Linux编程基础

陈俊清

jqchen@math.tsinghua.edu.cn

清华大学数学科学系

March 8, 2013

Part I

Linux编程基础

Outline

1. 程序的编译和运行
2. 程序调试
3. Makefile

Linux操作的一点补充

- 进入shell窗口：应用程序→系统工具→终端（或将其添加到收藏夹，然后从收藏夹内直接打开）
- 切换到超级用户：su

```
[jqchen@localhost ~]$ su -
口令:
[root@localhost ~]# pwd
/root

[jqchen@localhost ~]$ su
口令:
[root@localhost jqchen]# pwd
/home/jqchen
```

有些linux操作需要在超级用户下才能进行。

压缩归档文件

归档工具tar，压缩工具gzip,bzip2：

- 压缩归档文件：

```
$ tar czvf Tech.tar.gz Tech/
Tech/
Tech/tubiao.txt
Tech/mount.tec
$ tar cjvf Tech.tar.bz2 Tech/
Tech/
Tech/tubiao.txt
Tech/mount.tec
```

  生成压缩文件名：xxxx.tar.gz(xxxx.tgz)或是xxxx.tar.bz2
- 解压缩：tar zxvf xxxx.tar.gz 参数说明：
  - z表示gzip格式（bz2要用j）
  - x表示展开档案
  - v打出操作时的信息
  - f表示对紧随其后的文件操作。

应用软件管理(Fedora)

软件包管理： rpm (Redhat Package Manager)

```
$ su
$ Password: *********
$ rpm -ivh ****.rpm
```

i表示安装软件包（也可用U表示升级软件包）， v表示打印安装信息，h要求安装过程显示安装进度。

- rpm -e 软件包名称: 删除安装
- rpm -q 软件包名称: 查询软件包的版本信息
- rpm -qa ： 显示安装的所有软件包
- rpm -qf 文件名： 显示文件所属软件包

yum：通过网络安装（或升级）指定的软件包

```
$ yum install
$ yum update
$ yum search
```

进程管理 ps， kill

查看用户进程:

```
[jqchen@localhost Course]$ ps -u jqchen
       ...
 4921 pts/2    00:00:04 vim
 5488 ?        00:00:00 firefox
```

杀掉进程

```
[jqchen@localhost Course]$ kill -9 5488
       ...
4921 pts/2    00:00:04 vim
       ...
```

kill -9 表示强制杀掉，一般情况可以用 kill -TERM 来杀死进程。

远程登录

- Linux:SSH

```
$ ssh IP -l username
```

- Windows:
  - SSHSecureShellClient
  - X-Server: Exceed, Xmanager

文件传输

- Linux上传：scp filename jqchen@hpc.math.tsinghua.edu.cn:~/
  scp -r dirname jqchen@hpc.math.tsinghua.edu.cn:~/
- Windows: SSHSecureShellClient

Outline

1. 程序的编译和运行
2. 程序调试
3. Makefile

常用的编译器

- C语言的编译器为gcc(GNU C),cc等;
- Fortran编译器：gfortran,fc,g77(GNU Fortran),f90(Fortran 90),f95等
- 可用man命令查看手册：man gcc, man g77等等来查看编译选项。
- 命令行形式:
  - gcc [options] files [options]
  - gfortran [options] files [options]

程序的编译

文件的类型由文件的扩展名决定：

- C源代码：.c
- Fortran 77源代码：.f
- Fortran 90源代码：.f90
- C++源代码:.c++,.C,.cpp,.cc,.cxx;
- 目标文件：.o
- 库文件：.a,.so

动态库（共享库）与静态库的区别：静态库在程序编译的时候会链接到目标代码中，程序运行时不在需要该库， 而动态库不会链接到目标代码中，在程序运行的时候才载入该库。

编译选项

- -c 只编译，不链接，即只生成.o文件;
- -o filename:指定输出文件，缺省为 *.o,a.out等。
- -I 目录名 用来指定（增加）头文件的搜索路径
- -L 目录名 用来指定（增加）库文件的搜索路径
- -l库名 用来指定库文件,加载的真实文件为：lib库名.a 或lib库名.so， 比如常用的要链接数学函数库，用-lm参数，加载库为：libm.so(or libm.a)
- -g 该选项使得调试用的关于源程序的信息写入到目标文件和可执行文件中
- -On 该选项指定编译的优化级别，-O0,-O1,-O2,-O3
- -W 控制编译的警告信息，常有的有-Wall

