Спецкурс кафедры «Вычислительной математки»

Параллельные алгоритмы вычислительной алгебры

Александр Калинкин

Сергей Гололобов

Часть 3: Распараллеливание на компьютерах с распределенной памятью

Средства программирования для компьютеров с распределённой памятью (MPI)

Понятие процесса в вычислениях на компьютерах с распределённой памятью

Основные инструменты MPI

Коммуникации one-to-one, блокирующие и неблокирующие пересылки

Примеры элементарных ошибок

Коллективные коммуникации

Работа с группами и коммуникаторами

Средства программирования для компьютеров с распределённой памятью (MPI)

Message Passing Interface (MPI) – набор программ, разработанный для передачи сообщений между компьютерами

Первый стандарт разработан в 1993-94 годах коллективом разработчиков MPI Forum, в составе которых выходили: Уильямом Гроуппом, Эвином Ласком и др.

MPI v1 (MPI-1) стандарт включал в себя 128 функций поддерживающих C и Fortran-77 интерфейсы

MPI v2 (MPI-2) стандарт включал в себя уже более 500 функций, поддерживающий C, C++ и Fortran-90.

MPI 3.1 является расширением MPI-1 и 2, все функции которые были в MPI-1 и 2 поддерживаются и в MPI-3 стандарте. Выпущен 09’12. Дополнен 06’15.

Средства программирования для компьютеров с распределённой памятью (MPI)

MPICH – бесплатная реализация для UNIX и Windows.
Последняя версия - *MPICH 3.2*

MVAPICH — бесплатная реализация MPI для Windows / Linux.
Последняя версия - *MVAPICH2 2.2*

Open MPI — бесплатная реализация MPI для Windows / Linux.
Последняя версия – *Open MPI 2.0.1*

Intel MPI — коммерческая реализация для Windows / Linux.
Последняя версия – *Intel MPI 2017*

Средства программирования для компьютеров с распределённой памятью (MPI)

OpenMP – директивы компилятора, MPI – вызовы функций и процедур

Общее описание вызова MPI подпрограмм из C и Fortrana

C:

```
#include "mpi.h"
error = MPI_Xxxxx(parameter, ... );
```

Регистр важен! MPI_X верхний, остальное нижний

Fortran:

```
include 'mpif.h‘
call MPI_Xxxxx(parameter, ... , error)
```

Регистр неважен, есть дополнительный параметр.

Понятие процесса в вычислениях на компьютерах с распределённой памятью

Как сделать параллельную программу из последовательной?

error=MPI_Init();

Последовательная программа; только здесь могут появляться MPI

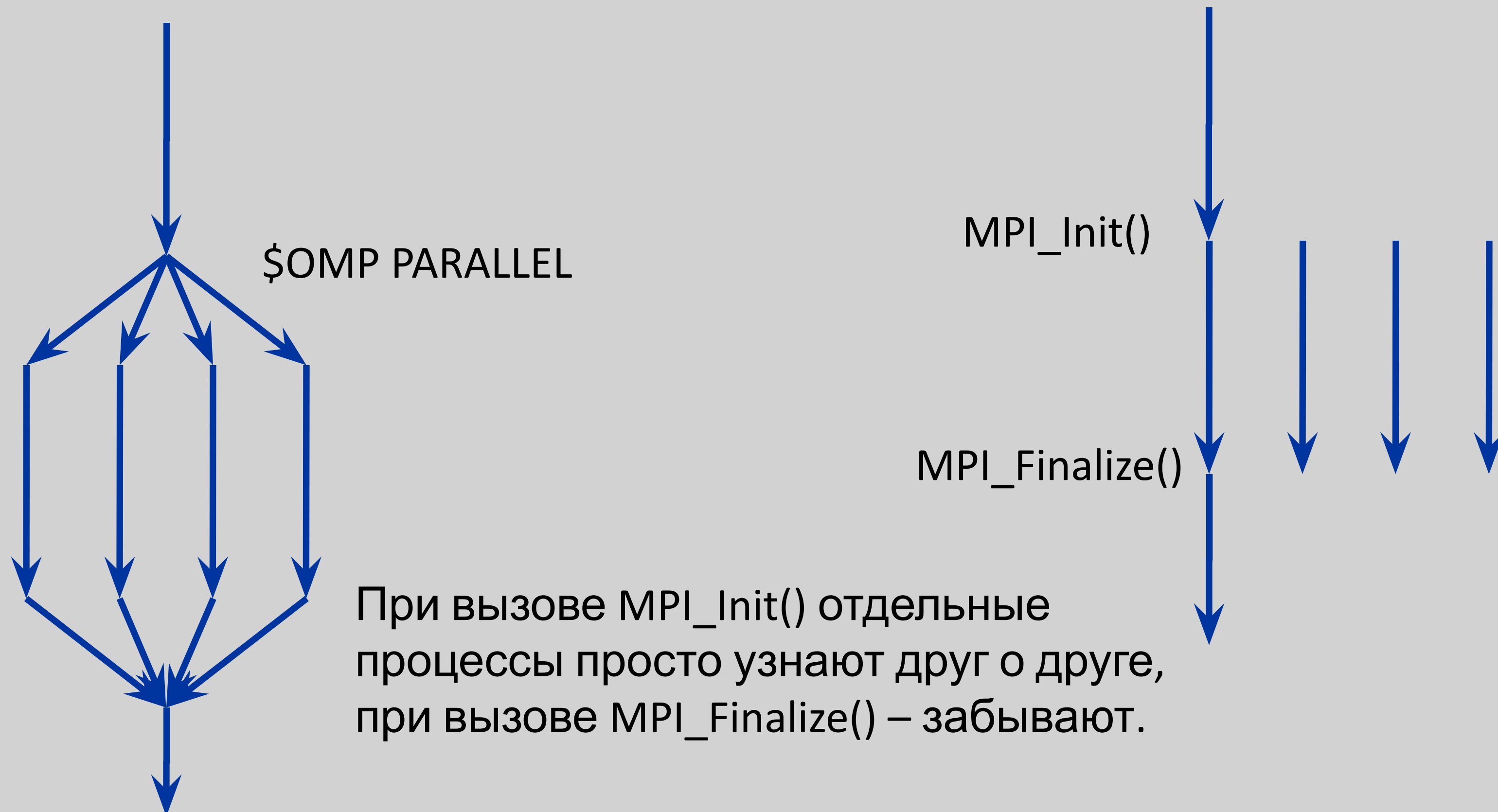
error =MPI_Finalize(); вызовы!

Что изменилось? НИЧЕГО! Ни результат, ни время расчета не изменится

В отличие от OpenMP, MPI не даёт автоматического параллелизма! Нужно хорошо поработать, чтобы получить параллельную программу.

Понятие процесса в вычислениях на компьютерах с распределённой памятью

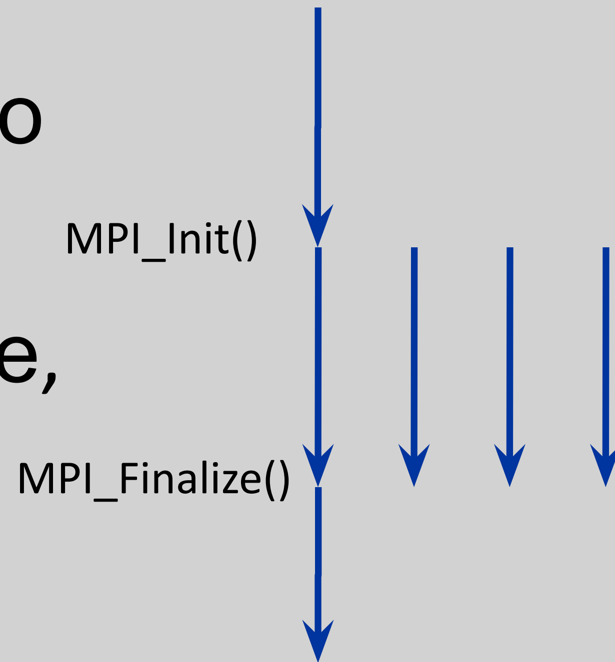
Отличие MPI_Init() от $OMP PARALLEL:



В отличии от OpenMP в MPI программах нет shared, firstprivate и т. п. переменных – все переменные private, каждый процесс выполняет свою самостоятельную программу.

Понятие процесса в вычислениях на компьютерах с распределённой памятью

- MPI процесс – это отдельный набор команд с данными (программа), исполняемый независимо на (вирутально) независимом компьютере, осведомлённый о существовании других подобных себе наборов команд с данными



Не путать с процессом в ОС!

Основные инструменты MPI

Основные функции:

int MPI_Init (int *argc, char **argv) инициализирует окружение MPI

int MPI_Finalize() выход из окружения MPI

int MPI_Comm_size (MPI_Comm comm, int *size) возвращает количество процессов

int MPI_Comm_rank (MPI_Comm comm, int *rank) возвращает номер текущего процесса (ранг = порядковый номер, отсчёт всегда начинается с 0)

comm = коммуникатор - структура, в которой хранятся все связи между процессами, информация о том, какие процессы вовлечены в вычисления и т.д.

Коммуникации one-to-one, блокирующие и неблокирующие передачи

Базовые функции пересылки:

int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) отправляет сообщение

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

tag – уникальный номер посылки, целое число больше нуля. В Recv может быть MPI_ANY_TAG – прекрасная возможность для ошибки в исполнении. Tag в Send соответствует tag в Recv!!!!

status – показывает, откуда и с каким tagом пришла посылка. Нужно, если пользуетесь MPI_ANY_TAG или MPI_ANY_SOURCE

source\dest – ранг(номер) процесса, который отправляет\получает

Коммуникации one-to-one, блокирующие и неблокирующие передачи

MPI_Datatype - типы данных в MPI:

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

Так же MPI позволяет создать свои типы данных, но как это делать – это уже высокая материя не для этого курса

Коммуникации one-to-one, блокирующие и неблокирующие передачи

Пример простейшей программы:

```
...
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  {Большая работа}
  if (rank == 0)
  {
   strcpy(message, "Hello, world!");
   for (i = 1; i < size; i++)
   MPI_Send(message, 20, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
   MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf( "Message from process = %d : %.14s\n", rank, message);
  MPI_Finalize();
  return 0;
}
```

MPI тип появился до старта MPI процессов

«Старт» MPI процессов

Самоидентификация MPI процессов

if способ распределения работы

for способ распределения работы

Не забывайте проверять коды ошибок!

«Финиш» MPI процессов

Потенциальная проблема - Send может производится без открытого Recv – программа выйдет с ошибкой

Коммуникации one-to-one, блокирующие и неблокирующие передачи

Основные неблокирующие функции пересылки:

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) отправляет сообщение

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request) получает сообщение

request – структура, которая хранит информацию, что отправленно/получено, от кого и с каким тагом…

При вызовах MPI_Isend/MPI_Irecv программа не ждет, пока посылка отправится/будет получена, исполнение идет дальше. Как узнать, отправилась ли посылка в итоге или дошла?

Коммуникации one-to-one, блокирующие и неблокирующие передачи

Основные неблокирующие функции пересылки:

int MPI_Wait (MPI_Request *request, MPI_Status *status) – барьер, пока посылка не дойдет/ не отправится

MPI_Isend()+MPI_Wait()=MPI_Send

int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status) – проверка, получена/отправлена посылка или нет.

flag – «логическая» переменная, которая отвечает на этот вопрос

Коммуникации one-to-one, блокирующие и неблокирующие передачи

Тот же пример, более корректный:

```
...
#include "mpi.h"
int main(int argc, char **argv )
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Request request;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank!=0) MPI_Irecv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &request);
  {Большая работа}
  if (rank == 0)
  {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
    MPI_Send(message, 14, MPI_CHAR, i, tag, MPI_COMM_WORLD);
   }
   else
    MPI_Wait(&request, &status);
  printf( "Message from process = %d : %.14s\n", rank, message);
  MPI_Finalize();
  return 0;
}
```

Но что если запустить эту программу в последовательном режиме? Она зависнет …

Примеры элементарных ошибок

Та же самая программа, но на фортране:

```
...
include ‘mpif.h’
program main
  char message(20)
  integer i, rank, size, tag
  integer*8 request
  integer*8 status
  call MPI_Init()
  call MPI_Comm_size(MPI_COMM_WORLD, size)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank)
  tag=99
  if (rank.ne.0) call MPI_Irecv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, request)
  Большая работа
  if (rank .eq. 0)  then
    message =  “Hello, world!"
    do i = 1, size
      call MPI_Send(message, 20, MPI_CHAR, i, tag, MPI_COMM_WORLD)
    enddo
   else
    call MPI_Wait(&request, &status)
  endif
  print *, ‘Message from process = ‘, rank, message
  call MPI_Finalize()
end
```

Во всех MPI функциях пропущен последний параметр – error, а в результате программа где-то падает...

Примеры элементарных ошибок

Другая популярная ошибка на примере этой же программы:

```
...
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Request request;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank!=0) MPI_Irecv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &request);
  {Большая работа}
  if (rank == 0)
  {
    strcpy(message, "Hello, world!");
    for (i = 0; i < size; i++)
    MPI_Send(message, 20, MPI_CHAR, i, tag, MPI_COMM_WORLD);
   }
   else
    MPI_Wait(&request, &status);
  printf( "Message from process = %d : %.14s\n", rank,message);
  MPI_Finalize();
  return 0;
}
```

Выход по ошибке: попытка отправить посылку с неоткрытым приемником. Всего лишь начали цикл с 0, а не с 1...

Примеры элементарных ошибок

Другая популярная ошибка на примере этой же программы:

```
...
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Request request;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank!=0) MPI_Irecv(message, 20, MPI_CHAR, 0, &tag, MPI_COMM_WORLD, &request);
 {Большая работа}
  if (rank == 0)
  {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
    MPI_Send(message, 14, MPI_CHAR, i, &tag, MPI_COMM_WORLD);
   }
   else
    MPI_Wait(&request, &status);
  printf( "Message from process = %d : %.14s\n", rank,message);
  MPI_Finalize();
  return 0;
}
```

Вместо tag идет &tag… Несмотря на то, что формально на всех процессах tag один и тот же, адрес у них может быть разный... А может и одинаковый, что придает программе еще большую оригинальность ☺

