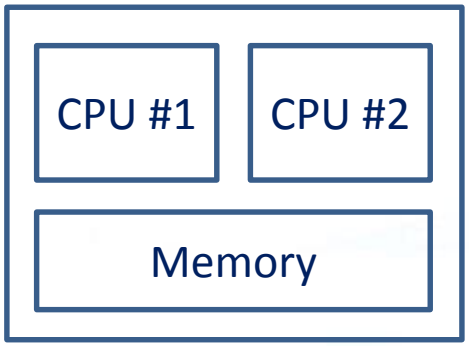
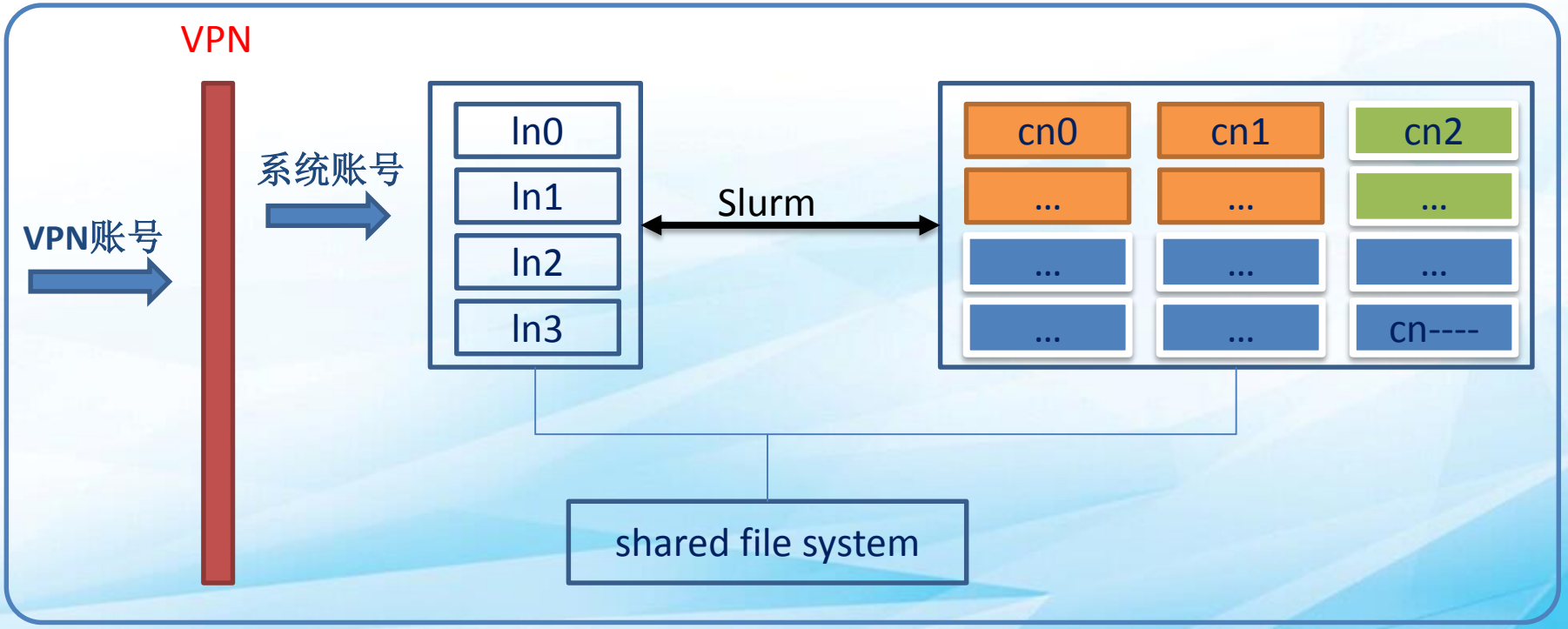
程序的编译运行

中山大学国家超级计算广州中心 应用部 李家辉

Preface

- 一个公共的HPC平台的使用
  - Linux 系统
  - 资源管理系统





国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Preface

- 查看正在使用的节点的信息
  - hostname、top、ssh

```
[nscc-gz_jiahuili@ln0%tianhe2-C ~]$ hostname
ln0
[nscc-gz_jiahuili@ln0%tianhe2-C ~]$ ssh ln1
Last login: Mon Dec  7 17:22:17 2015 from ln0-gn0
[nscc-gz_jiahuili@ln1%tianhe2-C ~]$
```

- 共享文件系统
  - WORKSPACE/
- 查看计算分区
  - yhi

```
PARTITION  AVAIL  TIMELIMIT  NODES  STATE NODELIST
GPU           up   infinite     19   idle cn[11648-11650,11653-11655,
78]
localdisk     up   infinite     27   idle cn[16256-16265,16267-16268,
83-16287]
MEM_128       up   infinite     17   idle cn[11589-11590,11600-11610,
docker        up   infinite     25  idle* cn[10244,10246-10250,10252-
docker        up   infinite      4   idle cn[10240-10242,10271]
docker_128    up   infinite      1  idle* cn10423
docker_128    up   infinite     19   idle cn[10403-10405,10407-10411,
19,10421,10425-10426,10428-10431]
hyyb          up   infinite     14   idle cn[13440-13441,13444-13455]
work*         up   infinite      2   comp cn[9703,10191]
work*         up   infinite     36   drng cn[9259,9307,9333,9469,9529
9958,10000,10107,10113,11989,12006,12078,12096,12098,12151,12169,1217
2300-12302,12398,12525,12564,12569-12570,12627-12628,13420,13513]
work*         up   infinite    504  alloc cn[9218,9222,9225-9226,9231
```

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

内容目录

Linux常用编译器简介

- 课程说明
  - 基于Linux系统
  - 掌握简单程序的编译、运行的流程
  - 本课程主要介绍C、C++、Fortran代码的编译和运行
  - 在天河二号上实践

Linux常用编译器简介

- GCC编译器
  - 功能特点：GNU Compiler Collection，支持C、C++、Objective-C、Fortran、Java、Ada和Go等语言。使用广泛，功能强大，获取方便。
  - 获取途径：开源免费，http://gcc.gnu.org
  - 使用命令：

编程语言	编译器调用命令
C	gcc
C++	g++
Fortran77	gfortran
Fortran90/95	gfortran

Linux常用编译器简介

- Intel编译器
  - 功能特点：Intel公司开发的一款编译器，支持C/C++/Fortran编程语言。编译器针对Intel处理器优化，性能优异。同样支持AMD处理器平台。
  - 获取途径：商业授权，通常通过厂商和intel的合作关系，获得授权
  - 使用命令：

编程语言	编译器调用命令
C	icc
C++	icpc
Fortran77	ifort
Fortran90/95	ifort

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

内容目录

简单程序编译和执行

- 编译器通过命令行调用，最基本的用法是：

gcc [options] [filenames]

icc [options] [filenames]

$gcc hello.c

– hello.c

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

- 程序在shell中的执行

$ ./a.out

$ yhrun -n 1 -p training ./a.out

– 不要在登录节点上运行程序
– -p training参数根据实际的分区确定
– 使用count.c测试

注1：课程都以C语言示例， C++、 Fortran等语言的编译运行流程跟C语言类似
注2：编译器为gcc或icc，它们的使用方法类似，请参见相关文档或man手册

简单程序编译和执行

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

- 程序在shell中的执行

  $ ./a.out

  $ yhrun -n 1 -p training ./a.out

  - 不要在登录节点上运行程序
  - 使用count.c测试

- yhrun [options] program
  - 向资源管理系统申请资源
  - 在申请到的计算节点上运行‘program’
  - yhrun –p training –N 1 –n 1 ./program

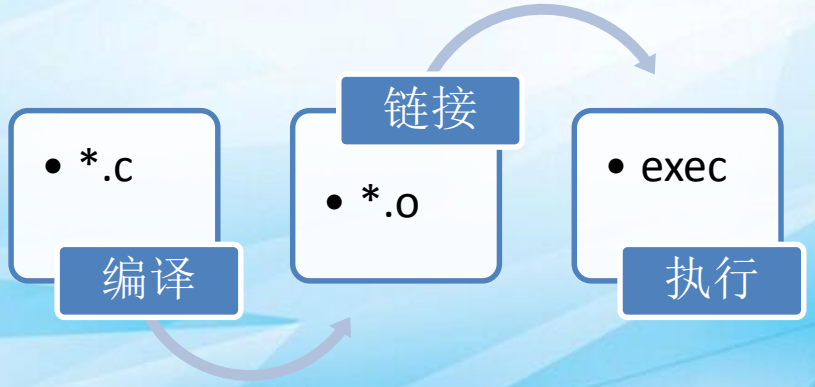
编译流程示例

- 实际使用过程中，比较常用的是分成编译和链接两步：
  - 参数都用“-”开始，“-o”表示输出文件的名称
  - 编译：

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

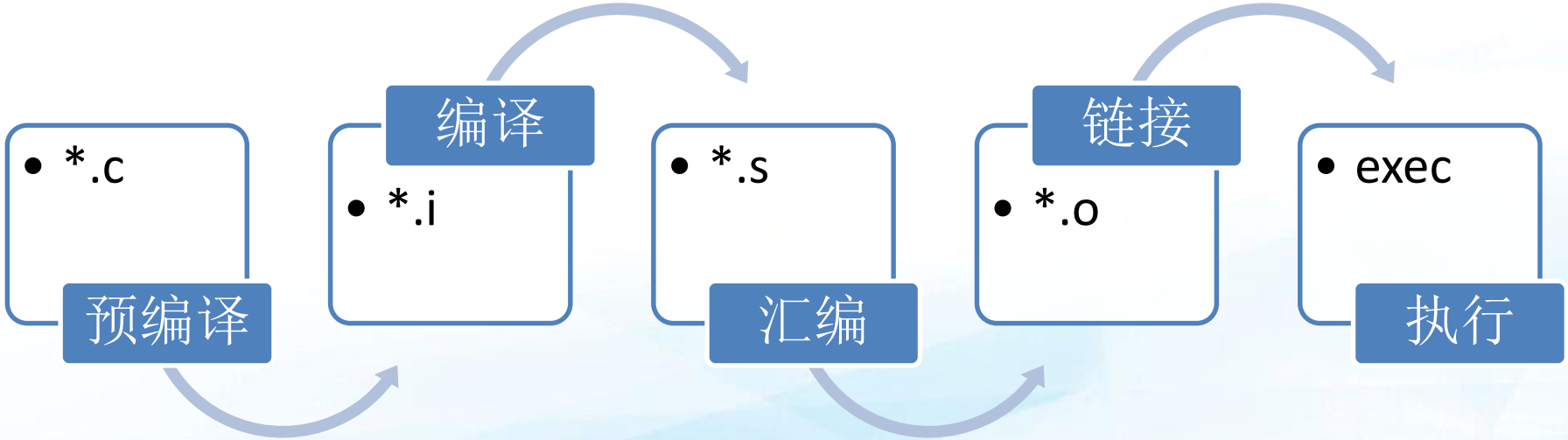
  - 链接：

```
$ gcc     hello.o  -o hello
```



简单程序编译和执行

- 程序的编译详细流程



- 把预编译、编译和汇编三步简称为编译

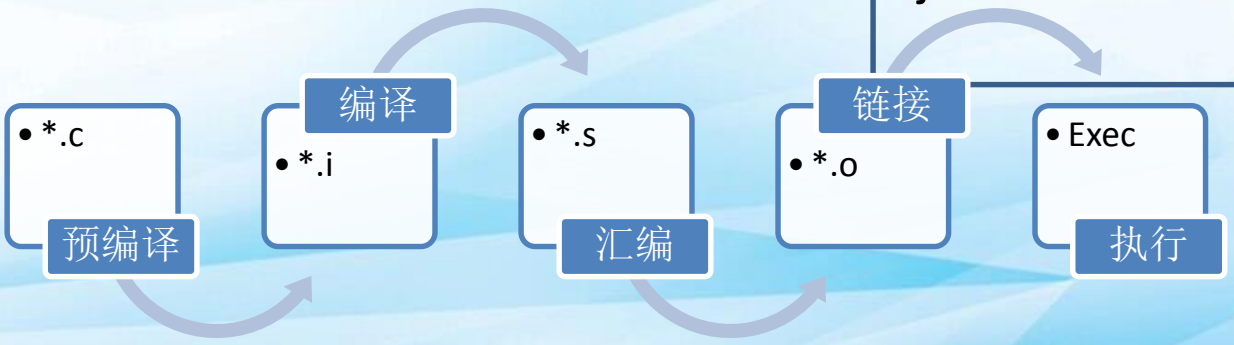
编译流程示例

- 编译流程示例
- 用vi查看每一个输出文件

```
$ gcc -E hello.c -o hello.i
$ gcc -S hello.i  -o hello.s
$ gcc -c hello.s  -o hello.o
$ gcc     hello.o  -o hello
```

