Л8. Параллельное программирование с использованием технологии MPI

1. Гергель В. П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория Базовых Знаний, 2007. – с. 110..124.

1. Общие сведения

Основным стандартом, широко используемым в настоящее время в практических приложениях, является интерфейс передачи сообщений (message passing interface - MPI). Наличие такого стандарта позволило разработать стандартные библиотеки программ (MPI-библиотеки), в которых оказалось возможным скрыть большинство архитектурных особенностей ПВС и, как результат, существенно упростить проблему создания параллельных программ. Более того, стандартизация базового системного уровня позволила в значительной степени обеспечить мобильность параллельных программ, поскольку в настоящее время реализации MPI-стандарта имеются для большинства компьютерных платформ.

2. MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью передачи сообщений

Эта достаточно объемная и сложная библиотека состоит примерно из 130 функций, в число которых входят:

функции инициализации и закрытия MPI-процессов;

функции, реализующие коммуникационные операции типа точка-точка;

функции, реализующие коллективные операции;

функции для работы с группами процессов и коммуникаторами;

функции для работы со структурами данных;

функции формирования топологии процессов.

Но почти любая параллельная программа может быть написана с использованием всего 6 MPI-функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций.

3. Структура программы с использованием MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
int rank, numprocs, i, message;
<Код без использования MPI>
// инициализация библиотеки MPI
MPI_Init(&argc, &argv);
<Код с использованием MPI>
// завершение библиотеки MPI
MPI_Finalize();
<Код без использования MPI>
return 0;
}
```

4. Первая программа

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
int rank, numprocs, i, message;
// инициализация библиотеки MPI
MPI_Init(&argc, &argv);
// получение количества процессов в программе
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
// получение ранга процесса
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("\n Hello from process %3d", rank);
// завершение библиотеки MPI
MPI_Finalize();
return 0;
}
```

5. Передача сообщений между двумя процессами

MPI - функция передачи сообщений (MPI_Send):
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm),
где

— **buf** — адрес буфера памяти, в котором располагаются данные отправляемого сообщения;
— **count** — количество элементов данных в сообщении;
— **type** — тип элементов данных пересылаемого сообщения;
— **dest** — ранг процесса, которому отправляется сообщение;
— **tag** — значение-тег, используемое для идентификации сообщения;
— **comm** — коммуникатор, в рамках которого выполняется передача данных.

6. Прием сообщений между двумя процессами

Функция приема сообщений (MPI_Recv):

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

где

- **buf, count, type** — буфер памяти для приема сообщения, назначение каждого отдельного параметра соответствует описанию в `MPI_Send`;
- **source** — ранг процесса, от которого должен быть выполнен прием сообщения;
- **tag** — тег сообщения, которое должно быть принято для процесса;
- **comm** — коммуникатор, в рамках которого выполняется передача данных;
- **status** – указатель на структуру данных с информацией о результате выполнения операции приема данных.

7. Базовые (предопределенные) типы данных

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

8. Дополнительные характеристики передачи сообщений между двумя процессами

- Для приема сообщения от любого *процесса*- отправителя в source указывается значение ***MPI_ANY_SOURCE*** (функция ***MPI_Send*** отсылает сообщение определенному адресату);
- Для приема сообщения с любым тегом в tag указывается значение ***MPI_ANY_TAG*** (функция ***MPI_Send*** должна содержать конкретное значение тега);
- параметр "коммуникатор" не имеет значения, означающего "любой коммуникатор";
- параметр status позволяет определить ряд характеристик принятого сообщения: ***-status.MPI_SOURCE*** — ранг *процесса* – отправителя принятого сообщения; ***-status.MPI_TAG*** — тег принятого сообщения.

Параметр status позволяет определить количество элементов данных в принятом сообщении при помощи функции:
int MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count),
где **status** — статус операции ***MPI_Recv***, **type** — тип принятых данных, **count** — количество элементов данных в сообщении.

9. Пример передачи сообщений между двумя процессами

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
int rank, n, i, message; MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &n);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0){
 printf("\n Hello from process %3d", rank);
for (i=1; i<n; i++){
MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
printf("\n Hello from process%3d", message);
}
  }else MPI_Send(&rank,1,MPI_INT,0,0,MPI_COMM_WORLD);
 MPI_Finalize();return 0;
}
```

10. Возможный порядок вывода сообщений

```
Hello from process 0
Hello from process 2
Hello from process 1
Hello from process 3
```

Несложно заметить, что сообщения от процессов выводятся "не по порядку". Причем, порядок вывода сообщений может изменяться от запуска к запуску. Для упорядочения надо контролировать ранг отправителя.

11. Пример упорядоченного вывода

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
int rank, n, i, message; MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &n);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
If (rank == 0){ Printf ("\n Hello from process %3d", rank);
for (i=1; i<n; i++){
MPI_Recv(&message, 1, MPI_INT, i, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
Printf("\n Hello from process %3d", message); }
}else
MPI_Send(&rank,1,MPI_INT,0,0,MPI_COMM_WORLD);
MPI_Finalize(); Return 0;
}
```

12. Разделение ветвей кода между процессорами

В общем случае программа имеет вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoProcess0();
else if ( ProcRank == 1 ) DoProcess1();
else if ( ProcRank == 2 ) DoProcess2();
```

При использовании схемы «менеджер – исполнители» структура программы упрощается:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoManagerProcess();
else DoWorkerProcesses();
```

13. Функции отсчета времени

Функция возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом (точки отсчета):
double MPI_Wtime(void).
Для хронометража участка программы вызов функции делается в начале участка, в конце участка и определяется разница между показаниями таймера:

```
{double starttime, endtime; starttime = MPI_Wtime();
… хронометрируемый участок …
endtime = MPI_Wtime();
printf("Выполнение заняло %f секунд\n",endtime-starttime);}
```

Функция MPI_Wtick, имеющая точно такой же синтаксис, возвращает разрешение таймера (минимальное значение кванта времени).

14. Синхронизация процессов

Одновременное достижение *процессами* тех или иных точек *процесса* вычислений, обеспечивается при помощи функции ***MPI: int MPI_Barrier(MPI_Comm comm)***,
где **comm** — коммуникатор, в рамках которого выполняется операция. Функция ***MPI_Barrier*** определяет коллективную операцию, и, тем самым, при использовании она должна вызываться всеми *процессами* используемого коммуникатора. При вызове функции ***MPI_Barrier*** выполнение *процесса* блокируется, продолжение вычислений *процесса* произойдет только после вызова функции ***MPI_Barrier*** всеми *процессами* коммуникатора.

15. Контроль выполнения MPI-программы

Все функции *MPI* (кроме ***MPI_Wtime*** и ***MPI_Wtick***) возвращают *код завершения*. При успешном выполнении код равен ***MPI_SUCCESS***. При других значениях кода завершения имеет место ошибка:
MPI_ERR_BUFFER — неправильный указатель на буфер; ***MPI_ERR_TRUNCATE*** — сообщение превышает размер приемного буфера; ***MPI_ERR_COMM*** — неправильный коммуникатор; ***MPI_ERR_RANK*** — неправильный ранг *процесса* и др. Полный список констант содержится в файле ***mpi.h***. Любая ошибка во время выполнения функции ***MPI*** приводит к немедленному завершению *параллельной программы*. Для корректного завершения: ***int MPI_Abort(MPI_Comm, int errorcode)***. Стандартный вызов:
MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER)