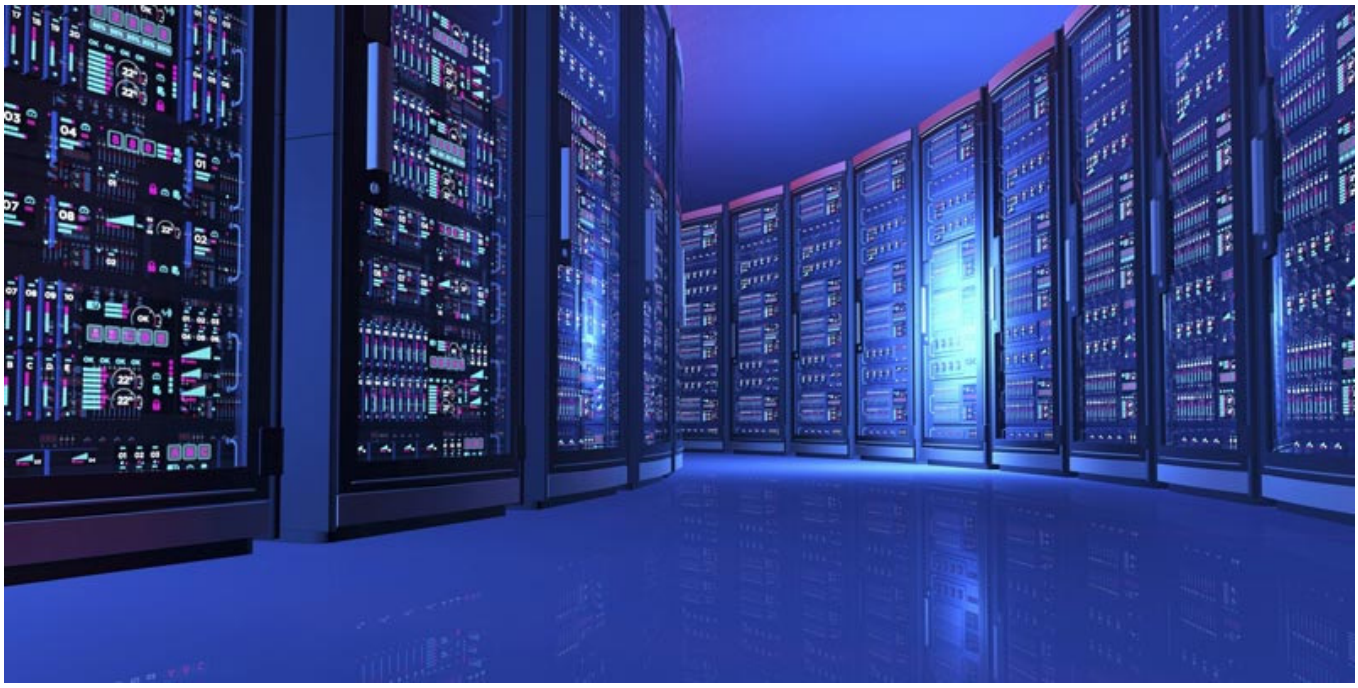
МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Учреждение образования
«Брестский государственный технический университет»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

для выполнения курсового проекта по дисциплине
«Компьютерные системы и сети»
для студентов специальности 1-04 01 01 «Программное обеспечение информационных технологий»

Реализация параллельной обработки для кластерных / мультипроцессорных систем на базе протокола MPI



2022

АННОТАЦИЯ

Методические указания предназначены для выполнения курсового проекта по дисциплине "Компьютерные системы и сети". Содержат описания способов проектирования программных средств параллельной обработки информации на основе библиотеки Message Passing Interface (MPI), являющейся наиболее универсальным современным средством разработки эффективных параллельных приложений для многокомпонентных (многомашинных, многопроцессорных) компьютерных систем и комплексов. Методические указания включают описания наиболее распространенных функций MPI, правила использования библиотеки при программировании в среде C++, общие правила разработки схем параллелизации алгоритмов. Задания и требования к курсовым проектам разработаны с учетом имеющейся на кафедре ИИТ БГТУ аппаратно–технической базы.

Методические указания предназначены для использования студентами специальности 1-04 01 01 «Программное обеспечение информационных технологий»

Составитель – Юрий Викторович Савицкий, к.т.н., доцент

1 ОБЩИЕ ПОЛОЖЕНИЯ

1.1 Принципы высокопроизводительных вычислений

Применение сложных, в том числе интеллектуальных, подходов к обработке данных, представленных мега-, гига- и терабайтами, требует особого внимания к обеспечению их высокопроизводительной обработки. В условиях стремительного наращивания объемов данных многие стандартные вычислительные средства и традиционные алгоритмические подходы для такой сложной и затратной обработки в значительной степени непригодны. Сформулируем несколько основных принципов организации высокопроизводительных вычислений.

Первый принцип организации высокопроизводительных вычислений заключается в увеличении производительности обработки данных за счет совершенствования вычислительной эффективности алгоритмов, позволяя применять достаточно сложную обработку данных, но в некоторых случаях доступную даже для ПЭВМ со стандартными характеристиками.

Сегодня доступны различные вычислительные мощности. Вычислительные кластеры организуют как на базе стандартных недорогих ПЭВМ, объединенных в локальную вычислительную сеть, так и в специализированных центрах, оснащенных суперкомпьютерной многопроцессорной техникой. Поэтому второй принцип определяет целесообразность использования таких подходов к обработке данных, которые применимы как на дорогостоящей суперкомпьютерной технике, так и на недорогих ПЭВМ, объединенных в сети. При этом подразумевается, что высокопроизводительная обработка достигается двумя способами, которые могут быть использованы совместно.

Первый способ предполагает обработку исключительно за счет применения более высокопроизводительной вычислительной техники. Второй – адаптацию существующих подходов для возможности не только канонического параллельного, но и так называемого распределенно-параллельного исполнения. Обширный класс параллельных вычислений характеризуется возможностью одновременного решения одной вычислительной задачи путем ее декомпозиции. Очевидно, параллельные вычисления могут быть реализованы на одной ПЭВМ в многопроцессном режиме под управлением многозадачной операционной системы. Параллельные вычисления на нескольких вычислительных узлах (например, нескольких ПЭВМ, объединенных в локальную вычислительную сеть, или нескольких процессорах суперкомпьютера) терминологически относят к распределенным вычислениям. Именно поэтому здесь и далее, применяя термин распределенно-параллельные вычисления, будем иметь в виду вычислительный процесс, реализуемый не только как параллельный, но и как распределенный.

Практическая организация высокопроизводительных распределенно-параллельных вычислений с использованием указанных выше принципов требует ряда важных пояснений.

Организация распределенных вычислений возможна с использованием множества различных подходов и архитектур. Однако при всем многообразии

возможных вариантов принципиально отличают два различных класса построения вычислительных систем – системы с общей (разделяемой) памятью и системы с распределенной памятью.

К первому варианту построения вычислительных систем относят системы с симметричной архитектурой и симметричной организацией вычислительного процесса – SMP-системы (англ. Symmetric MultiProcessing). Поддержка SMP-обработки данных присутствует в большинстве современных ОС при организации вычислительных процессов на многопроцессных (многоядерных) ПЭВМ. Однако разделяемая память требует решения вопросов синхронизации и исключительного доступа к разделяемым данным, а построение высокопроизводительных систем в этой архитектуре затруднено технологически сложностями объединения большого числа процессоров с единой оперативной памятью.

