

Параллельное программирование на основе MPI

3 часть

Содержание

- Управление группами процессов и коммутаторами
 - Виртуальные топологии
 - Декартовы топологии (решетки)
 - Топологии типа граф
 - Дополнительные сведения о MPI
 - Разработка параллельных программ с использованием MPI на языке Fortran
 - Общая характеристика среды выполнения MPI программ
 - Дополнительные возможности стандарта MPI-2
 - Заключение
-

Управление группами процессов и коммутаторами...

- **Группы процессов...**

- Процессы параллельной программы объединяются в *группы*. В группу могут входить все процессы параллельной программы или в группе может находиться только часть имеющихся процессов. Один и тот же процесс может принадлежать нескольким группам,
- Управление группами процессов предпринимается для создания на их основе коммутаторов,
- Группы процессов могут быть созданы только из уже существующих групп. В качестве исходной группы может быть использована группа, связанная с предопределенным коммутатором MPI_COMM_WORLD:

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

- **comm** — коммутатор;
- **group** — группа, связанная с коммутатором

Управление группами процессов и коммутаторами...

- **Группы процессов...**

- На основе существующих групп, могут быть созданы новые группы
 - создание новой группы ***newgroup*** из существующей группы ***oldgroup***, которая будет включать в себя ***n*** процессов, ранги которых перечисляются в массиве ***ranks***:

```
int MPI_Group_incl(MPI_Group oldgroup, int n, int *ranks,  
                  MPI_Group *newgroup);
```

- создание новой группы ***newgroup*** из группы ***oldgroup***, которая будет включать в себя ***n*** процессов, ранги которых не совпадают с рангами, перечисленными в массиве ***ranks***:

```
int MPI_Group_excl(MPI_Group oldgroup, int n, int *ranks,  
                  MPI_Group *newgroup);
```

Управление группами процессов и коммутаторами...

- **Группы процессов...**

- На основе существующих групп, могут быть созданы новые группы

- создание новой группы **newgroup** как объединения групп **group1** и **group2**:

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

- создание новой группы **newgroup** как пересечения групп **group1** и **group2**:

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

- создание новой группы **newgroup** как разности групп **group1** и **group2**:

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

Управление группами процессов и коммутаторами...

- Группы процессов

- Получение информации о группе процессов:

- получение количества процессов в группе:

```
int MPI_Group_size ( MPI_Group group, int *size );
```

- получение ранга текущего процесса в группе:

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

- После завершения использования группа должна быть удалена:

```
int MPI_Group_free ( MPI_Group *group );
```

Управление группами процессов и коммуникаторами...

- Коммуникаторы...

- Под *коммуникатором* в MPI понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст*), используемых при выполнении операций передачи данных,
 - Будем рассматривать управление *интракоммуникаторами*, используемыми для операций передачи данных внутри одной группы процессов.
-

Управление группами процессов и коммутаторами...

- **Коммутаторы...**

- Создание коммутатора:

- дублирование уже существующего коммутатора:

```
int MPI_Comm_dup ( MPI_Comm oldcom, MPI_Comm *newcomm );
```

- создание нового коммутатора из подмножества процессов существующего коммутатора:

```
int MPI_comm_create (MPI_Comm oldcom, MPI_Group group,  
MPI_Comm *newcomm);
```

Операция создания коммутаторов является коллективной и, тем самым, должна выполняться всеми процессами исходного коммутатора, После завершения использования коммутатор должен быть удален:

```
int MPI_Comm_free ( MPI_Comm *comm );
```


Управление группами процессов и коммутаторами

- Для пояснения рассмотренных функций можно привести пример создания коммутатора, в котором содержатся все процессы, кроме процесса, имеющего ранг 0 в коммутаторе `MPI_COMM_WORLD` (такой коммутатор может быть полезен для поддержки схемы организации параллельных вычислений "менеджер – исполнители")
-