– hello.c

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



简单程序编译和执行

- 后缀规范
  - 可执行文件没有后缀要求
  - 源代码、目标文件等有统一的规范（或使用 -x ）

文件类型	后缀名	文件类型	后缀名
C	.c	C++	.C, .cc, .cpp, .cxx
Fortran 77	.f, .for, .F, .FOR	Fortran 90/95	.f90, F90
汇编	.s	目标文件	.o
头文件	.h	Fortran90/95模块	.mod
动态库	.so	静态库	.a

多个源文件例子

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

– *hello_main.c*

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    sayHello();
    return 0;
}
```

– *hello_sub.c*

```
#include<stdio.h>
int sayHello()
{
    printf("Hello.\n");
    return 0;
}
```

多个源文件例子

– hello_main.c
– hello_sub.c
– 编译、链接

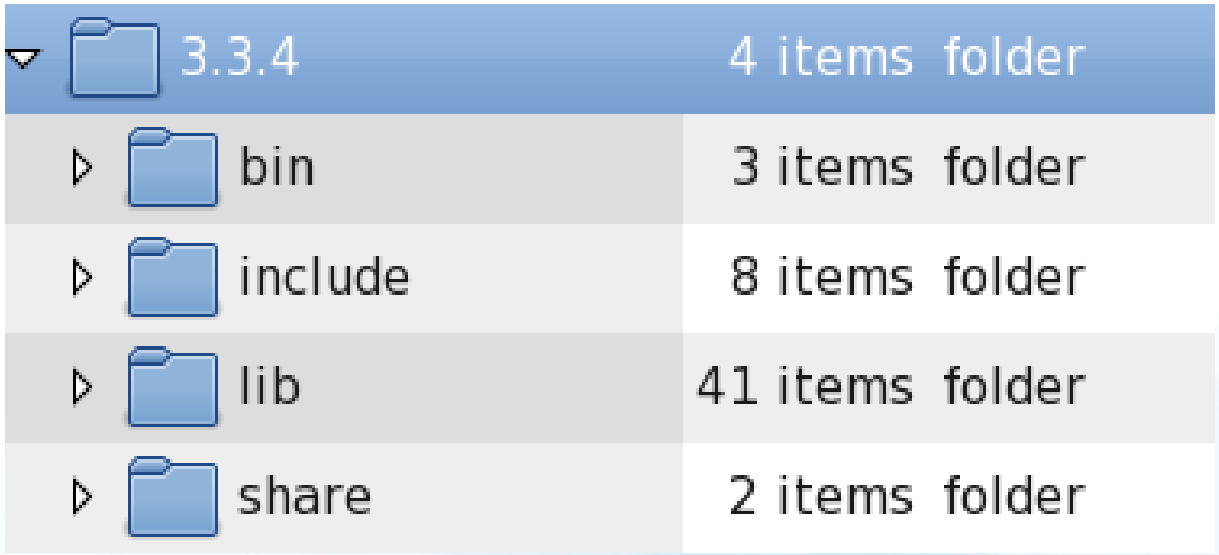
```
$ gcc -c hello_main.c   -o hello_main.o
$ gcc -c hello_sub.c   -o hello_sub.o
$ gcc   hello_main.o   hello_sub.o  -o hello.run

$ yhrun  -n 1 –p training ./hello.run
```

函数库的使用

- 概述
  - 函数库实际上是目标文件（*.o）的一个集合
  - 前面的例子实际上已经使用了系统库
  - 存在大量的开源函数库，可以让使用者通过简单的代码，实现复杂的功能
  - 这些开源库在Linux平台上使用更简便
  - 开发者也可以编译自己的库

函数库的使用

- 通常需要两部分：头文件和库文件



  - 编译阶段，需要使用头文件
  - 链接阶段，需要使用库文件

函数库的使用

- 编译阶段：头文件的使用
  - 编译器自动搜索的头文件目录
    - 源文件所在目录
    - 环境变量指定的目录，如：$CPATH
    - 编译器自己的头文件目录
    - 系统头文件目录
  - 指定头文件搜索路径：使用“-I”参数

-I<path1> -I<path2> ...

简单例子

- 使用fftw3库
- fftw_test.c

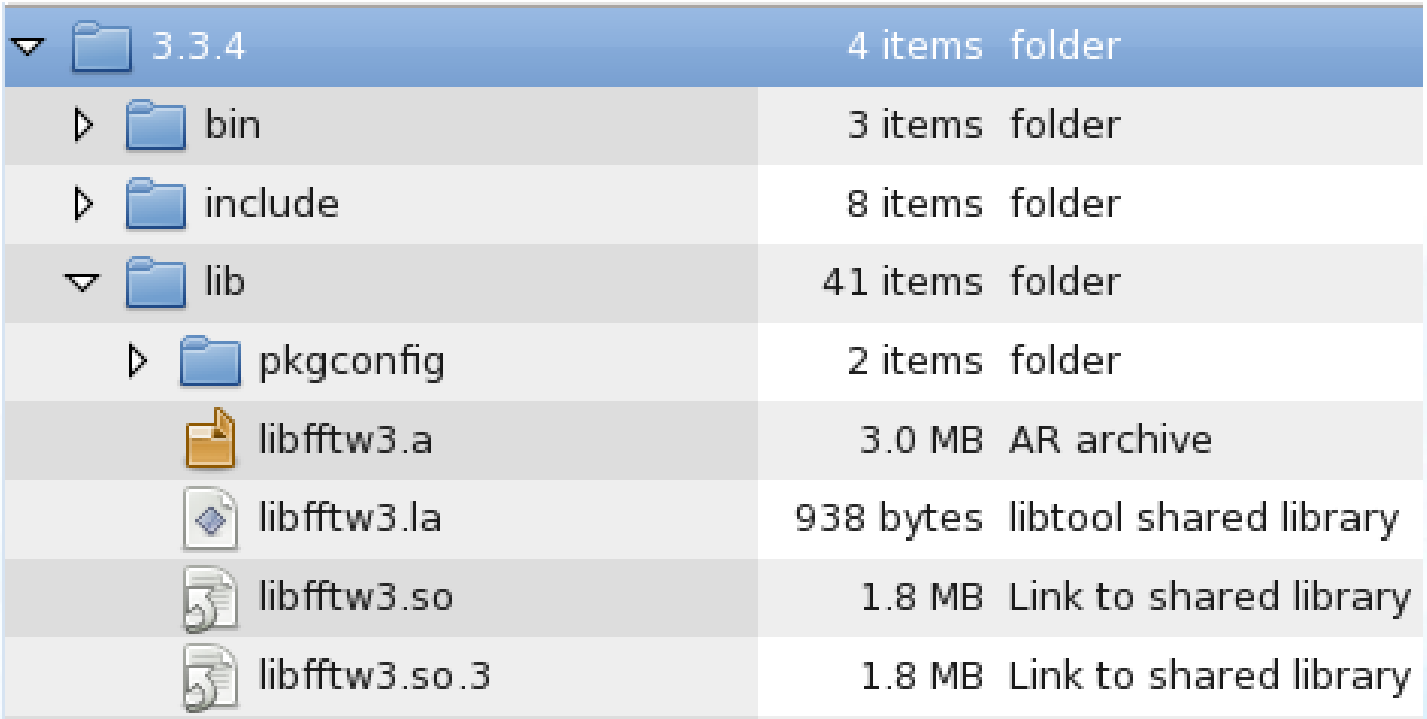
```
$ icc -c -I/WORK/app/fftw/3.3.4/include fftw_test.c -o fftw_test.o
```