Задания на понимание

1. **Нарисуйте блок схему, реализующую параллельное умножение матрицы на вектор, где матрица распределена по процессам**
    1. **по столбцам**
    2. **по строкам**
    3. **в шахматном порядке**
2. **Нарисуйте блок-схему, реализующую параллельное вычисление числа π. Один из способов вычисления числа π выглядит так: в квадрат 2 на 2 равномерно случайно набрасываются точки. Число π равно умноженному на 4 отношению количества точек, расстояние до которых от центра меньше 1 к общему числу точек.**
3. **Нарисуйте блок схему, реализующую параллельное вычисление 1-ой, 2-ой и равномерной нормы вектора**
4. **Реализуйте один из вышеперечисленных алгоритмов в виде MPI программы и выполните её на «кластере»**

Коллективные коммуникации

Задача: Напишите блок-схему, реализующую параллельное вычисление числа π. Псевдокод:

```
#include "mpi.h"
int main(int argc, char **argv)
{
  int i, rank, size, type = 99, max_proc; \\ maximum number of process
  double pi_proc[max_proc],pi;
  MPI_Request request[max_proc];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank!=0) for (i = 1; i < size; i++)  MPI_Irecv(pi_proc[i], 1, MPI_DOUBLE, i, type, MPI_COMM_WORLD, &request[i]);
  {Calculate pi on each process;}
  if (rank != 0)
  {
     MPI_Send(message, 1, MPI_DOUBLE, pi, type, MPI_COMM_WORLD);
   }
   else
  {
    for (i = 1; i < size; i++) MPI_Wait(&request[i], &status);
  }
for (i = 1; i < size; i++) pi = pi + pi_proc[i];
pi = pi/size;
MPI_Finalize();
return 0;
}
```

НЕУДОБНО!!!

Коллективные коммуникации

int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type, **MPI_Op op**, int root, MPI_Comm comm)

op – операция, которая должна быть выполнена над данными, например, суммирование, определение максимума и т.д. Примеры:

Операция	Описание
MPI_MAX / MPI_MIN	Определение max/min значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_LAND / MPI_LOR	Логические операции и/или
MPI_BAND / MPI_BOR	Битовые операции и/или
MPI_MAXLOC / MPI_MANLOC	Определение max / min значений и их индексов

И ряд других. Более того, операции могут определятся самостоятельно.

Основные ошибки:

- count, type, op, root, comm – на всех процессах одинаковы! Иначе непредсказуемое падение
- sendbuf, recvbuf – должны указывать на разные элементы! Иначе ответ неправильный

Коллективные коммуникации

Задача: Напишите блок-схему, реализующую параллельное вычисление числа π.
Псевдокод с помощью MPI_REDUCE:

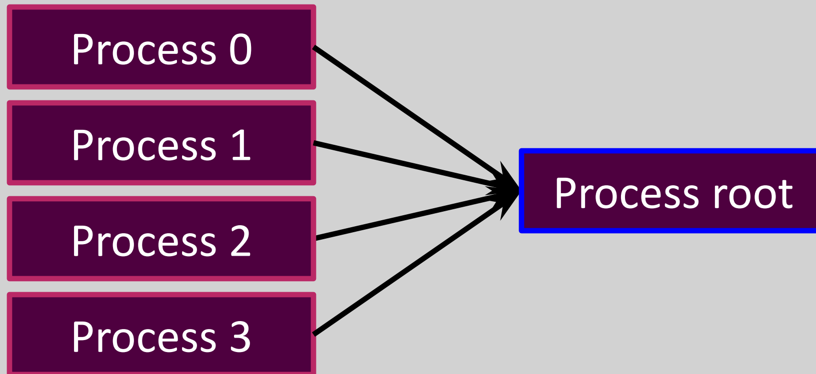
```
#include "mpi.h"
int main(int argc, char **argv)
{
  int i, rank, size, type = 99, max_proc; \\ maximum number of process
  int master_rank = 0;
  double pi_proc[max_proc],pi;
  MPI_Request request[max_proc];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  {Calculate pi on each process;}
  MPI_Reduce(&pi_proc[i], &pi, 1,MPI_Double, MPI_SUM, master_root, comm);
  MPI_Finalize();
  return 0;
}
```

Значительно удобнее, более того – значительно быстрее.

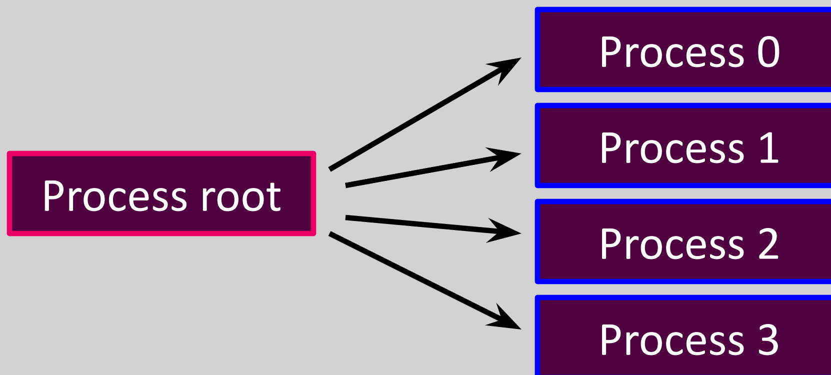
Коллективные операции обычно выполняются быстрее, чем соответствующий им набор неколлективных!

