5 Группы, контексты, коммуникаторы и кеширование.

5.1 Введение.

Эта глава представляет особенности MPI, которые помогают при разработке параллельных библиотек. Параллельные библиотеки необходимы, чтобы обособлять сложности присущие параллельной реализации ключевых алгоритмов. Они гарантируют совместимость таких процедур, и обеспечивают более "высокий уровень" переносимости, чем может обеспечить MPI сам по себе. Как таковые, библиотеки освобождают каждого программиста от рутинной работы определения последовательных структур данных, формата данных и методов, которые реализуют ключевые алгоритмы (такие как матричные операции). Поскольку лучшие библиотеки приходят с некоторыми различиями по параллельным системам (различные форматы данных, различные стратегии, зависящие от размера системы или задачи, или типа плавающей запятой), то необходимо чтобы это было скрыто от пользователя.

Мы отсылаем читателя к [26] и [3] для получения более подробной информации по написанию библиотек в MPI, используя особенности, описанные в этой главе.

5.1.1 Особенности необходимые для поддержки библиотек.

Ключевыми особенностями, необходимыми для поддержки создания жизнеспособных параллельных библиотек, являются:

- Защита пространства обменов, которая гарантирует, что библиотеки могут связываться как им необходимо, не конфликтуя между собой,
- Групповое поле действия для совместных операций, что позволяет библиотекам избегать ненужной синхронизации невовлекаемых процессов (возможно запуская несвязанный код),
- Именование абстрактного процесса, которое позволяет библиотекам описывать свои связи в терминах подходящих к их собственным структурам данных и алгоритмам,
- Способность "украшать" установку коммуникационных процессов с добавлением атрибутов, определяемых пользователем, таких как дополнительные совместные операции. Этот механизм будет обеспечивать средства для пользователя или создателя библиотек которые эффективно расширяют представление проходящих сообщений.

Кроме того, необходим унифицированный механизм или способ для удобного обозначения контекста связи, группы процессов, обслуживания абстрактного именования процессов и хранения дополнительной информации.

5.1.2 Поддержка MPI для библиотек

Специально для поддержки жизнеспособности библиотек, MPI обеспечивает соответствующие принципы, такие как:

- Контексты связи,
- Группы процессов,
- Виртуальные топологии,
- Атрибут кеширования,
- Коммуникаторы.

Коммуникаторы (см. [16,24]) объединяют все эти идеи для того, чтобы обеспечить подходящую область действий всем коммуникационным операциям в MPI. Коммуникаторы делятся на два вида: интра-коммуникаторы для операций внутри отдельной группы процессов и интер-коммуникаторы для двухточечной связи между двумя группами

процессов.

Кеширование. Коммуникаторы (см. ниже) обеспечивают механизм "кеширования", который позволяет им связывать новые атрибуты с коммуникаторами наравне со встроенными особенностями MPI. Это может быть использовано некоторыми пользователями чтобы усовершенствовать коммуникаторы в дальнейшем, а также интерфейсом MPI, при реализации некоторых функций коммуникатора. Например, функции виртуальной топологии, описанные в главе 6, вероятно будут поддержаны этим способом.

Группы. Группы определяют упорядоченную совокупность процессов, каждый имеет свой ранг в этой группе, который определяет низко-уровневые имена для межпроцессных связей (ранки используются для передачи и приема). Таким образом, группы определяют диапазон для имен процессов при двухточечной связи. Добавим, что на группах определяются групповые операции. В MPI можно манипулировать группами отдельно от коммуникаторов, но только коммуникаторы могут быть использованы для активизации операций связи.

Интра-коммуникаторы. Интра-коммуникаторы - это наиболее простые средства для реализации передачи сообщений в MPI. Интра-коммуникаторы содержат образец группы, контексты связи как для двухточечной, так и для групповых обменов и имеют возможность включать в себя виртуальную топологию и другие атрибуты. Особенности работы интра-коммуникаторов представлены ниже:

- Контексты обеспечивают возможность иметь в MPI раздельные защищенные "области" передаваемых сообщений. Контекст схож с дополнительным признаком, по которому различаются сообщения. Процессом распознавания система управляет. Использование отдельных контекстов связи определенными библиотеками (или вызовами определенных библиотек) изолирует внутреннюю связь, по отношению к выполнению библиотеки, от внешней связи. Это допускает вызов библиотеки даже если на "других" коммуникаторах находятся задержанные связи, и позволяет избегать синхронизации при входе и выходе из библиотечного кода. Задержка двухточечных связей также гарантирует изолированность от совместных связей внутри одного коммуникатора.

- Группа определяет участников обмена (см. выше) коммуникатора.

- Виртуальная топология определяет особое соответствие ранков в группе исходя из топологии, определяемой пользователем. Особые конструкторы коммуникаторов, определяемые в следующей главе, обеспечивает эту особенность. Интра-коммуникаторы, как описано в этой главе, не имеют топологий.

- Атрибуты кэша определяют локальную информацию о том, что пользователь или библиотека присоединились к коммуникатору для последующего сообщения.

  Совет пользователям. Существующая практика во многих библиотеках обменов состоит в том, что существует уникальное, предопределенное пространство обменов, которое включает все доступные во время запуска параллельной программы процессы; которым приписываются последовательные ранги. Участники двухточечной связи обозначаются своим рангом; групповые обмены (например, распространение сообщения) всегда включают в себя все процессы. Это практически может быть реализовано в MPI использованием предопределенного коммуникатора MPI_COMM_WORLD. Пользователи, которым этого достаточно, могут ставить MPI_COMM_WORLD, где бы ни потребовался параметр коммуникатора и могут пропустить оставшуюся часть этой главы. (Конец совета пользователю.)

Интер-коммуникаторы. До сих пор обсуждалась интра-связь: связь внутри группы. В MPI поддерживается также интер-связь: связь между двумя неперекрывающимися группами. Когда прикладная программа сформирована из нескольких параллельных модулей, удобно позволять одному модулю связываться с другим, используя локальные ранги для адресации в пределах второго модуля. Это особенно удобно при вычислительной парадигме клиент-сервер, где или клиент или сервер параллелен. Поддержка интер-связи обеспечивает механизм расширения MPI до динамической модели, где не все процессы определяются во время инициализации. В таких ситуациях становится необходимым поддерживать связь через "множества". Интер-связь обеспечивается интер-коммуникаторами. Эти объекты связывают вместе две группы с контекстами связи, разделяемыми обоими группами. Особенности работы интер-коммуникаторов представлены ниже:

- Контексты обеспечивают возможность иметь раздельные защищенные "области" сообщения между двумя группами. Передача из локальной группы всегда принимается в удаленной группе и наоборот. Система управляет этим процессом распознавания. Использование отдельных контекстов определенными библиотеками (или вызовами определенных библиотек) изолирует внутреннюю связь, по отношению к выполнению библиотеки, от внешней связи. Это допускает вызов библиотеки даже если на "других" коммуникаторах находятся задержанные связи, и позволяет избегать необходимость синхронизации входа и выхода из библиотечного кода. Для интер-коммуникаторов нет никаких универсальных групповых обменов, так что контексты используются только чтобы изолировать двухточечную связь.

- Локальная и удаленная группа определяют получателей и адресатов для интер-коммуникатора.

- Виртуальная топология не определена для интер-коммуникаторов.

- Как прежде, атрибуты кэша определяют локальную информацию о том, что пользователь или библиотека присоединились к коммуникатору для последующего сообщения.

MPI обеспечивает механизмы для создания и манипулирования интер-коммуникаторами. Они используются для двухточечной связи и относительно похожи на интра-коммуникаторы. Пользователи, которым не нужна интер-связь в их задачах, могут спокойно проигнорировать эту возможность. Пользователи, которым нужны совместные операции через интер-коммуникаторы, должны поместить это на вершину MPI. Пользователи, которым требуется интер-связь между перекрывающимися группами, также должны использовать эту возможность на вершине MPI.

5.2 Основные понятия

В этом разделе мы вернёмся к более формальному определению понятий, введенных выше.

5.2.1 Группы

Группа является упорядоченным множеством идентификаторов процесса (дальше процессов); процессы - объекты, зависящие от реализации. Каждый процесс в группе связан с целым ранком. Ранки идут подряд начиная с нуля. Группы представляются скрытыми объектами групп и, следовательно, не могут быть переданы от одного процесса к другому. Группа используется внутри коммуникатора, чтобы описывать участников в "области" связи и ранжировать таких участников (таким образом присваивая им уникальные имена внутри этой "области" обменов).

Существует особая предопределенная группа MPI_GROUP_EMPTY, которая является группой без элементов. Предопределенная константа

MPI_GROUP_NULL является значением, используемым для обработки неверной группы.

> Совет пользователям. MPI_GROUP_EMPTY, который является допустимым указателем на пустую группу, нельзя путать с MPI_GROUP_NULL, который в свою очередь является недопустимым указателем. Первый может использоваться как параметр групповых операций; последний, который возвращается, когда освобождается группа, не является допустимым параметром. (Конец совета пользователям.)
>
> Совет разработчикам. Группа может представляться таблицей перевода адреса в процесс. Каждый объект коммуникатора (см. ниже) будет иметь указатель на такую таблицу.
>
> Простые реализации MPI будут перечислять группы, так как в таблице. Однако, более развитые структуры данных имеют смысл для того чтобы улучшать масштабируемость и использование памяти с большим количеством процессов. В MPI возможны такие реализации. (Конец совета разработчикам.)

5.2.2 Контексты

Контексты - это свойство коммуникаторов (описаны ниже), которое позволяет разделять пространство связи. Сообщение, посланное в одном контексте, не может быть принято в другом. К тому же, где разрешено, групповые операции не зависят от задержки двухточечных операций. Контексты - неявные объекты MPI; они появляются только как часть реализации коммуникаторов (см.ниже).

> Совет разработчикам. Определенные коммуникаторы в одном и том же процессе имеют определенные контексты. Контекст - по существу управляемый системой признак (или признаки) необходимый, чтобы защитить коммуникатор для двухточечных и MPI-определенных групповых обменов.
>
> Защита заключается в том, что групповые и двухточечные обмены не пересекаются внутри коммутатора, и в том, что связь не создает помех на определенных коммуникаторах.
>
> Возможная реализация для контекста заключается в том, что дополнительный признак присоединяется к сообщению при передаче и проверяется при приеме. Каждый интра-коммуникатор хранит значения этих двух признаков (один для двухточечной и один для совместной связи). Функции порождения коммуникаторов используют групповые обмены для согласования нового уникального контекста для группы.
>
> Аналогично, при интер-связи (которая определенно является двухточечной связью) два признака контекста хранятся в коммуникаторе, первый используется группой A при передаче и группой B при приеме, второй используется группой B при передаче и группой A при приеме.
>
> Так как контексты являются неявными объектами, возможны также другие реализации. (Конец совета разработчикам.)

