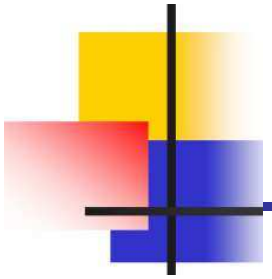


Параллельное программирование на основе MPI

2 часть

Содержание

- Операции передачи данных между двумя процессорами
  - Режимы передачи данных
  - Организация неблокирующих обменов данными между процессорами
  - Одновременное выполнение передачи и приема
- Коллективные операции передачи данных
  - Обобщенная передача данных от всех процессов одному процессу
  - Обобщенная передача данных от одного процесса всем процессам
  - Общая передача данных от всех процессов всем процессам
  - Дополнительные операции редукции данных
- Производные типы данных в MPI
  - Способы конструирования производных типов данных
- Заключение

Операции передачи данных между двумя процессами…

Режимы передачи данных…

- Стандартный (*Standard*):
  - обеспечивается функцией MPI_Send,
  - на время выполнения функции процесс-отправитель сообщения блокируется,
  - после завершения функции буфер может быть использован повторно,
  - состояние отправленного сообщения может быть различным - сообщение может располагаться в процессе-отправителе, может находиться в процессе передачи, может храниться в процессе-получателе или же может быть принято процессом-получателем при помощи функции *MPI_Recv*.

Операции передачи данных между двумя процессами…

Режимы передачи данных…

- *Синхронный* (*Synchronous*) *режим*
  - завершение функции отправки сообщения происходит только при получении от процесса-получателя подтверждения о начале приема отправленного сообщения:

    `MPI_Ssend` – функция отправки сообщения в синхронном режиме

- *Режим передачи по готовности (Ready)*
  - может быть использован только, если операция приема сообщения уже инициирована. Буфер сообщения после завершения функции отправки сообщения может быть повторно использован.

    `MPI_Rsend` – функция отправки сообщения в режиме по готовности

Операции передачи данных между двумя процессами…

Режимы передачи данных…

- *Буферизованный* (*Buffered*) *режим*
  - используются дополнительные системные буферы для копирования в них отправляемых сообщений; функция отправки сообщения завершается сразу же после копирования сообщения в буфер,

    MPI_Bsend – функция отправки сообщения в буферизованном режиме**,**
  - Для использования буферизованного режима передачи должны быть создан и передан MPI буфер памяти:

```
int MPI_Buffer_attach(void *buf, int size), где
   - buf  - буфер памяти для буферизации сообщений,
   - size – размер буфера.
```

  - После завершения работы с буфером он должен быть отключен от MPI при помощи функции:

```
int MPI_Buffer_detach(void *buf, int *size)
```

Операции передачи данных между двумя процессами…

Режимы передачи данных…

- Режим передачи *по готовности* формально является наиболее быстрым, но используется достаточно редко, т.к. обычно сложно гарантировать готовность операции приема,
- *Стандартный* и *буферизованный* режимы также выполняются достаточно быстро, но могут приводить к большим расходам ресурсов (памяти), могут быть рекомендованы для передачи коротких сообщений,
- *Синхронный* режим является наиболее медленным, т.к. требует подтверждения приема. В тоже время, этот режим наиболее надежен – можно рекомендовать его для передачи длинных сообщений.

Операции передачи данных между двумя процессами…

Организация неблокирующих обменов данными между процессорами…

- *Блокирующие функции* приостанавливают выполнение процессов до момента завершения работы вызванных функций,
- *Неблокирующие функции* обмена данными выполняются без блокировки процессов для совмещения процессов передачи сообщений и вычислений:
  - Более сложен для использования,
  - Может в значительной степени уменьшить потери эффективности параллельных вычислений из-за медленных коммуникационных операций.

Операции передачи данных между двумя процессами…

Организация неблокирующих обменов данными между процессорами…

Наименование неблокирующих аналогов образуется из названий соответствующих функций путем добавления префикса **I** (*Immediate*).

```
int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest,
      int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Issend(void *buf, int count, MPI_Datatype type,
      int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Ibsend(void *buf, int count, MPI_Datatype type,
      int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irsend(void *buf, int count, MPI_Datatype type,
      int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype type,
      int source, int tag, MPI_Comm comm, MPI_Request *request);
```

Операции передачи данных между двумя процессами…

Организация неблокирующих обменов данными между процессорами…

- Проверка состояния выполняемой неблокирующей операции передачи данных выполняется при помощи функции:

```
int MPI_Test( MPI_Request *request, int *flag,
      MPI_status *status),
где
- request - дескриптор операции, определенный при вызове
      неблокирующей функции,
- flag    - результат проверки (=true, если операция завершена),
- status  - результат выполнения операции обмена (только для
      завершенной операции.
```

Операция проверки является неблокирующей.

Операции передачи данных между двумя процессами…

Организация неблокирующих обменов данными между процессорами…

- Возможная схема вычислений и выполнения неблокирующей операции обмена:

```
MPI_Isend(buf,count,type,dest,tag,comm,&request);
…
do {
  …
  MPI_Test(&request,&flag,&status);
} while ( !flag );
```

- Блокирующая операция ожидания завершения операции:

```
int MPI_Wait( MPI_Request *request, MPI_status *status);
```

Операции передачи данных между двумя процессами…

Организация неблокирующих обменов данными между процессорами

- Дополнительные функций проверки и ожидания неблокирующих операций обмена:

```
MPI_Testall  - проверка завершения всех перечисленных операций
               обмена,
MPI_Waitall  – ожидание завершения всех операций обмена,
MPI_Testany  - проверка завершения хотя бы одной из перечисленных
               операций обмена,
MPI_Waitany  – ожидание завершения любой из перечисленных операций
               обмена,
MPI_Testsome - проверка завершения каждой из перечисленных операций
               обмена,
MPI_Waitsome - ожидание завершения хотя бы одной из перечисленных
               операций обмена и оценка состояния по всем операциям.
```

Операции передачи данных между двумя процессами

Одновременное выполнение передачи и приема

- Функция, позволяющая эффективно одновременно выполнить передачу и прием данных:

```
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),
где
- sbuf, scount, stype, dest,   stag - параметры передаваемого сообщения,
- rbuf, rcount, rtype, source, rtag - параметры принимаемого сообщения,
- comm - коммуникатор, в рамках которого выполняется передача данных,
- status – структура данных с информацией о результате выполнения операции.
```

- В случае, когда сообщения имеют одинаковый тип, имеется возможность использования единого буфера:

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype type,
  int dest, int stag, int source, int rtag, MPI_Comm comm,
  MPI_Status *status).
```

Коллективные операции передачи данных…

Под *коллективными операциями* в MPI понимаются операции данных, в которых принимают участие все процессы используемого коммуникатора

Коллективные операции передачи данных…

Обобщенная передача данных от одного процесса всем процессам…

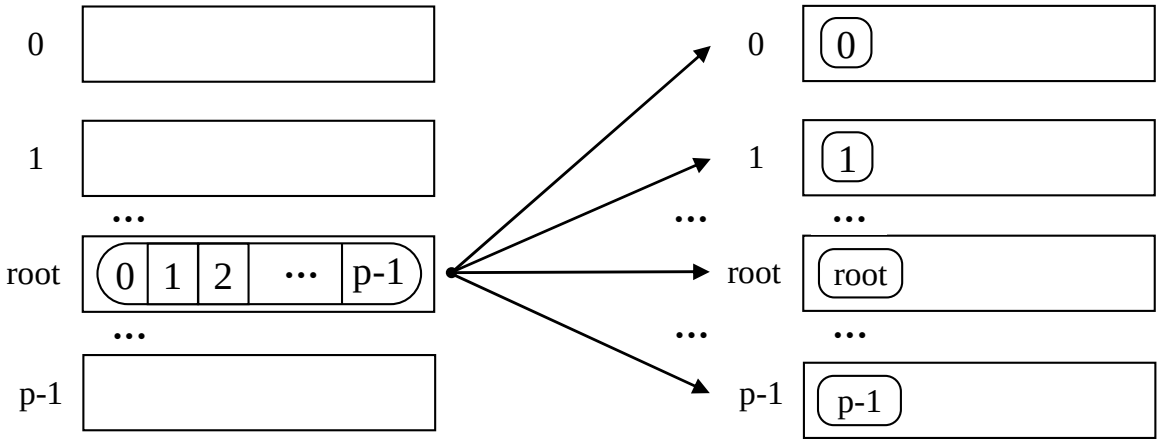
- *Распределение данных* – ведущий процесс (*root*) передает процессам различающиеся данные

```
int MPI_Scatter(void *sbuf,int scount,MPI_Datatype stype,
                void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
где
- sbuf, scount, stype - параметры передаваемого сообщения (scount
      определяет количество элементов, передаваемых на каждый
процесс),
- rbuf, rcount, rtype - параметры сообщения, принимаемого в
процессах,
- root – ранг процесса, выполняющего рассылку данных,
- comm - коммуникатор, в рамках которого выполняется передача данных.
```

Коллективные операции передачи данных…

Обобщенная передача данных от одного процесса всем процессам

- Вызов *MPI_Scatter* при выполнении рассылки данных должен быть обеспечен в каждом процессе коммуникатора,
- *MPI_Scatter* передает всем процессам сообщения одинакового размера. Если размеры сообщений для процессов могут быть разными, следует использовать функцию *MPI_Scatterv*.



а) до начала операции

б) после завершения операции

Коллективные операции передачи данных…

Обобщенная передача данных от всех процессов одному процессу…

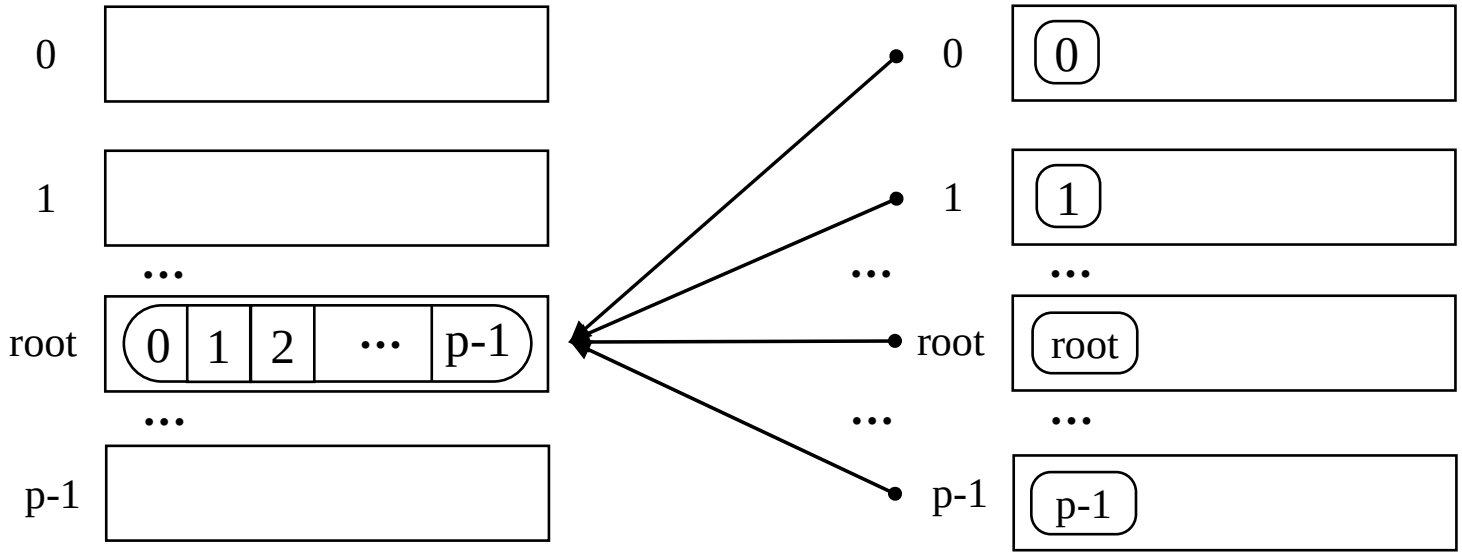
- Передача данных от всех процессоров одному процессу (*сбор данных*) является обратной к операции распределения данных

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype stype,
               void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
где
 - sbuf, scount, stype - параметры передаваемого сообщения,
 - rbuf, rcount, rtype - параметры принимаемого сообщения,
 - root – ранг процесса, выполняющего сбор данных,
 - comm - коммуникатор, в рамках которого выполняется передача
данных.
```

Коллективные операции передачи данных…

Обобщенная передача данных от всех процессов одному процессу…

- *MPI_Gather* определяет коллективную операцию, и ее вызов при выполнении сбора данных должен быть обеспечен в каждом процессе коммуникатора



а) после завершения операции

б) до начала операции

Коллективные операции передачи данных…

Обобщенная передача данных от всех процессов одному процессу

- *MPI_Gather* собирает данные на одном процессе. Для получения всех собираемых данных на каждом процессе нужно использовать *функцию сбора и рассылки*:

```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype
stype,
    void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm
comm).
```

- В случае, когда размеры передаваемых процессами сообщений могут быть различны, для передачи данных необходимо использовать функции *MPI_Gatherv* и *MPI_Allgatherv*.

Коллективные операции передачи данных…

Пример использования функций **MPI_Scatter** и **MPI_Gather** в параллельной программе умножения матрицы на вектор

Постановка задачи

В результате умножения матрицы **A** размерности **m × n** и вектора **b**, состоящего из **n** элементов, получается вектор **c** размера **m**, каждый **i**-й элемент которого есть результат скалярного умножения **i**-й строки матрицы **A** (обозначим эту строчку $\mathbf{a}_i$) и вектора **b**:

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, 1 \le i \le m.$$

Тем самым получение результирующего вектора **c** предполагает повторение **m** однотипных операций по умножению строк матрицы **A** и вектора b. Каждая такая операция включает умножение элементов строки матрицы и вектора **b** (n операций) и последующее суммирование полученных произведений (n-1 операций). Общее количество необходимых скалярных операций есть величина: **T1=m·(2n-1)**

Коллективные операции передачи данных…

Последовательный алгоритм

Последовательный алгоритм умножения матрицы на вектор может быть представлен следующим образом.

```
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < m; i++){
   c[i] = 0;
   for (j = 0; j < n; j++){
      c[i] += A[i][j]*b[j]
    }
}
```

Коллективные операции передачи данных…

Принципы распараллеливания

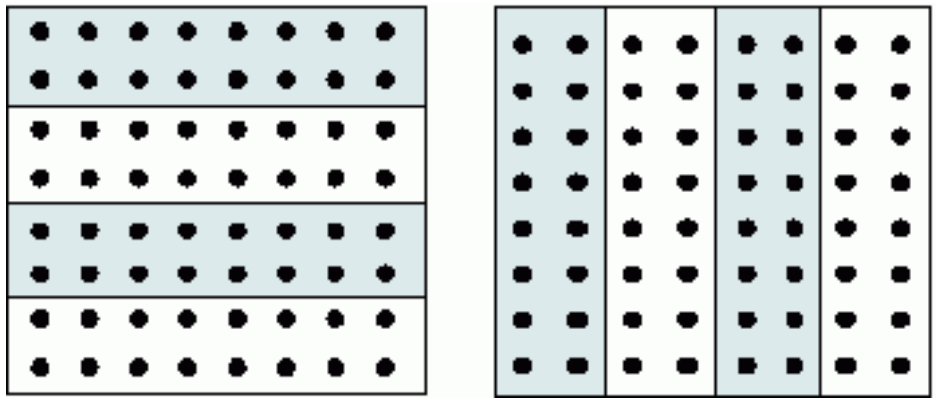
- Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данное свойство свидетельствует о наличии параллелизма по данным при выполнении матричных расчетов, и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд параллельных алгоритмов матричных вычислений.
- Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на полосы (по вертикали или горизонтали) или на прямоугольные фрагменты (блоки).

Коллективные операции передачи данных…

Ленточное разбиение матрицы.

При ленточном (block-striped) разбиении каждому процессору выделяется то или иное подмножество строк (rowwise или горизонтальное разбиение) или столбцов (columnwise или вертикальное разбиение) матрицы. Разделение строк и столбцов на полосы в большинстве случаев происходит на непрерывной (последовательной) основе. При таком подходе для горизонтального разбиения по строкам, например, матрица A представляется в виде.

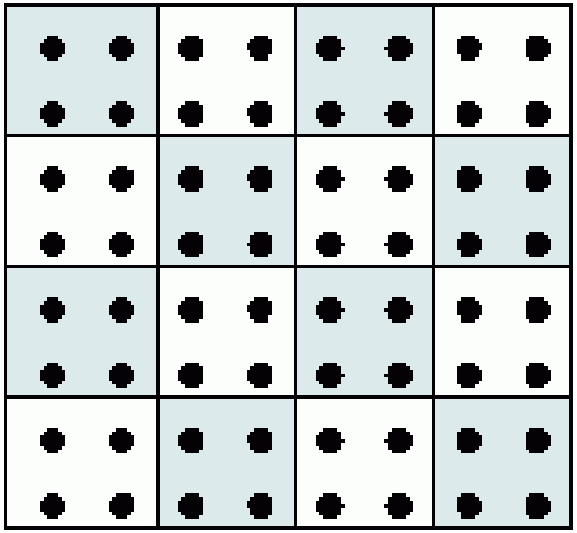
$$A = (A_0, A_1, \ldots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}), i_j = ik+j, 0 \le j < k, k = m/p,$$



Коллективные операции передачи данных…

Блочное разбиение матрицы.

При блочном (chessboard block) разделении матрица делится на прямоугольные наборы элементов – при этом, как правило, используется разделение на непрерывной основе. Пусть количество процессоров составляет $p = s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m = k \cdot s$ и $n = l \cdot q$. Представим исходную матрицу A в виде набора прямоугольных блоков следующим образом:



При таком подходе целесообразно, чтобы вычислительная система имела физическую или, по крайней мере, логическую топологию процессорной решетки из s строк и q столбцов. В этом случае при разделении данных на непрерывной основе процессоры, соседние в структуре решетки, обрабатывают смежные блоки исходной матрицы.

Коллективные операции передачи данных…

Разделение данных.

- При выполнении параллельных алгоритмов умножения матрицы на вектор, кроме матрицы A, необходимо разделить еще вектор b и вектор результата c. Элементы векторов можно продублировать, то есть скопировать все элементы вектора на все процессоры, составляющие многопроцессорную вычислительную систему, или разделить между процессорами. При блочном разбиении вектора из n элементов каждый процессор обрабатывает непрерывную последовательность из k элементов вектора (мы предполагаем, что размерность вектора n нацело делится на число процессоров, т.е. $n = k \cdot p$).
- Поясним, почему дублирование векторов b и c между процессорами является допустимым решением (далее для простоты изложения будем полагать, что m=n). Векторы b и c состоят из n элементов, т.е. содержат столько же данных, сколько и одна строка или один столбец матрицы. Если процессор хранит строку или столбец матрицы и одиночные элементы векторов b и c, то общее число сохраняемых элементов имеет порядок O(n). Если процессор хранит строку (столбец) матрицы и все элементы векторов b и c, то общее число сохраняемых элементов также порядка O(n). Таким образом, при дублировании и при разделении векторов требования к объему памяти из одного класса сложности.

Коллективные операции передачи данных…

Умножение матрицы на вектор при разделении данных по строкам.

❑Рассмотрим алгоритм умножения матрицы на вектор, основанный на представлении матрицы непрерывными наборами (горизонтальными полосами) строк. При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция скалярного умножения одной строки матрицы на вектор.

Выделение информационных зависимостей

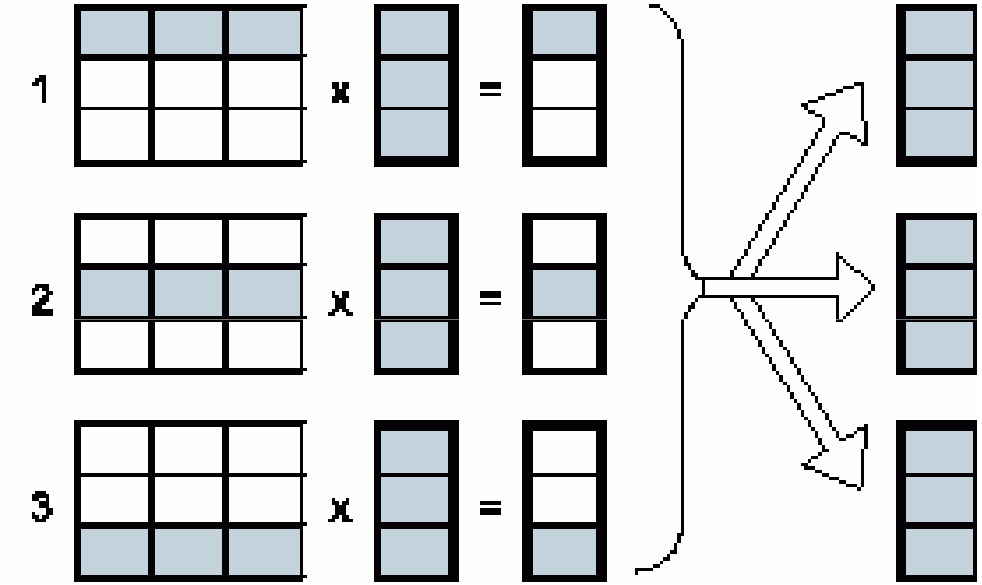
❑Для выполнения базовой подзадачи скалярного произведения процессор должен содержать соответствующую строку матрицы **A** и копию вектора **b**. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата **c**.

