ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СУПЕРКОМПЬЮТЕРА “МВС-1000/50Н”

Руководство программиста

Листов 44

Примечание [СВК1]: ??

2001

АННОТАЦИЯ

Документ содержит сведения, необходимые для программистов, работающих на СК “МВС-1000/50Н” и пользующихся услугами компонентов программного обеспечения. Он состоит из пяти разделов.

В разделе 1 представлены сведения о назначении, функциях и условиях применения среды параллельного программирования, а также всего ОПО СК “МВС-1000/50Н” в целом.

В разделе 2 приведены основные характеристики и общие принципы функционирования среды параллельного программирования MPI.

В разделе 3 содержит описание процедур MPI.

В разделе 4 приведены сведения о подготовке прикладных программ к выполнению на вычислительных блоках СК “МВС-1000/50Н”.

В разделе 5 представлена информация о способах отладки программных модулей.

СОДЕРЖАНИЕ

1 Назначение и условия применения среды параллельного программирования СК “МВС-1000/50Н”

1.1 Назначение и состав среды параллельного программирования

Компоненты общего программного обеспечения (ОПО) СК “МВС-1000/50Н” поддерживают все этапы разработки параллельных программ пользователей, а также обеспечивают непосредственно выполнение процессов содержательной обработки на решающем поле вычислительных узлов. Они функционируют на вычислительных узлах (ВУ) и управляющем узле (УУ).

В качестве среды для параллельного программирования используется программный интерфейс MPI, который фактически является стандартом для разработчиков параллельных программ. Кластеризация мощных вычислительных ресурсов повлекла за собой такое требование к среде параллельного программирования, как умение функционировать в неоднородных (гетерогенных) сетях.

Среда параллельного программирования реализована на базе интерфейса передачи сообщений MPI (Message Passing Interface) и включает в себя пакет MPICH for GM версии 1.2..8.

Информацию об особенностях работы с программным пакетом MPICH можно найти в руководстве пользователя по MPICH [2].

MPI - это программный инструментарий для обеспечения связи между ветвями параллельного приложения. Он предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные / многопроцессорные с общей/раздельной памятью), взаимного расположения ветвей (на одном процессоре / на разных) и программного интерфейса пользователя операционной системы.

Программа, использующая MPI, легче отлаживается (сужается простор для совершения стереотипных ошибок параллельного программирования) и быстрее переносится на другие платформы (в идеале, простой перекомпиляцией).

Стандарты MPI 1.2 и MPI 2.0 доступны на сайте http://www.mpi-forum.org в формате HTML. Более подробную информацию о работе с MPI можно найти в книгах, посвященных MPI [4, 5].

Синтаксис MPI облегчает создание приложений в модели SPMD (single program multiple data) - одна программа работает в разных процессорах со своими данными. Одна и та же функция сможет вызываться на узле-источнике и узлах-приемниках, а тип выполняемой операции (передача или прием) в этом случае определяется с помощью параметра. Такой синтаксис вызовов делает SPMD-программы существенно компактнее, хотя и труднее для понимания.

Основное отличие стандарта MPI от его предшественников - понятие коммуникатора. Все операции синхронизации и передачи сообщений локализуются внутри коммуникатора. С коммуникатором связывается группа процессов. В частности, все коллективные операции вызываются

одновременно на всех процессах, входящих в эту группу. Поскольку взаимодействие между процессами инкапсулируется внутри коммуникатора, на базе MPI можно создавать библиотеки параллельных программ.

1.2 Условия эксплуатации

Для работы компонентов программного обеспечения СК “МВС-1000/50Н” необходимы:

- управляющая рабочая станция Alpha 21264 на основе материнской платы UP2000;
- двухпроцессорные вычислительные модули на базе процессора Alpha21264;
- коммуникационная среда Myrinet, состоящая из сетевых плат, устанавливаемых в ВУ, и коммутатора;
- коммуникационная среда FastEthernet, состоящая из сетевых плат, устанавливаемых в ВУ и УУ, и коммутатора.

В состав ОПО СК “МВС-1000/50Н” входят:

- операционные системы ВУ и управляющего узла;
- инструментальные программные средства;
- программные средства высокопроизводительной коммуникационной среды Myrinet;
- программные средства коммуникационной среды FastEthernet;
- операционная среда параллельного программирования;
- подсистема удаленного доступа.

На управляющем узле СК “МВС-1000/50Н” и ВУ установлена операционная система Linux версии 7.1.

В состав инструментальных программных средств входят:

- компиляторы GNU, поставляемые в составе ОС Linux: C, C++ и Fortran77 (команды gcc, c++ и f77 соответственно);
- компиляторы фирмы Compaq для ОС Linux на платформе Alpha: C, C++, Fortran (команды ccc ,cxx и fort соответственно);
- редактор связей;
- отладчики (gdb,dbx,xxgdb);

Коммуникационная среда Myrinet поддерживается в современных реализациях интерфейса параллельного программирования MPI. В качестве программных средств коммуникационной среды Myrinet используется коммуникационная система GM. В ее состав входят:

- драйвер;
- служебные программы;
- тестовые программы;
- библиотека функций и заголовочный файл GM API;
- демонстрационные программы.

1.3 Терминология и обозначения

При работе с интерфейсом MPI используются следующие термины и обозначения:

а) номер процесса – целое неотрицательное число, являющееся уникальным атрибутом каждого процесса;

б) атрибуты сообщения – номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения; для них заведена структура `MPI_Status`, содержащая три поля: `MPI_Source` (номер процесса отправителя), `MPI_Tag` (идентификатор сообщения), `MPI_Error` (код ошибки); могут быть и добавочные поля;

в) идентификатор сообщения (`msgtag`) – атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767.

2 Характеристики среды параллельного программирования

2.1 Особенности программного интерфейса MPI

MPI расшифровывается как Message Passing Interface - Интерфейс с передачей сообщений, т.е. конкретному стандарту присвоено название всего представляемого им класса программного инструментария. В пакет, реализующий программный интерфейс на СК “МВС-1000/50Н”, входят два обязательных компонента:

- библиотека программирования для языков Си, Си++ и Фортран;
- загрузчик исполняемых файлов.

Характеристиками программного интерфейса MPI являются:

- обеспечение связи между ветвями параллельной программы;
- программирование по методу SPMD (Single Program – Multiple Data) с передачей сообщений;
- реализация в виде библиотеки для языков программирования С и Фортран и загрузчика приложений;
- поддержка гетерогенных вычислений;
- мобильность интерфейса и, как следствие, мобильность создаваемых программ.

2.2 Общие принципы функционирования среды параллельного програм-мирования, используемой на СК “МВС-1000/50Н”

- параллельная программа содержит код для всех ветвей сразу;
- загрузчиком запускается указываемое количество экземпляров программы;
- каждый экземпляр программы определяет свой порядковый номер, и, в зависимости от этого номера и общего размера вычислительного поля, выполняет ту или иную ветвь алгоритма;
- каждая ветвь имеет собственное пространство данных, полностью изолированное от других ветвей;
- ветви обмениваются данными только с помощью передачи сообщений операционной среды параллельного программирования.

Если MPI-приложение запускается в сети, запускаемый файл приложения должен быть доступен на каждом ВМ по тому же абсолютному пути, что и на управляющей рабочей станции.

2.3 Профилировочные библиотеки

Профилировочный интерфейс MPE (Message Passing Extensions) представляет собой инструмент для добавления процедур анализа

производительности в любую MPI-программу. С пакетом MPICH поставляется две профилировочные библиотеки.

2.3.1 Библиотека для накопления времени, затраченного в процедурах MPI

Первая библиотека достаточно проста. Профилировочная версия каждой процедуры MPI (`MPI_Xxx`) вызывает функцию `PMPI_Wtime` (возвращающую текущее время) перед и после каждого вызова соответствующей `PMPI_Xxx` процедуры. Времена накапливаются для каждого процесса и выводятся в файл (отдельный файл для каждого процесса) в профилировочной версии `MPI_Finalize`. В дальнейшем эти файлы можно использовать для создания отчета по всему приложению или по отдельным процессам. Текущая реализация библиотеки не обрабатывает вложенные циклы.