5.2.3 Интра-коммуникаторы

Интра-коммуникаторы сводят вместе понятия группы и контекста. Чтобы оставить возможность оптимизации с особенностями реализации и топологии прикладных программ (определяются в главе 6), коммуникаторы также могут "кешировать" дополнительную информацию (см. часть 5.7). Операции обменов MPI обращаются к коммуникатору, чтобы определить диапазон и "пространство обменов", в котором действуют двухточечные и групповые обмены.

Каждый коммуникатор содержит группу допустимых участников; эта

группа всегда содержит локальные процессы. Источник и приемник сообщения идентифицируются ранком сообщения внутри этой группы.

При групповой операции интра-коммуникатор определяет множество процессов, которые в ней участвуют (и их порядок, когда это значимо). Таким образом, коммуникатор ограничивает "пространственный" диапазон связи, и поддерживает машинонезависимый процесс адресации, через ранки.

Интра-коммуникаторы представляются скрытыми объектами интра-коммуникатора, и следовательно не могут быть переданы от одного процесса к другому.

5.2.4 Предопределенные интра-коммуникаторы

В начале интра-коммуникатор MPI_COMM_WORLD включает все процессы, он определяется после вызова MPI_INIT. Кроме того, поддерживается коммуникатор MPI_COMM_SELF, который включает только сам по себе процесс.

Предопределенная константа MPI_COMM_NULL является значением, используемым для обработки неверного коммуникатора.

> Совет разработчикам. В реализации со статическими процессами MPI, все процессы, которые участвуют в вычислении, доступны после инициализации MPI. В этом случае, MPI_COMM_WORLD является коммуникатором всех процессов, доступных для вычисления; этот коммуникатор имеет одно и тоже значение на всех процессах. В реализации MPI, где процессы могут динамически присоединяться MPI во время выполнения, это может привести к тому, что процесс начинает MPI вычисление без доступа к другим процессам. В такой ситуации, MPI_COMM_WORLD является коммуникатором, включающим все процессы с которыми соединяющийся процесс может немедленно связываться. Поэтому, MPI_COMM_WORLD может одновременно иметь различные значения на различных процессах. Все реализации MPI требуют поддержки коммуникатора MPI_COMM_WORLD. Он не может быть освобожден в процессе работы. Группа, соответствующая этому коммуникатору, не определяется сразу, но она может быть доступна при использовании MPI_COMM_GROUP (см.ниже). MPI не устанавливает соответствия между ранком процесса в MPI_COMM_WORLD и его (машинозависимым) абсолютным адресом. MPI также не определяет функцию главного процесса, если таковой вообще имеется. Возможны также другие предопределенные коммуникаторы, в зависимости от реализации. (Конец совета разработчикам.)

5.3 Управление группами

В этом разделе описывается манипулирование группами процессов в MPI. Эти операции локальны и их выполнение не требует межпроцессных обменов.

5.3.1 Доступ к группе

```
MPI_GROUP_SIZE(group, size)
IN     group         группа (handle)
OUT    size          количество процессов в группе (integer)

int MPI_Group_size(MPI_Group group, int *size)

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
       INTEGER GROUP, SIZE, IERROR

MPI_GROUP_RANK(group, rank)
IN     group         группа (handle)
OUT    rank          ранк вызываемого процесса в группе или
                     MPI_UNDEFINED если процесс не член группы (integer)

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

```
MPI_GROUP_RANK(GROUP, RANK, IERROR)
       INTEGER GROUP, RANK, IERROR

MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)
IN     group1        группа 1 (handle)
IN     n             число ранков в массивах ranks1 и ranks2 (integer)
IN     ranks1        массив из нуля или более ранков в группе 1
IN     group2        группа 2 (handle)
OUT    ranks2        массив соответствующих рангов в группе 2,
                     MPI_UNDEFINED когда соответствия не существует

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

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

Целесообразность. Кроме этой функции нет мобильного способа быстро определять относительные ранги процессов в двух группах. (Конец замечания.)

Совет пользователям. Эта функция важна для определения относительной нумерации одних и тех же процессов в двух разных группах. Например, если известны ранки определенных процессов в группе MPI_COMM_WORLD, то можно определить их ранки в подмножестве этой группы. (Конец совета пользователям.)

```
MPI_GROUP_COMPARE(group1, group2, result)
IN     group1        первая группа (handle)
IN     group2        вторая группа (handle)
OUT    result        результат (integer)
```

Результатом этого будет MPI_IDENT, если элементы и порядок групп одинаковы. Это происходит, например, если group1 и group2 являются одним и тем же хэндлером. Если члены группы одинаковы, а порядок различен, то получается MPI_SIMILAR. В любом другом случае получаем MPI_UNEQUAL.

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

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

5.3.2 Конструкторы групп

Конструкторы группы используются для создания подмножества или надмножества существующих групп. Они создают новые группы из существующих групп. Это локальные операции и определенные группы могут быть определены на разных процессах; процесс может определить группу, которая его не содержит. Последовательные определения требуются, когда группы используются как аргументы в функциях построения коммуникатора. MPI не обеспечивает механизм построения группы с самого начала, а только из других, ранее определенных групп. Основная группа, из которой определяются все остальные группы, связана с начальным коммуникатором MPI_COMM_WORLD (доступна через функцию MPI_COMM_GROUP).

Целесообразность. Нет смысла в функции дублирования группы аналогичной MPI_COMM_DUP, определенной дальше в этой главе. Так как нет необходимости в механизме дублирования группы. Группа, однажды созданная, может иметь несколько ссылок на себя при копировании хэндлеров. Последующие конструкторы адресов необходимы для подмножеств и надмножеств существующих групп. (Конец замечания.)

Совет разработчикам. Каждый конструктор группы ведет себя как будто бы он возвращает новый объект группы. Когда эта новая группа является копией существующей группы, тогда она может избежать создания таких новых объектов, используя механизм подсчета ссылок (Конец совета разработчикам.).

```
MPI_COMM_GROUP(comm, group)
IN     comm          коммуникатор (handle)
OUT    group         группа соответствующая коммуникатору (handle)

int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)

MPI_COMM_GROUP(COMM, GROUP, IERROR)
       INTEGER COMM, GROUP, IERROR
```

MPI_COMM_GROUP возвращает в group хэндлер для группы коммуникатора comm.

```
MPI_GROUP_UNION(group1, group2, newgroup)
IN     group1        первая группа (handle)
IN     group2        вторая группа (handle)
OUT    newgroup      группа образованная объединением (handle)

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

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

MPI_GROUP_INTERSECTION(group1, group2, newgroup)
IN     group1        первая группа (handle)
IN     group2        вторая группа (handle)
OUT    newgroup      группа образованная пересечением (handle)

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

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

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
IN     group1        первая группа (handle)
IN     group2        вторая группа (handle)
OUT    newgroup      группа образованная разностью (handle)

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

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
       INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

Определены следующие операции подобные операциям с множествами:

Объединение Все элементы первой группы (group1) объединяются со всеми элементами второй группы (group2), элементы второй группы располагаются за элементами первой группы.

Пересечение Все элементы первой группы, которые являются элементами второй группы, упорядочены как элементы первой группы.

Разность Все элементы первой группы, которые не принадлежат второй группе, упорядочены как элементы первой группы.

Заметим, что для этих операций порядок процессов в выходной группе определяется, прежде всего, порядком в первой группе (если возможно) и затем, если необходимо, порядком во второй группе. Ни операция объединения, ни операция пересечения не являются

коммутативными, а обе являются ассоциативными.
Новая группа может быть пустой, то есть MPI_GROUP_EMPTY.

```
MPI_GROUP_INCL(group, n, ranks, newgroup)
IN     group          группа (handle)
IN     n              число элементов в массиве ранков (и размер
                      newgroup) (integer)
IN     ranks          ранки процессов в group помещаемых в newgroup
                      (array of integers)
OUT    newgroup       новая группа полученная из указанной выше в порядке
                      определенном ranks (handle)

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

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

Совет пользователям. Функция MPI_GROUP_INCL создает группу newgroup, которая содержит n процессов в group с ранками rank[0],...,rank[n-1]; процесс с рангом i в newgroup является процессом с ранком rank[i] в group. Каждый из n элементов ranks может быть допустимым рангом в group и все элементы могут быть определенны, иначе программа будет ошибочной. Если n = 0, тогда newgroup - это MPI_GROUP_EMPTY. Эта функция может, например, быть использована для перенумерации элементов группы. Смотри также MPI_GROUP_COMPARE. (Конец совета пользователям.)

```
MPI_GROUP_EXCL(group, n, ranks, newgroup)
IN     group          группа (handle)
IN     n              число элементов в массиве ранков (integer)
IN     ranks          массив целых ранков в group непомещаемых в newgroup
OUT    newgroup       новая группа полученная из указанной выше и
                      сохраняющая порядок определенный в group (handle)

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

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

Совет пользователям. Функция MPI_GROUP_EXCL создает группу процессов newgroup, которая получается при удалении из group процессов с ранками rank[0],...,rank[n-1]. Порядок процессов в newgroup идентичен порядку в group. Каждый из n элементов массива ranks должен быть допустимым ранком в group и все элементы должны быть определены; иначе программа будет ошибочной. Если n = 0, тогда newgroup идентична group. (Конец совета пользователям.)

```
MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
IN     group          группа (handle)
IN     n              число триплексов в массиве ranges (integer)
IN     ranges         одномерный массив целых триплексов, в форме (первый
                      ранк, последний ранк, шаг) показывающей ранги в
                      group или процессы, которые должны быть включены в
                      newgroup
OUT    newgroup       новая группа полученная из указанной выше в порядке
                      определенном рангами ranks (handle)

int MPI_Group_range_incl (MPI_Group group, int n, int **ranges,
MPI_Group *newgroup)

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

Если ranges состоит из триплетов
(first1, last1, stride1), ..., (firstn, lastn, striden)
тогда newgroup состоит из последовательности процессов группы group
с ранками

Совет пользователям. Каждый вычисленный ранк должен быть допустимым ранком в group и все вычисленные ранки должны быть определены; иначе программа будет ошибочной. Заметим, что у нас могут быть значения firsti > lasti, и stridei, которые могут быть отрицательными, но не могут принимать нулевое значение. (Конец совета пользователям.)

Совет разработчикам. Функциональные возможности этой подпрограммы определяются так, чтобы быть эквивалентными к расширенному массиву диапазонов, к массиву, входящих рангов и передаче результирующего массива рангов и других параметров в MPI_GROUP_INCL. Вызов MPI_GROUP_INCL эквивалентен вызову MPI_GROUP_RANGE_INCL с каждым рангом i в ranks, расположенных тройками (i,i,1) в параметре ranges. (Конец совета разработчикам.)