例

- gcc -O2 -o prog file1.c file2.c files3.o file4.a
- gcc -c file.c
  gcc -o out file.o
- gcc -c -I/usr/local/mpi/include file.c
  gcc -o prog -L/usr/local/mpi/lib file.o -lmpi
  (等价于： gcc -o prog file.o /usr/local/mpi/lib/libmpi.a)

例子

Example: hello.c

```
#include<stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello, the world!\n");
    return 0;
}
```

对上述例子只要运行

```
[jqchen@localhost code]$ gcc -o hello hello.c
[jqchen@localhost code]$ ls
hello  hello.c
[jqchen@localhost code]$ ./hello
Hello, the world!
```

Outline

1. 程序的编译和运行
2. 程序调试
3. Makefile

程序调试

GDB

- 可让被调试的程序在你所指定的断点处停住
- 当程序停住时，可以检查此时你的程序中所发生的事情
- 需注意的是，程序编译时要加上选项 -g，它使得编译得到的可执行文件包含源文件信息

```
[jqchen@localhost code]$ gcc -g -o hello hello.c
[jqchen@localhost code]$ gdb hello
```

可以比较两次生成的可执行文件的大小，看看有什么不同。

GDB的使用

- 设置断点（break）： b 目标;
  目标可以是程序中的函数或是某一行等
- 开始运行（run）： r
- 单步运行（step）： s
- 本子程序中单步运行一行（next）： n
- 继续运行（continuous）： c
- 查看当前程序调用堆栈（backtrace）： bt
- 打印表达式的值（print）： p

程序调试

在程序调试的时候，常常会遇到因非法访问内存而导致程序崩溃的问题，比如内存没有分配就使用，越界访问， 内存分配而没有释放导致泄露等等。Linux下提供了不少针对内存检查调试的工具，valgrind 是著名的之一。

valgrind

假如要检查的可执行文件为 a.out,检查内存使用是否合法只要 用下述命令即可

```
valgrind --tool=memcheck ./a.out
```

Example: test.c

```
/* test.c, Example for GDB and Valgrind */
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
int main(int argc, char *argv[])
{
    int i;
    double sum=0.;
    double *result;
    result=(double*)malloc(21*sizeof(*result));
    for(i=0;i<=20;i++){
        sum+=cos(i);
        result[i]=1.0;
    }
    printf("Result = %f,%f\n",sum,result[21]);
    free(result);
    return 1;
}
```

Outline

1 程序的编译和运行

2 程序调试

3 Makefile

Makefile

- 命令形式:
  make [-f Makefile] [options] [target [target ...]]
  其中-f 选项给出定义规则的文件名（简称Makefile文件）,缺省情况使用当前目录下的GNUmakefile,makefile或Makefile， target给出要生成的目标，缺省时只生成Makefile中定义的第一个目标。
- 通过Makefile文件定义一组文件直接的依赖关系及处理命令，方便程序开发过程中的编译与维护。
- 处理规则的建立以特定的文件扩展名及文件修改时间为基础，缺省支持常用的文件扩展名：.c, .o, .f,.F,.a,.h等， 用户可以通过.SUFFIXES:目标 定义新的文件扩展名。

Makefile

Makefile的工作原理

- make会在当前目录下找名字叫"makefile"或"Makefile"的文件。
- 如果找到，它会找文中的相应目标。如果make没有指定，按默认处理
- 如果目标不存在，或是目标所依赖的文件的修改时间要比目标文件新，则执行后面所定义的命令。

Makefile

make的执行步骤

- 读入所有的makefile
- 读入被include的其他makefile
- 初始化文件中的变量
- 推导隐式规则，并分析所有规则
- 为所有文件建立依赖关系链
- 根据依赖关系决定哪些目标要生成
- 执行生成命令。

Makefile

- 基本规则：

  目标文件：依赖对象
  <tab> 处理命令
  <tab> ... ...