Второй вариант предполагает объединение нескольких вычислительных узлов (ВУ) с собственной памятью в единой коммуникационной среде, взаимодействие между узлами в которой осуществляется путем пересылки сообщений. Причем такими ВУ могут быть как стандартные недорогие ПЭВМ, так и процессоры дорогостоящего суперкомпьютера. Такая архитектура построения вычислительной системы характеризуется бóльшими возможностями построения высокопроизводительных вычислительных систем и значительного масштабирования доступных вычислительных мощностей. Однако, в отличие от SMP-систем, здесь более высокая латентность (существенные накладные коммуникационные расходы), негативно влияющая на производительность системы и требующая более внимательной организации распределенно-параллельной обработки. Причем в случае построения вычислительного кластера на базе ПЭВМ такая латентность, как правило, существенно выше, чем при построении кластера на базе суперкомпьютера, за счет значительно более развитой коммуникационной среды.

В соответствии с классификацией архитектур вычислительных систем М. Флинна наибольшее распространение на практике получили две модели параллельных вычислений – Multiple Process / Program – Multiple Data (MPMD, множество процессов / программ, множество данных) и Single Process / Program – Multiple Data (SPMD, один процесс / программа, множество данных). Модель MPMD предполагает, что параллельно выполняющиеся процессы исполняют различные программы (процессы, потоки) на различных процессорах. Модель SPMD обусловливает работу параллельно выполняющихся программ (процессов, потоков), но исполняющих идентичный код при обработке отличных (в общем случае) массивов данных.

Очевидно, организацию распределенных вычислений целесообразно реализовать с использованием программ с параллельной обработкой данных на основе модели SPMD. Во-первых, это более практично из-за необходимости разрабатывать и отлаживать лишь одну программу, а во-вторых, такие программы, как правило, применимы и при традиционном последовательном исполнении, что расширяет области их практической применимости.

Кроме вышеизложенных особенностей организации высокопроизводительных вычислений в различных архитектурах и на основе различных моделей, при разработке алгоритмического обеспечения параллельной обработки необходимо:

– определить фрагменты вычислений, которые могут быть исполнены параллельно;

– распределить данные по модулям локальной памяти отдельных ВУ;

– согласовать распределение данных с параллелизмом вычислений.

Важным является выполнение всех перечисленных условий, так как иначе значительные фрагменты вычислений не удастся представить как параллельно исполняемые и реализация алгоритма в распределенно-параллельной архитектуре не позволит добиться роста производительности. Задачи распределения данных по модулям памяти и задача согласования распределения данных важны для обеспечения низкой латентности между ВУ и обеспечения возможностей масштабирования системы.

Помимо этого, при построении такого рода вычислительных системы важно иметь четко измеримые показатели их эффективности. Первым таким критерием можно назвать параллельное ускорение, которое показывает ускорение выполнения по сравнению с последовательным вариантом.

1.2 Две парадигмы программирования

Развитие фундаментальных и прикладных наук, технологий требует применения все более мощных и эффективных методов и средств обработки информации. В качестве примера можно привести разработку реалистических математических моделей, которые часто оказываются настолько сложными, что не допускают точного аналитического их исследования. Единственная возможность исследования таких моделей, их верификации (то есть подтверждения правильности) и использования для прогноза - компьютерное моделирование, применение методов численного анализа. Другая важная проблема - обработка больших объемов информации в режиме реального времени. Все эти проблемы могут быть решены лишь на достаточно мощной аппаратной базе, с применением эффективных методов программирования.

В настоящее время наблюдается стремительный рост развития вычислительной техники. Производительность современных компьютеров на много порядков превосходит производительность первых ЭВМ и продолжает возрастать заметными темпами. Увеличиваются и другие ресурсы, такие как объем и быстродействие оперативной и постоянной памяти, скорость передачи данных между компонентами компьютера и т.д. Совершенствуется архитектура ЭВМ.

Вместе с тем следует заметить, что уже сейчас прогресс в области микроэлектронных компонент сталкивается с ограничениями, связанными с фундаментальными законами природы. Вряд ли можно надеяться на то, что в ближайшее время основной прогресс в производительности электронно-вычислительных машин будет достигнут лишь за счет совершенствования их элементной базы. Переход на качественно новый уровень производительности потребовал от разработчиков ЭВМ и новых архитектурных решений.

1.3 Две модели программирования: последовательная и параллельная

Традиционная архитектура ЭВМ была последовательной. Это означало, что в любой момент времени выполнялась только одна операция и только над одним операндом. Совокупность приемов программирования, структур данных, отвечающих последовательной архитектуре компьютера, называется моделью последовательного программирования. Ее основными чертами являются применение стандартных языков программирования (как правило, это ФОРТРАН-77, ФОРТРАН-90, C/C++), достаточно простая переносимость программ с одного компьютера на другой и невысокая производительность.

Появление в середине шестидесятых первого компьютера класса суперЭВМ, разработанного в фирме CDC Сеймуром Крэем, ознаменовало рождение новой - векторной архитектуры. Начиная с этого момента, суперкомпьютером принято называть высокопроизводительный векторный компьютер. Основная идея, положенная в основу новой архитектуры, заключалась в распараллеливании процесса обработки данных, когда одна и та же операция применяется одновременно к массиву (вектору) значений. В этом случае можно надеяться на определенный выигрыш в скорости вычислений Идея параллелизма оказалась плодотворной и нашла воплощение на разных уровнях функционирования компьютера.

Основными особенностями модели параллельного программирования являются высокая эффективность программ, применение специальных приемов программирования и, как следствие, более высокая трудоемкость программирования, проблемы с переносимостью программ.

1.4 Две парадигмы параллельного программирования

В настоящее время существуют два основных подхода к распараллеливаниювычислений. Это *параллелизм данных и параллелизм задач*. В англоязычной литературе соответствующие термины - *data parallel и message passing*. В основе обоих подходов лежит распределение вычислительной работы по доступным пользователю процессорам параллельного компьютера. При этом приходится решать разнообразные проблемы. Прежде всего, это достаточно равномерная загрузка процессоров, так как если основная вычислительная работа будет ложиться на один из процессоров, мы приходим к случаю обычных последовательных вычислений, и в этом случае никакого выигрыша за счет распараллеливания задачи не будет. Сбалансированная работа процессоров - это первая проблема, которую следует решить при организации параллельных вычислений. Другая и не менее важная проблема - скорость обмена информацией между процессорами. Если вычисления выполняются на высокопроизводительных процессорах, загрузка которых достаточно равномерная, но скорость обмена данными низкая, основная часть времени будет тратиться впустую на ожидание информации, необходимой для дальнейшей работы данного процессора. Рассматриваемые парадигмы программирования различаются методами решения этих двух основных проблем. Разберем более подробно параллелизм данных и параллелизм задач.

(i) *Параллелизм данных*

Основная идея подхода, основанного на параллелизме данных, заключается в том, что одна операция выполняется сразу над всеми элементами массива данных. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Распределением данных между процессорами занимается программа. Векторизация или распараллеливание в этом случае чаще всего выполняется уже на этапе компиляции - перевода исходного текста программы в машинные команды. Роль программиста в этом случае обычно сводится к заданию опций векторной или параллельной оптимизации компилятору, директив параллельной компиляции, использованию специализированных языков для параллельных вычислений. Наиболее распространенными языками для параллельных вычислений являются Высокопроизводительный ФОРТРАН (High Performance FORTRAN) и параллельные версии языка С (это, например, С*). Более детальное описание рассматриваемого подхода к распараллеливанию содержит указание на следующие его основные особенности:

- обработкой данных управляет одна программа;
- пространство имен является глобальным, то есть для программиста существует одна единственная память, а детали структуры данных, доступа к памяти и межпроцессорного обмена данными от него скрыты;
- слабая синхронизация вычислений на параллельных процессорах, то есть выполнение команд на разных процессорах происходит, как правило, независимо и только лишь иногда производится согласование выполнения циклов или других программных конструкций - их синхронизация. Каждый процессор выполняет один и тот же фрагмент программы, но нет гарантии, что в заданный момент времени на всех процессорах выполняется одна и та же машинная команда;
- параллельные операции над элементами массива выполняются одновременно на всех доступных данной программе процессорах.

