуру процесс, а выполнение не требует межпроцессорного обмена данными. Любой процесс может производить операции над любыми группами, в том числе над такими, которые не содержат данный процесс. При операциях над группами может получиться пустая группа MPI_GROUP_EMPTY.

Если при инициализации параллельной программы можно эффективно работать только с группой MPI_COMM_WORLD, то в дальнейшем при помощи функций-конструкторов групп можно создавать новые группы и коммуникаторы. Конструкторы групп применяются к подмножеству и расширенному множеству существующих групп. Эти конструкторы создают новые группы на основе существующих групп. Данные операции являются локальными и различные группы могут быть определены на различных процессах; процесс может также определять группу, которая не включает саму себя.

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

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

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

C:

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

Фортран:

MPI_COMM_GROUP(COMM, GROUP, IERROR)

INTEGER COMM, GROUP, IERROR

Аргументами функции являются:

− comm – коммуникатор группы;

− group – группа в коммуникаторе.

Функция MPI_Comm_group возвращает в group дескриптор группы из comm. Данная функция используется для того, чтобы получить доступ к груп-

пе, соответствующей определенному коммуникатору. После этого над группами можно проводить операции.

Возможные операции над множествами определяются следующим образом:

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

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

C:

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

Фортран:

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

Аргументами функции являются:

group1 – указатель на первую группу;

group2 – указатель на вторую группу;

newgroup – указатель на создаваемую группу.

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

C:

int MPI_Group_intersection (MPI_Group group1, MPI_Group group2,

*MPI_Group *newgroup);*

Фортран:

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP,
IERROR)

INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

Аргументами функции являются:

group1 – указатель на первую группу;

group2 – указатель на вторую группу;

newgroup – указатель на создаваемую группу.

Функция MPI_Group_intersection образует группу, которая является пересечением group1и group2, и записывает результат в newgroup.

C:

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

Фортран:

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)

INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

Аргументами функции являются:

group1 – указатель на первую группу;

group2 – указатель на вторую группу;

newgroup – указатель на создаваемую группу.

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

C:

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

Фортран:

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)

INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

Аргументами функции являются:

group – указатель на группу;

n – количество процессов, входящих в новую группу;

ranks – номера процессов из group, которые должны войти в создаваемую группу;

newgroup – указатель на создаваемую группу.

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

C:

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

Фортран:

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

Аргументами функции являются:

group – указатель на группу;

n – количество процессов, не входящих в новую группу;

ranks – номера процессов из group, которые не должны войти в создаваемую группу;

newgroup – указатель на создаваемую группу.

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

C:

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

Фортран:

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES,
NEWGROUP, IERROR)
INTEGER GROUP, N, RANGES(3,), NEWGROUP, IERROR*

Аргументами функции являются:

group – указатель на группу;

n – число триплетов в массиве ranges;

ranges – массив триплетов, указывающий номера процессов в group, которые включены в newgroup;

newgroup – указатель на создаваемую группу.

Функция MPI_Group_range_incl создает группу на основе последовательностей процессов из существующей группы. Результат записывается в newgroup. Последовательности процессов вычисляются при помощи триплетов (первый номер, последний номер, шаг).

Если аргументы ranges состоят из триплетов вида (first1, first2 + stride1), …, (firstn, firstn + striden), то newgroup состоит из последовательности процессов с номерами first1, first1 + stride1, …, first1 + stride1 × (last1 – first1) / stride1, … firstn, firstn + striden, …, firstn + striden × (lastn – firstn) / striden.

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

Возможности этой функции позволяют расширить массив номеров до массива включенных номеров и передать результирующий массив номеров и другие аргументы в MPI_Group_incl.

Вызов функции MPI_Group_incl эквивалентен вызову функции MPI_Group_range_incl с каждым номером i в ranks, замененным триплетом (i, i, 1) в аргументе ranges.

C:

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

Фортран:

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES,
NEWGROUP, IERROR)

INTEGER GROUP, N, RANGES(3,), NEWGROUP, IERROR*

Аргументами функции являются:

group – указатель на группу;

n – число триплетов в массиве ranges;

ranges – массив триплетов, указывающий номера процессов в group, которые исключены из newgroup;

newgroup – указатель на создаваемую группу.

Функция MPI_Group_range_excl создает группу процессов newgroup, которая получена путем удаления из group определенных последовательностей процессов. Эти последовательности задаются триплетами аналогично MPI_Group_range_incl.

Каждый вычисленный номер должен быть правильным номером в group, и все вычисленные номера должны быть различными, иначе программа будет неверной.

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

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

C:

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

Фортран:

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2,
RANKS2, IERROR)
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

Аргументами функции являются:

group1 – указатель на первую группу;

n – число элементов в массивах ranks1 и ranks2;

ranks1 – массив из нуля или более правильных номеров в первой группе;

group2 – указатель на вторую группу;

ranks2 – массив соответствующих номеров во второй группе или MPI_UNDEFINED, если такое соответствие отсутствует.

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

C:

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

Фортран:

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
INTEGER GROUP1, GROUP2, RESULT, IERROR

Аргументами функции являются:

group1 – указатель на первую группу;

group2 – указатель на вторую группу;

result – результат сравнения.

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

C:

*int MPI_Group_free (MPI_Group *group);*

Фортран:

MPI_GROUP_FREE(GROUP, IERROR)
INTEGER GROUP, IERROR

Аргументом функции является указатель на группу group.

Как и любой другой объект, созданный в памяти, группа должна быть удалена разработчиком после ее использования. Функция MPI_Group_free отправляет группу на удаление. Указатель на group устанавливается в состояние

MPI_GROUP_NULL. Любая операция, использующая в это время удаляемую группу, завершится нормально.

Коммуникаторы групп

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

Сразу после вызова функции MPI_Init создаются следующие коммуникаторы:

MPI_COMM_WORLD – коммуникатор, объединяющий все процессы параллельной программы;

MPI_COMM_NULL – значение, используемое для ошибочного коммуникатора;

MPI_COMM_SELF – коммуникатор, включающий только вызвавший процесс.

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

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

C:

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

Фортран:

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)

INTEGER COMM1, COMM2, RESULT, IERROR

Аргументами функции являются:

comm1 – коммуникатор первой группы;

comm2 – коммуникатор второй группы;

result – результат сравнения.

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

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

Результат MPI_SIMILAR имеет место, если члены группы обоих коммуникаторов являются одинаковыми, но порядок их нумерации различен. Во всех остальных случаях выдается результат MPI_UNEQUAL.

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

C:

int MPI_Comm_create (MPI_Comm comm, MPI_Group group,

*MPI_Comm *newcomm);*

Фортран:

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)

INTEGER COMM, GROUP, NEWCOMM, IERROR

Аргументами функции являются:

comm – коммуникатор группы;

group – группа, для которой создается коммуникатор;

newcomm – указатель на новый коммуникатор.

Функция MPI_Comm_create создает новый коммуникатор newcomm с коммуникационной группой, определенной аргументом group и новым контекстом. Из comm в newcomm не передается никакой кэшированной информации. Функция MPI_Comm_create возвращает MPI_COMM_NULL для процессов, не

входящих в group. Запрос неверен, если group не является подмножеством группы, связанной с comm. Заметим, что запрос должен быть выполнен всеми процессами в comm, даже если они не принадлежат новой группе.

Еще одна функция MPI_Comm_dup дублирует существующий коммуникатор.

C:

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

Фортран:

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)

INTEGER COMM, NEWCOMM, IERROR

Аргументами функции являются:

comm – коммуникатор группы;

newcomm – указатель на новый коммуникатор.

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

Операция MPI_Comm_dup возвращает в аргументе newcomm новый коммуникатор с той же группой, любой скопированной кэшированной информацией, но с новым контекстом.

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

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

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

C:

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

Фортран:

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

Аргументами функции являются:

comm – коммуникатор группы;

color – управление созданием подмножества;

key – управление назначением рангов;

newcomm – новый коммуникатор.

Функция MPI_Comm_split делит группу, связанную с comm, на непересекающиеся подгруппы, по одной для каждого значе- ния цвета color.

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

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

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

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

Функция MPI_Comm_split допускает более общее разделение группы на одну или несколько подгрупп с необязательным переупорядочением. Значение color должно быть неотрицательным.

Функция MPI_Comm_split – это чрезвычайно мощный механизм для разделения единственной коммуникационной группы процессов на k подгрупп, где k неявно выбрано пользователем (количеством цветов для раскраски про-

цессов). Полученные коммуникаторы будут не перекрывающимися. Такое деление полезно для организации иерархии вычислений, например для линейной алгебры. Чтобы преодолеть ограничение на перекрытие создаваемых коммуникаторов, можно использовать множественные обращения к MPI_Comm_split. Этим способом можно создавать множественные перекрывающиеся коммуникационные структуры.

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

Если значение ключа для всех процессов данного цвета сделано нулевым, то это означает, что порядок номеров процессов в новом коммуникаторе безразличен. Аргумент color не может иметь отрицательного значения, чтобы не конфликтовать со значением, присвоенным MPI_UNDEFINED.

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

C:

*int MPI_Comm_free (MPI_Comm *comm);*

Фортран:

MPI_COMM_FREE(COMM, IERROR)

INTEGER COMM, IERROR

Аргументом функции является удаляемый коммуникатор группы comm.

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

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

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

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

Функции коллективного взаимодействия

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

Особенности коллективных функций:

- коллективные функции не используются для связи процессов типа точка-точка;
- коллективные функции выполняются в режиме с блокировкой;
- возврат из коллективной функции после завершения передачи в каждом процессе происходит независимо от завершения передачи в других процессах;
- количество и объем принимаемых сообщений должно быть равно количеству и объему отправляемых сообщений;
- типы элементов отправляемых и принимаемых сообщений должны совпадать или быть совместимы;
- сообщения не имеют идентификаторов.
- к функциям коллективного взаимодействия относятся:
- синхронизация всех процессов одного коммуникатора с помощью функции MPI_Barrier;
- коллективные действия, в число которых входят:
- отправка информации от одного процесса всем остальным членам некоторой области связи с помощью функции MPI_Bcast;
- сборка (англ. gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (англ. root) процесса с помощью функций MPI_Gather и MPI_Gatherv;

- сборка распределенного массива в один массив с рассылкой его всем процессам некоторой области связи с помощью функций MPI_Allgather и MPI_Allgatherv;
- разбиение массива и рассылка его фрагментов (англ. scatter) всем процессам области связи с помощью функций MPI_Scatter и MPI_Scatterv;
- совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами в свой буфер приема с помощью функций MPI_Alltoall и MPI_Alltoallv.
- глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
- с сохранением результата в адресном пространстве одного процесса с помощью функции MPI_Reduce;
- с рассылкой результата всем процессам с помощью функции MPI_Allreduce;
- совмещенная операция Reduce/Scatter с помощью функции MPI_Reduce_scatter;
- префиксная редукция с помощью функции MPI_Scan.

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

1. Простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов.
2. «Векторный» вариант, который предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты функций имеют постфикс «v» в конце имени функции.

Основы параллельного программирования на OpenMP

Технология OpenMP является на сегодняшний день самым популярным средством для реализации параллелизма на системах с общей для языков С и Фортран. Разработчику предоставляется набор директив, функций и переменных окружения для создания параллельной версии приложения на основе последовательной реализации. Принцип OpenMP нацелен на то, чтобы пользователь имел один вариант программы для параллельного и последовательного выполнения.

Разработкой стандарта занимается некоммерческая организация OpenMP ARB (Architecture Review Board), в которую вошли представители крупнейших компаний – разработчиков SMP-архитектур и программного обеспечения. OpenMP поддерживает работу с языками Фортран и C/C++. Первая спецификация для языка Фортран появилась в октябре 1997 года, а спецификация для языка Си/Си++ – в октябре 1998 года. На данный момент последняя официальная спецификация стандарта – OpenMP 4.0 принята в июле 2013 года, и в отличие от всех предыдущих версий, позволяет реализовывать параллелизм не только по задачам, но и по данным.

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах в модели общей памяти.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор «подчиненных» (slave) потоков, и задача распределяется между ними.

Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами, причём количество процессоров не обязательно должно быть больше или равно количеству потоков. Тем не менее, в большинстве случаев для наибольшей эффективности выполнения приложения количество потоков равно количеству доступных ядер процессоров.

Существующие и по сегодняшний день предшественники OpenMP, напрмер, POSIX-интерфейс для организации потоков Pthreads, не так хорошо подходит для практического параллельного программирования. В частности, отсутствующая поддержка языка Фортран, слишком низкоуровневое программирование, отсутствие параллелизма по данным делает его непривлекательным. Кроме того, сам механизм потоков изначально разрабатывался не для це-

лей организации параллелизма и активно использоваля на системах с одним процессором. В свою очередь, OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads, в которой используется близкая терминология и модель программирования, динамически порождаемые потоки, общие и разделяемые данные, а так же средства для синхронизации.

В терминах Pthreads, любой процесс состоит из нескольких потоков управления, которые имеют общее адресное пространство, разные потоки команд и раздельные стеки. В простейшем случае процесс состоит из одного потока. Потоки иногда называют также нитями.

Одним из важных достоинств OpenMP является возможность реализации инкрементального параллелизма, то есть разработчик может постепенно находить необходимые участки кода и делать их параллельными с помощью механизма директив. Таким образом, нераспараллеленная часть постепенно становится всё меньше. Этот подход значительно облегчает процесс портирования последовательных приложений, а также делает проще процесс отладки и оптимизации.

Основные понятия

Стандарт OpenMP поддерживается на сегодняшний день большим количеством компиляторов. Тем не менее, нужно знать, какую именно версию они поддерживают. Версию 3.0 поддерживают пракктически все современные компиляторы. Для использования OpenMP необходимо компилировать исходный код с использованием специального ключа (например, в icc/ifort используется ключ компилятора –openmp, в gcc /gfortran –fopenmp, Sun Studio -xopenmp, в Visual C++ - /openmp, в PGI -mp). В этом случае компилятор интерпретирует директивы OpenMP, а на этапе линковки используются соответствующие библиотеки. При использовании компиляторов, не поддерживающих OpenMP, директивы игнорируются.

Любой компилятор, поддерживающий какой-либо стандарт OpenMP, определяет макрос _OPENMP, который может использоваться для условной компиляции. Этот макрос определён в формате yyyymm, где yyyy и mm – цифры года и месяца, когда был принят поддерживаемый стандарт OpenMP. Например, компилятор, поддерживающий стандарт OpenMP 3.0, определяет

_OPENMP в 200805, а поддерживающий стандарт OpenMP 4.0 – в значение 201306.

Примеры условной компиляции и определения поддержки OpenMP приведены в примере:

C:

```
#include <stdio.h>
int main(){
#ifdef _OPENMP
 printf("OpenMP is supported\n");
#endif
}
```

Фортран:

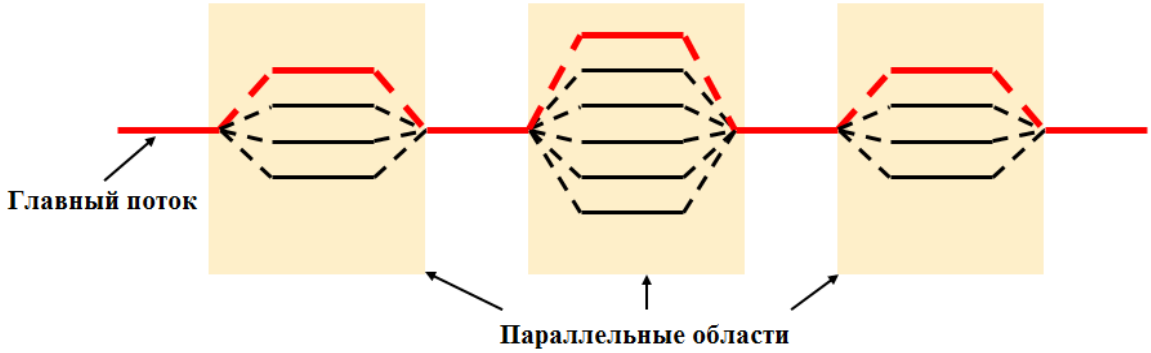
```
 program example
#ifdef _OPENMP
 print *, "OpenMP is supported"
#endif
 end
```

Для условной компиляции кода на языке Фортран строки могут также начинаться с пары символов !$, C$ или *$. В этом случае компилятор, поддерживающий OpenMP, заменит пару этих символов на два пробела.

Следующий пример показывает использование такого варианта условной компиляции.

```
 program example
!$ print *, "OpenMP is supported"
 end
```

Программа начинается с последовательной области – сначала работает один поток, при входе в параллельную область порождается ещё некоторое число потоков, между которыми в дальнейшем распределяются части кода:



По завершении параллельной области все потоки, кроме одного (потока-мастера), завершаются, и начинается последовательная область. В программе может быть любое количество параллельных и последовательных областей, а так же вложенные параллельные области. В отличие от работы с процессам, затраты с потоками намного меньше, поэтому частые порождения и завершения потоков существенно не меняют время выполнения программы.

Один из основных моментов при работе с потоками – балансировка нагрузки. Эффективное приложение должно равномерно загружать все потоки, а, соответственно, и ядра процессора, полезной работой. Реализуется это через различные механизмы OpenMP. Стоит отметить, что при неправильной загрузке потоков, скорость работы параллельной версии приложения может оказаться даже ниже последовательной.

Не менее существенным моментом является также необходимость синхронизации доступа к общим данным. С появлением множества потоков возникает конфликт при одновременном несогласованном доступе к данным, работа с которыми ранее осуществлялась только одним потоком. Поэтому значительная часть функциональности OpenMP предназначена для осуществления различных операций синхронизации.

OpenMP не выполняет синхронизацию доступа различных потоков к одним и тем же файлам. Если это необходимо для корректности программы, пользователь должен явно использовать директивы синхронизации или соответствующие библиотечные функции. При доступе каждому потоку к своему файлу никакая синхронизация не требуется.

Как отмечалось ранее, большая часть функционала OpenMP реализована через использование директив в коде. Разработчик явно вставляет их в нужные места и компилирует проект с поддержкой OpenMP, что позволяет выполнять программу параллельно.

Директивы OpenMP на языке Фортран оформляются комментариями и начинаются с комбинации символов !OMP, COMP или *$OMP, а в языке C – указаниями препроцессору, начинающимися с #pragma omp. Использование ключевого слова omp исключает возможность случайного совпадения имён директив с другими именами.

C:

#pragma omp directive-name [опция[[,] опция]...]

Фортране:

!$OMP directive-name [опция[[,] опция]...]
C$OMP directive-name [опция[[,] опция]...]
**$OMP directive-name [опция[[,] опция]...]*

Обычно директива действует на один оператор или блок, перед которым она расположена в исходном коде. В OpenMP такие операторы или блоки называются ассоциированными с директивой. Ассоциированный блок должен иметь одну точку входа в начале и одну точку выхода в конце. Кроме того, для директивы можно задавать список дополнительных опций, при этом их порядок в описании несущественен.

Директивы OpenMP версии 4.0 можно разделить на 5 категорий: определение параллельной области, распределение работы, синхронизация, векторизация, операции выгрузки на сопроцессор (offloading). Каждая директива может иметь несколько дополнительных атрибутов – опций (clause). Отдельно специфицируются опции для назначения классов переменных, которые могут быть атрибутами различных директив.

Для использования функции библиотеки OpenMP в языке C, необходимо подключить заголовочный файл omp.h, для программ на языке Фортран – файл omp_lib.h или модуль omp_lib. В случае использования только директив OpenMP, эти файлы не требуются. Все функции, используемые в OpenMP, начинаются с префикса omp_ и записываются строчными буквами, что критично для языка C.

Если компилятор не поддерживает OpenMP, то проблем с компиляцией кода всё равно не возникнет. Директивы будут просто игнорироваться, и мы получим последовательное приложение. Если же использовались вызовы функ-

ций, можно прилинковать специальную библиотеку, которая определит для каждой функции соответствующую «заглушку» (stub). Например, в компиляторе Intel соответствующая библиотека подключается заданием ключа –openmp-stubs.

После успешной компиляции параллельного приложения, необходимо запустить его на требуемом количестве процессоров или ядер. Для этого задается количество потоков, выполняющих параллельные области программы, через переменную окружения OMP_NUM_THREADS. Например, в Linux в командной оболочке bash это можно сделать при помощи следующей команды:

export OMP_NUM_THREADS=n

После запуска начинает работать один поток, а в каждой параллельной области одна и та же программа будет выполняться набором *n* потоков.

По аналогии с MPI, в OpenMP для определения времени работы приложения предусмотрены функции для работы с системным таймером.

Функция omp_get_wtime() возвращает в вызвавшем потоке астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом.

C:

double omp_get_wtime(void);

Фортран:

double precision function omp_get_wtime()

Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка.

Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменён за время существования процесса. Таймеры в разных потоках не синхронизированы и могут возвращать различные значения.

Функция omp_get_wtick() возвращает в вызвавшем потоке разрешение таймера в секундах. Это время можно рассматривать как меру точности таймера.

C:

double omp_get_wtick(void);

Фортран:

double precision function omp_get_wtick()

Пример показывает применение функций omp_get_wtime() и omp_get_wtick() .

C:

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
 double start, end, tick;
 start = omp_get_wtime();
 end = omp_get_wtime();
 tick = omp_get_wtick();
 printf("Время на замер времени %lf\n", end-start);
 printf("Точность таймера %lf\n", tick);
}
```

Фортран:

```
 program example
 include "omp_lib.h"
 double precision start, end, tick
 start = omp_get_wtime()
 end = omp_get_wtime()
 tick = omp_get_wtick()
 print *, "Время на замер времени ", end-start
 print *, "Точность таймера ", tick
 end
```

Производится замер начального времени и конечного времени. Разность времён даёт время, потраченное на замер времени. Кроме того, измеряется точность системного таймера.

Базовые директивы

Как отмечалось ранее, поток-мастер и только он исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные потоки. После порождения каждый поток получает свой уникальный номер, причем поток-мастер всегда имеет номер 0.

В параллельной области все переменные программы разделяются на два класса: общие (SHARED) и локальные (PRIVATE). Общая переменная всегда существует лишь в одном экземпляре для всей программы и доступна всем потокам под одним и тем же именем. Объявление же локальной переменной вызывает порождение своего экземпляра данной переменной для каждого потока. Изменение потоком значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других потоках. Схематично принцип работы с памятью и данными может быть представлен так:

Базовая директива OpenMP задаётся при помощи parallel (parallel ... end parallel), и определяет параллельную область:

C:

#pragma omp parallel [опция[[,] опция]...]

Фортран:

!$omp parallel [опция[[,] опция]...]

<код параллельной области>

!$omp end parallel

Для директивы можно задавать дополнительные опции:

- if(условие) – условное выполнение параллельной области. Вхождение в параллельную область осуществляется только при удовлетворении некоторого условия. В противном случае ди-

ректива не срабатывает и продолжается обычное выполнение программы;

- default(private|firstprivate|shared|none) – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс private, firstprivate или shared соответственно. firstprivate так же определяет все переменные, которым явно не назначен класс, как локальные (private), но ещё и инициализирует их значениями, которые были у переменных до входа в параллельную область. none означает, что всем переменным в параллельной области класс должен быть назначен явно; в языке C задаются только варианты shared или none;
- num_threads (целочисленное выражение) – явное задание количества потоков, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции omp_set_num_threads(), или значение переменной OMP_NUM_THREADS;
- private(список) – задаёт список переменных, для которых порождается локальная копия в каждом потоке; начальное значение локальных копий переменных из списка не определено;
- firstprivate(список) – задаёт список переменных, для которых порождается локальная копия в каждом потоке; локальные копии переменных инициализируются значениями этих переменных в потоке-мастере;
- shared(список) – задаёт список переменных, общих для всех потоков;
- copyin(список) – задаёт список переменных, объявленных как threadprivate, которые при входе в параллельную область инициализируются значениями соответствующих переменных в потоке-мастере;
- reduction(оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её анало-

ги); над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; оператор это: для языка C – +, *, -, &, |, ^, &&, ||, для языка Фортран – +, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску. Тем не менее, некоторые компиляторы реализуют возможность контролировать порядок суммирования, Например, компилятор Intel умеет это делать с помощью переменной окружения KMP_DETERMINISTIC_REDUCTION=1.

Ниже представлен пример, показывающий использование директивы parallel. В результате выполнения поток-мастер напечатает текст "Последовательная область 1", затем по директиве parallel порождаются новые потоки, каждый из которых напечатает текст "Параллельная область", затем порождённые потоки завершаются и оставшийся поток-мастер напечатает текст "Последовательная область 2".

C:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 printf("Последовательная область 1\n");
#pragma omp parallel
 {
 printf("Параллельная область\n");
 }
 printf("Последовательная область 2\n");
}
```