```c
#include <stdio.h>
#include "fftw3.h"

#define N 16

int main(int argc, char* argv[])
{
  int i;

  fftw_complex *in,*out;
  fftw_plan p;
  in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
  out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
```

函数库的使用

- 链接阶段
- 静态库
  - 命名规范为libxxx.a，如：libfftw3.a
  - 编译后库函数会被连接到可执行程序，可执行文件体积较大，库函数更新后，需要重新编译程序
  - 运行时不会出现找不到指定动态库的情况
- 动态库
  - 命名规范为libxxx.so，如：libfftw3.so
  - 编译后库函数不会被连接到可执行程序，可执行文，体积较小，可执行文件运行时，库函数动态载入
  - 使用灵活，库函数更新后，不需要重新编译程序
  - 运行时可能会出现找不到指定动态库的情况

- fftw3库


3.3.4 4 items folder
bin 3 items folder
include 8 items folder
lib 41 items folder
pkgconfig 2 items folder
libfftw3.a 3.0 MB AR archive
libfftw3.la 938 bytes libtool shared library
libfftw3.so 1.8 MB Link to shared library
libfftw3.so.3 1.8 MB Link to shared library

函数库的使用

- 方法一：链接时，把库当作目标文件使用
  - fftw_test.c

```
$ icc -c fftw_test.c  -I/WORK/app/fftw/3.3.4/include -o fftw_test.o
$ icc fftw_test.o  /WORK/app/fftw/3.3.4/lib/libfftw3.a  -o fftw_test
$ yhrun -n 1 -p training./fftw_test
```

  - fftw库编译的时候使用的是intel编译器，使用的时候最好使用对应编译器，可以降低编译的复杂度

函数库的使用

- 方法二：使用编译器*-lxxx* 参数，表示在指定库函数路径下搜索名为*libxxx.so*或*libxxx.a*的库文件

```
$ icc -c fftw_test.c  -I/WORK/app/fftw/3.3.4/include -o fftw_test.o
$ icc fftw_test.o -L/WORK/app/fftw/3.3.4/lib -lfftw3 -o fftw_test
$ yhrun -n 1 -p training ./fftw_test
```

函数库的使用

- 方法二：使用编译器*-lxxx* 参数，表示在指定库函数路径下搜索名为*libxxx.so*或*libxxx.a*的库文件
  - 按优先级从高到低搜索以下路径
    - -L指定的搜索路径，可多次指定，-L<path1> -L<path2>
    - LIBRARY_PATH（静态库）、 LD_LIBRARY_PATH（动态库）环境变量指定路径
    - 系统配置文件/etc/ld.so.conf中指定的动态库搜索路径
    - 系统的/lib(64)、 /usr/lib(64)等库文件目录
    - 如果在库函数路径下同时有静态库和动态库，会选择动态库

函数库的生成

- 静态库的生成
  - 编译子函数源代码，得到目标文件

$ gcc -c func1.c

$ gcc -c func2.c

  - 使用ar命令打包目标文件，产生静态库

$ ar cr libtest.a func1.o func2.o

```
/*   func12.h  */
int mySum(int i, int j);
int mySub(int i, int j);
```

```
/*   func1.c  */
int mySum(int i, int j)
{
  return i+j;
}
```

```
/*   func2.c   */
int mySub(int i, int j)
{
  return i-j;
}
```

函数库的生成

- 动态库的生成
  - 使用fPIC参数编译子函数源代码

```
$ gcc -c -fPIC  func1.c
$ gcc -c -fPIC  func2.c
```

  - 使用shared参数产生动态库

```
$gcc -o libtest2.so -shared
func1.o func2.o
```

```
/*   func12.h  */
int mySum(int i, int j);
int mySub(int i, int j);
```

```
/*   func1.c  */
int mySum(int i, int j)
{
  return i+j;
}
```

```
/*   func2.c   */
int mySub(int i, int j)
{
  return i-j;
}
```

编译：

$ gcc -c main.c

使用静态库：

$ gcc main.o -L. -ltest

$ yhrun -n 1 -p training ./a.out

使用动态库：

$ gcc main.o -L. -ltest2

$ ldd a.out

$ yhrun -n 1 -p training ./a.out

```
#include <stdio.h>
#include "func12.h"

int main(int argc, char* argv[])
{
int i, j, k;

i = 1;
j = 2;
k = 0;
printf("i = %d,  j = %d\n", i, j);
k = mySum(i, j);
printf("i+j = %d\n", k);

k = mySub(i, j);
printf("i-j = %d\n", k);

return 0;
}
```

程序的执行

- 运行方式
  - $ ./hello.run
  - $ yhrun -n 1 -p training ./hello.run
- 过程
  - 权限检查
  - 载入动态库，搜索路径优先级(从高到低)：
    - LD_LIBRARY_PATH 环境变量指定的路径
    - 配置文件/etc/ld.so.conf中指定的动态库搜索路径
    - 系统的/lib(64)、 /usr/lib(64)等库文件目录

程序的执行

- 权限检查

```
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ ll hello.run
-rwxr-xr-x 1 nscc-gz_jiahuili nscc-gz 6611 Mar 26 17:22 hello.run
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ ./hello.run
hello
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ chmod u-x hello.run
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ ll hello.run
-rw-r-xr-x 1 nscc-gz_jiahuili nscc-gz 6611 Mar 26 17:22 hello.run
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ ./hello.run
-bash: ./hello.run: Permission denied
```

程序的执行

- 动态库加载
  - 当程序运行时，提示某个库找不到，只需要找到库所在的文件夹，然后把这个文件夹的路径加到LD_LIBRARY_PATH环境变量中