Видим, таким образом, что в рамках данного подхода от программиста не требуется больших усилий по векторизации или распараллеливанию вычислений. Даже при программировании сложных вычислительных алгоритмов можно использовать библиотеки подпрограмм, специально разработанных с учетом конкретной архитектуры компьютера и оптимизированных для этой архитектуры. Подход, основанный на параллелизме данных, базируется на использовании при разработке программ базового набора операций:

- *операции управления данными;*
- *операции над массивами в целом и их фрагментами;*
- *условные операции;*
- *операции приведения;*
- *операции сдвига;*
- *операции сканирования;*
- *операции, связанные с пересылкой данных.*

Рассмотрим эти базовые наборы операций.

Управление данными

В определенных ситуациях возникает необходимость в управлении распределением данных между процессорами. Это может потребоваться, например, для обеспечения равномерной загрузки процессоров. Чем более равномерно загружены работой процессоры, тем более эффективной будет работа компьютера.

Операции над массивами

Аргументами таких операций являются массивы в целом или их фрагменты (сечения), при этом данная операция применяется одновременно (параллельно) ко всем элементам массива. Примерами операций такого типа являются вычисление поэлементной суммы массивов, умножение элементов массива на скалярный или векторный множитель и т.д. Операции могут быть и более сложными - вычисление функций от массива, например. Условные операции

Эти операции могут выполняться лишь над теми элементами массива, которые удовлетворяют какому-то определенному условию. В сеточных методах это может быть четный или нечетный номер строки (столбца) сетки или неравенствонулю элементов матрицы.

Операции приведения

Операции приведения применяются ко всем элементам массива (или его сечения), а результатом является одно единственное значение, например, сумма элементов массива или максимальное значение его элементов.

Операции сдвига

Для эффективной реализации некоторых параллельных алгоритмов требуются операции сдвига массивов. Примерами служат алгоритмы обработки изображений, конечно-разностные алгоритмы и некоторые другие.

Операции сканирования

Операции сканирования еще называются префиксными/суффиксными операциями. Префиксная операция, например, суммирование выполняется следующим образом. Элементы массива суммируются последовательно, а результат очередного суммирования заносится в очередную ячейку нового, результирующего массива, причем номер этой ячейки совпадает с числом просуммированных элементов исходного массива.

Операции пересылки данных

Это, например, операции пересылки данных между массивами разной формы (то есть имеющими разную размерность и разную протяженность по каждому измерению) и некоторые другие.

При программировании на основе параллелизма данных часто используются специализированные языки - CM FORTRAN, C*, FORTRAN+, MPP FORTRAN, Vienna FORTRAN, а также HIGH PERFORMANCE FORTRAN (HPF).

(i) *Параллелизм задач*

Стиль программирования, основанный на параллелизме задач подразумевает, что вычислительная задача разбивается на несколько относительносамостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Компьютер при этом представляет собой MIMD - машину. Как было отмечено ранее, аббревиатура MIMD обозначает в известной

классификации архитектур ЭВМ компьютер, выполняющий одновременно множество различных операций над множеством, вообще говоря, различных и разнотипных данных. Для каждой подзадачи пишется своя собственная программа на обычном языке программирования, обычно это ФОРТРАН или С. Чем больше подзадач, тем большее число процессоров можно использовать, тем большей эффективности можно добиться. Важно то, что все эти программы должны обмениваться результатами своей работы, практически такой обмен осуществляется вызовом процедур специализированной библиотеки. Программист при этом может контролировать распределение данных между процессорами и подзадачами и обмен данными. Очевидно, что в этом случае требуется определенная работа для того, чтобы обеспечить эффективное совместное выполнение различных программ. По сравнению с подходом, основанным на параллелизме данных, данный подход более трудоемкий, с ним связаны следующие проблемы:

- повышенная трудоемкость разработки программы и ее отладки;
- на программиста ложится вся ответственность за равномерную загрузку процессоров параллельного компьютера;
- программисту приходится минимизировать обмен данными между задачами, так как пересылка данных - наиболее "времяёмкий" процесс;
- повышенная опасность возникновения тупиковых ситуаций, когда отправленная одной программой посылка с данными не приходит к месту назначения.

Привлекательными особенностями данного подхода являются большая гибкость и большая свобода, предоставляемая программисту в разработке программы, эффективно использующей ресурсы параллельного компьютера и, как следствие, возможность достижения максимального быстродействия.

Примерами специализированных библиотек для организации параллельной обработки, являются:

- OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, C++ и Фортран. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью. Разработка спецификаций стандарта ведётся некоммерческой организацией OpenMP Architecture Review Board (ARB), в которую входят все основные производители процессоров, а также ряд суперкомпьютерных лабораторий и университетов. OpenMP реализует параллельные вычисления с помощью многопоточности, в которой ведущий (англ. master) поток создаёт набор ведомых потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров/ядер не обязательно должно быть больше или равно количеству потоков);
- PVM (Parallel Virtual Machine, дословно виртуальная параллельная машина) — общедоступный программный пакет, позволяющий объединять разнородный набор компьютеров в общий вычислительный ресурс («виртуальную параллельную машину») и предоставляющий возможности управления процессами с помощью механизма передачи сообщений. PVM — разработка Окриджской Национальной Лаборатории, университетов штата

Теннеси и Эмори (Атланта). Существуют реализации PVM для самых различных платформ.

- MPI (Message Passing Interface, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Библиотека MPI разработана в Аргоннской Национальной Лаборатории (США). MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу.

2 MPI - MESSAGE PASSING INTERFACE

MPI – это большая библиотека функций для передачи сообщений, которая позволяет пользователю переделать свою последовательную программу в программу, которая будет полностью использовать параллельную архитектуру используемой вычислительной системы. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные/многопроцессорные с общей памятью / раздельной памятью), взаимного расположения ветвей (на одном процессоре / на разных процессорах).

Программа, использующая MPI, легче отлаживается и быстрее переносится на другие платформы. Минимально в состав MPI входят библиотека программирования (заголовочные и библиотечные файлы для языков C, C++ и Фортран) и загрузчик приложений.

Для MPI необходимо создавать приложения, содержащие код всех вервей сразу. MPI-загрузчиком запускается указываемое количество экземпляров программы. Каждый экземпляр определяет свой порядковый номер в запущенном коллективе, и в зависимости от этого номера и размера коллектива выполняет ту или иную ветвь алгоритма. Такая модель параллелизма называется Single Program/Multiple Data (SPMD) и является частным случаем модели Multiple Instruction/Multiple Data (MIMD). Каждая ветвь имеет пространство данных, полностью изолированное от других ветвей. Обмениваются данными ветви только в виде сообщений MPI. Все ветви запускаются загрузчиком одновременно. Количество ветвей фиксировано – это означает, что в ходе работы порождение новых вервей невозможно.

