Оглавление

ЛАБОРАТОРНАЯ РАБОТА №8 Распределённые вычисления в кластере MPI.

Цель работы: ***разработать программу, осуществляющую решение поставленной задачи в кластере MPI***

1. Теоретические сведения

1.1 Основные функции MPICH

Интерфейс на основе передачи сообщений (MPI - Message Passing Interface) является стандартной моделью программирования для разработки приложений с явным параллелизмом (т.е., параллельные части программы определяет программист), в которых параллельные процессы взаимодействуют между собой путем передачи сообщений.

Для различных операционных систем и разнообразных сетей передачи данных, используемых в кластерах, разработаны и продолжают разрабатываться специальные реализации MPI. MS MPI (Microsoft MPI) есть стандартная реализация интерфейса передачи сообщений для операционной системы Windows. MS MPI, в свою очередь, базируется на MPICH2 - открытой (open source) реализации стандарта MPI 2.0, разработка которой первоначально была начата в Аргонской национальной лаборатории (Argonne National Laboratory), США.

MS MPI может использоваться в программах, написанных на языках Fortan-77, Fortran-90, C и C++. Пакет Compute Cluster Pack не предоставляет библиотеки классов для MPI в рамках .NET Framework. Тем не менее, программы, реализуемые как последовательные задачи для исполнения под управлением CCS, могут разрабатываться на языках, поддерживаемых платформой .NET. С другой стороны, программы, использующие MPI и написанные на языках, поддерживаемых .NET, могут обращаться к MPI-функциям посредством механизма P/Invoke. В будущих версиях CCS ожидается поддержка MPI в виде стандартных классов .NET Framework.

Общая характеристика интерфейсов MPI-1 и MPI-2

Цель разработки MPI заключалась в создании переносимого, эффективного и гибкого стандарта для параллельных программ, использующих модель на основе передачи сообщений.

Стандарт MPI-1 включает в себя следующие базовые функции:

- управление вычислительным окружением,
- передача сообщений типа "точка-точка",
- коллективные операции взаимодействия,
- использование производных типов данных,
- управление группами и коммуникаторами,
- виртуальные топологии.

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

Стандарт MPI-2, помимо функциональности стандарта MPI-1, включает в себя функции:

- динамического порождения процессов,
- однонаправленной передачи сообщений,
- расширенных коллективных операций,
- параллельного ввода/вывода.

Кроме реализаций MPICH и MPICH-2 Аргонской национальной лаборатории, на которых базируется MS MPI, имеется еще множество других реализаций стандарта MPI, как коммерческих, так и свободно доступных. Примером коммерческой версии является система ScaMPI фирмы Scali, ориентированная, в частности, на поддержку быстрого интерконнекта SCI. Примером широко используемой свободно доступной реализации является система LAM MPI, разработанная в Суперкомпьютерном центре штата Огайо, США.

Функции управления вычислительным окружением

Команды (функции) управления вычислительным окружением стандарта MPI используются для целого ряда целей, таких как инициализация и завершение работы MPI-окружения, получение информации о свойствах и параметрах этого окружения и др.

- `MPI_Init`

Эта функция инициализирует MPI-окружение. Она должна вызываться в каждой MPI-программе до вызова любых других MPI-функций, и, кроме того, она должна вызываться только один раз. В C-программах, эта функция обычно используется для передачи аргументов командной строки каждому из параллельных процессов, хотя это не требуется стандартом MPI и зависит от реализации стандарта.

Формат вызова: `MPI_Init ( &argc, &argv)`

- `MPI_Initialized`

Эта функция определяет вызывалась ли функция инициализации MPI_Init, и возвращает флаг в виде логической истины (1) или логической лжи(0). Необходимость этой функции обусловлена тем, что в сложных программах разные модули могут требовать использования MPI и, так как функция MPI_Init может быть вызвана один раз и только раз каждым процессом, то указанная функция помогает модулю решить нужно ли вызывать MPI_Init или же окружение MPI уже было проинициализировано другим модулем.

Формат вызова: `MPI_Initialized ( &flag )`

- `MPI_Finalize`

Эта функция завершает работу вычислительного окружения MPI. Вызов этой функции должен быть последним обращением к какой-либо MPI-функции в программе: после нее никакая другая MPI-функция вызвана быть не может.

Формат вызова: `MPI_Finalize()`

- `MPI_Comm_size`

Все параллельные процессы, из которых состоит MPI-программа, объединяются в группы, которые управляются так называемыми коммуникаторами (communicators). Именно коммуникаторы обеспечивают взаимодействие параллельных процессов внутри группы.

Функция MPI_Comm_size определяет количество процессов в группе, связанной с данным коммуникатором. Специальный встроенный коммуникатор с именем MPI_COMM_WORLD управляет всеми MPI-процессами в приложении, и потому чаще всего используется в качестве аргумента в данной функции:

Формат вызова: `MPI_Comm_size ( comm., &size )`

- `MPI_Comm-rank`

В рамках группы, связанной с данным коммуникатором, каждый процесс имеет свой уникальный номер, который присваивается процессу системой при инициализации, и который называется рангом процесса. Ранг процесса часто используется для управления исполнением программы, а также для указания отправителя и получателя сообщений, пересылаемых между MPI-процессами.

Данная функция определяет ранг вызывающего процесса внутри группы, связанной с заданным коммуникатором. В разных коммуникаторах, в общем случае, MPI-процесс имеет различные ранги.

Формат вызова: `MPI_Comm-rank ( comm, &rank )`

```
#include "mpi.h"
main( int argc, char **argv )
{   char message[20];
     int myrank;
     MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)  /* код для процесса 0 */
    {     strcpy(message,"Hello, there");
          MPI_Send(message,strlen(message),MPI_CHAR,1,99,
                              MPI_COMM_WORLD);
    }
    else   /* код для процесса 1 */
    {   MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
                             &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

Рис. 1. Пример простой программы MPI

Коммуникационные режимы MPI

Стандартный коммуникационный режим используется в вызове send. В этом режиме решение о том, будет ли исходящее сообщение буферизовано или нет, принимает MPI. MPI может буферизовать исходящее сообщение. В таком случае операция посылки может завершиться до того, как будет вызван соответствующий прием. С другой стороны, буферное пространство может отсутствовать или MPI может отказаться от буферизации исходящего сообщения из-за ухудшения характеристик обмена. В этом случае вызов send не будет завершен, пока данные не будут перемещены в процесс-получатель. В стандартном режиме посылка может стартовать вне зависимости от того, выполнен ли соответствующий прием. Она может быть завершена до окончания приема. Посылка в стандартном режиме является нелокальной операцией: она может зависеть от условий приема. Нежелание разрешать в стандартном режиме буферизацию происходит от стремления сделать программы переносимыми. Поскольку при повышении размера сообщения в любой системе буферных ресурсов может оказаться недостаточно, MPI занимает позицию, что правильная (и, следовательно, переносимая) программа не должна зависеть в стандартном режиме от системной буферизации. Буферизация может улучшить характеристики правильной программы, но она не влияет на результат выполнения программы.

Буферизованный режим операции посылки может стартовать вне зависимости от того, инициирован ли соответствующий прием. Однако, в отличие от стандартной посылки, эта операция является локальной, и ее завершение не зависит от приема. Следовательно, если посылка выполнена и никакого соответствующего приема не инициировано, то MPI буферизует исходящее сообщение, чтобы позволить завершиться вызову send. Если не имеется достаточного объема буферного пространства, будет иметь место ошибка. Объем буферного пространства задается пользователем.

При синхронном режиме посылка может стартовать вне зависимости от того, был ли начат соответствующий прием. Посылка будет завершена успешно, только если соответствующая операция приема стартовала. Завершение синхронной передачи не только указывает, что буфер отправителя может быть повторно использован, но также и отмечает, что получатель достиг определенной точки в своей работе, а именно, что он начал выполнение приема. Если и посылка, и прием являются блокирующими операциями, то использование синхронного режима обеспечивает синхронную коммуникационную семантику: посылка не завершается на любой стороне обмена, пока оба процесса не выполнят рандеву в процессе операции обмена. Выполнение обмена в этом режиме не является локальным.

При обмене по готовности посылка может быть запущена только тогда, когда прием уже инициирован. В противном случае операция является ошибочной, и результат будет неопределенным. Завершение операции посылки не зависит от состояния приема и указывает, что буфер посылки может быть повторно использован. Операция посылки, которая использует режим готовности, имеет ту же семантику, как и стандартная или синхронная передача. Это означает, что отправитель обеспечивает систему дополнительной информацией (именно, что прием уже инициирован), которая может уменьшить накладные расходы. Вследствие этого в правильной программе посылка по готовности может быть замещена стандартной передачей без влияния на поведение программы (но не на характеристики).

Для трех дополнительных коммуникационных режимов используются три дополнительные функции передачи. Коммуникационный режим отмечается одной префиксной буквой: B – для буферизованного, S – для синхронного и R – для режима

готовности.

Операции передачи данных в MPI типа "точка-точка"

Операции передачи данных в MPI типа "точка-точка" представляют собой передачу сообщений между, в точности, двумя MPI-процессами. Один процесс, при этом, выполняет команду Send (послать), тогда как другой процесс выполняет команду Receive (принять).

Выполнение команд Send и Receive осуществляется посредством вызова соответствующих MPI-функций, которые имеют различные типы, или, другими словами, различное назначение:

- блокирующий Send / блокирующий Receive;
- неблокирующий Send / неблокирующий Receive;
- синхронный Send;
- буферированный Send;
- комбинированный Send / Receive;
- send по готовности ("ready" Send).

С любым типом операции Send может состоять в паре любой тип операции Receive.

Функции передачи данных типа "точка-точка" имеют список аргументов одного из следующих форматов:

- блокирующий Send:

```
MPI_Send ( buffer, count, type, dest, tag, comm )
```

- неблокирующий Send:

```
MPI_Isend ( buffer, count, type, dest, tag, comm, request )
```

- Блокирующий Receive:

```
MPI_Recv ( buffer, count, type, source, tag, comm., status )
```

- Неблокирующий Receive:

```
MPI_Irecv ( buffer, count, type, source, tag, comm., request
)
```

Аргументы в этих функциях имеют следующее назначение:

- buffer – место хранения данных, которые посылаются или принимаются;
- count – количество элементов данных конкретного типа, которые посылаются или принимаются;
- type – тип элементарных данных, задаваемый через встроенные MPI-типы, такие как (для языка C):
  - MPI_CHAR;
  - MPI_SHORT;
  - MPI_INT;
  - MPI_LONG;
  - MPI_FLOAT;
  - MPI_DOUBLE;

    - MPI_BYTE;
    - MPI_PACKED и др.
- dest – указывает процесс, которому должно быть доставлено сообщение - задается через ранг принимающего процесса;
- source – аргумент функций приема сообщений, указывающий номер посылающего процесса; указание значения MPI_ANY_SOURCE означает прием сообщения от любого процесса;
- tag – произвольное неотрицательное целое число, присваиваемое программистом для однозначной идентификации сообщения; у парных операций Send и Receive эти числа должны совпадать; указание у операции Receive значения MPI_ANY_TAG может быть использовано для приема любого сообщения, независимо от значения tag ;
- comm – указывает на коммуникатор, в рамках которого трактуются значения аргументов dest и source ; чаще всего используется встроенный коммуникатор MPI_COMM_WORLD ;
- status – для операции Receive, указывает источник (source) сообщения и его тег (tag); в языке C, этот аргумент есть указатель на встроенную структуру MPI_Status ; из этой же структуры может быть получено количество принятых байт посредством функции MPI_Get_count ;
- request – используется в неблокирующих операциях Send и Receive, и задает уникальный "номер запроса"; в языке C, этот аргумент является указателем на встроенную структуру MPI_Request.

К наиболее часто используемым блокирующим функциям передачи сообщений относятся следующие функции:

- `MPI_Send`

Базовая блокирующая операция посылки сообщения. Заканчивает свою работу только тогда, когда программный буфер, из которого берутся данные для посылки, готов для повторного использования.

Формат вызова: `MPI_Send ( &buf, count, datatype, dest, tag, comm)`

- `MPI_Recv`

Принимает сообщения и блокирует вызывающий эту функцию процесс до тех пор, пока в программном буфере не станут доступными принятые данные.

Формат вызова: `MPI_Recv (&buf, count, datatype, source, tag, comm, &status)`

- `MPI_Ssend`

Синхронная блокирующая операция посылки сообщения: посылает сообщение и блокирует вызвавший эту функцию процесс, пока программный буфер не будет готов к повторному использованию и пока процесс-получатель не начал принимать посылаемые сообщения.

Формат вызова: `MPI_Ssend (&buf, count, datatype, dest, tag, comm)`

- `MPI_Bsend, MPI_Buffer_attach`

Перед вызовом MPI_BSend, программист должен вызвать функцию MPI_Buffer_attach для размещения буфера, используемого в MPI_Bsend. Буферированная блокирующая операция посылки сообщения заканчивает свою работу, когда данные из программного буфера скопированы в буфер посылки.

Форматы вызовов:

```
MPI_Buffer_attach (&buffer, size)

MPI_Bsend (&buf, count, datatype, dest, tag, comm)
```

Основные особенности и отличия радиовещательных (коллективных) обменов данными от обменов типа "точка-точка" состоят в следующем:

- принимают и/или передают данные одновременно все процессы группы для указываемого коммуникатора;
- радиовещательная (коллективная) функция выполняет одновременно и прием, и передачу данных, а потому она имеет параметры, часть из которых относится к приему, а часть - к передаче данных;
- как правило, значения всех параметров (за исключением адресов буферов) должны быть идентичны во всех процессах;
- коллективные операции являются блокирующими;
- коллективные операции могут использоваться только для встроенных (predefined) MPI-типов данных, но не могут использоваться для производных (derived) MPI-типов данных.

- `MPI_Bcast`

Посылает сообщение от процесса с рангом "root" (обычно, это процесс с рангом 0) всем другим процессам в группе.

Формат вызова: `MPI_Bcast (&buffer, count, datatype, root, comm)`

- `MPI_Gather`

Собирает сообщения от каждого из процессов в группе в приемный буфер процесса с рангом 'root".

Формат вызова:
```
MPI_Gather (&sendbuf, sendcount, sendtype, &recvbuf,
                          recvcount, recvtype, root, comm)
```

Замечание. sendtype и recvtype, в общем случае, могут различаться, а потому будут задавать разную интерпретацию данных на приемной и передающей стороне;
процесс root также отправляет данные, но в свой же приемный буфер.

- `MPI_Scatter`

Эта функция является обратной к функции MPI_Gather: отдельные части передающего буфера процесса с рангом 'root& распределяются по приемным буферам всех других процессов в группе.

Формат вызова: `MPI_Scatter (&sendbuf, sendcount, sendtype, &recvbuf, recvcount,recvtype, root, comm)`

- `MPI_Allgather`

Эта функция аналогична функции MPI_Gather, за исключением того, что прием данных осуществляет не один процесс, а все процессы: каждый процесс имеет специфическое содержимое в передающем буфере, но все процессы получают в итоге одинаковое содержимое в приемном буфере.

Формат вызова: `MPI_Allgather (&sendbuf, sendcount, sendtype, &recvbuf, recvcount,recvtype, comm)`

- `MPI_Alltoall`

Каждый процесс отдельные части своего передающего буфера рассылает всем остальным процессам; каждый процесс получает эти части от всех остальных и размещает их по порядку рангов процессов, от которых они получены.

Формат вызова: `MPI_Alltoall (&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)`

Коллективные операции

Коллективные операции в MPI выполняют следующие функции:

- MPI_Reduce;
- MPI_Allreduce;
- MPI_Reduce_scatter;
- MPI_Scan.

Помимо встроенных, пользователь может определять использовать свои собственные коллективные операции. Для этого служат функции MPI_Op_create и MPI_Op_free, а также специальный тип данных MPI_Usr_function.

Алгоритм исполнения всех коллективных функций одинаков: в каждом процессе имеется массив с данными и над элементами с одинаковым номеров в каждом из процессов производится одна и та же операция (сложение, произведение, вычисление максимума/минимума и т.п.). Встроенные коллективные функции отличаются друг от друга способом размещения результатов в процессах.

- `MPI_Reduce`

Данная функция выполняет коллективную операцию во всех процессах группы и помещает результат в процесс с рангом root.

Формат вызова:

```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, comm)
```

Встроенных коллективных операций в MPI насчитывается 12:

- MPI_MAX и MPI_MIN - поэлементные максимум и минимум;

- MPI_SUM – сумма векторов;
- MPI_PROD – произведение векторов;
- MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR – логические и двоичные (бинарные) операции И, ИЛИ, исключающее ИЛИ;
- MPI_MAXLOC, MPI_MINLOC – поиск индекса процесса с максимумом/минимумом значения и самого этого значения;

Эти функции могут работать только со следующими типами данных (и только ними):
- MPI_MAX, MPI_MIN – целые и вещественные;
- MPI_SUM, MPI_PROD – целые, вещественные (комплексные – для Фортрана);
- MPI_LAND, MPI_LOR, MPI_LXOR – целые;
- MPI_BAND, MPI_BOR, MPI_BXOR – целые и типа MPI_BYTE;
- MPI_MAXLOC, MPI_MINLOC – вещественные;

- `MPI_Allreduce`

Применяет коллективную операцию и рассылает результат всем процессам в группе.

Формат вызова: `MPI_Allreduce (&sendbuf, &recvbuf, count, datatype, op, comm)`

- `MPI_Reduce_scatter`

Функция применяет вначале коллективную операцию к векторам всех процессов в группе, а затем результирующий вектор разбивается на непересекающиеся сегменты, которые распределяются по процессам. Данная операция эквивалентна вызову функции MPI_Reduce, за которым производится вызов MPI_Scatter.

Формат вызова:

```
MPI_Reduce_scatter (&sendbuf, &recvbuf, recvcount, datatype, op, comm)
```

- `MPI_Scan`

Данная операция аналогична функции MPI_Allreduce в том отношении, что после ее выполнения каждый процесс получает результирующий массив. Главное отличие данной функции состоит в том, что содержимое результирующего массива в процессе i является результатом выполнения коллективной операции над массивами из процессов с номерами от 0 до i включительно.

Формат вызова: `MPI_Scan ( &sendbuf, &recvbuf, count, datatype, op, comm )`

Управление процессами в MPI

Управление процессами в MPI происходит посредством организации их в группы, управляемые коммуникаторами

Группа есть упорядоченное множество процессов. Каждому процессу в группе присваивается уникальный целочисленный номер – ранг. Значения ранга изменяются от 0

до N - 1, где N есть количество процессов в группе. В MPI, группа представляется в памяти компьютера в виде объекта, доступ к которому программист осуществляет с помощью "обработчика" (handle) MPI_Group. С группой всегда связывается коммуникатор, также представляемый в виде объекта.

Коммуникатор обеспечивает взаимодействие между процессами, относящимися к одной и той же группе. Поэтому, во всех MPI-сообщениях одним из аргументов задается коммуникатор. Коммуникаторы как объекты также доступны программисту с помощью обработчиков. В частности, обработчик коммуникатора, который включает в себя все процессы задачи, называется:

`MPI_COMM_WORLD.`

Основные цели средств организации процессов в группы:

- позволяют организовывать задачи, объединяя в группы процессы, основываясь на их функциональном назначении;
- позволяют осуществлять коллективные операции (см. Раздел 4) только на заданном множестве процессов;
- предоставляют базис для организации пользователем виртуальных топологий;
- обеспечивают безопасную передачу сообщений в рамках одной группы.

Группы/коммуникаторы являются динамическими – они могут создаваться и уничтожаться во время исполнения программы.

Процессы могут относиться к более, чем одной группе/коммуникатору. В каждой группе/коммуникаторе, каждый процесс имеет уникальный номер (ранг).

MPI обладает богатой библиотекой функций, относящихся к группам, коммуникаторам и виртуальным топологиям, типичный алгоритм использования которых представлен ниже:

- получить обработчик глобальной группы, связанной с коммуникатором MPI_COMM_WORLD, используя функцию MPI_Comm_group;
- создать новую группу как подмножество глобальной группы, используя функцию MPI_Group_incl;
- создать новый коммуникатор для вновь созданной группы, используя функцию MPI_Comm_create;
- получить новый ранг процесса во вновь созданном коммуникаторе, используя функцию MPI_Comm_rank;
- выполнить обмен сообщениями между процессами в рамках вновь созданной группы;
- по окончании работы, освободить (уничтожить) группу и коммуникатор, используя функции MPI_Group_free и MPPI_Comm_free.

Организация логических топологий процессов

В терминах MPI, виртуальная топология описывает отображение MPI процессов на некоторую геометрическую конфигурацию процессоров.

В MPI поддерживается два основных типа топологий – декартовые (решеточные) топологии и топологии в виде графа.

MPI-топологии являются виртуальными – связь между физической структурой параллельной машины и топологией MPI-процессов может и отсутствовать.

Виртуальные топологии строятся на основе групп и коммуникаторов, и "программируется" разработчиком параллельного приложения.
Смысл использования виртуальных топологий заключается в том, что они в некоторых случаях удобны для задач со специфической коммуникационной структурой. Например, декартова топология удобна для задач, в которых обрабатывающие элементы в процессе вычислений обмениваются данными только со своими 4-мя непосредственными соседями. В конкретных реализациях, возможна оптимизация отображения MPI-процессов на физическую структуру заданной параллельной машины.

1.2 Организация распределённых вычислений в кластере MPI MPICH

Запуск приложений в MPI MPICH 1.2

Запуск приложений осуществляется с помощью команды MPIRun.exe. Это наиболее распространенный способ запуска приложений. Команда MPIRun.exe находится в [MPICH Launcher Home]\bin directory.

Использование команды:
- MPIRun configfile [-logon] [args ...]
- MPIRun -np #processes [-logon][-env "var1=val1|var2=val2..."] executable [args ...]
- MPIRun -localonly #processes [-env "var1=val1|var2=val2..."]executable [args ...]

Аргументы в скобках являются опциями.

Формат файла конфигурации config следующий:

```
exe c:\somepath\myapp.exe   или   \\host\share\somepath\myapp.exe
[args arg1 arg2 arg3 ...]
[env VAR1=VAL1|VAR2=VAL2|...|VARn=VALn]
hosts
hostA #procs [path\myapp.exe]
hostB #procs [\\host\share\somepath\myapp2.exe]
hostC #procs
```

Можно описать путь к исполняемому коду отдельной строкой для каждого хоста, тем самым вводя MPMD–программирование. Если путь не описывается, тогда используется по умолчанию путь из строки exe.

```
exe c:\temp\slave.exe
env MINX=0|MAXX=2|MINY=0|MAXY=2
args -i c:\temp\cool.points
hosts
fry 1 c:\temp\master.exe
fry 1
#light 1
jazz 2
```

Рис. 1. Пример простого файла конфигурации

Во втором случае запускается количество процессов, равное #processes, начиная с текущей машины и затем по одному процессу на каждую следующую машину, описанную на этапе инсталляции, пока все процессы не исчерпаны. Если процессов больше, чем машин, то распределение повторяется по циклу. В третьем случае команда запускает

выполнение нескольких процессов на локальной машине, использующей устройство разделяемой памяти.

```
-env "var1=val1|var2=val2|var3=val3|...varn=valn"
```

Эта опция устанавливает переменные среды, описанные в строке, перед запуском каждого процесса. Следует помнить, что надо взять в кавычки строку, чтобы приглашение команды не интерпретировало вертикальные линии как конвейерную команду.