2.3.2 Создание файла журнала и утилита UpShot

Вторая профилировочная библиотека предназначена для генерации файла журнала (`log`-файла), в которых фиксируются привязанные ко времени события.

Для сохранения определенных типов событий в памяти, в процессе выполнения вызывается процедура `MPI_Log_event`, а сборка и объединение этих частей памяти с информацией о событиях происходит в `MPI_Finalize`. Для остановки и перезапуска операций записи событий во время выполнения программы может использоваться процедура `MPI_Pcontrol`.

Анализ файла журнала может быть осуществлен при помощи разнообразных программных средств. Используемый для этой цели в MPICH инструмент, называется UpShot. Состояния процессов в UpShot показаны с помощью параллельных осей времени для каждого процесса. Окно внизу экрана показывает гистограмму продолжительностей процессов с несколькими корректируемыми параметрами.

2.3.3 Анимация процесса работы программы в реальном времени

Графическая библиотека MPE предоставляет возможности для простой анимации в реальном времени. Библиотека содержит процедуры, которые позволяют разделять X-дисплей нескольким процессам. На основе данной библиотеки существует возможность графически изображать процесс передачи сообщений и их интенсивность в процессе работы программы.

Для сборки программы с использованием графических библиотек MPE при компиляции можно указать опцию `-lmpe`.

В MPICH предусмотрены также опции для компиляции и сборки программ с различными профилировочными библиотеками MPE:

`-mpitrace`
: Для компиляции и сборки с отладочными библиотеками.

`-mpianim`
: Для компиляции и сборки с анимационными библиотеками.

`-mpilog`

Для компиляции и сборки с регистрирующими библиотеками .

Пример.

```
mpif77 -mpilog -o fpilog fpilog.f
```

Более подробную информацию об использовании профилировочных библиотек и о синтаксисе процедур MPE можно найти в документации по MPE [3, 1] а также в страницах справочного руководства (man pages).

3 Обращение к MPI

3.1 Связь между ветвями параллельной программы “точка-точка”

Базовым механизмом связи между процессами в MPI является посылка и прием сообщений. Основными функциями, обеспечивающими связь “точка-точка” являются операторы `send` и `receive`.

3.2 Коммуникаторы

Процессы объединяются в группы; могут быть вложенные группы. Внутри группы все процессы пронумерованы. С каждой группой ассоциирован свой коммуникатор. Поэтому при осуществлении пересылки необходимо указать идентификатор группы, внутри которой производится эта пересылка. Все процессы содержатся в группе с предопределенным идентификатором MPI_COMM_WORLD.

Две информационных функции сообщают размер группы (то есть общее количество задач, подсоединенных к ее области связи) и порядковый номер вызывающей задачи:

```
1. int size, rank;
2. MPI_Comm_size( MPI_COMM_WORLD, &size );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank ).
```

3.3 Прием и посылка сообщений. Процедуры MPI

Процедура `MPI_Send`

Формат процедуры:

`int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm),` где

- `buf` - адрес начала буфера посылки сообщения;
- `count` - число передаваемых элементов в сообщении;
- `datatype` - тип передаваемых элементов;
- `dest` - номер процесса-получателя;
- `msgtag` - идентификатор сообщения;
- `comm` - идентификатор группы.

Данная процедура осуществляет блокирующую посылку сообщения с идентификатором `msgtag`, состоящего из `count` элементов типа `datatype`, процессу с номером `dest`. Все элементы сообщения расположены подряд в буфере `buf`. Значение `count` может быть нулем. Тип передаваемых элементов `datatype` должен указываться с помощью предопределенных констант типа.

Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу `dest`, остается за MPI. Следует специально отметить, что возврат из подпрограммы `MPI_Send` не означает ни того, что сообщение уже передано процессу `dest`, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший `MPI_Send`.

Процедура `MPI_Recv`

Формат процедуры:

`int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_comm comm, MPI_Status *status)`, где

- `OUT buf` - адрес начала буфера приема сообщения;
- `count` - максимальное число элементов в принимаемом сообщении;
- `datatype` - тип элементов принимаемого сообщения;
- `source` - номер процесса-отправителя;
- `msgtag` - идентификатор принимаемого сообщения;
- `comm` - идентификатор группы;
- `OUT status` - параметры принятого сообщения.

Процедура осуществляет прием сообщения с идентификатором `msgtag` от процесса `source` с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения `count`. Если число принятых элементов меньше значения `count`, то гарантируется, что в буфере `buf` изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой `MPI_Probe`.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере `buf`.

В качестве номера процесса-отправителя можно указать предопределенную константу `MPI_ANY_SOURCE` - признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу `MPI_ANY_TAG` - признак того, что подходит сообщение с любым идентификатором.

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову `MPI_Recv`, то первым будет принято то сообщение, которое было отправлено раньше.

Процедура `MPI_Get_Count`

Формат процедуры:

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

- `status` - параметры принятого сообщения;
- `datatype` - тип элементов принятого сообщения;
- `OUT count` - число элементов сообщения.

По значению параметра `status` данная процедура определяет число уже принятых (после обращения к `MPI_Recv`) или принимаемых (после обращения к `MPI_Probe` или `MPI_IProbe`) элементов сообщения типа datatype.

Процедура `MPI_Probe`

Формат процедуры:

`int MPI_Probe( int source, int msgtag, MPI_Comm comm, MPI_Status *status)`, где

- `source` - номер процесса-отправителя или `MPI_ANY_SOURCE;`
- `msgtag` - идентификатор ожидаемого сообщения или `MPI_ANY_TAG;`
- `comm` - идентификатор группы;
- `OUT status` - параметры обнаруженного сообщения.

Данная процедура обеспечивает получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра `status`. Следует обратить внимание, что подпрограмма определяет только факт прихода сообщения, но реально его не принимает.

Процедура `MPI_Isend`

Формат процедуры:

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

- `buf` - адрес начала буфера посылки сообщения;
- `count` - число передаваемых элементов в сообщении;
- `datatype` - тип передаваемых элементов;
- `dest` - номер процесса-получателя;
- `msgtag` - идентификатор сообщения;
- `comm` - идентификатор группы;
- `OUT request` - идентификатор асинхронной передачи.

Процедура осуществляет передачу сообщения, аналогичную `MPI_Send`, однако возврат из процедуры происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере `buf`. Это означает, что нельзя повторно использовать

данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. того момента, когда можно переиспользовать буфер `buf` без опасения испортить передаваемое сообщение) можно определить с помощью параметра `request` и процедур `MPI_Wait` и `MPI_Test`.

Сообщение, отправленное любой из процедур `MPI_Send` и `MPI_ISend`, может быть принято любой из процедур `MPI_Recv` и `MPI_IRecv`.

Процедура `MPI_Irecv`

Формат процедуры:

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

- `OUT buf` - адрес начала буфера приема сообщения;
- `count` - максимальное число элементов в принимаемом сообщении;
- `datatype` - тип элементов принимаемого сообщения;
- `source` - номер процесса-отправителя;
- `msgtag` - идентификатор принимаемого сообщения;
- `comm` - идентификатор группы;
- `OUT request` - идентификатор асинхронного приема сообщения.

Данная процедура осуществляет прием сообщения, аналогичный `MPI_Recv`, однако возврат из процедуры происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере `buf`. Окончание процесса приема можно определить с помощью параметра `request` и процедур `MPI_Wait` и `MPI_Test`.

Процедура `MPI_Wait`

Формат процедуры:

`int MPI_Wait(MPI_Request *request, MPI_Status *status)`, где

- `request` - идентификатор асинхронного приема или передачи;
- `OUT status` - параметры сообщения.

Процедура осуществляет ожидание завершения асинхронных процедур `MPI_ISend` или `MPI_IRecv`, ассоциированных с идентификатором `request`. В случае приема, атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра `status`.

Процедура `MPI_WaitAll`

Формат процедуры:

`int MPI_WaitAll(int count, MPI_Request *requests, MPI_Status *statuses)`, где

- `count` - число идентификаторов;

– `requests` - массив идентификаторов асинхронного приема или передачи;
– `OUT statuses` - параметры сообщений.

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива `statuses` будет установлено в соответствующее значение.

