基于MPI+OpenMP的并行程序设计

崔焕庆

2020年10月3日

目录
CONTENTS

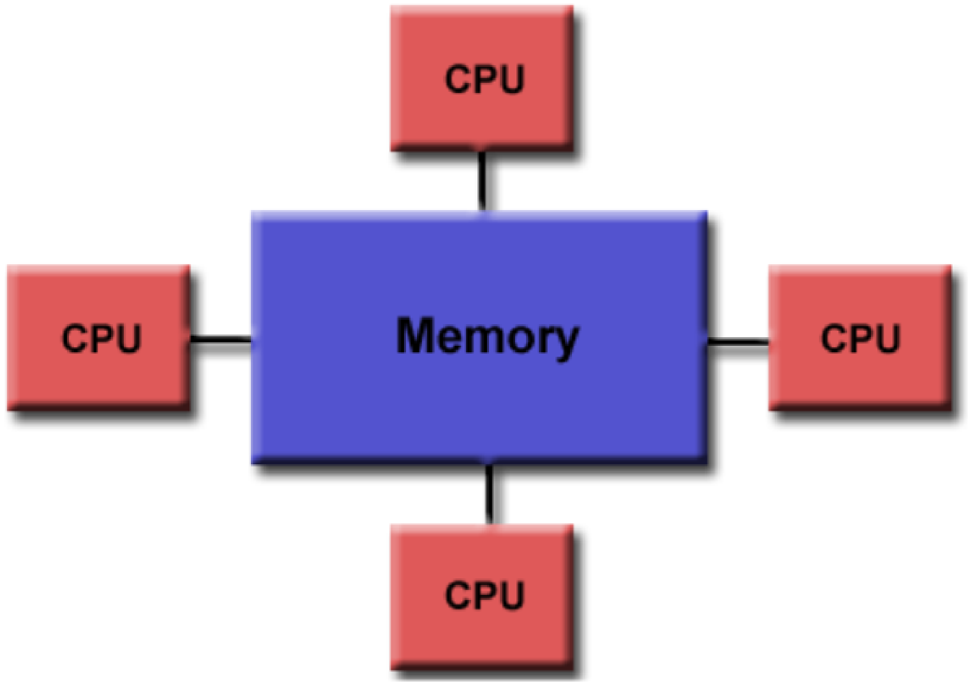
1 OpenMP简介

1.1 OpenMP的优势

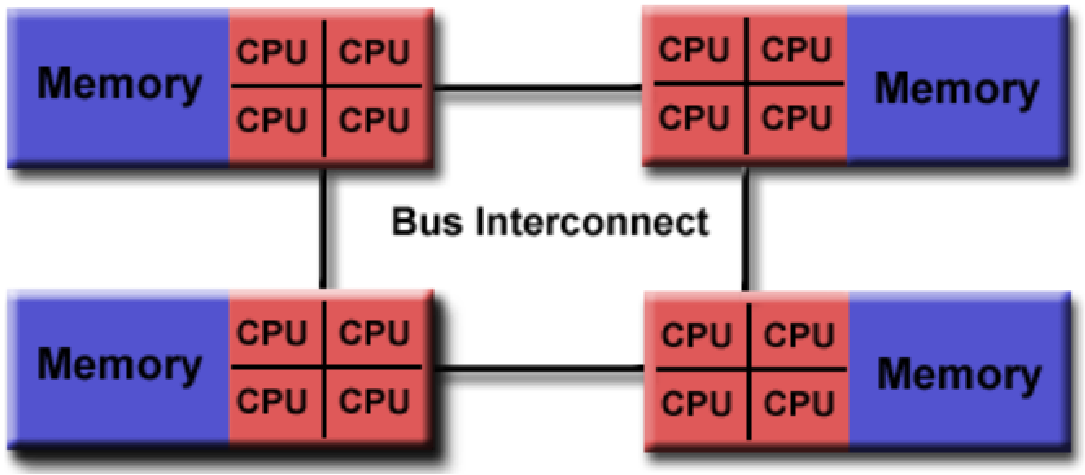
OpenMP是由OpenMP Architecture Review Board牵头提出的，并已被广泛接受的、用于**共享内存并行系统**的多线程程序设计的一套编译指令(Compiler Directive)。OpenMP支持、**C、C++**和Fortran；而支持OpenMP的编译器包括Sun Compiler、**GNU Compiler**和Intel Compiler等。OpenMP提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的pragma来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些pragma，或者编译器不支持OpenMP时，程序又可退化为通常的程序(一般为串行)，代码仍然可以正常运作，只是不能利用多线程来加速程序执行。

1 OpenMP简介

1.1 OpenMP的优势



统一内存访问

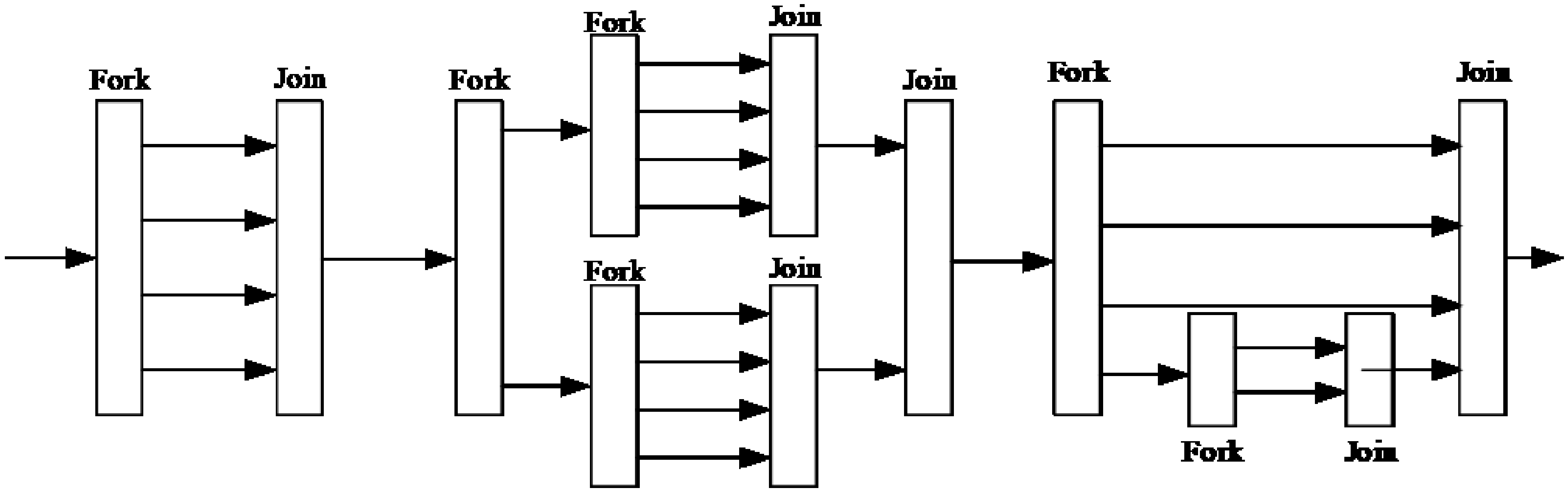


非统一内存访问

1 OpenMP简介

1.2 OpenMP的Fork-Join并行执行模型

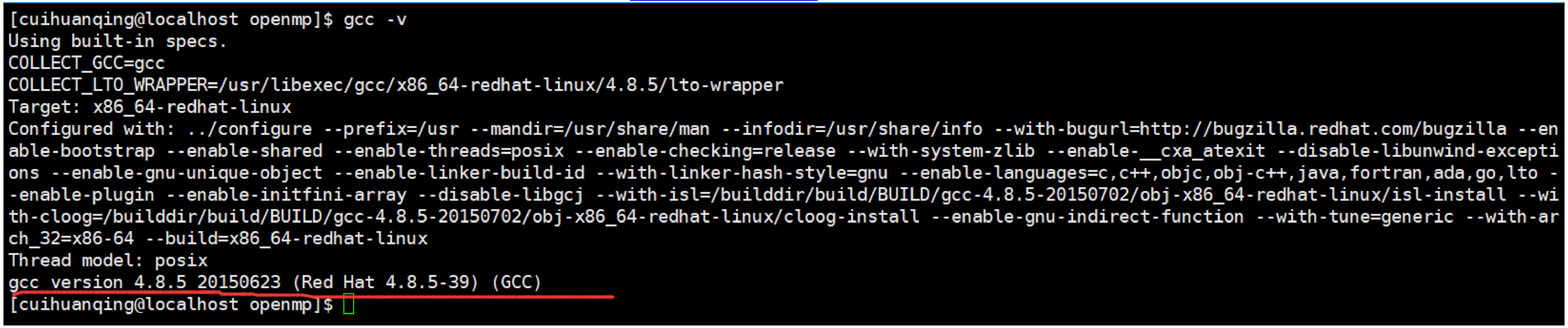
起始时只有一个主线程，当遇到**Fork**操作时，创建或唤醒多个子线程进入并行任务执行，并行执行结束后，实现隐式的同步，汇合到主线程中，即**Join**。相邻的Fork和Join操作之间称为一个**并行域**，**并行域可以嵌套**。



1 OpenMP简介

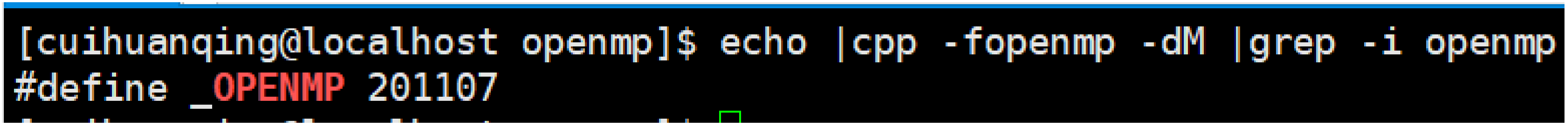
1.3 OpenMP的版本

$gcc -v



```
[cuihuanqing@localhost openmp]$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/4.8.5/lto-wrapper
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --en
able-bootstrap --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-__cxa_atexit --disable-libunwind-excepti
ons --enable-gnu-unique-object --enable-linker-build-id --with-linker-hash-style=gnu --enable-languages=c,c++,objc,obj-c++,java,fortran,ada,go,lto -
-enable-plugin --enable-initfini-array --disable-libgcj --with-isl=/builddir/build/BUILD/gcc-4.8.5-20150702/obj-x86_64-redhat-linux/isl-install --wi
th-cloog=/builddir/build/BUILD/gcc-4.8.5-20150702/obj-x86_64-redhat-linux/cloog-install --enable-gnu-indirect-function --with-tune=generic --with-ar
ch_32=x86-64 --build=x86_64-redhat-linux
Thread model: posix
gcc version 4.8.5 20150623 (Red Hat 4.8.5-39) (GCC)
[cuihuanqing@localhost openmp]$
```

$echo |cpp -fopenmp -dM |grep -i openmp

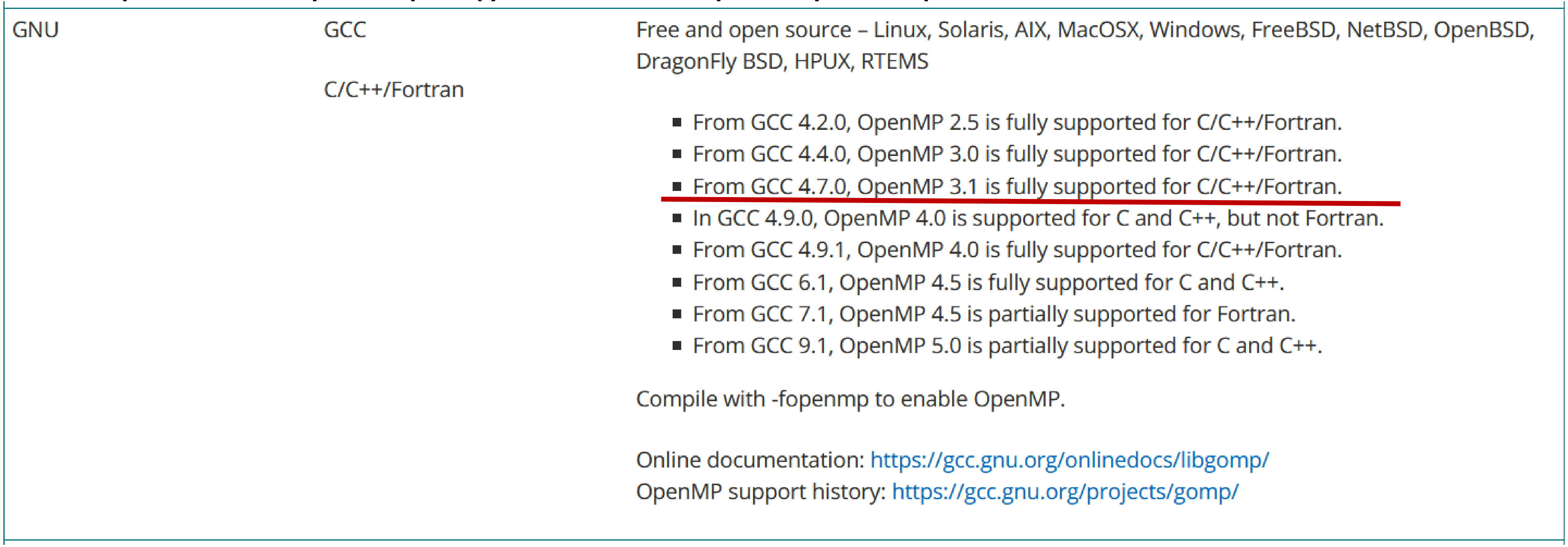


```
[cuihuanqing@localhost openmp]$ echo |cpp -fopenmp -dM |grep -i openmp
#define _OPENMP 201107
```

1 OpenMP简介

1.3 OpenMP的版本

https://www.openmp.org/resources/openmp-compilers-tools/



GNU

GCC

C/C++/Fortran

Free and open source – Linux, Solaris, AIX, MacOSX, Windows, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, HPUX, RTEMS

- From GCC 4.2.0, OpenMP 2.5 is fully supported for C/C++/Fortran.
- From GCC 4.4.0, OpenMP 3.0 is fully supported for C/C++/Fortran.
- From GCC 4.7.0, OpenMP 3.1 is fully supported for C/C++/Fortran.
- In GCC 4.9.0, OpenMP 4.0 is supported for C and C++, but not Fortran.
- From GCC 4.9.1, OpenMP 4.0 is fully supported for C/C++/Fortran.
- From GCC 6.1, OpenMP 4.5 is fully supported for C and C++.
- From GCC 7.1, OpenMP 4.5 is partially supported for Fortran.
- From GCC 9.1, OpenMP 5.0 is partially supported for C and C++.