```
-logon
```

Эта опция mpirun приглашает установить имя пользователя(account) и пароль (password). Если использовать эту опцию, можно описать исполняемую программу, которая размещена в разделяемой памяти. Если не применять –logon, то исполняемая программа должна находиться на каждом хосте локально. Необходимо использовать mpiregister.exe, чтобы закодировать имя и пароль в регистре и избежать приглашения.

Процедура MPIConfig.exe

Чтобы выполнять приложение на различных хостах без описания их в конфигурационном файле, процедура запуска должна знать, какие хосты уже инсталлированы. MPIConfig.exe – это простая программа, которая находит хосты, где процедура запуска уже установлена, и записывает этот список хостов в регистр, который представлен в специальном окне “MPI Configuration Tool”. По этой информации MPIRun может выбрать из списка в окне хосты, где следует запустить процессы. Операции с окном следующие:

- Refresh опрашивает сеть для обновления списка имен хостов;
- Find подключается к регистру на каждом хосте и определяет по списку, успешно ли уже установлена процедура запуска. Когда он заканчивается, то избранные хосты такие, где эта процедура установлена;
- Verify приводит к тому, что mpiconfig подключается к каждому из избранных хостов и убеждается, что DCOM–сервер достижим;
- Set приводит к тому, что вызывается окно – “MPICH Registry settings” – и выполняется следующий диалог:
    - если выбрать "set HOSTS", mpiconfig создаст группу из всех избранных хостов и запишет этот список имен в регистр на каждом хосте. Когда MPIRun выполняется из любого хоста группы с опцией –np, хосты будут выбираться из этого списка;
    - если выбрать "set TEMP", mpiconfig запишет этот директорий в регистр каждого хоста. remote shell server должен создать временный файл для связи с первым запущенным процессом, и этот файл должен располагаться в ячейке, которая пригодна для read/write как для remote shell service, так и для процедуры запуска mpich приложения. remote shell server использует этот вход, чтобы определить, где записать этот файл. По умолчанию устанавливается C:\;
    - позиция “launch timeout” указывает, как долго MPIRun будет ждать, пока не убедится, что процесс запустить нельзя. Время задается в миллисекундах.

Процедура MPIRegister.exe

Процедура MPIRegister.exe используется для того, чтобы закодировать имя и пароль в регистр для текущего пользователя. Он находится в [MPICH Launcher Home]\bin directory. Эта информация используется командой MPIRun.exe для запуска приложений в контексте

этого пользователя. Если mpiregister не используется, то mpirun будет постоянно приглашать ввести имя и пароль.

Использование:

- `MPIRegister`
- `MPIRegister -remove`

Сначала команда MPIRegister попросит ввести имя пользователя. Введите имя в форме [Domain\]Account, где domain name есть опция (например, mcs\ashton or ashton). Затем дважды выполнится приглашение для ввода пароля. Затем последует вопрос, желаете ли Вы хранить эти параметры постоянно. Если Вы говорите "да", то эти данные будут сохранены на жестком диске. Если "нет ", данные останутся только в памяти. Это означает, что Вы можете запускать mpirun много раз и при этом не потребуется вводить имя и пароль. Однако при перезагрузке машины и использовании mpirun опять возникнет приглашение для ввода имени и пароля. remove приводит к удалению информации из регистра.

Библиотека MPE (Multi-Processing Environment)

MPE–процедуры делятся на несколько категорий [19]:

- параллельная графика (Parallel X graphics). Эти процедуры обеспечивают доступ всем процессам к разделяемому X–дисплею. Они создают удобный вывод для параллельных программ, позволяют чертить текст, прямоугольники, круги, линии и так далее.
- регистрация (Logging). Одним из наиболее распространенных средств для анализа характеристик параллельных программ является файл трассы отмеченных во времени событий – логфайл (logfile). Библиотека MPE создает возможность легко получить такой файл в каждом процессе и собрать их вместе по окончанию работы. Она так же автоматически обрабатывает рассогласование и дрейф часов на множественных процессорах, если система не обеспечивает синхронизацию часов. Это библиотека для пользователя, который желает создать свои собственные события и программные состояния.
- последовательные секции (Sequential Sections). Иногда секция кода, которая выполняется на ряде процессов, должна быть выполне на только по одному процессу за раз в порядке номеров этих процессов. MPE имеет функции для такой работы.
- обработка ошибок (Error Handling). MPI имеет механизм, который позволяет пользователю управлять реакцией реализации на ошибки времени исполнения, включая возможность создать свой собственный обработчик ошибок.

В настоящее время MPE предлагает следующие три профилирующие библиотеки:

- Tracing Library (библиотека трассирования) – трассирует все MPI–вызовы. Каждый вызов предваряется строкой, которая содержит номер вызывающего процесса в MPI_COMM_WORLD и сопровождается другой строкой, указывающей, что вызов завершился. Большинство процедур send и receive также указывают значение count, tag и имена процессов, которые посылают или принимают данные;
- Animation Libraries (анимационная библиотека) – простая форма программной анимации в реальном времени, которая требует процедур X–окна;
- Logging Libraries (библиотека регистрации) – самые полезные и широко используемые профилирующие библиотеки в MPE. Они формируют базис для генерации логфайлов из пользовательских программ. Сейчас имеется три

различных формата логфайлов, допустимых в MPE. По умолчанию используется формат CLOG. Он содержит совокупность событий с единым отметчиком времени. Формат ALOG больше не развивается и поддерживается для обеспечения совместимости с ранними программами. И наиболее мощным является формат – SLOG (для Scalable Logfile), который может быть конвертирован из уже имеющегося CLOG–файла или получен прямо из выполняемой программы (для этого необходимо установить пременную среды MPE_LOG_FORMAT в SLOG).

Создание файла регистрации MPE

Чтобы создать файл регистрации, необходимо вызвать процедуру MPE_Log_event. Кроме того, каждый процесс должен вызвать процедуру MPE_Init_log, чтобы приготовиться к регистрации, и MPE_Finish_log, чтобы объединить файлы, сохраняемые локально при каждом процессе в единый логфайл. MPE_Stop_log используется, чтобы приостановить регистрацию, хотя таймер продолжает работать. MPE_Start_log возобновляет регистрацию.

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

```
#include "mpi.h"
#include "mpe.h"
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[ ])
{  int  n, myid, numprocs;
    double PI25DT = 3.14159265358979323846264643;
    double mypi, pi, h, sum, x, startwtime = 0.0, endwtime;
    int   event1a, event1b, event2a, event2b,event3a, event3b,
event4a, event4b;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPE_Init_log();
/*  Пользователь не дает имена событиям, он получает их из MPE */
/*         определяем      8   событий      для   4   состояний
Bcast",”Compute",”Reduce" ,”Sync" */
    event1a = MPE_Log_get_event_number();
    event1b = MPE_Log_get_event_number();
    event2a = MPE_Log_get_event_number();
    event2b = MPE_Log_get_event_number();
    event3a = MPE_Log_get_event_number();
    event3b = MPE_Log_get_event_number();
    event4a = MPE_Log_get_event_number();
    event4b = MPE_Log_get_event_number();
    if (myid == 0) {
```

```
/* задаем  состояние "Bcast" как время между событиями event1a и
event1b. */
39
MPE_Describe_state(event1a, event1b, "Broadcast", "red");
/* задаем  состояние "Compute" как время между событиями event2a
и vent2b. */
MPE_Describe_state(event2a, event2b, "Compute",   "blue");
/* задаем состояние "Reduce" как время между событиями event3a и
event3b. */
MPE_Describe_state(event3a, event3b, "Reduce",    "green");
/* задаем состояние "Sync" как время между событиями event4a и
event4b. */
MPE_Describe_state(event4a, event4b, "Sync",   "orange");
    }
    if (myid == 0)
    { n = 1000000;
startwtime = MPI_Wtime();
     }
     MPI_Barrier(MPI_COMM_WORLD);
     MPE_Start_log();
                              /* регистрируем событие event1a */
     MPE_Log_event(event1a, 0, "start broadcast");
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
                             /* регистрируем событие event1b */
     MPE_Log_event(event1b, 0, "end broadcast");
                              /* регистрируем событие event4a */
     MPE_Log_event(event4a,0,"Start Sync");
     MPI_Barrier(MPI_COMM_WORLD);
                            /* регистрируем событие event4b */
     MPE_Log_event(event4b,0,"End Sync");
                            /* регистрируем событие event2a */
     MPE_Log_event(event2a, 0, "start compute");
     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;
                              /* регистрируем событие event2b */
      MPE_Log_event(event2b, 0, "end compute");
                                 /* регистрируем событие event3a
*/
      MPE_Log_event(event3a, 0, "start reduce");

MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
                                 /* регистрируем событие event3b
*/
      MPE_Log_event(event3b, 0, "end reduce");
      MPE_Finish_log("cpilog");
      if (myid == 0)
      { endwtime = MPI_Wtime();
printf("pi is approximately %.16f, Error is %.16f\n",
```

```
                    pi, fabs(pi - PI25DT));
printf("wall clock time = %f\n", endwtime-startwtime);
     }
40
    MPI_Finalize();
    return(0);
}
```

Рис.2. Пример программы вычисления π с использование MPE-лога

Анализ лог-файлов MPE

После выполнения программы MPI, которая содержит процедуры MPE для регистрации событий, директорий, где она выполнялась, содержит файл событий, отсортированных по времени, причем время скорректировано с учетом «плавания» частоты генераторов. Можно написать много программ для анализа этого файла и представления информации.

Информативнее графическое представление обеспечивают специализированные программы, например, upshot и jampshot.

Входные процедуры MPE используются, чтобы создать логфайлы (отчеты) о событиях, которые имели место при выполнении параллельной программы. Форматы этих процедур представлены ниже:

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

Эти процедуры позволяют пользователю включать только те события, которые ему интересны в данной программе. Базовые процедуры − MPE_Init_log, MPE_Log_event и MPE_Finish_log:

- MPE_Init_log должна вызываться всеми процессами, чтобы инициализировать необходимые структуры данных. MPE_Finish_log собирает отчеты из всех процессов, объединяет их, выравнивает по общей шкале времени. Затем процесс с номером 0 в коммуникаторе MPI_COMM_WORLD записывает отчет в файл, имя которого указано в аргументе;
- одиночное событие устанавливается процедурой MPE_Log_event, которая задает тип события (выбирает пользователь), целое число и строку для данных. Чтобы разместить логфайл, который будет нужен для анализа или для программы визуализации (подобной upshot);
- процедура MPE_Describe_state позволяет добавить события и описываемые состояния, указать точку старта и окончания для каждого состояния. При желании для визуализации отчета можно использовать цвет.
- MPE_Stop_log и MPE_Start_log предназначены для того, чтобы динамически включать и выключать создание отчета.

Эти процедуры используются в одной из профилирующих библиотек, поставляемых с дистрибутивом для автоматического запуска событий библиотечных вызовов MPI.

Две переменные среды TMPDIR и MPE LOG FORMAT нужны пользователю для установки некоторых параметров перед генерацией логфайлов.

MPE LOG FORMAT – определяет формат логфайла, полученного после исполнения приложения, связанного с MPE–библиотекой. MPE LOG FORMAT может принимать только значения CLOG, SLOG и ALOG. Когда MPE LOG FORMAT установлен в NOT, предполагается формат CLOG.

TMPDIR – описывает директорий, который используется как временная память для каждого процесса. По умолчанию, когда TMPDIR есть NOT, будет использоваться /“tmp”. Когда пользователю нужно получить очень длинный логфайл для очень длинной MPI–работы, пользователь должен убедиться, что TMPDIR достаточно велик, чтобы хранить временный логфайл, который будет удален, если объединенный файл будет создан успешно.

1.3 Примеры программ

Ленточное перемножения матриц

```
void MatrixMultiplicationMPI(double *&A, double *&B, double *&C, int &Size) {
        int dim = Size;
        int i, j, k, p, ind;
        double temp;
        MPI_Status Status;
        int ProcPartSize = dim/ProcNum;
        int ProcPartElem = ProcPartSize*dim;
        double* bufA = new double[ProcPartElem];
        double* bufB = new double[ProcPartElem];
        double* bufC = new double[ProcPartElem];
        int ProcPart = dim/ProcNum, part = ProcPart*dim;
        if (ProcRank == 0) {
                Flip(B, Size);
        }

        MPI_Scatter(A, part, MPI_DOUBLE, bufA, part, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
        MPI_Scatter(B, part, MPI_DOUBLE, bufB, part, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

        temp = 0.0;
        for (i=0; i < ProcPartSize; i++) {
                for (j=0; j < ProcPartSize; j++) {
                        for (k=0; k < dim; k++)
                                temp += bufA[i*dim+k]*bufB[j*dim+k];
                        bufC[i*dim+j+ProcPartSize*ProcRank] = temp;
                        temp = 0.0;
                }
        }

        int NextProc; int PrevProc;
        for (p=1; p < ProcNum; p++) {
                NextProc = ProcRank+1;
                if (ProcRank == ProcNum-1)
                        NextProc = 0;
                PrevProc = ProcRank-1;
                if (ProcRank == 0)
```

```
                        PrevProc = ProcNum-1;
                MPI_Sendrecv_replace(bufB, part, MPI_DOUBLE, NextProc, 0,
PrevProc, 0, MPI_COMM_WORLD, &Status);
                temp = 0.0;
                for (i=0; i < ProcPartSize; i++) {
                        for (j=0; j < ProcPartSize; j++) {
                                for (k=0; k < dim; k++) {
                                        temp += bufA[i*dim+k]*bufB[j*dim+k];
                                }
                                if (ProcRank-p >= 0 )
                                        ind = ProcRank-p;
                                else ind = (ProcNum-p+ProcRank);
                                bufC[i*dim+j+ind*ProcPartSize] = temp;
                                temp = 0.0;
                        }
                }
        }

        MPI_Gather(bufC, ProcPartElem, MPI_DOUBLE, C, ProcPartElem,
MPI_DOUBLE, 0, MPI_COMM_WORLD);

        delete []bufA;
        delete []bufB;
        delete []bufC;
}
```

Алгоритм Фокса перемножения матриц

1. Главная функция программы. Определяет основную логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 7.1
// Алгоритм Фокса умножения матриц – блочное представление данных
//   Условия выполнения программы: все матрицы квадратные,
//   размер блоков и их количество по горизонтали и вертикали
//   одинаково, процессы образуют квадратную решетку
int ProcNum = 0;        // Количество доступных процессов
int ProcRank = 0;       // Ранг текущего процесса
int GridSize;           // Размер виртуальной решетки процессов
int GridCoords[2];      // Координаты текущего процесса в процессной
// решетке
MPI_Comm GridComm;      // Коммуникатор в виде квадратной решетки
MPI_Comm ColComm;       // коммуникатор – столбец решетки
MPI_Comm RowComm;       // коммуникатор – строка решетки

void main ( int argc, char * argv[] ) {
  double* pAMatrix;     // Первый аргумент матричного умножения
  double* pBMatrix;     // Второй аргумент матричного умножения
  double* pCMatrix;     // Результирующая матрица
  int Size;             // Размер матриц
  int BlockSize;        // Размер матричных блоков, расположенных
                        // на процессах
  double *pAblock;      // Блок матрицы A на процессе
  double *pBblock;      // Блок матрицы B на процессе
  double *pCblock;      // Блок результирующей матрицы C на процессе
  double *pMatrixAblock;
  double Start, Finish, Duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init(&argc, &argv);
```

```
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  GridSize = sqrt((double)ProcNum);
  if (ProcNum != GridSize*GridSize) {
    if (ProcRank == 0) {
      printf ("Number of processes must be a perfect square \n");
    }
  }
  else {
    if (ProcRank == 0)
      printf("Parallel matrix multiplication program\n");

    // Создание виртуальной решетки процессов и коммуникаторов
    // строк и столбцов
    CreateGridCommunicators();

    // Выделение памяти и инициализация элементов матриц
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock,
      pBblock, pCblock, pMatrixAblock, Size, BlockSize );
    // Блочное распределение матриц между процессами
    DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
      BlockSize);

    // Выполнение параллельного метода Фокса
    ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
      pCblock, BlockSize);

    // Сбор результирующей матрицы на ведущем процессе
    ResultCollection(pCMatrix, pCblock, Size, BlockSize);

    // Завершение процесса вычислений
    ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
      pCblock, pMatrixAblock);
  }

  MPI_Finalize();
}
```

Функция CreateGridCommunicators. Данная функция создает коммуникатор в виде двумерной квадратной решетки, определяет координаты каждого процесса в этой решетке, а также создает коммуникаторы отдельно для каждой строки и каждого столбца.

Создание решетки производится при помощи функции MPI_Cart_create (вектор Periodic определяет возможность передачи сообщений между граничными процессами строк и столбцов создаваемой решетки). После создания решетки каждый процесс параллельной программы будет иметь координаты своего положения в решетке; получение этих координат обеспечивается при помощи функции MPI_Cart_coords.

Формирование топологий завершается созданием множества коммуникаторов для каждой строки и каждого столбца решетки в отдельности (функция MPI_Cart_sub).

```
// Создание коммуникатора в виде двумерной квадратной решетки
// и коммуникаторов для каждой строки и каждого столбца решетки
void CreateGridCommunicators() {
  int DimSize[2];       // Количество процессов в каждом измерении
                        // решетки
  int Periodic[2];      // =1 для каждого измерения, являющегося
                        // периодическим
  int Subdims[2];       // =1 для каждого измерения, оставляемого
                        // в подрешетке
```

```
  DimSize[0] = GridSize;
  DimSize[1] = GridSize;
  Periodic[0] = 0;
  Periodic[1] = 0;

  // Создание коммуникатора в виде квадратной решетки
  MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);

  // Определение координат процесса в решетке
  MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

  // Создание коммуникаторов для строк процессной решетки
  Subdims[0] = 0;  // Фиксация измерения
  Subdims[1] = 1;  // Наличие данного измерения в подрешетке
  MPI_Cart_sub(GridComm, Subdims, &RowComm);

  // Создание коммуникаторов для столбцов процессной решетки
  Subdims[0] = 1;
  Subdims[1] = 0;
  MPI_Cart_sub(GridComm, Subdims, &ColComm);
}
```

Функция ProcessInitialization. Данная функция определяет параметры решаемой задачи (размеры матриц и их блоков), выделяет память для хранения данных и осуществляет ввод исходных матриц (или формирует их при помощи какого-либо датчика случайных чисел). Всего в каждом процессе должна быть выделена память для хранения четырех блоков – для указателей на выделенную память используются переменные pAblock, pBblock, pCblock, pMatrixAblock. Первые три указателя определяют блоки матриц A, B и C соответственно. Следует отметить, что содержимое блоков pAblock и pBblock постоянно меняется в соответствии с пересылкой данных между процессами, в то время как блок pMatrixAblock матрицы A остается неизменным и применяется при рассылках блоков по строкам решетки процессов (см. функцию AblockCommunication).

Для определения элементов исходных матриц будем использовать функцию RandomDataInitialization, реализацию которой читателю предстоит выполнить самостоятельно.

```
// Функция для выделения памяти и инициализации исходных данных
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, double* &pAblock, double* &pBblock,
  double* &pCblock, double* &pTemporaryAblock, int &Size,
  int &BlockSize ) {
  if (ProcRank == 0) {
    do {
      printf("\nВведите размер матриц: ");
      scanf("%d", &Size);

      if (Size%GridSize != 0) {
        printf ("Размер матриц должен быть кратен размеру сетки! \n");
      }
    }
    while (Size%GridSize != 0);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  BlockSize = Size/GridSize;

  pAblock = new double [BlockSize*BlockSize];
  pBblock = new double [BlockSize*BlockSize];
  pCblock = new double [BlockSize*BlockSize];
  pTemporaryAblock = new double [BlockSize*BlockSize];
```

```
  for (int i=0; i<BlockSize*BlockSize; i++) {
    pCblock[i] = 0;
  }
  if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    RandomDataInitialization(pAMatrix, pBMatrix, Size);
  }
}
```

Функции DataDistribution и ResultCollection. После задания исходных матриц на нулевом процессе необходимо осуществить распределение исходных данных. Для этого предназначена функция DataDistribution. Может быть предложено два способа выполнения блочного разделения матриц между процессорами, организованными в двумерную квадратную решетку. В первом из них для организации передачи блоков в рамках одной и той же коммуникационной операции можно сформировать средствами MPI производный тип данных. Во втором способе можно организовать двухэтапную процедуру. На первом этапе матрица разделяется на горизонтальные полосы. Эти полосы распределяются на процессы, составляющие нулевой столбец процессорной решетки. Далее каждая полоса разделяется на блоки между процессами, составляющими строки процессорной решетки.

Для выполнения сбора результирующей матрицы из блоков предназначена функция ResultCollection. Сбор данных также можно выполнить либо с использованием производного типа данных, либо при помощи двухэтапной процедуры, зеркально отображающей процедуру распределения матрицы.

Реализация функций DataDistribution и ResultCollection представляет собой задание для самостоятельной работы.

Функция AblockCommunication. Функция выполняет рассылку блоков матрицы A по строкам процессорной решетки. Для этого в каждой строке решетки определяется ведущий процесс Pivot, осуществляющий рассылку. Для рассылки используется блок pMatrixAblock, переданный в процесс в момент начального распределения данных. Выполнение операции рассылки блоков осуществляется при помощи функции MPI_Bcast. Следует отметить, что данная операция является коллективной и ее локализация пределами отдельных строк решетки обеспечивается за счет использования коммуникаторов RowComm, определенных для набора процессов каждой строки решетки в отдельности.

```
// Рассылка блоков матрицы A по строкам решетки процессов
void ABlockCommunication (int iter, double *pAblock,
  double* pMatrixAblock, int BlockSize) {

  // Определение ведущего процесса в строке процессной решетки
  int Pivot = (GridCoords[0] + iter) % GridSize;

  // Копирование передаваемого блока в отдельный буфер памяти
  if (GridCoords[1] == Pivot) {
    for (int i=0; i<BlockSize*BlockSize; i++)
      pAblock[i] = pMatrixAblock[i];
  }

  // Рассылка блока
  MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot,
    RowComm);
}
```

Функция BlockMultiplication. Функция обеспечивает перемножение блоков матриц A и B. Следует отметить, что для более легкого понимания рассматриваемой программы приводится простой вариант реализации функции – выполнение операции блочного умножения может быть существенным образом оптимизировано для сокращения времени вычислений. Данная оптимизация может быть направлена, например, на повышение эффективности использования кэша процессоров, векторизации выполняемых операций и т.п.

```
// Умножение матричных блоков
void BlockMultiplication (double *pAblock, double *pBblock,
  double *pCblock, int BlockSize) {
  // Вычисление произведения матричных блоков
  for (int i=0; i<BlockSize; i++) {
    for (int j=0; j<BlockSize; j++) {
      double temp = 0;
      for (int k=0; k<BlockSize; k++ )
        temp += pAblock [i*BlockSize + k] * pBblock [k*BlockSize + j];
      pCblock [i*BlockSize + j] += temp;
    }
  }
}
```

Функция BblockCommunication. Функция выполняет циклический сдвиг блоков матрицы B по столбцам процессорной решетки. Каждый процесс передает свой блок следующему процессу NextProc в столбце процессов и получает блок, переданный из предыдущего процесса PrevProc в столбце решетки. Выполнение операций передачи данных осуществляется при помощи функции MPI_SendRecv_replace, которая обеспечивает все необходимые пересылки блоков, используя при этом один и тот же буфер памяти pBblock. Кроме того, эта функция гарантирует отсутствие возможных тупиков, когда операции передачи данных начинают одновременно выполняться несколькими процессами при кольцевой топологии сети.

```
// Циклический сдвиг блоков матрицы B вдоль столбца процессной
// решетки
void BblockCommunication (double *pBblock, int BlockSize) {
  MPI_Status Status;
  int NextProc = GridCoords[0] + 1;
  if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
  int PrevProc = GridCoords[0] - 1;
  if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;
  MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
    NextProc, 0, PrevProc, 0, ColComm, &Status);
}
```

Функция ParallelResultCalculation. Для непосредственного выполнения параллельного алгоритма Фокса умножения матриц предназначена функция ParallelResultCalculation, которая реализует логику работы алгоритма.

```
// Функция для параллельного умножения матриц
void ParallelResultCalculation(double* pAblock, double* pMatrixAblock,
   double* pBblock, double* pCblock, int BlockSize) {
  for (int iter = 0; iter < GridSize; iter ++) {
    // Рассылка блоков матрицы A по строкам процессной решетки
    ABlockCommunication (iter, pAblock, pMatrixAblock, BlockSize);
    // Умножение блоков
    BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);
    // Циклический сдвиг блоков матрицы B в столбцах процессной
    // решетки
    BblockCommunication(pBblock, BlockSize);
  }
}
```

Распределённое решение СЛАУ методом Гаусса

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве базовой подзадачи можно принять тогда все вычисления, связанные с обработкой одной строки матрицы A и соответствующего элемента вектора b.