Процедура `MPI_WaitAny`

Формат процедуры:

`int MPI_WaitAny(int count, MPI_Request *requests, int *index, MPI_Status *status)`, где

– `count` - число идентификаторов;
– `requests` - массив идентификаторов асинхронного приема или передачи;
– `OUT index` - номер завершенной операции обмена;
– `OUT status` - параметры сообщений.

Выполнение процесса блокируется до тех пор, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если несколько операций могут быть завершены, то случайным образом выбирается одна из них. Параметр `index` содержит номер элемента в массиве `requests`, содержащего идентификатор завершенной операции.

Процедура `MPI_WaitSome`

Формат процедуры:

`int MPI_WaitSome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)`, где

– `incount` - число идентификаторов;
– `requests` - массив идентификаторов асинхронного приема или передачи;
– `OUT outcount` - число идентификаторов завершившихся операций обмена;
– `OUT indexes` - массив номеров завершившихся операции обмена;
– `OUT statuses` - параметры завершившихся сообщений.

Выполнение процесса блокируется до тех пор, пока по крайней мере одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр `outcount` содержит число завершенных операций, а первые `outcount` элементов массива `indexes` содержат номера элементов массива `requests` с их идентификаторами.

Первые outcount элементов массива statuses содержат параметры завершенных операций.

Процедура MPI_Test

Формат процедуры:

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

- request - идентификатор асинхронного приема или передачи;
- OUT flag - признак завершенности операции обмена;
- OUT status - параметры сообщения.

Данная процедура осуществляет проверку завершенности асинхронных процедур MPI_ISend или MPI_IRecv, ассоциированных с идентификатором request. В параметре flag возвращает значение 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра status.

Процедура MPI_TestAll

Формат процедуры:

int MPI_TestAll(int count, MPI_Request *requests, int *flag, MPI_STatus *statuses), где

- count - число идентификаторов;
- requests - массив идентификаторов асинхронного приема или передачи;
- OUT flag - признак завершенности операций обмена;
- OUT statuses - параметры сообщений.

В результате выполнения данной процедуры, параметре flag возвращается значение 1, если все операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве statuses). В противном случае возвращается 0, а элементы массива statuses не определены.

Процедура MPI_TestAny

Формат процедуры:

int MPI_TestAny(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status), где

- count - число идентификаторов;
- requests - массив идентификаторов асинхронного приема или передачи;
- OUT index - номер завершенной операции обмена;
- OUT flag - признак завершенности операции обмена;
- OUT status - параметры сообщения.

Если к моменту вызова данной процедуры хотя бы одна из операций обмена завершилась, то в параметре `flag` возвращается значение 1, `index` содержит номер соответствующего элемента в массиве `requests`, а `status` - параметры сообщения.

Процедура `MPI_TestSome`

Формат процедуры:

`int MPI_TestSome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)`, где

- `incount` - число идентификаторов;
- `requests` - массив идентификаторов асинхронного приема или передачи;
- `OUT outcount` - число идентификаторов завершившихся операций обмена;
- `OUT indexes` - массив номеров завершившихся операции обмена;
- `OUT statuses` - параметры завершившихся операций.

Данная процедура работает так же, как и `MPI_TestAny`, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение `outcount` будет равно нулю.

Процедура `MPI_Iprobe`

Формат процедуры:

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

- `source` - номер процесса-отправителя или `MPI_ANY_SOURCE`;
- `msgtag` - идентификатор ожидаемого сообщения или `MPI_ANY_TAG`;
- `comm` - идентификатор группы;
- `OUT flag` - признак завершенности операции обмена;
- `OUT status` - параметры обнаруженного сообщения.

Данная процедура обеспечивает получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре `flag` возвращает значение 1, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично `MPI_Probe`), и значение 0, если сообщения с указанными атрибутами еще нет.

Для снижения накладных расходов, возникающих в рамках одного процессора при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером, могут использоваться процедуры: `MPI_Send_Init`, `MPI_Recv_Init` и MPI_Start_All. Несколько запросов на прием и/или передачу могут объеди-

няться вместе для того, чтобы далее их можно было бы запустить одной командой. Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

Процедура `MPI_Send_Init`

Формат процедуры:

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

- `buf` - адрес начала буфера посылки сообщения;
- `count` - число передаваемых элементов в сообщении;
- `datatype` - тип передаваемых элементов;
- `dest` - номер процесса-получателя;
- `msgtag` - идентификатор сообщения;
- `comm` - идентификатор группы;
- `OUT request` - идентификатор асинхронной передачи.

Данная процедура обеспечивает формирование запроса на выполнение пересылки данных. Все параметры точно такие же, как и у подпрограммы `MPI_ISend`, однако в отличие от нее пересылка не начинается до вызова подпрограммы `MPI_StartAll`.

Процедура `MPI_Recv_Init`

Формат процедуры:

`int MPI_Recv_Init(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)`, где

- `OUT buf` - адрес начала буфера приема сообщения;
- `count` - число принимаемых элементов в сообщении;
- `datatype` - тип принимаемых элементов;
- `source` - номер процесса-отправителя;
- `msgtag` - идентификатор сообщения;
- `comm` - идентификатор группы;
- `OUT request` - идентификатор асинхронного приема.

Данная процедура обеспечивает формирование запроса на выполнение приема данных. Все параметры точно такие же, как и у подпрограммы `MPI_IReceive`, однако в отличие от нее реальный прием не начинается до вызова подпрограммы `MPI_StartAll`.

Процедура `MPI_Start_All`

Формат процедуры:

`int MPI_Start_All (int count, MPI_Request *requests)`, где

- `count` - число запросов на взаимодействие;
- `OUT requests` - массив идентификаторов приема/передачи.

Данная процедура обеспечивает запуск всех отложенных взаимодействий, ассоциированных вызовами подпрограмм `MPI_Send_Init` и `MPI_Recv_Init` с элементами массива запросов `requests`. Все взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью процедур `MPI_Wait` и `MPI_Test`.

Процедура `MPI_Sendrecv`

Формат процедуры:

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

- `sbuf` - адрес начала буфера посылки сообщения;
- `scount` - число передаваемых элементов в сообщении;
- `stype` - тип передаваемых элементов;
- `dest` - номер процесса-получателя;
- `stag` - идентификатор посылаемого сообщения;
- `OUT rbuf` - адрес начала буфера приема сообщения;
- `rcount` - число принимаемых элементов сообщения;
- `rtype` - тип принимаемых элементов;
- `source` - номер процесса-отправителя;
- `rtag` - идентификатор принимаемого сообщения;
- `comm` - идентификатор группы;
- `OUT status` - параметры принятого сообщения.

Данная процедура объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией `MPI_Sendrecv`, может быть принято обычным образом, и точно также операция `MPI_Sendrecv` может принять сообщение, отправленное обычной операцией `MPI_Send`. Буфера приема и посылки обязательно должны быть различными.

3.4 Коллективные функции

Для коллективного взаимодействия процессов используются процедуры `MPI_Bcast`, `MPI_Gather`, `MPI_AllReduce` и `MPI_Reduce`. В процессе коллективного взаимодействия участвуют все процессы приложения. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры

коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

3.4.1 Обмен данными

Процедура `MPI_Bcast`

Формат процедуры:

`int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)`, где

- `OUT buf` - адрес начала буфера посылки сообщения;
- `count` - число передаваемых элементов в сообщении;
- `datatype` - тип передаваемых элементов;
- `source` - номер рассылающего процесса;
- `comm` - идентификатор группы.

Процедура обеспечивает рассылку сообщения от процесса `source` всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера `buf` процесса `source` будет скопировано в локальный буфер процесса. Значения параметров `count`, `datatype` и `source` должны быть одинаковыми у всех процессов.

Процедура `MPI_Gather`.

Формат процедуры:

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

- `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;
- `OUT ierror` - код ошибки.

Данная процедура обеспечивает сбор данных со всех процессов в буфере `rbuf` процесса `dest`. Каждый процесс, включая `dest`, посылает содержимое своего буфера `sbuf` процессу `dest`. Собирающий процесс сохраняет данные в буфере `rbuf`, располагая их в порядке возрастания номеров процессов. Параметр `rbuf` имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров `count`, `datatype` и `dest` должны быть одинаковыми у всех процессов.