Управление группами процессов и коммутаторами

```
MPI_Group WorldGroup, WorkerGroup;  
MPI_Comm Workers;  
int ranks[1];  
ranks[0] = 0;  
// Получение группы процессов в MPI_COMM_WORLD  
MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);  
// Создание группы без процесса с рангом 0  
MPI_Group_excl(WorldGroup, 1, ranks, &WorkerGroup);  
// Создание коммутатора по группе  
MPI_Comm_create(MPI_COMM_WORLD, WorkerGroup, &Workers);  
...  
MPI_Group_free(&WorkerGroup);  
MPI_Comm_free(&Workers);
```

Управление группами процессов и коммуникаторами

- **Коммуникаторы**

- Одновременное создание нескольких коммуникаторов :

```
int MPI_Comm_split ( MPI_Comm oldcomm, int split, int key,  
    MPI_Comm *newcomm ),
```

где

- **oldcomm** – исходный коммуникатор,
- **split** – номер коммуникатора, которому должен принадлежать процесс,
- **key** – порядок ранга процесса в создаваемом коммуникаторе,
- **newcomm** – создаваемый коммуникатор

Вызов функции ***MPI_Comm_split*** должен быть выполнен в каждом процессе коммуникатора ***oldcomm***,

Процессы разделяются на непересекающиеся группы с одинаковыми значениями параметра ***split***. На основе сформированных групп создается набор коммуникаторов. Порядок нумерации процессов соответствует порядку значений параметров ***key*** (процесс с большим значением параметра ***key*** будет иметь больший ранг).

Управление группами процессов и коммутаторами

В качестве примера можно рассмотреть задачу представления набора процессов в виде двумерной решетки. Пусть $p=q*q$ есть общее количество процессов; следующий далее фрагмент программы обеспечивает получение коммутаторов для каждой строки создаваемой топологии:

```
MPI_Comm comm;  
int rank, row;  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
row = rank / q;  
MPI_Comm_split(MPI_COMM_WORLD, row, rank, &comm);
```

При выполнении данного примера, например, при $p=9$, процессы с рангами (0, 1, 2) образуют первый коммутатор, процессы с рангами (3, 4, 5) – второй и т. д.

Виртуальные топологии...

- Под *топологией* вычислительной системы понимают структуру узлов сети и линий связи между этими узла. Топология может быть представлена в виде графа, в котором вершины есть процессоры (процессы) системы, а дуги соответствуют имеющимся линиям (каналам) связи.
 - Парные операции передачи данных могут быть выполнены между любыми процессами коммутатора, в коллективной операции принимают участие все процессы коммутатора. Следовательно, логическая топология линий связи между процессами в параллельной программе имеет структуру *полного графа*.
 - Возможно организовать логическое представление любой необходимой *виртуальной топологии*. Для этого достаточно сформировать тот или иной механизм дополнительной адресации процессов.
-

Виртуальные топологии...

- **Декартовы топологии (решетки)...**

- В декартовых топологиях множество процессов представляется в виде прямоугольной *решетки*, а для указания процессов используется декартова системы координат,
- Для создания декартовой топологии (решетки) в MPI предназначена функция:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims,  
int *periods, int reorder, MPI_Comm *cartcomm),
```

где:

- **oldcomm** – исходный коммуникатор,
- **ndims** – размерность декартовой решетки,
- **dims** – массив длины `ndims`, задает количество процессов в каждом измерении решетки,
- **periods** – массив длины `ndims`, определяет, является ли решетка периодической вдоль каждого измерения,
- **reorder** – параметр допустимости изменения нумерации процессов,
- **cartcomm** – создаваемый коммуникатор с декартовой топологией процессов.

Управление группами процессов и коммутаторами

- Для пояснения назначения параметров функции `MPI_Cart_create` рассмотрим пример создания двумерной решетки 4x4, в которой строки и столбцы имеют кольцевую структуру (за последним процессом следует первый процесс):

// Создание двумерной решетки 4x4

MPI_Comm GridComm;

int dims[2], periods[2], reorder = 1;

dims[0] = dims[1] = 4;

periods[0] = periods[1] = 1;

**MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
 &GridComm);**

- Следует отметить, что в силу кольцевой структуры измерений сформированная в рамках примера топология является тором.
-

Виртуальные топологии...

- Декартовы топологии (решетки)...

- Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int MPI_Card_coords(MPI_Comm comm, int rank, int ndims, int *coords),
```