```
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ gfortran fhello.F90 -o fhello.run
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ yhrun ./fhello.run
/HOME/nscc-gz_jiahuili/test/./fhello.run: error while loading shared libr
aries: libgfortran.so.3: cannot open shared object file: No such file or
directory
yhrun: error: cn738: task 0: Exited with exit code 127
```

程序的执行

- LD_LIBRARY_PATH 环境变量
  - 手动添加路径：

$ export LD_LIBRARY_PATH=/WORK/app/fftw/3.3.4/lib:$LD_LIBRARY_PATH

  - 使用配置文件： .bashrc

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
export LD_LIBRARY_PATH=/WORK/app/fftw/3.3.4/lib:$LD_LIBRARY_PATH
```

  - 使用module load

$ module load fftw/3.3.4-default

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

module的使用

- 环境变量管理工具
  - module 通过配置 modulefile 支持环境变量的动态修改，能够控制软件不同版本对环境变量的依赖关系。
  - 通常主要修改$PATH和$LD_LIBRARY_PATH两个环境变量
  - module avail：查看可用的模块的列表
  - module display [modulesfile] : 显示模块的相关内容
  - module load [modulesfile]：能够加载需要使用的 modulefiles

```
[nscc-gz_jiahuili@ln3%tianhe2-C ~]$ echo $PATH
/HOME/nscc-gz_jiahuili/bigdata/mysql/bin:/usr/local/mpi3-dynamic/bin:

[nscc-gz_jiahuili@ln3%tianhe2-C ~]$ echo $LD_LIBRARY_PATH
/opt/intel/composer_xe_2013_sp1.2.144/compiler/lib/intel64:

[nscc-gz_jiahuili@ln3%tianhe2-C ~]$ module load fftw/3.3.4-default

[nscc-gz_jiahuili@ln3%tianhe2-C ~]$ echo $PATH
/WORK/app/fftw/3.3.4/bin:/HOME/nscc-gz_jiahuili/bigdata/mysql/bin:

[nscc-gz_jiahuili@ln3%tianhe2-C ~]$ echo $LD_LIBRARY_PATH
/WORK/app/fftw/3.3.4/lib:/opt/intel/composer_xe_2013_sp1.2.144/compiler/lib/intel64:
```

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

使用yhbatch提交作业

- yhrun的限制
  - 需要保持连接
  - 只能完成简单的工作流
- yhbatch
  - 提交批处理作业，推荐使用
  - yhbatch –N 2 –p training ./job.sh

Bash脚本

- Bash: Bourne Again Shell, GNU Project
- 大多数Linux发行版的默认Shell
- Bash的执行方式
  - 交互式：Linux终端登录后，即进入交互式的执行环境

```
[nscc-gz_jiahuili@ln3%tianhe2-C ~]$ date
Wed Dec  9 14:30:08 CST 2015
[nscc-gz_jiahuili@ln3%tianhe2-C ~]$ sleep 10&
[1] 59230
```

  - 脚本方式

Bash脚本

- 脚本方式
  - 将一些Bash语句写在文本文件中，批量执行

```
#!/bin/bash
NAME=nscc
echo "Hi, $NAME"
```

  - 在新的子Shell中运行：
    - bash test.sh 、./test.sh
  - 不开启子Shell，在当前Shell中运行：
    - source test.sh、. test.sh

```
[nscc-gz_jiahuili@ln3%tianhe2-C 7]$ chmod +x test.sh
[nscc-gz_jiahuili@ln3%tianhe2-C 7]$ ./test.sh
Hi, nscc
[nscc-gz_jiahuili@ln3%tianhe2-C 7]$ echo $NAME

[nscc-gz_jiahuili@ln3%tianhe2-C 7]$ source test.sh
Hi, nscc
[nscc-gz_jiahuili@ln3%tianhe2-C 7]$ echo $NAME
nscc
[nscc-gz_jiahuili@ln3%tianhe2-C 7]$
```

Bash脚本

- 一个简单的脚本

```
#!/bin/bash
#This line is common
NAME=nscc
echo "Hi, $NAME"
exit 0 # This is return code
```

- 第一行 #!/bin/bash是声明，使用bash shell执行脚本，其他地方使用#是注释
- 变量
- 程序主体
- 脚本返回值

Bash脚本

- Bash的变量
  - 变量定义、查看、清除
    - var=value #注意=号前后无空格，不需要声明类型
    - echo $var
    - unset var
  - 全局变量
    - 子Shell会继承
    - export var=value

Bash脚本

- Bash的变量
  - 系统变量
    - HOSTNAME
    - HOME
    - echo $VARNAME
  - 特殊变量

变量名	含义
$?	前一个命令的退出状态，正常退出返回0，异常退出返回非0值
$#	脚本或函数位置参数的个数
$0	脚本或函数的名称
$1, $2, …	传递给脚本或函数的位置参数

Bash脚本

- 主体
  - 完成脚本功能的命令和语法
    - 判断
    - 循环
    - yhrun
    - 重定向

```
#!/bin/bash
#This script should locate at the directory where
#the count.run exist.
LOG_FILE=count.log
yhrun -p free -n 1 ./count.run > $LOG_FILE
```

    - yhbatch –n 1 –p training job.sh

Bash脚本

- 主体
  - 重定向
    - Linux 默认的三个I/O 通道：
    - stdin（标准输入，文件描述符：0） – 默认是键盘
    - stdout（标准输出，文件描述符：1） – 默认是终端
    - stderr（标准错误，文件描述符：2） – 默认是终端
    - < 重定向stdin到文件
    - > 重定向stdout到文件（新建或覆盖）
    - >> 重定向stdout到文件（追加）
    - 2> 重定向stderr到文件（新建或覆盖）
    - 2>> 重定向stderr到文件（追加）
    - 2>&1 重定向stderr到stdout
    - >& 重定向stdout和stderr到文件

