Технологии параллельного программирования на C++

Семинар 8

MPI

Асинхронный обмен сообщениями

- Вызов неблокирующей (асинхронной) функции приводит к инициации запрошенной операции передачи, после чего сразу же происходит возврат из функции (без обработки сообщения), и процесс может продолжить свои действия.
- Перед своим завершением неблокирующая функция определяет переменную **request**, которая далее может использоваться для проверки завершения инициированной операции обмена.
- Проверка состояния выполняемой неблокирующей операцией передачи данных выполняется при помощи функций:

  MPI_Wait и **MPI_Test**

Асинхронная передача сообщения

- **int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request)**

  buf - адрес начала буфера посылки сообщения

  count - число передаваемых элементов в сообщении

  datatype - тип передаваемых элементов

  dest - номер процесса-получателя

  tag - идентификатор сообщения

  comm - идентификатор коммуникатора

  request- получаемый идентификатор асинхронной передачи

- После возврата из асинхронной функции передачи сообщения нельзя повторно использовать данный буфер **buf** для других целей без получения дополнительной информации о завершении данной посылки.

Асинхронная передача сообщения

- **int MPI_Issend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request)**

  - асинхронная передача сообщения с синхронизацией
- **int MPI_Ibsend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request)**

  - асинхронная передача сообщения с буферизацией
- **int MPI_Irsend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request)**

  - асинхронная передача сообщения по готовности.

Асинхронный прием сообщения

- **int MPI_Irecv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *request)**

  buf - получаемый адрес начала буфера приема сообщения

  count - максимальное число элементов в принимаемом сообщении

  datatype - тип элементов принимаемого сообщения

  source - номер процесса-отправителя (или **MPI_ANY_SOURCE**)

  tag - идентификатор принимаемого сообщения (или **MPI_ANY_TAG**)

  comm - идентификатор коммуникатора

  request - получаемый идентификатор асинхронного приема сообщения
- Возврат из функции происходит **сразу после инициализации** процесса приема без ожидания получения сообщения в буфере **buf**. Окончание процесса приема можно определить (так же, как и в асинхронных функциях передачи) с помощью параметра **request** и процедур **MPI_Wait** и **MPI_Test**.
- Сообщение, отправленное любой из функций **MPI_Send** и **MPI_Isend** (и их модификаций), может быть принято любой из функций **MPI_Recv** и **MPI_Irecv**.

Асинхронная проверка приёма сообщения

- **int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)**

  source - номер процесса-отправителя (или **MPI_ANY_SOURCE)**

  tag - идентификатор ожидаемого сообщения (или **MPI_ANY_TAG)**

  comm - идентификатор коммуникатора

  flag - получаемый признак завершенности операции обмена

  status - получаемые параметры обнаруженного сообщения

- Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре **flag** будет значение **1**, если сообщение с подходящими атрибутами уже может быть принято (действие аналогично **MPI_Probe**), и значение **0**, если сообщения с указанными атрибутами еще нет.

Ожидание завершения асинхронной операции, ассоциированной с идентификатором REQUEST

- **int MPI_Wait(MPI_Request *request, MPI_Status *status)**

  request - идентификатор асинхронного приема или передачи

  status – получаемые параметры сообщения

  Ожидание завершения асинхронных процедур **MPI_Isend** (и модификаций) или **MPI_Irecv**, ассоциированных с идентификатором **request**. В случае приема, атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра **status**.

  После выполнения функции идентификатор неблокирующей операции **request** устанавливается в значение **MPI_REQUEST_NULL**.

Ожидание завершения асинхронной операции, ассоциированной с идентификатором REQUEST

- **int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)**

  count - число идентификаторов

  requests - массив идентификаторов асинхронного приема или передачи

  statuses - получаемые параметры сообщений

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

Ожидание завершения асинхронной операции, ассоциированной с идентификатором REQUEST

- **int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status)**

  index – получаемый номер завершенной операции обмена

  status - получаемые параметры сообщений

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

- **int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)**

  outcount - получаемое число идентификаторов завершившихся операций обмена

  indexes- получаемый массив номеров завершившихся операции обмена

  statuses-получаемые параметры завершившихся сообщений

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

Проверка завершения асинхронных операций

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

  flag- получаемый признак завершенности операции обмена

Проверка завершенности асинхронных функций **MPI_Isend** или **MPI_Irecv**, ассоциированных с идентификатором **request**. В параметре **flag** будет значение **1**, если соответствующая операция завершена, и значение **0** в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра **status**.