❑Для объединения результатов расчета и получения полного вектора **c** на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных, в которой каждый процессор передает свой вычисленный элемент вектора с всем остальным процессорам. Этот шаг можно выполнить, например, с использованием функции **MPI_Allgather** из библиотеки MPI.

❑В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений показана на следующем рисунке.

Коллективные операции передачи данных…

Умножение матрицы на вектор при разделении данных по строкам



Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

Коллективные операции передачи данных…

Масштабирование и распределение подзадач по процессорам

В процессе умножения плотной матрицы на вектор количество вычислительных операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае когда число процессоров **p** меньше числа базовых подзадач **m**, мы можем объединить базовые подзадачи таким образом, чтобы каждый процессор выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы A. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора **c**.

- Распределение подзадач между процессорами вычислительной системы может быть выполнено произвольным образом.

Коллективные операции передачи данных…

Программная реализация

- Представим возможный вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по строкам. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимании общей схемы параллельных вычислений.

Алгоритм 1: ленточная схема (разбиение матрицы по строкам)…

- **Программная реализация**
  - Первый этап: Инициализация и распределение данных между процессорами:
    - Получение конкретных числовых данных для размера матрицы выделено в функции *GetRowSize* и *GetRowNum*,
    - определение исходных данных для матрицы ***A*** (переменная *pMatrix*) и вектора *b* (переменная *pVector*) на нулевом процессе (функция *DataGeneration*),
    - Пересылка исходных данных между процессорами вынесено в отдельную функцию *DataDistribution*.

Коллективные операции передачи данных…

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

```
void main(int argc, char* argv[]) {
double* pMatrix;     // Первый аргумент – исходная матрица
 double* pVector;    // Второй аргумент – исходный вектор
double* pResult;     // Результат умножения матрицы на вектор
 int Size;           // Размеры исходных матрицы и вектора
double* pProcRows;  double* pProcResult;
 int RowNum;
double Start, Finish, Duration;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

   // Выделение памяти и инициализация исходных данных
```

Коллективные операции передачи данных…

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

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

 // Параллельное выполнение умножения матрицы на вектор
ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);

// Сбор результирующего вектора на всех процессах
ResultReplication(pProcResult, pResult, Size, RowNum);

// Завершение процесса вычислений
 ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);
 MPI_Finalize();
 }
```

Коллективные операции передачи данных…

Функция ProcessInitialization. Эта функция задает размер и элементы для матрицы A и вектора b. Значения для матрицы A и вектора b определяются в функции RandomDataInitialization.

```
// Функция для выделения памяти и инициализации исходных данных
void ProcessInitialization (double* &pMatrix, double* &pVector,   double* &pResult,
double* &pProcRows, double* &pProcResult,   int &Size, int &RowNum) {
 int RestRows;      // Количество строк матрицы, которые еще
                    // не распределены
 int i;
 if (ProcRank == 0) {
   do {
      printf("\nВведите размер матрицы: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
         printf("Размер матрицы должен превышать количество процессов! \n ");
       }
   }
```

Коллективные операции передачи данных…

```
while (Size < ProcNum);
 }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
 RestRows = Size;
for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
RowNum = RestRows/(ProcNum-ProcRank);
 pVector = new double [Size];
 pResult = new double [Size];
 pProcRows = new double [RowNum*Size];
 pProcResult = new double [RowNum];
 if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    RandomDataInitialization(pMatrix, pVector, Size);
  }
}
```

Коллективные операции передачи данных…

Функция DataDistribution. Осуществляет рассылку вектора b и распределение строк исходной матрицы A по процессам вычислительной системы. Следует отметить, что когда количество строк матрицы n не является кратным числу процессоров p, объем пересылаемых данных для процессов может оказаться разным и для передачи сообщений необходимо использовать функцию MPI_Scatterv библиотеки MPI.

```
// Функция для распределения исходных данных между процессами
void DataDistribution(double* pMatrix, double* pProcRows,
      double* pVector, int Size, int RowNum) {
      int *pSendNum;          // Количество элементов, посылаемых процессу
      int *pSendInd;          // Индекс первого элемента данных,
                              // посылаемого процессу
      int RestRows=Size;      // Количество строк матрицы, которые еще
                              // не распределены
      MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

      // Выделение памяти для хранения временных объектов
      pSendInd = new int [ProcNum];
      pSendNum = new int [ProcNum];
```

Коллективные операции передачи данных…

```
// Определение положения строк матрицы, предназначенных
// каждому процессу
RowNum = (Size/ProcNum);
pSendNum[0] = RowNum*Size;
pSendInd[0] = 0;
for (int i=1; i<ProcNum; i++) {
  RestRows -= RowNum;
  RowNum = RestRows/(ProcNum-i);
  pSendNum[i] = RowNum*Size;
  pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
 }
 // Рассылка строк матрицы
MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Освобождение памяти
delete [] pSendNum;
delete [] pSendInd;
 }
```

Алгоритм 1: ленточная схема (разбиение матрицы по строкам)…

- **Программная реализация**
  - Второй этап: умножение на вектор тех строк матрицы, которые распределены на данный процесс и, таким образом, получение блока результирующего вектора ***b*** реализовано в функции **ParallelResultCalculation**

Программа

Коллективные операции передачи данных…

Функция ParallelResultCaculation. Данная функция производит умножение на вектор тех строк матрицы, которые распределены на данный процесс, и таким образом получается блок результирующего вектора c.

```
// Функция для вычисления части результирующего вектора
void ParallelResultCalculation(double* pProcRows, double* pVector,
       double* pProcResult, int Size, int RowNum) {
 int i, j;
 for (i=0; i<RowNum; i++) {
    pProcResult[i] = 0;
    for (j=0; j<Size; j++)
       pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
    }
}
```

Алгоритм 1: ленточная схема (разбиение матрицы по строкам)…

- **Программная реализация**
  - Третий этап: Функция *ResultReplication* объединяет блоки результирующего вектора ***b***, полученные на разных процессах, и копирует вектор результата на все процессы вычислительной системы (если количество строк матрицы ***n*** не является кратным числу процессоров ***p***, объем пересылаемых данных для процессов может оказаться разным, и для передачи сообщений необходимо использовать функцию *MPI_Allgatherv* библиотеки MPI)

Программа

Коллективные операции передачи данных…

Функция ResultReplication. Объединяет блоки результирующего вектора c, полученные на разных процессах, и копирует вектор результата на все процессы вычислительной системы.

