2. MPI简介、安装和点对点通信

罗翔宇
中国人民大学统计与大数据研究院

此课件内容主要基于Blaise Barney的网络资料Message Passing Interface (MPI)及Wes Kendall的MPI Tutorial

为什么要学习MPI

- 并行计算广泛存在于日常生活中，包括个人电脑、手机或者其他科技。无论你是为了学习并行计算课程、或为了工作、或仅仅是因为它有趣而学习它，你选择了一项能够使你在接下来的几年中无比有价值的技能。通过学习MPI，你踏上了一个正确的轨道去学习并行计算。尽管MPI是比一些并行计算库(比如Hadoop, Spark)更为低阶，但是MPI能够为你并行计算的知识打下坚实的基础。

Figure credit: https://computing.llnl.gov/tutorials/mpi/

为什么要学习MPI

- MPI 是Message Passing Interface(消息传递接口)的英文简称
- MPI 是在HPC上编写消息传递程序的“业界标准”
- MPI主要处理消息传递的并行编程模型
  - 通过通信操作将数据从一个处理器转移到另一个处理器
- MPI的目的是提供一个广泛可用的编写消息传递程序的标准
  - 具有实用性、便携性、高效性、灵活性
- MPI适用于分布式内存结构、分布式-共享内存混合结构

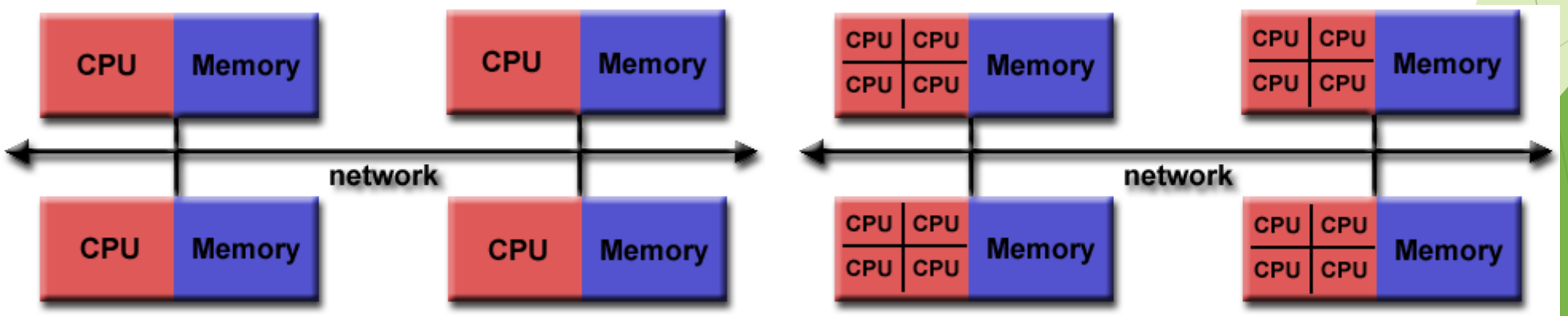


Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI的特点

- 规范化：唯一能被看成标准的消息传递库；在所有HPC平台上支持
- 便携性：将你的应用移植到另一支持MPI标准的平台(比如操作系统)时，几乎不需要修改源代码
- 功能性：超过430个常规操作，但大多数的MPI程序少于十几个常规操作即可实现

MPI的历史

- 在1990年前，程序员没有我们这样幸运。为不同计算机架构写并行应用是一个困难而且繁琐的工作。在那段日子，许多库可以用来建造并行应用，但是没有一个标准且可接受的方法。现在，在科学和研究领域中大量的需要运用并行计算。在并行计算库中最普遍的模型即是消息传递模型。什么是消息传递模型呢？它是指一种可以在不同进程中传递消息以完成某任务的应用。这种模型在实际中表现得非常好。比如，主进程将工作任务传递给从进程；在每个进程上将部分数据进行排序，再将得到的局部结果融合起来。几乎任何的并行应用都能够用消息传递模型表示。
- 由于MPI库大同小异，在1992年的Supercomputing会议上，这些库的作者们定义了一个标准用于消息传递的接口---消息传递接口（MPI）。这个标准的接口能够允许程序员写可以移植在所有主流并行构架上的并行程序，并且具有很多他们熟悉的特征和模型。

MPI的历史

- MPI的演化历史

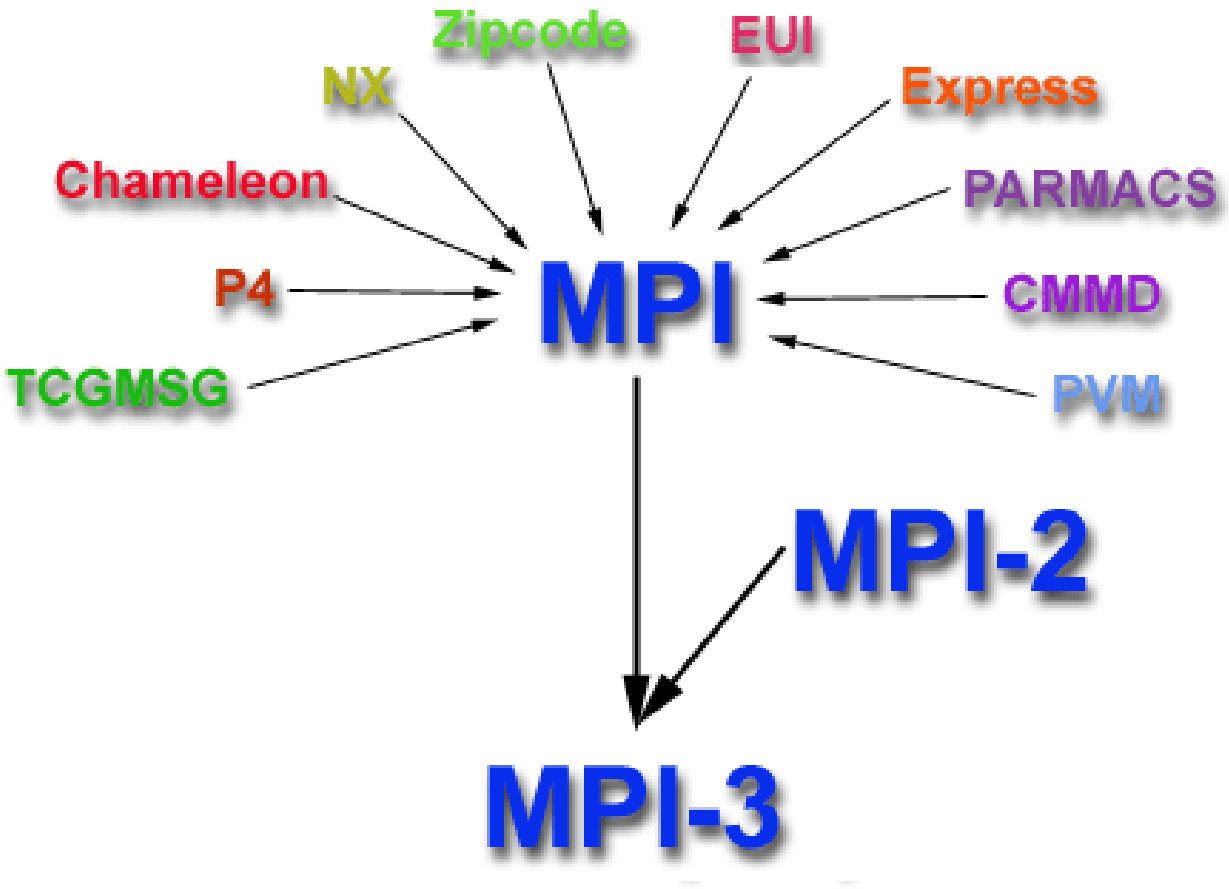


Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI标准的执行库和编译器

MPI Library	Where?	Compilers
MVAPICH	Linux clusters	GNU, Intel, PGI, Clang
Open MPI	Linux clusters	GNU, Intel, PGI, Clang
Intel MPI	Linux clusters	Intel, GNU
IBM BG/Q MPI	BG/Q clusters	IBM, GNU
IBM Spectrum MPI	Coral Early Access and Sierra clusters	IBM, GNU, PGI, Clang

Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI标准的执行库和编译器

MPI Build Scripts - Linux Clusters			
Implementation	**Language**	**Script Name**	**Underlying Compiler**
MVAPCH2	**C**	`mpicc`	C compiler for loaded compiler package
	C++	`mpicxx` `mpic++`	C++ compiler for loaded compiler package
	Fortran	`mpif77`	Fortran77 compiler for loaded compiler package. Points to mpifort.
		`mpif90`	Fortran90 compiler for loaded compiler package. Points to mpifort.
		`mpifort`	Fortran 77/90 compiler for loaded compiler package.
Open MPI	**C**	`mpicc`	C compiler for loaded compiler package

	C++	`mpiCC`
		`mpic++`
		`mpicxx`
			C++ compiler for loaded compiler package

	Fortran	`mpif77`	Fortran77 compiler for loaded compiler package. Points to mpifort.
		`mpif90`	Fortran90 compiler for loaded compiler package. Points to mpifort.
		`mpifort`	Fortran 77/90 compiler for loaded compiler package.

Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI的基本概念

- 通信器(communicator)定义了一组能够相互通信的进程。在这一组进程中，每个进程有一个唯一的编号(rank)，它们通过这个rank来进行交流

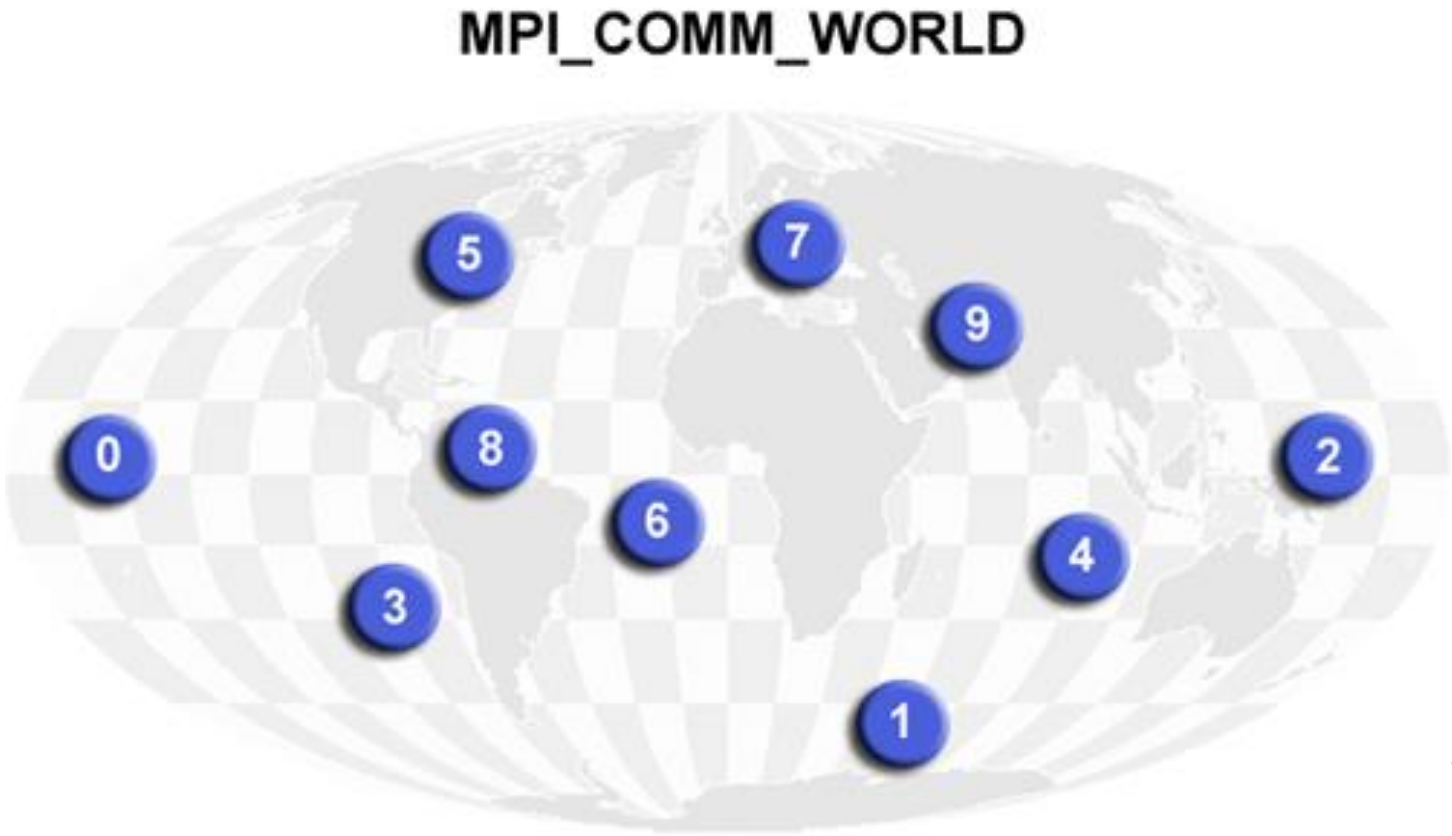


Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI的基本概念

- 交流的基础是在进程中发送和接收信息。一个进程通过提供接收信息的进程rank以及对信息的标识(tag)来发送信息。接收信息的进程通过这个tag以及消息来源的rank来接收信息，之后利用所接收数据来进行操作。像这样具有一个发送进程(sender)和一个接收进程(receiver)的交流叫做点对点交流(point-to-point communications)
- 还有其他例子，主进程将消息通过广播的形式发送给所有的从进程。在这样的条件下，写若干个发送、接收很浪费时间。实际上，MPI可以进行很多聚合通信(collective communications). 通过将点对点通信和聚合通信的混合可以构建很复杂的并行程序。

Figure credit: https://computing.llnl.gov/tutorials/mpi/

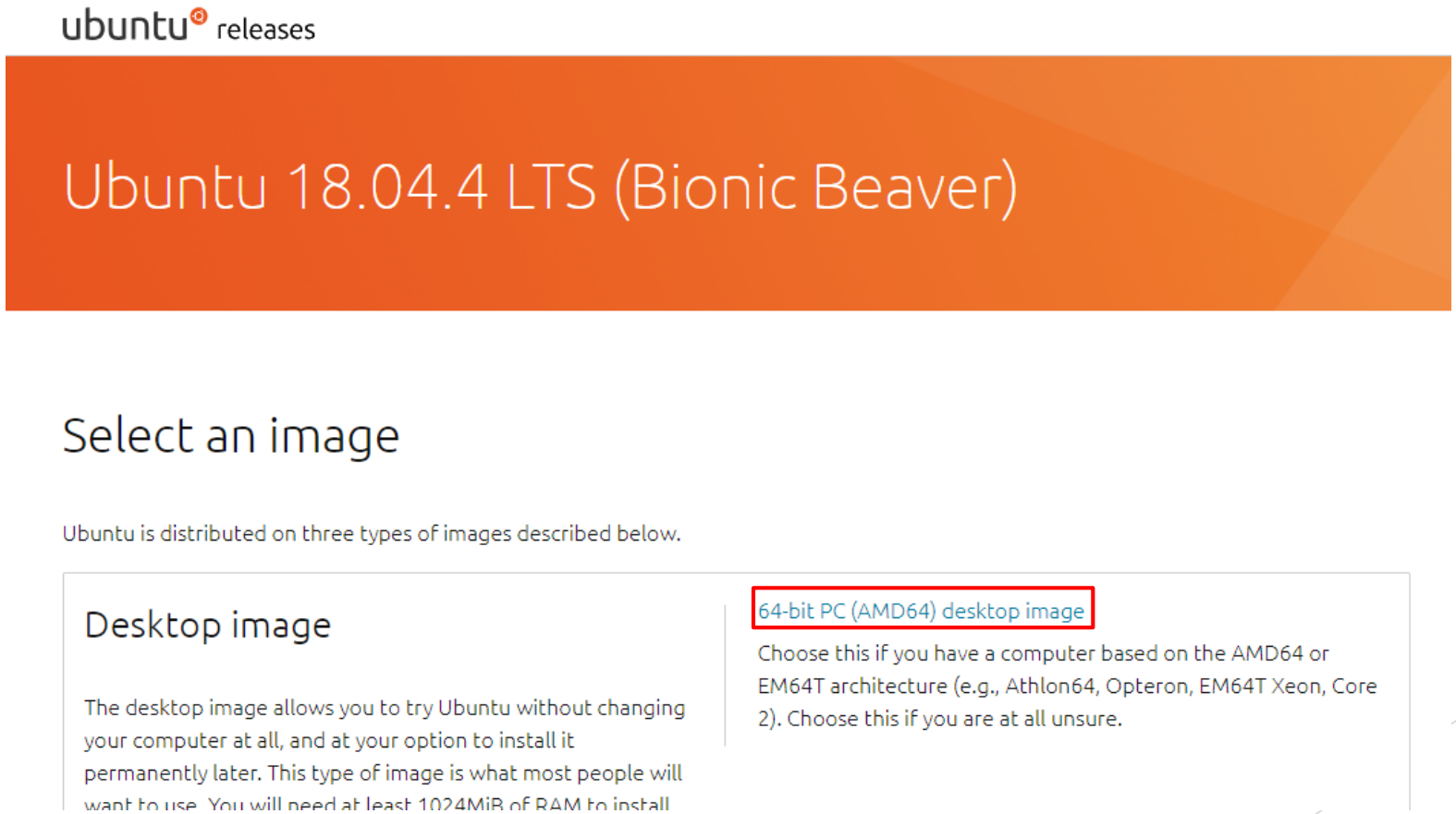
下载虚拟盒子、Ubuntu操作系统

- 此页仅对于Windows用户
- 下载VirtualBox: https://www.virtualbox.org/wiki/Downloads

下载虚拟盒子、Ubuntu操作系统

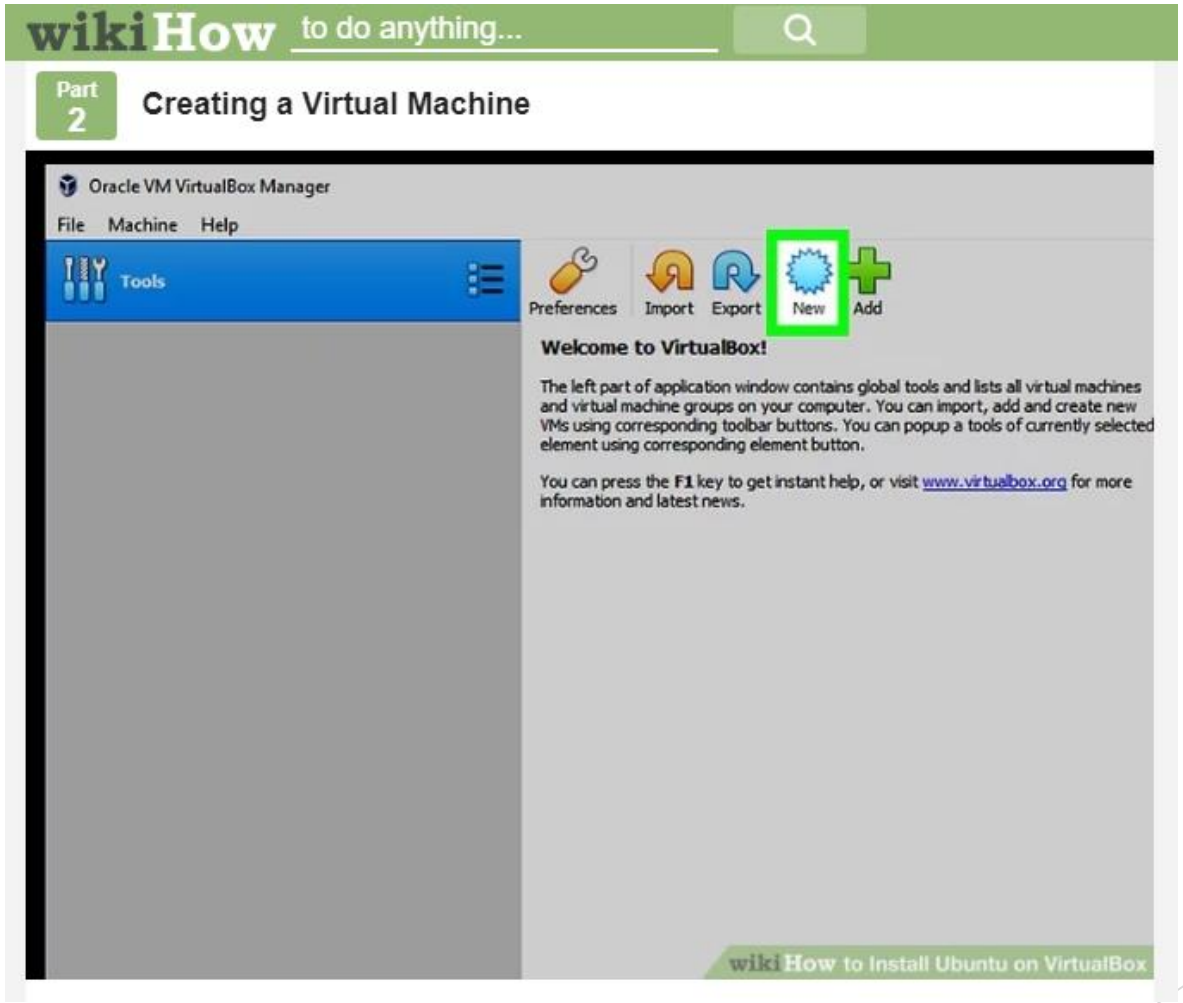
- 此页仅对于Windows用户
- 下载操作系统Ubuntu 18.04.3 LTS

  http://releases.ubuntu.com/bionic/



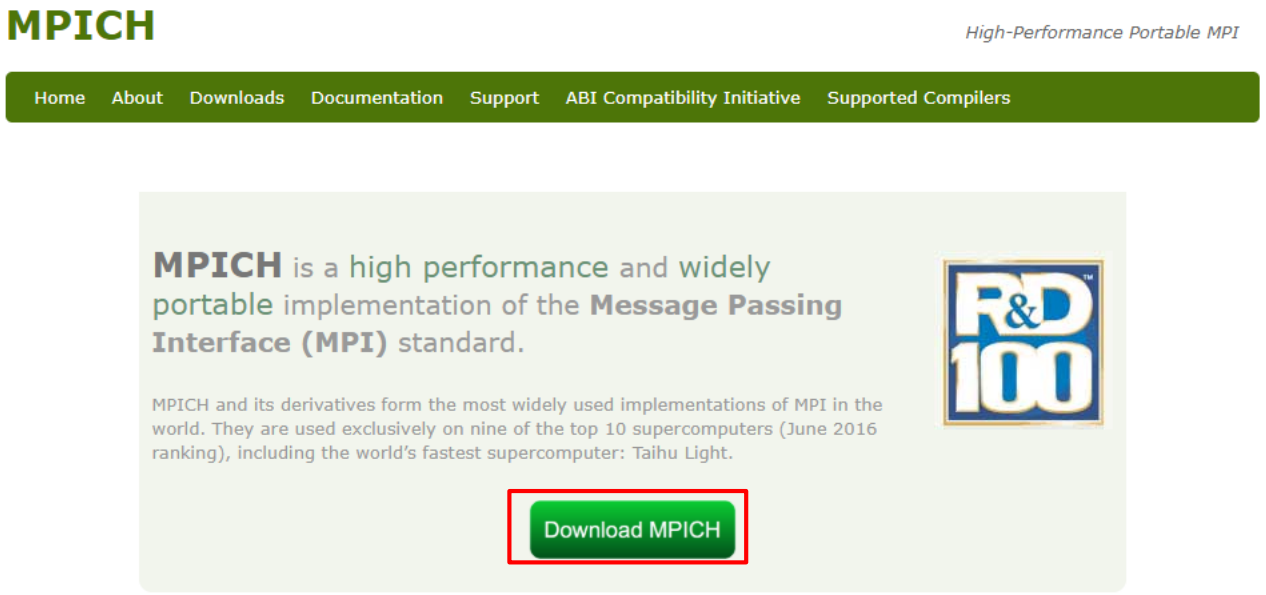
在虚拟盒子上安装Ubuntu操作系统

- 此页仅对于Windows用户
- Install Ubuntu on VirtualBox: https://www.wikihow.com/Install-Ubuntu-on-VirtualBox



MPI的安装

- Install MPICH2 on a single computing machine
- MPI是仅仅一个标准，它却有很多的实现。其中一个很流行的实现叫做MPICH2。MPICH2是一个广泛运用的MPI实现，是由美国Argonne National Laboratory开发，其他流行的实现包括OpenMPI。截止2016年6月，MPICH被用在全球十台超级计算机中的九台，其中包括当时世界上最快的超级计算机：太湖之光。
- MPICH2：http://www.mpich.org/ (Windows用户进入虚拟机后下载)



MPI的安装

```
xiangyu@xiangyu-VirtualBox:~$ ls
breeze.linalg._  Downloads          parallel_computing_files  Templates
Desktop          examples.desktop   Pictures                  tmp.txt
Documents        Music              Public                    Videos
xiangyu@xiangyu-VirtualBox:~$ cd Downloads
xiangyu@xiangyu-VirtualBox:~/Downloads$ ls
jdk-13_linux-x64_bin.tar.gz  scala-2.13.1.tgz
mpich-3.3.1                  spark-2.4.4-bin-hadoop2.7.tgz
mpich-3.3.1.tar.gz           spark-core_2.10-1.3.0.jar
mpich-3.3.2.tar.gz
```

MPI的安装

- 在终端中，解压源文件、进入解压后的文件夹

```
xiangyu@xiangyu-VirtualBox:~/Downloads$ tar -xzf mpich-3.3.2.tar.gz
xiangyu@xiangyu-VirtualBox:~/Downloads$ ls
jdk-13_linux-x64_bin.tar.gz  mpich-3.3.2          spark-2.4.4-bin-hadoop2.7.tgz
mpich-3.3.1                  mpich-3.3.2.tar.gz   spark-core_2.10-1.3.0.jar
mpich-3.3.1.tar.gz           scala-2.13.1.tgz
xiangyu@xiangyu-VirtualBox:~/Downloads$ cd mpich-3.3.2
xiangyu@xiangyu-VirtualBox:~/Downloads/mpich-3.3.2$
```

- 在tar命令中
  - -x表示解压
  - -z表示解压文件带有gzip属性的
  - -f 表示指定要操作的文件名，必须为最后一个参数，其后接解压文件名

MPI的安装

```
xiangyu@xiangyu-VirtualBox:~/Downloads/mpich-3.3.2$ ls
aclocal.m4      configure      examples      mpich.def          src
autogen.sh      configure.ac   lib           mpich-doxygen.in   subsys_include.m4
CHANGES         contrib        maint         mpi.def            test
confdb          CONTRIBUTING   Makefile.am   README             www
config.log      COPYRIGHT      Makefile.in   README.envvar
config.system   doc            man           RELEASE_NOTES
xiangyu@xiangyu-VirtualBox:~/Downloads/mpich-3.3.2$ ./configure --disable-fortr
an
```

- ./configure or ./configure --disable-fortran (对安装进行配置)
- 由于我们不需要用到fortran，因此选用后一个命令

MPI的安装

```
xiangyu@xiangyu-VirtualBox:~/Downloads/mpich-3.3.2$ make;sudo make install
if test ! -h ./src/include/mpio.h ; then \
    rm -f ./src/include/mpio.h ; \
    ( cd ./src/include &&       \
        ln -s ../mpi/romio/include/mpio.h ) ; \
fi
make  all-recursive
```

```
xiangyu@xiangyu-VirtualBox:~/Downloads/mpich-3.3.2$ mpiexec -version
HYDRA build details:
    Version:                                 3.3.2
    Release Date:                            Tue Nov 12 21:23:16 CST 2019
    CC:                              gcc
    CXX:                             g++
    F77:
    F90:
    Configure options:                       '--disable-option-checking' '--pre
```

- make; sudo make install (对MPICH2进行安装)
- mpiexec –version (安装成功看看所用版本)

MPI Hello World!

阅读代码mpi_hello_world.c

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                        processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

MPI Hello World!

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

- 建立一个MPI程序的第一点是需要包含MPI的头文件

MPI Hello World!

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

- 通过MPI_Init(int* argc, char*** argv)对MPI环境进行初始化

MPI Hello World!

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

- MPI_Comm_size(MPI_Comm communicator, int* size) 返回通讯器的大小。MPI_COMM_WORLD包含所有进程

MPI Hello World!

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

- MPI_Comm_rank(MPI_Comm communicator, int* rank) 返回通讯器里每个进程的号码。这个号码用来识别每个进程，号码取非负整数，从0开始

MPI Hello World!

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

- MPI_Get_processor_name(char *name, int* name_length) 能够得到进程的名字

MPI Hello World!

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

- MPI_Finalize() 用于清理MPI环境

MPI Hello World!

- MPI中各个函数的介绍可查阅：
- https://docs.microsoft.com/en-us/message-passing-interface/mpi-functions
- 在单台机器上编译及执行

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o mpi_hello_world
 mpi_hello_world.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 3 ./mpi_hello
_world
Hello world from processor xiangyu-VirtualBox, rank 0 out of 3 processors
Hello world from processor xiangyu-VirtualBox, rank 1 out of 3 processors
Hello world from processor xiangyu-VirtualBox, rank 2 out of 3 processors
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 3 ./mpi_hello
_world
Hello world from processor xiangyu-VirtualBox, rank 2 out of 3 processors
Hello world from processor xiangyu-VirtualBox, rank 0 out of 3 processors
Hello world from processor xiangyu-VirtualBox, rank 1 out of 3 processors
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$
```

- 在集群上执行，mpirun -np 5 -hosts node-1-6,node-1-7 ./mpi_hello_world

MPI环境管理的常规命令

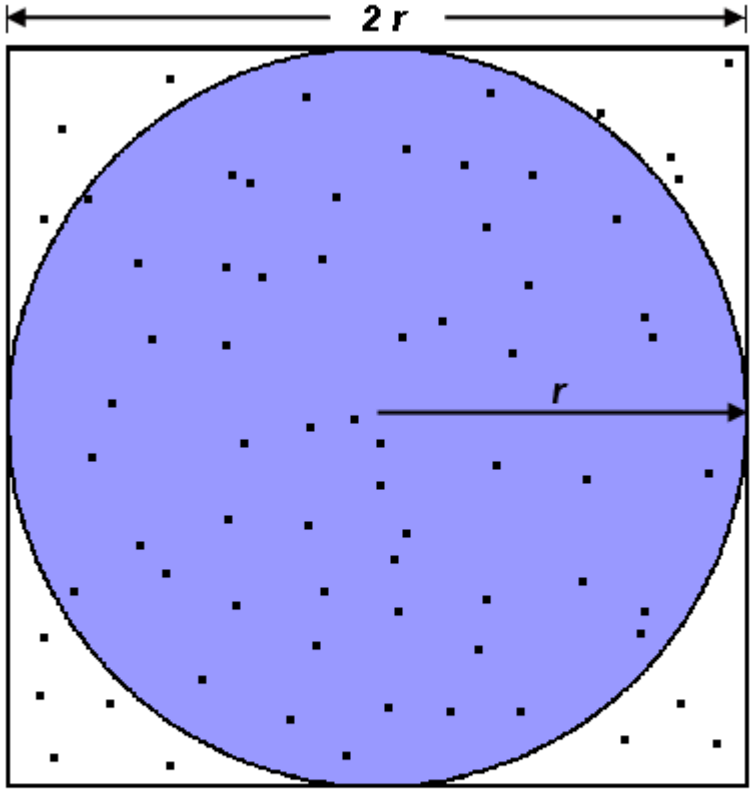
- int MPI_Init (int* argc, char*** argv) 初始化MPI执行环境
  - 每一个MPI程序必须调用
  - 在所有其他MPI函数之前调用
  - 在每个MPI程序中仅调用一次
  - MPI_Init可以将命令行中的参数传递给所有进程
- int MPI_Comm_size (MPI_Comm comm, int* size) 返回指定通讯器中MPI进程的总数
  - 若通讯器指定为MPI_COMM_WORLD，它代表程序中所有可利用的MPI进程数量
- int MPI_Comm_rank (MPI_Comm comm, int* rank)返回指定通讯器中MPI进程的rank
- int MPI_Abort (MPI_Comm comm, int errorcode)终止指定通讯器的所有MPI进程
- int MPI_Get_processor_name (char* name, int* resultlength) 返回进程的名字和其名字的长度
  - 变量name的缓冲器长度最多是MPI_MAX_PROCESSOR_NAME

MPI环境管理的常规命令

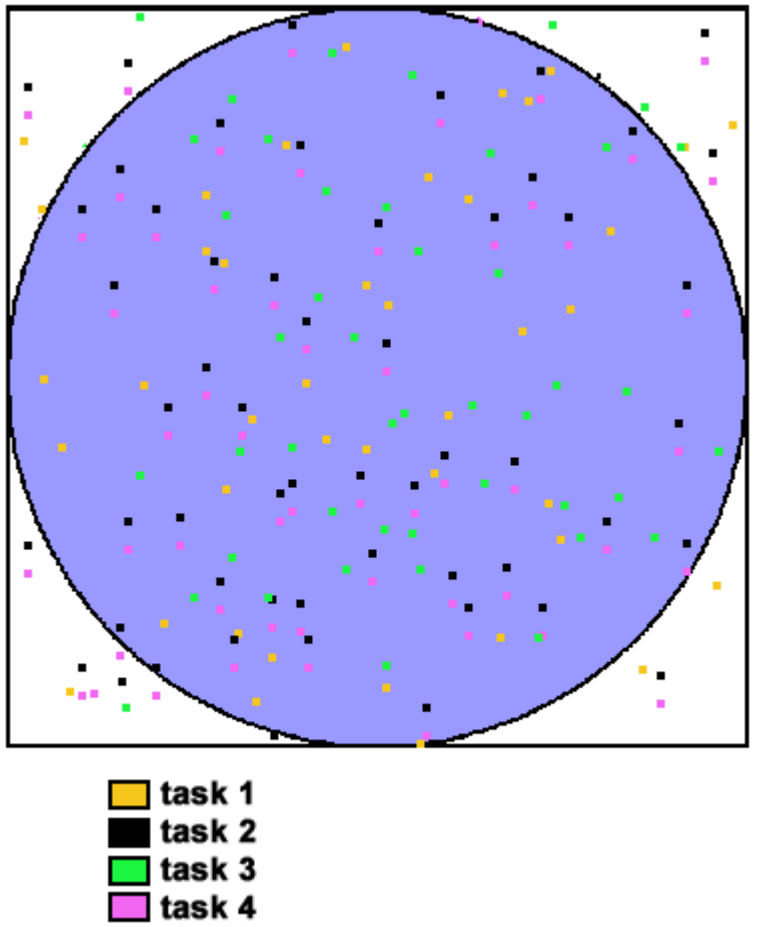
- int MPI_Get_version (int* version, int* subversion) 返回当前所用MPI标准的版本和子版本
- int MPI_Initialized (int* flag) 表明是否调用了MPI_Init函数，返回真(1)或假(0)
- double MPI_Wtime (void) 返回某进程的系统时间(单位为秒，双精度)
- int MPI_Finalize (void) 终止MPI执行环境
  - 每一个MPI程序中最后调用的常规MPI函数

MPI的点对点通信：计算圆周率

- 一个embarrassingly parallel的问题
- 最后一步需要用到Send, Rev接受结果

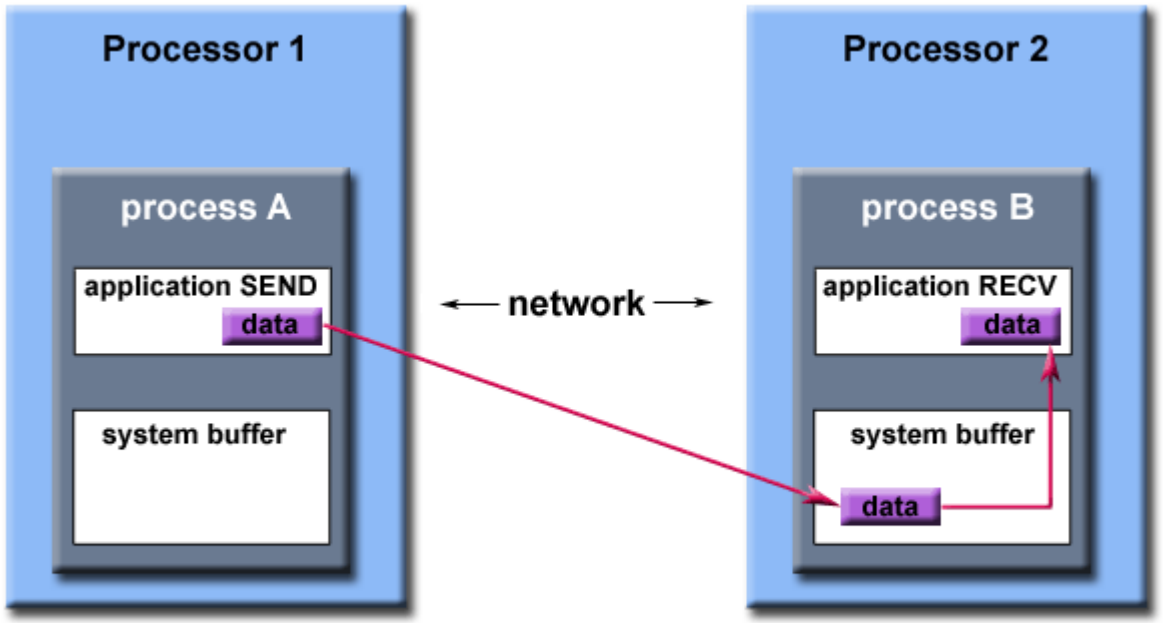


$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4 \times \frac{A_C}{A_S}$$



MPI的点对点通信：缓冲(Buffering)

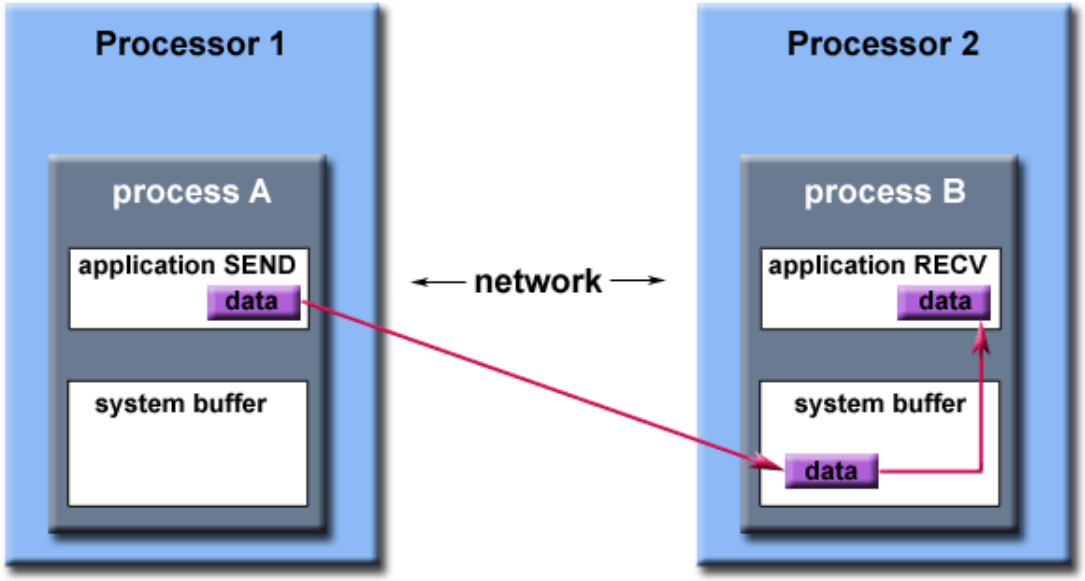
- 在最完美的情形下，传送(send)和接收(receive)是完全同步的。在实际生活中，传送和接收通常是不同步的：
  - 传送发生在接收前5秒，在这个时间内，消息在哪里呢？
- 系统缓冲(system buffer)区域用来在传输过程中保留数据
  - 应用缓冲(application buffer)指用户管理的地址空间(比如程序变量)



Path of a message buffered at the receiving process

MPI的点对点通信：阻塞和非阻塞

- 大多数MPI点对点通信可以用阻塞或非阻塞模式
- 对于**阻塞通信**（MPI_Send, MPI_Recv）
  - 进程1：当可以**安全更改应用缓冲**的时候（消息已到达系统缓冲），阻塞性传送函数才会被返回。“安全”指修改不会影响将被接收的数据，但“安全”并不意味着数据已经被接收
  - 进程2：当**数据到达且可以被程序利用**后，阻塞性接收函数才会被返回

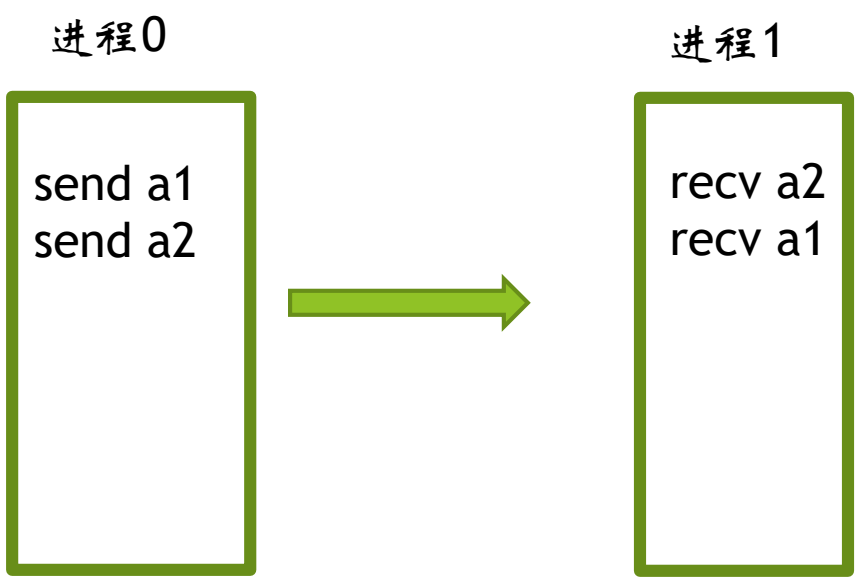


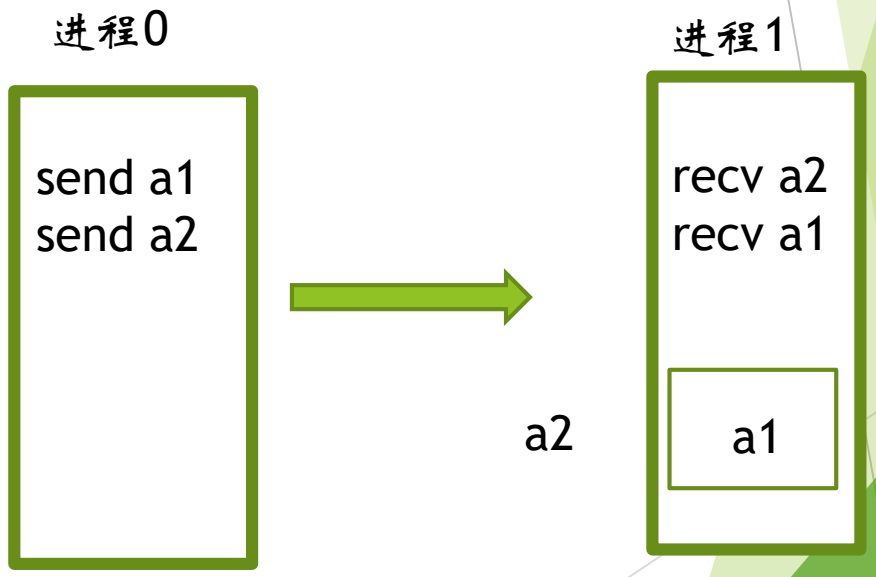
Path of a message buffered at the receiving process

Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI的点对点通信的特点

- 次序：MPI保证消息的传递是按代码顺序的
  - 对于以下阻塞性通信，进程1是先接受第二个信息吗？（是的）
  - 但是，若a1和a2的容量太大，大于了进程1系统缓冲，导致阻塞通信“send a2”一直不能返回，且进程1中recv a2也不能返回，将会出现“死锁”现象。





  - 假如接收进程有两个接收命令，它们都用于接收相同消息，第一个接收命令将会率先接收

MPI的点对点通信：阻塞和非阻塞

- 大多数MPI点对点通信可以用阻塞或非阻塞模式
- 对于**非阻塞通信**（MPI_Isend, MPI_Irecv）
  - 非阻塞性传输和接收函数将会立即返回
  - 主要用于有交流的计算和实现可能的性能增益
    - 若不小心，会导致一些问题

Blocking Send	Non-blocking Send

myvar = 0;
 	
for (i=1; i<ntasks; i++) {
task = i;
MPI_Send (&myvar ... ... task ...);
myvar = myvar + 2
 	
/* do some work */
 	
}
	myvar = 0;
	
	for (i=1; i<ntasks; i++) {
	task = i;
	MPI_Isend (&myvar ... ... task ...);
	myvar = myvar + 2;
	
	/* do some work */
	
	MPI_Wait (...);
	}

Safe. Why?	Unsafe. Why?

Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI的阻塞性点对点通信

- MPI Send and Receive
- 发送和接收是MPI中最基本的概念。几乎MPI中的每个函数都可以通过发送和接收这两个函数来执行。在本次课程中我们将介绍如何运用MPI的阻塞性发送和接收函数。
- MPI的发送和接收操作通过以下方式执行。首先，进程A确定一个需要发送给进程B的消息。接着，进程A将所有相关的数据打包放置在缓存里。缓存就如同一个信封，因为数据在传输前打包成一个消息，这就像信件在拿到邮局前需要放在信封里。将数据放在缓存里后，通讯设备（通常是网络）负责将消息传递到合适的位置。消息的位置即是通过进程的rank来确定。
- 即便消息传递到了进程B，进程B仍然必须声明它想要接受从进程A来的数据。一旦有了这个接受声明，数据就可以成功传递。当进程A得知数据被进程B接收后，它就可以做其他的事情了。
- 有时进程A可能给进程B多种信息。MPI允许两个进程给消息一个ID，我们称为标签(tag)。当进程B仅仅接收某个标签的消息时，其他的消息将仍处于缓存中直到进程B为它们“开门”。

MPI的点对点通信

- int MPI_Send(void* buf, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)
- int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)
- 当阅读这些参数时，你会觉得很多，但是你会越来越觉得记住这些参数很容易，因为几乎每个MPI声明都会利用相似的语法。第一个参数是数据缓存。第二个和第三个参数分别表示数据的个数和缓存中的数据类型。MPI_Send发送精确的数据量，MPI_Recv最多接收这么多的数据量。第四个和第五个参数分别代表接收进程或发送进程的秩以及消息的标签。第六个参数明确所处的通讯器，MPI_Recv的最后一个参数表明接收消息的状态。

MPI的点对点通信

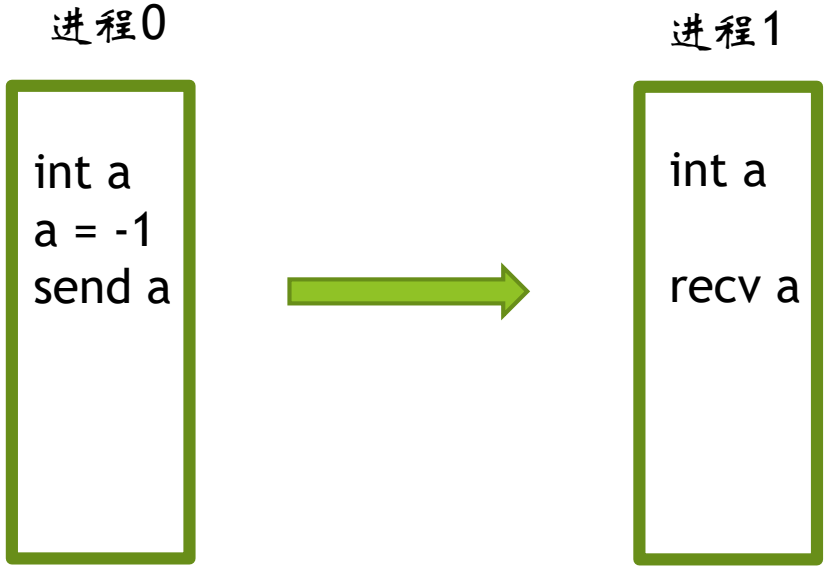
- 传输消息的时候需要明确传输数据的类型，如果传输一个整数的话，那么它的数据类型是MPI_INT。所有的数据类型如下图所示
- 你也可以自己创建能够刻画更为复杂的MPI数据类型

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

Figure credit: https://mpitutorial.com/

MPI的例子

- MPI 传送/接收程序：进程0将数字-1传输给进程1



MPI的例子

- MPI 传送/接收程序:进程0将数字-1传输给进程1

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if(world_size < 2){
                printf("The number of arguments is %d\n", argc);
                printf("The seconde argument is %s\n", argv[1]);
                fprintf(stderr, "World size must be greater than 1 for %s\n",\
                        argv[0]);

                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        int number;

        if(world_rank == 0){
                number = -1;
                MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        }else if(world_rank == 1){
                MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, \
                        MPI_STATUS_IGNORE);
               printf("Process 1 received number %d from process 0\n",\
                        number);
        }

        MPI_Finalize();
}
```

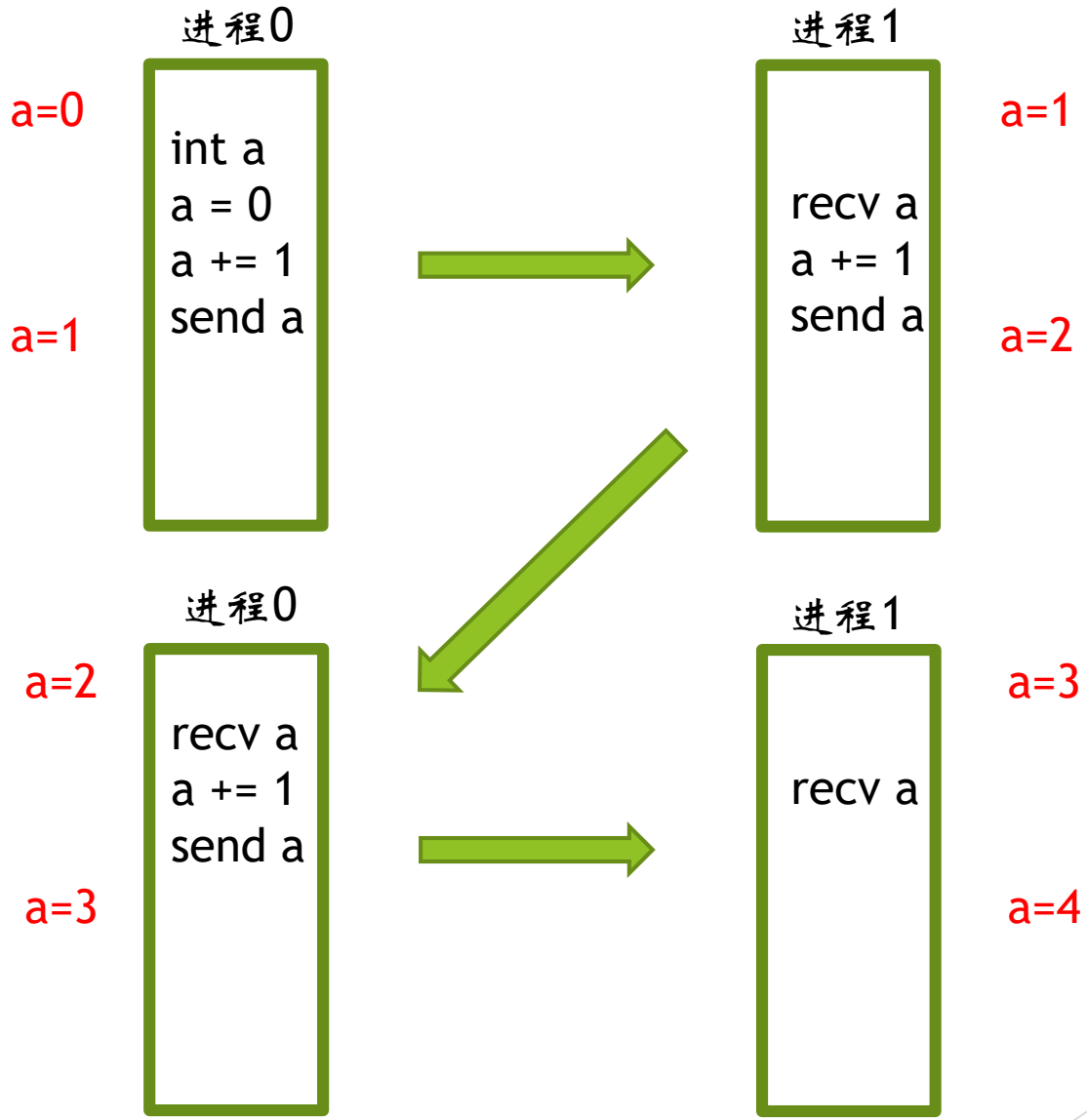
MPI的例子

- mpicc编译后的多个执行结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 1 ./send_recv
The number of arguments is 1
The seconde argument is (null)
World size must be greater than 1 for ./send_recv
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 1 ./send_recv
 test
The number of arguments is 2
The seconde argument is test
World size must be greater than 1 for ./send_recv
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 2 ./send_recv
Process 1 received number -1 from process 0
```

MPI的例子

- MPI 乒乓程序:进程0将“乒乓”(数字0加一)传给进程1，进程1接收到“乒乓”后，自加1再传给进程0,重复此过程，直到乒乓到达10.



MPI乒乓代码

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
        const int PING_PONG_LIMIT = 10;
        MPI_Init(NULL, NULL);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if(world_size != 2){
                fprintf(stderr, "World size must be two for %s\n", argv[0]);
                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        int ping_pong_count = 0;
        int partner_rank = (world_rank + 1) % 2;
        while(ping_pong_count < PING_PONG_LIMIT){
                if(world_rank == ping_pong_count % 2){
                        ping_pong_count++;
                        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0\
                                , MPI_COMM_WORLD);
                        printf("%d sent and incremented ping_pong_count %d to %
d\n", world_rank, ping_pong_count, \
                                partner_rank);
                }else{
                        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank,\
                                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        printf("%d received ping_pong_count %d from %d\n",\
                                world_rank, ping_pong_count, partner_rank);
                }
        }

        MPI_Finalize();
}
```

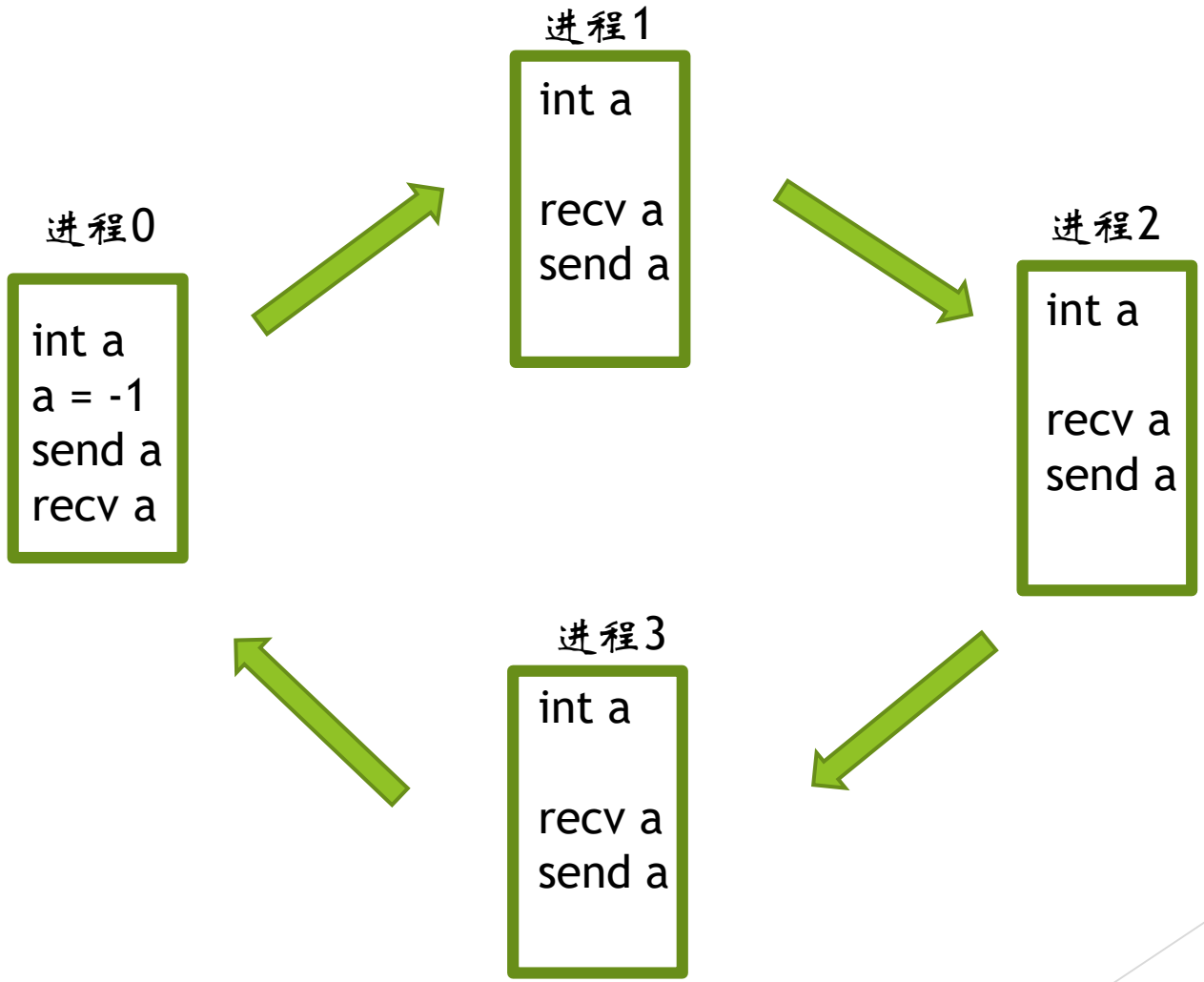
MPI乒乓代码输出结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o ping_pong ping_
pong.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 3 ./ping_pong
World size must be two for ./ping_pong
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 2 ./ping_pong

0 sent and incremented ping_pong_count 1 to 1
1 received ping_pong_count 1 from 0
1 sent and incremented ping_pong_count 2 to 0
0 received ping_pong_count 2 from 1
0 sent and incremented ping_pong_count 3 to 1
1 received ping_pong_count 3 from 0
1 sent and incremented ping_pong_count 4 to 0
0 received ping_pong_count 4 from 1
0 sent and incremented ping_pong_count 5 to 1
1 received ping_pong_count 5 from 0
1 sent and incremented ping_pong_count 6 to 0
0 received ping_pong_count 6 from 1
0 sent and incremented ping_pong_count 7 to 1
1 received ping_pong_count 7 from 0
1 sent and incremented ping_pong_count 8 to 0
0 received ping_pong_count 8 from 1
0 sent and incremented ping_pong_count 9 to 1
1 received ping_pong_count 9 from 0
1 sent and incremented ping_pong_count 10 to 0
0 received ping_pong_count 10 from 1
```

MPI的例子

- 指环程序
  - 指环程序在进程0中初始化一个值，然后这个值被依次传递到另外的进程中。当此值从最后一个进程传递到进程0的时候，整个程序便终止了。



指环程序代码

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        int token, prev, next;
        prev = world_rank - 1;
        next = world_rank + 1;

        if(world_rank == 0){
                prev = world_size - 1;
        }

        if(world_rank == world_size - 1){
                next = 0;
        }
```

指环程序代码

```
        if(world_rank == world_size - 1){
                next = 0;
        }

        if(world_rank == 0){
                token = -1;
                MPI_Send(&token, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
                printf("Process %d sent token %d to process %d\n",
                        world_rank, token, next);
                MPI_Recv(&token, 1, MPI_INT, prev, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                printf("Process %d received token %d from process %d\n",
                        world_rank, token, prev);
        }else{
                MPI_Recv(&token, 1, MPI_INT, prev, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                printf("Process %d received token %d from process %d\n",
                        world_rank, token, prev);
                MPI_Send(&token, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
                printf("Process %d sent token %d to process %d\n",
                        world_rank, token, next);
        }

        MPI_Finalize();

}
```

指环程序代码结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ vi ring.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o ring ring.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 6 ./ring
Process 0 sent token -1 to process 1
Process 1 received token -1 from process 0
Process 1 sent token -1 to process 2
Process 2 received token -1 from process 1
Process 2 sent token -1 to process 3
Process 3 received token -1 from process 2
Process 3 sent token -1 to process 4
Process 4 received token -1 from process 3
Process 4 sent token -1 to process 5
Process 5 received token -1 from process 4
Process 5 sent token -1 to process 0
Process 0 received token -1 from process 5
```

注：例子的C代码文件可以从网页找到https://mpitutorial.com/tutorials/，但为了深刻理解，建议最好自己写一遍。