程序编译

静态链接库与动态链接库

静态链接库和动态链接库的区别在于，在编译器执行连接操作时是否将库文件全部打包到可执行文件中。

静态链接库：在连接生成可执行文件时把库文件的所有内容都拷贝到可执行文件中，此时运行可执行文件时不需要任何依赖，因为在其内部已经有了库中的对应实现。静态链接库在windows上是`.lib`文件，在linux上是`.a`文件。

动态链接库：在连接生成可执行文件时，并不把库文件拷贝到可执行文件中，而是告诉程序这个库文件的位置，在程序运行时才动态载入库文件。此时程序运行需要依赖系统，因为程序内部还没有库的实现。动态链接库在windows上是`.dll`文件，在linux上是`.so`文件

在linux系统上动态库搜索路径的顺序：

1.编译目标代码时指定的动态库搜索路径，通过`-l`选项指定调用的库名，通过`-L`选项指定库所在的路径

2.环境变量`LD_LIBRARY_PATH`指定的动态库搜索路径；

3.配置文件/etc/ld.so.conf中指定的动态库搜索路径；

4.默认的动态库搜索路径/lib和/usr/lib；

编译器和MPI

常用的两套编译器：GCC和INTEL编译器

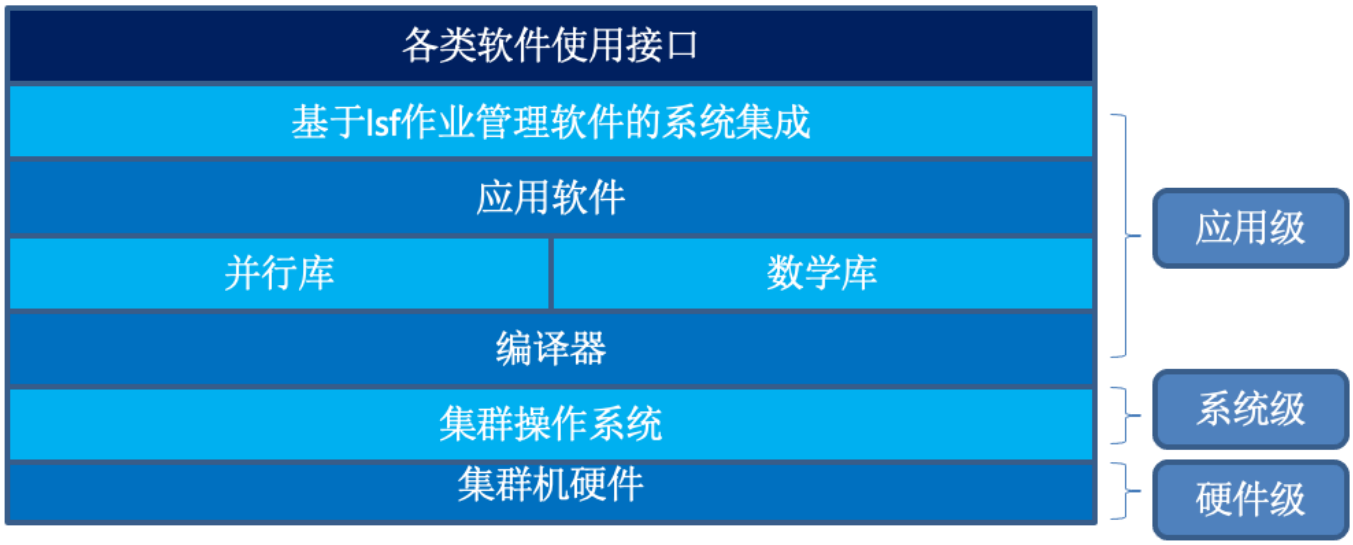
	GCC	INTEL
C编译器	gcc	icc
C++编译器	g++	icpc
Fortran 77编译器	g77	ifort
Fortran 90编译器	gfortran	ifort

openmp和mpi本身都不是语言，均是并行程序设计的标准，根据不同语言有不同的实现形式。其中OpenMP是一种线程级的面向共享存储的并行程序设计标准，其编写较为简单，只要在源文件中的特定位置添加编译制导语句`#pragma`即可，在编译时指定`-fopenmp`参数，编译器就会按照openmp协议对程序执行并行化了。MPI是进程级的面向分布式存储的并行程序设计标准，适用于集群中多机多节点的并行化。其编译较openmp更为复杂，需要在普通编译器的基础上添加必要的MPI参数来执行编译，基于以上两套普通编译器，常用的有两套对应的MPI实现：

	openmpi	intermpi
调用的基础编译器	GCC	INTEL
C编译器	mpicc	mpiicc
C++编译器	mpicxx	mpiicpc
Fortran 77编译器	mpif77	mpifort
Fortran 90编译器	mpif90	mpifort