内容目录

并行编程模型

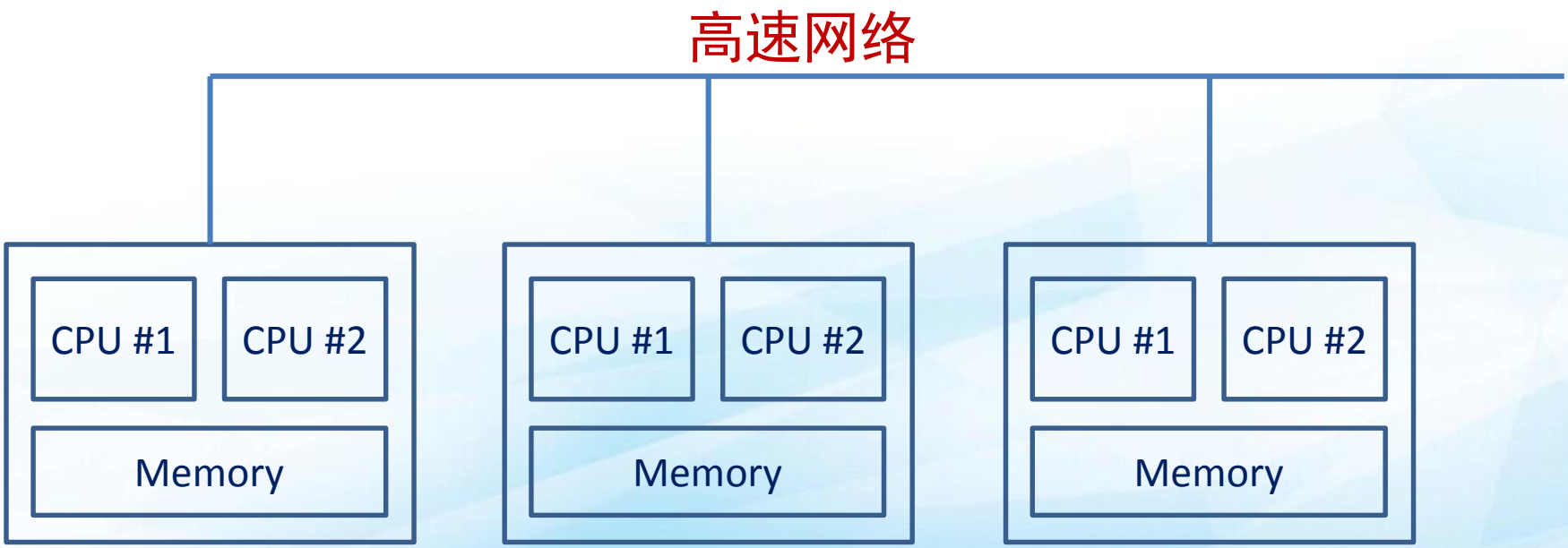
- 并行编程模型

特征	消息传递	共享存储	数据并行
典型代表	MPI，PVM	OpenMP,Pthreads, Cilk	HPF
并行粒度	进程级大粒度	线程级细粒度	进程级细粒度
数据存储模式	分布式存储	共享存储	共享存储
学习入门难度	较难	容易	偏易
可扩展性	好	差	一般

混合使用

- 指令级的并行

- 用高速网络把大量的结点连接在一起，每个结点都是一个共享内存体系。



结点：共享内存体系

OpenMP

- 共享内存式多线程并行
- 支持C、C++、Fortran
- 编译器制导，由编译器提供支持
- 组成部分：
  - 编译器制导指令
  - 运行时库函数
  - 环境变量


```
export OMP_NUM_THREADS=4
```

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[])
{
  int tid=-1;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  printf("%d : Hello World!\n", tid);
}
  return 0;
}
```

OpenMP程序的编译

- OpenMP通过编译器支持，编译时使用恰当的编译参数即可
- 常用编译器的OpenMP编译参数
- GCC编译器：

```
$ gcc  -fopenmp omp.c  -o omp.run
```

- Intel编译器

```
$ icc   -openmp omp.c -o omp.run
```

OpenMP程序的编译

- 当参数使用不当时，编译器提示错误：

$ icc -c omp.c -o omp.o

```
omp.c(8): warning #3180: unrecognized OpenMP #pragma
  #pragma omp parallel
          ^
```

$ icc omp.o -o omp.run

```
omp.o: In function `main':
omp.c:(.text+0x33): undefined reference to `omp_get_thread_num'
```

OpenMP程序的运行

- 只能够单结点运行
- 正确设置环境变量

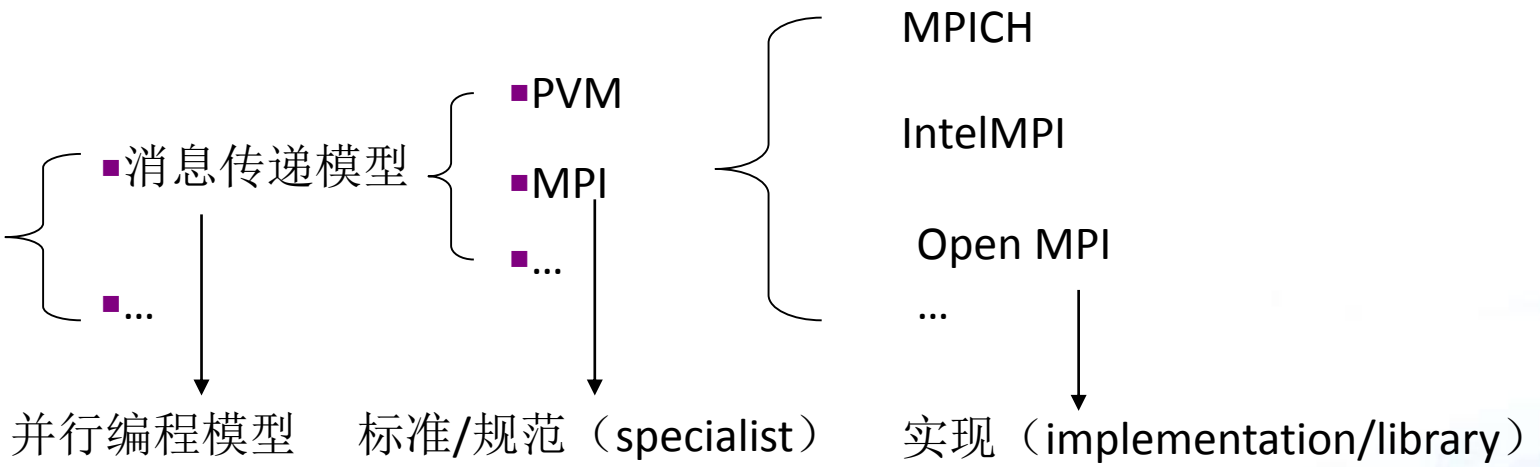
```
$ export OMP_NUM_THREADS=8
$ export KMP_AFFINITY = scatter  #none/scatter/compact
```

- 运行方式跟普通串行程序类似：