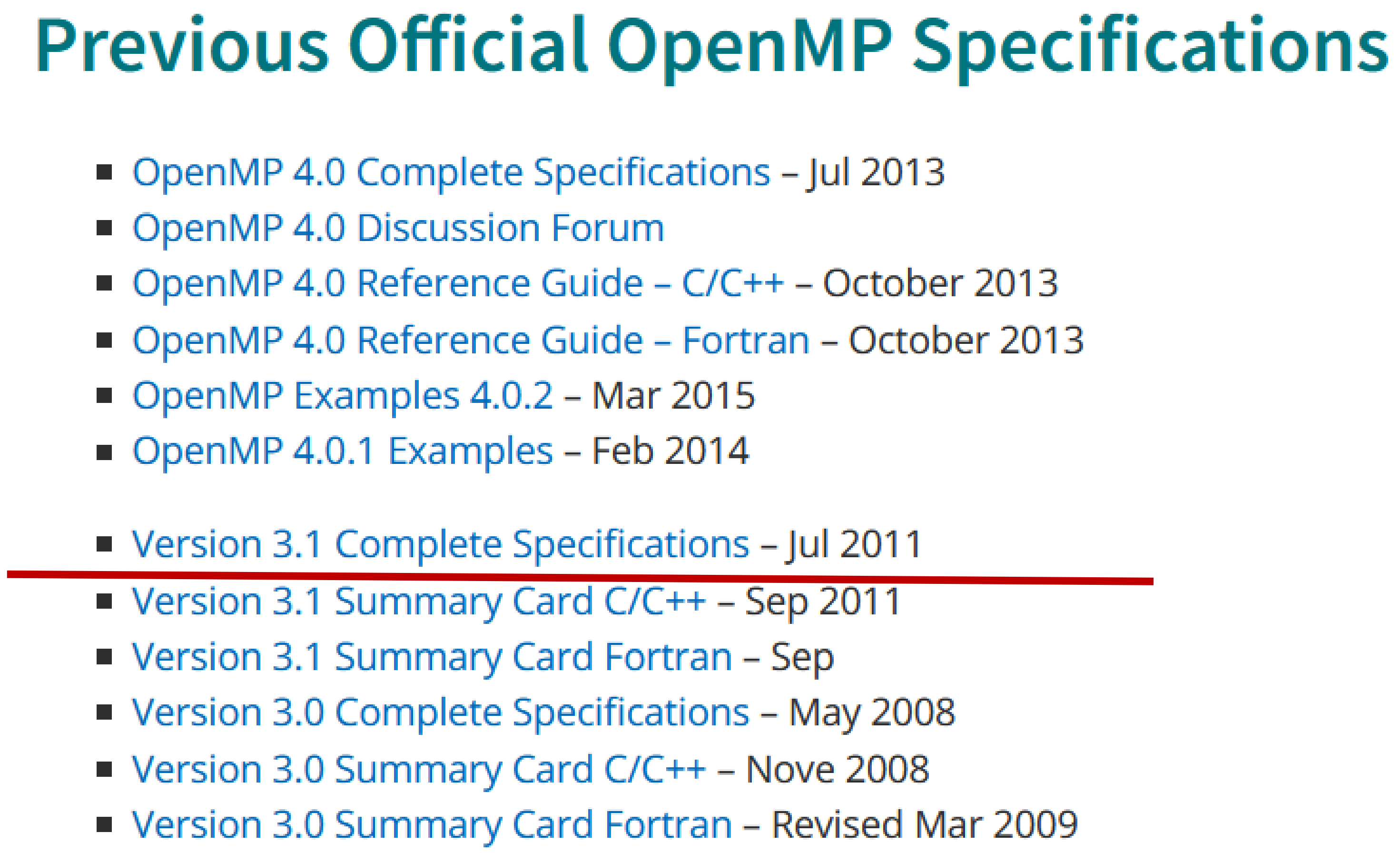
Compile with -fopenmp to enable OpenMP.

Online documentation: https://gcc.gnu.org/onlinedocs/libgomp/
OpenMP support history: https://gcc.gnu.org/projects/gomp/

1 OpenMP简介

1.3 OpenMP的版本

https://www.openmp.org/specifications/



Previous Official OpenMP Specifications

- OpenMP 4.0 Complete Specifications – Jul 2013
- OpenMP 4.0 Discussion Forum
- OpenMP 4.0 Reference Guide – C/C++ – October 2013
- OpenMP 4.0 Reference Guide – Fortran – October 2013
- OpenMP Examples 4.0.2 – Mar 2015
- OpenMP 4.0.1 Examples – Feb 2014

- Version 3.1 Complete Specifications – Jul 2011
- Version 3.1 Summary Card C/C++ – Sep 2011
- Version 3.1 Summary Card Fortran – Sep
- Version 3.0 Complete Specifications – May 2008
- Version 3.0 Summary Card C/C++ – Nove 2008
- Version 3.0 Summary Card Fortran – Revised Mar 2009

1 OpenMP简介

1.4 OpenMP编译制导

所有的编译制导指令必须以“#pragma omp”开头，directive-name是特定的指令名，指明需要OpenMP完成的动作，clause是从句，是可选项，用于作为对指定指令的进一步说明。需要指出的是，在C/C++语言中，编译制导指令是区分大小写的。

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
```

1 OpenMP简介

1.4 OpenMP编译制导

(1) **parallel指令**

- 用于构造一个并行块，如果它所构造的并行块有多于1条的语句，需要用一对大括号括起来。
- 并行区域必须是一个完整的代码块，不能使用goto或其他语句跳出或转入。

1 OpenMP简介

1.4 OpenMP编译制导

(1) **parallel指令**

```
$gcc hello.c -o hello -fopenmp
```

```
#include <stdio.h>
int main(){
    #pragma omp parallel
    printf("The parallel region is run by thread %d.\n",omp_get_thread_num());
    return 0;
}
```

运行时库函数，用于返回在线程组中一个线程的编号。

1 OpenMP简介

1.4 OpenMP编译制导

(2) **for指令**

- 用于对循环工作进行并行化，它通常需要与parallel合并使用。
- for循环必须具备一定的规范格式：

$$\text{for}\ \ (i = start; i \left\{ \begin{matrix} < \\ <= \\ >= \\ > \end{matrix} \right\} end; \left\{ \begin{matrix} i++ \\ ++i \\ i-- \\ --i \\ i+=inc \\ i-=inc \\ i = inc + i \\ i = i + inc \\ i = i - inc \end{matrix} \right\}$$

1 OpenMP简介

1.4 OpenMP编译制导

(2) **for指令**

```
//没有与parallel同时使用，不能并行
#include <stdio.h>
int main(){
    int i;
    #pragma omp for
    for (i = 0; i < 4; i++)
        printf("i = %d, threadId = %d.\n", i, omp_get_thread_num());
    return 0;
}
```

输出：

```
i = 0, threadId = 0.
i = 1, threadId = 0.
i = 2, threadId = 0.
i = 3, threadId = 0.
```

1 OpenMP简介

1.4 OpenMP编译制导

(2) **for指令**

```
//与parallel同时使用，能并行
#include <stdio.h>
int main(){
    int i;
    #pragma omp parallel for
    for (i = 0; i < 4; i++)
        printf("i = %d, threadId = %d.\n", i, omp_get_thread_num());
    return 0;
}
```

输出：

```
i = 0, threadId = 0.
i = 1, threadId = 1.
i = 2, threadId = 2.
i = 3, threadId = 3.
```

1 OpenMP简介

1.4 OpenMP编译制导

(2) **for指令**

```
//与parallel同时使用，能并行
#include <stdio.h>
int main(){
   int i;
   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < 4; i++)
         printf("i = %d, threadId = %d.\n", i, omp_get_thread_num());
   }
   return 0;
}
```

输出：

```
i = 0, threadId = 0.
i = 1, threadId = 1.
i = 2, threadId = 2.
i = 3, threadId = 3.
```

1 OpenMP简介

1.4 OpenMP编译制导

(3) **sections和section指令**

- 用于使各个线程执行不同的工作。
- 每个section必须是一个结构化的代码块，不能有分支转入或跳出。
- sections中可以定义多个section，每个section仅被一个一线程执行一次。
- 当线程多于section数量时，每个线程最多执行一个section。
- 当线程少于section数量时，有线程会执行多于一个的section。

1 OpenMP简介

1.4 OpenMP编译制导

(3) **sections和section指令**

■ 用法

```
#pragma omp sections [clause[[,] clause]⋯]
{
    [#pragma omp section]
        structured  block
    [#pragma omp section]
        structured block
    ……
}
```

标识构件的开始

标识不同的section

1 OpenMP简介

1.4 OpenMP编译制导

(3) **sections和section指令**

```
#include <stdio.h>
int main(){
    #pragma omp parallel sections
    {
        #pragma omp section
        printf("Hello from %d.\n", omp_get_thread_num());
        #pragma omp section
        printf("Hi from %d.\n", omp_get_thread_num());
        #pragma omp section
        printf("Bye from %d.\n", omp_get_thread_num());
    }
    return 0;
}
```

输出：

```
Bye from 17.
Hi from 28.
Hello from 36.
```

1 OpenMP简介

1.4 OpenMP编译制导

(4) **single指令**

■ 用于让紧随其后的语句串行执行。

```
#include <stdio.h>
int main(){
   int A=100, i;
   #pragma omp parallel for private(A)
   for(i = 0; i<10;i++){
            printf("id: %d, A: %d\n",omp_get_thread_num(), A);
   }
   return 0;
}
```

输出：

```
Run in parallel, thread id = 11.
Run in sequence, thread id = 11.
Run in parallel, thread id = 19.
Run in parallel, thread id = 30.
```

1 OpenMP简介

1.4 OpenMP编译制导

(5) **private数据属性**

- 将一个或多个变量声明为线程的私有变量。
- 每个线程都有它自己的变量私有副本，其他线程无法访问。
- 即使在并行区域外有同名的共享变量，共享变量在并行区域内不起任何作用，并且并行区域内不会操作到外面的共享变量。
- 并行区域内的private变量和并行区域外同名的变量没有存储关联。
- 如果需要**继承原有共享变量的值**，则应使用**firstprivate**子句。
- 如果需要在**退出并行区域时将私有变量最后的值赋值给对应的共享变量**，则可使用**lastprivate**子句。

1 OpenMP简介

1.4 OpenMP编译制导

(5) **private数据属性**

//一个错误的程序

```
#include <stdio.h>
int main(){
    int A=100;
    int i;
    #pragma omp parallel for private(A)
    for(i = 0; i<10;i++){
            printf("%d\n",A);
    }
    return 0;
}
```

输出：

```
id: 2, A: 0
id: 5, A: 0
id: 7, A: 0
id: 3, A: 0
id: 8, A: 0
id: 6, A: 0
id: 4, A: 0
id: 1, A: 0
id: 9, A: 0
id: 0, A: 1218647392
```

1 OpenMP简介

1.4 OpenMP编译制导

(5) **private数据属性**

```
//第二个错误的程序
#include <stdio.h>
int main(){
   int B, i;
#pragma omp parallel for private(B)
   for(i = 0; i<10;i++){
          B = 100;
   }
   printf("%d\n",B);
   return 0;
}
```

输出：

