Коммуникационные операции в MPI

Марчевский И.К., Попов А.Ю.

МГТУ им. Н.Э. Баумана

Принцип работы технологии MPI I

В отличие от технологии OpenMP, которая предполагает работу с общей памятью (в едином адресном пространстве) и является расширением языков C и Fortran, MPI представляет собой реализацию *системы передачи сообщений*. Ее основные характеристики:

- MPI является библиотекой, а не языком. Она содержит функции, константы и пр. конструкции (все начинаются на `MPI_`), которые используются в программах на языках C и Fortran. Для сборки используются привычные компиляторы этих языков, компоновщик (linker) добавляет библиотеку MPI.
- MPI представляет собой стандарт/спецификацию, а не конечную реализацию. Основные дистрибутивы в той или иной мере поддерживают спецификацию MPI (версий 1 и 2 — почти полностью), что позволяет использовать (собирать и запускать) одну и ту же пользовательскую программу с разными дистрибутивами без изменений.

Принцип работы технологии MPI II

- MPI реализует собой модель параллельных вычислений с помощью передачи вычислений, т.е. каждый процесс имеет доступ только к локальной памяти, а доступ к данным в памяти другого процесса возможен только путем передачи сообщений, причем в этой процедуре действия выполняют оба участника.

Операции передачи данных в MPI производятся в *коммуникаторах* — пространствах процессов, удовлетворяющих следующим свойствам:

1. сообщение принимается в том же коммуникаторе, в котором оно было отправлено;
2. сообщения, отправленные в разных коммуникаторах, не пересекаются друг с другом.

Всегда доступен предопределенный коммуникатор `MPI_COMM_WORLD`, в который входят процессы (а также `MPI_COMM_SELF`).

Базовые функции

- `MPI_Init(int *argc, char ***argv)` — инициализация системы для вызова функций MPI. Программа должна содержать ровно один вызов этой процедуры, он не обязан идти в самом начале программы, но использование функций MPI возможно только после него.
- `MPI_Finalize(void)` — завершение работы с MPI. Не производит освобождение памяти для объектов, порожденных функциями MPI (для этого используются соответствующие функции `MPI_XXX_FREE`).
- `double MPI_Wtime(void)` — таймер (возвращает время в секундах, отсчитывая от некоторого момента в прошлом).

Отправка данных от одного узла другому I

Операции передачи данных «точка — точка» (в отличие от коллективных операций) предполагает подготовку и отправку данных одним процессом в адрес другого, и получение и запись данных последним. При этом остальные процессы продолжают выполнение программы.

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

Аргументы `buf` (адрес переменной для отправки данных), `count` и `datatype` определяют собственно передаваемые данные (аналог содержимого письма). Указание количества единиц данных и их типа (вместо количества байт) сделано в целях универсализации и отвязки от конкретной реализации типа данных на конкретной архитектуре.

Аргументы `dest` (ранг получателя), `tag` (метка письма, число от 0 до `MPI_TAG_UB`) и `comm` (коммуникатор) вместе с рангом отправителя (указывается MPI неявно) определяют сопроводительную информацию о пакете данных (аналог конверта письма).

Отправка данных от одного узла другому II

Некоторые базовые типы данных в MPI (целочисленные константы):

Тип данных MPI	Тип данных C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Получение данных при отправке от одного узла другому

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

Как и в случае отправки, первые 3 аргументы относятся к данным (`buf` — адрес переменной для записи данных), следующие 3 — сопроводительную информацию. В качестве значения `source` вместо конкретного ранга процесса-отправителя может быть указана константа `MPI_ANY_SOURCE`, в качестве значения `tag` — вместо конкретной метки — константа `MPI_ANY_TAG`.

В случае использования хотя бы одной из них требуемая информация может быть получена из переменной типа `MPI_Status`. Соответствующая структура будет содержать как минимум поля `MPI_SOURCE`, `MPI_TAG` и `MPI_ERROR` (код ошибки, при ее отсутствии — `MPI_SUCCESS`). Если информация не требуется, то для экономии времени на ее заполнение можно передавать в качестве аргумента константу `MPI_STATUS_IGNORE`.

Особенности выполнения операций «точка — точка» I

Пересылки с помощью `MPI_Send` и `MPI_Recv` называются блокирующими операциями, в этом случае работа процесса-отправителя не продолжится, пока отправленные данные не получены и снова не появится возможность модифицировать отправленную переменную.