```
$ ./omp.run
$ yhrun  -n  1  -p training omp.run
```

使用yhbatch提交

MPI



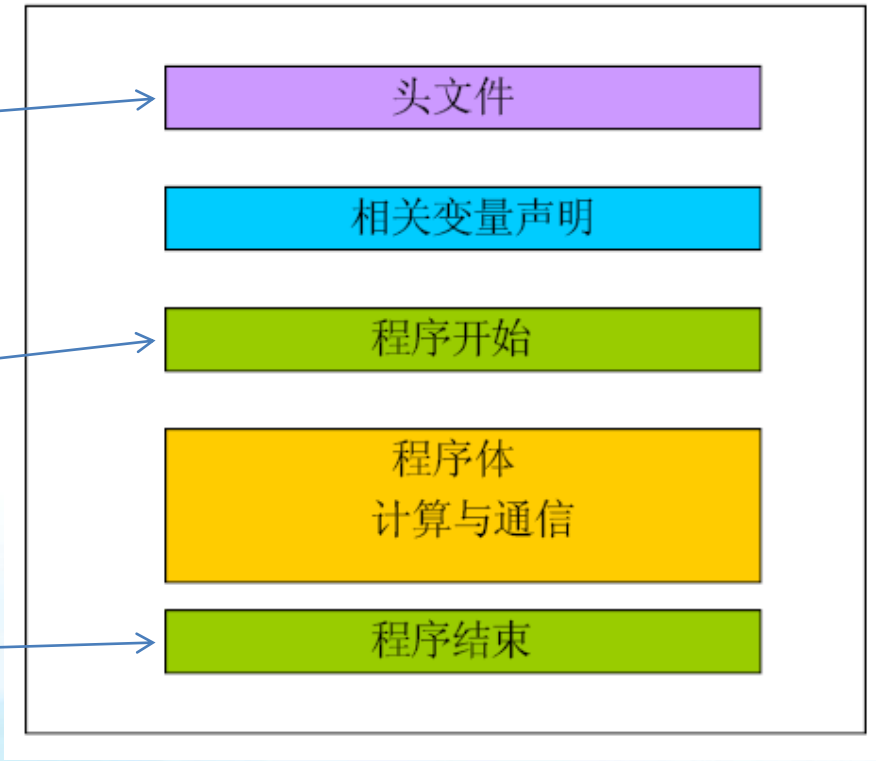
- MPI (Message-Passing Interface) 是消息传递并行模型的一套规范。
- 用户编写程序的时候只需要学习MPI标准接口，不用关心具体的实现
- www.mpi-forum.org

MPI程序编译环境

- 传统语言（C，Fortran，C++）编译器+ 符合MPI标准的库
  - **Intel + IMPI(Intel MPI)**
  - **Intel+MPICH**
  - **GCC+MPICH**

Hello world

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

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

      printf("Hello world!\n");

      MPI_Finalize();
}
```



Hello world!

```
// filename: mpi_hello.c
#include <stdio.h>
#include  “mpi.h”

int main(int argc, char *argv[])
{
    int MyID, NumProcess, NameLen;

    // MPI_MAX_PROCESSOR_NAME Maximun computer name
    char Processor_Name[MPI_MAX_PROCESSOR_NAME];

    // MPI program starts
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &MyID);              // Tag of current process
    MPI_Comm_size(MPI_COMM_WORLD, &NumProcess);       // Total number of processes
    MPI_Get_processor_name(Processor_Name, &NameLen); // Computer name of current process

    printf("Process No.%d of %d on %s \n\n", MyID, NumProcess, Processor_Name);

    // MPI program ends
    MPI_Finalize();

    return 0;
}
```

MPI程序编译

	GCC + MPICH	Intel + MPICH	Intel + IMPI
C	mpicc	mpicc	mpiicc
C++	mpicxx	mpicxx	mpiicpc
Fortran	mpif77/mpif90	mpif77/mpif90	mpiifort

- **$ mpicc -c mpi_hello.c -o mpi_hello.o**
- **$ mpicc mpi_hello.o -o mpi_hello.run**

- **$ which mpicc**
- **$ mpicc -show**

```
[nscc-gz_jiahuili@ln3%tianhe2-C test]$ mpicc -show
icc -fPIC -I/usr/local/mpi3-dynamic/include -L/usr/local/mpi3-
dynamic/lib -Wl,-rpath -Wl,/usr/local/mpi3-dynamic/lib -Wl,--e
nable-new-dtags -lmpi
```

MPI程序的运行

- 使用的动态库：

  $ ldd mpi_hello.run

- 可以单结点运行，也可以多节点运行
- MPI程序启动器
  - 常见MPI实现都会提供MPI程序的启动器
  - mpirun , mpiexec

MPI程序的运行

- 运行的时候需要指定进程数和运行的结点列表
  - 直接指定节点列表

```
$ mpirun  -np  4   -machinefile   nlist   ./mpi_hello.run
```