Коллективные коммуникации

MPI_Reduce:



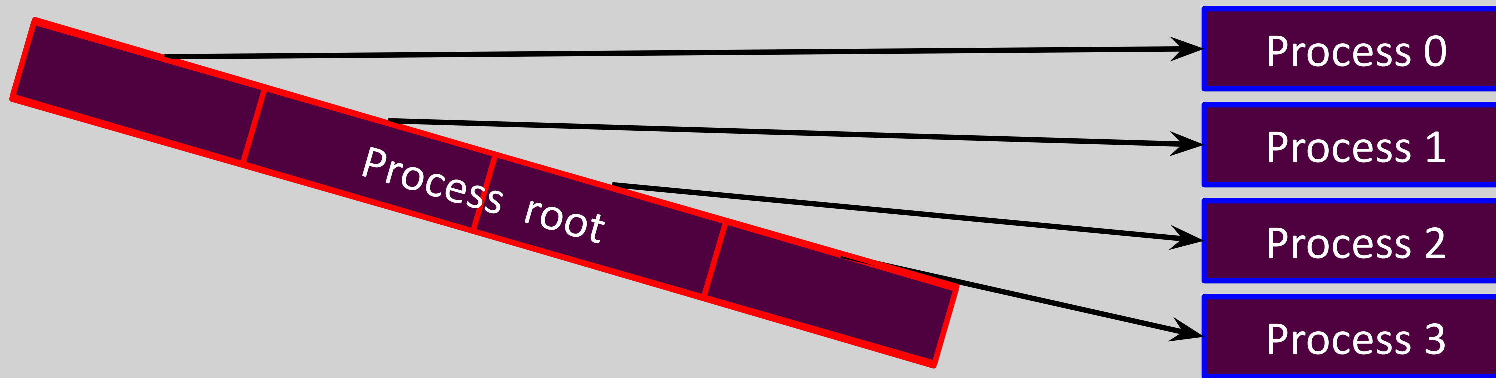
MPI_Bcast(void *buf, int count, MPI_Datatype type, int root, MPI_Comm comm):



buf – адрес отправки для root и адрес приема для всех остальных

Коллективные коммуникации

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



sbuf, rbuf – адреса посылки соответственно для отправителя и получателя

scount – количество элементов отправляемых на каждый процесс. На каждый процесс отправляется одинаковое количество элементов!!!

stype, rtype – типы элементов в посылке соответственно для отправителя и получателя, формально могут быть разные

rcount – количество элементов, принимаемых получателем.

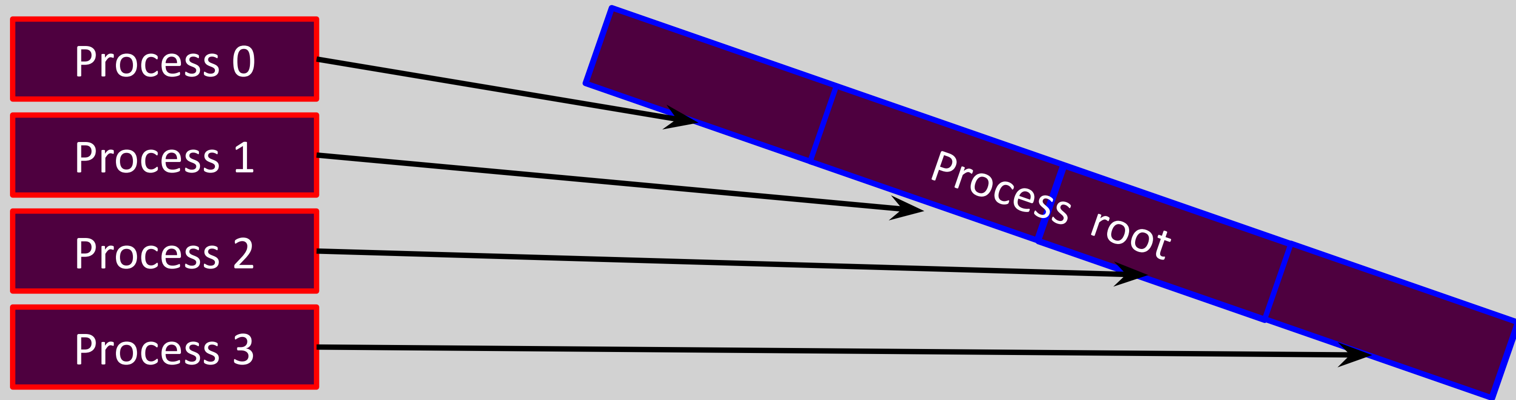
ВАЖНО! sizeof(stype)*scount=sizeof(rtype)*rcount. Лучше всегда

stype=rtype;

scount=rcount;

Коллективные коммуникации

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



sbuf, rbuf – адреса посылки для отправителя и получателя соответственно

scount – количество элементов отправляемых с каждого процесса. Каждый процессор отправляет одинаковое количество элементов!!!

stype, rtype – типы элементов в посылке соответственно для отправителя и получателя, формально могут быть разные

rcount – количество элементов, принимаемых получателем

ВАЖНО! sizeof(stype)*scount=sizeof(rtype)*rcount. Лучше всегда

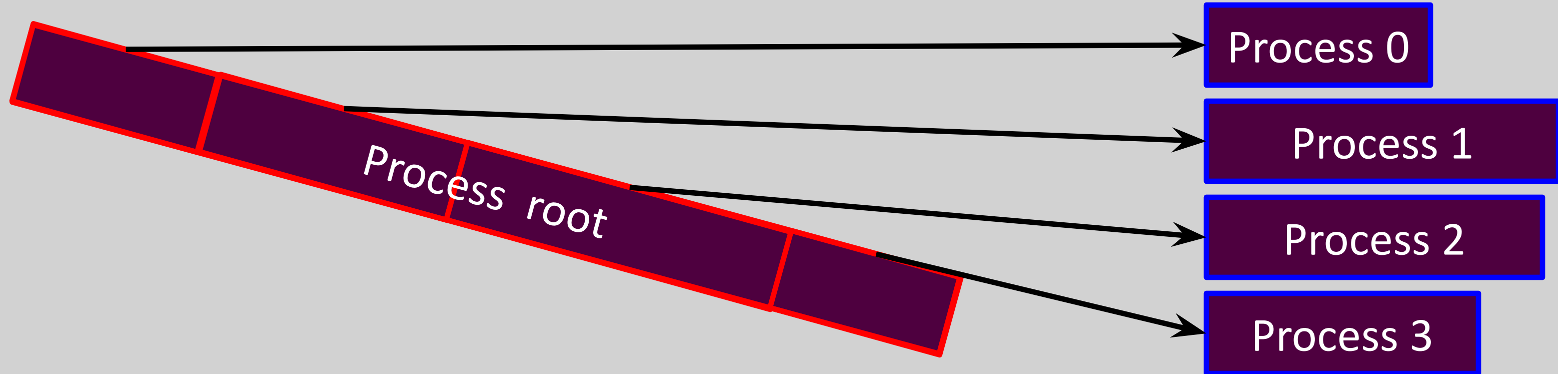
stype=rtype;

scount=rcount;

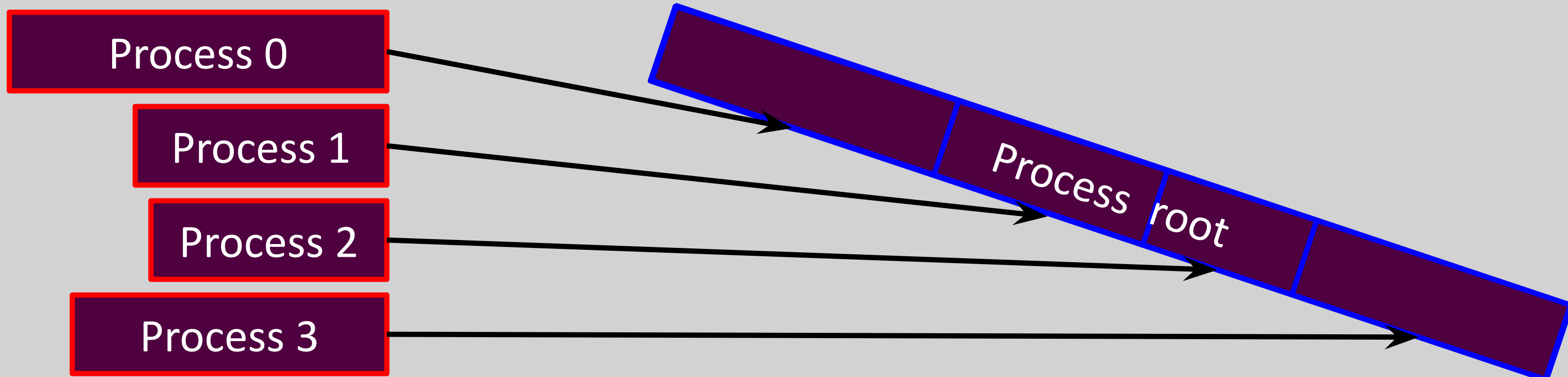
Коллективные коммуникации

MPI_Gather и MPI_Scatter расссылают посылки одинакового объема, что бывает неудобно. Для рассылки разного веса используются такие же функции с дополнительной “v” (vector) в конце. Такой принцип используется для многих функций MPI:

int MPI_Scatterv(void *sbuf, int *sendcnts, int *displs, MPI_Datatype sendtype, void *rbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