Разные MPI-процессы могут выполняться как на разных процессорах, так и на одном и том же – для MPI-программы это роли не играет, поскольку в обоих случаях механизм обмена данными одинаков. Процессы обмениваются друг с другом данными в виде сообщений. Сообщения проходят под идентификаторами, которые позволяют программе и библиотеке связи отличать их друг от друга. Для совместного проведения тех или иных расчетов процессы внутри приложения объединяются в группы. Каждый процесс может узнать у библиотеки связи свой номер внутри группы, и, в зависимости от номера, приступает к выполнению соответствующей части расчетов. MPI-ветвь запускается и работает как обычный процесс, связанный через MPI с остальными процессами, входящими в приложение. В остальном процессы следует считать изолированными друг от друга – у них разные области кода, стека и данных. Особенностью MPI является введение понятия *области связи (communication domains).* При запуске приложения все процессы помещаются в создаваемую для приложения общую область связи. При необходимости они могут создавать новые области связи на базе существующих. Все области связи имеют независимую друг от друга нумерацию процессов. Программе пользователя в распоряжение предоставляется *коммуникатор* – описатель области связи. Многие функции MPI имеют среди входных аргументов коммуникатор, который ограничивает сферу их действия той областью связи, к которой он прикреплен. Для одной области связи может существовать несколько коммуникаторов таким образом, что приложение будет работать с ней как с несколькими разными областями. Так, в исходных текстах примеров для MPI

часто используется идентификатор **MPI_COMM_WORLD**. Это название коммуникатора, создаваемого библиотекой автоматически. Он описывает стартовую область связи, объединяющую все процессы приложения. Далее в материале рассмотрены наиболее употребительные функции MPI.

2.1 Наиболее общие функции MPI и правила их использования

Существует несколько категорий функций:
- блокирующие;
- локальные;
- коллективные.

Блокирующие – останавливают (блокируют) выполнение процесса до тех пор, пока производимая ими операция не будет выполнена. *Неблокирующие* функции возвращают управление немедленно, а выполнение операции продолжается в фоновом режиме; за завершением операции надо проследить особо. Неблокирующие функции возвращают *квитанции (requests),* которые погашаются при завершении. До погашения квитанции с переменными и массивами, которые были аргументами не блокирующей функции, ничего делать нельзя. *Локальные* функции не инициируют пересылок данных между ветвями. Большинство информационных функций является локальными, т.к. копии системных данных уже хранятся в каждой ветви. Например, рассматриваемые ниже функция передачи **MPI_Send** и функция синхронизации **MPI_Barrier** не являются локальными, поскольку производят пересылку. Следует заметить, что, к примеру, функция приема **MPI_Recv** (парная для **MPI_Send**) является локальной: она всего лишь пассивно ждет поступления данных, ничего не пытаясь сообщить другим ветвям. *Коллективные* – должны быть вызваны всеми ветвями–абонентами того коммуникатора, который передается им в качестве аргумента. Несоблюдение для них этого правила приводит к ошибкам на стадии выполнения программы (как правило, к повисанию) [5].

*Правила использования идентификаторов **MPI***

Все идентификаторы начинаются с префикса "**MPI_**". Это правило без исключений. Не рекомендуется заводить пользовательские идентификаторы, начинающиеся с этой приставки, а также с приставок "**MPID_**", "**MPIR_**" и "**PMPI_**", которые используются в служебных целях. Если идентификатор сконструирован из нескольких слов, слова в нем разделяются подчерками: **MPI_Get_count**, **MPI_Comm_rank**. Иногда, однако, разделитель не используется: **MPI_Sendrecv, MPI_Alltoall**. Порядок слов в составном идентификаторе выбирается по принципу "от общего к частному": сначала префикс "**MPI_**", потом название категории (**Type, Comm, Group, Attr, Errhandler** и т.д.), потом название операции (**MPI_Errhandler_create, MPI_Errhandler_set**,...). Наиболее часто употребляемые функции выпадают из этой схемы: они имеют "анти–методические", но короткие и стереотипные названия, например **MPI_Barrier**, или **MPI_Unpack**. Имена констант (и неизменяемых пользователем переменных) записываются полностью заглавными буквами: **MPI_COMM_WORLD, MPI_FLOAT**. В именах функций первая за префиксом буква – заглавная, остальные маленькие: **MPI_Send, MPI_Comm_size**.

Существует набор функций, которые используются в любом, даже самом коротком приложении MPI. Занимаются они не столько собственно передачей данных, сколько ее обеспечением. Ниже рассматривается ряд наиболее употребимых функций этого класса.

*Подпрограммы управления средой **MPI***

Первое, что параллельная программа должна делать в течение времени выполнения, состоит в том, чтобы создать параллельную среду. Это означает, что она резервирует и устанавливает N узлов, на которых программа должна выполниться, и инициализирует MPI среду. Количество узлов (процессов) N входит в программу как системная переменная. Ниже в подразделе приведены наиболее важные из них, используемые практически в любом приложении MPI.

MPI_Init(&argc, &argv)

Данная функция обеспечивает инициализацию библиотеки. Является одной из первых инструкций в функции **main** (главной функции приложения). Она получает адреса аргументов, стандартно получаемых самой **main** от операционной системы и хранящих параметры командной строки.

MPI_Finalize()

Нормальное закрытие библиотеки. Никакая другая MPI функция не может быть вызвана после этого. Настоятельно рекомендуется не забывать вписывать эту инструкцию перед возвращением из программы, т.е.:

- перед вызовом стандартной функции языка C - **exit** ;
- перед каждым после MPI_Init оператором **return** в функции **main**;
- если функции **main** назначен тип **void**, и она не заканчивается оператором **return,** то **MPI_Finalize()** следует поставить в конец **main**.

Пример 1 демонстрирует правила применения описанных функций при организации MPI-приложения.

Пример 1

```
#include "mpi.h"
#include <stdio.h>
/* mpi.h обеспечивает основные определения и типы MPI */
int main(argc, argv)
int argc;
char **argv;
{
 MPI_Init(&argc, &argv);
 printf("Hello world\n");
 MPI_Finalize();
 return 0;
}
```

MPI_Abort (comm, error)

Вызов **MPI_Abort** из любой задачи принудительно завершает работу ВСЕХ задач, подсоединенных к заданной области связи. Если указан описатель области связи **comm** в виде **MPI_COMM_WORLD**, будет завершено все приложение (все его задачи) целиком, что, по-видимому, и является наиболее правильным

решением. Если неизвестно, как охарактеризовать ошибку **error** в классификации MPI, рекомендуется использовать код ошибки **MPI_ERR_OTHER**.

Следующие две информационных функции сообщают размер группы (т.е. общее количество задач, подсоединенных к ее области связи) и порядковый номер вызывающей задачи.

MPI_Comm_size (comm, *size)

Число процессов возвращает функция **MPI_Comm_size**. **size** – это число всех процессов в пределах некоторой группы названной **comm**. Обычно **comm** является предопределенной коммуникатором **MPI_COMM_WORLD,** кото рый включает все процессы в прикладную программу MPI.

MPI_Comm_rank (comm, *rank)

Каждый процесс характеризуется собственным целочисленным идентификатором называемым рангом (**rank**). Ранги непрерывны и начинаются с нуля и заканчиваются **size–1**. Процесс с нулевым рангом обычно называется ГЛАВНЫМ или **MASTER** процессом. Функция **MPI_Comm_rank** возвращает ранг вызываемого процесса. Пример 2 демонстрирует использование **MPI_Comm_size** и **MPI_Comm_rank**.