- **comm** – коммуникатор с топологией решетки,
- **rank** – ранг процесса, для которого определяются декартовы координаты,
- **ndims** – размерность решетки,
- **coords** – возвращаемые функцией декартовы координаты процесса.

- Обратное действие – определение ранга процесса по его декартовым координатам – обеспечивается при помощи функции:

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank),
```

где

- **comm** – коммуникатор с топологией решетки,
- **coords** – декартовы координаты процесса,
- **rank** – возвращаемый функцией ранг процесса.

Виртуальные топологии...

- Декартовы топологии (решетки)...

- процедура разбиения решетки на подрешетки меньшей размерности обеспечивается при помощи функции:

```
int MPI_Card_sub(MPI_Comm comm, int *subdims, MPI_Comm  
*newcomm) ,
```

где:

- **comm** – исходный коммуникатор с топологией решетки,
- **subdims** – массив для указания, какие измерения должны остаться в создаваемой подрешетке,
- **newcomm** – создаваемый коммуникатор с подрешеткой.

В ходе своего выполнения функция *MPI_Cart_sub* определяет коммуникаторы для каждого сочетания координат фиксированных измерений исходной решетки.

Виртуальные топологии...

- **Декартовы топологии (решетки)...**
 - **Пример:** создание двумерной решетки 4×4 , в которой строки и столбцы имеют кольцевую структуру (за последним процессом следует первый процесс).
Определяются коммутаторы с декартовой топологией для каждой строки и столбца решетки в отдельности. Создаются 8 коммутаторов, по одному для каждой строки и столбца решетки. Для каждого процесса определяемые коммутаторы *RowComm* и *ColComm* соответствуют строке и столбцу процессов, к которым данный процесс принадлежит

[Программа](#)

Управление группами процессов и коммутаторами

**// Создание коммутаторов для каждой строки и столбца
решетки**

MPI_Comm RowComm, ColComm; int subdims[2];

// Создание коммутаторов для строк

subdims[0] = 0; // фиксации измерения

subdims[1] = 1; // наличие данного измерения в подрешетке

MPI_Cart_sub(GridComm, subdims, &RowComm);

// Создание коммутаторов для столбцов

subdims[0] = 1;

subdims[1] = 0;

MPI_Cart_sub(GridComm, subdims, &ColComm);

Виртуальные топологии...

- Декартовы топологии (решетки)...

- Дополнительная функция *MPI_Cart_shift* обеспечивает поддержку операции сдвига данных:
 - Циклический сдвиг на k элементов вдоль измерения решетки – в этой операции данные от процесса i пересылаются процессу $(i+k) \bmod \text{dim}$, где **dim** есть размер измерения, вдоль которого производится сдвиг,
 - Линейный сдвиг на k позиций вдоль измерения решетки – в этом варианте операции данные от процессора i пересылаются процессору $i+k$ (если таковой существует),
 - Функция *MPI_Cart_shift* только определяет ранги процессов, между которыми должен быть выполнен обмен данными в ходе операции сдвига. Непосредственная передача данных, может быть выполнена, например, при помощи функции *MPI_Sendrecv*.
-

Виртуальные топологии...

- Декартовы топологии (решетки)
 - Функция *MPI_Cart_shift* обеспечивает получение рангов процессов, с которыми текущий процесс (процесс, вызвавший функцию *MPI_Cart_shift*) должен выполнить обмен данными:

```
int MPI_Cart_shift(MPI_Comm comm, int dir, int disp, int *source,  
                  int *dst),
```

где:

- **comm** – коммуникатор с топологией решетки,
- **dir** – номер измерения, по которому выполняется сдвиг,
- **disp** – величина сдвига (<0 – сдвиг к началу измерения),
- **source** – ранг процесса, от которого должны быть получены данные,
- **dst** – ранг процесса которому должны быть отправлены данные.

Виртуальные топологии...

- Топология графа...

- Создание коммуникатора с топологией типа граф:

```
int MPI_Graph_create(MPI_Comm oldcomm, int nnodes, int *index,  
    int *edges, int reorder, MPI_Comm *graphcomm),
```

где:

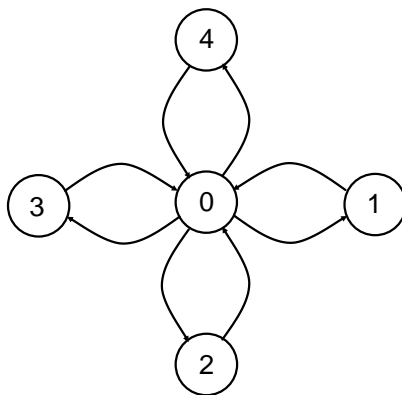
- **oldcomm** – исходный коммуникатор,
- **nnodes** – количество вершин графа,
- **index** – количество исходящих дуг для каждой вершины,
- **edges** – последовательный список дуг графа,
- **reorder** – параметр допустимости изменения нумерации процессов,
- **cartcomm** – создаваемый коммуникатор с топологией типа граф.

Операция создания топологии является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

Виртуальные топологии...

- **Топология графа (пример)...**

Граф для топологии звезда, количество процессоров равно **5**, порядки вершин принимают значения **(4,1,1,1,1)**, а матрица инцидентности имеет вид:



Процессы Линии связи	
0	1,2,3,4
1	0
2	0
3	0
4	0

Для создания топологии с графом данного вида необходимо выполнить следующий программный код:

```
int index[] = { 4,1,1,1,1 };
int edges[] = { 1,2,3,4,0,0,0,0 };
MPI_Comm StarComm;
MPI_Graph_create(MPI_COMM_WORLD, 5, index, edges, 1, &StarComm);
```


Дополнительные сведения о MPI...

- Разработка параллельных программ с использованием MPI на языке Fortran...
 - Подпрограммы библиотеки MPI являются процедурами и, тем самым, вызываются при помощи оператора вызова процедур CALL,
 - Коды завершения передаются через дополнительный параметр целого типа, располагаемый на последнем месте в списке параметров процедур,
 - Переменная ***status*** является массивом целого типа из *MPI_STATUS_SIZE* элементов,
 - Типы *MPI_Comm* и *MPI_Datatype* представлены целых типом INTEGER.
-

Дополнительные сведения о MPI...

- Разработка параллельных программ с использованием MPI на языке Fortran

