Технологии параллельного программирования на C++

Семинар 7

MPI

Основы MPI

- MPI –*Message Passing Interface*, интерфейс передачи сообщений. http://www.mpi-forum.org/
- Основная цель MPI – предоставить широко используемый стандарт для написания параллельных приложений построенных на передаче сообщений
- Различные программные реализации стандарта:

  MPICH, Open MPI, Intel MPI, HP-MPI, Mvapichи др.
- На учебных машинах установлен Open MPI.
- Подробная информация: **ompi_info**
- Активация Open MPI: **module load mpi/openmpi-x86_64**

Компиляция и запуск

- **Компиляция программы:**

  mpic++ –o test test.cpp

- **Запуск программы:**

  mpirun –np 12 test (12 число процессов)

Основные понятия MPI

- Все процессы программы последовательно перенумерованы от 0 до *p-1* (*p* - общее количество процессов). Номер процесса именуется *рангом* процесса.
- **Коммуникатор** - специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров, используемых при выполнении операций передачи данных.
- Процессы могут взаимодействовать только внутри некоторого коммуникатора!
- Два основных атрибута процесса: коммуникатор и номер процесса в коммуникаторе
- Операции: точка–точка, коллективные
- При старте программы все процессы работают в рамках коммуниктор **MPI_COMM_WORLD**

Основы MPI

- Префикс **MPI_**.
- #include "mpi.h"
- Сообщение—набор данных некоторого типа.
- Атрибуты сообщения: номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения, коммуникатор и др.
- Идентификатор сообщения-целое неотрицательное число в диапазоне от 0 до 32767
- Большинство функций MPI возвращают информацию об успешности завершения.
- В случае успешного выполнения функция вернет значение **MPI_SUCCESS**, иначе—код ошибки.
- Предопределенные значения, соответствующие различным ошибочным ситуациям, определены в файле **mpi.h**

Типы данных в MPI

- Базовые типы данных
- Есть возможность создавать производные типов данных

MPI_BYTE
MPI_CHAR
MPI_DOUBLE
MPI_FLOAT
MPI_INT
MPI_LONG
MPI_LONG_DOUBLE
MPI_PACKED
MPI_SHORT
MPI_UNSIGNED_CHAR
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_SHORT

Инициализация и завершение MPI программ

- **int MPI_Init(int* argc, char*** argv)**

  Инициализация параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза. Если MPI уже был инициализирован, то никакие действия не выполняются и происходит немедленный возврат из подпрограммы. Все оставшиеся MPI-процедуры могут быть вызваны только после вызова **MPI_Init.**

Возвращает: в случае успешного выполнения - *MPI_SUCCESS*, иначе -код ошибки.

- **int MPI_Finalize(void)**

  Завершение параллельной части приложения. Все последующие обращения к любым MPI-процедурам, в том числе к **MPI_Init**, запрещены. К моменту вызова **MPI_Finalize** некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Сложный тип аргументов **MPI_Init** предусмотрен для того, чтобы передавать всем процессам аргументы main:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
     программный код без использования MPI функций
    MPI_Init ( &agrc, &argv );
     программный код с использованием MPI функций
    MPI_Finalize();
     программный код без использования MPI функций
return 0;
}
```

Количество и ранг процессов

- **int MPI_Comm_size(MPI_Comm comm, int* size)**

  Определение общего числа параллельных процессов в коммуникаторе *comm*.

  comm-идентификатор коммуникатора

  size – получаемый размер коммуникатора

- **Int MPI_Comm_rank(MPI_Comm comm, int* rank)**

  Определение ранга процесса в коммуникаторе *comm*. Значение, возвращаемое по адресу *&rank*, лежит в диапазоне от 0 до *size_of_group-1*.

  comm-идентификатор коммуникатора

  rank- получаемый номер вызывающего процесса в коммуникаторе *comm*

Пример

```
#include "mpi.h"
#include <iostream>
using namespace std;