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить (n-1) итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольному виду.

Выполнение итерации i, 0<=i<n-1, прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i, соответствующего исключаемой переменной x_i. Поскольку строки матрицы A распределены по подзадачам, для поиска максимального значения подзадачи с номерами k, k<i, должны обменяться своими элементами при исключаемой переменной x_i. После сбора всех необходимых данных в каждой подзадаче может быть определено, какая из подзадач содержит ведущую строку и какое значение является ведущим элементом.

Далее для продолжения вычислений ведущая подзадача должна разослать свою строку матрицы A и соответствующий элемент вектора b всем остальным подзадачам с номерами k, k<i. Получив ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

При выполнении обратного хода метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, 0<=i<n-1, определяет значение своей переменной x_i, это значение должно быть разослано всем подзадачам с номерами k, k<i. Далее подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора b.

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и сбалансированным объемом передаваемых данных. В случае когда размер матрицы, описывающей систему линейных уравнений, оказывается большим, чем число доступных процессоров (т.е. p<n), базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. Однако применение последовательной схемы разделения данных для параллельного решения систем линейных уравнений приведет к неравномерной вычислительной нагрузке процессоров: по мере исключения (на прямом ходе) или определения (на обратном ходе) неизвестных в методе Гаусса для все большей части процессоров все необходимые вычисления будут завершены и процессоры окажутся простаивающими. Возможное решение проблемы балансировки вычислений может состоять в использовании ленточной циклической схемы для распределения данных между укрупненными подзадачами. В этом случае матрица A делится на наборы (полосы) строк (рис. 3).

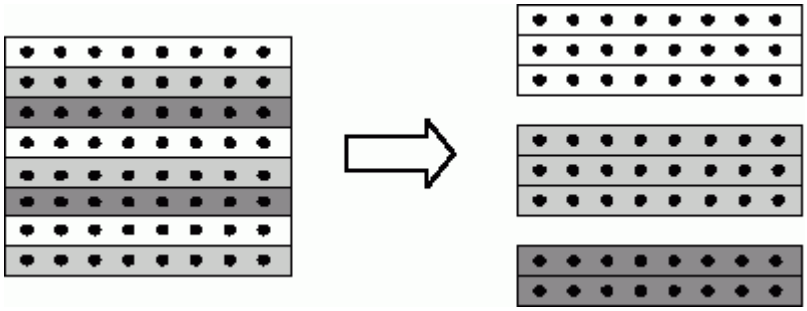

Рис. 3. Пример использования ленточной циклической схемы разделения строк матрицы между тремя процессорами

Сопоставив схему разделения данных и порядок выполнения вычислений в методе Гаусса, можно отметить, что использование циклического способа формирования полос позволяет обеспечить лучшую балансировку вычислительной нагрузки между подзадачами.

Распределение подзадач между процессорами должно учитывать характер выполняемых в методе Гаусса коммуникационных операций. Основным видом информационного взаимодействия подзадач является операция передачи данных от одного процессора всем процессорам вычислительной системы. Как результат, для эффективной реализации требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должны иметь структуру гиперкуба или полного графа.

Рассмотрим возможный вариант параллельной реализации метода Гаусса для решения систем линейных уравнений. Следует отметить, что для получения более простого вида программы для разделения матрицы между процессами используется ленточная последовательная схема (полосы матрицы образуют последовательные наборы соседних строк матрицы). В описании программы реализация отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 8.1. — Алгоритм Гаусса решения систем линейных уравнений
int ProcNum;            // Число доступных процессоров
int ProcRank;           // Ранг текущего процессора
int *pParallelPivotPos;        // Номера строк, которые были выбраны ведущими
int *pProcPivotIter;    // Номера итераций, на которых строки
                        // процессора использовались в качестве ведущих
void main(int argc, char* argv[]) {
  double* pMatrix;              // Матрица линейной системы
  double* pVector;              // Вектор правых частей линейной системы
  double* pResult;              // Вектор неизвестных
  double *pProcRows;    // Строки матрицы A
  double *pProcVector;          // Блок вектора b
  double *pProcResult; // Блок вектора x
  int     Size;         // Размер матрицы и векторов
  int     RowNum;       // Количество строк матрицы
  double start, finish, duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank );
```

```
  MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum );

  if (ProcRank == 0)
    printf("Параллельный метод Гаусса для решения систем линейных
уравнений\n");

  // Выделение памяти и инициализация данных
  ProcessInitialization(pMatrix, pVector, pResult,
    pProcRows, pProcVector, pProcResult, Size, RowNum);

  // Распределение исходных данных
  DataDistribution(pMatrix, pProcRows, pVector, pProcVector,
  Size, RowNum);

  // Выполнение параллельного алгоритма Гаусса
  ParallelResultCalculation(pProcRows, pProcVector, pProcResult, Size,
    RowNum);

  // Сбор найденного вектора неизвестных на ведущем процессе
  ResultCollection(pProcResult, pResult);

  // Завершение процесса вычислений
  ProcessTermination(pMatrix, pVector, pResult, pProcRows,
    pProcVector, pProcResult);

  MPI_Finalize();
}
```

Следует пояснить использование дополнительных массивов. Элементы массива pParallelPivotPos определяют номера строк матрицы, выбираемых в качестве ведущих, по итерациям прямого хода метода Гаусса. Именно в этом порядке должны выполняться затем итерации обратного хода для определения значений неизвестных системы линейных уравнений. Массив pParallelPivotPos является глобальным, и любое его изменение в одном из процессов требует выполнения операции рассылки измененных данных всем остальным процессам программы.

Элементы массива pProcPivotIter определяют номера итераций прямого хода метода Гаусса, на которых строки процесса использовались в качестве ведущих (т.е. строка i процесса выбиралась ведущей на итерации pProcPivotIter[i]). Начальное значение элементов массива устанавливается нулевым, и, тем самым, нулевое значение элемента массива pProcPivotIter[i] является признаком того, что строка i процесса все еще подлежит обработке. Кроме того, важно отметить, что запоминаемые в элементах массива pProcPivotIter номера итераций дополнительно означают и номера неизвестных, для определения которых будут использованы соответствующие строки уравнения. Массив pProcPivotIter является локальным для каждого процесса.

Функция ProcessInitialization определяет исходные данные решаемой задачи (число неизвестных), выделяет память для хранения данных, осуществляет ввод матрицы коэффициентов системы линейных уравнений и вектора правых частей (или формирует эти данные при помощи какого-либо датчика случайных чисел).

Функция DataDistribution реализует распределение матрицы линейной системы и вектора правых частей между процессорами вычислительной системы.

Функция ResultCollection осуществляет сбор со всех процессов отдельных частей вектора неизвестных.

Функция ProcessTermination выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

Функция ParallelResultCalculation. Реализует логику работы параллельного алгоритма Гаусса, последовательно вызывает функции, выполняющие прямой и обратный ход метода.

```
// Функция для параллельного выполнения метода Гаусса
void ParallelResultCalculation (double* pProcRows,
double* pProcVector, double* pProcResult, int Size, int RowNum) {
  ParallelGaussianElimination (pProcRows, pProcVector, Size,
  RowNum);
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult,
  Size, RowNum);
}
```

Функция ParallelGaussianElimination. Функция выполняет параллельный вариант прямого хода алгоритма Гаусса.