3.4.2 Распределенные вычисления

Процедура `MPI_AllReduce`

Формат процедуры:

`int MPI_AllReduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`, где

- `sbuf` - адрес начала буфера для аргументов;
- `OUT rbuf` - адрес начала буфера для результата;
- `count` - число аргументов у каждого процесса;
- `datatype` - тип аргументов;
- `op` - идентификатор глобальной операции;
- `comm` - идентификатор группы.

Процедура обеспечивает выполнение `count` глобальных операций `op` с возвратом `count` результатов во всех процессах в буфере `rbuf`. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров `count` и `datatype` у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция `op` обладает свойствами ассоциативности и коммутативности.

Процедура `MPI_Reduce`.

Формат процедуры:

`int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`, где

- `sbuf` - адрес начала буфера для аргументов;
- `OUT rbuf` - адрес начала буфера для результата;
- `count` - число аргументов у каждого процесса;
- `datatype` - тип аргументов;
- `op` - идентификатор глобальной операции;
- `root` - процесс-получатель результата;
- `comm` - идентификатор группы.

Процедура аналогична предыдущей, но результат будет записан в буфер `rbuf` только у процесса `root`.

3.4.3 Точки синхронизации

Процедура синхронизации процессов блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы `comm` также не выполнят эту процедуру.

Процедура `MPI_Barrier`

Формат процедуры:

`int MPI_Barrier(MPI_Comm comm)`, где `comm` - идентификатор группы.

3.4.4 Типы данных

В стандарте MPI существует несколько предопределенных типов, среди них:

– `MPI_Status` - структура; атрибуты сообщений; содержит три обязательных поля:
  - `MPI_Source` (номер процесса отправителя);
  - `MPI_Tag` (идентификатор сообщения);
  - `MPI_Error` (код ошибки);
– `MPI_Request` - системный тип; идентификатор операции посылки-приема сообщения;
– `MPI_Comm` - системный тип; идентификатор группы (коммуникатора);
– `MPI_COMM_WORLD` - зарезервированный идентификатор группы, состоящей их всех процессов приложения.

Таблица 1

Константы MPI	Тип в C
MPI_CHAR	signed char
MPI_SHORT	signed int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Более подробно о правилах написания параллельных программ, а также синтаксис процедур MPI можно найти в книге “Writing Message-Passing Parallel Programs with MPI” [6].

3.5 Латентность и пропускная способность

3.5.1 Понятия латентности и пропускной способности

Латентность - это время между инициированием передачи данных в процессе посылки и прибытия первого байта в процессе приема. Латентность часто зависит от длины посылаемых сообщений. Ее значение может изменяться в зависимости от того, послано ли большое количество маленьких сообщений или нескольких больших сообщений.

Пропускная способность - это величина, обратная времени, необходимого для передачи одного байта. Пропускная способность обычно выражается в мегабайтах в секунду. Пропускная способность важна, когда размеры передаются большие сообщения.

Для улучшения характеристик латентности и пропускной способности необходимо:

- подсчитать кол-во каналов передачи данных между процессами при разработке крупномодульных приложений;
- использовать архивацию данных для больших сообщений, а также использовать описываемые типы данных вместо `MPI_PACK` и `MPI_UNPACK` если возможно;
- использовать при возможности коллективные операции; это устраняет вызов `MPI_Send` и `MPI_RECV` каждый раз при коммуникации процессов;
- определять номер принимающего процесса при вызове подпрограммы MPI; использование `MPI_ANY_SOURCE` может увеличивать латентность;
- использовать `MPI_RECV_INIT` и `MPI_STARTALL` вместо вызова `MPI_Irecv` в цикле в случаях, когда запросы/прием не могут быть выполнены сразу.

Например, вы написали программу, содержащую фрагмент:

```
j = 0
for (i=0; i<size; i++) {
if (i==rank) continue;
MPI_Irecv(buf[i], count, dtype, i, 0, comm,
&requests[j++]);
}
MPI_Waitall(size-1, requests, statuses);
```

Предположим, что одна из итераций с вызовом `MPI_IRECV` не завершилась перед следующей итерацией цикла. В этом случае, MPI пробует выполнить оба запроса. Это может продолжаться, приводя к большему времени ожидания. Чтобы избежать этого, можно переписать эту часть кода так:

```
j = 0
for (i=0; i<size; i++) {
if (i==rank) continue;
MPI_Recv_init(buf[i], count, dtype, i, 0, comm,
&requests[j++]);
}
MPI_Startall(size-1, requests);
MPI_Waitall(size-1, requests, statuses);
```

В этом случае все итерации с вызовом `MPI_RECV_INIT` выполняются только один раз при вызове `MPI_STARTALL`. При таком подходе вы не получите дополнительного времени ожидания при использовании `MPI_Irecv` и может улучшить латентность приложения.

3.5.2 Выбор подпрограмм/функций MPI

Для достижения наименьшей латентности и наибольшей пропускной способности сообщений для синхронной передачи "точка-точка", используйте блокирующие функции MPI `MPI_Send` и `MPI_RECV`. Для асинхронной передачи, используйте неблокирующие функции MPI `MPI_Isend` и `MPI_IRECV`.

При использовании блокирующих функций, старайтесь избегать ожидающих запросов.

Для задач требующих использования коллективных операций, используют соответствующую коллективную функцию MPI.

4 Входные и выходные данные

4.1 Этапы создания программ

Создание параллельной программы состоит из следующих этапов:

- последовательный алгоритм подвергается декомпозиции (распараллеливанию), т.е. разбивается на независимо работающие ветви; для взаимодействия в ветви вводятся две дополнительные нематематические операции: прием и передача данных;
- распараллеленный алгоритм записывается в виде программы, в которой операции приема и передачи записываются в терминах конкретной системы связи между ветвями;
- полученная таким образом программа компилируется и компонуется с библиотеками среды параллельного программирования при помощи компилятора, используемого на данной системе для получения машинно-зависимого кода.

4.2 Компиляция модулей

Для компиляции и сборки программного модуля, написанного на C, используется команда `mpicc`. Аналогично для C++ используется команда `mpiCC`, для Fortran 77 используется `mpif77`, а для Fortran 90 – `mpif90`.

Эти команды предусматривают некоторые опции и подключают специальные библиотеки, необходимые для компиляции и сборки программ MPI.

Опция `-c` указывается для выполнения только компиляции файла, не создавая объектный файл. При задании опции `-o` осуществляется сборка и компиляция, а также создается объектный и запускаемый файлы.

Примеры.

1. Компиляция программного модуля `myprog.c`

```
mpicc –c myprog.c
```

2. Сборка и компиляция программного модуля myprog.c с созданием выходного файла `myfile`

```
mpicc myprog.c –o myfile
```

4.3 Запуск программ, использующих MPI

Запуск на исполнение MPI-программы производится с помощью команды:

```
mpirun –np <число_используемых_процессоров>
<имя_модуля> [параметры_mpirun...]  <имя_программы>
параметры_программы...]  [-host <host>]
```

Параметры команды `mpirun` слелующие:

`-h`
интерактивная подсказка по параметрам команды `mpirun`;

`-np <число_процессоров>`
число процессоров, требуемое программе;

`-maxtime <максимальное_время>`
максимальное время счета. От этого времени зависит положение задачи в очереди. После истечения этого времени задача принудительно заканчивается;

`-quantum <значение_кванта_времени>`
этот параметр указывает, что задача является фоновой, и задает размер кванта для фоновой задачи;

`-stdiodir <имя_директории>`

этот параметр задает имя каталога стандартного вывода, в который будут записываться протокол запуска задачи, файл стандартного вывода и имена модулей, на которых запускалась задача.

Более подробное описание параметров команды запуска задач и постановки задачи в очередь приведено в руководстве по СУППЗ.

5 Отладка приложений

Отладка параллельных программ довольно трудна, но в mpich встроено несколько решений, которые могут использоваться при отладке программ MPI.