В случае таких операций возможен *тупик* (deadlock), когда процесс приостановит выполнение вычислений на неопределенное время из-за незавершенной операции коммуникации (например, прием до передачи либо прием от ранга или метки, для которых не отправлено сообщение).
Выполнение блокирующих операций зависит от режима, режим выбирается в зависимости от размера сообщения и от реализации MPI.

Особенности выполнения операций «точка — точка» II

буферизованный — позволяет отвязать отправку от получения путем записи сообщения в буфер, после которой отправитель может продолжить работу, не дожидаясь завершения приема сообщения получателем;

синхронный — отправка считается успешно законченной только после того, как пришел ответ о начатом приеме данных получателем;

режим «по готовности» — отправка может быть начата только если уже имеется информация о готовности принимать от получателя. Позволяет несколько сократить время, но в случае отсутствия запроса от получателя на момент отправки в программе возникнет ошибка.

В случае небольших сообщений MPI буферизует сообщения, иначе используется синхронный режим.

Особенности выполнения операций «точка — точка» III

Можно увеличить используемый буфер путем использования новой переменной:
`MPI_Buffer_attach(void *buffer, int size)`. Одновременно можно привязать только один буфер. Для отсоединения используется функция `MPI_Buffer_detach(void *buffer_addr, int *size)`.

Режим можно выбирать принудительно путем вызова одной из функций `MPI_Bsend`, `MPI_Ssend`, `MPI_Rsend`, набор аргументов один и тот же, совпадающий с `MPI_Send`. Функция получения для всех трех режимах единая (`MPI_Recv`).

Отправка и получение сообщения одним и тем же процессом с помощью блокирующей пересылки запрещена во избежание возникновения тупика.

Объединенная пересылка между двумя узлами

При массовом выполнении блокирующих пересылок процессами, особенно в режиме цепочки (одна часть процессов отправляет данные, затем — получает, другие — наоборот), могут возникнуть тупиковые ситуации в силу циклических зависимостей. Для их устранения можно использовать одновременную операцию отправки и получения:

```
MPI_Sendrecv(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
int recvcount, MPI_Datatype recvtype, int source, int
recvtag, MPI_Comm comm, MPI_Status *status),
```