Пример 2

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

int main(argc, argv)
int argc;
char **argv;
{
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Hello world! I'm %d of %d\n",rank, size);
MPI_Finalize();
return 0;
}
```

MPI_Wtime()

Функция возвращает затраченное время (в секундах с двойной точностью) на вызов процесса. *Эта функция полезна, чтобы проверить время выполнения программы.* Пример 3 показывает, как использовать приведенные функции.

Пример 3

```
#include "mpi.h"
#include  <stdio.h>
void main(int argc, char *argv [])
{
  int numtasks, rank, rc;
```

```
  rc = MPI_Init(&argc, &argv);
  if (rc!= 0) {
  printf (" Ошибка,при запуске MPI программы ");
  printf(".Завершение\n ");
  MPI_Finalize();
   }
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf(" Число задач =%d Мой ранг =%d\n",numtasks,rank);
  printf(" Время начала: % lf \n ", MPI_Wtime());

  /******* делает некоторую работу *******/

  printf(" Время конца: %lf \n ", MPI_Wtime());
  MPI_Finalize();
}
```

В языке программирования С все MPI-подпрограммы после завершения работы возвращают целое число. Если возвращаемое значение ноль – успешное завершение работы, в противном случае – ненулевое значение. Эта информация может использоваться для обработки ошибок как в примере 3, рассмотренном выше.

*Процедуры **MPI** передачи сообщений*

Связь между задачами возможна через многие MPI подпрограммы. Здесь внимание сосредоточено на трех подпрограммах, которые являются наиболее полезными для разработки параллельных приложений.

Функция **MPI_Send** рассылает данные из буфера **buf**, количество данных **count**, тип – **datatype**. Ниже приведен синтаксис этой функции.

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

Параметры

- **buf** – адрес буфера с информацией;
- **count** – количество элементов в буфере;
- **datatype** – тип элемента в буфере;
- **dest** – ранг процесса, которому посылаются данные;
- **tag** – таг сообщения;
- **comm** – дескриптор.

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

Функция **MPI_Recv** получает данные.

Параметры

- **buf** – адрес буфера с информацией;
- **count** – максимальное количество элементов в буфере;
- **datatype** – тип элемента в буфере;
- **tag** – таг сообщения;

- **comm** – дескриптор;
- **status** – объект статуса;
- **source** – ранг процесса, который посылает сообщение.

Параметр **dest** (**source**) определяет процесс, которому должно быть доставлено (из которого отправлено) сообщение и определяется как ранг процесса, который получает (отправляет) сообщение. Каждое сообщение помечается своей уникальной целочисленной отметкой (**tag**). Параметр **tag** опознает сообщение и должен соответствовать одной из получаемых задач. Для получателя символ **MPI_ANY_TAG** может использоваться, чтобы получить любое сообщение независимо от его отметки (**tag**). Наконец, операция **status** указывает на источник и отметку полученного сообщения. В Си этот параметр обозначает указатель на структуру **MPI_STATUS** (**stat. MPI_SOURCE, stat. MPI_TAG**).

Следует заметить, что параметр **count** содержит максимальное количество элементов в буфере. Его значение можно определить с помощью **MPI_Get_count**.

Все типы в MPI предопределены. Наиболее употребимые из них приведены в табл. 1.

Таблица 1.

MPI datatype	*C datatype*
MPI CHAR	signed char
MPI SHORT	signed short int
MPI INT	signed int
MPI LONG	signed long int
MPI UNSIGNED CHAR	unsigned char
MPI UNSIGNED SHORT	unsigned short int
MPI UNSIGNED	unsigned int
MPI UNSIGNED LONG	unsigned long int
MPI FLOAT	float
MPI DOUBLE	double
MPI LONG DOUBLE	long double
MPI PACKED	

Последний тип MPI_PACKED не соответствуют стандартному типу языка С. Правила его использования описаны далее.

Замечание. *Все MPI функции (за исключением **MPI_Wtime** и **MPI_Wtick**) возвращают информацию об ошибках. В функциях системы программирования С – это возвращаемое значение функции. Перед тем, как функция завершит свою работу, вызывается текущий обработчик ошибок. По умолчанию он прекращает работу этой функции. Значение обработчика ошибок может быть изменено в **MPI_Errhandler_set**. Значения, которые может приобретатьобработчик ошибок, сведены в табл. 2.*

Таблица 2.

MPI_SUCCESS	Нет ошибок. MPI функция завершилась успешно
MPI_ERR_COMM	Недопустимый коммуникатор. В вызове необходимо использовать нулевой коммуникатор.
MPI_ERR_COUNT	Недопустимый параметр **count**. Параметр с**ount** должен быть положительным или нулевым.
MPI_ERR_TYPE	Неправильный тип параметра.
MPI_ERR_TAG	Неправильный параметр тег. Теги должны бать положительными. В функциях **MPI_Recv**, **MPI_Irecv**, **MPI_Sendrecv**, и т.п. они могут принимать значение **MPI_ANY_TAG**. Максимальное значение тегаможно получить через **MPI_TAG_UB**.
MPI_ERR_RANK	Недопустимое значение ранга. Ранг должен находиться в границах от 0до **size–1** (size – размер коммуникатора). В функциях **MPI_Recv, MPI_Irecv**, **MPI_Sendrecv,** и т.п. он может принимать значение **MPI_ANY_SOURCE.**

Пример 4 показывает использование **MPI_Send** и **MPI_Recv**. Приведенная программа берет данные нулевого процесса и рассылает их всем остальным процессам по кругу. Это означает, что процесс ***i*** должен получить данные от процесса ***i–1*** и переслать их процессу ***i+1*.**

Пример 4

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

int main(argc, argv)
int argc;
char **argv;
{
    int rank, value, size;
    MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    do {
        if (rank == 0) {
          scanf("%d", &value);
          MPI_Send(&value, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
        }
        else {
          MPI_Recv(&value, 1, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD, &status);
          if (rank < size - 1)
             MPI_Send(&value, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
        }
        printf("Process %d got %d\n", rank, value);
    } while (value >= 0);
MPI_Finalize();
    return 0;
}
```

Рассмотрим еще одну программу с использованием этих функций (пример 5). Здесь проверяется, правильно ли рассылаются и получаются сообщения. В этой программе все процессы (кроме нулевого) посылают 3 сообщения нулевому процессу. Нулевой процесс печатает это сообщение как только он его получает (используем **MPI_ANY_SOURCE** и **MPI_ANY_TAG** в **MPI_Recv**).