После выполнения процедуры идентификатор неблокирующей операции **request**устанавливается в значение **MPI_REQUEST_NULL**

- **int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses)**

  count - число идентификаторов

  requests-массив идентификаторов асинхронного приема или передачи

  flag - признак завершенности операций обмена

В параметре **flag** будет значение **1**, если **все** операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве **statuses**). В противном случае возвращается **0**, и определенность элементов массива **statuses** не гарантируется.

Проверка завершения асинхронных операций

- **int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)**
- Если к моменту вызова подпрограммы хотя бы одна из операций обмена завершилась, то в параметре **flag** будет значение **1**, **index** содержит номер соответствующего элемента в массиве **requests**, а ***status***-параметры сообщения.
- **int MPI_Testsome(intincount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)**
- Данная функция работает так же, как и **MPI_Waitsome**, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение **outcount** будет равно нулю.

Возникновение тупиковых ситуаций

процесс 0:	процесс 1:
MPI_RECV от процесса 1 MPI_SEND процессу 1	MPI_RECV от процесса 0 MPI_SEND процессу 0

процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_SEND процессу 0 MPI_RECV от процесса 0

процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_RECV от процесса 0 MPI_SEND процессу 0

Процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_IRECV от процесса 0 MPI_SEND процессу 0 MPI_WAIT

Возникновение тупиковых ситуаций

процесс 0:	процесс 1:
MPI_RECV от процесса 1 MPI_SEND процессу 1	MPI_RECV от процесса 0 MPI_SEND процессу 0

Возникает тупик!

процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_SEND процессу 0 MPI_RECV от процесса 0

Может возникнуть тупик!

процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_RECV от процесса 0 MPI_SEND процессу 0

Тупик не возникает

Процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_IRECV от процесса 0 MPI_SEND процессу 0 MPI_WAIT

Тупик не возникает

Одновременное выполнение передачи и приема

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

  sbuf - адрес начала буфера посылки сообщения

  scount - число передаваемых элементов в сообщении

  stype - тип передаваемых элементов

  dest - номер процесса-получателя

  stag - идентификатор посылаемого сообщения

  rbuf - *получаемый* адрес начала буфера приема сообщения

  rcount - число принимаемых элементов сообщения

  rtype - тип принимаемых элементов

  source - номер процесса-отправителя

  rtag - идентификатор принимаемого сообщения

  comm - идентификатор коммуникатора

  status - *получаемый* параметры принятого сообщения

- Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией **MPI_Sendrecv**, может быть принято обычным образом, и точно также операция **MPI_Sendrecv** может принять сообщение, отправленное обычной операцией **MPI_Send**. Буфера приема и посылки обязательно должны быть различными.
- **int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype, int dest, int stag, int source, int rtag, MPI_Comm comm, MPI_Status *status)**

Коллективные операции

- Под **коллективными операциями** в MPI понимаются операции над данными, в которых принимают участие **все** процессы используемого коммуникатора.
- Возврат из функции коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено.
- Как и для блокирующих функций, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата.

Коллективные операции

- **int MPI_Barrier(MPI_Comm comm)**

  comm - идентификатор коммуникатора

  Блокирует работу процессов, вызвавших данную функцию, до тех пор, пока все оставшиеся процессы коммуникатора **comm** также не выполнят эту функцию.