```
// Функция для параллельного выполнения прямого хода метода Гаусса
void ParallelGaussianElimination (double* pProcRows,
  double* pProcVector, int Size, int RowNum) {
  double MaxValue;     // Значение ведущего элемента на процессоре
  int    PivotPos;     // Положение ведущей строки в полосе линейной
                       // системы даннного процессора
  // Структура для выбора ведущего элемента
  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;

  // pPivotRow используется для хранения ведущей строки матрицы и
  // соответствующего элемента вектора b
  double* pPivotRow = new double [Size+1];
  // Итерации прямого хода метода Гаусса
  for (int i=0; i<Size; i++) {

    // Нахождение ведущей строки среди строк процесса
    double MaxValue = 0;
    for (int j=0; j<RowNum; j++) {
      if ((pProcPivotIter[j] == -1) &&
          (MaxValue < fabs(pProcRows[j*Size+i]))) {
        MaxValue = fabs(pProcRows[j*Size+i]);
        PivotPos = j;
      }
    }
    ProcPivot.MaxValue = MaxValue;
    ProcPivot.ProcRank = ProcRank;

    // Нахождение ведущего процесса (процесса, который содержит
    // максимальное значение переменной MaxValue)
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT,
MPI_MAXLOC, MPI_COMM_WORLD);

    // Рассылка ведущей строки
    if ( ProcRank == Pivot.ProcRank ){
      pProcPivotIter[PivotPos]= i; // номер итерации
      pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos;
    }
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
      MPI_COMM_WORLD);
```

```
    if ( ProcRank == Pivot.ProcRank ){
      // заполнение ведущей строки
      for (int j=0; j<Size; j++) {
        pPivotRow[j] = pProcRows[PivotPos*Size + j];
      }
      pPivotRow[Size] = pProcVector[PivotPos];
    }
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
      MPI_COMM_WORLD);
  ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow,
  Size, RowNum, i);
  }
}
```

Функция ParallelEliminateColumns проводит вычитание ведущей строки из строк процесса, которые еще не использовались в качестве ведущих (т.е. для которых элементы массива pProcPivotIter равны нулю).

Функция ParallelBackSubstitution. Функция реализует параллельный вариант обратного хода Гаусса.

```
// Функция для параллельного выполнения обратного хода метода Гаусса
void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  int IterProcRank;    // Ранг процессора, который содержит ведущую строку
  int IterPivotPos;    // Положение ведущей строки в полосе процессора
  double IterResult;   // Вычисленное значение очередной неизвестной
  double val;

  // Итерации обратного хода метода Гаусса
  for (int i=Size-1; i>=0; i--) {

    // Вычисление ранга процессора, который содержит ведущую строку
    FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank,
      IterPivotPos);

    // Вычисление значения неизвестной
    if (ProcRank == IterProcRank) {
      IterResult = pProcVector[IterPivotPos]/pProcRows[IterPivotPos*Size+i];
      pProcResult[IterPivotPos] = IterResult;
    }
    // Рассылка значения очередной неизвестной
    MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);

    // Обновление вектора правых частей
    for (int j=0; j<RowNum; j++)
      if ( pProcPivotIter[j] < i ) {
        val = pProcRows[j*Size + i] * IterResult;
        pProcVector[j]=pProcVector[j] - val;
    }
  }
}
```

Функция FindBackPivotRow определяет строку, из которой можно вычислить значение очередного элемента результирующего вектора. Номер этой строки хранится в массиве pParallelPivotIter. По номеру функция FindBackPivotRow определяет номер процесса, на котором эта строка хранится, и номер этой строки в полосе pProcRows этого процесса.

2. Индивидуальные задания

Решить СЛАУ заданным методом согласно варианта (таблица 8.1) средствами MPI. Результаты вычислений сравнить с однопоточным приложением и с многопоточными приложениями, реализованными через Win API, .Net, POSIX Linux.

Распараллеливание реализовать на уровне алгоритма.

Программа должна обеспечивать решение одной заданной СЛАУ, при этом в файле A.txt должна содержаться исходная матрица системы, в файле B.txt – столбец свободных членов. В результате решения должен быть сформирован файл X.txt, содержащий вектор решения. Программа должна обеспечивать решение СЛАУ порядка до 10000 неизвестных.

Вариант	Метод решения СЛАУ	Способ хранения матрицы системы
1	LU – разложение: L - нижнетреугольная матрица, U - верхнетреугольная матрица с единицами на главной диагонали	Полнопрофильное размещение
2	U^TU – разложение: U – верхнетреугольная с полнопрофильным размещением	Полнопрофильное размещение
3	L*U - разложение: L* - нижнетреугольная матрица с единицами на главной диагонали, U – верхнетреугольная матрица	Полнопрофильное размещение
4	LL^T - разложение: L - нижнетреугольная матрица	Полнопрофильное размещение
5	LD^*L^T - разложение: L - нижнетреугольная матрица, D* - диагональная матрица, причем dii = ±1	Полнопрофильное размещение
6	L^*DL^{*T} - разложение: L* - нижнетреугольная матрица с 1 на диагонали, D - диагональная матрица	Полнопрофильное размещение
7	L*DU* - разложение: L* - нижнетреугольная матрица с 1 на диагонали, D - диа- гональная матрица, U* - верхнетреугольная матрица с 1 на диагонали	Полнопрофильное размещение
8	LU(sq) - разложение: L - нижнетреугольная матрица, U - верхнетреугольная матрица и Vi: $l_{ii} = u_{ii}$	Полнопрофильное размещение
9	Метод Гаусса с размещением по строкам	Полнопрофильное размещение
10	Метод Гаусса с циклическим размещением по строкам	Полнопрофильное размещение
11	Метод Гаусса с размещением по подматрицам	Полнопрофильное размещение
12	Метод Гаусса с размещением по столбцам	Полнопрофильное размещение
13	Метод Гаусса с циклическим размещением по столбцам	Полнопрофильное размещение
14	Метод Гаусса с циклическим размещением по подматрицам	Полнопрофильное размещение
15	Метод сопряжённых градиентов	Полнопрофильное размещение
16	LU - разложение: L - нижнетреугольная матрица, U - верхнетреугольная матрица с единицами на главной диагонали	Диагональное размещение
17	U^TU - разложение: U - верхнетреугольная	Диагональное размещение
18	L*U - разложение: L* - нижнетреугольная матрица с единицами на главной диагонали, U - верхнетреугольная матрица	Диагональное размещение
19	LL^T - разложение: L - нижнетреугольная матрица	Диагональное размещение
20	LD^*L^T - разложение: L - нижнетреугольная матрица, D* - диагональная матрица, причем dii = ±1	Диагональное размещение
21	L^*DL^{*T} - разложение: L* - нижнетреугольная матрица с 1 на диагонали, D - диагональная матрица	Диагональное размещение
22	L*DU* - разложение: L* - нижнетреугольная матрица с 1 на диагонали, D - диа- гональная матрица, U* - верхнетреугольная матрица с 1 на диагонали	Диагональное размещение
23	LU(sq) - разложение: L - нижнетреугольная матрица, U - верхнетреугольная матрица и Vi: $l_{ii} = u_{ii}$	Диагональное размещение
24	Метод Гаусса с размещением по строкам	Диагональное размещение
25	Метод Гаусса с циклическим размещением по строкам	Диагональное размещение
26	Метод Гаусса с размещением по подматрицам	Диагональное размещение

27	Метод Гаусса с размещением по столбцам	Диагональное размещение
28	Метод Гаусса с циклическим размещением по столбцам	Диагональное размещение
29	Метод Гаусса с циклическим размещением по подматрицам	Диагональное размещение
30	Метод сопряжённых градиентов	Диагональное размещение

Вопросы к защите

1. Распределённые системы обработки информации
2. ГРИД-системы
3. Кластерные системы
4. Построение кластера на базе MPI
5. Запуск и управление задания средствами MPI MPICH
6. Архитектура программ с использование MPI
7. Настройка компилятора для создания кластерных приложений
8. Основные коммуникационные функции MPI MPICH
9. Блокирующий и неблокирующий обмен
10. Передача значений переменных. Широковещательная рассылка
11. Логическое окружение в MPI