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

IN	group	группа (handle)
IN	n	число элементов в массиве ранков (integer)
IN	ranges	одномерный массив целых триплексов, в форме (первый ранк, последний ранк, шаг) показывающей ранки в группе процессов, которые должны быть исключены из newgroup.
OUT	newgroup	новая группа полученная из указанной выше и сохраняющая порядок group (handle)

int MPI_Group_range_excl (MPI_Group group, int n, int **ranges, MPI_Group *newgroup)

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

Каждый вычисленный ранк должен быть допустимым ранком в group и все вычисленные ранки должны быть определены; иначе программа будет ошибочной.

Совет разработчикам. Функциональные возможности этой подпрограммы определяются так, чтобы быть эквивалентными к расширенному массиву диапазонов, к массиву, исключенных рангов и передаче результирующего массива рангов и других параметров в MPI_GROUP_EXCL. Вызов MPI_GROUP_EXCL эквивалентен вызову MPI_GROUP_RANGE_EXCL с каждым рангом i в ranks, расположенных тройками (i,i,1) в параметре ranges. (Конец совета разработчикам.)

Совет пользователям. Операции диапазона не нумеруют явно ранки, и следовательно, если это эффективно реализовано, являются более масштабируемыми. Поэтому, мы рекомендуем программистам MPI использовать их всякий раз как только возможно, поскольку высококачественные реализации будут пользоваться преимуществом этого факта. (Конец совета пользователям.)

Совет разработчикам. Операции диапазона должны выполняться, если возможно, без перечисления элементов группы, чтобы получить лучшую масштабируемость (по времени и пространству). (Конец совета разработчикам.)

5.3.3 Деструкторы групп

MPI_GROUP_FREE(group)

INOUT group группа (handle)

int MPI_Group_free(MPI_Group group *group)

MPI_GROUP_FREE(GROUP, IERROR)
INTEGER GROUP, IERROR

Эта операция помечает объект группы для освобождения. Хэндлер group вызовом устанавливается в MPI_GROUP_NULL. Любая продолжающаяся операция, использующая эту группу, будет нормально завершена.

Совет разработчикам. Она может хранить счетчик ссылок, который увеличивается при каждом вызове MPI_COMM_CREATE и MPI_COMM_DUP, и уменьшается при каждом вызове MPI_GROUP_FREE или MPI_COMM_FREE; объект группы окончательно освобождается, когда счетчик ссылок обнуляется. (Конец совета разработчикам.)

5.4 Управление коммуникаторами

Этот раздел описывает манипуляции с коммуникаторами в MPI. Операции, которые осуществляют доступ к коммуникаторам являются локальными и их выполнение не требует межпроцессных обменов. Операции, которые создают коммуникаторы являются групповыми и могут требовать межпроцессных обменов.

Совет разработчикам. Высокое качественные реализации должны сглаживать накладки, связанные с созданием коммуникаторов (для той же группы или её подмножеств) некоторыми вызовами, посредством размещения множества контекстов. (Конец совета разработчикам.)

5.4.1 Доступ к коммуникаторам

Все следующие операции являются локальными.

MPI_COMM_SIZE(comm, size)
IN comm коммуникатор (handle)
OUT size количество процессов в группе comm (integer)

int MPI_Comm_size (MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR

Целесообразность. Эта функция эквивалентна доступу к группе коммуникатора функцией MPI_COMM_GROUP (см.ниже), вычислению размера с использованием MPI_GROUP_SIZE и затем временному освобождению группы через MPI_GROUP_FREE. Однако, эта функция обычно используется также и тогда, когда это сокращение введено. (Конец замечания.)

Совет пользователям. Эта функция показывает число процессов, включаемых в коммуникатор. Для MPI_COMM_WORLD она показывает общее число имеющихся процессов (в этой версии MPI нет стандартного способа изменять количество процессов после однажды произведенной инициализации).

Этот вызов часто используется вместе со следующим вызовом, который определяет количество имеющихся совпадений для особой библиотеки или программы. При следующем вызове, MPI_COMM_RANK показывает ранк процесса, который его вызывает, в диапазоне 0,.., size-1, где size это величина, возвращаемая MPI_COMM_SIZE. (Конец совета пользователям.)

MPI_COMM_RANK(comm, rank)
IN comm коммуникатор (handle)

OUT rank ранк вызываемого процесса в группе comm (integer)

int MPI_Comm_rank (MPI_Comm comm, int *rank)

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

Целесообразность. Эта функция эквивалентна доступу к группе коммуникатора функцией MPI_COMM_GROUP (см.ниже), вычислению размера с использованием MPI_GROUP_RANK и затем временному освобождению группы через MPI_GROUP_FREE. Однако, эта функция обычно используется также и тогда, когда это сокращение введено. (Конец замечания.)

Совет пользователям. Эта функция дает ранк процесса в особой группе коммуникатора. Это полезно, как отмечено выше, в соединении с MPI_COMM_SIZE.

Многие программы написаны с использованием модели master-slave, где один процесс (такой как процесс с рангом 0) играет роль ведущего, а другой процесс служит для вычисления узлов. В этой структуре два предшествующих вызова полезны для определения роли различных процессов коммуникатора. (Конец совета пользователям.)

MPI_COMM_COMPARE(comm1, comm2, result)
IN comm1 первый коммуникатор (handle)
IN comm2 второй коммуникатор (handle)
OUT result результат (integer)

Результат равен MPI_INDENT тогда и только тогда, когда comm1 и comm2 это хэндлеры одного и того же объекта (идентичных групп с одинаковыми контекстами). MPI_CONGRUENT является результатом, если сравниваемые группы имеют идентичные элементы и их порядок; т.е. эти коммуникаторы отличаются только контекстом. MPI_SIMILAR получается, если члены группы обоих коммуникаторов одни и те же, а ранки имеют различный порядок. Во всех остальных случаях получается MPI_UNEQUAL.

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

MPI_COMM_RANK(COMM1, COMM2, RESULT, IERROR)
INTEGER COMM1, COMM2, RESULT, IERROR

5.4.2 Конструкторы коммуникаторов

Следующие функции являются групповыми и вызываются всеми процессами в группе связанной с comm.

Целесообразность. Заметим, что аспект "курицы и яйца" в MPI состоит в том, что коммуникатор необходим для создания нового коммуникатора. Базовым коммуникатором для всех коммуникаторов MPI, определяемых в MPI, является MPI_COMM_WORLD. Эта модель была принята после бурного обсуждения и была выбрана для улучшения надежности программ, написанных в MPI. (Конец замечания.)

MPI_COMM_DUP(comm, newcomm)
IN comm коммуникатор (handle)
OUT newcomm скопированный коммуникатор comm (handle)

int MPI_Comm_dup (MPI_Comm comm, MPI_Comm *newcomm)

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR

MPI_COMM_DUP дублирует существующий коммуникатор comm со

скрытыми атрибутами копируемыми или некопируемыми (согласно определению каждого кешируемого атрибута). В newcomm возвращается новый коммуникатор с такой же группой, такой же скрытой информацией, но с новым контекстом.

Совет пользователям. Эта операция используется, чтобы обеспечить вызов параллельной библиотеки с дублированием пространства связи, которое имеет такие же свойства как и оригинальный коммуникатор. Она включает любые атрибуты (см.ниже) и топологии (см.главу 6). Этот вызов является правильным даже если существует задержка двухточечных связей, включающих коммуникатор comm. Типичная программа может включать MPI_COMM_DUP в начале параллельного вызова и MPI_COMM_FREE этого дублируемого коммуникатора в конце вызова. Возможны также другие модели управления коммуникатором.

Этот вызов применим как к интра- так и к интер-коммуникаторам. (Конец совета пользователям.)

Совет разработчикам. Не нужно копировать групповую информацию, а только добавить новую ссылку и увеличить счетчик ссылок. Копирование на запись может быть использовано для кешированой информации. (Конец совета разработчикам.)

MPI_COMM_CREATE(comm, group, newcomm)

IN	comm	коммуникатор (handle)
IN	group	группа, которая является подмножеством группы коммуникатора comm (handle)
OUT	newcomm	новый коммуникатор (handle)

```
int MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm
*newcomm)

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
        INTEGER COMM, GROUP, NEWCOMM, IERROR
```

Эта функция создает новый коммуникатор newcomm с группой связи определенной group и новым контекстом. Некешируемая информация передается из comm в newcomm. Функция возвращает MPI_COMM_NULL для процессов, которые не принадлежат group. Вызов является ошибочным, если не все аргументы group имеют одни и те же значения, или если group не является подмножеством группы связанной с comm. Заметим, что вызов выполняется всеми процессами в comm, даже если они не относятся к новой группе.

Совет пользователям. MPI_COMM_CREATE обеспечивает значения подмножества группы процессов с целью разделения MIMD вычислений, с раздельным пространством связи. newcomm, который получается после MPI_COMM_CREATE может быть использован в последующих вызовах MPI_COMM_CREATE (или других конструкторах коммуникаторов), способствуя разделению вычислений на параллельные подвычисления. Наиболее общее обслуживание обеспечивается MPI_COMM_SPLIT (см.ниже).

Этот вызов применим только в интра-коммуникаторах. (Конец совета пользователям.)

Совет разработчикам. Так как все процессы, вызывающие MPI_COMM_DUP или MPI_COMM_CREATE обеспечивают такой же group аргумент, то теоретически возможно получить уникальный групповой контекст без обменов. Однако, локальное исполнение этих функций требует использования большего пространства имени контекста и понижает качество проверки ошибки. Реализации могут вносить различные компромиссы в эти конфликтные ситуации.