5.1 Обработчики ошибок

Стандарт MPI определяет механизм для установки пользовательских обработчиков ошибок и определяет поведение двух встроенных обработчиков: MPI_ERRORS_RETURN и MPI_ERRORS_ARE_FATAL. В библиотеку mpe встроено еще два обработчика ошибок для облегчения использования отладчика dbx с программами, написанными с использованием стандарта MPI:

```
MPE_Errors_call_dbx_in_xterm
MPE_Signals_call_debugger
```

Данные обработчики ошибок расположены в директории mpe, основного дерева каталогов MPICH. При конфигурировании MPICH с опцией -mpedbg, эти отладчики включаются в основные библиотеки MPICH, и появляется возможность (с помощью аргумента командной строки mpedbg) установить обработчик MPE_Errors_call_dbx_in_xterm вызываемым по умолчанию обработчиком ошибок (вместо MPI_ERRORS_ARE_FATAL).

5.2 Аргументы командной строки для mpirun

mpirun предоставляет программисту некоторые возможности для облегчения использования отладчика со своей программой.

Команда:

```
mpirun -dbx -np 2 program
```

начинает выполнение программы на двух машинах, запуская локальную копию программы в отладчике dbx. Опция -gdb позволяет использовать в качестве gdb отладчика, а опция -xxgdb запускает программу в X Window интерфейсе для gdb – xxgbd.

5.3 Аргументы MPI для программ пользователя

Приведенные ниже аргументы являются недокументированными возможностями MPICH. Некоторые из приведенных аргументов требуют, чтобы MPICH был сконфигурирован и собран со специальными параметрами:

-mpedbg

при возникновении ошибки в программе пользователя, запускает xterm, присоединенный к процессу, вызвавшему ошибку. MPICH должен быть сконфигурирован с опцией -mpedbg. Данный аргумент работает не на всех системах;

-mpiversion

печатает версию MPICH и аргументы, использованные при его конфигурировании;

`-mpichdebug`

генерирует детальную информацию по каждой производимой MPICH операции;

`-mpiqueue`

описывает состояние очередей вызова `MPI_Finalize`. Может быть использован для поиска “потерянных” сообщений.

Данные аргументы указываются программе пользователя, а не команде `mpirun`. Например,

```
mpirun -np 2 a.out -mpichdebug
```

Приложение А
(справочное)
Примеры программ с использованием функций MPI

Таблица 2

Имя файла	Язык	Комментарий	Число CPU
send_receive.f	Fortran 77	Простой пример, иллюстрирующий использование операций посылки и приема сообщений.	np>=2
ping_pong.c	C	Измеряет время, которое требуется для посылки и приема данных между двумя процессами.	np=2
compute_pi.f	Fortran 77	Вычисляет число pi, интегрируя f (x) =4 / (1+x2).	np>=1
master_worker.f90	Fortran 90	Выполняет параллельное вычисление массива по частям.	np>=2
cart.C	C++	Генерирует виртуальную топологию.	np=4
communicator.c	C	Копирует заданный по умолчанию коммуникатор MPI_COMM_WORLD.	np=2

А1 Схематичные примеры MPI-программ

А1.1

```
main(int argc, char **argv)
{
   int me, size;
   . . .
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &me);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   (void)printf ("Process %d size %d\n", me, size);
   . . .
   MPI_Finalize();
}
```

А1.2

```
#include "mpi.h"
main (argc, argv)
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)    /* code for process zero */
   {
     strcpy (message, "Hello, there");
      MPI_Send(messege,     strlen(messege),     MPI_CAHR,     1,     99,
MPI_COMM_WORLD);
   }
   else   /* code for process one */
   {
      MPI_Recv   (message,   20,   MPI_CHAR,   0,   99,   MPI_COMM_WORLD,
&status);
      printf ("receiveds :%s:\n", message);
   }
   MPI_Finalize();
}
```

A1.3

```
main(int argc, char **argv)
{
   int me, size;
   int SOME_TAG=0;
   MPI_Status status;
   . . .
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &me);    /* local */
   MPI_Comm-size (MPI_COMM_WORLD, &size);  /* local */
   if (me % 2)==0)
   {

/* send unless highest-numbered process */
     if ((me+1) < size)
        MPI_Send (..., me+1, SOME_TAG, MPI_COMM_WORLD);

   }
   else
     MPI_Recv (..., me-1, SOME_TAG, MPI_COMM_WORLD, &status);
   . . .
   MPI_Finalize();
}
```

A2 Примеры тестов

A2.1 Тест скорости пересылок (**`transfer`**):

Данный пример измеряет время, необходимое для передачи и приема сообщений между процессами.

```
/* ============================================================
     Test 4:  bi-directional transfer, ready mode send
                           Scheme
                      Node0 Node1
                      IRecv IRecv
                      Barrier    Barrier
                      RSend RSend
                      Wait  Wait
============================================================ */
```

```
#include "transfer.h"


static MPI_Status Statuses[2];
static MPI_Request Requests[2];

const char test_description[] = "Bi-directional MPI transfer: Ready-
mode Irecv/Irsend(4)";

void transfer_test_operation(void *sendbuf,void *recvbuf,int
msglen,int rank,int TIMES,int proc)
{
     int the_other;
     int i;
     if(proc == MASTER)
          return;
     if(rank == MASTER || rank == proc)
     {
          the_other = (rank == MASTER) ? proc : MASTER;
          for(i = 0; i < TIMES; i ++)
          {
            MPI_Irecv(recvbuf,msglen,MPI_CHAR,the_other,
          TAG1,comm,&Requests[0]);
            MPI_Barrier(comm);
            MPI_Irsend(sendbuf,msglen,MPI_CHAR,the_other,
          TAG1,comm,&Requests[1]);
            MPI_Waitall(2,Requests,Statuses);
          }
     }
}
```

A2.2 Тест логических топологий (**nettest**)