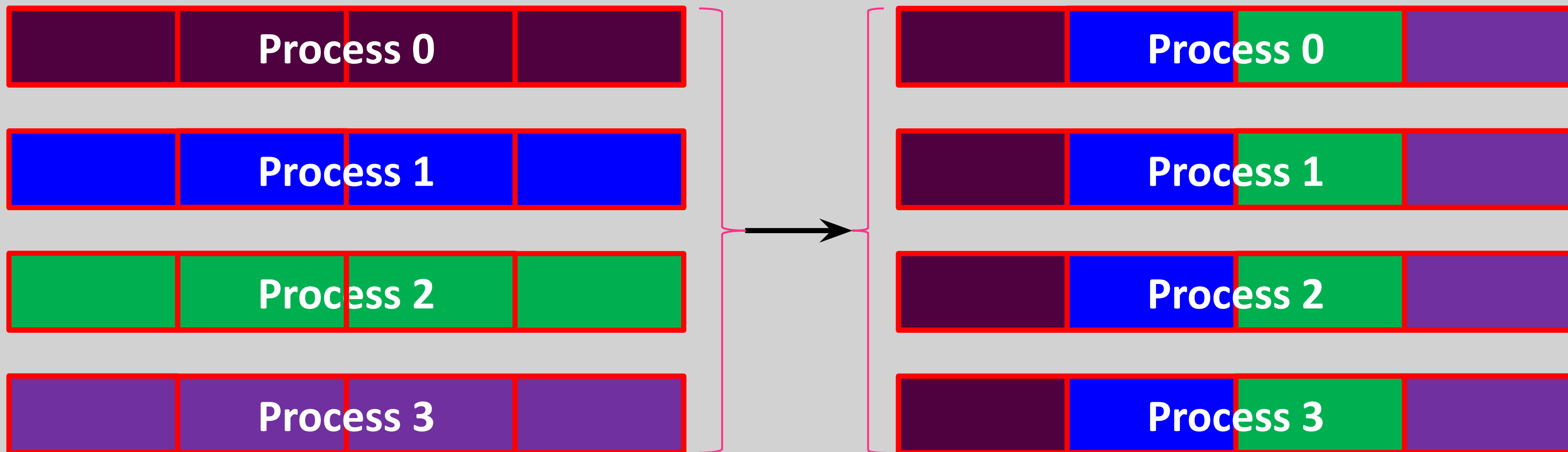


int MPI_Gatherv (void *sbuf, int sendcnt, MPI_Datatype sendtype, void *rbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)



Коллективные коммуникации

int MPI_Alltoall(**void *sbuf**, **int scount**, **MPI_Datatype stype**, **void *rbuf**, **int rcount**, **MPI_Datatype rtype**, MPI_Comm comm)



sbuf, rbuf – адреса посылки для отправителя и получателя соответственно

scount, rcount – количество элементов отправляемых/получаемых для каждого процесс

stype, rtype – типы элементов в посылке для отправителя и получателя соответственно

Коллективные коммуникации

Полезные мелочи:

int MPI_Barrier(MPI_Comm comm) – останавливает MPI процессы в comm до того момента, пока они все не дойдут до данной точки.

double MPI_Wtime(void) – глобальный счетчик времени

Пример использования:

```
t1 = MPI_Wtime();
{…}
t2 = MPI_Wtime();
dt = t2 - t1;
```

double MPI_Wtick(void) – время в секундах одного тика MPI_Wtime()

```
int main( int argc, char *argv[] )
{
   double tick;
   MPI_Init();
   tick = MPI_Wtick ();
   printf("A single MPI tick is %0.9f seconds\n", tick);
   fflush(stdout);
   MPI_Finalize( );
   return 0;
}
```

Работа с группами и коммуникаторами

MPI_Comm*comm – коммуникатор

Что это такое? Структура, хранящая информацию о процессах, используемых в работе. Есть три константы, с которыми можно работать, как с коммуникатором:

1. MPI_COMM_WORLD – все процессы, которые поданы пользователем, собраны в этом коммуникаторе

2. MPI_COMM_SELF – в коммуникаторе находится только процесс, на котором используется данный коммуникатор

3. MPI_COMM_NULL – пустой/нулевой коммуникатор

Пример использования:

MPI_Comm_size(MPI_COMM_SELF, &size);

size всегда будет равна 1

Работа с группами и коммуникаторами

Можно ли присвоить один коммуникатор другому, например,

Comm = MPI_COMM_WORLD;?

НЕТ! Правильно:

```
MPI_Comm new_comm;
MPI_Comm_dup( MPI_COMM_WORLD, &new_comm);
```

В общем виде:

```
MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
```

Как создать новый коммуникатор, который объединяет в себе ряд процессов из предыдущего, но не все?

int MPI_comm_create (MPI_Comm oldcom, **MPI_Group group**, MPI_Comm *newcomm).