Главное: Если новые коммуникаторы созданы без синхронизации

включаемых процессов, тогда система связи должна справляться с сообщениями приходящими в контексте, который еще не расположен в принимающем процессе. (Конец совета разработчикам.)

```
MPI_COMM_SPLIT(comm, color, kew, newcomm)
IN     comm          коммуникатор (handle)
IN     color         управление подмножеством передачи (integer)
IN     key           управление ранком передачи (integer)
OUT    newcomm       новый коммуникатор (handle)

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

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

Это функция делит группу ассоциированную с comm на непересекающиеся подгруппы, по одной для каждого значения color. Каждая подгруппа содержит все процессы с одинаковым цветом. Внутри каждой подгруппы процессы выстраиваются в порядке определяемом значением аргумента key, с разрушенными связями согласно их ранку в старой группе. Новый коммуникатор создается для каждой подгруппы и возвращается в newcomm. Процесс может иметь значение цвета MPI_UNDEFINED, в этом случае в newcomm возвращается MPI_COMM_NULL. Это групповой вызов, но каждому процессу разрешается давать различные значения color и key.

Целесообразность. Вызов MPI_COMM_CREATE(comm, group, newcomm) эквивалентен вызову MPI_COMM_SPLIT(comm, color, kew, newcomm), где все члены group имеют color = 0 и key = ранку в group, и все процессы, которые не являются членами group, имеют color = MPI_UNDEFINED. Функция MPI_COMM_SPLIT позволяет более общее разделение группы на одну или несколько подгрупп с необязательным изменением порядка. (Конец замечания.)

Совет пользователям. Это чрезвычайно мощный механизм для разделения отдельной коммуникационной группы процессов на k цветов, с k неявно выбранным пользователем (количество цветов, утвержденным для всех процессов). Все получившиеся коммуникаторы будут неперекрывающимися. Такое деление может быть полезно для определения иерархии вычислений, таких как в многомерных решётках или линейной алгебры.

Многочисленные вызовы MPI_COMM_SPLIT могут быть использованы, чтобы обойти требование, при котором любой вызов не даёт пересекающихся коммуникаторов (каждый процесс имеет только один цвет на вызов). В этом случае, может быть создано множественно пересекающихся структур связи. В таких расщепляющих операциях поощряется творческое использование color и key.

Заметим, что для фиксированного цвета, ключи могут быть различны. Обязанностью MPI_COMM_SPLIT является сортировка процессов в возрастающем порядке согласно этому ключу и разрыв связей в соответствующем случае. Если все ключи специфицируются одним и тем же способом, тогда все процессы данного цвета будут иметь относительный порядок ранков такой же какой они имели в родительской группе. (Вообще, они будут иметь различные ранки.)

По существу, присвоение ключу значения ноль для всех процессов данного цвета означает, что не имеет значения какой будет порядок ранков процессов в новом коммуникаторе.

Этот вызов применим только в интра-коммуникаторах. (Конец совета пользователям.)

5.4.3 Деструкторы коммуникаторов

```
MPI_COMM_FREE(comm)
INOUT         comm          коммуникатор, который должен быть уничтожен
                            (handle)

int MPI_Comm_free (MPI_Comm *comm)

MPI_COMM_FREE(COMM, IERROR)
        INTEGER COMM, IERROR
```

Совет пользователям. Эта групповая операция помечает объект связи для освобождения. Хэндлер устанавливается в MPI_COMM_NULL. Этот объект действительно считается освобожденным, если на него не существует активных ссылок. Связанную группу также можно освободить, если на нее нет активных ссылок.

Этот вызов применим в интра- и интер-коммуникаторах. (Конец совета пользователям.)

Совет разработчикам. Механизм счета ссылок используется следующим образом: счетчик ссылок увеличивается при каждом вызове MPI_COMM_DUP, и уменьшается при каждом вызове MPI_COMM_FREE. Объект окончательно освобождается, когда счетчик обнуляется.

Все-таки коллектив предвидел, что эта операция будет реализована как локальная, хотя отладочная версия библиотеки MPI может выбираться по синхронизации. (Конец совета разработчикам.)

5.5 Мотивирующие примеры

5.5.1 Текущая практика #1

Пример #1a:

```
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();
 }
```

Пример #1a ничего не делающая программа, которая легально инициализирует сама себя, и ссылается на "весь" коммуникатор и выводит сообщение. Она также легально сама себя завершает. Этот пример не подразумевает, что MPI сам поддерживает printf-подобную связь.
Пример #1b (предполагается, что size является четным):

```
main(int argc, char **argv)
 {
       int me, size;
       int SOME_TAG = 0;
       ...
       MPI_Init(&argc, &argv);

       MPI_Comm_rank(MPI_COMM_WORLD, &me);                    /* local */
       MPI_Comm_size(MPI_COMM_WORLD, &size);                  /* local */
```

```
        if((me %2) == 0)
                {       /* послать, если это не процесс нумерованный выше */
                if((me + 1) < size)
                        MPI_Send(..., me+1, SOME_TAG, MPI_COMM_WORLD);
                }
        else
                MPI_Recv(..., me-1, SOME_TAG, MPI_COMM_WORLD);
        ...
        MPI_Finalize();
 }
```

Пример #1b схематично иллюстрирует обмен сообщением между "четным" и "нечетным" процессами во "всеобщем" коммуникаторе.

5.5.2 Текущая практика #2

```
main(int argc, char **argv)
 {
        int me, count;
        void *data;
        ...

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &me);
        if (me == 0)
                {
                /* принять вводимое, создать буфер "data" */
                ...
                }
        MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
        ...
        MPI_Finalize();
 }
```

Этот пример показывает использование групповых обменов.

5.5.3 (Приблизительная) Текущая практика #3

```
main(int argc, char **argv)
 {
        int me, count, count2;
        void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
        MPI_Group MPI_GROUP_WORLD, grprem;
        MPI_Comm commslave;
        static int ranks[] = {0};
        ...
        MPI_Init(&argc, &argv);
        MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
        MPI_Comm_rank(MPI_COMM_WORLD, &me);                                     /* local
*/
        MPI_Comm_excl(MPI_COMM_WORLD, 1, ranks, &grprem); /* local */
        MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

        if(me != 0)
                {
                /* вычисления на подчиненном */
                ...
                MPI_Reduce(send_buf, recv_buff, MPI_INT, MPI_SUM, 1,
                                commslave);
                ...
                }
        /* нулевой попадает сюда немедленно, остальные позже... */
        MPI_Reduce(send_buf2, recv_buff2, count2, MPI_INT, MPI_SUM, 0,
                MPI_COMM_WORLD);
        MPI_Comm_free(&commslave);
        MPI_Group_free(&MPI_GROUP_WORLD);
```

```
       MPI_Group_free(&grprem);
       MPI_Finalize();
 }
```

Этот пример иллюстрирует, как создается группа состоящая из всех кроме нулевого, процессов "полной" группы и, затем, как формируется коммуникатор (commslave) для этой новой группы. Новый коммуникатор используется в групповом вызове, и все процессы выполняют групповой вызов в контексте MPI_COMM_WORLD. Этот пример иллюстрирует, как два коммуникатора (имеющие явные контексты) защищают связь. Таким образом, связь в MPI_COMM_WORLD изолируется от связи в commslave, и наоборот.

Вообще, "сохранность группы" достигается через коммуникаторы потому, что явные контексты внутри коммуникаторов сделаны таким образом, чтобы быть уникальными в любом процессе.

5.5.4 Пример #4

Следующий пример предназначен для иллюстрации "безопасности" двухточечных и групповых обменов. MPI гарантирует, что отдельный коммуникатор может иметь безопасные двухточечные и групповые связи.

```
#define       TAG_ARBITRARY 12345
#define       SOME_COUNT       50

main(int argc, char **argv)
 {
       int me;
       MPI_Request request[2];
       MPI_Status status[2];
       MPI_Group MPI_GROUP_WORLD, subgroup;
       int ranks[] = {2, 4, 6, 8};
       MPI_Comm the_comm;
       ...
       MPI_Init(&argc, &argv);
       MPI_Comm_group(MPI_GROUP_WORLD, &MPI_GROUP_WORLD);

       MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup);/*local*/
       MPI_Group_rank(subgroup, &me);                          /* local */

       MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
       if(me != MPI_UNDEFINED)
              {
              MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE,
                                   TAG_ARBITRARY, the_comm, request);
              MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4,
                                        TAG_ARBITRARY, the_comm, request+1);
              }
       for(i = 0; i < SOME_COUNT, i++)
              MPI_Reduce(..., the_comm);
       MPI_Waitall(2, request, status);

       MPI_Comm_free(&the_comm);
       MPI_Group_free(&MPI_GROUP_WORLD);
       MPI_Group_free(&subgroup);
       MPI_Finalize();
 }
```

5.5.5 Библиотечный пример #1

Основная программа:

```
main(int argc, char **argv)
 {
       int done = 0;
       user_lib_t *libh_a, *libh_b;
       void *dataset1, *dataset2;
```

```
        ...
        MPI_Init(&argc, &argv);
        ...
        init_user_lib(MPI_COMM_WORLD, &libh_a);
        init_user_lib(MPI_COMM_WORLD, &libh_b);
        ...
        user_start_op(libh_a, dataset1);
        user_start_op(libh_b, dataset2);
        ...
        while(!done)
                {
                /* работа */
                ...
                MPI_Reduce(..., MPI_COMM_WORLD);
                ...
                /* смотри если сделано */
                ...
                }
        user_end_op(libh_a);
        user_end_op(libh_b);

        uninit_user_lib(libh_a);
        uninit_user_lib(libh_b);
        MPI_Finalize();
 }
```

Подпрограмма инициализации библиотеки пользователя:

```
void init_user_lib(MPI_Comm *comm, user_lib_t **handle)
 {
        user_lib_t *save;

        user_lib_initsave(&save); /* local */
        MPI_Comm_dup(comm, &(save -> comm);

        /* другие блоки */
        ...
        *handle = save;
 }
```

Программа старта пользователя:

```
void user_start_op(user_lib_t *handle, void *data)
 {
        MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
        MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
 }
```

Программа разрыва связи пользователя:

```
void user_end_op(user_lib_t *handle)
 {
        MPI_Status *status;
        MPI_Wait(handle -> isend_handle, status);
        MPI_Wait(handle -> irecv_handle, status);
 }
```

Программа очистки объекта пользователя:

```
void uninit_user_lib(user_lib_t *handle)
 {
        MPI_Comm_free(&(handle -> comm));
        free(handle);
 }
```

5.5.6 Библиотечный пример #2

Основная программа:

```
main(int argc, char **argv)
 {
       int ma, mb;
       MPI_Group MPI_GROUP_WORLD, group_a, group_b;
       MPI_Comm comm_a, comm_b;
       static int list_a[] = {0, 1};
#if defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
       static int list_b[] = {0, 2, 3};
#else /* EXAMPLE_2A */
       static int list_b[] = {0, 2}
#endif
       int size_list_a = sizeof(list_a)/sizeof(int);
       int size_list_b = sizeof(list_b)/sizeof(int);
       ...
       MPI_Init(&argc, &argv);
       MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
       MPI_Group_incl(MPI_COMM_WORLD, size_list_a, list_a, &group_a);
       MPI_Group_incl(MPI_COMM_WORLD, size_list_b, list_b, &group_b);

       MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
       MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

       MPI_Comm_rank(comm_a, &ma);
       MPI_Comm_rank(comm_b, &mb);

       if(ma != MPI_UNDEFINED)
              lib_call(comm_a);
       if(mb != MPI_UNDEFINED)
              {
              lib_call(comm_b);
              lib_call(comm_b);
              }