```
PROGRAM MAIN
  include 'mpi.h'
  INTEGER PROCNUM, PROCRANK, RECVRANK, IERR
  INTEGER STATUS(MPI_STATUS_SIZE)
  CALL MPI_Init(IERR)
  CALL MPI_Comm_size(MPI_COMM_WORLD, PROCNUM, IERR)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, PROCRANK, IERR)
  IF ( PROCRANK.EQ.0 ) THEN
    ! Действия, выполняемые только процессом с рангом 0
    PRINT *, "Hello from process ", PROCRANK
    DO i = 1, PROCNUM-1
      CALL MPI_RECV(RECVRANK, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, STATUS, IERR)
      PRINT *, "Hello from process ", RECVRANK
    END DO
  ELSE ! Сообщение, отправляемое всеми процессами, кроме процесса с
    рангом 0
    CALL MPI_SEND(PROCRANK, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, IERR)
  END IF
  CALL MPI_FINALIZE(IERR)
  STOP
END
```

Дополнительные сведения о MPI...

- **Общая характеристика среды выполнения MPI программ...**
 - Для проведения параллельных вычислений в вычислительной системе должна быть установлена среда выполнения MPI программ:
 - Для разработки, компиляции и компоновки, как правило, достаточно обычных средств разработки программ (например, Microsoft Visual Studio) и той или иной библиотеки MPI,
 - Для выполнения параллельных программ от среды выполнения требуется наличие средств указания используемых процессоров, операций удаленного запуска программ и т.п.,
 - Желательно наличие в среде выполнения средств профилирования, трассировки и отладки параллельных программ.
-

Дополнительные сведения о MPI...

- **Общая характеристика среды выполнения MPI программ**
 - Запуск MPI программы зависит от среды выполнения, в большинстве случаев выполняется при помощи команды **mpirun**. Возможные параметры команды:
 - *Режим выполнения* – локальный или многопроцессорный. Локальный режим обычно указывается при помощи ключа – `localonly`,
 - *Количество процессов*, которые необходимо создать при запуске параллельной программы,
 - *Состав используемых процессоров*, определяемый тем или иным конфигурационным файлом,
 - *Исполняемый файл* параллельной программы,
 - *Командная строка* с параметрами для выполняемой программы.
-

Дополнительные сведения о MPI

- **Дополнительные возможности стандарта MPI-2**

- *Динамическое порождение процессов*, при котором процессы параллельной программы могут создаваться и уничтожаться в ходе выполнения,
 - *Одностороннее взаимодействие процессов*, что позволяет быть инициатором операции передачи и приема данных только одному процессу,
 - *Параллельный ввод/вывод*, обеспечивающий специальный интерфейс для работы процессов с файловой системой,
 - *Расширенные коллективные операции*, в числе которых, например, процедуры для взаимодействия процессов из нескольких коммутаторов одновременно,
 - *Интерфейс для алгоритмического языка C++*.
-

Заключение

- В третьей презентации раздела обсуждаются вопросы управления группами процессов и коммутаторами.
 - Изложены возможности MPI по использованию виртуальных топологий.
 - Приведены дополнительные сведения о стандарте MPI: принципы разработки параллельных программ на языке Fortran, краткая характеристика сред выполнения MPI программ и обзор дополнительных возможностей стандарта MPI-2.
-

Ссылки

- Информационный ресурс Интернет с описанием стандарта MPI: <http://www.mpiforum.org>
 - Одна из наиболее распространенных реализаций MPI библиотека MPICH представлена на <http://www-unix.mcs.anl.gov/mpi/mpich>
 - Библиотека MPICH2 с реализацией стандарта MPI-2 содержится на <http://www-unix.mcs.anl.gov/mpi/mpich2>
 - Русскоязычные материалы о MPI имеются на сайте <http://www.parallel.ru>
-

Литература...

- **Гергель В.П.** (2007). Теория и практика параллельных вычислений. – М.: Интернет-Университет, БИНОМ. Лаборатория знаний.
 - **Воеводин В.В., Воеводин Вл.В.** (2002). Параллельные вычисления. – СПб.: [БХВ-Петербург](#).
 - **Немнюгин С., Стесик О.** (2002). Параллельное программирование для многопроцессорных вычислительных систем – СПб.: БХВ-Петербург.
 - **Group, W., Lusk, E., Skjellum, A.** (1994). Using MPI. Portable Parallel Programming with the Message-Passing Interface. – MIT Press.
 - **Group, W., Lusk, E., Skjellum, A.** (1999a). Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.
-

Литература

- **Group**, W., Lusk, E., Thakur, R. (1999b). Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.
 - **Pacheco**, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.
 - **Quinn**, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
 - **Snir**, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. (1996). [MPI: The Complete Reference](#). - [MIT Press](#), Boston, 1996.
-