Пример генерирует виртуальную топологию, и иллюстрирует обмен сообщениями.

```
/*  ------------------------------------------------------
     MPI performance test suite
     Test for MPI logical topologies
     Alexander N. Andreyev (alexander@parallel.ru)
     Laboratory of Parallel Information Technologies, SRCC
     Tests originally developed by Ivan F. Rouzanov (CC RAS)
     See http://parallel.ru/testmpi/
     Copyright (c) 1997-2000

     this file: nettest.c (test shell)
     to build the test, attach the following files:
     star.c istar.c chaos.c ichaos.c ring.c ring2.c env.c
------------------------------------------------------ */

const char last_revision[] = "Feb 13, 2000";

/*****************************************************
  Include section
*****************************************************/

#include "mpi-test.h"
```

```
/*****************************************************
  Global data section
*****************************************************/

int i_am_the_master = 0;
static int msglen_min = MIN_MESSAGE_LENGTH;
static int msglen_max = MAX_MESSAGE_LENGTH;
static int TIMES = DEFAULT_TIMES;
static int msglen_step = MESSAGE_INCREMENT;
static int msglen_multiplier = 1;

FILE *out;
static char output_file_name[256];

int pool_size,rank;

#define COMM MPI_COMM_WORLD

/*****************************************************
  Code section
*****************************************************/


void main (int argc, char *argv[])
{
      int i;
      int msglen;
      double R[10]; /* rates */
      double t0,t1,TestTime;

      out = stdout;
      MPI_Init (&argc,&argv);
      MPI_Comm_size(COMM,&pool_size);
      MPI_Comm_rank(COMM,&rank);
      t0 = Wtime();

      if (pool_size < 3)
            finish( "At least 3 nodes required" );

      i_am_the_master = (rank == MASTER ) ? 1 : 0;

/* parse command line */
      if(i_am_the_master)
      {
        printf("\nMPI Network Test\n" );
        printf("(c) Alexander N. Andreyev, last revision:
%s\n",last_revision);

         for(i = 1; i < argc; i ++)
            switch(argv[i][0])
            {
              case 'm': msglen_min = atoi(argv[i]+1); break;
              case 'M': msglen_max = atoi(argv[i]+1); break;
              case 'T': TIMES = atoi(argv[i]+1); break;
              case 's': msglen_step = atoi(argv[i]+1); msglen_multiplier
= 1; break;
```

```
             case 'K': msglen_multiplier = atoi(argv[i]+1); msglen_step
= 0; break;

            case 'o': strcpy(output_file_name,argv[i]+1);
                      out = fopen(output_file_name,"a+");
                      if(out == NULL)
                        {
                          printf("WARNING: Can't open or create %s,
                        writing to stdout\n",output_file_name);
                          out = stdout;
                        }

                      else
                        printf("Writing output to %s\n",
                        output_file_name);

                        break;

             default:
                 fprintf(stderr,"WARNING: unrecognized option:
                      %s\n",argv[i]);
                 break;
           }

       fprintf(out,"\nRunning MPI Network Test on %d CPU(s)\n\n",
pool_size );
       fprintf(out,"Messages: %d to %d, step %d, multiplier %d; %d
times\n",msglen_min,msglen_max,msglen_step,msglen_multiplier,TIMES);
       fprintf(out,"\n====== Transfer rates =====
\n\nSize\tStar\tChaos\tRing\tiStar\tiChaos\tiRing \n\n");
     }

     MPI_Bcast(&msglen_min,1,MPI_INT,MASTER,COMM);
     MPI_Bcast(&msglen_max,1,MPI_INT,MASTER,COMM);
     MPI_Bcast(&msglen_step,1,MPI_INT,MASTER,COMM);
     MPI_Bcast(&msglen_multiplier,1,MPI_INT,MASTER,COMM);
     MPI_Bcast(&TIMES,1,MPI_INT,MASTER,COMM);


     for(msglen = msglen_min; msglen <= msglen_max; msglen = msglen *
     msglen_multiplier + msglen_step)
     {

#if 0
     fprintf(stderr,"%d Running Star test, msglen =
     %d...\n",rank,msglen); fflush(stderr);
#endif

     /* Star  topology (uni-directional) test. */
     R[0] = Star(msglen,TIMES);

#if 0
     fprintf(stderr,"%d Running Chaos test...\n",rank);
     fflush(stderr);
#endif
     /* Chaos topology (uni-directional) test. */
     R[1] = Chaos(msglen,TIMES);
#if 0
```

```
      fprintf(stderr,"%d Running Ring test...\n",rank); fflush(stderr);
#endif

      /* Ring  topology (uni-directional) test. */
      R[2] = Ring(msglen,TIMES);

#if 0
      fprintf(stderr,"%d Running iStar test...\n",rank);
      fflush(stderr);
#endif
      /* Star  topology (bi-directional) test. *
      R[3] = iStar(msglen,TIMES);     /

#if 0
      fprintf(stderr,"%d Running iChaos test...\n",rank);
      fflush(stderr);
#endif
      /* Chaos topology (bi-directional) test. */
      R[4] = iChaos(msglen,TIMES);

#if 0
      fprintf(stderr,"%d Running Ring2 test, msglen =
      %d...\n",rank,msglen); fflush(stderr);
#endif

      /* Ring  topology (bi-directional) test. */
      R[5] = Ring2(msglen,TIMES);

      if(i_am_the_master)
        {
        fprintf(out,"%d\t%.3lf\t%.3lf\t%.3lf\t%.3lf\t%.3lf\t%.3lf\n",
      msglen,R[0],R[1],R[2],R[3],R[4],R[5]);
        fflush(out);
        }
      }

      t1 = Wtime();
      TestTime = t1 - t0;

      if (i_am_the_master)
           printf ( "\n\nMPI Network Test complete in %lf
           sec\n",TestTime);

      MPI_Finalize();
      exit(0);

} /* end of main() */

/* End of file 'mpi-test.c' */
```

A2.3 Тест измерения скорости дисковых операций (`filetest`)

```
#include"filetest.h"

static int iTIMES = 1;
static int fsize_min = MIN_FSIZE, fsize_max = MAX_FSIZE;
static int fsize_step = 0, fsize_multiplier = 2;
```

```
static int buf_size = 0, test_space = 256*MEGABYTE;
static char folder[256] = ".";
static char output_file_name[256] = ".";
static FILE* out;

void print_io_rates(const io_rates_struct *ior)
{
  if (ior != NULL)
    {
      if(ior->iter >= 1)
        {
          double N = ior->iter;
          fprintf(out,"%d\t%d\t%d\t%.1f\t%.1f\t%.1f\t%.2f\t%.2f
\t%.2f\t%.2f\n",
           ior->size >> 10,
           ior->seg >> 10,
           ior->iter,
           ior->write_rate / N,
           ior->read_rate / N,
           ior->overwrite_rate / N,
           ior->create_time / N,
           ior->close_time / N,
           ior->open_time / N,
           ior->unlink_time / N);
         }
     }
  else
     {
     fprintf(out,"\nSize,K\tSeg,K\tNfiles\tWrite\tread
     \tOvrWr\tCreate\tClose\tOpen\tUnlink\n");
     }
}

int main(int argc,char* argv[])
{

  int fsize;
  io_rates_struct ior;
  int i;
  out = stdout;

  printf("File System Perfomance test\n");
  printf("(c) Laboratory of Parallel Information Technologies, 1997-
2000\n");
  for(i = 1; i < argc; i ++)
    switch(argv[i][0])
      {
        case 'm': fsize_min = atoi(argv[i]+1)*KILOBYTE; break;
        case 'M': fsize_max = atoi(argv[i]+1)*KILOBYTE; break;
        case 's': fsize_step = atoi(argv[i]+1)*KILOBYTE;
                  fsize_multiplier = 1; break;
        case 'K': fsize_multiplier = atoi(argv[i]+1); fsize_step = 0;
                  break;
        case 'b': buf_size = atoi(argv[i]+1)*KILOBYTE; break;
        case 'S': test_space = atoi(argv[i]+1)*KILOBYTE; break;
        case 'o': strcpy(output_file_name,argv[i]+1);
                  out = fopen(output_file_name,"a+");
```

```
                if(out == NULL)
                  {
                     printf("WARNING: Can't open or create %s,
                     writing to stdout\n",output_file_name);

                     out = stdout;
                   }
                  else
                     printf("Writing output to %s\n",
                     output_file_name);
                     break;

        case 'f': strcpy(folder,argv[i]+1);
                  printf("Creating test files in folder:
                    %s\n",folder);
                  break;

        default: fprintf(stderr,"WARNING: unrecognized option:
                  %s\n",argv[i]);
                 break;
    }

    initTiming();

    fprintf(out,"\nFile sizes from %dK to %dK, total test space
    %dM\n",fsize_min >> 10, fsize_max >> 10, test_space >> 20);

    print_io_rates(NULL);

    for(fsize = fsize_min; fsize <= fsize_max; fsize = fsize *
    fsize_multiplier + fsize_step)
      {
        ior.size = fsize;
        ior.iter = test_space / fsize;
        ior.seg = (buf_size == 0) ? fsize : buf_size;
        ior.open_time = ior.create_time = ior.unlink_time = 0;
        ior.write_rate = ior.overwrite_rate = ior.read_rate = 0;

        testFileWrite(folder,&ior);
        testFileRead(folder,&ior);
        testFileWrite(folder,&ior);
        testFileUnlink(folder,&ior);
        print_io_rates(&ior);
     }
     return 0;
}
```

A2.4 Программа `send_receive.f`

Это пример программы, написанной на языке Fortran 77, где процесс 0 посылает массив данных другим процессам группы `MPI_COMM_WORLD`.

```
      program main
      include 'mpif.h'
      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
```

```
integer status(MPI_STATUS_SIZE)
double precision data(100)
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
if (size .eq. 1) then
print *, 'must have at least 2 processes'
call MPI_Finalize(ierr)
stop
endif
print *, 'Process ', rank, ' of ', size, ' is alive'
dest = size - 1
src = 0
if (rank .eq. src) then
to = dest
count = 10
tag = 2001
do i=1, 10

data(i) = 1
enddo
call MPI_Send(data, count, MPI_DOUBLE_PRECISION,
+ to, tag, MPI_COMM_WORLD, ierr)
endif
if (rank .eq. dest) then
tag = MPI_ANY_TAG
count = 10
from = MPI_ANY_SOURCE
call MPI_Recv(data, count, MPI_DOUBLE_PRECISION,
+ from, tag, MPI_COMM_WORLD, status,

ierr)
call MPI_Get_Count(status, MPI_DOUBLE_PRECISION,
+ st_count, ierr)
st_source = status(MPI_SOURCE)
st_tag = status(MPI_TAG)
print *, 'Status info: source = ', st_source,
+ ' tag = ', st_tag, ' count = ', st_count
print *, rank, ' received', (data(i),i=1,10)
endif
call MPI_Finalize(ierr)
stop
end
```