       MPI_Comm_free(&comm_a);
       MPI_Comm_free(&comm_b);
       MPI_Group_free(&group_a);
       MPI_Group_free(&group_b);
       MPI_Group_free(&MPI_GROUP_WORLD);
       MPI_Finalize();
 }
```

Библиотека:

```
void lib_call(MPI_Comm comm)
 {
       int me, done = 0;
       MPI_Comm_rank(comm, &me);
       if(me == 0)
              while(!done)
                     {
                     MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm);
                     ...
                     }
       else
              {
              /* работа */
              MPI_Send(..., 0, ARBITRARY_TAG, comm);
              ...
              }
#ifdef EXAMPLE_2C
       /* include (resp, exclude) for safety (resp, no safety): */
       MPI_Barrier(comm);
#endif
 }
```

Описанный выше пример в действительности представляет три примера, в зависимости от того, действительно ли мы включаем ранк 3 в list_b, и действительно ли синхронизация включается в lib_call. Этот пример иллюстрирует, что, несмотря на контексты, последующие вызовы в lib_call с одними и теми же контекстами не нужно оберегать друг от друга (в разговорной речи - "обратное маскирование"). Если присутствует MPI_Barrier реализуется такая безопасность. Это демонстрирует то, что библиотеки должны быть написаны аккуратно даже с контекстами. Когда исключается ранк 3, тогда не нужна синхронизация для того, чтобы иметь безопасность от обратного маскирования.

Алгоритмы похожие на "reduce" и "allreduce" имеют достаточно сильные исходные свойства избирательности такие, какие они имеют по своей природе (нет обратного маскирования), при условии, что MPI обеспечивает базисные гарантии. Таким образом, есть многократные обращения к типичному алгоритму распространение по дереву с тем же корнем или разными корнями (см. [27]). Здесь мы полагаемся на две гарантии MPI: попарное упорядочивание сообщений между процессами в одних и тех же контекстах, и выбор источника, - стирание этих особенностей уничтожает гарантию того, что обратное маскирование не произойдёт.

Алгоритмы, которые пытаются сделать недетерминированы передачи или другие вызовы, включающие wild-card-операции, обычно не имеют хороших свойств детерминированных реализаций "reduce", "allreduce" и "broadcast". Такие алгоритмы должны использовать монотонно увеличивающиеся признаки (внутри диапазона коммуникатора), чтобы сохранить содержимое правильным.

Все вышеупомянутое предполагает "групповые вызовы" с использованием двухточечных операций. Реализации MPI могут или не могут выполнять групповые вызовы использующие двухточечные операции. Эти алгоритмы используются, чтобы проиллюстрировать результат правильности и безопасности, независящий от того, как MPI реализует свои групповые вызовы. Смотри также раздел 5.8.

5.6 Интер-связь

Эта часть вводит понятие интер-связи и описывает ту часть MPI, которая её поддерживает. Она описывает поддержку написания программ, которые содержат серверы пользовательского уровня.

Все двухточечные обмены описанные до сих пор включали обмены между процессами, которые являлись членами одной и той же группы. Этот тип обменов назывался "интра-связью" и используемый коммуникатор назывался "интра-коммуникатором", как было ранее описано в этой главе.

В модульных и многодисциплинарных приложениях различные группы процессов исполняются определенными модулями и процессы, внутри различных модулей, соединяются с другими процессами при помощи конвейера или более общего графа. В этих приложениях наиболее естественным способом для процесса, который специфицирует целевой процесс, является использование ранка принимающего процесса внутри своей группы. В тех приложениях, которые содержат внутренние серверы пользовательского уровня, каждый сервер может быть группой процессов, которая обеспечивает обслуживание одного или более клиентов, и где каждый клиент может быть группой процессов, которая использует обслуживание одного или более серверов. В этих приложениях, к тому же, более естественно определять приёмный процесс при помощи его ранка внутри целевой группы. Этот тип связи называется "интер-связью", а используемый коммуникатор называется "интер-коммуникатором", как было отмечено ранее.

Интер-связь это двухточечная связь между процессами в различных группах. Группа, содержащая процесс, который инициирует операцию интер-связи, называется "локальной группой", т.е. передатчиком при передаче и приемником при приеме. Группа, содержащая целевой процесс, называется "удаленной группой", т.е. приемником при передаче и передатчиком при приеме. Как и в интра-связи целевой процесс специфицируется парой (communicator, rank). В отличие от интра-связи, ранк связан со второй, удаленной

группой.

Все конструкторы интер-коммуникаторов блокирующие и требуют, чтобы локальная и удаленная группы были разъединены во избежании зависания.

Здесь представлены основные свойства интер-связи и интер-коммуникаторов:

- Синтаксис двухточечной связи одинаков как для интер-, так и для интра-связи. Один и тот же коммуникатор может быть использован как для операций передачи, так и для операций приема.

- Целевой процесс адресуется своим ранком в удаленной группе как при передаче, так и при приеме.

- Гарантируется, что обмены, использующие интер-коммуникатор, не будут конфликтовать с любыми обменами, которые используют другой коммуникатор.

- Интер-коммуникатор не может быть использован для групповой связи.

- Коммуникатор будет обеспечивать или интра- или интер-связь, но никогда обе сразу.

Программу MPI_COMM_TEST_INTER можно использовать для определения: является ли коммуникатор интер- или интра-коммуникатором. Интер-коммуникаторы могут быть использованы, как аргументы в некоторых программах доступа к коммуникатору. Интер-коммуникаторы не могут быть использованы, как входные данные в некоторых программах конструкторов для интра-коммуникаторов (например, MPI_COMM_CREATE).

Совет разработчикам. При двухточечной связи коммуникаторы могут быть представлены записью содержащей:
group
send_context
receive_context
source

Для интер-коммуникаторов group описывает удаленную группу, а source является ранком процесса в локальной группе. Для интра-коммуникаторов group является группой коммуникатора, а source - ранком процесса в этой группе, и send_context идентичен receive_context. Группа представлена таблицей перевода ранка в абсолютный адрес.

Интер-коммуникатор нельзя серьёзно обсуждать без рассмотрения процессов как в локальной, так и в удаленной группе. Представим процесс P в группе P, которая имеет коммуникатор CP и процесс Q в группе Q, которая имеет коммуникатор CQ. (Заметим, что P и Q не должны быть явными.) Тогда

- CP.group описывает группу Q и CQ.group описывает группу P.

- CP.send_context = CQ.receive_context и контекст уникален в Q; CP.receive_context = CQ.send_context контекст уникален в P.

- CP.source является ранком P в P и CQ.source является ранком Q в Q.

Предположим, что P посылает сообщение Q, используя интер-коммуникатор. Тогда P использует таблицу group, чтобы найти абсолютный адрес Q; к сообщению добавляются source и send_context.

Предположим, что Q получает сообщение с явным аргументом

источника, используя интер-коммуникатор. Тогда Q сравнивает receive_context с контекстом сообщения и аргумент источника с источником сообщения.

Такой алгоритм применим так же и для интра-коммуникаторов.

Для того чтобы поддержать доступы и конструкторы интер-коммуникаторов необходимо добавить в эту модель дополнительные структуры, которые сохраняют информацию о группе локальной связи и дополнительных сохраняющих контекстах. (Конец совета разработчикам.)

5.6.1 Доступ к интер-коммуникаторам

```
MPI_COMM_TEST_INTER(comm, flag)
IN     comm          коммуникатор (handle)
OUT    flag          (logical)

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
       INTEGER COMM, IERROR
       LOGICAL FLAG
```

Эта локальная программа позволяет вызывающему процессу определять: является ли коммуникатор интер- или интра-коммуникатором. Если она возвращает true, то это интер-коммуникатор, иначе - false.

Когда интер-коммуникатор используется как входной аргумент для функций доступа к коммуникатору, описанных выше в разделе об интра-связи, следующая таблица описывает его поведение.

```
MPI_COMM_* Значение функции
(в режиме Интер-связи)

MPI_COMM_SIZE
MPI_COMM_GROUP
MPI_COMM_RANK

возвращает размер локальной группы
возвращает локальную группу
возвращает ранк в локальной группе
```

Кроме того, операция MPI_COMM_COMPARE верна для интер-коммуникаторов. Оба коммуникатора должны быть интра- или интер-коммуникаторами, иначе в результате будет получено MPI_UNEQUAL. При правильном сравнении соответствующих локальных и удаленных групп получаем MPI_CONGRUENT и MPI_SIMILAR. В частности, возможно, результатом будет MPI_SIMILAR, потому что локальные или удаленные группы были схожи, но не идентичны.

Следующие функции определяют параметры удаленной группы интер-коммуникатора:

Все операции локальны.

```
MPI_COMM_REMOTE_SIZE(comm, flag)
IN     comm          интер-коммуникатор (handle)
OUT    size          количество процессов в удаленной группе
                     коммуникатора comm (integer)

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
       INTEGER COMM, SIZE, IERROR

MPI_COMM_REMOTE_GROUP(comm, group)
IN     comm          интер-коммуникатор (handle)
OUT    group         удаленная группа соответствующая коммуникатору comm
                     (handle)
```

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
       INTEGER COMM, GROUP, IERROR
```

Целесообразность. Симметричный доступ как к локальной так и к удаленной группам интер-коммуникатора является важным, поэтому обеспечивается такая функция как MPI_COMM_REMOTE_SIZE. (Конец замечания.)

5.6.2 Операции над интер-коммуникаторами

Этот раздел представляет четыре блокирующих операции для интер-коммуникатора. MPI_INTERCOMM_CREATE используется, чтобы связать два интра-коммуникатора в один интер-коммуникатор; функция MPI_INTERCOMM_MERGE создает интра-коммуникатор слиянием локальной и удаленной групп интер-коммуникатора. Функции MPI_COMM_DUP и MPI_COMM_FREE предварительно представляют и соответственно дублируют или освобождают интер-коммуникатор.

Частичное совпадение (пересечение) локальной и удаленной групп, которые включаются в интер-коммуникатор, запрещено. Если есть пересечение, тогда программа ошибочна и возможно зависание. (Если процесс многопотоковый и MPI вызывает блок для потока, а не для процесса, тогда может поддерживаться "двойное членство". Тогда на совести пользователя остается удостовериться, что вызовы от имени двух "функций" процесса выполняются двумя независимыми потоками.)

Функция MPI_INTERCOMM_CREATE может быть использована для создания интер-коммуникатора из двух существующих интра-коммуникаторов в следующей ситуации: как минимум один выбранный член из каждой группы ("лидер группы") имеет возможность связываться с выбранным членом другой группы; т.е. "парный" коммуникатор существует там, где находятся оба лидера, и каждый лидер знает ранк другого лидера в этом парном коммуникаторе (два лидера могут быть одним и тем же процессом). Кроме того, члены каждой группы знают ранк своих лидеров.