不同版本的mpi实现对应着不同的基础编译器版本，不同的编译器版本也对应着不同动态库的版本，使用时需搭配使用。比如要使用gcc-4.7.1编译器及其对应的openmpi环境，则应分别修改PATH和LD_LIBRARY_PATH环境变量，把所需要的gcc和openmpi可执行文件的bin目录及对应动态库的lib目录分别添加到这两个环境变量中。当然有了environment modules工具后这些都可以一起用`module load`命令加载了。

软件安装



在集群上应用软件的安装是三部分的组合：特定版本的编译器，对应的并行标准实现，以及数学库（如intel MKL数学库或者单独的BLAS库或LAPACK库等），因此在安装时必须保证当前所处的环境满足软件安装所需的对应版本。一般来说，软件都提供了不同编译器对应的makefile，当然一般情况下intel实现是最快的。

下一节用到的shell命令和makefile

- 管道符号：command 1 | command 2

  把命令1的结果作为命令2的输入

- 逻辑与运算符：command 1 && command 2

  如果命令1返回真，则执行命令2，否则不执行

- 逻辑非运算符：command 1 || command 2

  如果命令1返回假，则执行命令2，否则不执行

- shell脚本的参数：

  $1 - $n 分别指代第一个和第n个参数，$0指代shell脚本本身的名字，这个和C或者C++中的逻辑是一样的：`int main(int argc, char * argv[])`，其中argc代表参数的数目，默认为1；argv为参数字符串数组，其中第0个元素是源文件本身的完整路径，第i个元素是第i个参数

- shell函数：

```
func(){
echo "第一个参数为$1"
echo "参数共有$#个"
echo "作为一个字符串输出所有参数$*"
}
```

函数内部的参数获取和shell脚本内的参数获取的语法是一样的，这里是局部作用域

- shell循环：

```
for i in 1 2 3;do echo $i;done
```

从do开始，以done结束

- Makefile中的shell，每一行是一个进程，不同行之间变量值不能传递。所以，Makefile中的shell不管多长也要写在一行。因此很多makefile中的shell都是以“; \”结尾的，以保证代码在一行中，使得他们同属于一个进程，变量和逻辑可以得到保存。另一方面，makefile的变量分为两个作用域，一个是makefile文件本身的，一个是shell进程的，在makefile中运行shell命令时，先会用自己的变量扩展`$(var)`，然后传递给shell。（注意在**command**区域定义的变量是相当于传递给shell命令的，因此不具备全局可见性）对比下面两种情况：

```
var=3
target:
  echo ${var}
```

make后的结果为：

echo 3
3

```
var=3
target:
  var=4; echo ${var}
```

make后的结果为：

echo 3
3

```
var=3
target:
  var=4; echo $${var}
```

make后的结果为：

echo ${var}
4

从结果的角度来说，只有两个$$的变量才能扩展为shell变量，因为makefile会自动去掉一次。

因此，对文件做批处理时，下面这个命令常常出现在makefile中：

```
@for file in $(FILES); do $(SHELL) address_file.sh $${file} ; done
```

- makefile中的后缀替换规则：

```
objects = foo.o bar.o baz.o,
$(objects:.o=.c) 和 $(patsubst %.o,%.c,$(objects))
```

实现的功能是 一样的，把objects中后缀为.o的部分替换为.c，注意这个只能匹配后缀：

```
target: package-A package-B package-C

package-%:
  echo "install $(@:age-A=--------)"
```

make输出为：

```
echo "install pack--------"
install pack--------
echo "install package-B"
install package-B
echo "install package-C"
install package-C
```

这个在实际中有什么用呢？makefile可以根据所选定的不同目标执行不同包的安装工作，lammps就是这样操作的：

```
target: package-A package-B package-C

package-%:
  echo "install $(@:%=%)"
```

make后输出为：（$@指依次取出目标的每个值）

```
echo "install package-A"
install package-A
echo "install package-B"
install package-B
echo "install package-C"
install package-C
```

- test命令：

  test命令用于检查某个条件是否成立，可以进行数值、字符和文件三方面的测试

  `test "haha" = "haha"`检测两个字符串是否相等，`test -z "haha"`若字符串长度为0则为真，`test -e file`文件存在则为真，`test -f`等等

- makefile中的@：

  make在执行之前会把要执行的命令输出，称之为回显，如果要执行的命令以@开始，则不会回显，比如下面的例子：

```
file:
  @echo a silent hello
  echo hello!
```