которая объединяет в себе информацию о данных и о самой пересылке как части отправки, так и в части получения. Возможно использования одной и той же переменной и для отправки, и для получения:

```
MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

Потребность в коллективных коммуникационных операциях I

Во многих программах существует необходимость рассылки исходных данных (полностью или частично) по процессам, сборки решения, вычисление интегральных характеристик (сумма, поиск max и пр.).

- Например, рассылка 1 числа или вектора всем узлам может быть организована с помощью `MPI_Send` и `MPI_Recv` от процесса 0 — всем остальным.
- Гораздо эффективнее будет рассылка, если организовать ее по дереву коммуникаций: на шаге 1 узел 0 отправляет данные узлу 1, на шаге 2 — $0 \to 2, 1 \to 3$, на шаге 3 — $0 \to 4, 1 \to 5, 2 \to 6, 3 \to 7$.
- Однако рассылку можно организовать иначе: на шаге 1 — $0 \to 4$, на шаге 2 — $0 \to 2, 4 \to 6$, на шаге 3 — $0 \to 1, 2 \to 3, 4 \to 5, 6 \to 7$. При этом потребуется организовать это дерево самостоятельно, причем оно значительно зависит от топологии сети.

Потребность в коллективных коммуникационных операциях II

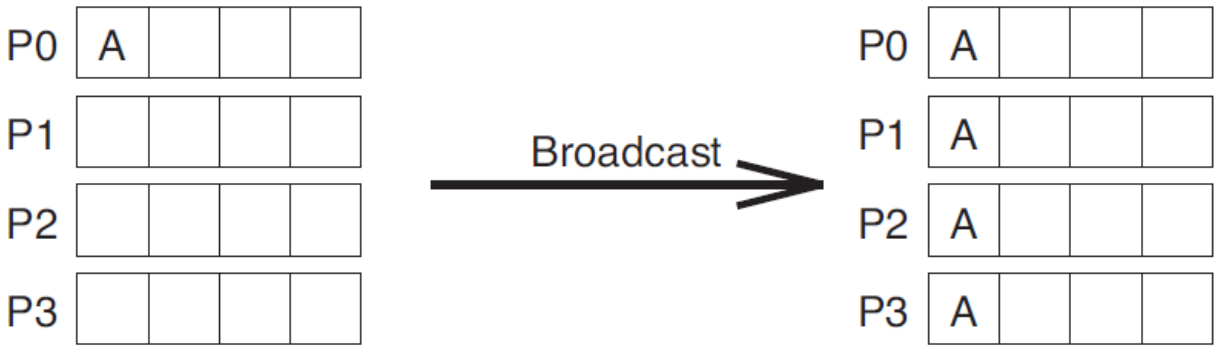
Коллективные операции передачи данных в MPI реализуют групповые коммуникационные операции, в которых участвуют все процессы коммуникатора, причем эффективным образом, с высокой долей оптимизации. Особенности коллективных операций:

1. высокий уровень абстрактности, указываются только основные параметры — коммуникатор, передаваемые данные, их размер, тип и т.п., без указаний на топологию сети;
2. операция выполняется всеми процессами-участниками, вызова функции `MPI_Recv` не предусмотрено;
3. операция в зависимости от своего назначения может содержать или не содержать выделенный процесс (root);
4. в коллективных операциях нет метки (тэга) сообщения.

Рассылка данных от одного узла остальным

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

Функция рассылает данные в количестве `count` значений типа `datatype` от процесса `root` (любой процесс, не обязательно 0) всем, включая его самого. Команда выполняется всеми процессами, в результате чего у каждого узла в переменной `buffer` будут одинаковые данные.



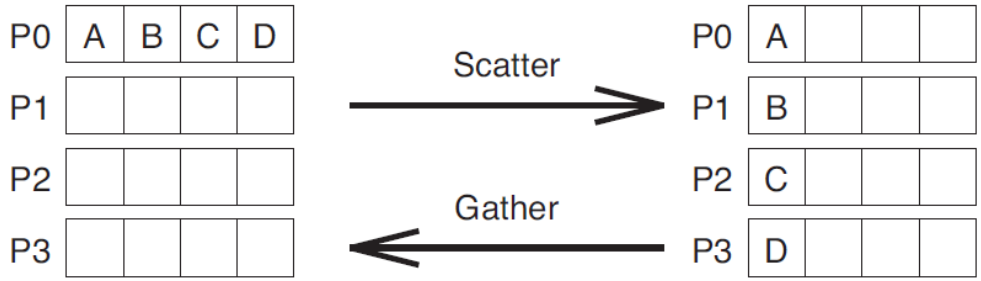
Распределение данных между процессами I

При выполнении параллельных операций над частями данных (например, вычисление скалярного произведения двух векторов, умножение матрицы на вектор) вместо рассылки полного объема данных (с помощью `MPI_Bcast`) более эффективно рассылать каждому процессу необходимую ему часть данных. Рассылка выполняется с помощью функции

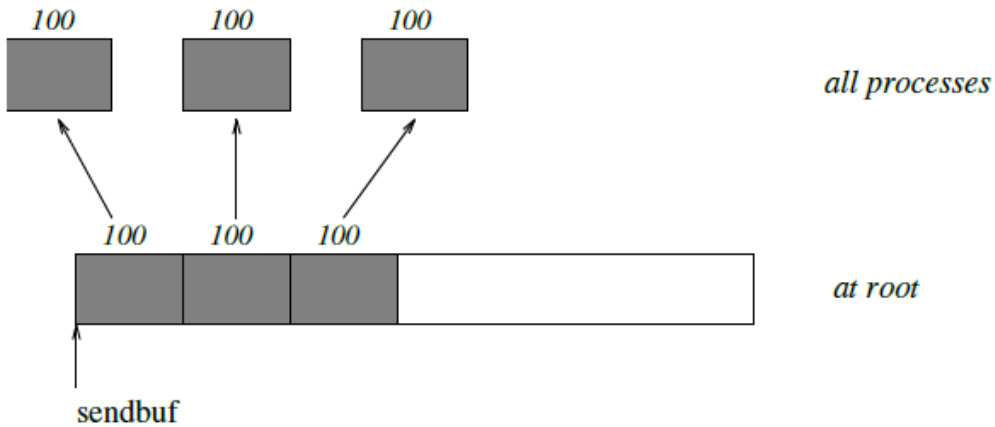
```
MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

Первые 3 аргумента определяют рассылаемые данные и имеют значения *только для процесса* `root` (на остальных игнорируются). Остальные определяют место записи получаемых данных `recvbuf` (для каждого процесса — свое), их количество и тип. Предполагается, что между рассылаемыми частями массива данных нет пересечений (каждая позиция считывается ровно один раз).