例：

```
prog: file1.c file2.c file3.c
      gcc -O2 -o prog file1.c file2.c file3.c
```

其含义是：如果目标（prog）不存在，或则任何一个依赖对象（file1.c,file2.c,file3.c）比目标文件新，则执行指定的命令.

处理命令前要以Tab键开始，而不能是空格键。

Makefile

- 宏定义:

```
SRC=file1.c file2.c file2.c
prog:$(SRC)
	gcc -O2 -o prog $(SRC)
```

  环境变量可以在Makefile中作为宏使用，如$(HOME).
- 常用预定义的宏:
  - $@:代表目标名（上例中为prog）
  - $ < :第一个依赖对象名（上例中为file1.c）
  - $ ^ :全部依赖对象名（上例中为file1.c,file2.c,file3.c）
  - $? :全部比目标新的依赖对象名
  - $* :用在隐式规则中，代表不含扩展名的依赖对象

Makefile

- 隐式规则:

```
prog: file1.o file2.o file3.o
      gcc -O2 -o prog $?
.c.o:
      gcc -O2 -c $*.c
```

这里的 $*.c 表示 file1.c file2.c,file3.c,而.c.o定义了从.c文件生成.o文件的隐式规则。

Makefile

每个Makefile中都应该写一个清空目标文件（.o和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。 一般的格式是:

```
clean:
    rm $(objects)
```

更为稳健的做法是:

```
.PHONY : clean
clean :
    -rm $(objects)
```

.PHONY意思表示clean是一个“伪目标”，而且总满足更新条件。 而在rm命令前面加了一个减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。 当然，clean的规则不要放在文件的开头，不然，这就会变成make的默认目标，相信谁也不愿意这样。 不成文的规矩是——“clean从来都是放在文件的最后”。

Makefile

- 引用其它的Makefile
  在Makefile中使用include关键字可以把别的Makefile 包含进来。include的语法是:
  include filename;
- Makefile中的注释: #...
- Makefile中 用 反斜杠（\）表示 换行。

Makefile

更多隐式规则

- 隐式规则可看作一种惯例，make会按照这种惯例运行，那怕我们的Makefile中没有书写这样的规则。例如，把[.c]文件编译成[.o]文件这一规则，你根本就不用写出来，make会自动推导出这种规则，并生成我们需要的[.o]文件。

  hello:hello.o

  gcc -o hello hello.o

- make 还有更多其它功能，如果想深入了解可以查阅在线文档或其他书籍

一个例子 I

```
cc = gcc
INCLUDE = -I/usr/local/hypre/include
LIBS = -L/usr/local/hypre/lib
lib = -lhypre

all: myexe
main.o: main.c hypre.h
        $(cc) $(INCLUDE) -c main.c
pro1.o: pro1.c
        $(cc) $(INCLUDE) -c pro1.c
myexe: main.o pro1.o
        $(cc) -o myexe main.o pro1.o $(LIBS) $(lib)
clean:
        rm -f main.o pro1.o myexe
.PHONY: clean
```

Part II

MPI基础知识

Outline

4. 消息传递编程平台MPI

5. MPI基础知识
   - MPI的几个基本函数
   - MPI程序的基本结构

6. 作业

消息传递编程平台MPI

MPI是什么?
MPI(Message Passing Interface,消息传递接口)：消息传递是广泛应用于多种并行计算机，尤其是分布式内存并行计算机的一种编程规范。消息传递的种类很多，MPI就是其中一种消息传递规范。MPI为程序员提供一个并行程序库，程序员通过调用MPI的库程序来达到数据通信目的。MPI提供C，Fortran和C＋＋语言接口。

消息传递编程平台MPI

除各厂商提供的MPI系统外，一些高校、科研部门也在开发免费的通用MPI系统，其中最著名的有：

- MPICH(http://www.mcs.anl.gov/mpi/mpich)
- LAM MPI(http://www.lam-mpi.org/)

他们均提供源代码，并支持绝大多数并行计算机系统。MPI的第一个标准MPI 1.0于1994年推出。最新的标准为2.0版，于1998年推出。

单机环境下MPI的安装

假如你下载了软件包 mpich-1.2.6.tar.gz

```
tar xzvf mpich-1.2.6.tar.gz
cd mpich-1.2.6
./configure --prefix=/usr/local/mpich-1.2.6\
            -rsh=ssh
make
su
make install
```

MPI设置

创建文件：/etc/profile.d/mpich.sh(适用于Bash),其内容为

```
export PATH=${PATH}:/usr/local/mpi/bin
export MANPATH=${MANPATH}:/usr/local/mpi/man
```

也可以在家目录下的.bashrc中加入上面两行并运行：

```
. .bashrc
```

使用C shell的用户应该在文件 ~/.cshrc中加入：

```
setenv PATH ${PATH}:/usr/local/mpi/bin
setenv MANPATH ${MANPATH}:/usr/local/mpi/man
```

这个设置在用户重新登录后生效。这样设置后可以直接通过man来查看MPI函数文档， 也可以直接使用/usr/local/mpi/bin下的命令（mpicc,mpirun,mpif77等）.

设置 SSH服务

在目录/home/jqchen/.ssh下执行如下命令:

```
ssh-keygen -t dsa
cp id_dsa.pub authorized_keys
chmod go-rwx authorized_keys
ssh-agent $SHELL ssh-add
```

这样以后，使得运行并行程序的时候可以不用输入口令.
另外，如果你所安装的是MPICH2,在运行并行程序之前要先启动

```
mpd &
```

(目前最新版本的MPICH2-1.4.1p1可以直接通过（yum）网络安装，安装后也不需要单独设置ssh及启动mpd)

机群环境下安装MPICH

若干台计算机通过以太网连接，假设四台结点机的主机名和IP地址分别为 node1/10.0.0.1,node2/10.0.0.2,node3/10.0.0.3,node4/10.0.0.4.修改4台机器上的/etc/hosts文件如下：

```
127.0.0.1 localhost.localdomain localhost
10.0.0.1 node1.mydomian node1
10.0.0.2 node2.mydomian node2
10.0.0.3 node3.mydomian node3
10.0.0.4 node4.mydomian node4
```

- 设置ssh. 先在node1，按单机方式配置ssh，然后对其余机器设置ssh，并将node1上的.ssh目录覆盖node1， node2，node3，node4目录。
- 配置nfs，将MPICH安装目录和家目录网络共享
- 配置nis，使各结点间共享用户信息
- 编译安装mpich（最好是下载编译安装）

具体细节可以参看《并行计算导论》3.1.2节。

MPI程序的编译与运行

- MPI程序的编译：

```
[jqchen@localhost examples]$ mpicc -O2 -o mpiprog mpipro
```

- MPI程序的运行：

```
[jqchen@localhost examples]$ mpirun -np 4 mpiprog
```

MPI程序实例：/home/jqchen/mpich2-1.2.1p1/examples/cpi.c。
指定运行结点：-machinefile

```
mpirun -machinefile mfile -np 2 mpiprog
```

mfile中指定节点名称。

测试你的安装是否成功

在单机上编译运行简单测试程序:

```
[jqchen@localhost mpich2-1.2.1p1]$ cd examples/
[jqchen@localhost examples]$ mpicc -o cpi cpi.c
[jqchen@localhost examples]$ mpd &
[jqchen@localhost examples]$ mpirun -np 1 cpi
[jqchen@localhost examples]$ mpirun -np 2 cpi
```

如果你的ssh设置有误，在运行上述命令的时候会出现"permision denied"错误，或是要一次次地输入口令。

MPI程序和串行程序编译调试的区别

- MPI仅是一个"函数库"而已
- 串行编译：gcc， g++， g77， f95等

```
gcc -o execute source_file.c
```

并行编译环境：mpicc， mpif77， mpif90， mpicxx

```
mpicc -o execute source_file.c
```

- 运行串行程序：

```
./execute
```

运行并行程序：

```
mpirun -np N ./execute
```

- 程序调试：

```
gdb,valgrind ...
```

gdb调试MPI程序

在使用gdb之前，编译并行程序时一定要使用“-g”选项。

```
mpirun -np 2 ./myprog
```

然后查看 myprog的进程号，

```
ps -aux|grep myprog
```

找到相应的进程后，就可以分别对每一个进程进行调试。

```
gdb myprog process_number
```

注：有些版本的mpirun支持以下命令：

```
mpirun -gdb -np 2 ./myprog
```

但是这个方法只能调试主节点进程。
查找内存泄漏

```
mpirun -np 2 valgrind ./myprog
```

Outline

4 消息传递编程平台MPI

5 MPI基础知识
- MPI的几个基本函数
- MPI程序的基本结构

6 作业

下载MPI的文档

- MPI1.1:
  http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
- MPI2.0:
  http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

MPI2.0的文档中只有在MPI2.0版本上增加的内容，故应与老版本结合着看。

名词与概念

- **程序与代码**：我们这里所说的程序不是指以文件形式存在的源代码、可执行代码等，而是指为完成一个计算任务而进行的一次运行过程。
- **进程（process）**：一个MPI并行程序由一组运行在相同或不同计算机／计算节点上的进程或线程构成。为统一起见，我们将MPI程序中一个独立参与通信的个体称为一个进程。在Unix系统中，MPI的进程统称是一个Unix进程，在共享内存／消息传递混合编程模式中，一个MPI进程可能代表一组Unix线程。
- **进程组（process group）**：指一个MPI程序的全部进程集合的一个有序子集。进程组中每个进程被赋予一个在该组中唯一的序号（rank），用于在该组中标识该进程，序号的取值是0到进程数−1。

名词与概念

- **通信器（communicator）**：或称通信子，是完成进程间通信的基本环境，它描述了一组可以相互通信的进程以及他们之间的联接关系等信息。MPI的所有通信必须在模个通信器中进行。通信器分为域内通信器（intracommunicator）和域间通信器（intercommunicator）两类，前者用于属于同一进程组的进程间的通信，后者用于分属两个不同进程组间的通信。
  MPI系统在一个MPI程序运行时会自动创建两个通信器，一个称为MPI_COMM_WORLD，它包含该MPI程序中的所有进程，另一个称为MPI_COMM_SELF，它指单个进程自己所构成的通信器。
- **序号（rank）**：序号用来在一个进程组或通信器中标识一个进程。MPI中的进程由进程组／序号或通信器／序号所唯一确定。序号是相对于进程组或通信器而言的：同一个进程在不同的进程组或通信器中可以有不同的序号，进程的序号是在进程组或通信器被创建时赋予的。
  MPI 系统提供了一个特殊的进程号：MPI_PROC_NULL,表示一个空进程（不存在的进程），与之通信不起任何作用。

名词与概念

- **消息（message）**： MPI程序中在进程间传送的数据称为消息。一个消息由通信器、源地址、目的地址、消息标签和数据构成。
- **通信 (communication)**： 指在进程间进行消息的收发、同步等操作。

编程模式

- SPMD编程模式 Single Programm Multiple Data的缩写,指构成一个程序的所有进程运行的是同一份可执行代码。不同进程根据自己的序号可能执行该代码中的不同分支。这是MPI编程中最常用的编程方式，用户只需要编写维护一份源代码。
- MPMD编程模式 Multiple Program Multiple Data的缩写，指构成一个程序的不同进程运行不同的可执行代码。用户需要编写、维护多份源代码。
- 主／从编程模式（Master／Slave），是MPMD模式的一个特例，也是MPMD编程模式中最常见的方式。构成一个程序的进程之一负责所有进程件的协调及任务调度，该进程称为主进程（Master），其余的称为从进程（Slave），通常用户要维护两份源代码。

MPI函数的一般形式

- C:一般形式为:
  int MPI_Xxxxx(...)
  MPI的C函数名中下划线第一个字母大写，其余字母小写。
  除MPI_Wtime()和MPI_ Wtick（）外，所有的MPI的C函数均返回一个整型错误码，当它等于MPI_ SUCCESS(0)时表示调用成功，否则调用中产生错误。
- Fortran 77:一般形式为:
  SUBROUTINE MPI_ XXXXX(...,IERR)
  除MPI_ WTIME和MPI_ WTICK外，Fortan 77的MPI过程全部是Fortrann 77子程序，它们与MPI的C函数同名（但不区分大小写），参数表除了最后一位IERR和参数类型不同外，其余都一样。

MPI的原始数据类型

MPI系统中数据的发送与接收都是基于数据类型的，数据类型可以是MPI预定义的，称为原始数据类型，也可以是用户自己在原始数据类型的基础上自己定义的数据类型。

MPI	C	Fortran
MPI_INT	int	INTEGER
MPI_FLOAT	float	REAL
MPI_DOUBLE	double	DOUBLE PRECISION
MPI_CHAR	char	
MPI_CHARACTER		CHARACTER*1
MPI_COMPLEX		COMPLEX
MPI_LOGICAL		LOGICAL
...	...	...

初始化MPI系统

- C:

```
int MPI_Init(int *argc, char ***argv);
```

- Fortran 77:

```
MPI_INIT(IERR)
INTEGER IERR
```

初始化MPI系统，通常它应该是第一个被调用的MPI函数。
除MPI_Initialize()外，其它任何MPI函数都只能在它调用后再调用。
在C接口中MPI系统通过argc和argv得到命令行参数。

检测MPI系统是否已经初始化

- C:

```
int MPI_Initialized(int *flag);
```

- Fortran:

```
MPI_INITIALIZED(FLAG,IERR)
LOGICAL FLAG
INTEGER IERR
```

如果已经调用过MPI_Init，则返回flag=true,否则flag＝false，这是唯一可以在MPI_Init之前调用的MPI函数。

得到通信器的进程数及进程在通信器中的序号

- C:

```
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

- Fortran:

```
MPI_COMM_SIZE(COMM,SIZE,IERR)
INTEGER COMM,SIZE,IERR
MPI_COMM_RANK(COMM,RANK,IERR)
INTEGER COMM,RANK,IERR
```

上述两个函数分别返回指定通信器中的进程数及本进程在该通信器中的序号。

退出MPI系统

- C:

```
int MPI_Finalized(void)
```

- Fortran:

```
MPI_FINALIZE(IERR)
INTEGER IERR
```

退出MPI系统。所有MPI进程在正常退出前都必须调用该函数。它是MPI程序中最后一个被调用的MPI函数。 调用该函数之前应该确认所有通信均已完成。
调用该函数之后不允许再调用任何MPI函数。

异常终止

- C:

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

- Fortran:

```
MPI_ABORT(COMM,ERRORCODE,IERR)
INTEGER COMM,ERRORCODE,IERR
```

调用该函数时表明因为出现了某种致命错误而希望异常终止MPI程序的执行。MPI会设法终止通信器comm中的所有进程。

查询处理器名称

- C:

```
int MPI_Get_processor_name(char *name,int *resultlen)
```

-

```
MPI_GET_PROCESSOR_NAME(NAME,RESULTLEN,IERR)
CHARACTER *(*) NAME
INTEGER RESULTLEN,IERR
```

该函数在name中返回进程所在处理器的名称，参数name应该提供不少于MPI_MAX_PROCESSOR_NAME个字节的存储空间。

获取墙上时间及时钟精度

- C:

```
double MPI_Wtime(void)
double MPI_Wtick(void)
```

- Fortran:

```
DOUBLE PRECISION FUNCTION MPI_WTIME()
DOUBLE PRECISION MPI_WTICK()
```

MPI_Wtime返回当前墙上时间，以从某一时刻算起的秒数为单位，而MPI_Wtick则返回MPI_Wtime函数的时钟精度， 也是以秒为单位。 例如：假设MPI_Wtime使用的硬件时钟计数器每1／1000秒增加1，则MPI_Wtick的返回值为10^{-3}, 表示用函数MPI_Wtime得到的时间精度为千分之一秒。

C程序

```
#include "mpi.h"
int main(int argc, char *argv[])
{
int myrank,nprocs;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
...
MPI_Finalize();
return 0;
}
```

一个例子

```
#include<stdio.h>
#include"mpi.h"
int main(int argc, char *argv[])
{
    int myrank, nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Processor %d of %d say: Hello, the parallel world!\
    MPI_Finalize();
    return 0;
}
```

Outline

4 消息传递编程平台MPI

5 MPI基础知识
- MPI的几个基本函数
- MPI程序的基本结构

6 作业

练习

- 编译运行Hello,World!程序
- 练习Makefile文件的编写
- 在你安装的Linux系统上成功安装配置MPICH（可以安装1.2或2.0版本）。
- 利用自己的计算机搭建机群并安装MPICH，可以三个同学一组，自愿结合（提交一个报告描述你的操作过程及心得）。