int main () {
   int ProcNum, ProcRank;
   MPI_Init (nullptr,nullptr);
   MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
   MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank);
   MPI_Finalize();
   cout<<ProcNum<<endl;
   return 0;
}
```

Обмен сообщениями

Два типа:

- Точка-точка
- Коллективные

Два класса:

- Блокирующие
- Неблокирующие

Блокирующая передача сообщения MPI

- **Int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm)**
- **buf** - адрес начала буфера памяти, в котором располагаются данные отправляемого сообщения,
- **count** - число передаваемых элементов в сообщении
- **datatype** - тип передаваемых элементов
- **dest** - номер процесса-получателя
- **msgtag** - идентификатор сообщения
- **comm** -идентификатор коммуникатора
- Блокирующая посылка сообщения с идентификатором **msgtag**, состоящего из **count** элементов типа **datatype**, процессу с номером **dest**. Все элементы сообщения расположены подряд в буфере **buf**. Значение **count** может быть нулем. Тип передаваемых элементов **datatype** должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.
- Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу **dest**, остается за MPI.
- Возврат из **MPI_Send** не означает ни того, что сообщение уже передано процессу **dest**, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший **MPI_Send**.
- Отправляемое сообщение определяется через указание блока памяти (*буфера*), в котором это сообщение располагается. Используемая для указания буфера триада (buf, count, datatype)

Пример

- Передача числа

```
int a[5];
...
MPI_Send(a, 5, MPI_INT, rank, tag, MPI_COMM_WORLD);
```

- Передача статического массива

```
int a;
...
MPI_Send(&a, 1, MPI_INT, rank, tag, MPI_COMM_WORLD);
```

Блокирующая передача сообщения MPI

Дополнительные режимы передачи сообщений

- **int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**

  передача сообщения с буферизацифункция отправки сообщения в буферизованном режиме

- **int MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**

  функция отправки сообщения в синхронном режиме,

- **int MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**

  функция отправки сообщения в режиме по готовности.

Для буферизованного режима

- **int MPI_Buffer_attach(void* buffer, int size)**

  Назначение массива **buffer** размера **size**для использования при посылке сообщений с буферизацией. В каждом процессе может быть только один такой буфер.

  MPI_BSEND_OVERHEAD
- **Int MPI_Buffer_detach(void* buffer_addr, int* size)**

  Освобождение массива **buffer** для других целей. Процесс блокируется до того момента, когда все сообщения уйдут из данного буфера.

Блокирующий приём сообщения MPI

- **int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Status*status)**
- **buf** - получаемый адрес начала буфера приема сообщения
- **count** - максимальное число элементов в принимаемом сообщении
- **datatype** - тип элементов принимаемого сообщения
- **source** - номер процесса-отправителя
- **msgtag**- идентификатор принимаемого сообщения
- **comm** - идентификатор группы
- **status**- получаемые параметры принятого сообщения
- Прием сообщения с идентификатором **msgtag** от процесса **source** с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения **count**. Если число принятых элементов меньше значения **count**, то гарантируется, что в буфере **buf** изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой **MPI_Probe**.
- Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере **buf**.

Приём сообщения

- При необходимости приема сообщения от любого процесса-отправителя для параметра **source** может быть указано значение **MPI_ANY_SOURCE**
- При необходимости приема сообщения с любым тегом для параметра **tag** может быть указано значение **MPI_ANY_TAG**
- Параметр **status** позволяет определить ряд характеристик принятого сообщения:

  status.MPI_SOURCE – ранг процесса-отправителя принятого сообщения

  status.MPI_TAG – тег принятого сообщения

  status.MPI_ERROR – код ошибки принятого сообщения

- В случае, когда структура **status** не нужна можно вместо нее использовать**:**

  MPI_STATUS_IGNORE и MPI_STATUSES_IGNORE (для массива)

Приём сообщения

- **MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count)**

Возвращает в переменной **count** количество элементов типа **type** в принятом сообщении

По значению параметра **status** данная подпрограмма определяет число уже принятых (после обращения к **MPI_Recv**) или принимаемых (после обращения к **MPI_Probe** или **MPI_Iprobe**) элементов сообщения типа **datatype**.

Прием сообщения

- **int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status* status)**

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

  msgtag - идентификатор ожидаемого сообщения или **MPI_ANY_TAG**

  comm - идентификатор группы

  status - получаемые параметры обнаруженного сообщения

  Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра **status**. Следует обратить внимание, что подпрограмма определяет только факт прихода сообщения, но реально его не принимает.

Пример

```
#include "mpi.h"
#include <iostream>
using namespace std;
int main(){
  int Num, ProcRank, RecvRank;
  MPI_Status Status;
  MPI_Init(nullptr, nullptr);
  MPI_Comm_size(MPI_COMM_WORLD, &Num);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  if ( ProcRank == 0 ){
    cout<<ProcRank<<endl;
    for ( int i=1; i<Num; i++ ) {
      MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
      MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
      cout<<RecvRank<<endl;
    }
  }
  else
    MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

Задача 1

- Найти норму вектора.

Реализовать различные вариации:

- Send, Recv
- Bsend, Recv
- Ssend, Probe, Recv

Задача 2

- Сортировка массива