Пример 5

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

int main(argc, argv)
int argc;
```

```
char **argv;
{
    int rank, size, i, buf[1];
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (rank == 0) {
        for (i=0; i<100*(size-1); i++) {
            MPI_Recv( buf, 1, MPI_INT, MPI_ANY_SOURCE,
                      MPI_ANY_TAG, MPI_COMM_WORLD, &status );
            printf( "Msg from %d with tag %d\n",
                    status.MPI_SOURCE, status.MPI_TAG );
        }
    }
    else {
        for (i=0; i<100; i++)
      MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
    }
    MPI_Finalize();
    return 0;
}
```

MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Функция **MPI_Bcast** пересылает информацию с процесса **root** ко всем остальным.

Параметры

- **buf** – адрес буфера с информацией;
- **count** – количество элементов в буфере;
- **datatype** – тип элемента в буфере;
- **root** – ранг **root** процесса;
- **comm** – дескриптор.

Здесь **root** – это ранг процесса, который рассылает данные. Часто требуется, чтобы один из процессов читал данные с клавиатуры или командной строки, а потом рассылал эту информацию всем остальным процессам.

Рассмотрим программу, которая читает целое число, вводимое пользователем, и рассылает это его остальным процессам (пример 6). Каждый процесс должен напечатать свой ранг и значение, которое он получил. Значения вводятся, пока не будет введено отрицательное число.

Пример 6

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

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value;
    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    do {
        if (rank == 0)
            scanf( "%d", &value );

        MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );

        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);

    MPI_Finalize( );
    return 0;
}
```

MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

С командой **MPI_Reduce** узел **root** собирает данные от всех узлов, включая себя и применяет к ним операцию **op.**

Параметры

- **sendbuf** – адрес буфера с информацией;
- **count** – количество элементов в буфере;
- **datatype** – тип элемента в буфере;
- **op** – операция;
- **root** – ранг главного процесса;
- **recvbuf** – адрес буфера, который будет принимать данные;
- **comm** – дескриптор.

sendbuf содержит данные, которые должны быть посланы, а **recvbuf** получает их. Операцией **op** (operation) выполняется заданная операция над данными. Наиболее общие операции предопределены как:

- **MPI_MAX** – максимум;
- **MPI_MIN** – минимум;
- **MPI_SUM** – суммирование;
- **MPI_PROD** – программа.

Обычно процесс сбора – **MASTER** процесс, т.е процесс с нулевым рангом. Рассмотрим пример 7, в котором вычисляется значение π. Подсчитывается интеграл ***4/(1+x*x)*** в промежутке от ***–1/2*** и ***1/2*****.** Алгоритм простой: интеграл аппроксимируется суммой ***n*** интервалов. Аппроксимация интеграла на каждом интервале ***– (1/n)*4/(1+x*x)*.** Главный процесс (ранг 0) делает запрос на введения количества интервалов. Далее он рассылает это число всем процессам. Каждый процесс складывает *n* интервалов ***(x=–1/2+rank/n, –1/2+rank/n+size/n,...)***. В конце, суммы, вычисляемые каждым процессом, складываются вместе.

Пример 7

```
#include "mpi.h"
#include <stdio.h>
int main(argc,argv)
int argc;
char *argv[];
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done)
    {
        if (myid == 0) {
            printf("Enter the number of intervals:(0 quits)
");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;

        h   = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }
        mypi = h * sum;

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                   MPI_COMM_WORLD);

        if (myid == 0)
            printf("pi is approximately %.16f, Error is
%.16f\n",
                   pi, fabs(pi - PI25DT));
    }
```

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

MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int dest, MPI_Comm comm)

Функция *MPI_Gather* собирает в приемный буфер задачи dest передающие буфера остальных задач

- **sbuf** - адрес начала буфера посылки;
- **scount** - число элементов в посылаемом сообщении;
- **stype** - тип элементов отсылаемого сообщения;
- **OUT rbuf** - адрес начала буфера сборки данных;
- **rcount** - число элементов в принимаемом сообщении;
- **rtype** - тип элементов принимаемого сообщения;
- **dest** - номер процесса, на котором происходит сборка данных;
- **comm** - идентификатор группы.

MPI_Barrier(MPI_Comm comm)

Является функцией синхронизации – останавливает выполнение вызвавшей ее задачи до тех пор, пока не будет вызвана изо всех остальных задач, подсоединенных к указываемому коммуникатору. Гарантирует, что к выполнению следующей за **MPI_Barrier** инструкции каждая задача приступит одновременно с остальными. Данная функция является единственной, вызовами которой гарантированно синхронизируется во времени выполнение различных ветвей. Некоторые другие коллективные функции в зависимости от реализации могут обладать, а могут и не обладать свойством одновременно возвращать управление всем ветвям. Однако для них это свойство является побочным и необязательным. Поэтому, если в программе нужна синхронность, необходимо использовать только **MPI_Barrier**.

Приведенные ниже 4 функции рассмотрены более детально в примере 8.

MPI_Address(void *location, MPI_Aint *address)

Получает адрес параметра в памяти (**address**) . Во многих системах адрес, возвращаемый этой функцией аналогичен оператору & в Си.

MPI_Type_Struct

Создает структурный тип данных

MPI_Type_Commit (MPI_datatype *datatype)

Передает тип данных.

MPI_Datatype_free (MPI_Datatype *datatype)

Освобождает тип данных.

Рассмотрим в примере 8 использование 4 последних функций. В программе примера процессу 0 передаются целое и действительное числа. Из них создается структура данных, которая функцией **MPI_Bcast** рассылается всем остальным процессам. Ввод завершается при введении отрицательного целого.

Пример 8

```
#include <stdio.h>
#include "mpi.h"
int main(argc, argv)
int argc;
char **argv;
{
    int          rank;
    struct { int a; double b } value;
    MPI_Datatype mystruct;
    int          blocklens[2];
    MPI_Aint     indices[2];
    MPI_Datatype old_types[2];

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* По одному значению каждого типа */
    blocklens[0] = 1;
    blocklens[1] = 1;
    /* Типы параметров */
    old_types[0] = MPI_INT;
    old_types[1] = MPI_DOUBLE;
    /* Расположение каждого элемента */
    MPI_Address( &value.a, &indices[0] );
    MPI_Address( &value.b, &indices[1] );
    /* Делает зависимыми */
    indices[1] = indices[1] - indices[0];
    indices[0] = 0;
    MPI_Type_struct( 2, blocklens, indices, old_types,
&mystruct );
    MPI_Type_commit( &mystruct );

    do {
        if (rank == 0)
            scanf( "%d %lf", &value.a, &value.b );

        MPI_Bcast( &value, 1, mystruct, 0, MPI_COMM_WORLD );

        printf( "Process %d got %d and %lf\n", rank, value.a,
value.b );
    } while (value.a >= 0);
```

```
    /* Удаление типа*/
    MPI_Type_free( &mystruct );
    MPI_Finalize( );
    return 0;
}
```

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

Упаковывает тип данных в непрерывную область памяти.

Параметры
входные:
- **inbuf** – начало буфера;
- **incount** – количество вводимых элементов;
- **datatype** – тип вводимых элементов;
- **outcount** – выходной размер буфера (в байтах);
- **position** – текущая позиция в буфере (в байтах);
- **comm** – коммуникатор для упакованных сообщений;

выходные:
- **outbuf** – конец буфера.

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

Распаковывает тип данных.

Параметры
входные:
- **inbuf** – начало буфера;
- **insize** – размер буфера (в байтах);
- **position** – текущая позиция в буфере (в байтах);
- **outcount** – количество элементов, что должны быть распакованы;
- **datatype** – тип выходных элементов;
- **comm** – коммуникатор для упакованых данных;

выходные:
- **outbuf** – конец буфера.

Рассмотрим предыдущую программу с использованием двух последних функций в примере 9. Используем **MPI_Pack** для упаковки данных в буфер (для простоты используем packbuf[100]). Следует заметить, что **MPI_Bcast**, в отличие от **MPI_Send/MPI_Recv**, требует, чтобы одинаковое количество данных было послано и получено. Таким образом, необходимо удостовериться, что все процессы имеют одно и тоже значение параметра **count** в **MPI_Bcast**.