Распределение данных между процессами II



Для процесса `root` разрешается совпадения буферов отправки и приема, в таком случае для него вместо `recvbuf` указывается константа `MPI_IN_PLACE`. Размер рассылаемых частей массива данных должен быть равным:



Сборка данных, хранящихся в распределенной памяти I

Обратная операция — сборка единого массива данных на процессе `root` из частей, хранящихся отдельно в памяти вычислительных узлов (например, результат произведения матрицы на вектор). Выполняется функцией

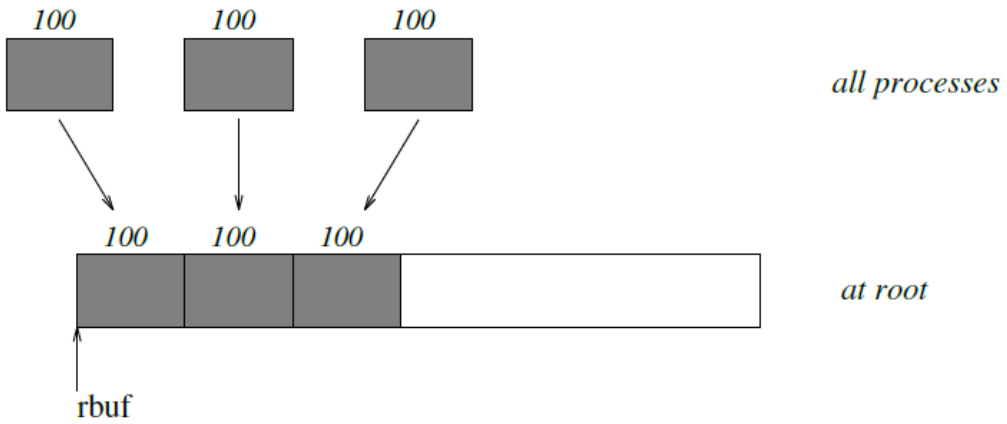
```
MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

Первые 3 аргумента должны быть заполнены для всех процессов; `recvbuf`, `recvcount` и `recvtype` важны только для процесса `root`. Аргумент `recvcount` определяет количество элементов, принимаемое от каждого процесса, а не суммарное. Каждый элемент массива `recvbuf` должен быть заполнен ровно 1 раз (пересечения запрещены).

Для процесса `root` разрешается в качестве `recvbuf` указывать опцию `MPI_IN_PLACE`, в таком случае предполагается, что его часть данных уже находится в предназначенном месте целевого буфера.

Сборка данных, хранящихся в распределенной памяти II

Размер принимаемых частей массива данных должен быть равным:



При использовании `MPI_Gather` результирующий массив будет собран только на процессе `root`. Имеется возможность одновременной сборки данных всеми процессами (аналог выполнения `MPI_Gather` с $\texttt{root} = 0, \ldots, np - 1$):