Передача данных от одного процесса всем процессам программы

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

  buf - адрес начала буфера посылки сообщения

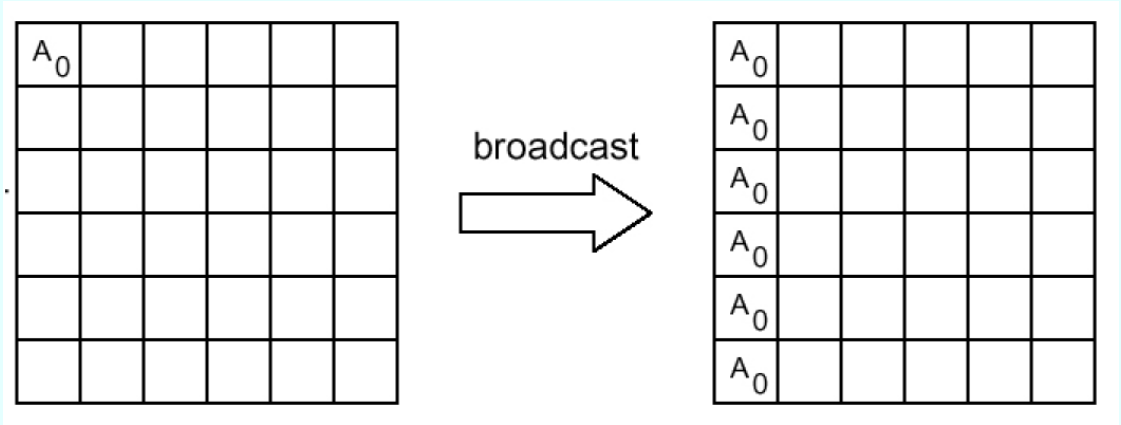
  count - число передаваемых элементов в сообщении

  datatype - тип передаваемых элементов

  source - номер рассылающего процесса

  comm - идентификатор коммуникатора

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

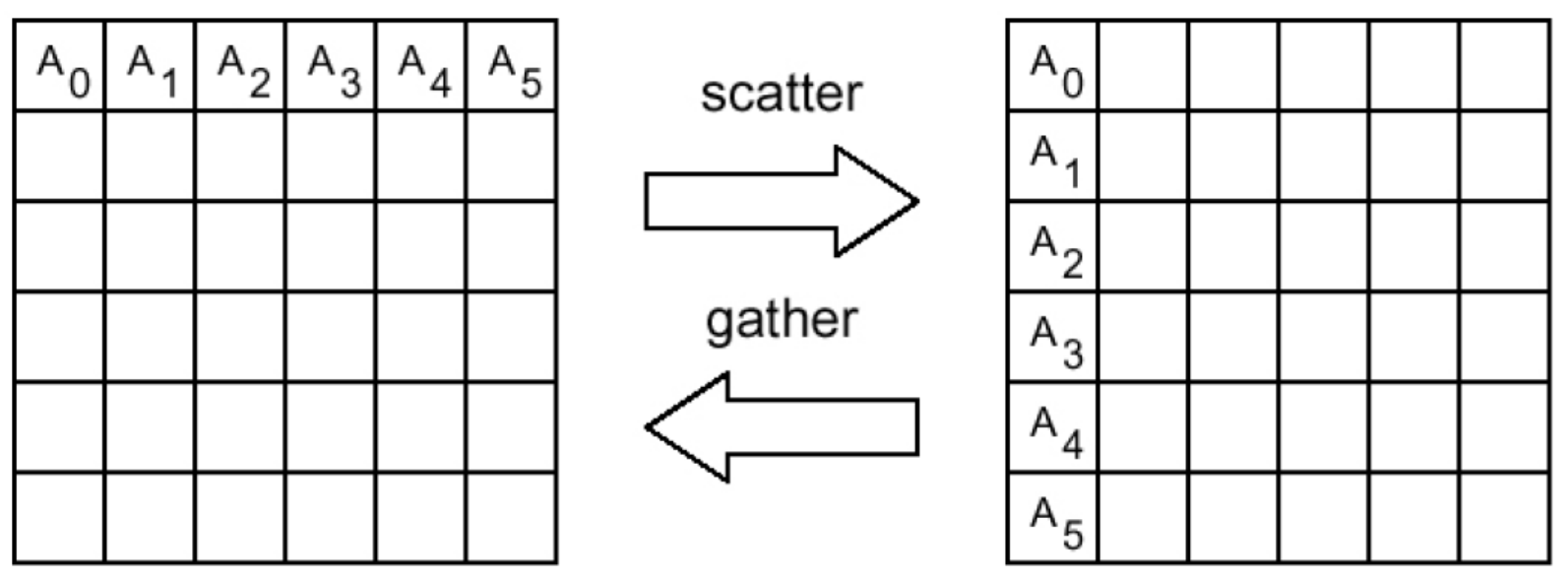


Пример

- Например, чтобы переслать от процесса 3 всем процессам в рамках коммуникатора comm массив buf из 100 целочисленных элементов, нужно, чтобы во всех процессах встретился вызов:

- MPI_Bcast(buf, 100, MPI_INT, 3, comm)

Сборка и рассылка данных



Сборка и рассылка данных

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

  sbuf - адрес начала буфера посылки

  scount - число элементов в посылаемом сообщении

  stype - тип элементов отсылаемого сообщения

  rbuf – получаемый адрес начала буфера сборки данных

  rcount - число элементов в принимаемом сообщении

  rtype - тип элементов принимаемого сообщения

  dest - номер процесса, на котором происходит сборка данных

  comm - идентификатор коммуникатора
- Сборка данных со всех процессов в буфере **rbuf** процесса **dest**. Каждый процесс, включая **dest**, посылает содержимое своего буфера **sbuf** процессу **dest**. Собирающий процесс сохраняет данные в буфере **rbuf**, располагая их в порядке возрастания номеров процессов. Параметр **rbuf** имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров **comm** и **dest** должны быть одинаковыми у всех процессов.
- Вместо sbuf на процессе dest можно использовать **MPI_IN_PLACE**

Сборка и рассылка данных

- **int MPI_Gatherv(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcounts[], int displs[], MPI_Datatype rtype, int dest,**

  sbuf - адрес начала буфера посылки

  scount - число элементов в посылаемом сообщении (может быть различным у разных процессов)

  stype - тип элементов отсылаемого сообщения

  rbuf - получаемый адрес начала буфера сборки данных

  rcounts - массив, в котором i элемент указывает число элементов в принимаемом сообщении от i-го процесса.

  displs – целочисленный массив, в котором i элемент указывает смещение в буфере rbuf для принимаемого сообщения от i-го процесса

  rtype - тип элементов принимаемого сообщения

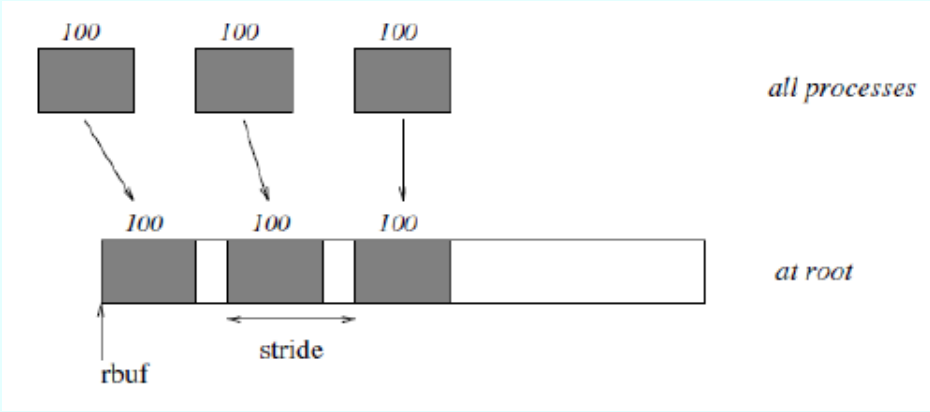
  dest - номер процесса, на котором происходит сборка данных

  comm - идентификатор коммуникатора

- Сборка данных со всех процессов в буфере **rbuf** процесса **dest**. Каждый процесс, включая **dest**, посылает содержимое своего буфера **sbuf** процессу **dest**. Собирающий процесс сохраняет данные в буфере **rbuf**, располагая их в соответствии со смещениями из **displs**. Параметр **rbuf** имеет значение только на собирающем процессе и на остальных игнорируется. Значения параметров **comm** и **dest** должны быть одинаковыми у всех процессов. **Должно быть соответствие между scount на разных процессах и элементами массива rcounts[].** Можно использовать **MPI_IN_PLACE.**