Пример 9.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{
    int           rank;
    int           packsize, position;
    int           a;
    double        b;
    char          packbuf[100];

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    do {
        if (rank == 0) {
            scanf( "%d %lf", &a, &b );
            packsize = 0;
            MPI_Pack( &a, 1, MPI_INT, packbuf, 100, &packsize,
                      MPI_COMM_WORLD );
            MPI_Pack( &b, 1, MPI_DOUBLE, packbuf, 100,
&packsize,
                      MPI_COMM_WORLD );
        }
        MPI_Bcast( &packsize, 1, MPI_INT, 0, MPI_COMM_WORLD );
        MPI_Bcast( packbuf, packsize, MPI_PACKED, 0,
MPI_COMM_WORLD );
        if (rank != 0) {
            position = 0;
             MPI_Unpack( packbuf, packsize, &position, &a, 1,
MPI_INT,
                         MPI_COMM_WORLD );
            MPI_Unpack( packbuf, packsize, &position, &b, 1,
MPI_DOUBLE,
                        MPI_COMM_WORLD );
        }

        printf( "Process %d got %d and %lf\n", rank, a, b );
    } while (a >= 0);

    MPI_Finalize( );
    return 0;
}
```

3. ОСНОВНЫЕ ЭТАПЫ КУРСОВОГО ПРОЕКТИРОВАНИЯ

Название курсового проекта

Проектирование системы параллельной обработки данных с использованием MPI.

Цель курсового проектирования

Приобретение навыков по разработке и реализации параллельных алгоритмов для многокомпонентных систем обработки данных.

Разработка параллельных приложений использованием интерфейса MPI.

Сравнительный анализ временных характеристик выполнения обработки данных.

Конфигурирование MPI-окружения и запуск приложения (на примере библиотеки WMPI for WINDOWS)

Для организации работы параллельного MPI-приложения необходимо сконфигурировать файл MPI-окружения **<**имя_файла_приложения**>.pg** и разместить его в каталоге приложения. При запуске программы происходит обращение к файлу с расширением **pg**, который должен иметь такое же имя, как и имя файла, содержащего код программы. В нем должно быть указано количество процессов, которые необходимо создать и адреса рабочих станций, на которых процессы должны быть созданы. Например, файл **test.pg** содержит следующую информацию:

```
local 3
c310_1 2 g:\bkss\test.exe
```

Слово **local** означает, что указанные после него процессы будут созданы на той рабочей станции, на которой была запущена программа. При создании процессов на других рабочих станциях указывается адрес рабочей станции, количество создаваемых процессов, путь к каталогу, в котором располагается файл с кодом программы (вторая строка, приведенного в примере файла);

Исходные данные

Имеется многокомпонентная вычислительная система, состоящая из вычислительных модулей (ВМ), каждый из которых имеет память для хранения обрабатываемых данныхи результатов (в рамках курсового проектирования – это либо кластер в виде набора рабочих станций в составе локальной вычислительной сети, либо мультипроцессорная / мультиядерная архитектура). Известно, *что время выполнения одной арифметической операции намного меньше, чем время передачи одного числа от ВМ к ВМ*. Известно также, что выгоднее по времени передавать большие блоки данных, причем, чем больше пакет, тем лучше.

Дополнительно задаются следующие исходные данные:

- описание задачи обработки данных (согласно варианта задания);
- описание размерности решаемой задачи (согласно варианта задания);
- количество процессов, необходимых для реализации параллельного алгоритма (согласно варианта задания);

- язык программирования - C ++,
- MPI-библиотека.

Замечание**:** *для выполнения раздела 5 курсового проекта необходимо реализовать в программах средства для проведения исследований по оцениванию времени выполнения алгоритма с учетом и без учета межпроцессных пересылок MPI.*

Порядок выполнения курсового проекта

- на основе анализа варианта задания разработать последовательный алгоритм обработки;
- разработать программную систему, реализующую последовательный алгоритм, используя язык программирования C ++;
- изучить основы программирования задач с использованием интерфейса MPI;
- выполнить анализ последовательного алгоритма с целью выделения независимых по данным и управлению фрагментов, обработку которых возможно организовывать в параллельном режиме. Составить варианты схем параллелизации, обосновать выбор одного из них для последующей реализации на основе MPI. *При выборе варианта крайне необходимо учитывать тот факт, что каждая межмодульная пересылка сопровождается значительными временными издержками, связанными с организацией канала пересылки. Следовательно, целесообразно минимизировать количество межмодульных пересылок и увеличивать объем пересылаемых данных для каждой пересылки.* Для выбранного варианта составить временную диаграмму этапов параллельной обработки и этапов межмодульного обмена;
- разработать алгоритм, реализующий выбранную схему параллелизации; на основе последовательной программы разработать программную систему, реализующую параллельный алгоритм, используя язык программирования C++ и MPI. При этом, с целью анализа производительности программы, обеспечить систему средствами, позволяющими оценить реальное время обработки;
- выполнить тестирование и рассчитать среднюю производительность разработанных программ с последовательной ($\lambda_1=1/T_1$) и параллельной ($\lambda_p=1/T_p$) обработкой на исходных данных заданной размерности (согласно варианту задания), T_1, T_p - фактическое среднее время последовательной и параллельной программы. Для этого рассчитать средние значения времен выполнения процессов по результатам нескольких (не менее 10) экспериментов; при этом зафиксировать как наблюдаемые, так и средние значения в сводной таблице;
- на основе результатов предыдущего этапа рассчитать коэффициент

1

увеличения производительности (ускорения) за счет параллелизации:

$$m=\lambda_p/\lambda_1= E_1/T_p$$

Сделать выводы относительно эффективности разработанной схемы параллелизации.

Требования к содержанию разделов пояснительной записки

Во введении рассматриваются и анализируются (в краткой форме) цель и задачи курсового проекта, особенности прикладной задачи, требующей решения в процессе выполнения курсового проекта.

В первом разделе выполняется детальный анализ алгоритмов решения задачи, приводятся форматы исходных данных; анализируются особенности реализации алгоритмов обработки на языке программирования с учетом размерности исходных данных, размеров выделяемых буферов и других факторов, обуславливающих особенности реализуемых алгоритмов. Приводится схема алгоритма и ее детальноеописание.

Во втором разделе приводятся особенности реализации алгоритма на языке C++, описание ключевых процедур решения задачи.

Во третьем разделе на основе анализа последовательного алгоритма должны быть предложены и детально описаны варианты схем параллелизации алгоритма с учетом имеющегося ассортимента функций MPI; выполнен анализ вариантов с точки зрения заданного количества процессов, объема передаваемых данных между процессами, трудоемкостью параллелизуемых ветвей; обоснованно выбран в качестве базового вариант схемы. Приводится выбранная схема алгоритма параллельнойобработки.

В четвертом разделе приводится детальное описание структуры системы, программных модулей системы параллельной обработки с использованием MPI, описание и особенности использования конкретных функций MPI с приведением точного размера и типа передаваемых данных между процессами; допускается вставлять в текст раздела фрагменты программы с комментариями. Разрабатывается и приводится схема ресурсов системы параллельной обработки.

В пятом разделе детально описываются численные эксперименты по анализу временных характеристик выполнения приложения; должны быть приведены условия проведения исследований, формулы и результаты расчетов с комментариями. Численные данные свести в таблицу. Приводятся скриншоты результатов функционирования программ, в том числе численных.

В заключении кратко подводится итог результатам выполнения заданий курсового проекта.

Приложение должно содержать распечатку текстов программ последовательной и параллельной обработки; в приложения могут быть включены иные материалы по желанию разработчика.

Тестовые теоретические вопросы для защиты курсового проекта по дисциплине «Компьютерные системы и сети» по теме «Программирование параллельных приложений на базе технология MPI»

Примечание: защита курсового проекта состоит из двух этапов:
1) проверка пояснительной записки и разработанного ПО, опрос по представленным результатам;
2) проверка теоретических знаний по тематике курсового проекта методом тестирования.
Полный список тестовых вопросов с целью подготовки приведен ниже. Из указанного списка студенту будет предложена контрольная выборка. В каждом тестовом вопросе правильными могут являться один, два или три варианта ответов, таким образом, положительным ответом на тестовый вопрос является указание студентом соответствующей комбинации правильных ответов.

Вопрос № 1
Последовательная модель программирования это...
Вопрос № 2
Параллельная модель программирования это...
Вопрос № 3
Параллельное программирование с использованием MPI имеет дело с параллелизмом на уровне...
Вопрос № 4
Особенностью параллельной модели программирования является...
Вопрос № 5
Особенностью последовательной модели программирования является...
Вопрос № 6
Какие есть схемы организации параллельных программ?
Вопрос № 7
Чем характеризуются потоки выполнения, относящиеся к одному параллельному приложению?
Вопрос № 8
Чем характеризуются процессы, относящиеся к одному параллельному приложению?
Вопрос № 9
Что такое MPICH?
Вопрос № 10
На программирование для систем с распределенной памятью ориентированы: POSIX Threads, Intel (R) Cluster OpenMP, OpenMP (выбрать правильные варианты)
Вопрос № 11
Какие существуют распространенные средства разработки многопоточных программ?
Вопрос № 12
Какие существуют распространенные средства разработки программ, основанных на модели обмена сообщениями?
Вопрос № 13
Компиляция MPI-программы выполняется командой...
Вопрос № 14
Запуск MPI-программы выполняется командой...
Вопрос № 15
Запуск демонов mpd выполняется командой...
Вопрос № 16
Какие устанавливается соответствия между типами MPI и стандартными типами языка в MPI-программах на языке С?
Вопрос № 17
Получить значение ранга процесса можно с помощью подпрограммы MPI...
Вопрос № 18
Какие значения получает ранг процесса?
Вопрос № 19
Коммуникатор это...
Вопрос № 20
MPI_COMM_NULL это...
Вопрос № 21
MPI_COMM_WORLD это...
Вопрос № 22
Вызов какой подпрограммы MPI может быть первым по порядку вызовом?

Вопрос № 23
Вызов какой подпрограммы MPI может быть последним по порядку вызовом?
Вопрос № 24
Какова правильная последовательность обращений к подпрограммам MPI_Comm_rank, MPI_Comm_size , MPI_Init
Вопрос № 25
Сколько процессов могут участвовать в двухточечном обмене сообщениями
Вопрос № 26
Каковы особенности двухточечного обмена?
Вопрос № 27
Каковы особенности многоточечного обмена?
Вопрос № 28
В MPI существуют следующие типы двухточечных обменов...
Вопрос № 29
В MPI существуют следующие типы одноточечных обменов...
Вопрос № 30
В MPI существуют следующие типы многоточечных обменов:
Вопрос № 31
Стандартная блокирующая двухточечная передача выполняется подпрограммой...
Вопрос № 32
Двухточечная передача с буферизацией выполняется подпрограммой...
Вопрос № 33
При организации одноточечного обмена с буферизацией используется подпрограмма...
Вопрос № 34
При организации двухточечного обмена с буферизацией размер буфера должен превосходить объём пересылаемых данных на величину...
Вопрос № 35
Для чего используется "Джокер" в подпрограмме двухточечного приема сообщения?
Вопрос № 36
Пусть значение параметра count в подпрограмме приёма двухточечного сообщения больше, чем количество элементов в принятом сообщении. Что при этом произойдет?
Вопрос № 37
При стандартной блокирующей двухточечной передаче сообщения: 1) после завершения вызова можно ли использовать любые переменные, использовавшиеся в списке параметров? Приостанавливается ли выполнение параллельной программы до тех пор, пока сообщение будет принято процессом-адресатом?
Вопрос № 38
Каковы особенности работы с буфером для буферизованного двухточечного обмена?
Вопрос № 39
Какие возможности дает двухточечный обмен "по готовности"?
Вопрос № 40
Каковы особенности выполнения блокирующей передачи "по готовности"?
Вопрос № 41
Для каких видов обмена существует операция совместных приёма и передачи:
Вопрос № 42
Каковы особенности выполнения неблокирующего двухточечного обмена:
Вопрос № 43
Для каких видов обмена существует неблокирующий вариант операций передачи сообщений?
Вопрос № 44
Неблокирующий прием сообщений реализован в MPI подпрограммой(мами)...
Вопрос № 45
Какая подпрограмма(мы) выполняет(ют) стандартную передачу в MPI?
Вопрос № 46
Какая подпрограмма(мы) выполняет(ют) неблокирующий прием в MPI?
Вопрос № 47
Какая подпрограмма(мы) выполняет(ют) неблокирующую передачу с буферизацией в MPI?
Вопрос № 48
Первый этап выполнения неблокирующего обмена это...
Вопрос № 49
Второй этап выполнения неблокирующего обмена это...
Вопрос № 50
Повышение каких показателей функционирования программы позволяет неблокирующий обмен?
Вопрос № 51
Подпрограмма MPI_Wait предназначена для...

Вопрос № 52
Подпрограмма MPI_Test предназначена для...
Вопрос № 53
Что происходит после завершения вызова MPI_Wait?
Вопрос № 54
Подпрограммы-пробники предназначены для...
Вопрос № 55
с помощью подпрограммы MPI можно определить размер полученного сообщения?
Вопрос № 56
Функции подпрограммы MPI_Iprobe это...
Вопрос № 57
Функции подпрограммы MPI_Testall это...
Вопрос № 58
Функции подпрограммы MPI_Testany это...
Вопрос № 59
Функции подпрограммы MPI_Waitall это...
Вопрос № 60
Сколько процессов принимают участие в коллективном обмене?
Вопрос № 61
Являются ли блокирующими/неблокирующими операции коллективного обмена для инициировавших их процессов?
Вопрос № 62
Какие предварительные требуются для выполнения коллективного обмена?
Вопрос № 63
Широковещательная рассылка сообщения выполняется подпрограммой...
Вопрос № 64
Коллективная передача данных может ли сочетаться с двухточечным приемом?
Вопрос № 65
Какие операции над данными относятся к числу коллективных обменов?
Вопрос № 66
Какие операции над данными, используемые в MPI, не относятся к числу коллективных обменов?
Вопрос № 67
Данная операция является ли операцией приведения: 1) умножение двух матриц; 2) суммирование элементов массива; 3) определение максимального элемента?
Вопрос № 68
Подпрограмма MPI_Bcast выполняет...
Вопрос № 69
Подпрограмма MPI_Gather выполняет...
Вопрос № 70
Подпрограмма MPI_Scatter выполняет...
Вопрос № 71
Какие подпрограммы MPI выполняют сборку данных?
Вопрос № 72
Данная подпрограмма(мы) выполняет(ют) ли сбор данных: 1) MPI_Get_count; 2) MPI_Scatterv; 3) MPI_Allgather?
Вопрос № 73
Какие подпрограммы MPI являются векторным вариантом операции коллективного обмена?
Вопрос № 74
Что такое "Толщина барьера" при барьерной синхронизации в MPI?
Вопрос № 75
Подпрограмма MPI_Alltoall выполняет...
Вопрос № 76
Подпрограмма MPI_Scan выполняет...