nlist

```
cn10010
cn10010
cn10923
cn10923
cn1093
```

  - 调度软件提供结点列表，如PBS，slurm

```
$ mpirun  -np  4  ./mpi_hello.run
```

MPI程序的运行

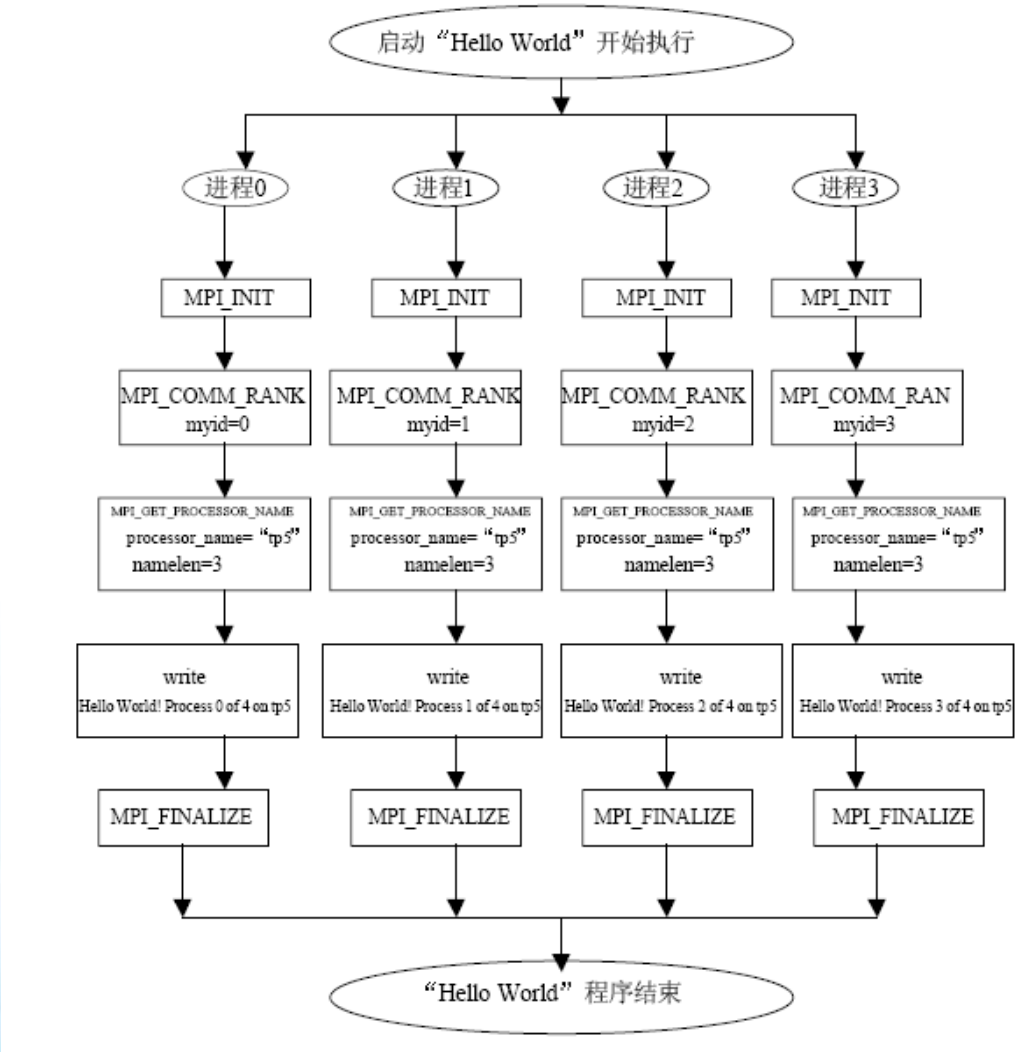
- 天河二号上运行
  - 直接使用yhrun运行

```
$yhrun   -n   4  -p training ./mpi_hello.run
```

  - 使用yhbatch提交作业

```
$ yhbatch -N 1 -p training ./job.sh
```

```
#!/bin/bash
yhrun -n 4 ./mpi_hello.run
```


启动“Hello World”开始执行
进程0
进程1
进程2
进程3
MPI_INIT
MPI_INIT
MPI_INIT
MPI_INIT
MPI_COMM_RANK
myid=0
MPI_COMM_RANK
myid=1
MPI_COMM_RANK
myid=2
MPI_COMM_RAN
myid=3
MPI_GET_PROCESSOR_NAME
processor_name= “tp5”
namelen=3
MPI_GET_PROCESSOR_NAME
processor_name= “tp5”
namelen=3
MPI_GET_PROCESSOR_NAME
processor_name= “tp5”
namelen=3
MPI_GET_PROCESSOR_NAME
processor_name= “tp5”
namelen=3
write
Hello World! Process 0 of 4 on tp5
write
Hello World! Process 1 of 4 on tp5
write
Hello World! Process 2 of 4 on tp5
write
Hello World! Process 3 of 4 on tp5
MPI_FINALIZE
MPI_FINALIZE
MPI_FINALIZE
MPI_FINALIZE
“Hello World”程序结束

内容目录

Make工具

- 开发较大的项目时，需要使用很多的源文件和库，手工编译将十分困难和耗时
- 源文件经过修改后，需要重新编译、链接
- 在Linux系统中，一般使用make工具来自动维护目标文件。
- make工具的优点在于它只重新编译修改过的源文件，而没有修改过的源文件则不作处理。还有一些简化操作的隐式规则和通配符。

Make工具

- 调用命令
  - GNU Make 的主要工作是读进一个文本文件，并根据这个文本文件中的规则执行相应的编译、链接命令
  - make　(默认使用Makefile或makefile)
  - make -f myMakeFile(指定makefile)

Make工具

- Makefile文件
  - make工具会根据Makefile的内容进行编译、链接
- Makefile由规则组成，每一条规则由三部分组成：
  - 目标
  - 依赖文件列表
  - 命令

```
object : dependency
\tab        commands
#注意：行首这里是一个tab，不是空格
```

简单例子

```
$ gcc -c hello_main.c -o hello_main.o
$ gcc -c hello_sub.c -o hello_sub.o
$ gcc hello_main.o hello_sub  -o hello.run
```

Makefile文件内容：

```
# First example for make
hello.run : hello_main.o hello_sub.o
        gcc hello_main.o hello_sub.o  -o hello.run

hello_main.o : hello_main.c
        gcc -c hello_main.c -o hello_main.o
hello_sub.o : hello_sub.c
        gcc -c hello_sub.c -o hello_sub.o
```

– *hello_main.c*

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    sayHello();
    return 0;
}
```

– *hello_sub.c*

```
#include<stdio.h>
int sayHello()
{
    printf("hello\n");
    return 0;
}
```

Make工具

- 使用make命令：

```
$ make  hello.run

gcc -c hello_main.c -o hello_main.o
gcc -c hello_sub.c -o hello_sub.o
gcc hello_main.o hello_sub.o  -o hello.run
```

  - 运行make时，可以接一目标名(如：make hello_sub.o)作为参数，表示要处理的目标。如没有指定，则处理第一个目标
  - 编译完成后，如果只修改了hello_sub.c，执行make，只重新编译hello_sub.c,不会重新编译hello_main.c

Make工具

- make工具执行非编译任务

```
# First example for make
hello.run : hello_main.o hello_sub.o
        gcc hello_main.o hello_sub.o  -o hello.run

hello_main.o : hello_main.c
        gcc -c hello_main.c -o hello_main.o
hello_sub.o : hello_sub.c
        gcc -c hello_sub.c -o hello_sub.o
clean :
        rm -f   hello.run *.o
```

- 执行make clean，将删除目标文件和hello.run

  $ make clean

Make工具

- Makefile使用变量，定义后可以在后面引用
- 变量定义格式：

  变量名 = 变量的值，如：OBJS = main.o sub.o
- 变量的使用

  $(变量名)，如：$(OBJS)

Make工具

- Makefile使用变量

```
# First example for make
EXE = hello.run
OBJS = hello_main.o hello_sub.o
CC = gcc
$(EXE) : $(OBJS)
        $(CC)  $(OBJS) -o $(EXE)
hello_main.o : hello_main.c
        $(CC) -c hello_main.c -o hello_main.o
hello_sub.o : hello_sub.c
        $(CC) -c hello_sub.c -o hello_sub.o
clean :
        rm -f   $(EXE) $(OBJS)
```

Make工具

- 编译开源软件库
  - ./configure [options]
  - make
  - make install

  - cmake /path/of/the/CMakeList/
  - make
  - make install

谢谢！