Создание одного интер-коммуникатора из двух интра-коммуникаторов требует отдельных групповых операций в локальной группе и в удаленной группе, также как и двухточечная связь между процессом в локальной группе и процессом в удаленной группе.

В стандартных реализациях MPI (со статическим распределением процессов при инициализации) коммуникатор MPI_COMM_WORLD (предпочтительно для этого использовать его копию) может быть этим парным коммуникатором. В динамических реализациях MPI, где, например, процесс может порождать новые дочерние процессы в течении выполнения MPI, родительский процесс может быть "мостом" между старой областью связи и новым пространством связи, которое включает родительские и порождённые процессы.

Применение функций топологии, описанных в главе 6, не возможно для интра-коммуникаторов. Пользователи, которым требуется эта возможность должны использовать MPI_INTERCOMM_MERGE, чтобы построить интра-коммуникатор, затем применить возможности граф или декартовой топологии к этому интра-коммуникатору, создавая подходящий тополого-ориентированный интра-коммуникатор. В противном случае, может быть разумно изобрести свое собственное применение механизмов топологии, без потери общности.

```
MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm,
                remote_leader, tag, newintercomm)
IN     local_comm           локальный интра-коммуникатор (handle)
IN     local_leader         ранк локального лидера группы в локальном
                            интра-коммуникаторе local_comm (integer)
IN     peer_comm            "peer" интра-коммуникатор; значимо только для
                            local_leader (handle)
IN     remote_leader        ранк удаленного лидера группы в peer_comm;
                            значимо только для local_leader (integer)
```

```
IN     tag                  "сохраняющийся" тэг (integer)
OUT    newintercomm         новый интер-коммуникатор (handle)

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm
*newintercomm)

MPI_COMM_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM,
              REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR)
       INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
              TAG, NEWINTERCOMM, IERROR
```

Этот вызов создает интер-коммуникатор. Это есть группа надстроенная над объединением локальной и удаленной групп. Процессы должны обеспечивать идентичность аргументов local_comm и local_leader внутри каждой группы. Произвольные значения запрещены для remote_leader, local_leader и tag.

Этот вызов использует двухточечную связь с коммуникатором peer_comm и с признаком tag между лидерами. Таким образом, нужно позаботиться о том, чтобы не было зависающей связи на peer_comm, который может пересекаться с этой обменом.

> Совет пользователю. Рекомендуется использовать исходный парный коммуникатор, такой как копия MPI_COMM_WORLD, чтобы избежать неприятностей с парными коммуникаторами. (Конец совета пользователю.)

```
MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)
IN     intercomm            интер-коммуникатор (handle)
IN     high                 (logical)
OUT    newintracomm         новый интра-коммуникатор (handle)

int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm
*newintracomm)

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
       INTEGER INTERCOMM, INTRACOMM, IERROR
       LOCAL HIGH
```

Эта функция создает интра-коммуникатор из объединения двух групп, которые связаны с intercomm. Все процессы должны обеспечивать одно и тоже значение high внутри каждой из двух групп. Если процессы в одной группе обеспечивают значение high = false, а процессы в другой группе обеспечивают значение high = true, тогда в объединении "низкая" группа стоит перед "высокой". Если все процессы обеспечивают один и тот же аргумент high, тогда порядок объединения произвольный. Этот вызов является блокирующим и совместным внутри объединения двух групп.

> Совет разработчикам. Реализация MPI_INTERCOMM_MERGE, MPI_COMM_FREE и MPI_COMM_DUP схожа с реализацией MPI_INTERCOMM_CREATE, за исключением того, что частности контекстов во входном интер-коммуникаторе используются для связи между лидерами групп, а не контекстами внутри мостового коммуникатора. (Конец совета разработчикам.)

5.6.3 Примеры интер-связей

Пример 1: Трехгрупповой "Конвейер"

Группы 0 и 1 соединены. Группы 1 и 2 соединены. Следовательно, группе 0 требуется один интер-коммуникатор, группе 1 требуется два интер-коммуникатора и группе 2 требуется один интер-коммуникатор.

```
main(int argc, char **argv)
 {
       MPI_Comm myComm; /* интра-коммуникатор локальной подгруппы*/
```

```
       MPI_Comm myFirstComm; /* интер-коммуникатор */
       MPI_Comm mySecondComm; /* второй интер-коммуникатор (только
                                         для 1 группы) */
       int membershipKey;
       int rank;

       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &rank);

       /* Код пользователя должен генерировать membershipKey в ряду
[0, 1, 2] */
       membershiKey = rank % 3;

       /* Построение интра-коммуникатора для локальной подгруппы */
       MPI_Comm_split(MPI_COMM_WORLD, membershiKey, rank, &myComm);

       /* Построение интер-коммуникаторов. Tags сложнокодируемы */
       if (membershipKey == 0)
              {                     /* Группа 0 связывается с группой 1. */
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                           1, 1, &myFirstComm);
              }
       else if (membershipKey == 1)
              {             /* Группа 1 связывается с группами 0 и 2. */
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                  0, 1, &myFirstComm);
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                  2, 12, &mySecondComm);
              }
       else if (membershipKey == 2)
              {                     /* Группа 2 связывается с группой 1. */
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                  1, 12, &myFirstComm);
              }

              /* Работай ... */

       switch(membershipKey) /* освобождаем коммуникаторы */
              {
              case 1:
                     MPI_Comm_free(&mySecondComm);
              case 0:
              case 2:
                     MPI_Comm_free(&myFirstComm);
                     break;
              }
       MPI_Finalize();
 }
```

Пример 2: "Кольцо" из трёх групп

Группы 0 и 1 связаны. Группы 1 и 2 связаны. Группы 0 и 2 связаны. Следовательно, каждая группа требует по два интер-коммуникатора.

```
main(int argc, char **argv)
 {
       MPI_Comm myComm; /* интра-коммуникатор локальной подгруппы*/
       MPI_Comm myFirstComm; /* интер-коммуникаторы */
       MPI_Comm mySecondComm; /* интер-коммуникаторы */
       MPI_Status status;
       int membershipKey;
       int rank;

       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_KEY_WORLD, &rank);
       ...
```

```
       /* Код пользователя должен генерировать membershipKey в ряду
[0, 1, 2] */
       membershiKey = rank % 3;

       /* Построение интра-коммуникатора для локальной подгруппы */
       MPI_Comm_split(MPI_COMM_WORLD, membershiKey, rank, &myComm);

       /* Построение интер-коммуникаторов. Tags сложнокодируемы */
       if (membershipKey == 0)
              {             /* Группа 0 связывается с группами 1 и 2 */
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                    1, 1, &myFirstComm);
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                    2, 2, &mySecondComm);
              }
       else if (membershipKey == 1)
              {             /* Группа 1 связывается с группами 0 и 2 */
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                    0, 1, &myFirstComm);
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                    2, 12, &mySecondComm);
              }
       else if (membershipKey == 2)
              {             /* Группа 2 связывается с группами 0 и 1 */
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                    0, 2, &myFirstComm);
              MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD,
                                                    1, 12, &mySecondComm);
              }

       /* Ну, сделай что-нибудь ... */

       /* За тем коммуникаторы освобождаются перед завершением...*/
       MPI_Comm_free(&myFirstComm);
       MPI_Comm_free(&mySecondComm);
       MPI_Comm_free(&myComm);
       MPI_Finalize();
 }
```

Пример 3: Построение именного сервиса

Следующие процедуры являются примером процесса, с помощью которого пользователь может построить именной сервис, если реализация MPI сбоит, тогда это обеспечивается через один или более специально определенных коммуникаторов и соответствующих лидеров. После того как все процессы MPI выполнят MPI_INIT, каждый процесс вызывает эталонную функцию Init_server(), определенную ниже. Затем, если в new_world возвращается NULL, то этот процесс, требует реализовать функцию сервера в реактивном цикле Do_server(). Иначе каждый просто делает предписанные ему вычисления, используя new_world как новый эффективный "глобальный" коммуникатор. Сконструированный процесс вызывает Undo_Server(), чтобы избавиться от сервера, когда он не нужен.

Особенности этого подхода:

¦Поддержка множества именных серверов
¦Способность специфицировать процесс в диапазоне имен пользователя
¦Способность делать серверы какие требуется - переходящие и идущие.

```
#define INIT_SERVER_TAG_1               666
#define UNDO_SERVER_TAG_1               777

static int server_key_val;
static int server_key_init = 0;

int Init_server(peer_comm, rank_of_server, server_comm, new_world);
MPI_Comm peer_comm;
```

```
int rank_of_server;
MPI_Comm *server_comm;
MPI_Comm *new_world; /* новый эффективный мир, sans server */
 {
       MPI_Comm temp_comm, lone_comm;
       MPI_Group peer_group, temp_group;
       int rank_in_peer_comm, size, color, key = 0;
       int peer_leader,peer_leader_rank_in_temp_comm;

       MPI_Comm_rank(peer_comm, &rank_in_peer_comm);
       MPI_Comm_size(peer_comm, &size);

       /* Управление атрибутами для server_comm */
       int handle_copy_fn(); /* copy a handle per draft spec. */

       if ((size < 2) || (0 > rank_of_server) ||
(rank_of_server>=size))
              return (MPI_ERR_OTHER);

       /* создание двух коммуникаторов расщеплением peer_comm
       в процессе сервера и или каком-нибудь еще */

       peer_leader = (rank_of_server + 1) % size; /* произвольный
выбор */
       if (( color = (rank_in_peer_comm == rank_of_server)))
              {
              MPI_Comm_split(peer_comm, color, key, &lone_comm);
              MPI_Intercomm_create(0, lone_comm, peer_leader,
                     peer_comm, INIT_SERVER_TAG_1, server_comm);
              MPI_Comm_free(&lone_comm);
              *new_world = (MPI_Comm) 0;
              }
       else
              {
              MPI_Comm_Split(peer_comm, color, key, &temp_comm);
              MPI_Comm_group(peer_comm,&peer_group);
              MPI_Comm_group(temp_comm,&temp_group);
              MPI_Group_translate_ranks(peer_group, 1, &peer_leader,
temp_group, &peer_leader_rank_in_temp_comm);
              MPI_Intercomm_create(peer_leader_rank_in_temp_comm,
temp_comm, rank_of_server, peer_comm,INIT_SERVER_TAG_1,
server_comm);
              /* присоединить атрибут связи new_world к server_comm */

              if(!sever_key_init++); /* КРИТИЧЕСКАЯ СЕКЦИЯ
                                                  ДЛЯ МУЛЬТИСВЯЗЕЙ */
                     {
                     /* получение имени локального процесса
       для сервера keyval */
                     MPI_Attr_keyval_create(handle_copy_fn, NULL,
&server_keyval, NULL);
                     }

              *new_world = temp_comm;

              /* Кешировать handle интра-коммуникатора в
интер-коммуникатор*/
              MPI_Attr_put(server_comm, server_keyval,
(void*)(*new_world));
       }

       return (MPI_SUCCESS);
 }
```