```
// Функция для сбора результирующего вектора на всех процессах
void ResultReplication(double* pProcResult, double* pResult, int Size, int RowNum) {
int *pReceiveNum;  // Количество элементов, посылаемых процессом
int *pReceiveInd;     // Индекс элемента данных в результирующем  векторе
int RestRows=Size; // Количество строк матрицы, которые еще не
                          // распределены
int i;
 // Выделение памяти для временных объектов
pReceiveNum = new int [ProcNum];
pReceiveInd = new int [ProcNum];
// Определение положения блоков результирующего вектора
```

Коллективные операции передачи данных…

```
// Определение положения блоков результирующего вектора
pReceiveInd[0] = 0;
pReceiveNum[0] = Size/ProcNum;
for (i=1; i<ProcNum; i++) {
    RestRows -= pReceiveNum[i-1];
    pReceiveNum[i] = RestRows/(ProcNum-i);
pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
 }
// Сбор всего результирующего вектора на всех процессах
MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank],     MPI_DOUBLE, pResult,
pReceiveNum, pReceiveInd,     MPI_DOUBLE, MPI_COMM_WORLD);

// Освобождение памяти
delete [] pReceiveNum;
delete [] pReceiveInd;
}
```

Коллективные операции передачи данных…

Результаты вычислительных экспериментов

- Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности приведенного выше параллельного алгоритма умножения матрицы на вектор.
- Эксперименты проводились на вычислительном кластере на базе процессоров Intel Xeon 4 EM64T, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft Compute Cluster Server

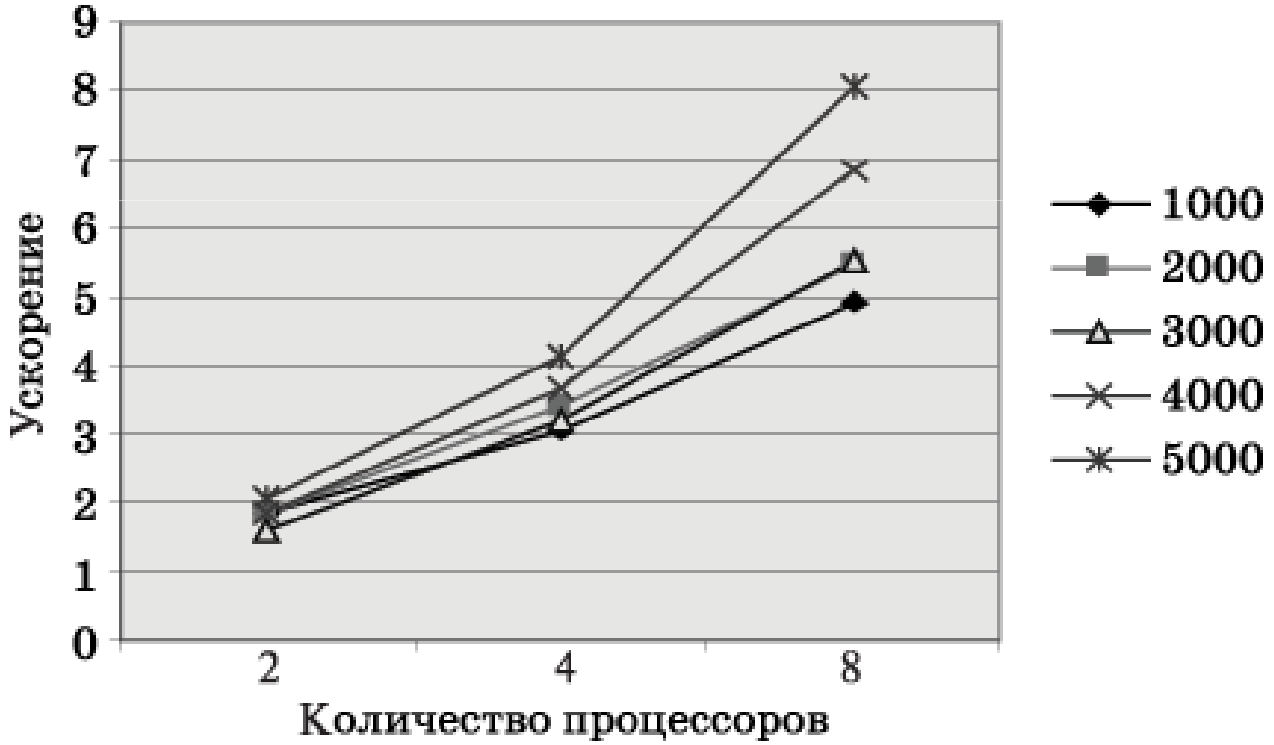
Коллективные операции передачи данных…

Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по столбцам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	0,0041	0,0022	1,8352	0,0015	3,1538	0,0008	4,9409
2000	0,016	0,0085	1,8799	0,0046	3,4246	0,0029	5,4682
3000	0,031	0,019	1,6315	0,0095	3,2413	0,0055	5,5456
4000	0,062	0,0331	1,8679	0,0168	3,6714	0,0090	6,8599
5000	0,11	0,0518	2,1228	0,0265	4,1361	0,0136	8,0580

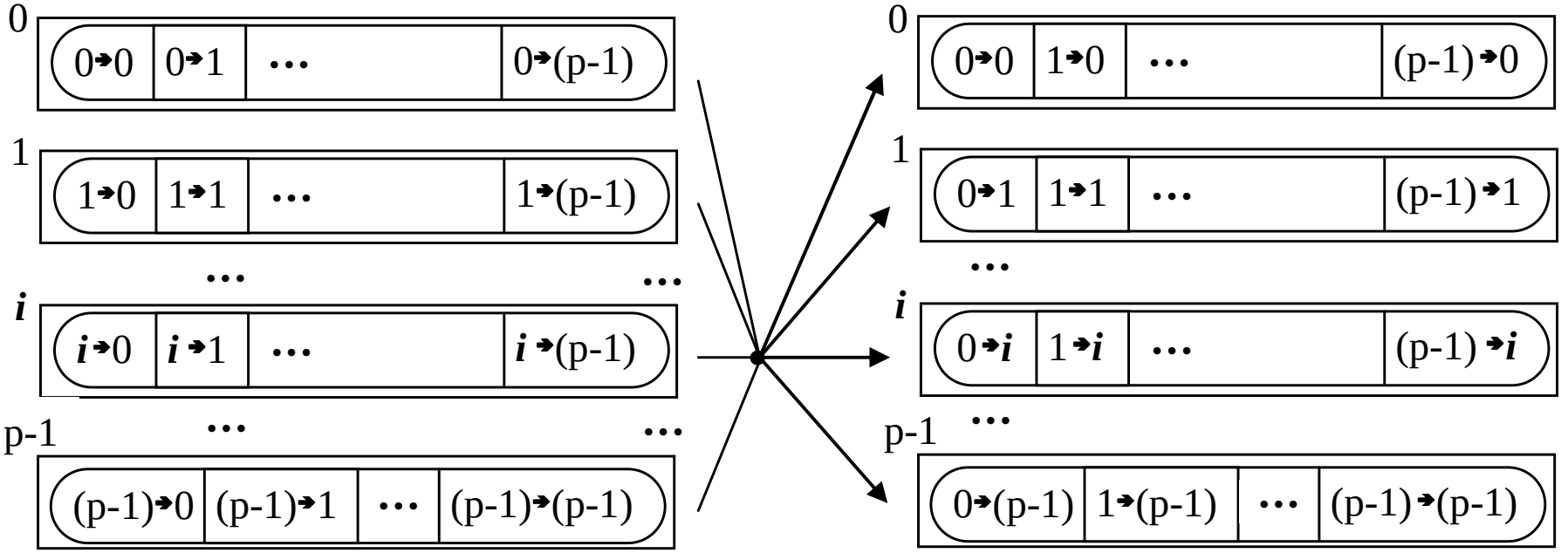
Коллективные операции передачи данных…

Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма умножения матрицы на вектор (ленточное разбиение матрицы по столбцам) для разных размеров матриц



Коллективные операции передачи данных…

Общая передача данных от всех процессов всем процессам…



а) до начала операции

б) после завершения операции

Коллективные операции передачи данных…

Обобщенная передача данных от всех процессов всем процессам