Фортран:

```
 program example
 print *, "Последовательная область 1"
!$omp parallel
 print *, "Параллельная область"
!$omp end parallel
 print *, "Последовательная область 2"
```

```
 end
```

Ещё один пример демонстрирует применение опции reduction. Производится подсчет общего количества порождённых потоков. Каждый поток инициализирует локальную копию переменной count значением 0. Далее, каждый поток увеличивает значение собственной копии переменной count на единицу и выводит полученное число. На выходе из параллельной области происходит редукция значений переменных count по всем потокам, и полученная величина становится новым значением переменной count в последовательной области.

C:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 int count = 0;
#pragma omp parallel reduction (+: count)
 {
 count++;
 printf("Текущее значение count: %d\n", count);
 }
 printf("Число потоков: %d\n", count);
}
```

Фортран:

```
program example
 integer count
 count=0
!$omp parallel reduction (+: count)
 count=count+1
 print *, "Текущее значение count: ", count
!$omp end parallel
 print *, "Число потоков: ", count
 end
```

Для того чтобы выполнить какой-либо участок параллельной области только один раз, нужно использовать директиву single (single ... end single).

C:

```
#pragma omp single [опция [[,] опция]...]
```

Фортран:

!$omp single [опция [[,] опция]...]

<код для одного потока>

!$omp end single [опция [[,] опция]...]

Возможные опции во многом схожи с аналогичными у директивы parallel, например, такие как private, firstprivate. Интересны опции copyprivate и nowait.

copyprivate(список) – после выполнения потока, содержащего конструкцию single, новые значения переменных списка будут доступны всем одноименным частным переменным (private и firstprivate), описанным в начале параллельной области и используемым всеми её потоками.

nowait отменяет неявную синхронизацию для некоторых директив, позволяя, тем самым, потокам, закончившим выполнение, не дожидаться окончания выполнения других потоков и продолжать выполнение.

В программах на языке С все опции указываются у директивы single, а в программах на языке Фортран опции private и firstprivate относятся к директиве single, а опции copyprivate и nowait – к директиве end single.

Какой именно поток будет выполнять выделенный участок программы, не специфицируется. Только один поток будет выполнять данный фрагмент, а все остальные будут ожидать завершения его работы, если только не указана опция nowait. Схематично принцип работы области single для случая 4 потоков можно показать так:

Необходимость использования директивы single часто возникает при работе с общими переменными.

Директивы master (master ... end master) выделяют участок кода, который будет выполнен только потоком-мастером. Остальные потоки просто пропус-

кают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает.

C:

#pragma omp master

Фортран:

!$omp master

<код для потока-мастера>

!$omp end master

Как говорилось ранее, модель данных в OpenMP предполагает наличие как общей для всех потоков области памяти, так и локальной области памяти для каждого потока. Если несколько переменных одновременно записывают значение общей переменной без выполнения синхронизации или если как минимум один поток читает значение общей переменной и как минимум один поток записывает значение этой переменной без выполнения синхронизации, то возникает ситуация так называемой «гонки данных» (data race), при которой результат выполнения программы непредсказуем и может меняться от запуска к запуску.

По умолчанию, все переменные, порождённые вне параллельной области, при входе в эту область остаются общими (shared). Исключение составляют переменные, являющиеся счетчиками итераций в цикле. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными (private). Явно назначить класс переменных по умолчанию можно с помощью опции default.

В языке C статические (static) переменные, определённые в параллельной области программы, являются общими (shared). Динамически выделенная память также является общей, однако указатель на неё может быть как общим, так и локальным.

В языке Фортран по умолчанию общими (shared) являются элементы COMMON-блоков. Отдельные правила определяют назначение классов переменных при входе и выходе из параллельной области или параллельного цикла при использовании опций reduction, firstprivate, lastprivate, copyin.

Если в параллельной области встречается оператор цикла, то он будет выполняться всеми потоками текущей группы, то есть каждый поток выполнит

все итерации данного цикла. Естественно, это требуется далеко не всегда и для распределения итераций цикла между различными потоками можно использовать директиву for (do ... [end do]).

C:

#pragma omp for [опция [[,] опция]...]

Фортран:

!$omp do [опция [[,] опция]...]

<блок циклов do>

[!$omp end do [nowait]]

Эта директива относится к идущему следом за данной директивой блоку, включающему операторы for (do).

В список возможных опций входят уже рассмотренные ранее private, firstprivate, reduction, и ряд новых:

- lastprivate(список) – переменным, перечисленным в списке, присваивается результат с последней итерации цикла;
- schedule(type[, chunk]) – опция задаёт, каким образом итерации цикла распределяются между нитями;
- collapse(n) – опция указывает, что n последовательных вложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция collapse не задана, то директива относится только к одному непосредственно следующему за ней циклу;
- ordered – опция, говорящая о том, что в цикле могут встречаться директивы ordered; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;
- nowait – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих потоков: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция nowait позволяет потокам, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными. Если директива end do в явном виде не указана, то в

конце параллельного цикла синхронизация все равно будет выполнена.

На циклы накладываются достаточно жёсткие ограничения, вполне логично вытекающие из сути параллельного выполнения. Предполагается, что корректная программа не должна зависеть от порядка выполнения итераций, то есть от того, какой поток какую итерацию параллельного цикла выполняет. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции schedule, не должен изменяться в рамках цикла.

Вид циклов на языке С можно представить следующим образом:

for([целочисленный тип] i = инвариант цикла;

i {<,>,=,<=,>=} инвариант цикла;

i {+,-}= инвариант цикла)

Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций. Переменная, являющаяся счетчиком итераций распределяемого цикла, должна быть локальной, поэтому даже если она специфицирована общей, то при входе в цикл она всё равно будет использоваться как локальная каждым потоком. После завершения цикла значение этой переменной не определено, если она не указана в опции lastprivate.

Для распределения нагрузки между потоками используется параметр type опции schedule, который может задавать следующие типы:

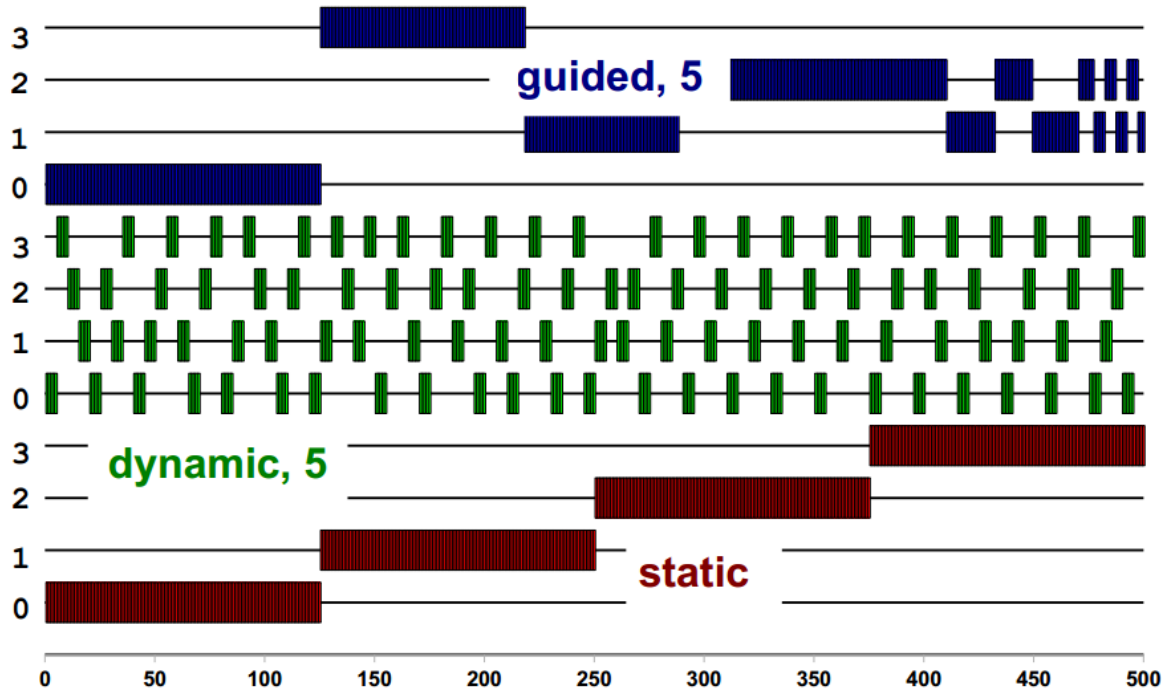
- static – блочно-циклическое распределение итераций цикла; размер блока – chunk. Первый блок из chunk итераций выполняет нулевой поток, второй блок – следующий и т.д. до последнего потока, затем распределение снова начинается с нулевого потока. Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера и полученные порции итераций распределяются между потоками. Например, для цикла в 16 итерация и с 4 потоками, мы будем иметь следующее распределение итераций (без установленного размера блока и равным 2):

поток	**0**	**1**	**2**	**3**
no chunk*	1-4	5-8	9-12	13-16

chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

- dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждый поток получает chunk итераций (по умолчанию chunk=1. Тот поток, который заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций. Освободившиеся потоки получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.
- guided – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству ещё не распределённых итераций, делённому на количество потоков, выполняющих цикл. Такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку потоков в некоторых случаях. Количество итераций в последней порции может оказаться меньше значения chunk.
    - auto – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр chunk при этом не задаётся.
    - runtime – способ распределения итераций выбирается во время работы программы по значению переменной среды OMP_SCHEDULE. Параметр chunk при этом не задаётся.

Диаграмма показывает, каким образом происходит распределение 500 итерация цикла по 4 потокам (от 0 до 3) при различных вариантах планировки:



Порой возникает необходимость параллельно выполнить действия, которые не являются итерациями цикла. Конечно, можно воспользоваться для этих целей простой директивой parallel, но тогда придется писать дополнительный код, чтобы различную работу распределить между потоками. Более просто эту задачу можно решить с помощью параллельных секций.

C:

#pragma omp sections [опция [[,] опция]...]
{
#pragma omp section
структурный блок
[#pragma omp section
структурный блок
...]
}

Фортран:

!$omp sections [опция [[,] опция]...]
c$omp section
структурный блок
[c$omp section

структурный блок

...]

c$omp end sections [nowait]

Возможно задавать опции private, firstprivate, lastprivate и reduction.

Каждая секция выполняется в отдельном потоке, что позволяет производить декомпозицию по коду. Точкой синхронизации является конец блока sections. В случае, когда необходимо чтобы основной поток не ждал завершения остальных потоков следует использовать условие nowait.

Синхронизация

Целый набор директив в OpenMP предназначен для синхронизации работы нитей.

Самый распространенный способ синхронизации в OpenMP – барьер. Его можно использовать с помощью директивы barrier.

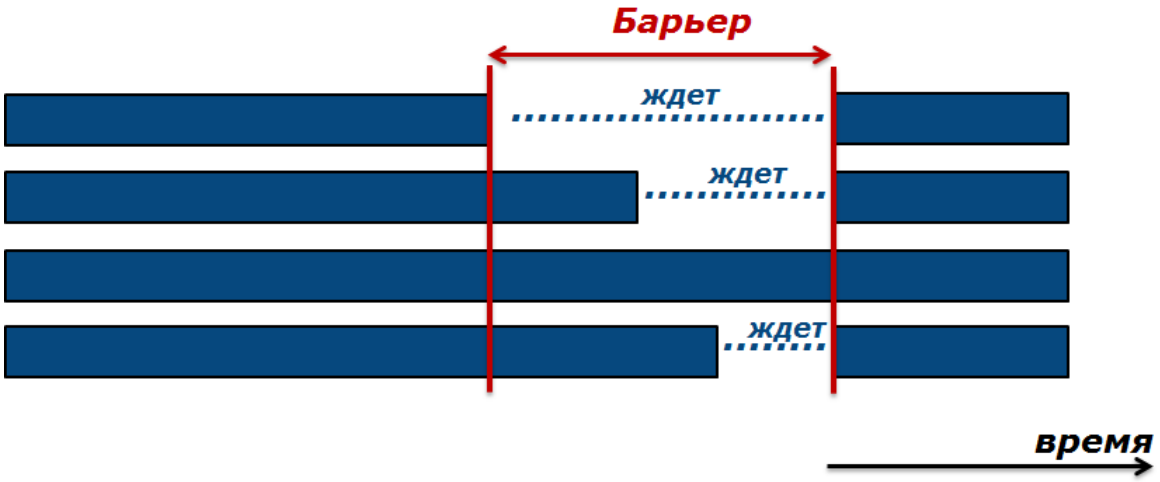
C:

#pragma omp barrier

Фортран:

!$omp barrier

Потоки, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все потоки не дойдут до этой точки программы, после чего продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые потоки завершили все порождённые ими задачи (task). Случай для 4 потоков и берьером показан на следующем рисунке:



С помощью директив critical (critical ... end critical) оформляется критическая секция программы.

С:

#pragma omp critical [(<имя_критической_секции>)]

Фортран:

!$omp critical [(имя)]

<код критической секции>

!$omp end critical [(имя)]

В каждый момент времени работу в критической секции может выполнять не более одного потока. Если критическая секция уже выполняется каким-либо потоком, то все другие потоки, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедший поток не закончит выполнение данной критической секции. Как только работавший поток выйдет из критической секции, один из заблокированных на входе потоков войдет в неё. Если на входе в критическую секцию стояло несколько потоков, то случайным образом выбирается один из них, а остальные заблокированные потоки продолжают ожидание.

Случай использования 4 потоков и критической секции представлен ниже:

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная sum является общей и оператор вида sum=sum+expr находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими потоками можно получить случай «гонкт данных» и, в итоге, некорректный результат. Чтобы избежать этой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой atomic.

С:

#pragma omp atomic

Фортран:

!$omp atomic

Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент потокам, кроме того, который выполняет операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.

Ещё один из вариантов синхронизации в OpenMP реализуется через механизм замков (locks). В качестве замков используются общие целочисленные переменные (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации.

Замок может находиться в одном из трёх состояний: неинициализированный, разблокированный или заблокированный. Разблокированный замок может быть захвачен некоторым потоком, при этом он переходит в заблокированное состояние. Только тот поток, который захватил замок, может его освободить, после чего замок возвращается в разблокированное состояние.

Различают два типа замков: простые и множественные. Множественный замок может многократно захватываться одним потоком перед его освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие коэффициента захваченности (nesting count). По сути это счетчик, который изначально устанавливается в ноль. При каждом следующем захватывании он увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

Для инициализации простого или множественного замка используются соответственно функции omp_init_lock() и omp_init_nest_lock().

C:

*void omp_init_lock(omp_lock_t *lock);*

*void omp_init_nest_lock(omp_nest_lock_t *lock);*

Фортран:

subroutine omp_init_lock(svar)

integer (kind=omp_lock_kind) svar

subroutine omp_init_nest_lock(nvar)

integer (kind=omp_nest_lock_kind) nvar

После выполнения функции замок переводится в разблокированное состояние. Для множественного замка коэффициент захваченности устанавливается в ноль.

Функции omp_destroy_lock() и omp_destroy_nest_lock() используются для переведения простого или множественного замка в неинициализированное состояние.

C:

*void omp_destroy_lock(omp_lock_t *lock);*

*void omp_destroy_nest_lock(omp_nest_lock_t *lock);*

Фортран:

subroutine omp_destroy_lock(svar)

integer (kind=omp_lock_kind) svar

subroutine omp_destroy_nest_lock(nvar)

integer (kind=omp_nest_lock_kind) nvar

Для захватывания замка используются функции omp_set_lock() и omp_set_nest_lock().

C:

*void omp_set_lock(omp_lock_t *lock);*

*void omp_set_nest_lock(omp_nest_lock_t *lock);*

Фортран:

subroutine omp_set_lock(svar)

integer (kind=omp_lock_kind) svar

subroutine omp_set_nest_lock(nvar)

integer (kind=omp_nest_lock_kind) nvar

Вызвавший эту функцию поток дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние.

Если множественный замок уже захвачен данным потоком, то нить не блокируется, а коэффициент захваченности увеличивается на единицу.

Для освобождения замка используются функции omp_unset_lock() и omp_unset_nest_lock().

C:

*void omp_unset_lock(omp_lock_t *lock);*

*void omp_unset_nest_lock(omp_lock_t *lock);*

Фортран:

subroutine omp_unset_lock(svar)

integer (kind=omp_lock_kind) svar

subroutine omp_unset_nest_lock(nvar)

integer (kind=omp_nest_lock_kind) nvar

Вызов этой функции освобождает простой замок, если он был захвачен вызвавшим потоком. Для множественного замка уменьшает на единицу коэф-

фициент захваченности. Если коэффициент станет равен нулю, замок освобождается. Если после освобождения замка есть потоки, заблокированные на операции, захватывающей данный замок, замок будет сразу же захвачен одним из ожидающих потоков.

Задачи

В стандарте OpenMP 3.0 впервые было введено понятие задачи (tasks), что позволило решить проблемы, возникающие при реализации параллелизма в определенных случаях. В качестве задачи может быть выбран некий блок операторов. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям.

Для выделения отдельной независимой задачи в OpenMP используется директива task (task ... end task).

C:

#pragma omp task [опция [[,] опция]...]

Фортран:

!$omp task [опция [[,] опция]...]

<код задачи>

!$omp end task

Возможные опции:

if(условие) – порождение новой задачи только при выполнении некоторого условия; если условие не выполняется, то задача будет выполнена текущим потоком и немедленно;

untied – в случае отложенного выполнения задача может быть продолжена любым потоком, выполняющим данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившим её потоком;

Опции default, private, firstprivate и shared используются, как и с другими директивами, для работы с данным.

Для гарантированного завершения в точке вызова всех запущенных задач используется директива taskwait.

C:

#pragma omp taskwait

Фортран:

!$omp taskwait

Поток, выполнивший данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данной нитью независимые задачи.

В стандартах 3.1 и 4.0 возможности по работе с задачами были существенно расширены, в частности, через добавление таких возможных опций к директиве task, как final, mergeable, depend. Не все компиляторы на сегодняшний день поддерживают последние стандарты OpenMP. Нужно внимательно проверять поддерживаемый функционал, чтобы избежать ошибок компиляции.

Векторизация

Реализуя параллелизм по данным, векторизация циклов на сегодняшний день является мощным инструментом в руках разработчиков для существенного повышения производительности. Выполняя одну операцию не над скаляром – одним элементом массива, а над несколькими элементами массива данных - вектором, мы можем существенно сократить время выполнения циклов. Принцип векторизации показан на примере обычного цикла, в котором суммируются два массива:

```
for (i=0;i<=MAX;i++)
    c[i]=a[i]+b[i];
```

Векторизация на сегодняшний день осуществляется средствами компилятора, который может генерировать специальные ассемблерные инструкции, реализующие векторные операции над данными. При этом важна поддержка этих инструкций и на аппаратном уровне в процессорах.

Первая технология, поддерживающая работу с векторами данных, называлась MMX - Multimedia Extensions. Процессоры, поддерживающие MMX, имели 64 битные регистры и позволяли работать с двумя 32 битными, четырьмя 16 битными или восемью 8 битными целыми числами за одну операцию. Было доступно 47 различных инструкций, которые делились на несколько

групп: перемещение данных, арифметические, сравнения, конвертации, логические, распаковки, сдвига и пустые инструкции получения состояния. Идея MMX исторически возникла от инструкций сопроцессора для обработки чисел с плавающей точкой. Устройство обработки чисел с плавающей точкой (FPU) было интегрировано в процессор 80486 DX и ввело 8 новых 80 битных регистра для хранения чисел с плавающей точкой. Итогом стало увеличение скорости работы с вещественными числами.

Для того, чтобы позволить использовать FPU в процессоре Pentium MMX и была предложена новая технология MMX. Ее идея заключалась в том, что в процессор добавлялись новые 8 регистров MM0-MM7 длиной 64 бита, которые адресовались к FPU регистрам и набор инструкций микропроцессора дополнялся рядом инструкций SIMD - Single Instruction, Multiple Data, работающих с этими регистрами и выполнявшими над ними целочисленные операции. Тем не менее, такой подход не позволял одновременно работать с целыми и вещественными числами.

Развитием идеи MMX стал набор новых инструкций, позволяющий работать с регистрами ещё большей длины - Streaming SIMD Extensions (SSE), реализованный в процессорах серии Pentium III. SSE включает в архитектуру процессора восемь 128-битных регистров XMM0- XMM7 и набор соответствующих инструкций для работы с данными. Каждый регистр может содержать четыре 32-битных значения с плавающей точкой одинарной точности. Следующее поколение SSE2 решило проблему поддержки всех типов данных. Начиная с неё, векторные инструкции работали как с целыми числами, так и с числами с плавающей точкой (одинарной и двойной точности). Дарее, на протяжении долгого времени увеличивалось число различных инструкций, и появлялись новые версии SSE: SSE3 (13 новых инструкций), SSSE3 (32 новых инструкций), SSE4.1 (47 новых инструкций), SSE4.2 (7 новых инструкций). Принципиальным стало очередное увеличение длины регистров до 256 бит и смена набора инструкций на Advanced Vector Extensions (AVX), которая произошла несколько лет назад. Роль векторизации значительно возросла, потому что возможный что прирост производительности существенно увеличился. В ближайшем будущем появится процессор с поддержкой новой технологии AVX512, увеличивающий длину регистров до 512 бит.

Рассмотрим процесс векторизации более подробно. В качестве примера возьмем следующий код:

```
for(i=0; i<*p; i++)
{
  A[i] = B[i]*C[i];
  sum = sum + A[i];
}
```

Перед тем, как выполнить автоматическую векторизацию кода, компилятор пытается проверить выполнение следующих условий:

- *p является инвариантом цикла;
- A, B и C являются инвариантами цикла;
- A[] не является другим именем для B[], C[] и/или sum (нет перекрытия по памяти между этими данными);
- sum не является другим именем для B[] и/или C[] (нет перекрытия по памяти между этими данными);
- операция "+" является ассоциативной;
- ожидается ускорение векторной версии данного кода по отношению к скалярной.

Если ответ на все эти вопросы положителен, тогда компилятор выполняет автоматическую векторизацию. Однако компилятор не всегда может дать однозначно положительный ответ на один или несколько подобных вопросов в силу сложности участка кода. И в этом случае программист может помочь компилятору принять правильное решение.

Например, для компилятора часто сложным является ответ на вопрос о том, что массив A[] не перекрываются с массивами B[] и C[]. Для того чтобы отразить это в синтаксисе языка, можно объявить указатель A с ключевым словом restrict:

```
float* restrict A;
```

Такое объявление говорит компилятору о том, что массив A[] не перекрывается с другими массивами.

В определенных случаях компилятор не способен осуществить векторизацию, поэтому появился целый ряд директив, позволяющих помогать ему в векторизации циклов. Долгое время они не были стандартизированы и реализа-

ция отличалась в зависимости от компилятора. Стандарт OpenMP 4.0 решил данную проблему, введя ряд директив для осуществления векторизации.

Директива simd позволяет осуществлять векторизацию циклов и отключать дополнительные проверки компилятора. Нужно быть аккуратным с использованием данной директивы, потому что ответственность за векторизацию цикла ложиться теперь на разработчика.

C:

#pragma omp simd [опция [[,] опция]...]

Фортран:

!$omp simd [опция [[,] опция]...]

<код задачи>

!$omp end simd

Возможные опции:

- safelen(n) – указывается допустимая максимальная длина вектора, при которой возможна векторизация;
- linear(список[:n]) – указывает, что проход по данным из списка осуществляется линейно с шагом n;
- aligned(список[:n]) – показывает, что указанные в списке данные выравнены по n байт.

Опции private, lastprivate, reduction и collapse аналогичны рассмотренным ранее для других директив.

Использование OpenMP

Любая система на сегодняшний день является как минимум многоядерной, поэтому для повышения производительности приложения нужно использовать все доступные вычислительные ядра. В большинстве случаев используется один поток на каждое ядро.

Для реализации параллельной версии, нужно на первом этапе определить ресурс параллелизма программы, то есть, найти в ней участки, которые могут выполняться независимо разными потоками. Если таких участков относительно немного, то для распараллеливания чаще всего используются конструкции, задающие неитеративный параллелизм, например, параллельные секции.

Но чаще всего наибольший ресурс параллелизма в программах сосредоточен в циклах. Если между итерациями некоторого цикла нет зависимостей по данным, то их можно распределить по потокам для одновременного исполнения. Различные способы распределения итераций позволяют добиваться максимально равномерной загрузки потоков, между которыми распределяются итерации цикла. Кроме того, нужно не забывать о векторизации для максимально эффективного использования аппаратной платформы.

Статический способ распределения итераций позволяет уже в момент написания программы точно определить, какому потоку достанутся какие итерации. Однако он не учитывает текущей загруженности ядер, соотношения времён выполнения различных итераций и некоторых других факторов. Эти факторы в той или иной степени учитываются динамическими способами распределения итераций.

Обмен данными в OpenMP происходит через общие переменные. Это приводит к необходимости разграничения одновременного доступа разных нитей к общим данным. Для этого предусмотрены достаточно развитые средства синхронизации. При этом нужно учитывать, что использование излишних синхронизаций может существенно замедлить программу.

Использование идеи инкрементального распараллеливания позволяет при помощи OpenMP быстро получить параллельный вариант программы. Взяв за основу последовательный код, пользователь шаг за шагом добавляет новые директивы, описывающие новые параллельные области. Нет необходимости сразу распараллеливать всю программу, её создание ведется последовательно, что упрощает и процесс программирования, и отладку.

OpenMP может и должен использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания на одном узле как по данным, так и по задачам.

Пример подготовки существующего приложения к запуску на высокопроизводительной системе

Было портировано существующее приложение, написанное на языке программирования Фортран, для последующего запуска на системе с распределенной памятью – кластере ННГУ.

Достаточно большое количество существующего на сегодняшний день кода проектировалось под ОС Windows*, в том числе и рассматриваемое в данном примере. В то же время, практически все высокопроизводительные системы работают под управлением Linux. Таким образом, одним из ключевых шагов к перепроектированию приложения является портирование под Linux. Для этого важно понимать, что процесс компиляции и линковки могут значительно различаться на этих ОС с точки зрения разработчика.

На Windows стандартным решением была и остаётся среда разработки Microsoft Visual Studio. Став преемником Microsoft PowerStation, она так же даёт возможность пользоваться всеми удобствами графического интерфейса при создании приложений. Стоит отметить, что распространённым решением является использование компилятора компании Intel, который интегрируется в данную среду разработки. Будем отталкиваться от того факта, что существующий проект Visual Studio успешно собирается именно компилятором Intel, и наша первая цель заключается в том, чтобы собрать проект под Linux с помощью соответствующего компилятора.

Для успешного портирования на Linux необходимо хорошо понимать основные способы компиляции. Возможно использование отдельной команды для сборки каждого файла, но если проект достаточно большой и содержит сотни исходных файлов, такой способ неэффективен. Кроме того, если возникнет необходимость перекомпиляции файла, от которого существуют зависимости, придётся пересобирать весь проект. Классическим решением на Linux является использование утилиты GNU make и специальных make-файлов. Таким образом, первым шагом на пути к портированию приложению является понимание основ работы данной утилиты.

Утилита Make

Make — утилита, автоматизирующая процесс компиляции исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки, и входит в большинство дистрибутивов GNU/Linux. Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

Формат, в котором используется команда:

make [-f make-файл] [цель] ...

Файл ищется в текущей директории. Если опция -f не указана, используется имя по умолчанию для make-файла — Makefile (однако, в разных реализациях make кроме этого могут проверяться и другие файлы, например GNUmakefile).

make открывает make-файл, считывает правила и выполняет команды, необходимые для создания указанной цели. Стандартные цели для сборки дистрибутивов GNU:

all — выполнить сборку пакета

install — установить пакет из дистрибутива (производит копирование исполняемых файлов, библиотек и документации в системные директории)

uninstall — удалить пакет (производит удаление исполняемых файлов и библиотек из системных директорий)

clean — очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы созданные в процессе компиляции)

distclean — очистить все созданные при компиляции файлы и все вспомогательные файлы созданные утилитой ./configure в процессе настройки параметров компиляции дистрибутива

По умолчанию make использует самую первую цель в make-файле.

make-файл состоит из правил и переменных. Правила имеют следующий синтаксис:

цель1 цель2 ...: реквизит1 реквизит2 ...

команда1

команда2

...

Правило представляет собой набор команд, выполнение которых приведёт к сборке файлов-целей из файлов-реквизита.

Правило сообщает make, что файлы, получаемые в результате работы команд (цели) являются зависимыми от соответствующих файлов-реквизита. make никак не проверяет и не использует содержимое файлов-реквизита, однако, указание списка файлов-реквизита требуется только для того, чтобы make убедилась в наличии этих файлов перед началом выполнения команд и для отслеживания зависимостей между файлами.

Обычно цель представляет собой имя файла, который генерируется в результате работы указанных команд. Целью также может служить название некоторого действия, которое будет выполнено в результате выполнения команд (например, цель clean в make-файлах для компиляции программ обычно удаляет все файлы, созданные в процессе компиляции).

Строки, в которых записаны команды, должны начинаться с символа табуляции.

Рассмотрим несложную программу на Си. Пусть программа program состоит из пары файлов кода — main.c и lib.c, а также из одного заголовочного файла — defines.h, который подключён в обоих файлах кода. Поэтому, для создания program необходимо из пар (main.c defines.h) и (lib.c defines.h) создать объектные файлы main.o и lib.o, а затем скомпоновать их в program. При сборке вручную требуется исполнить следующие команды:

icc -c main.c

icc -c lib.c

icc -o program main.o lib.o

Если в процессе разработки программы в файл defines.h будут внесены изменения, потребуется перекомпиляция обоих файлов и линковка, а если изменим lib.c, то повторную компиляцию main.o можно не выполнять.

Таким образом, для каждого файла, который мы должны получить в процессе компиляции, нужно указать, на основе каких файлов и с помощью какой команды он создаётся. Программа make на основе этих данных собирает из правильную последовательность команд для получения требуемых результирующих файлов и инициирует создание требуемого файла только в случае, если такого файла не существует, или он старше, чем файлы, от которых он зависит.

Если при запуске make явно не указать цель, то будет обрабатываться первая цель в make-файле, имя которой не начинается с символа «.».

Для программы program достаточно написать следующий make-файл:

```
program: main.o lib.o
    cc -o program main.o lib.o
main.o lib.o: defines.h
```

Стоит отметить ряд особенностей. В имени второй цели указаны два файла и для этой же цели не указана команда компиляции. Кроме того, нигде явно не указана зависимость объектных файлов от «*.c»-файлов. Дело в том, что программа make имеет предопределённые правила для получения файлов с определёнными расширениями. Так, для цели-объектного файла (расширение «.o») при обнаружении соответствующего файла с расширением «.c» будет вызван компилятор «cc -c» с указанием в параметрах этого «.c»-файла и всех файлов-зависимостей.

Синтаксис для определения переменных:

переменная = значение

Значением может являться произвольная последовательность символов, включая пробелы и обращения к значениям других переменных. С учётом сказанного, можно модифицировать наш make-файл следующим образом:

```
OBJ = main.o lib.o
program: $(OBJ)
    cc -o program $(OBJ)
$(OBJ): defines.h
```

Нужно отметить, что вычисление значений переменных происходит только в момент использования (используется так называемое ленивое вычисление). Например, при сборке цели all из следующего make-файла на экран будет выведена строка «Huh?».

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?
all:
```

echo $(foo)

Предположим, что к проекту добавился второй заголовочный файл lib.h, который включается только в lib.c. Тогда make-файл увеличится ещё на одну строчку:

lib.o: lib.h

Таким образом, один целевой файл может указываться в нескольких целях. При этом полный список зависимостей для файла будет составлен из списков зависимостей всех целей, в которых он участвует, создание файла будет производиться только один раз.

Компиляция

Основной трудностью для существующего, компилирующегося на Windows проекта, является создание рабочего make-файла. С ним останется лишь выбрать необходимый компилятор и запустить команду make.

Готовых решений для разработчиков на Фортране, которые бы автоматически преобразовали проекты Visual Studio в make-файлы, на данный момент, не существует. Тем не менее, на странице fortran.com можно найти полезный скрипт на языке Perl, который поможет сделать очень большой объём работы.

Скрипт может быть найден по адресу:

http://www.fortran.com/makemake.perl

Он не позволяет преобразовывать проекты VS, а только автоматически их генерирует из тех исходных файлов, которые находит.

Таким образом, на первом этапе необходимо просто собрать все исходные файлы, используемые в проекте на Windows, в одну папку с данным скриптом. Работает он со стандартными расширениями типа *.f90 *.f *.F, но так же есть возможность его модифицировать для использования с другими расширениями.

Скрипт поможет сгенерировать костяк make-файла, а главное – отследит зависимости между файлами. Очень часто та или иная функциональность в существующих приложениях на языке Фортран реализовывается с помощью модулей, которые впоследствии используются в других местах кода через оператор USE. В случае, если происходит изменение модуля, необходима перекомпиляция всех зависимых частей и соответствующих файлов, а значит это долж-

но быть прописано в make-файле. Большую часть работы скрипт сделает сам, но правильность его работы необходимо будет проверить самостоятельно. Проблемы заключается в том, что возможны циклические зависимости, которые он не умеет отслеживать, и эту задачу придётся решать в ручном режиме, удаляя лишнее из файла. Для этого нам помогут знания о структуре файла и принципе работы утилиты, рассмотренные ранее.

После создания make-файла, утилита будет вызывать компилятор для сборки приложения. Необходимо помнить, что при переходе на другую ОС могут меняться названия ключей компиляции, поэтому придется убедиться, что все ключи работают правильно и компилятор их распознал. Кроме того, компилятор на Linux может иметь более жесткие требования к написанному коду, например, к точному соответствию типов данных, передаваемых в функции.

После решения всех описанных вопросов, приложение будет успешно создано на Linux. Самый первый шаг на пути к использованию его на кластере будет сделан. Далее, анализируя алгоритм и осуществляя профилировку приложения с помощью специальных средств, например, Intel® Vtune™ Amplififer XE, можно найти те места в коде, где есть ресурс для параллелизма, и применять различные модели параллельного программирования. Как отмечалось в предыдущих разделах, для реализации параллелизма по данным и по задачам можно использовать описанный стандарт OpenMP. Для распределения вычислений по узлам кластера используется MPI.

Контрольные вопросы

1. Какие типы параллелизма существуют?
2. Как можно классифицировать вычислительные системы?
3. Что такое поток в параллельном программировании?
4. Чем отличается кластер от обычного многоядерного компьютера?
5. Какие атрибуты есть у каждого посылаемого сообщения в MPI?
6. Для чего нужна блокировка при отправке/приеме сообщений в MPI?
7. Возможно ли определение атрибутов сообщения MPI, не принимая его?
8. Что означает завершение операции для различных видов пересылки данных с блокировкой в MPI?
9. В чем отличие коллективных операций от операций типа точка-точка в MPI?
10. В чем отличие группы процессов и коммуникаторов в MPI?
11. Какие группы процессов существуют при запуске приложения в MPI?
12. Как создать новую группу из процессов 3, 4, и 7 коммуникатора MPI_COMM_WORLD?
13. Может ли приложение на OpenMP состоять только из параллельных областей?
14. В чем отличие потока-мастера от остальных потоков в OpenMP?
15. Чем отличаются директивы single и master?
16. Может ли одна и та же переменная в OpenMP быть общей в одной части программы, и локальной – в другой?
17. Что происходит, если несколько потоков в OpenMP обращаются к одной и той же переменной? Есть ли зависимость от типа обращения (чтение/запись)?
18. Как можно распределять работу в параллельном цикле for OpenMP?
19. Что такое векторизация?
20. Зачем нужна утилита make? Существуют ли аналоги для ОС Windows?

Литература

1. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М., Изд-во МГУ, 2009.- 77 с.
2. Сальников А.М., Ярошенко Е.А., Гребенник О.С., Спиридонов С.В. Введение в параллельные вычисления. Основы программирования на языке СИ с использованием интерфейса MPI. - М.: ИПУ РАН, 2009 - 123 с.
3. Богачев В.В. Основы параллельного программирования.– М.: БИНОМ. Лаборатория знаний, 2003.- 342 с.
4. Воеводин В.В., Воеводин Вл. В. Параллельные вычисления, – СПб., БХВ-Петербург, 2002.– 608 с.
5. Антонов А.С. Введение в параллельные вычисления (методическое пособие), – М., Изд-во Физфака МГУ, 2002.- 70 с.
6. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. - М.: Изд-во "Вильямс", 2003. - 512 с.
7. Миллер Р., Боксер Л. Последовательные и параллельные алгоритмы: Общий подход. - М.: БИНОМ. Лаборатория знаний, 2006. - 406 с.

Программное обеспечение и Интернет-ресурсы

1. Информационно-аналитический центр по параллельным вычислениям.- http://www.parallel.ru/
2. Свободная энциклопедия Википедия http://ru.wikipedia.org/
3. Центр суперкомпьютерных технологий. Нижегородский государственный университет - http://www.software.unn.ac.ru/ccam/
4. СУПЕРКОМПЬЮТЕРНЫЙ ЦЕНТР коллективного пользования. Институт динамики систем и теории управления СО РАН (г. Иркутск).- http://mvs.icc.ru/index.html
5. Пакет средств для разработки параллельных приложений Intel Parallel Studio XE:

   https://software.intel.com/en-us/intel-parallel-studio-xe/try-buy

Леонид Александрович **Игумнов**
Игорь Валерьевич **Воробцов**
Светлана Юрьевна **Литвинчук**

ПОДГОТОВКА НАУЧНО-ИССЛЕДОВАТЕЛЬСКИХ ПРИЛОЖЕНИЙ К ИСПОЛЬЗОВАНИЮ НА ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ СИСТЕМАХ

Учебное пособие

Формат 60×84 1/16. Бумага офсетная. Печать офсетная.
Гарнитура «Таймс». Уч.-изд. л. . Усл.-печ. л. .
Тираж 100 экз. Заказ №

Нижегородский государственный университет им. Н.И. Лобачевского
603950, Нижний Новгород, пр. Гагарина, 23

Типография Нижегородского госуниверситета им. Н.И. Лобачевского
603000, Нижний Новгород, ул. Б. Покровская, 37