A2.5 Программа `ping_pong.c`

Это пример программы на языке C для замера скорости пересылки сообщений.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#define NLOOPS 1000
#define ALIGN 4096
```

```
main(argc, argv)
int argc;
char *argv[];
{
int i, j;
double start, stop;
int nbytes = 0;
int rank, size;
MPI_Status status;
char *buf;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size != 2) {
  if ( ! rank) printf("ping_pong: must have two processes\n");
  MPI_Finalize();
  exit(0);
}
nbytes = (argc > 1) ? atoi(argv[1]) : 0;
if (nbytes < 0) nbytes = 0;

/* Page-align buffers and displace them in the cache to avoid
collisions.*/

buf = (char *) malloc(nbytes + 524288 + (ALIGN - 1));
if (buf == 0) {
  MPI_Abort(MPI_COMM_WORLD, MPI_ERR_BUFFER);
  exit(1);
}
buf = (char *)((((unsigned long) buf)+(ALIGN-1)) & ~(ALIGN-1));
if (rank == 1) buf += 524288;
memset(buf, 0, nbytes);

/* Ping-pong.*/

if (rank == 0) {
  printf("ping-pong %d bytes ...\n", nbytes);
  /*
  * warm-up loop
  */
  for (i = 0; i < 5; i++) {
    MPI_Send(buf, nbytes, MPI_CHAR,1,1, MPI_COMM_WORLD);
    MPI_Recv(buf, nbytes, MPI_CHAR,1,1, MPI_COMM_WORLD,&status);
  }
  /*
  * timing loop
  */
  start = MPI_Wtime();
  for (i = 0; i < NLOOPS; i++) {
  #ifdef CHECK
     for (j = 0; j < nbytes; j++) buf[j] = (char)(j + i);
  #endif
  MPI_Send(buf, nbytes, MPI_CHAR, 1, 1000 + i, MPI_COMM_WORLD);
  #ifdef CHECK
  memset(buf, 0, nbytes);
  #endif
```

```
  MPI_Recv(buf, nbytes, MPI_CHAR, 1, 2000 + i, MPI_COMM_WORLD,
    &status);
  #ifdef CHECK
  for (j = 0; j < nbytes; j++) {
  if (buf[j] != (char) (j + i)) {
    printf("error: buf[%d] = %d, not %d\n", j, buf[j], j + i);
    break;
  }
  }
  #endif
}
stop = MPI_Wtime();
printf("%d bytes: %.2f usec/msg\n",
nbytes, (stop - start) / NLOOPS / 2 * 1000000);
if (nbytes > 0) {
  printf("%d bytes: %.2f MB/sec\n", nbytes,
  nbytes / 1000000. /
  ((stop - start) / NLOOPS / 2));
  }
}
else {
  /*
  * warm-up loop
  */
  for (i = 0; i < 5; i++) {
    MPI_Recv(buf, nbytes, MPI_CHAR,0,1, MPI_COMM_WORLD, &status);
    MPI_Send(buf, nbytes, MPI_CHAR,0,1, MPI_COMM_WORLD);
  }
  for (i = 0; i < NLOOPS; i++) {
    MPI_Recv(buf, nbytes, MPI_CHAR, 0, 1000 + i, MPI_COMM_WORLD,
     &status);
    MPI_Send(buf, nbytes, MPI_CHAR, 0, 2000 + i, MPI_COMM_WORLD);
  }
}
MPI_Finalize();
exit(0);
}
```

A2.6 Программа `compute_pi.f`

Это пример программы на Fortran 77, которая вычисляет число pi интегрируя функцию f(x) = 4/(1 + x^2).

Каждый процесс:

- получает число интервалов, используемых в приближении;
- вычисляет области его прямоугольников;
- синхронизирует для глобального суммирования.

Программа выдает результат вычислений, погрешность и время вычисления.

```
program main
include 'mpif.h'
double precision PI25DT
parameter(PI25DT = 3.14159265358979323846264338327950288419716939937510d0)
double precision mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr
C
```

```
C Function to integrate
C
f(a) = 4.d0 / (1.d0 + a*a)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
print *, "Process ", myid, " of ", numprocs, " is alive"
sizetype = 1
sumtype = 2
if (myid .eq. 0) then
n = 100
endif
call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
C
C Calculate the interval size.
C
h = 1.0d0 / n
sum = 0.0d0
do 20 i = myid + 1, n, numprocs
x = h * (dble(i) - 0.5d0)
sum = sum + f(x)
20 continue
mypi = h * sum
C
C Collect all the partial sums.
C
call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0,
 MPI_COMM_WORLD, ierr)
C
C Process 0 prints the result.
C
if (myid .eq. 0) then
 write(6, 97) pi, abs(pi - PI25DT)

97 format(' pi is approximately: ', F18.16, ' Error is: ',
 F18.16)
endif
call MPI_FINALIZE(ierr)
stop
end
```

A2.6 Программа `master_worker.f90`

В этом примере для Fortran 90, главный процесс задает число рабочих процессов (`numtasks-1`). Главный процесс разделяет массив на равные части и посылает каждому рабочему процессу. Каждый рабочий процесс получает свою часть массива и устанавливает значение каждого элемента равным <`индекс элемента`>+1. Затем каждый рабочий процесс посылает свою часть измененного массива назад главному процессу.

```
program array_manipulation
include 'mpif.h'
integer (kind=4) :: status(MPI_STATUS_SIZE)
integer (kind=4), parameter :: ARRAYSIZE = 10000, MASTER = 0
integer (kind=4) :: numtasks, numworkers, taskid, dest, index,
  i
```

```
integer (kind=4) :: arraymsg, indexmsg, source, chunksize,
  int4, real4
real (kind=4) :: data(ARRAYSIZE), result(ARRAYSIZE)
integer (kind=4) :: numfail
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, taskid, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, numtasks, ierr)
numworkers = numtasks - 1
chunksize = (ARRAYSIZE / numworkers)
arraymsg = 1
indexmsg = 2
int4 = 4
real4 = 4
numfail = 0
! ***************************** Главная задача *****************
if (taskid .eq. MASTER) then
data = 0.0
index = 1
do dest = 1, numworkers
call MPI_Send(index, 1, MPI_INTEGER, dest, 0,
MPI_COMM_WORLD, ierr)
57
call MPI_Send(data(index), chunksize, MPI_REAL, dest, 0,
 &MPI_COMM_WORLD, ierr)
index = index + chunksize
end do
do i = 1, numworkers
source = i
call MPI_Recv(index, 1, MPI_INTEGER, source, 1, MPI_COMM_WORLD,
 &status, ierr)
call MPI_Recv(result(index), chunksize, MPI_REAL, source, 1,
 &MPI_COMM_WORLD, status, ierr)
end do
do i = 1, numworkers*chunksize
if (result(i) .ne. (i+1)) then
print *, 'element ', i, ' expecting ', (i+1), '
actual is ', result(i)
numfail = numfail + 1
endif
enddo
if (numfail .ne. 0) then
print *, 'out of ', ARRAYSIZE, ' elements, ', numfail, ' wrong
 answers'
else
print *, 'correct results!'
endif
endif
! *************** Задача Рабочего *********************
if (taskid .gt. MASTER) then
call MPI_Recv(index, 1, MPI_INTEGER, MASTER, 0,
 MPI_COMM_WORLD, &status, ierr)
call MPI_Recv(result(index), chunksize, MPI_REAL, MASTER,
 0, &MPI_COMM_WORLD, status, ierr)
do i = index, index + chunksize -1
result(i) = i + 1
end do
call MPI_Send(index, 1, MPI_INTEGER, MASTER, 1,
 MPI_COMM_WORLD, ierr)
```

```
58
call MPI_Send(result(index), chunksize, MPI_REAL, MASTER,
 1, &MPI_COMM_WORLD, ierr)
end if
call MPI_Finalize(ierr)
end program array_manipulation
```

A2.5 Программа `cart.C`

Эта программа на C++ эмулирует генерацию виртуальной топологии. Класс Node описывает узел 2-мерного тора. В каждом узле хранятся целочисленные данные, которыми, при помощи операций сдвига, узел обменивается с соседними узлами. Таким образом, после сдвига данных вверх-вправо-вниз-влево (или север-восток-юг-запад), каждый узел получит свои начальные данные.