```
int MPI_Alltoall(void *sbuf,int scount,MPI_Datatype stype,
    void *rbuf,int rcount, MPI_Datatype rtype,MPI_Comm comm),
где
 - sbuf, scount, stype - параметры передаваемых сообщений,
 - rbuf, rcount, rtype - параметры принимаемых сообщений
 - comm - коммуникатор, в рамках которого выполняется передача
          данных.
```

- Вызов функции *MPI_Alltoall* при выполнении операции общего обмена данными должен быть выполнен в каждом процессе коммуникатора,
- Вариант операции общего обмена данных, когда размеры передаваемых процессами сообщений могут быть различны обеспечивается при помощи функций *MPI_Alltoallv*.

Коллективные операции передачи данных…

Дополнительные операции редукции данных…

- *MPI_Reduce* обеспечивает получение результатов редукции данных только на одном процессе,
- Функция *MPI_AllReduce редукции и рассылки* выполняет рассылку между процессами всех результатов операции редукции:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,int count,
                  MPI_Datatype type,MPI_Op op,MPI_Comm comm)
```

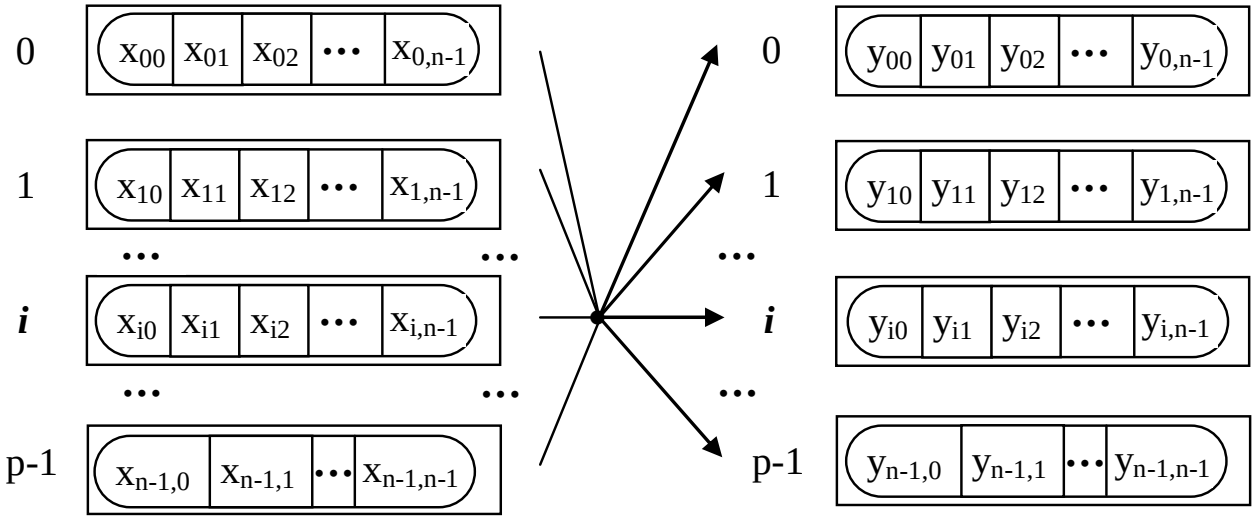
- Возможность управления распределением этих данных между процессами предоставляется функций *MPI_Reduce_scatter,*
- Функция MPI_Scan производит операцию сбора и обработки данных, при которой обеспечивается получение и всех частичных результатов редуцирования

```
int MPI_Scan(void *sendbuf, void *recvbuf,int count,
             MPI_Datatype type, MPI_Op op,MPI_Comm comm)
```

Коллективные операции передачи данных

Дополнительные операции редукции данных

- При выполнении функции *MPI_Scan* элементы получаемых сообщений представляют собой результаты обработки соответствующих элементов передаваемых процессами сообщений, при этом для получения результатов на процессе с рангом *i*, $0 \le i < n$, используются данные от процессов, ранг которых меньше или равен *i*



а) до начала операции

б) после завершения операции

Производные типы данных в MPI…

- Во всех ранее рассмотренных примерах использования функций передачи данных предполагалось, что сообщения представляют собой некоторый непрерывный вектор элементов предусмотренного в MPI типа,
- В общем случае, необходимые к пересылке данные могут располагаться не рядом и состоять из разного типа значений:
  - разрозненные данные могут быть переданы с использованием нескольких сообщений (такой способ ведет к накоплению латентности множества выполняемых операций передачи данных),
  - разрозненные данные могут быть предварительно упакованы в формат того или иного непрерывного вектора (появление лишних операций копирования данных).

Производные типы данных в MPI…

- *Производный тип данных* в MPI - описание набора значений предусмотренного в MPI типа, значения могут не располагаться непрерывно по памяти:
  - Задание типа в MPI принято осуществлять при помощи *карты типа* (*type map*) в виде последовательности описаний входящих в тип значений, каждое отдельное значение описывается указанием типа и смещения адреса месторасположения от некоторого базового адреса:

    TypeMap = {($type_0$, $disp_0$), ($type_1$, $disp_1$), … , ($type_{n-1}$, $disp_{n-1}$)}

  - Часть карты типа с указанием только типов значений именуется в MPI *сигнатурой типа*:

    TypeSignature = {$type_0$, $type_1$, … , $type_{n-1}$}

Производные типы данных в MPI…

- **Пример.**
  - Пусть в сообщение должны входить значения переменных:

```
double a; /* адрес 24 */
double b; /* адрес 40 */
int    n; /* адрес 48 */
```

  - Тогда производный тип для описания таких данных должен иметь карту типа следующего вида:

```
{(MPI_DOUBLE,0),
 (MPI_DOUBLE,16),
 (MPI_INT,24)
}
```

Производные типы данных в MPI…

- Для производных типов данных в MPI используется следующих ряд новых понятий:
  - *нижняя граница* типа:

$$lb\ (TypeMap) = \min_j (disp_j)$$

  - *верхняя граница* типа:

$$ub\ (TypeMap) = \max_j (disp_j + sizeof\ (type_j)) + \Delta$$

  - *протяженность* типа (размер памяти в байтах, который нужно отводить для одного элемента производного типа):

$$extent\ (TypeMap) = ub\ (TypeMap) - lb\ (TypeMap)$$

  - *размер* типа данных - это число байтов, которые занимают данные.

Различие в значениях протяженности и размера состоит в величине округления для выравнивания адресов.

Производные типы данных в MPI…

- Для получения значения протяженности и размера типа в MPI предусмотрены функции:

```
int MPI_Type_extent ( MPI_Datatype type, MPI_Aint *extent );
int MPI_Type_size   ( MPI_Datatype type, MPI_Aint *size );
```

- Определение нижней и верхней границ типа может быть выполнено при помощи функций:

```
int MPI_Type_lb ( MPI_Datatype type, MPI_Aint *disp );
int MPI_Type_ub ( MPI_Datatype type, MPI_Aint *disp );
```

- Важной и необходимой при конструировании производных типов является функция получения адреса переменной:

```
int MPI_Address ( void *location, MPI_Aint *address );
```

Производные типы данных в MPI…

- Способы конструирования производных типов данных:
  - ***Непрерывный*** способ позволяет определить непрерывный набор элементов существующего типа как новый производный тип,
  - ***Векторный*** способ обеспечивает создание нового производного типа как набора элементов существующего типа, между элементами которого существуют регулярные промежутки по памяти. При этом, размер промежутков задается в числе элементов исходного типа,
  - ***Индексный*** способ отличается от векторного метода тем, что промежутки между элементами исходного типа могут иметь нерегулярный характер,
  - ***Структурный*** способ обеспечивает самое общее описание производного типа через явное указание карты создаваемого типа данных.

Производные типы данных в MPI…

- **Непрерывный способ конструирования**:

```
int MPI_Type_contiguous(int count,MPI_Data_type oldtype,
                        MPI_Datatype *newtype);
```

  – Как следует из описания, новый тип *newtype* создается как *count* элементов исходного типа *oldtype*. Например, если исходный тип данных имеет карту типа

```
{ (MPI_INT,0),(MPI_DOUBLE,8) },
```

  то вызов функции *MPI_Type_contiguous* с параметрами

```
MPI_Type_contiguous (2, oldtype, &newtype);
```

  приведет к созданию типа данных с картой типа:

```
{ (MPI_INT,0),(MPI_DOUBLE,8),(MPI_INT,16),(MPI_DOUBLE,24) }.
```

Производные типы данных в MPI…

- **Векторный способ конструирования…**
  - при векторном способе новый производный тип создается как набор блоков из элементов исходного типа, при этом между блоками могут иметься регулярные промежутки по памяти.

```
int MPI_Type_vector ( int count, int blocklen, int stride,
  MPI_Data_type oldtype, MPI_Datatype *newtype ),
где
- count    – количество блоков,
- blocklen – размер каждого блока,
- stride   – количество элементов, расположенных между двумя
             соседними блоками
- oldtype  - исходный тип данных,
- newtype  - новый определяемый тип данных.
```

  - Если интервалы между блоками задаются в байтах, а не в элементах исходного типа данных, следует использовать функцию:

```
int MPI_Type_hvector ( int count, int blocklen, MPI_Aint stride,
   MPI_Data_type oldtype, MPI_Datatype *newtype );
```

Производные типы данных в MPI…

- **Векторный способ конструирования:**

  Как следует из описания, при векторном способе новый производный тип создается как набор блоков из элементов исходного типа, при этом между блоками могут иметься регулярные промежутки по памяти. Приведем несколько примеров использования данного способа конструирования типов:

•конструирование типа для выделения половины (только четных или только нечетных) строк матрицы размером n×n:

MPI_Type_vector(n / 2, n, 2 * n, &StripRowType, &ElemType),

•конструирование типа для выделения столбца матрицы размером n×n:

MPI_Type_vector(n, 1, n, &ColumnType, &ElemType),

•конструирование типа для выделения главной диагонали матрицы размером n×n:

MPI_Type_vector(n, 1, n + 1, &DiagonalType, &ElemType).

Производные типы данных в MPI…

- **Векторный способ конструирования:**
  - создание производных типов для описания подмассивов многомерных массивов

```
int MPI_Type_create_subarray ( int ndims, int *sizes,
      int *subsizes, int *starts, int order,
      MPI_Data_type oldtype, MPI_Datatype *newtype ), где

- ndims    – размерность массива,
- sizes    – количество элементов в каждой размерности исходного
             массива,
- subsizes – количество элементов в каждой размерности
определяемого
             подмассива,
- starts   – индексы начальных элементов в каждой размерности
             определяемого подмассива,
- order    - параметр для указания необходимости переупорядочения,
- oldtype  - тип данных элементов исходного массива,
- newtype  - новый тип данных для описания подмассива.
```

Производные типы данных в MPI…

- **Индексный способ конструирования:**
  - новый производный тип создается как набор блоков разного размера из элементов исходного типа, при этом между блоками могут иметься разные промежутки по памяти.

```
int MPI_Type_indexed ( int count, int blocklens[], int indices[],
  MPI_Data_type oldtype, MPI_Datatype *newtype ),
где
- count      – количество блоков,
- blocklens – количество элементов в каждов блоке,
- indices    – смещение каждого блока от начала типа (в количестве
               элементов исходного типа),
- oldtype    - исходный тип данных,
- newtype    - новый определяемый тип данных.
```

  - Если интервалы между блоками задаются в байтах, а не в элементах исходного типа данных, следует использовать функцию:

```
int MPI_Type_hindexed ( int count, int blocklens[],
 MPI_Aint indices[],MPI_Data_type oldtype,MPI_Datatype *newtype);
```

Производные типы данных в MPI…

- **Индексный способ конструирования:**
  - конструирования типа для описания верхней треугольной матрицы размером *n*x*n*:

```
// конструирование типа для описания верхней треугольной матрицы
for ( i=0, i<n; i++ ) {
  blocklens[i] = n - i;
  indices[i]   = i * n + i;
}
MPI_Type_indexed ( n, blocklens, indices, &UTMatrixType,
                   &ElemType );
```

Производные типы данных в MPI…

- **Структурный способ конструирования:**
  - является самым общим методом конструирования производного типа данных при явном задании соответствующей карты типа:

```
int MPI_Type_struct ( int count, int blocklens[],
  MPI_Aint indices[], MPI_Data_type oldtypes[],
  MPI_Datatype *newtype ),
где
 - count       – количество блоков,
 - blocklens   – количество элементов в каждов блоке,
 - indices     – смещение каждого блока от начала типа (в байтах),
 - oldtypes    - исходные типы данных в каждом блоке в
отдельности,
 - newtype     - новый определяемый тип данных.
```

Производные типы данных в MPI…

- **Объявление производных типов и их удаление:**
  - перед использованием созданный тип *должен быть объявлен* :

```
int MPI_Type_commit (MPI_Datatype *type );
```

  - При завершении использования производный тип должен быть аннулирован при помощи функции:

```
int MPI_Type_free (MPI_Datatype *type );
```