Пример

```
MPI_Comm comm;
int size, sbuf[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
...
MPI_Comm_size(comm, &size);
rbuf= (int*)malloc(size*stride*sizeof(int));
displs= (int*)malloc(size*sizeof(int));
rcounts= (int*)malloc(size*sizeof(int));
for (i=0; i<size; ++i) {
  displs[i] = i*stride;
  rcounts[i] = 100;
}
MPI_Gatherv(sbuf, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT, root, comm);
```



Сборка и рассылка данных

- **int MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, int source, MPI_Comm comm)**

  sbuf - адрес начала буфера посылки

  scount - число элементов посылаемых каждому процессу

  stype - тип элементов

  rbuf – получаемый адрес начала буфера сборки данных

  rcount - число элементов в принимаемом сообщении

  rtype - тип элементов принимаемого сообщения

  source - номер процесса, который рассылает данные

  comm - идентификатор коммуникатора
- Рассылка данных с процесса **source** всем процессам из буфера **sbuf**. Каждый процесс, включая **source**, принимает в буфер **rbuf** свою часть данных из буфера **sbuf** процесса **source**. Данные рассылаются в порядке возрастания номеров процессов. На процессе **source** существенными являются все параметры, на остальных только **rbuf**, **rcount**, **rtype**, **source** и **comm**.
- Значения параметров **comm** и **source** должны быть одинаковыми у всех процессов.
- Вместо **rbuf** на процессе **source** можно использовать **MPI_IN_PLACE.**
- Процесс **source не пересылает сам себе** блок из **sbuf.**

Сборка и рассылка данных

- **int MPI_Scatterv(void* sbuf, int scounts[], int displs[], MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, int source, MPI_Comm comm)**

  sbuf - адрес начала буфера посылки

  scounts - целочисленный массив, i-ыйэлемент число элементов посылаемых i-ому процессу

  displs – целочисленныймассив, в котором i-ыйэлемент указывает смещение в буфере sbufпри отправке сообщения i-ому процессу

  stype - тип элементов

  rbuf - получаемый адрес начала буфера сборки данных

  rcount - число элементов в принимаемом сообщении

  rtype - тип элементов принимаемого сообщения

  source - номер процесса, который рассылает данные

  comm - идентификатор коммуникатора

- Рассылка данных с процесса source всем процессам из буфера **sbuf**. Процесс source отсылает i-ому процессу **scounts[i]** элементов из буфера **sbuf** начиная с **displs[i]**. На процессе source существенными являются все параметры, на остальных только **rbuf**, **rcount**, **rtype**, **source** и **comm**.
- Значения параметров comm и source должны быть одинаковыми у всех процессов. **Должно быть соответствие между элементами массива scounts[]и rcount на разных процессах.** Использование **MPI_IN_PLACE** аналогично **MPI_Scatter.**

Процедуры редукции MPI

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

  sbuf - адрес начала буфера для аргументов

  rbuf - получаемый адрес начала буфера для результата

  count - число аргументов у каждого процесса

  datatype - тип аргументов

  op - идентификатор глобальной операции (например: **MPI_SUM**, **MPI_PROD**)