```
0
```

1 OpenMP简介

1.4 OpenMP编译制导

(5) **private数据属性**

//第三个错误的程序
```
#include <stdio.h>
int main(){
   int C = 100,i;
#pragma omp parallel for private(C)
   for(i = 0; i<10;i++){
         C = 200;
         printf("%d\n",C);
   }
   printf("%d\n",C);
   return 0;
}
```

输出：
```
200
200
200
200
200
200
200
200
200
200
100
```

1 OpenMP简介

1.4 OpenMP编译制导

(5) **private数据属性**

```
#include <stdio.h>
int main(){
   int k, i;
   k = 100;
   #pragma omp parallel for firstprivate(k),lastprivate(k)
   for (i = 0; i < 8; i++)
   {
      k += i;
      printf("k = %d in thread %d.\n", k, omp_get_thread_num());
   }
   printf("Finally, k = %d.\n", k);
   return 0;
}
```

输出：

```
k = 101 in thread 1.
k = 107 in thread 7.
k = 105 in thread 5.
k = 100 in thread 0.
k = 103 in thread 3.
k = 104 in thread 4.
k = 102 in thread 2.
k = 106 in thread 6.
Finally, k = 107.
```

1 OpenMP简介

1.4 OpenMP编译制导

(5) **private数据属性**

```
//默认是shared
#include <stdio.h>
int main(){
  int i, j;
  #pragma omp parallel for
  for (i = 0; i < 2; i++){
    for (j = 0; j < 5; j++) //j共享，线程1只执行了1次
        printf("i=%d, j=%d from id=%d.\n", i, j, omp_get_thread_num());
  }
  return 0;
}
```

输出：

```
i=0, j=0 from id=0.
i=0, j=1 from id=0.
i=0, j=2 from id=0.
i=0, j=3 from id=0.
i=0, j=4 from id=0.
i=1, j=0 from id=1.
```

1 OpenMP简介

1.4 OpenMP编译制导

(5) **private数据属性**

```
#include <stdio.h>
int main(){
  int i, j;
  #pragma omp parallel for private(j)
  for (i = 0; i < 2; i++){
    for (j = 0; j < 5; j++)
        printf("i=%d, j=%d from id=%d.\n", i, j, omp_get_thread_num());
  }
  return 0;
}
```

输出：

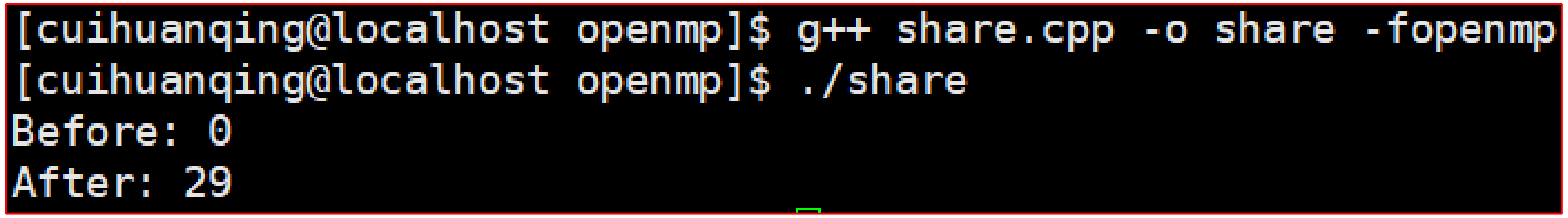
```
i=0, j=0 from id=0.
i=0, j=1 from id=0.
i=0, j=2 from id=0.
i=0, j=3 from id=0.
i=0, j=4 from id=0.
i=1, j=0 from id=1.
i=1, j=1 from id=1.
i=1, j=2 from id=1.
i=1, j=3 from id=1.
i=1, j=4 from id=1.
```

1 OpenMP简介

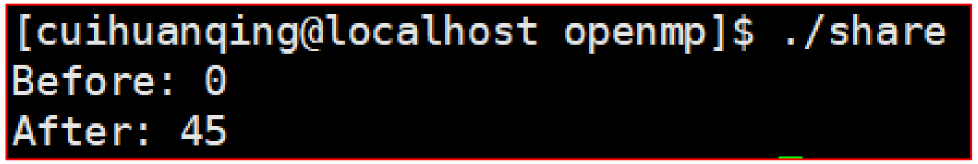
1.4 OpenMP编译制导

(6) **shared数据属性**

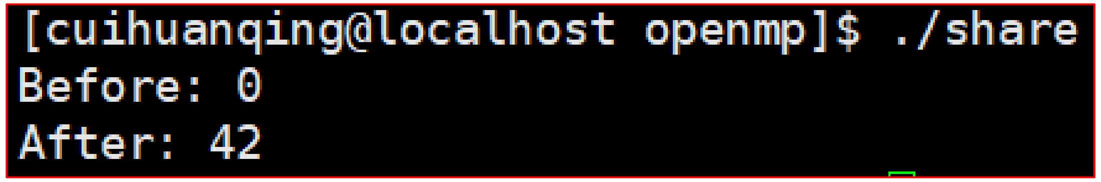
```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    int sum = 0;
    cout << "Before: " << sum << endl;
#pragma omp parallel for shared(sum)
    for (int i = 0; i < 10; ++i) {
        sum += i;  //存在数据竞争
}
    cout << "After: " << sum << endl;
    return 0;
}
```



```
[cuihuanqing@localhost openmp]$ g++ share.cpp -o share -fopenmp
[cuihuanqing@localhost openmp]$ ./share
Before: 0
After: 29
```



```
[cuihuanqing@localhost openmp]$ ./share
Before: 0
After: 45
```



```
[cuihuanqing@localhost openmp]$ ./share
Before: 0
After: 42
```

1 OpenMP简介

1.4 OpenMP编译制导

(6) **reduction子句**

- 为变量指定一个**操作符**，每个线程都会创建reduction变量的**私有拷贝**，在OpenMP区域**结束处**，将使用各个线程的私有拷贝的值通过制定的操作符**进行迭代运算**，并赋值给原来的变量。
- 语法：**recutioin(operator:list)**
- **operator：** **+ * - & ^ | && || max min**

1 OpenMP简介

1.4 OpenMP编译制导

(6) **reduction子句**

```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    int sum = 0;
    cout << "Before: " << sum << endl;
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < 10; ++i) {
        sum += i;
    }
    cout << "After: " << sum << endl;
    return 0;
}
```

1 OpenMP简介

1.5 OpenMP运行库例程与环境变量

(1)设置线程数量。

在OpenMP中，有三种方法可以设定线程个数，按照优先级从高到低分别是：

- 编译制导的num_threads()子句。如：#pragma omp parallel num_threads(8)
- 在并行区域外，使用运行库例程omp_set_num_threads()设定并行区域中使用。如： omp_set_num_threads(10);
- 环境变量OMP_NUM_THREADS。如： export OMP_NUM_THREADS = 3

1 OpenMP简介

1.5 OpenMP运行库例程与环境变量

（2）获取正在使用的线程数量和编号。

- int omp_get_num_threads(void)：返回当前并行区域中的线程数量。
- int omp_get_thread_num(void)：返回值当前并行区域中，当前线程在线程组中的编号。这个编号从0开始。

（3）获取程序可使用的CPU核心数。

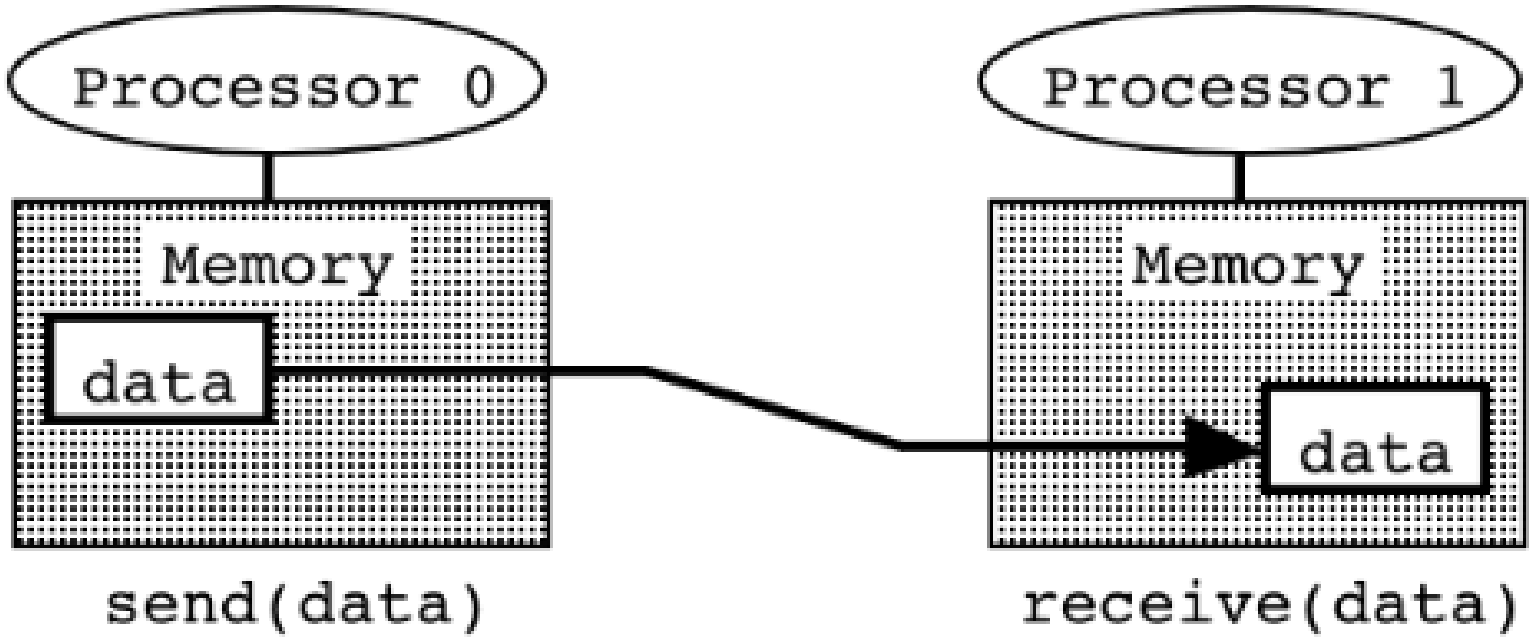
- int omp_get_num_procs(void)：返回值为当前程序可以使用的CPU核数。

（4）获取墙上时间。

- double omp_get_wtime(void)：返回值是以秒为单位的墙上时间。在并行区域开始前和结束后分别调用该函数，并求取两次返回值的差，便可计算出并行执行的时间。

2 MPI简介

2.1 消息传递编程模型



2 MPI简介

2.2 What is MPI?

- ■Message Passing Interface Specification
- ■MPI is a *standardized* and *portable message-passing* system designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the *syntax* and *semantics* of a core of *library routines* useful to a wide range of users writing portable message-passing programs in *C*, *C++,* and *Fortran*.

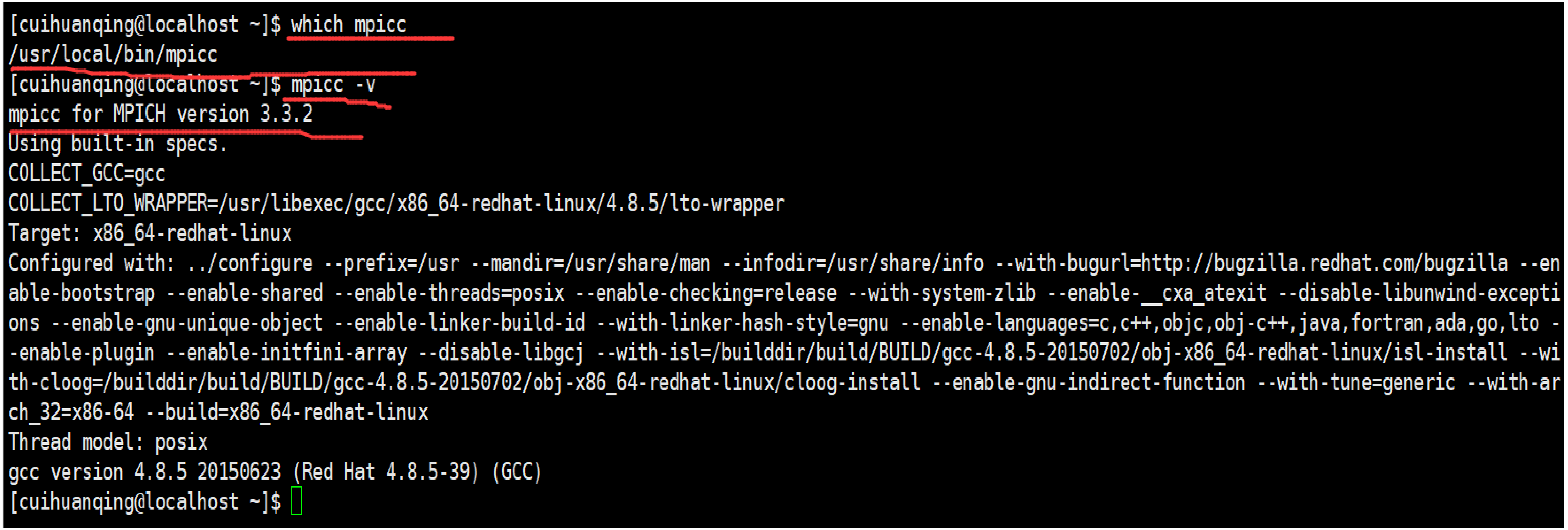
2 MPI简介

2.2 What Is MPI?

- ■Point-to-point communication
- ■Collective operations
- ■Process groups, Communication contexts, and Process topologies
- ■Environmental management and inquiry
- ■Process creation and management
- ■Other functions

2 MPI简介

2.3 Check the Installation and Version

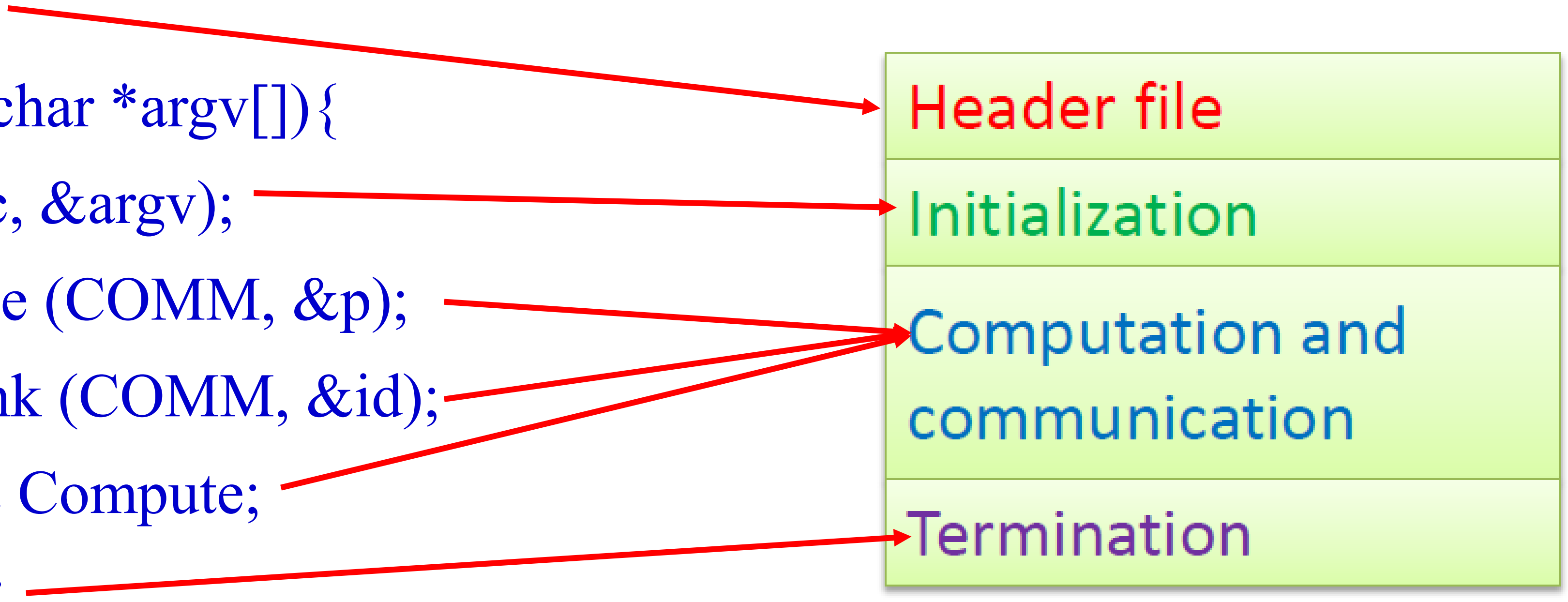


```
[cuihuanqing@localhost ~]$ which mpicc
/usr/local/bin/mpicc
[cuihuanqing@localhost ~]$ mpicc -v
mpicc for MPICH version 3.3.2
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/4.8.5/lto-wrapper
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --en
able-bootstrap --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-__cxa_atexit --disable-libunwind-excepti
ons --enable-gnu-unique-object --enable-linker-build-id --with-linker-hash-style=gnu --enable-languages=c,c++,objc,obj-c++,java,fortran,ada,go,lto -
-enable-plugin --enable-initfini-array --disable-libgcj --with-isl=/builddir/build/BUILD/gcc-4.8.5-20150702/obj-x86_64-redhat-linux/isl-install --wi
th-cloog=/builddir/build/BUILD/gcc-4.8.5-20150702/obj-x86_64-redhat-linux/cloog-install --enable-gnu-indirect-function --with-tune=generic --with-ar
ch_32=x86-64 --build=x86_64-redhat-linux
Thread model: posix
gcc version 4.8.5 20150623 (Red Hat 4.8.5-39) (GCC)
[cuihuanqing@localhost ~]$
```

2 MPI简介

2.4 MPI Programs’ Structure

```
# include "mpi.h"
int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    MPI_Comm_size (COMM, &p);
    MPI_Comm_rank (COMM, &id);
    Communicate & Compute;
    MPI_Finalize( );
     return 0;
}
```



2 MPI简介

2.5 MPI Basic Functions

■Communicators

- ●*Encapsulate all* of these ideas in order to provide the appropriate scope for all communication operations.
- ●Can be regarded as an *ordered list of processes*.
- ●Each process has a *unique rank*, which starts from *0* (root).
- ●It is the *context* of MPI communicators and operations.
- ●When a function is called to send data to *all* processes, MPI needs to *understand* what “all” means.

2 MPI简介

2.5 MPI Basic Functions

■Communicators

- *MPI_COMM_WORLD*: the *default* communicator that *contains all processes* running the MPI program
- Communicators are divided into two kinds:
  - ➢*intra-communicators* for operations within a *single* group of processes.
  - ➢*inter-communicators* for operations between *two* groups of processes.
- *A process* can belong to *multiple communicators.*
- The *rank* is *usually different.*

2 MPI简介

2.5 MPI Basic Functions

■Getting Communicator Information

- ●Get the rank of the calling process in group
  - ➢**int MPI_Comm_rank(MPI_Comm comm, int *rank);**
- ●Get the size in a communicator
  - ➢**int MPI_Comm_size(MPI_Comm comm, int *size);**

2 MPI简介

2.5 MPI Basic Functions

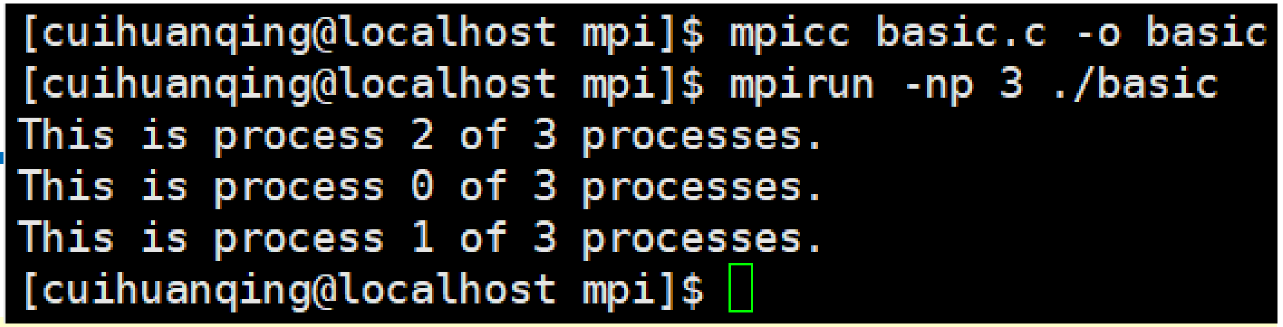
■An example

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char**argv){
  MPI_Init(&argc, &argv);
  printf("Hello world.\n");
  MPI_Finalize();
  return 0;
}
```

2 MPI简介

2.5 MPI Basic Functions

■An example



```
[cuihuanqing@localhost mpi]$ mpicc basic.c -o basic
[cuihuanqing@localhost mpi]$ mpirun -np 3 ./basic
This is process 2 of 3 processes.
This is process 0 of 3 processes.
This is process 1 of 3 processes.
[cuihuanqing@localhost mpi]$ 
```

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
  MPI_Comm comm = MPI_COMM_WORLD;
  int size, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  printf("This is process %d of %d processes.\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

2 MPI简介

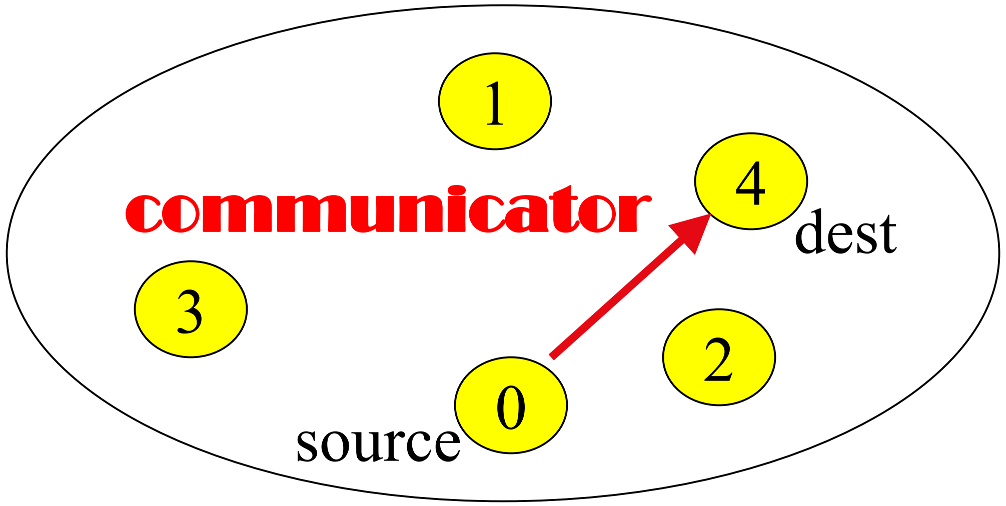
2.6 Point-to-point communication

■Communication between *two* processes

■*Source* process sends message to *destination* process

■Communication takes place *within a communicator*

■Destination process is identified by *its rank* in the communicator



2 MPI简介

2.6 Point-to-point communication

MPI_Send(**Send a message**

void* data,	starting address of the data to be sent
int count,	number of elements to be sent (not bytes)
MPI_Datatype datatype,	MPI datatype of each element
int destination,	rank of destination process
int tag,	message identifier (set by user)
MPI_Comm comm)	MPI communicator of processors involved

2 MPI简介

2.6 Point-to-point communication

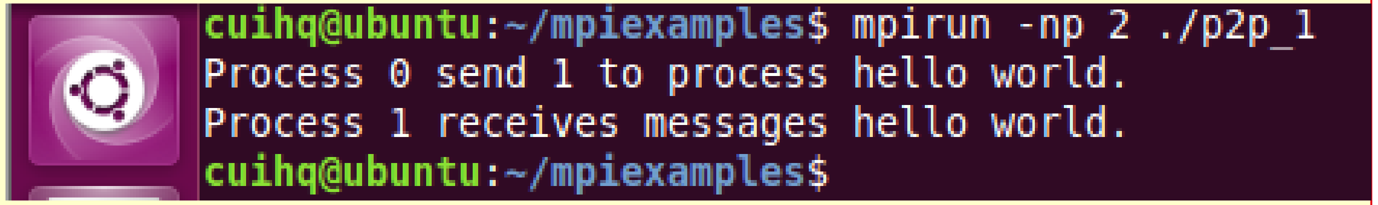
Receive a message

MPI_Recv(	
void* data,	starting address of buffer to store message
int count,	number of elements to be received (not bytes)
MPI_Datatype datatype,	MPI datatype of each element
int source,	rank of source process
int tag,	message identifier (set by user)
MPI_Comm comm,	MPI communicator of processors involved
MPI_Status* status)	structure of information about the message

2 MPI简介

2.6 Point-to-point communication

```
int main(int argc, char **argv){
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Status status;  int size, rank;  char str[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(comm, &size);  MPI_Comm_rank(comm, &rank);
  if (rank == 0)  {
     strcpy(str, "hello world");
     printf("Process 0 send 1 to process %s.\n", str);
     MPI_Send(str, strlen(str) + 1, MPI_CHAR, 1, 99, comm);
  }
  else if (rank == 1)  {
     MPI_Recv(str, 100, MPI_CHAR, 0, 99, comm, &status);
     printf("Process 1 receives messages %s.\n", str);
  }
  MPI_Finalize();
  return 0;
}
```



```
cuihq@ubuntu:~/mpiexamples$ mpirun -np 2 ./p2p_1
Process 0 send 1 to process hello world.
Process 1 receives messages hello world.
cuihq@ubuntu:~/mpiexamples$
```

2 MPI简介

2.7 Elementary MPI datatypes

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

2 MPI简介

2.8 Collective Communication

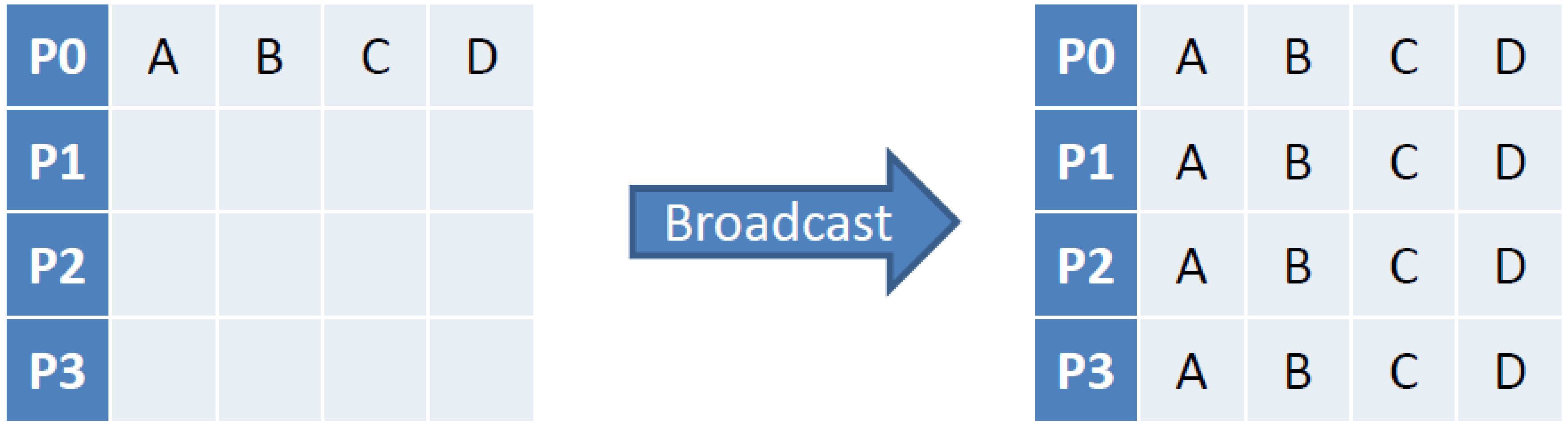
■Collective *vs.* Point-to-point

- ■More *concise* program : *One collective* operation can *replace multiple point-to-point* operations
- ■*Optimized collective* communications usually are *faster than* the corresponding *point-to-point* communications

2 MPI简介

2.8 Collective Communication

■Data Movement: Broadcast



■ *Broadcast copies* data from the memory of *one* processor to that of *other* processors —— *One to all* operation

2 MPI简介

2.8 Collective Communication

■Data Movement: Broadcast

int MPI_Bcast(

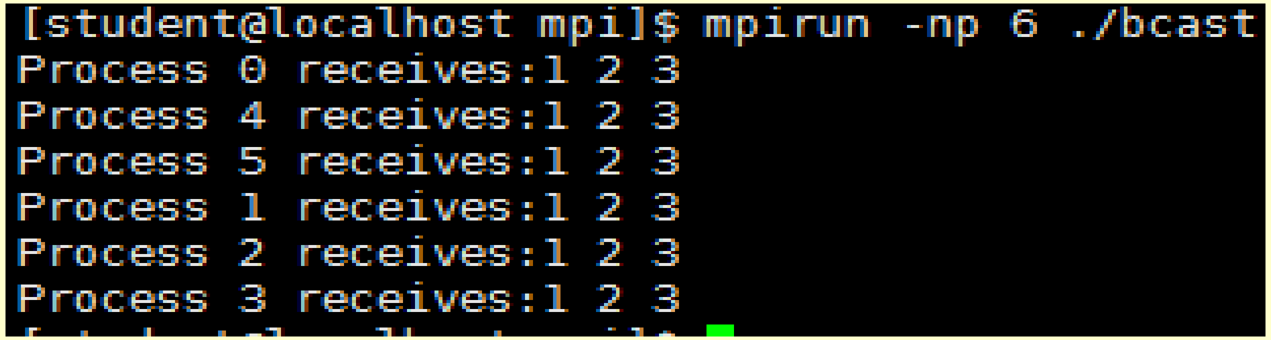
void* buffer,	starting address of buffer
int count,	number of entries in buffer
MPI_Datatype datatype,	data type of buffer
int root,	rank of broadcast root
MPI_Comm comm)	communicator

2 MPI简介

2.8 Collective Communication

■Data Movement: Broadcast

```
int main(int argc, char** argv){
   int arr[3], i, rank;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0){
     for (i = 0; i < 3; i++)
        arr[i] = i + 1;
   }
   MPI_Bcast(arr, 3, MPI_INT, 0, MPI_COMM_WORLD);
   printf("Process %d receives:", rank);
   for (i = 0; i < 3; i++)
      printf("%d ", arr[i]);
   putchar('\n');
   MPI_Finalize();
   return 0;
}
```

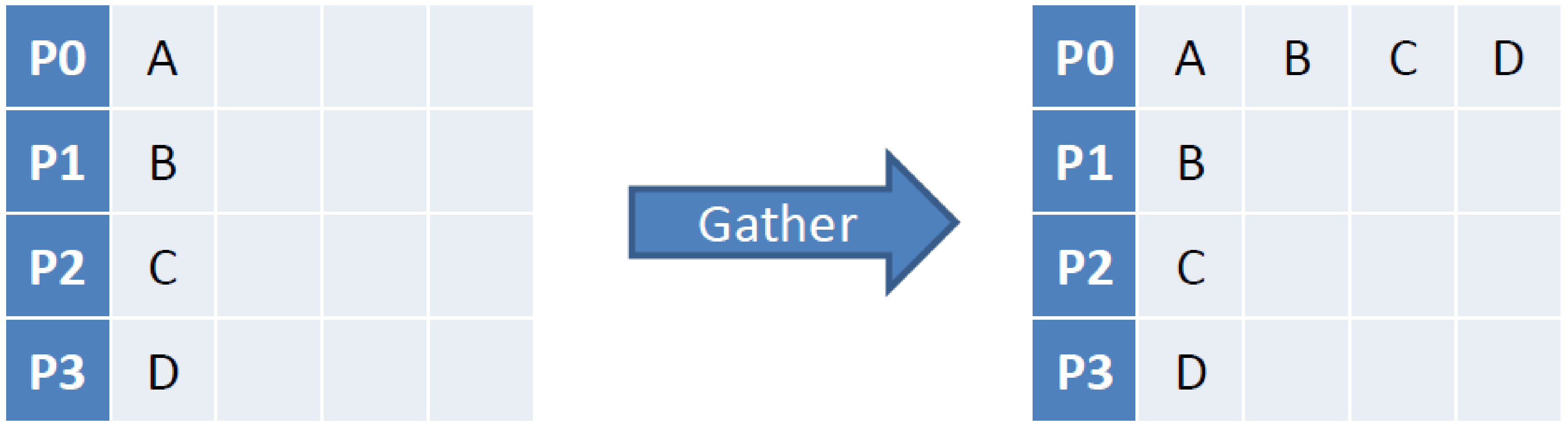


```
[student@localhost mpi]$ mpirun -np 6 ./bcast
Process 0 receives:1 2 3
Process 4 receives:1 2 3
Process 5 receives:1 2 3
Process 1 receives:1 2 3
Process 2 receives:1 2 3
Process 3 receives:1 2 3
```

2 MPI简介

2.8 Collective Communication

■Data Movement: Gather



- *Gather copies* data from *each process* to *one* process, where it is *stored in rank order*—— *One to all* operation

2 MPI简介

2.8 Collective Communication

■Data Movement: Gather

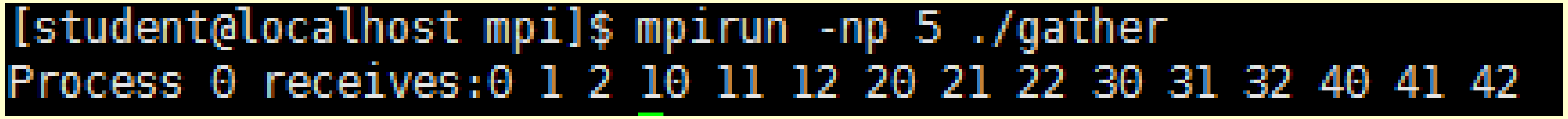
int MPI_Gather(	
const void* sendbuf,	starting address of send buffer
int sendcount,	number of elements in send buffer
MPI_Datatype sendtype,	data type of send buffer elements
void* recvbuf,	address of receive buffer (significant only at root)
int recvcount,	number of elements for any single receive (significant only at root)
MPI_Datatype recvtype,	data type of recv buffer elements(significant only at root)
int root,	rank of receiving process
MPI_Comm comm)	communicator

2 MPI简介

2.8 Collective Communication

■Data Movement: Gather

```
int main(int argc, char **argv){
   int rank, size, sbuf[3], *rbuf, i;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   for (i = 0; i < 3; i++)      sbuf[i] = rank * 10 + i;
   if (rank == 0)      rbuf = (int*)malloc(sizeof(int) * 3 * size);
   MPI_Gather(sbuf, 3, MPI_INT, rbuf, 3, MPI_INT, 0, MPI_COMM_WORLD);
   if (rank == 0){
      printf("Process 0 receives:");
      for (i = 0; i < size * 3; i++)
        printf("%d ", rbuf[i]);
      putchar('\n');
   }
   MPI_Finalize();
   return 0;
}
```

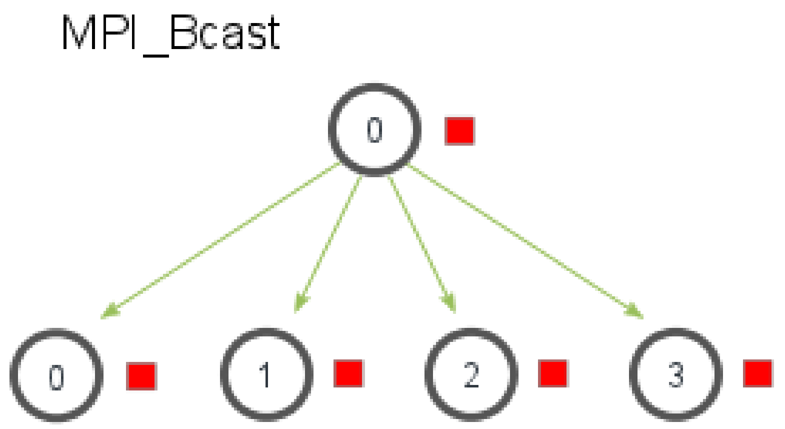


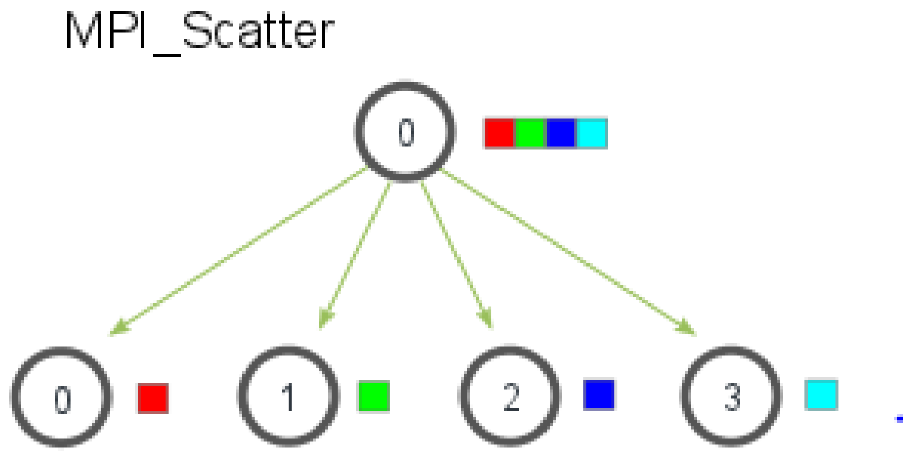
```
[student@localhost mpi]$ mpirun -np 5 ./gather
Process 0 receives:0 1 2 10 11 12 20 21 22 30 31 32 40 41 42
```

2 MPI简介

2.8 Collective Communication

■Data Movement: Scatter





■ Typically data is in an array on the root process and we want to send a *different portion* of the array to each worker process (often including the root).

2 MPI简介

2.8 Collective Communication

■Data Movement: Scatter

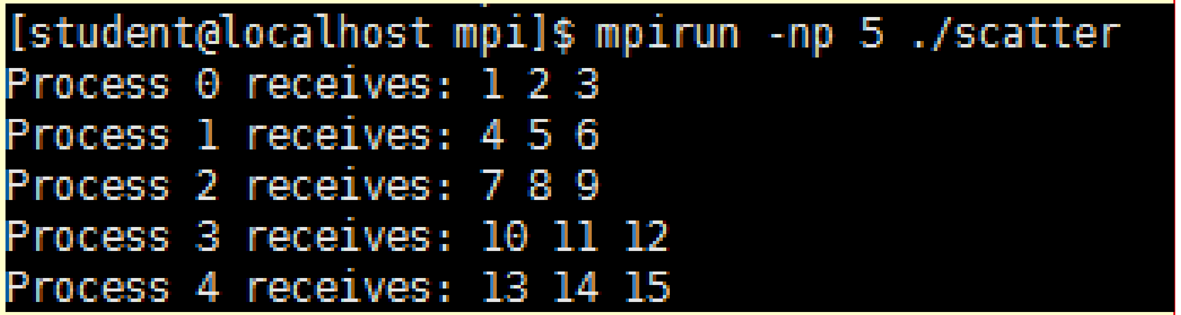
```
int MPI_Scatter (
void * sendbuf , // pointer to send buffer
int sendcount , // items to send per process
MPI_Datatype sendtype , // type of send buffer data
void * recvbuf , // pointer to receive buffer
int recvcount , // number of items to receive
MPI_Datatype recvtype , // type of receive buffer data
int root , // rank of sending process
MPI_Comm comm ) // MPI communicator to use
```

2 MPI简介

2.8 Collective Communication

■Data Movement: Gather

```
int main(int argc, char** argv){
  int rank, size, *sbuf, rbuf[3], i;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0){
    sbuf = (int *) malloc(sizeof(int) * 3 * size);
    for (i = 0; i < size * 3; i++)        sbuf[i] = i + 1;
  }
  MPI_Scatter(sbuf, 3, MPI_INT, rbuf, 3, MPI_INT, 0, MPI_COMM_WORLD);
  printf("Process %d receives: ", rank);
  for (i = 0; i < 3; i++)    printf("%d ", rbuf[i]);
  putchar('\n');
  MPI_Finalize();
  return 0;
}
```

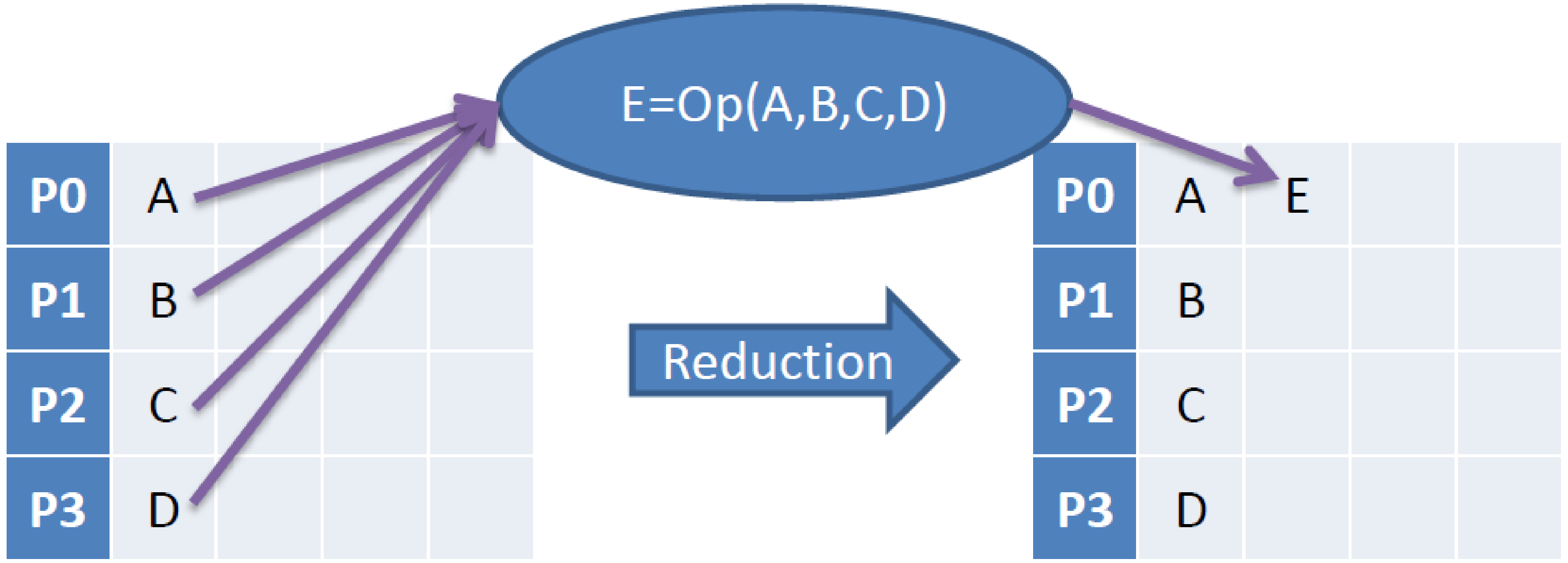


```
[student@localhost mpi]$ mpirun -np 5 ./scatter
Process 0 receives: 1 2 3
Process 1 receives: 4 5 6
Process 2 receives: 7 8 9
Process 3 receives: 10 11 12
Process 4 receives: 13 14 15
```

2 MPI简介

2.8 Collective Communication

■Data Movement: Reduce



- MPI reduction collects data from each process, reduces them to a single value, and store it in the memory of one process.

2 MPI简介

2.8 Collective Communication

■Data Movement: Reduce

MPI_Reduce(

void* send_data,	address of send buffer
void* recv_data,	address of receive buffer
int count,	number of elements in send buffer
MPI_Datatype datatype,	data type of elements of send buffer
MPI_Op op,	reduce operation
int root,	rank of root process
MPI_Comm communicator)	communicator

2 MPI简介

2.8 Collective Communication

■Data Movement: Reduce

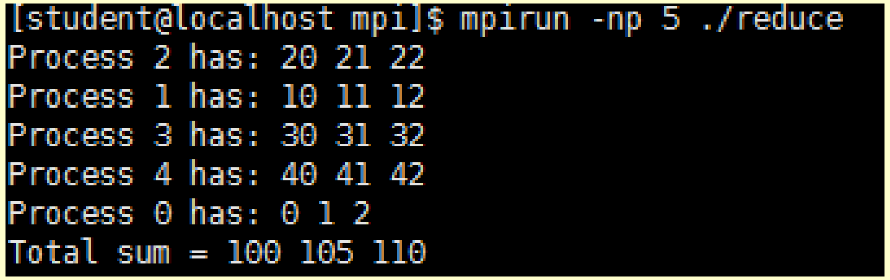
Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

2 MPI简介

2.8 Collective Communication

■Data Movement: Reduce

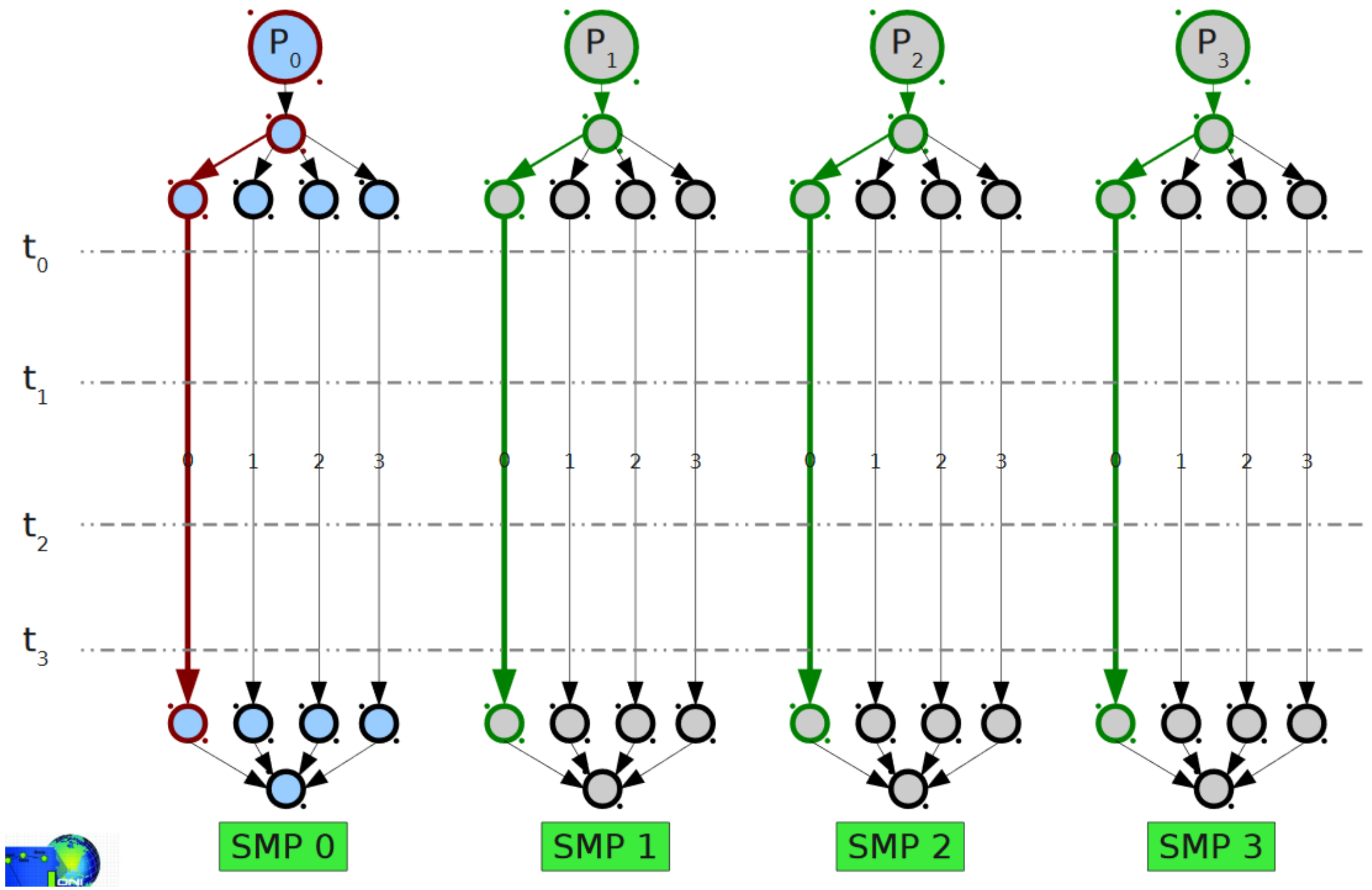
```
int main(int argc, char** argv) {
  int size, rank, sbuf[3], rbuf[3], i;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  for (i = 0; i < 3; i++)       sbuf[i] = rank * 10 + i;
  printf("Process %d has: ", rank);
  for (i = 0; i < 3; i++)      printf("%d ", sbuf[i]);
  putchar('\n');
  MPI_Reduce(sbuf, rbuf, 3, MPI_INT, MPI_SUM, 0,MPI_COMM_WORLD);
  if (rank == 0) {
    printf("Total sum = ");
    for (i = 0; i < 3; i++)      printf("%d ",rbuf[i]);
    putchar('\n');
  }
  MPI_Finalize();
}
```



```
[student@localhost mpi]$ mpirun -np 5 ./reduce
Process 2 has: 20 21 22
Process 1 has: 10 11 12
Process 3 has: 30 31 32
Process 4 has: 40 41 42
Process 0 has: 0 1 2
Total sum = 100 105 110
```

3 MPI+OpenMP混合编程简介

What Does This Scenario Look Like?

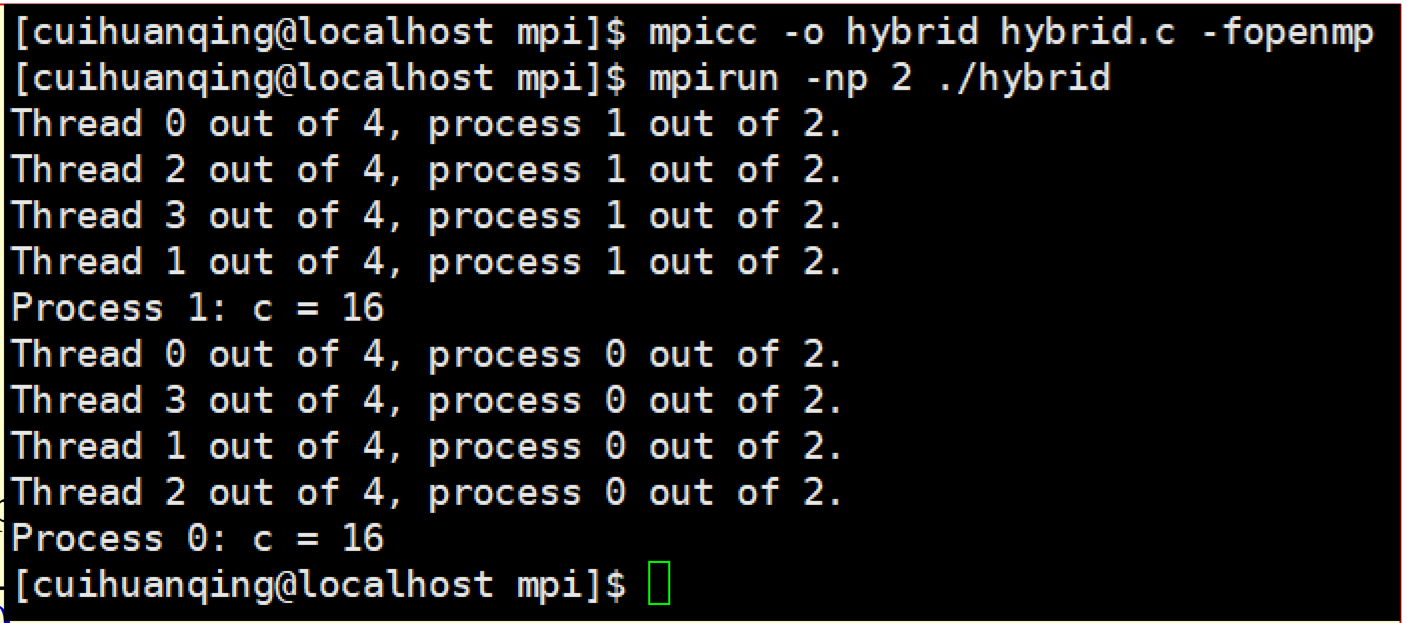


A Common Execution Scenario：

- A single MPI process is launched on each SMP node in the cluster;
- Each process spawns N threads on each SMP node;
- At some global sync point, the master thread on MPI process 0 communicates with the master thread on all other nodes;
- The threads belonging to each process continue until another sync point or completion;

3 MPI+OpenMP混合编程简介

```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4
int main(int argc, char *argv[]) {
  int numprocs,my_rank,c, iam, np;
  omp_set_num_threads(_NUM_THREADS);
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nump
  MPI_Comm_rank(MPI_COMM_WORLD,&my_
  #pragma omp parallel reduction(+:c) private(iam)
  {
      np = omp_get_num_threads();
      iam = omp_get_thread_num();
      c = omp_get_num_threads();
      printf("Thread %d out of %d, process %d out of %d.\n", iam, np, my_rank, numprocs);
  }
  printf("Process %d: c = %d\n",my_rank, c);
  MPI_Finalize();
  return 0;
}
```

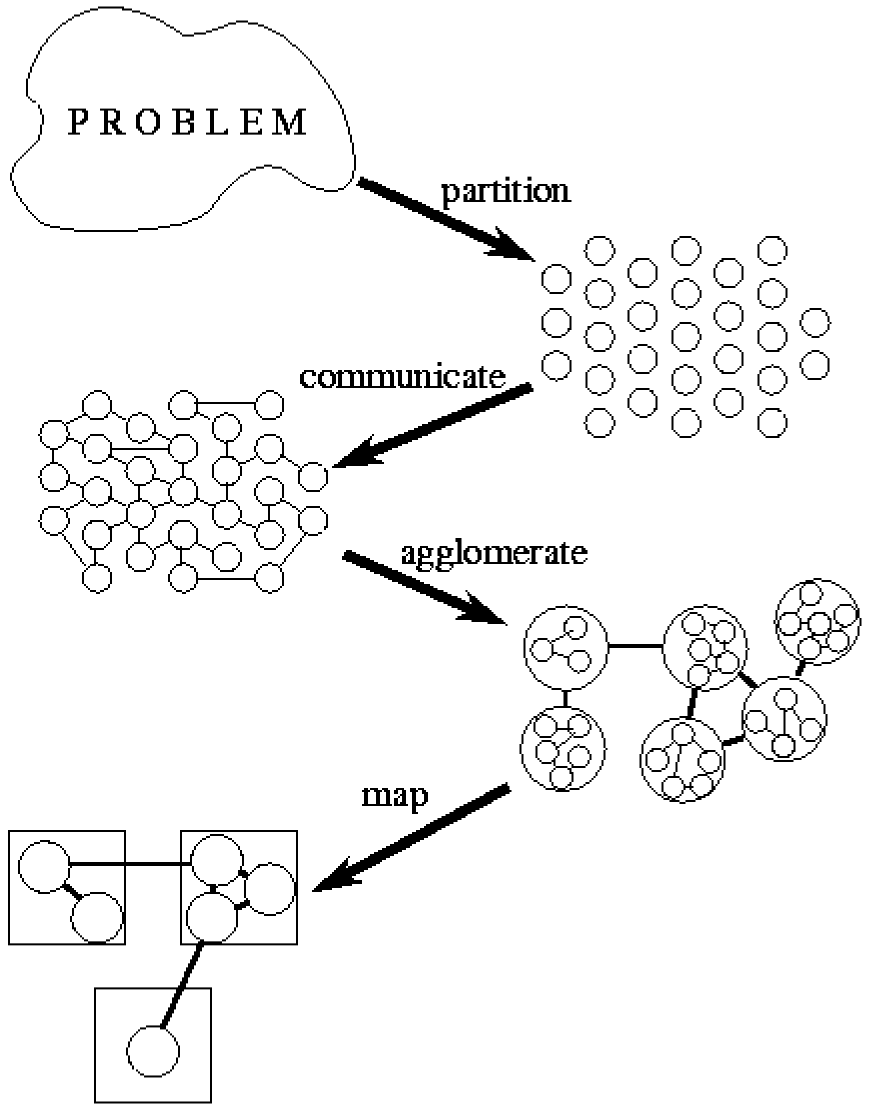


```
[cuihuanqing@localhost mpi]$ mpicc -o hybrid hybrid.c -fopenmp
[cuihuanqing@localhost mpi]$ mpirun -np 2 ./hybrid
Thread 0 out of 4, process 1 out of 2.
Thread 2 out of 4, process 1 out of 2.
Thread 3 out of 4, process 1 out of 2.
Thread 1 out of 4, process 1 out of 2.
Process 1: c = 16
Thread 0 out of 4, process 0 out of 2.
Thread 3 out of 4, process 0 out of 2.
Thread 1 out of 4, process 0 out of 2.
Thread 2 out of 4, process 0 out of 2.
Process 0: c = 16
[cuihuanqing@localhost mpi]$
```

4 Design of Parallel Programs

■Methodical Design

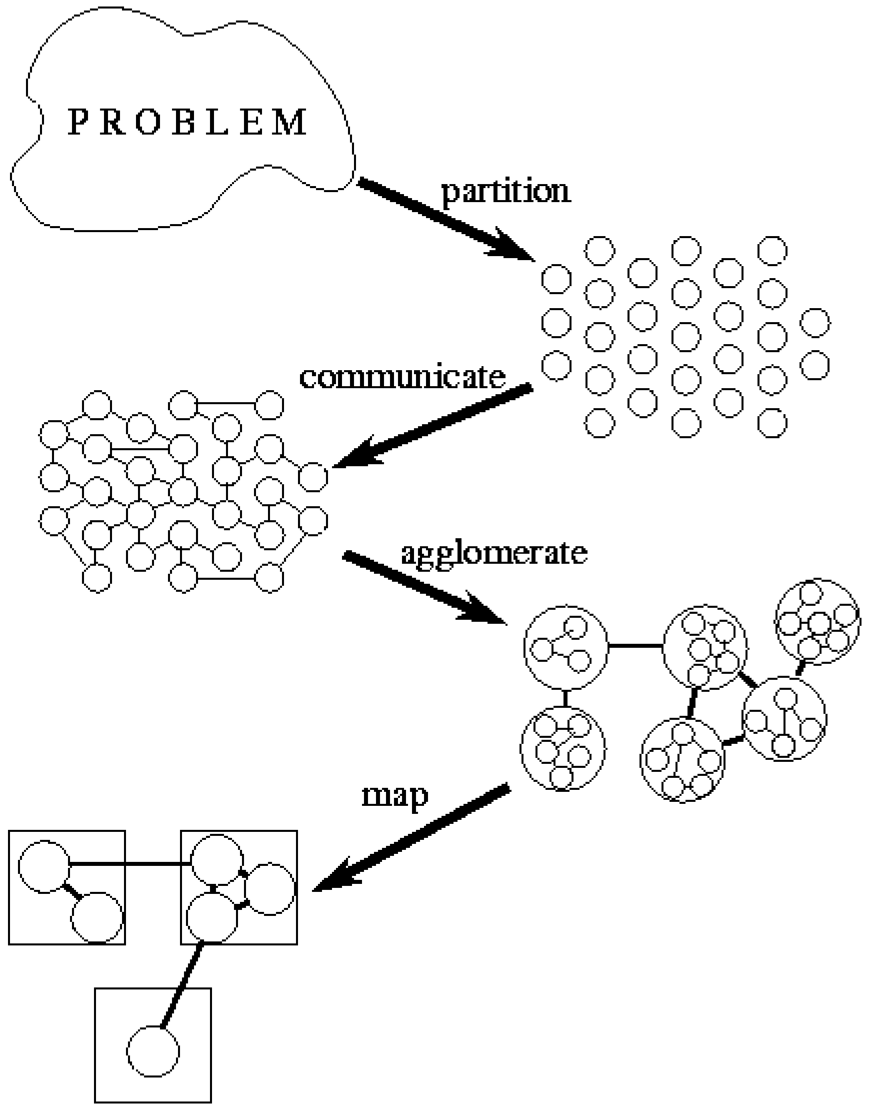
1.Partitioning. The *computation* that is to be performed and the *data* operated on by this computation are *decomposed* into small tasks. *Practical issues* such as the number of processors in the target computer are *ignored*, and *attention* is focused on recognizing *opportunities for parallel execution*.



4 Design of Parallel Programs

■Methodical Design

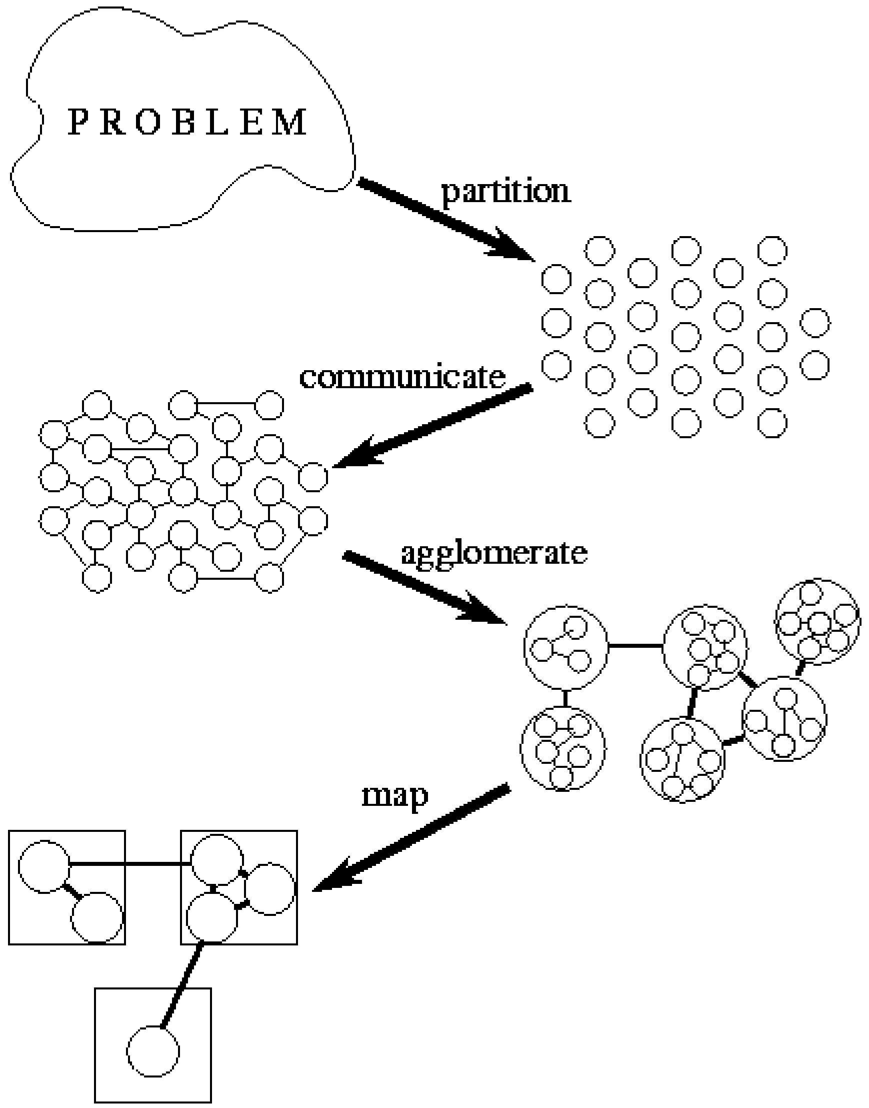
2.Communication. The *communication* required to coordinate task execution is determined, and *appropriate communication* structures and algorithms are defined.



4 Design of Parallel Programs

■Methodical Design

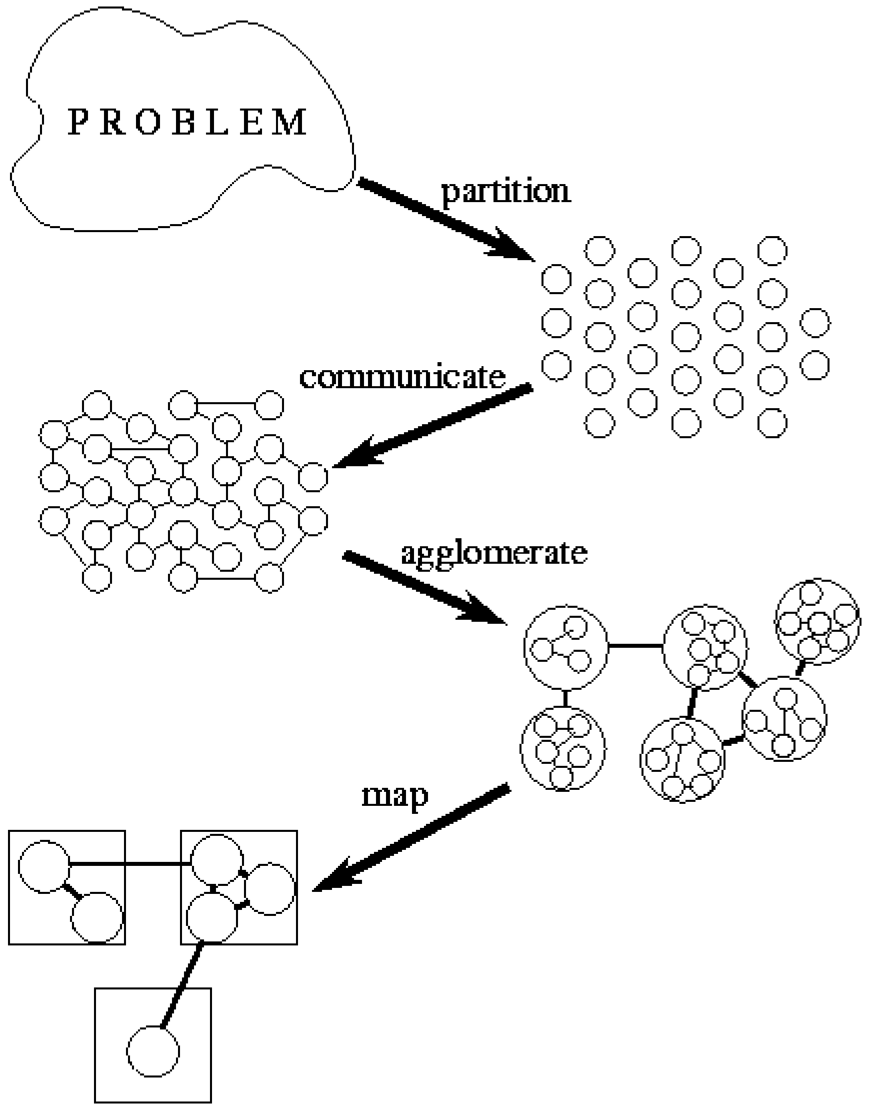
3.Agglomeration. The task and communication structures defined in the first two stages of a design are evaluated with respect to *performance requirements* and *implementation costs*. If necessary, *tasks are combined* into larger tasks to improve performance or to reduce development costs.



4 Design of Parallel Programs

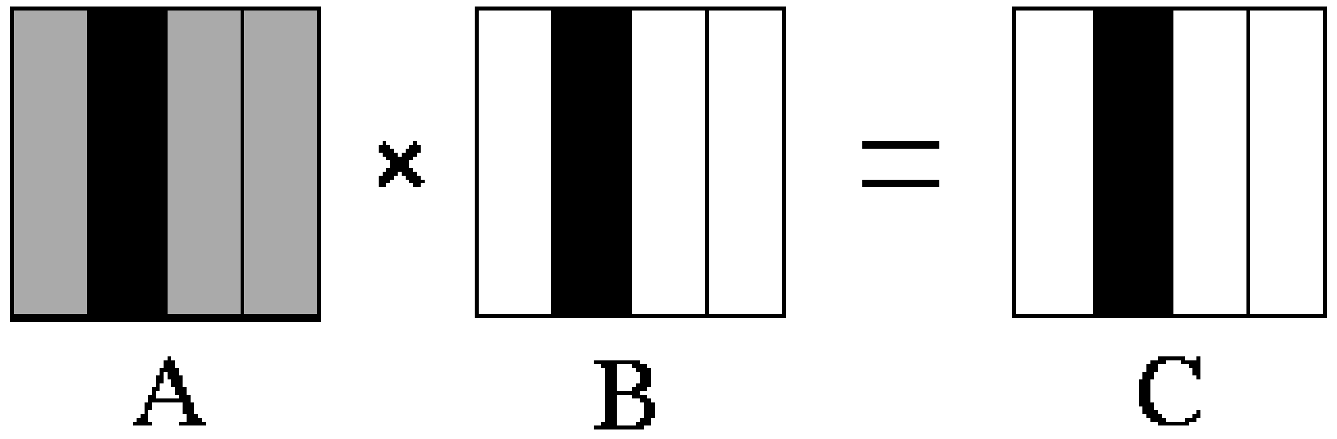
■Methodical Design

4.Mapping. *Each task* is *assigned* to a *processor* in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. *Mapping* can be specified *statically* or *determined at runtime* by load-balancing algorithms.



4 Design of Parallel Programs

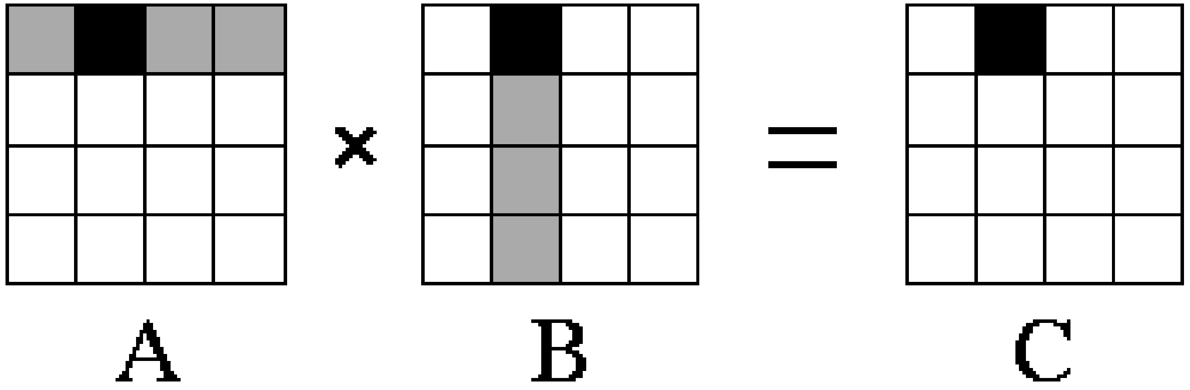
■Example: Matrix multiplication



Each process requires all of matrix A, and has several columns of matrix B, to compute the corresponding columns of matrix C. Finally, a root process collects all results in one matrix.

4 Design of Parallel Programs

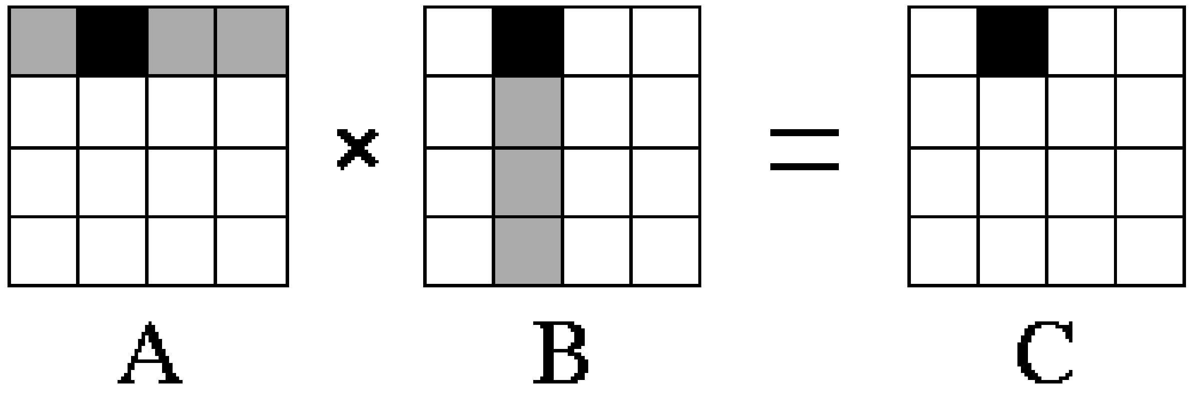
■Example: Matrix multiplication



Each task requires several rows of matrix A, and several columns of matrix B, to compute the corresponding sub-matrix of C.

4 Design of Parallel Programs

■Example: Matrix multiplication

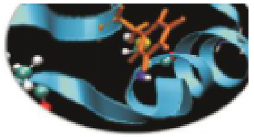


Each task requires several rows of matrix A, and several columns of matrix B, to compute the corresponding sub-matrix of C.

5 作业、项目和成绩评定

5.1 作业（每个同学独立完成程序和实验报告）

E2 – example – Pi by quadrature



It is known that the mathematical constant π can be approximated by computing the following formula:

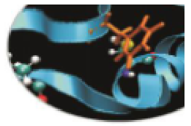
$$\pi = 4\int_0^1 \frac{1}{1+x*x}\,dx$$

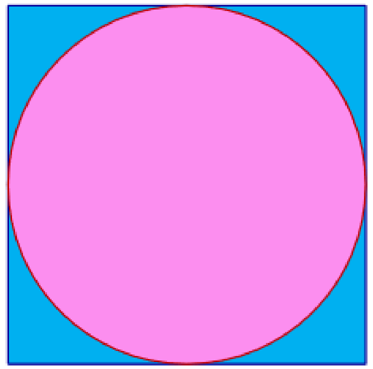
The value of the above integral can be approximated by numerical integration, i.e. by computing the mean value of the function f(x) = $\frac{1}{1+x*x}$ in a number of points and multiplying per the x range. This can be easily done in parallel by dividing the [0,1] range into a number of intervals.

5 作业、项目和成绩评定

5.1 作业（每个同学独立完成程序和实验报告）

E3 – exercise – Montecarlo Pi





The value of the constant π can be approximated also by recalling that, given A_c the area of the circle and A_s the area of the circumscribing square:

$$\pi = 4\frac{A_c}{A_s}$$

Thus π could be approximated in a MonteCarlo style by counting the number of the random points in a square that are contained in the inscribed circle.

5 作业、项目和成绩评定

5.1 作业（每个同学独立完成程序和实验报告）

（1）使用MPI、OpenMP、MPI+OpenMP编写上述两种求解PI的并行程序。

（2）使用VTune等工具对程序进行瓶颈分析和优化。

（3）提交程序源代码、变量和语句的详细说明。

（4）在实验报告中通过图表说明CPU串行程序和三种并行程序在各种规模的运行时间。

（5）（选做）在实验报告中通过图表说明三种并行程序使用不同的数据分配方法在各种规模的运行时间。

5 作业、项目和成绩评定

5.2 项目（按照小组完成）

题目1：用OpenMP、MPI、MPI+OpenMP设计一个KNN分类算法（K近邻算法）程序。

说明：在特征空间中查找K个最相似或者距离最近的样本，然后根据K个最相似的样本对未知样本进行分类。通过训练集和测试集给出算法的正确率。

5 作业、项目和成绩评定

5.2 项目（按照小组完成）

题目2：编写一个矩阵乘法的OpenMP、MPI、MPI+OpenMP并行程序，并且与对应规模的串行程序进行运行时间的比对（n=10，100，200，500，1000，1500，2000……），矩阵A（n，n）、矩阵B（n，n）、C = A x B

要求：

1.完成程序的开发并验证其正确性，完成一个实验报告（程序源代码、变量和语句的详细说明；

2.在实验报告中通过图表说明CPU串行程序和三种并行程序在各种规模的运行时间；

3（选做）.在实验报告中通过图表说明三种并行程序使用不同的数据分配方法在各种规模的运行时间。

5 作业、项目和成绩评定

5.3 成绩和要求

要求：

1、要求每个学生必须独立完成作业要求的内容（及格）

2、要求每个团队至少正确完成一个项目（良好）

3、正确完成2个项目（优秀）

成绩组成：

工作态度：10%

作品质量和掌握程度：30%

实训报告质量：20%

沟通汇报能力：10%

独立分析与解决问题能力：20%

团队与合作：10%

参考资料

- [美]帕切克　著，邓倩妮　等译，并行程序设计导论，机械工业出版社，2013-1
- [德]贝蒂尔·施密特（Bertil Schmidt）乔治·冈萨雷斯-多明格，并行程序设计：概念与实践，机械工业出版社，2020-06-25
- 雷洪，胡许冰 著，多核并行高性能计算 OpenMP，冶金工业出版社，2016-5
- 张武生，薛巍，李建江，郑纬民编著，MPI并行程序设计实例教程，清华大学出版社，2009年