Текущий процесс сервера будет передавать управление следующей программе:

```
int Do_server(server_comm)
MPI_Comm server_comm;
 {
       void init_queue();
       int en_queue(), de_queue(); /* сохранить триплексы для
                     последующего сравнения (fns не показывать) */
       MPI_Comm comm;
       MPI_Status;
       int client_tag, client_source;
       int client_rank_in_new_world, pairs_rank_in_new_world;
       int buffer[10], count = 1;

       void *queue;
       init_queue(&queue);

       for (;;)
              {
              MPI_Recv(buffer, count, MPI_INT, MPI_ANY_SOURCE,
                     MPI_ANY_TAG, server_comm, &status);
                     /* принять от любого клиента*/

              /* определение клиента */
              client_tag = status.MPI_TAG;
              client_source = status.MPI_SOURCE;
              client_rank_in_new_world = buffer[0];

              if (client_tag == UNDO_SERVER_TAG_1) /* клиент, который
                                                    уничтожает сервер */
                     {
                     while (de_queue(queue, MPI_ANY_TAG,
                     &pairs_rank_in_new_world, &pairs,_rank_in_server))
                            ;
                     MPI_Intercomm_free(&server_comm);
                     break;
                     }
              if (de_queue(queue, client_tag, &pairs_rank_in_new_world,
                     &pairs_rank_in_server))
                     {                          /* matched pair with same tag,
                                  tell them about each other! */
                     buffer[0] = pairs_rank_in_new_world;
                     MPI_Send(buffer, 1, MPI_INT, client_src, client_tag,
                                                server_comm);

                     buffer[0] = client_rank_in_new_world;
                     MPI_Send(buffer, 1, MPI_INT, pairs_rank_in_server,
                                                client_tag, server_comm);
                     }
              else
                     en_queue(queue, client_tag, client_source,
                                                       client_rank_in_new_world);
              }
 }
```

Отдельный процесс будет нести ответственность за окончание работы сервера, когда он больше не требуется. Обращение к Undo_server приведет к завершению работы сервера.

```
int Undo_server(server_comm)              /* пример клиента, который кончает
                                          сервер */
MPI_Comm *server_comm;
 {
       int buffer = 0;
       MPI_Send(&buffer, 1, MPI_INT, 0, UNDO_SERVER_TAG_1,
       *server_comm);
       MPI_intercomm_free(server_comm);
 }
```

Здесь представлены функциональные возможности изначально предполагавшиеся как часть MPI, но определенные в последнем стандарте как библиотечный вызов. Следующее является блокирующее обслуживание по имени для интер-связи, с такими же семантическими ограничениями как в MPI_Intercomm_create, но с упрощенным синтаксисом. Это использует только что определенные функциональные возможности для создания обслуживания по имени.

```
int Intercomm_name_create(local_comm, server_comm, tag, comm)
MPI_Comm local_comm, server_comm;
int tag;
MPI_Comm *comm;
 {
       int error;
       int found; /* приобретение атрибута mgmt для new_world
                           comm в server_comm */
       void *val;

       MPI_Comm new_world;

       int buffer[10], rank;
       int local_leader = 0;

       MPI_Attr_get(server_comm, server_keyval, &val, &found);
       new_world = (MPI_Comm)val; /* найти кешированный handle */

       MPI_Comm_rank(server_comm, &rank);/*ранг в локальной группе*/

       if (rank == local_leader)
              {
              buffer[0] = rank;
              MPI_Send(&buffer, 1, MPI_INT, 0, tag, server_comm);
              MPI_Recv(&buffer, 1, MPI_INT, 0, tag, server_comm);
              }

       error = MPI_Intercomm_create(local_leader, local_comm,
                                           buffer[0], new_world, tag, comm);
       return(error);
 }
```

5.7 Кеширование

Целесообразность. MPI обеспечивает возможность "кеширования", которая позволяет подсоединять произвольные фрагменты информации, называемые атрибутами, к коммуникаторам. Более точно, возможность кеширования позволяет совместимой библиотеке делать следующее:

- проводить информацию между вызовами связывая ее с MPI интра- или интер-коммуникаторами,
- быстро возвращать эту информацию и
- гарантировать, что информация за пределами данных никогда не восстановится, даже если коммуникатор освобожден и его хэндлер в последствии многократно используется в MPI.

Способности кеширования используются, в некоторой форме, встроенными программами MPI, такими как групповые обмены и топологии. Определение интерфейса с этими возможностями, как части стандарта MPI, является ценным потому, что это позволяет разрабатывать программы, подобные групповым обменам и топологии как совместимые коды, и также потому, что это делает MPI более гибким, позволяя прикладным программам использовать стандартные последовательности вызовов MPI. (Конец замечания.)

Совет пользователям. Коммуникатор MPI_COMM_SELF является пригодным для посылки процесс-локальных атрибутов через этот

атрибуто-кэширующий механизм. (Конец совета пользователям.)

5.7.1 Функциональность

Атрибуты присоединяются к коммуникатору. Атрибуты локальны для коммуникатора и специфичны для процесса, на котором они присоединены. Атрибуты в MPI не размножаются от одного коммуникатора к другому, за исключение тех случаев, когда коммуникатор дублируется с использованием MPI_COMM_DUP (и затем каждое применение должно давать специальное разрешение через обратный вызов функции для копирования атрибутов).

Совет разработчикам. Атрибуты имеют скалярные значения, равные или большие по размеру чем указатели в языке C. Атрибуты всегда могут храниться как хэндлеры MPI. (Конец совета разработчикам.)

Интерфейс кеширования определен здесь этими атрибутами, скрыто сохраняемыми MPI внутри коммуникатора. Функции доступа включают следующее:

- получение ключевого значения (используемого для идентификации атрибута); пользователь указывает функцию "обратного вызова", которой MPI информирует прикладную задачу, когда коммуникатор уничтожается или копируется.

- сохранение и возвращение значения атрибута;

Совет разработчикам. Функции кеширования и обратного вызова вызываются только синхронно, в ответ на точные запросы прикладной задачи. Это позволяет избежать проблем, которые являются результатом повторного пересечения пространств пользователя и системы. (Этот принцип синхронного вызова является основным свойством MPI).

Выбор ключевого значения осуществляется под контролем MPI. Это позволяет MPI оптимизировать разработки наборов атрибутов. Это также позволяет избежать конфликта между независимыми модулями кешированной информации на одинаковых коммуникаторах.

Значительно меньший интерфейс, состоящий только из средств обратного вызова, предоставляет портативные процедуры, обеспечивающие полноценные средства кеширования. Однако, в минимальном интерфейсе обратного вызова применяется некоторая форма поиска по таблице. Напротив, более сложный интерфейс определяемый здесь позволяет быстрый доступ к атрибутам используя указатели в коммуникаторах (чтобы находить таблицу атрибутов) и разумный выбор ключевых значений (чтобы выводить индивидуальные атрибуты). В свете эффективности присущей минимальному интерфейсу, более сложный интерфейс определённый здесь кажется излишним. (Конец совета разработчикам.)

Совет пользователям. MPI обеспечивает следующие возможности кеширования. Они все локальны. (Конец совета пользователям.)

```
MPI_KEYVAL_CREATE(copy_fn, delete_fn, extra_state)
IN     copy_fn              Копировать функцию обратного вызова для keyval
IN     delete_fn            Уничтожить функцию обратного вызова для keyval
OUT    keyval               ключевое значение доступа в будущем (integer)
IN     extra_state          Экстра-состояние функции обратного вызова

int MPI_Keyval_create(int copy_fn(), int delete_fn(), int *keyval,
void* extra_state)

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
       EXTERNAL COPY_FN, DELETE_FN
       INTEGER KEYVAL, EXTRA_STATE, IERROR
```

Порождает новый атрибут. Ключи локально уникальны для процесса и скрыты от пользователя, хотя они определённо хранятся в целых. После размещения ключевое значение можно использовать чтобы связаться с коммуникатором и получить к нему доступ любому локально определённому коммуникатору.

Функция copy_fn вызывается, когда коммуникатор дублируется. Си прототип для такой функции следующий:

```
typedef int MPI_copy_function(MPI_Comm *oldcomm, MPI_Comm *newcomm,
              int *keyval, void *extra_state)
```

Fortran прототип для этой функции следующий:

```
FUNCTION COPY_FN(OLDCOMM, NEWCOMM, KEYVAL, EXTRA_STATE)
       INTEGER OLDCOMM, NEWCOMM, KEYVAL, EXTRA_STATE
```

Эта функция делает то, что нужно с новым коммуникатором newcomm основываясь на том какая информация атрибутов была в старом коммуникаторе oldcomm; она возвращает 1 если успешна, и 0 иначе. Верная функция копирования это либо функция, которая полностью дублирует информацию, делая копию всех структур данных применяемых атрибутами; либо делает ещё одну ссылку на эти структуры данных, используя механизм подсчёта ссылок. Другие типы атрибутов могут не копироваться совсем (они могут быть важны только для oldcomm). Только MPI_COMM_DUP вызывает copy_fn. В качестве copy_fn можно использовать указатель на пустую функцию в Си и MPI_NULL_FN как в Си так и в Fortran'е, в этом случае для keyval не происходит копирования обратных вызовов. Заметим, что Си версии MPI_COMM_DUP предполагает, что функции обратного вызова следуют Си прототипу, а соответствующая Fortran версия предполагает Fortran прототип.

Аналогично copy_fn функция стирания обратного вызова определяется следующим образом:

```
typedef int MPI_delete_function(MPI_Comm *comm, int *keyval,void
*extra_state)
```

Fortran прототип для этой функции следующий:

```
FUNCTION DELETE_FN(COMM, KEYVAL, EXTRA_STATE)
       INTEGER COMM, KEYVAL, EXTRA_STATE
```

Эта функция вызывается MPI_COMM_FREE и MPI_ATTR_DELETE, чтобы сделать всё для удаления атрибута. Можно использовать указатель на пустую функцию в Си и MPI_NULL_FN как в Си так и в Fortran'е, в этом случае для keyval не происходит стирания обратных вызовов.

Специальное ключевое значение MPI_KEYVAL_INVALID не является верным аргументом для функций атрибутов, но оно полезно для статической инициализации ключевых значений.

```
MPI_KEYVAL_FREE(keyval)
INOUT         keyval        Освобождает целое ключевое значение (integer)

int MPI_Keyval_free(int *keyval)

MPI_KEYVAL_FREE(KEYVAL, IERROR)
       INTEGER KEYVAL, IERROR
```

Освобождает внешний ключ атрибута. Эта функция устанавливает значение keyval в MPI_KEYVAL_INVALID. Отметим, что не является ошибкой освобождение используемого атрибута, потому что реальное освобождения на происходит до тех пор пока не освободятся все ссылки (в других коммуникаторах в процессе) на этот ключ. Эти ссылки должны освобождаться программой, либо с помощью вызова MPI_ATTR_DELETE, который освобождает один атрибут, или MPI_COMM_FREE, который освобождает все атрибуты ассоциированные с коммуникатором.

Совет разработчикам. Функция MPI_NULL_FN не должна быть (void (*))0 в Си, хотя это удобно. Это должна быть нормально

вызывается, которую можно профилировать и так далее. Для Fortran'а наиболее удобно, чтобы MPI_NULL_FN была законной ничего неделающей функцией. (Конец совета разработчикам.)

```
MPI_ATTR_PUT(comm, keyval, attribute_val)
IN     comm                коммуникатор, к которому прикрепляется атрибут
                           (handle)
IN     keyval              ключевое значение возвращаемое
                           MPI_KEYVAL_CREATE (integer)
IN     attribute_val       значение атрибута
```

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
```

```
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
       INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
```

Эта функция запоминает обусловленное значение атрибута attribute_val для последующего вывода с помощью MPI_ATTR_GET. Если значение уже есть, то результат получается как если бы сначала вызывалась MPI_ATTR_DELETE, чтобы стереть предыдущее значение (и выполняется функция обратного вызова delete_fn), затем запоминается новое значение. Вызов будет ошибочен если нет ключа со значением keyval; в частности MPI_KEYVAL_INVALID это ошибочное значение ключа.

```
MPI_ATTR_GET(comm, keyval, attribute_val, flag)
IN     comm                коммуникатор, к которому прикреплен атрибут
                           (handle)
IN     keyval              ключевое значение (integer)
OUT    attribute_val       значение атрибута, если нет flag=false
OUT    flag                true - если значение атрибута было извлечено;
                           false - если никакой атрибут не связан с
                           ключом
```

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void **attribute_val,
int *flag)
```

```
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
       INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
       LOGICAL FLAG
```

Выводит значение атрибута по ключу. Вызов ошибочен если нет ключа со значением keyval. С другой стороны вызов верен если значение ключа существует, но для этого ключа к коммуникатору comm не прикреплён никакой атрибут; в этом случае возвращается flag = false. В частности MPI_KEYVAL_INVALID это ошибочное значение ключа.

```
MPI_ATTR_DELETE(comm, keyval)
IN     comm          коммуникатор, к которому прикреплен атрибут (handle)
IN     keyval        ключевое значение уничтожаемого атрибута (integer)
```

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

```
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
       INTEGER COMM, KEYVAL, IERROR
```

Совет пользователям. Стирает атрибут из кэша по ключу. Эта функция использует функцию стирания атрибута delete_fn,которая была отмечена при создании keyval.

Когда коммуникатор дублируется с помощью MPI_COMM_DUP, привлекаются все функции копирования для атрибутов, которые установлены (в произвольном порядке). Когда коммуникатор стирается используя функцию MPI_COMM_FREE привлекаются все функции стирания для атрибутов, которые установлены. (Конец совета пользователям.)

5.7.2 Примеры атрибутов

Совет пользователям. Этот пример показывает как написать операцию групповых обменов, которая использует кеширование для большей эффективности после первого вызова. Стиль программирования допускает, что вызовы MPI возвращают только верные статусы. (Конец совета пользователям.)

```
static int gop_key = MPI_ATTR_KEYVAL_INVALID;                          /* ключ для
                                                                       этого модуля
*/
typedef struct
       {
       int ref_count;              /* счетчик ссылок */
       /* все иное что мы хотим */
       } gop_stuff_type;

Efficient_Collective_Op (comm, ...)
MPI_Comm comm;
 {
       gop_stuff_type       *gop_stuff;
       MPI_Group                   group;
       int                    foundflag;

       MPI_Comm_group(comm, &group);

       if (gop_key == MPI_ATTR_KEYVAL_INVALID)                 /* получить ключ на
                                                               первом же вызове */
              {
              if ( ! MPI_Attr_keyval_create( gop_stuff_copier,
                            gop_stuff_destructor, &gop_key, (void *)0));
              /* get the key while assigning its copy and delete
                            callback behavior */
              MPI_Abort ("Insuficient keys available");
              }
       MPI_Attr_get (comm, gop_key, &gop_stuff, &foundflag);
       if (foundflag)
              { /* Этот модуль выполняется в этой группе вначале. Мы
                     будем использовать кешируемую информацию */
              }
       else
              {
              /* Это группа в которой еще ничего не кешировано. Далее
                     мы будем делать следующим образом. */

              /* Во-первых, мы зарезервируем память для stuff и
                     инициализируем счетчик ссылок */
              gop_stuff = (gop_stuff_type *) malloc
                     (sizeof(gop_stuff_type));
              if (gop_stuff == NULL)
                     { /*прерывание по недостатку памяти */ }

              gop_stuff -> ref_count = 1;

              /* Во-вторых, заполним *gop_stuff чем хотим. Эта часть
                     здесь не представлена. */

              /* В-третьих, сохраним gop_stuff как значение атрибута */
              MPI_Attr_put (comm, gop_key, gop_stuff);
              }
       /* Затем, в любом случае, используем содержимое *gop_stuff,
       чтобы выполнить глобальную операцию (op)...*/
 }

/* Следующая программа вызывается MPI, когда группа освобождается*/

gop_stuff_destructor (comm, keyval, gop_stuff, extra)
```

```
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff;
void* extra;
 {
       if (keyval != gop_key) { /* Выход -- программная ошибка */}

       /* Группа освобождается переносом ссылки в gop_stuff */
       gop_stuff -> ref_count -= 1;

       /* Если не осталось ссылок, то освобождается вся память */
       if (gop_stuff -> ref_count == 0) {
              free((void *)gop_stuff);
              }
 }

 /* Следующая программа вызывается MPI, когда группа копируется */
gop_stuff_copier (comm, keyval, gop_stuff, extra)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff;
void *extra;
 {     if (keyval != gop_key) { /* Выход -- программная ошибка */ }

       /* Новая группа добавляет ссылку в gop_stuff */
       gop_stuff -> ref_count += 1;
 }
```

5.8 Формализация неточно синхронной модели

В этой части мы делаем некоторые утверждения о неточно синхронной модели, с особым вниманием к интра-коммуникаторам.

5.8.1 Основные утверждения

Когда вызывающий передаёт коммуникатор (который содержит контекст и группу) вызываемому, этот коммуникатор должен быть свободным от побочных эффектов во время выполнения подпрограммы: не должно быть активных операций с этим коммуникатором, которые могут включать процесс. Это обеспечивает модель, в которой библиотеки могут быть написаны и работать "безопасно". В библиотеке построенной таким образом вызываемый имеет возможность использовать обмены с коммуникатором, имея гарантию, что они не пересекутся с другими обменами. Так как мы допускаем хорошие реализации создающие коммуникаторы без синхронизации (такие как распределённые заранее контексты коммуникаторов), это не ведёт к значительным издержкам.

Эта форма безопасности аналогична другим общепринятым употреблениям, таким как передача дескриптора массива библиотечной процедуре. Библиотечная программа имеет все права ожидать, что этот дескриптор будет верным и изменяемым.

5.8.2 Модели выполнения

В неточно синхронной модели передача управления параллельной процедуре осуществляется с помощью выполнения этой процедуры каждым процессом. Вызов групповых операций выполняется всеми процессами группы, и вызовы одинаково упорядочены на всех процессах группы. Однако вызовы не обязательно должны быть синхронизованы.

Мы говорим, что параллельная процедура активна на процессе если процесс входит в группу, которая может коллективно выполнять эту процедуру, и какой-то член группы в данный момент выполняет её. Если параллельная процедура активна на процессе, то этот процесс может получить сообщение относящееся к этой процедуре, даже если он не выполняет эту процедуру в данный момент.

Статическое размещение коммуникаторов

Это охватывает случай, когда в любой момент времени только одна параллельная процедура может быть активна на любом процессе, и группа выполняющих процессов постоянна. Например, все параллельные процедуры включают все процессы, процессы однопутевые и нет рекурсивных вызовов.

В этом случае коммуникатор можно размещать статически для каждой процедуры. Статическое размещение можно произвести заранее, как часть инициализации. Если параллельные процедуры можно организовать в библиотеки, так что только одна процедура каждой библиотеки была одновременно активна на каждом процессе, то достаточно выделить по одному коммуникатору на библиотеку.

Динамическое размещение коммуникаторов

Вызовы параллельных процедур гнездовые, если новая процедура всегда вызывается на подмножестве группы выполняющей такую же параллельную процедуру. Таким образом процессы, которые выполняют одну и ту же параллельную процедуру, имеют одинаковый стек выполнения.

В этом случае новый коммуникатор должен динамически размещаться для каждого вызова параллельной процедуры. Размещение производится вызываемым. Новый коммуникатор может порождаться с помощью вызова MPI_COMM_DUP, если вызываемая группа идентична вызвавшей группе, или с помощью вызова MPI_COMM_SPLIT если вызывающая группа делится на несколько подгрупп выполняющих различные параллельные процедуры. Новый коммуникатор посылается как аргумент вызываемой процедуре.

Необходимость порождения нового коммуникатора при каждом вызове можно облегчить или избежать совсем в некоторых случаях: если выполняющая группа не делится, тогда можно предварительно разместить стек коммуникаторов и затем управлять им подражая управлению стеков рекурсивных вызовов.

Можно также воспользоваться свойством хорошей упорядоченности обменов, чтобы избежать смешения обменов вызывающего и вызываемого, даже если оба используют один коммуникатор. Чтобы добиться этого, необходимо соблюдать следующие два правила:

- сообщения посланные до вызова процедуры (или перед возвратом из процедуры) также принимаются до соответствующего вызова (или возврата) на принимающем конце;

- сообщения всегда выбираются с помощью источника (не используя MPI_ANY_SOURCE).

Общий случай

В общем случае может быть много одновременно активных вызовов одной параллельной процедуры внутри одной группы; вызовы могут не быть гнездовыми. Новые коммуникаторы должны создаваться для каждого вызова. На пользователя возлагается ответственность за обеспечение того, чтобы две разных параллельных процедуры не вызывались одновременно на пересекающихся множествах процессов, тогда создание коммуникаторов будет хорошо координировано.