  root - процесс-получатель результата

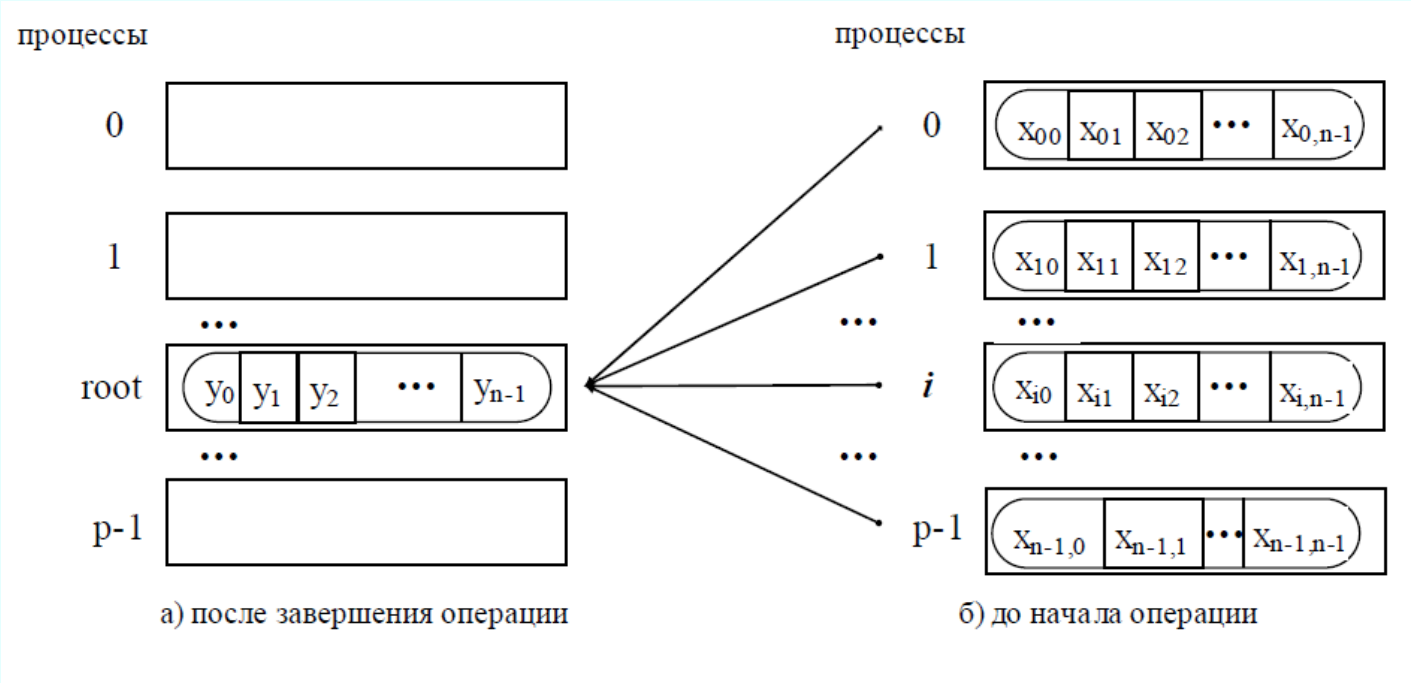
  comm - идентификатор коммуникатора
- Выполнение **count** глобальных операций **op** с возвратом **count** результатов в буфере **rbuf** процесса **root**. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров **count**, **datatype** и **comm** у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция **op** обладает свойствами ассоциативности и коммутативности(но, может не обладать свойством коммутативности).
- На процессе **root** вместо **sbuf** можно использовать **MPI_IN_PLACE**, в этом случае содержимое sbuf должно находиться в **rbuf**.

Типы предопределенных глобальных операций

- **MPI_MAX**, **MPI_MIN** – максимальное и минимальное значения;
- **MPI_SUM**, **MPI_PROD** – глобальная сумма и глобальное произведение;
- **MPI_LAND**, **MPI_LOR**, **MPI_LXOR** – логические “И”, “ИЛИ”, искл. “ИЛИ”;
- **MPI_BAND**, **MPI_BOR**, **MPI_BXOR** – побитовые “И”, “ИЛИ”, искл. “ИЛИ”;
- **MPI_MAXLOC**, **MPI_MINLOC** – максимальное и минимальное значения и их расположение

Процедура редукции

- Общая схема операции сбора и обработки на одном процессе данных от всех процессов



Последовательный алгоритм умножения матриц

Умножение матриц:

$$C = A \cdot B$$

или

$$\begin{pmatrix} c_{0,0}, & c_{0,1}, & \dots, & c_{0,l-1} \\ & & \dots & \\ c_{m-1,0}, & c_{m-1,1}, & \dots, & c_{m-1,l-1} \end{pmatrix} = \begin{pmatrix} a_{0,0}, & a_{0,1}, & \dots, & a_{0,n-1} \\ & & \dots & \\ a_{m-1,0}, & a_{m-1,1}, & \dots, & a_{m-1,n-1} \end{pmatrix} \begin{pmatrix} b_{0,0}, & b_{0,1}, & \dots, & a_{0,l-1} \\ & & \dots & \\ b_{n-1,0}, & b_{n-1,1}, & \dots, & b_{n-1,l-1} \end{pmatrix}$$