```
MPI_Allgather(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm)
```

Аналогично доступна опция `MPI_IN_PLACE`

Выполнение редукции по данным в распределенной памяти I

Для выполнения глобальной операции редукции по данным различных узлов (min, max, сумма и пр.) используется функция
`MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

Операция `op` выполняется над `count` элементами типа `datatype` в переменной `sendbuf` по всем процессам (должны иметь одинаковое количество элементов) и сохраняется в переменной `recvbuf` на процессе с номером `root`. Если элементов несколько, то операция выполняется отдельно по всем элементами `sendbuf[0]`, `sendbuf[1]`,...

Если необходимо обеспечить доступность результата редукции на всех узлах, используется функция
`MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

Выполнение редукции по данным в распределенной памяти II

Операции в редукции обязаны быть ассоциативными, встроенные операции также коммутативны:

Обозначение	Операция
`MPI_MAX`, `MPI_MIN`	Максимум, минимум
`MPI_SUM`, `MPI_PROD`	Сумма, произведение
`MPI_LAND`, `MPI_BAND`	Логическое/побитовое «И»
`MPI_LOR`, `MPI_BOR`	Логическое/побитовое «ИЛИ»
`MPI_LXOR`, `MPI_BXOR`	Логическое/побитовое «Исключающее ИЛИ»
`MPI_MAXLOC`, `MPI_MINLOC`	Максимум, минимум с указанием позиции

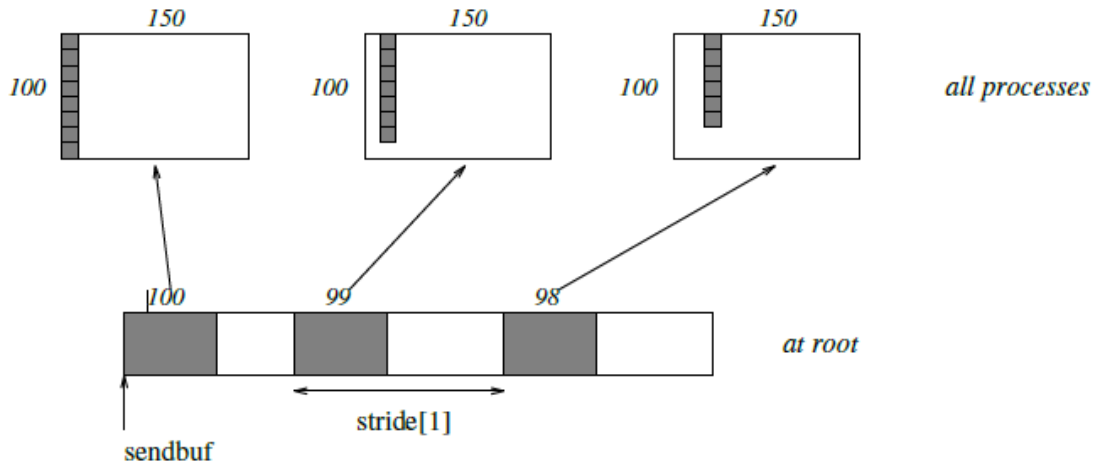
Можно добавлять свои собственные операции редукции с помощью функции (`user_fn` — функция, `commute` — признак коммутативности) `MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)`. В конце программы необходимо высвободить операцию с помощью `MPI_Op_free(MPI_Op *op)`.

Операции в случае данных различной длины

Зачастую размер данных для различных процессов может быть разным. Имеются варианты функций `MPI_Gather/MPI_Scatter` для этого случая:

- `MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm);`
- `MPI_Scatterv(const void *sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm).`

Дополнительные целочисленные массивы `recvcounts/sendcounts` — количество элементов для каждого процесса, `displs` — сдвиги относительно начала массива.



Проблема организации пересылок

- В сложной программе с большим количеством пересылок, особенно если их порядок определяется при работе, возникает необходимость правильной организации их последовательности — каждой отправке должен в правильном порядке соответствовать прием. В противном случае возникнет ситуация тупика.
- Ситуацию может улучшить буферизованная отправка, однако размер буфера является ограниченным.
- Программа на MPI считается «безопасной» (“safe”), если все пересылки организованы так, что не требуется буферизация и при замене всех отправок на синхронные не произойдет тупик. На практике, однако, стараются делать менее детерминированные, но все же робастные программы.

Идея неблокирующих коммуникационных операций

- Неблокирующие операции позволяют выполнить наложение различных пересылок и тем самым избежать тупиковой ситуации, поскольку они в таком случае не обязаны выполняться в единой строгой последовательности.
- Неблокирующие операции позволяют выполнить также и наложение операций передачи данных и вычислений, поскольку в таком случае первые можно «скрыть» за вторыми.
- При выполнении блокирующей операции отправки программа не может продолжить выполнение прежде, чем данные будут скопированы в буфер отправки и начнется сама передача данных. Неблокирующая же команда только *инициирует* отправку, но еще не завершает ее. Для того чтобы убедиться, что данные скопированы в буфер для отправки и могут быть перезаписаны программой, необходимо выполнить еще одну операцию.
- Аналогично, неблокирующая команда приема только *инициирует* прием, но для того, чтобы использовать полученные и сохраненные данные, необходимо выполнить еще завершающую операцию.

Неблокирующая операция отправки

Отправка данных в неблокирующем режиме производится командой (префикс I означает incomplete/immediate — неполный/немедленный, имеется вид в виду возврат к выполнению дальнейших команд))

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

Функция содержит те же аргументы, что и `MPI_Send`, с добавлением объекта типа `MPI_Request` (запрос), который можно использовать затем для проверки завершенности передачи данных.