MPI_Group group – новый термин, структура, которая определяет набор процессов. Похожа на MPI_Comm, но значительно проще, не несет в себе способов коммуникации и т.п.

Любой коммукатор создается на основе группы! Чтоб научиться создавать любые коммуникаторы, необходимо научиться работать с группой.

Работа с группами и коммуникаторами

*int MPI_Group_size (MPI_Group *group, int *size)* – считает размер группы

*int MPI_Group_rank (MPI_Group group, int *rank)* – считает номер данного процесса в группе

*int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)* – строит группу на основе данного коммуникатора

*int MPI_Group_union (MPI_Group group1, MPI_Group group2, MPI_Group *group_out)* – строит группу как объединение двух

*int MPI_Group_intersection (MPI_Group group1, MPI_Group group2, MPI_Group *group_out)* – строит группу как пересечение двух

*int MPI_Group_difference (MPI_Group group1, MPI_Group group2, MPI_Group *group_out)* – строит группу как прямую разницу двух

*int MPI_Group_incl (MPI_Group group, int n, int *ranks, MPI_Group *group_out)* – строит новую группу из членов старой с номерами rank[0], rank[1],..,rank[n-1]

*int MPI_Group_excl (MPI_Group group, int n, int *ranks, MPI_Group *newgroup)* – строит новую группу из старой исключая номера rank[0], rank[1],..,rank[n-1]

*int MPI_Group_free (MPI_Group *group)* – уничтожение группы

Работа с группами и коммуникаторами

Пример: построить коммуникатор, который содержит только процессы той же четности, что и вызывающий (результат: должно получится 2 коммуникатора)

```
#include "mpi.h"
int main(int argc, char **argv)
{
  int i, rank, size, size_new, max_proc;
  int ranks[max_proc];
  MPI_Group world_group, new_group;
  MPI_Comm new_comm;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int MPI_Comm_group(MPI_COMM_WORLD, &world_group);
  if (2*(rank/2)==rank)
    for (i = 0; i < (size+1)/2; i++) ranks[i] = 2*i;
      size_new = (size+1)/2;
  else
    for (i = 0; i < size/2; i++) ranks[i] = 2*i+1;
      size_new = size/2;
MPI_Group_incl(world_group, size_new, ranks, &new_group);
MPI_Comm_create (MPI_COMM_WORLD, new_group, &new_comm);
MPI_Comm_size(new_comm, &size);
printf("I am process number %d, size of my new group if %d\n", rank, size);
MPI_Group_free(&group);
MPI_Comm_free (&new_comm);
MPI_Finalize();
return 0;
}
```

Не забывайте удалять созданные вами объекты

Работа с группами и коммуникаторами

Другой способ построить новый коммуникатор без использования групп:

int MPI_Comm_split(MPI_Comm comm, **int color**, **int key**, MPI_Comm *comm_out)

color – способ разбиения процессов. Процессы с одинаковым “цветом” окажутся в одном коммуникаторе

key – номер данного процесса в новом коммуникаторе (прекрасная возможность для ошибки, например одинаковый key для разных процессов)

Работа с группами и коммуникаторами

Тот же пример, но уже с помощью MPI_Comm_split: построить коммуникатор, который содержит только процессоры той же четности, что и вызывающий (как результат, должно получится 2 коммуникатора)

```
#include "mpi.h"
int main(int argc, char **argv)
{
  int i, rank, size, color, key, max_proc;
  int ranks[max_proc];
  MPI_Comm new_comm;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  color = rank-2*(rank/2);
  key = rank/2;
  MPI_Comm_split(MPI_COMM_WORLD, color, key, &new_comm);
  MPI_Comm_size(new_comm, &size);
  printf(“I am process number %d, size of my new group if %d\n”, rank, size);
  MPI_Comm_free (&new_comm);
  MPI_Finalize();
  return 0;
}
```

Резюме

MPI распараллеливание основано на вызове подпрограмм в отличие от OpenMP

MPI процессы знают, что они не одни в этом мире, но весь обмен информацией должны обеспечить вы (программист). Иначе все процессы сделают «одно и то же».

MPI программа исполняет один и тот же код для всех процессов, но данные легко могут быть разными на разных процессах несмотря на одинаковое название переменных!

Отладка MPI программы сложна, так как зависит не только от кластера, но и от реализации MPI и диспетчера. Дополнительная сложность возникает, если у вашей программы нет серийного варианта.

Задания на понимание

1. **Напишите MPI программу проверяющую, что фигуры стоящие на шахматной доске не бьют друг друга. Всего процессов 64, каждый процесс соответствует одной шахматной клетке, глобальная нумерация num = (i-1)*8+j, где i,j – соответственно строки и столбы шахматной доски. Написать задачу с условием того, что на шахматной доске расставлены только:**
   a) **Ладьи**
   b) **Слоны**
   c) **Ферзи**
   d) **Короли**