Задача умножения матрицы на матрицу может быть сведена к выполнению $m \cdot l$ независимых операций умножения строк матрицы A на столбцы матрицы B

$$c_{ij} = \left(a_i, b_j^T\right) = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l$$

В основу организации параллельных вычислений может быть положен принцип распараллеливания по данным

Ленточные алгоритмы умножения матриц

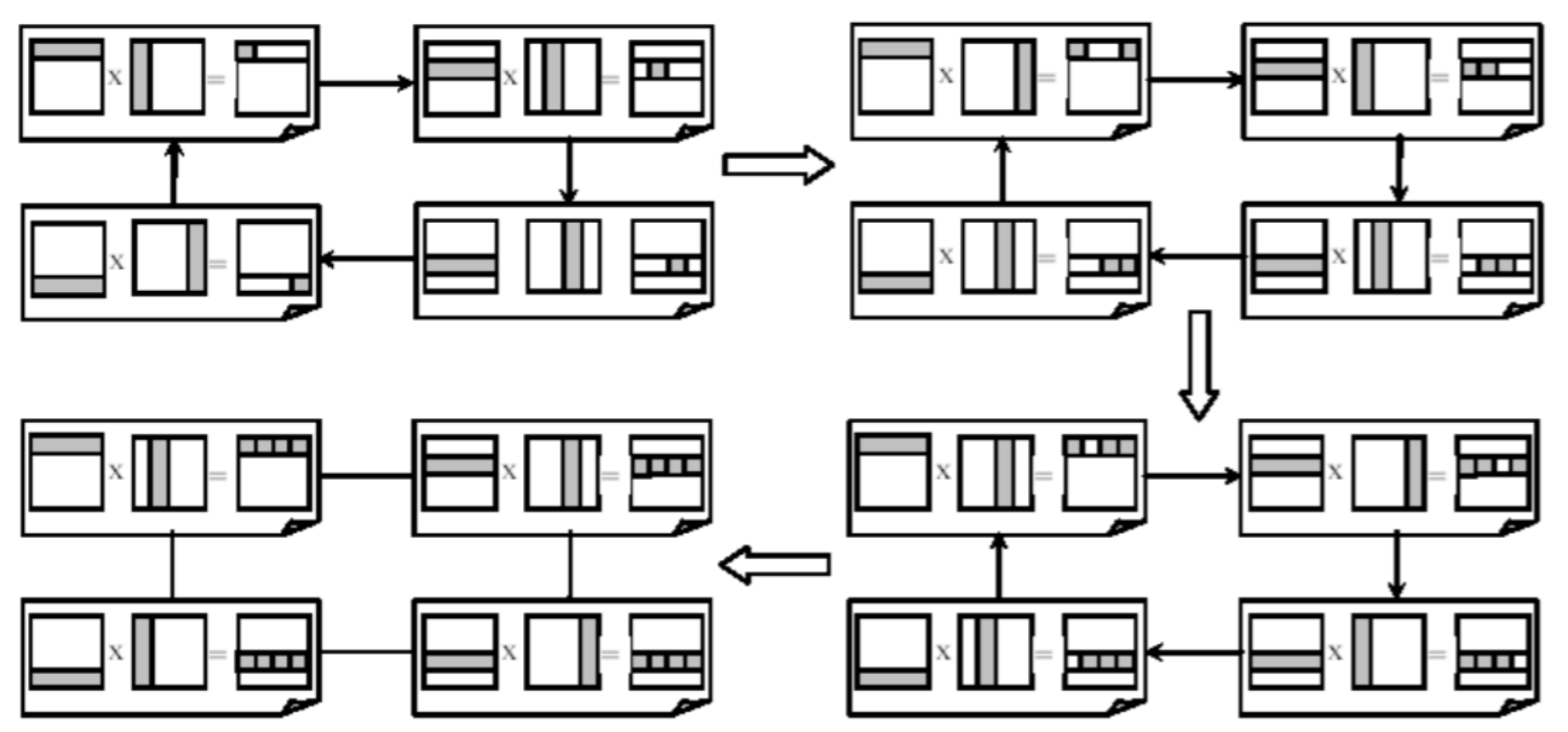
- Ленточная схема 1 умножения матриц

  Разбиение матрицы A по строкам, матрицы B по столбцам

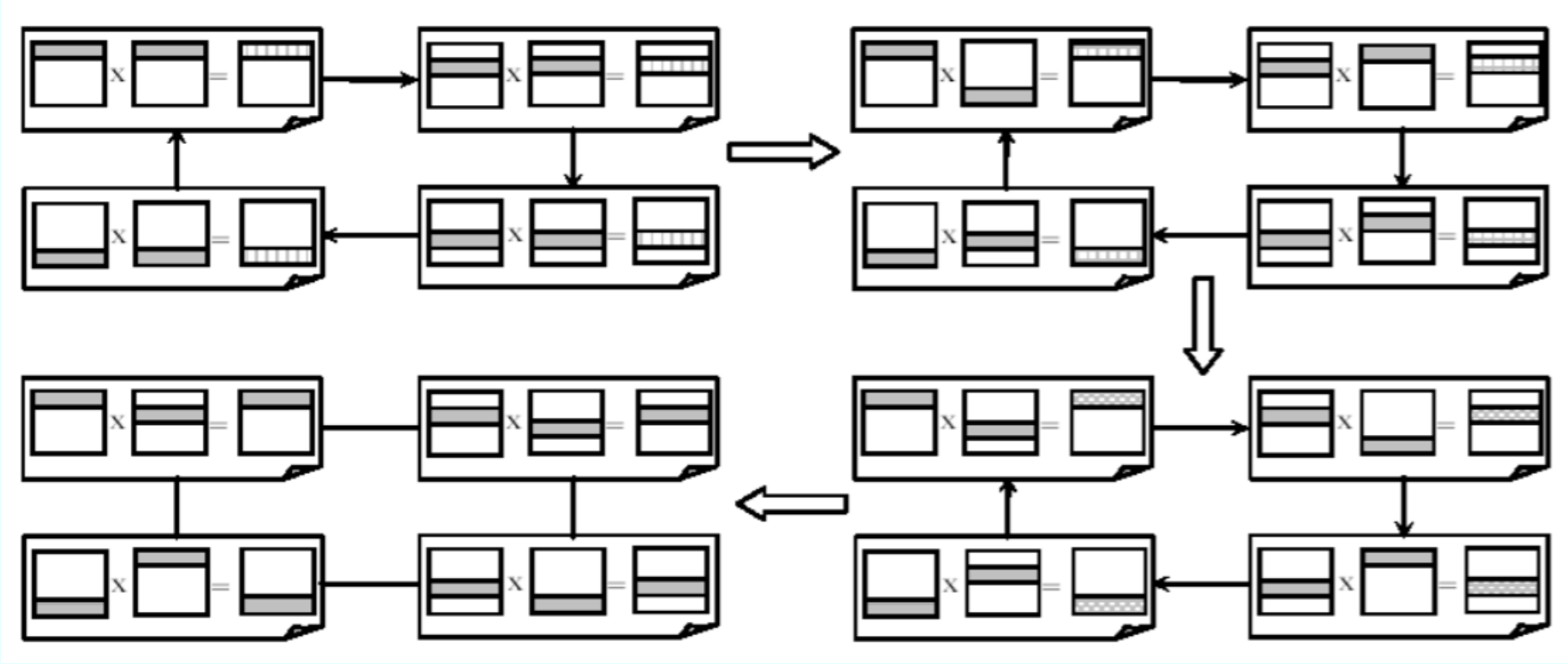
- Ленточная схема 2 умножения матриц

  Разбиение матрицы A по строкам, матрицы B по строкам

Ленточная схема 1



Ленточная схема 2



Задание

- Реализовать один из методов по выбору:
  ленточная схема 1 или ленточная схема 2