По аналогии с блокирующими операциями имеются команды `MPI_Ibsend` (буферизованный режим), `MPI_Issend` (синхронный режим) и `MPI_Irsend` (режим «по готовности»).

Неблокирующая операция приема

Прием данных в неблокирующем режиме производится командой `MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`

Функция содержит те же аргументы, что и `MPI_Recv`, но вместо статуса (объект типа `MPI_Status`) имеется объект типа `MPI_Request`, поскольку на момент выполнения этой команды собственно прием данных еще не выполнен, а только инициирован.

Режим операций отправки и приема может быть разным: например, разрешается неблокирующая отправка и блокирующая операция приема и наоборот.

В версии MPI 4.0 также доступна неблокирующая операция `MPI_Isendrecv`.

Завершение неблокирующих операций передачи данных I

Функции `MPI_Wait` и `MPI_Test` используются для того, чтобы убедиться, что неблокирующая операция завершена и данные можно перезаписывать (в случае отправки) либо использовать (при приеме).

- Функция `MPI_Wait(MPI_Request *request, MPI_Status *status)` приостанавливают дальнейшее исполнение программы до тех пор, пока связанная с запросом `request` коммуникационная операция не будет завершена. Когда прием данных будет завершен, сопутствующая информация будет доступна в объекте `status` (тэг, отправитель, код ошибки).
- Функция `MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)` проверяет завершенность операции, связанной с запросом, но не блокирует дальнейшее исполнение (может периодически вызываться в цикле). Состояние операции записывается в логическую переменную `flag`.

Завершение неблокирующих операций передачи данных II

В случае выполнения целого набора неблокирующих пересылок бывает целесообразно проверять (либо принудительно ожидать) завершенность хотя бы одной/некоторых/всех пересылок:

1. функция `MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)` требует завершения хотя бы одной пересылки из массива запросов (номер первой из них возвращается в переменной `index`);
2. функция `MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])` ожидает, пока не будут завершены все пересылки из массива запросов;
3. функция `MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[])` требует завершения хотя бы 1 пересылки, но возвращает информацию (номера, статусы) не только о самой первой, но о всех завершившихся к этому моменту.

Завершение неблокирующих операций передачи данных III

Также доступны аналогичные функции `MPI_Testany`, `MPI_Testall` и `MPI_Testsome`. Если не требуется заполнение статуса, вместо него передается константа `MPI_STATUS_IGNORE` либо `MPI_STATUSES_IGNORE` для функций `...SOME` и `...ALL`.

Дополнительно доступна функция ожидания поступления сообщения с заданным тэгом от указанного отправителя без его непосредственного получения (**не только для неблокирующих операций**):
`MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`

Собственно получение содержимого сообщения может быть выполнено позднее (не требуется сразу после возврата из `MPI_Probe`), что позволяет сделать за это время дополнительные действия (выделить память и пр.).

Также можно только проверять поступление сообщения, но не ожидать его принудительно (результат будет записан в переменную `flag`):
`MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)`

Повторяющиеся операции пересылок

Если одни и те же операции пересылки повторяются в параллельной программе в рамках некоторого цикла, то можно уменьшить накладные расходы за счет предварительной подготовки этих операций (создание буферов, заполнение сопроводительной информации и т.д.). В таком случае предварительно вызываются функции

`MPI_Send_init(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)` либо `MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`,

после чего объект `request` может быть использован многократно без обновления информации в нем. Инициализация соответствующей операции затем каждый раз делается с помощью `MPI_Start(MPI_Request *request)` или `MPI_Startall(int count, MPI_Request array_of_requests[])`, а проверка завершения — `MPI_Wait` и т.п.

Начиная с версии MPI 3.0, также доступны неблокирующие коллективные операции передачи данных.