```
#include <stdlib.h>
#include <mpi.h>
#define NDIMS 2
typedef enum { NORTH, SOUTH, EAST, WEST } Direction;
// A node in 2-D torus
class Node {
private:
MPI_Comm comm;
int dims[NDIMS], coords[NDIMS];
int grank, lrank;
int data;
public:
Node(void);
~Node(void);
void profile(void);
void print(void);
void shift(Direction);
};
// A constructor
Node::Node(void)
{
int i, nnodes, periods[NDIMS];
// Create a balanced distribution
MPI_Comm_size(MPI_COMM_WORLD, &nnodes);
for (i = 0; i < NDIMS; i++) { dims[i] = 0; }
MPI_Dims_create(nnodes, NDIMS, dims);
// Establish a Cartesian topology communicator
for (i = 0; i < NDIMS; i++) { periods[i] = 1; }
MPI_Cart_create(MPI_COMM_WORLD, NDIMS, dims, periods, 1, &comm);
// Initialize the data
MPI_Comm_rank(MPI_COMM_WORLD, &grank);

if (comm == MPI_COMM_NULL) {
lrank = MPI_PROC_NULL;
data = -1;
} else {
MPI_Comm_rank(comm, &lrank);
data = lrank;
MPI_Cart_coords(comm, lrank, NDIMS, coords);
}
```

```
      }
      // A destructor
      Node::~Node(void)
      {
      if (comm != MPI_COMM_NULL) {
      MPI_Comm_free(&comm);
      }
      }
      // Shift function
      void Node::shift(Direction dir)
      {
      if (comm == MPI_COMM_NULL) { return; }
      int direction, disp, src, dest;
      if (dir == NORTH) {
      direction = 0; disp = -1;
      } else if (dir == SOUTH) {
      direction = 0; disp = 1;
      } else if (dir == EAST) {
      direction = 1; disp = 1;
      } else {
      direction = 1; disp = -1;
      }
      MPI_Cart_shift(comm, direction, disp, &src, &dest);
      MPI_Status stat;
      MPI_Sendrecv_replace(&data,   1,   MPI_INT,   dest,   0,   src,   0,
comm,&stat);
      }
      // Synchronize and print the data being held
      void Node::print(void)
      {
      if (comm != MPI_COMM_NULL) {
      MPI_Barrier(comm);
      if (lrank == 0) { puts(""); } // line feed
      MPI_Barrier(comm);
      printf("(%d, %d) holds %d\n", coords[0], coords[1], data);
      }
      }
      // Print object's profile
      void Node::profile(void)
      {
      // Non-member does nothing
      if (comm == MPI_COMM_NULL) { return; }
      // Print "Dimensions" at first
      if (lrank == 0) {
      printf("Dimensions: (%d, %d)\n", dims[0], dims[1]);
      }
      MPI_Barrier(comm);
      // Each process prints its profile
      printf("global rank %d: cartesian rank %d, coordinate (%d,
      %d)\n",
      grank, lrank, coords[0], coords[1]);
      }
      // Program body
      //
      // Define a torus topology and demonstrate shift operations.
      //
      void body(void)
      {
```

```
Node node;
node.profile();
node.print();
node.shift(NORTH);

node.print();
node.shift(EAST);
node.print();
node.shift(SOUTH);
node.print();
node.shift(WEST);
node.print();
}
//
// Main program---it is probably a good programming practice to
call
// MPI_Init() and MPI_Finalize() here.
//
int main(int argc, char **argv)
{
MPI_Init(&argc, &argv);
body();
MPI_Finalize();
}
```

A2.5 Программа `communicator.c`

Этот C пример показывает, как делать копию заданного по умолчанию коммуникатора `MPI_COMM_WORLD`.

```
#include <stdio.h>
#include <mpi.h>
main(argc, argv)
int argc;
char *argv[];
{
int rank, size, data;
MPI_Status status;
MPI_Comm libcomm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_dup(MPI_COMM_WORLD, &libcomm);
if (rank == 0) {
  data = 12345;
  MPI_Send(&data, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
  data = 6789;
  MPI_Send(&data, 1, MPI_INT, 1, 5, libcomm);
} else {
  MPI_Recv(&data, 1, MPI_INT, 0, 5, libcomm, &status);
  printf("received libcomm data = %d\n", data);
  MPI_Recv(&data, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &status);
  printf("received data = %d\n", data);
}
MPI_Comm_free(&libcomm);
MPI_Finalize();
exit(0);
```

```
}
```

A2.5 Программа `hello_world.c`

Это C-версия простейшей программы “Hello world!”. Каждая ветвь данной программы печатает текстовую строку “Hello world! I'm *r* of *s* on *host*” (“Привет, мир! Мой номер *r* из *s* на *host*”), где *r* – ранг процесса, *s* – размер коммуникатора, и host – узел, на котором запущена ветвь.

```
/* hello_world.c */
#include <stdio.h>
#include <mpi.h>
main(argc, argv)
int argc;
char *argv[];
{
  int rank, size, len;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Get_processor_name(name, &len);
  printf ("Hello world! I'm %d of %d on %s\n", rank, size, name);
  MPI_Finalize();
  exit(0);
}
```

A2.5 Программа `cpilog.c`

Эта программа вычисляет число π, и выводит информацию о времени вычисления для каждого процесса отдельно. Программа использует профилировочные библиотеки MPE. Исходный текст cpilog.c находится в /common/mpich/share/examples.

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

double f( double );
double f( double a)
{
    return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
    int  n, myid, numprocs, i, j;
    double PI25DT = 3.14159265358979323846264643;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int namelen;
    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);

    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d running on %s\n", myid,
processor_name);

    /*
        MPE_Init_log() & MPE_Finish_log() are NOT needed when
        liblmpe.a is linked with this program.  In that case,
        MPI_Init() would have called MPE_Init_log() already.
    */
/*
    MPE_Init_log();
*/

    /* Get event ID from MPE, user should NOT assign event ID  */
    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) {
      MPE_Describe_state(event1a, event1b, "Broadcast", "red");
      MPE_Describe_state(event2a, event2b, "Compute",   "blue");
      MPE_Describe_state(event3a, event3b, "Reduce",    "green");
      MPE_Describe_state(event4a, event4b, "Sync",      "orange");
    }

    if (myid == 0)
    {
      n = 1000000;
      startwtime = MPI_Wtime();
    }
    MPI_Barrier(MPI_COMM_WORLD);

    MPE_Start_log();

    for (j = 0; j < 5; j++)
    {
      MPE_Log_event(event1a, 0, "start broadcast");
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPE_Log_event(event1b, 0, "end broadcast");

      MPE_Log_event(event4a,0,"Start Sync");
      MPI_Barrier(MPI_COMM_WORLD);
      MPE_Log_event(event4b,0,"End Sync");

      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 += f(x);
     }
     mypi = h * sum;
     MPE_Log_event(event2b, 0, "end compute");

     MPE_Log_event(event3a, 0, "start reduce");
     MPI_Reduce(&mypi,   &pi,   1,   MPI_DOUBLE,   MPI_SUM,   0,
MPI_COMM_WORLD);
     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);
    }
    MPI_Finalize();
    return(0);
}
```

Перечень ссылочных документов

1. William Gropp and Ewing Lusk.
Installation Guide to mpich, a Portable Implementation of MPI version 1.2.1.
http://www-unix.mcs.anl.gov/mpi/mpich/docs/install/paper.htm

2. William Gropp and Ewing Lusk.
User's Guide for mpich, a Portable Implementation of MPI version 1.2.1.
http://www-unix.mcs.anl.gov/mpi/mpich/docs/userguide/paper.htm

3. William Gropp and Ewing Lusk
User's Guide for MPE: Extensions for MPI Programs
http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeguide/paper.htm

4. William Gropp and Ewing Lusk.
A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard
http://www-unix.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html

5. Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, Jack Dongarra.
MPI: The Complete Reference
http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

6. Neil MacDonald, Elspeth Minty, Mario Antonioletti, Joel Malard, Tim Harding, Simon Brown.
Writing Message-Passing Parallel Programs with MPI -
http://www.epcc.ed.ac.uk/epic/mpi/notes/mpi-course-epic.book_1.html