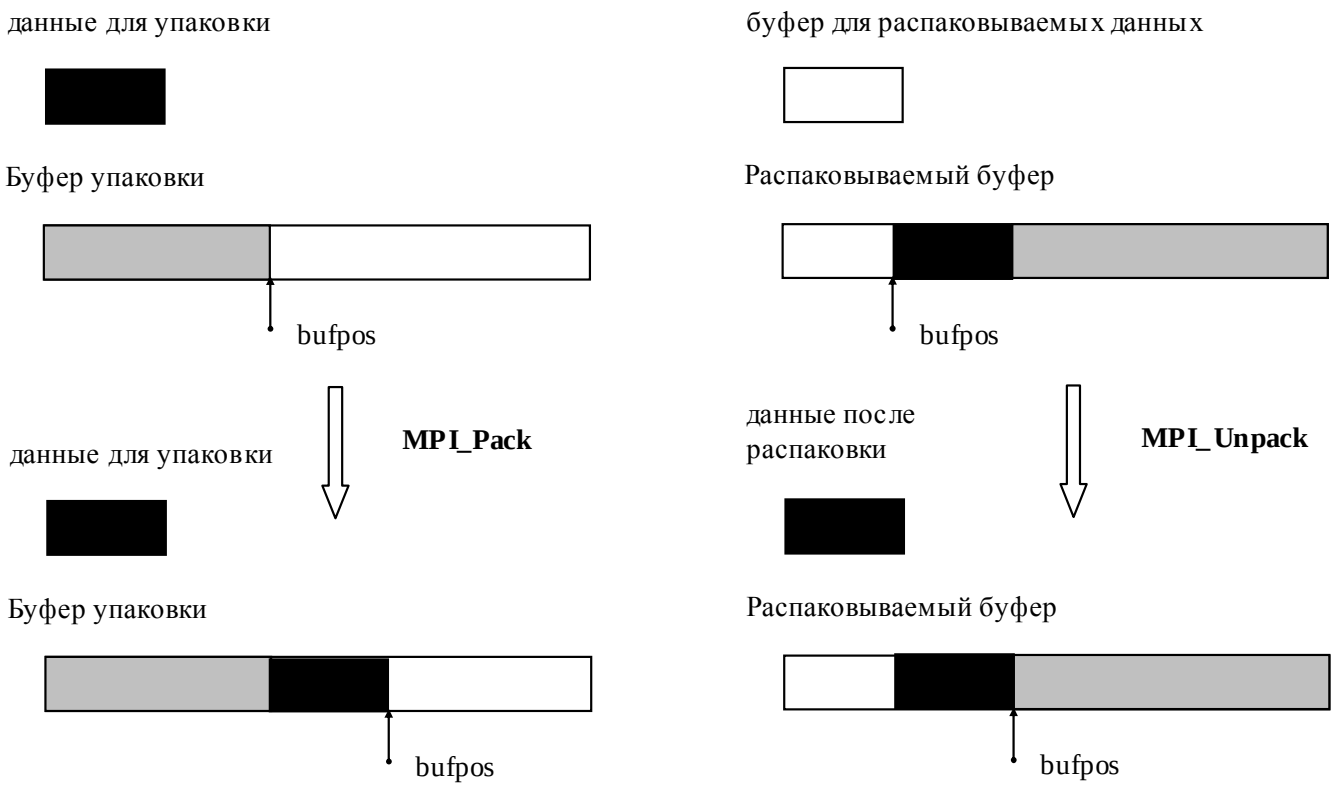
Производные типы данных в MPI…

- **Формирование сообщений при помощи упаковки и распаковки данных…**
  - явный способ сборки и разборки сообщений, в которые могут входить значения разных типов и располагаемых в разных областях памяти:

```
int MPI_Pack ( void *data, int count, MPI_Datatype type,
  void *buf, int bufsize, int *bufpos, MPI_Comm comm), где
 - data   – буфер памяти с элементами для упаковки,
 - count  – количество элементов в буфере,
 - type   – тип данных для упаковываемых элементов,
 - buf    - буфер памяти для упаковки,
 - buflen – размер буфера в байтах,
 - bufpos – позиция для начала записи в буфер (в байтах от начала
            буфера),
 - comm   - коммуникатор для упакованного сообщения.
```

Производные типы данных в MPI…

- **Формирование сообщений при помощи упаковки и распаковки данных…**



а) упаковка данных

б) распаковка данных

Производные типы данных в MPI…

- **Формирование сообщений при помощи упаковки и распаковки данных…**
  - Для определения необходимого размера буфера для упаковки может быть использована функция:

```
MPI_Pack_size (int count, MPI_Datatype type, MPI_Comm comm,
               int *size);
```

  - После упаковки всех необходимых данных подготовленный буфер может быть использован в функциях передачи данных с указанием типа *MPI_PACKED*,
  - После получения сообщения с типом *MPI_PACKED* данные могут быть распакованы при помощи функции:

```
int MPI_Unpack (void *buf, int bufsize, int *bufpos,
  void *data, int count, MPI_Datatype type, MPI_Comm comm);
```

Производные типы данных в MPI…

- **Формирование сообщений при помощи упаковки и распаковки данных…**
  - Вызов функции *MPI_Pack* осуществляется последовательно для упаковки всех необходимых данных. Если в сообщение должны входить данные

```
double a /* адрес 24 */; double b /* адрес 40 */; int    n /* адрес 48 */;
```

  то для их упаковки необходимо выполнить:

```
bufpos = 0;
MPI_Pack(a,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(b,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(n,1,MPI_INT,buf,buflen,&bufpos,comm);
```

  - Для распаковки упакованных данных необходимо выполнить:

```
bufpos = 0;
MPI_Unpack(buf,buflen,&bufpos,a,1,MPI_DOUBLE,comm);
MPI_Unpack(buf,buflen,&bufpos,b,1,MPI_DOUBLE,comm);
MPI_Unpack(buf,buflen,&bufpos,n,1,MPI_INT,comm);
```

Производные типы данных в MPI

- **Формирование сообщений при помощи упаковки и распаковки данных:**
  - Данный подход приводит к необходимости дополнительных действий по упаковке и распаковке данных,
  - Он может быть оправдан при сравнительно небольших размерах сообщений и при малом количестве повторений,
  - Упаковка и распаковка может оказаться полезной при явном использовании буферов для буферизованного способа передачи данных.

Заключение…

- Во второй презентации раздела рассмотрены имеющиеся в MPI операции передачи данных между двумя процессами, а также возможные режимы выполнения этих операций.
- Обсуждается вопрос организации неблокирующих обменов данными между процессами.
- Подробно рассмотрены коллективные операции передачи данных.
- Представлены все основные способы конструирования и использования производных типов данных в MPI.

Темы заданий для самостоятельной работы...

- **Операции передачи данных между двумя процессами**
  1. Подготовьте варианты ранее разработанных программ с разными режимами выполнения операций передачи данных. Сравните время выполнения операций передачи данных при разных режимах работы.
  2. Подготовьте варианты ранее разработанных программ с использованием неблокирующего способа выполнения операций передачи данных. Оцените необходимое количество вычислительных операций, для того чтобы полностью совместить передачу данных и вычисления. Разработайте программу, в которой бы полностью отсутствовали задержки вычислений из-за ожидания передаваемых данных.
  3. Выполните задание 3 с использованием операции одновременного выполнения передачи и приема данных. Сравните результаты вычислительных экспериментов.

Темы заданий для самостоятельной работы...

- **Коллективные операции передачи данных**

4. Разработайте программу-пример для каждой имеющейся в MPI коллективной операции.
5. Разработайте реализации коллективных операций при помощи парных обменов между процессами. Выполните вычислительные эксперименты и сравните время выполнения разработанных программ и функций MPI для коллективных операций.
6. Разработайте программу, выполните эксперименты и сравните результаты для разных алгоритмов реализации операции сбора, обработки и рассылки данных всех процессам (функция MPI_Allreduce).

Темы заданий для самостоятельной работы

- **Производные типы в MPI**

7. Разработайте программу-пример для каждого имеющегося в MPI способа конструирования производных типов данных.
8. Разработайте программу-пример с использованием функций упаковки и распаковки данных. Выполните эксперименты и сравните с результатами при использовании производных типов данных.
9. Разработайте производные типы данных для строк, столбцов, диагоналей матриц.
10. Разработайте программу-пример для каждой из рассмотренных функций для управления процессами и коммуникаторами.
11. Разработайте программу для представления множества процессов в виде прямоугольной решетки. Создайте коммуникаторы для каждой строки и столбца процессов. Выполните коллективную операцию для всех процессов и для одного из созданных коммуникаторов. Сравните время выполнения операции.
12. Изучите самостоятельно и разработайте программы-примеры для передачи данных между процессами разных коммуникаторов.