  运行`make`后的输出为：

```
a silent hello
echo hello!
hello!
```

- shell中的判断语句：

```
if [ command ];then
   符合该条件执行的语句
elif [ command ];then
   符合该条件执行的语句
else
   符合该条件执行的语句
fi
```

  在if里有一些单独提供的接口，比如`if[-e file]`检测文件是否存在

lammps安装逻辑

通过lammps这个大型项目的组织能学习到很多有益的知识，lammps的源文件在src目录中，makefile中比较重要的部分如下：

```
PACKAGE = asphere body class2 colloid ...

PACKUSER = user-adios user-atc user-awpmd user-bocs ...

YESDIR = $(call uppercase,$(@:yes-%=%))
yes-all:
  @for p in $(PACKALL); do $(MAKE) yes-$$p; done

yes-%:
  # ...
  echo "Installing package $(@:yes-%=%)"; \
  cd $(YESDIR); $(SHELL) ../Install.sh 1; cd ..; \
  $(SHELL) Depend.sh $(YESDIR) 1;
  # ...
```

当需要某些不是默认安装的包时，比如class2包，他们都在src下的子目录中，需要执行`make yes-class2`命令，这时shell会进入class2对应包的位置，执行src目录中的`Install.sh`脚本，进行包的安装工作。当然这个并不是真正的安装了，`Install.sh`脚本如下：

```
# Install/unInstall package files in LAMMPS
# mode = 0/1/2 for uninstall/install/update

# this is default Install.sh for all packages
# if package has an auxiliary library or a file with a dependency,
# then package dir has its own customized Install.sh

mode=$1

# enforce using portable C locale
LC_ALL=C
export LC_ALL

# arg1 = file, arg2 = file it depends on

action () {
  if (test $mode = 0) then
    rm -f ../$1
  elif (! cmp -s $1 ../$1) then
```

```
    if (test -z "$2" || test -e ../$2) then
      cp $1 ..
      if (test $mode = 2) then
        echo "  updating src/$1"
      fi
    fi
  elif (test -n "$2") then
    if (test ! -e ../$2) then
      rm -f ../$1
    fi
  fi
}

# all package files with no dependencies

for file in *.cpp *.h; do
  test -f ${file} && action $file
done
```

使用Install.sh脚本"安装"对应的包时，其实就是把包中的源文件拷贝到src目录中，这样接下来真正执行编译连接操作时，他们就会被构建到可执行文件中。在选定了要安装的包后，执行make mpi执行并行编译，或采用intel编译器进行编译make intel_cpu_intelmpi，lammps提供了多种编译选择（当然在编译器要切换到相应的编译器和mpi环境），这个通过makefile中的.DEFAULT隐含规则定义实现：

```
.DEFAULT:
  @if [ $@ = "serial" ]; \
    then cd STUBS; $(MAKE); cd ..; fi
  @test -f MAKE/Makefile.$@ -o -f MAKE/OPTIONS/Makefile.$@ -o \
    -f MAKE/MACHINES/Makefile.$@ -o -f MAKE/MINE/Makefile.$@
  @if [ ! -d $(objdir) ]; then mkdir $(objdir); fi
  @echo 'Gathering installed package information (may take a little
while)'
  @$(SHELL) Make.sh style
  @$(SHELL) Make.sh packages
  @$(MAKE) $(MFLAGS) lmpinstalledpkgs.h gitversion
  @echo 'Compiling LAMMPS for machine $@'
  @if [ -f MAKE/MACHINES/Makefile.$@ ]; \
    then cp MAKE/MACHINES/Makefile.$@ $(objdir)/Makefile; fi
  @if [ -f MAKE/OPTIONS/Makefile.$@ ]; \
    then cp MAKE/OPTIONS/Makefile.$@ $(objdir)/Makefile; fi
  @if [ -f MAKE/Makefile.$@ ]; \
```

```
    then cp MAKE/Makefile.$@ $(objdir)/Makefile; fi
  @if [ -f MAKE/MINE/Makefile.$@ ]; \
    then cp MAKE/MINE/Makefile.$@ $(objdir)/Makefile; fi
  @if [ ! -e Makefile.package ]; \
    then cp Makefile.package.empty Makefile.package; fi
  @if [ ! -e Makefile.package.settings ]; \
    then cp Makefile.package.settings.empty Makefile.package.settings;
fi
  @cp Makefile.package Makefile.package.settings $(objdir)
  @cd $(objdir); rm -f .depend; \
  $(MAKE) $(MFLAGS) "SRC = $(SRC)" "INC = $(INC)" depend || :
  @rm -f $(ARLINK) $(SHLINK) $(EXE)
ifeq ($(mode),static)
  @cd $(objdir); \
  $(MAKE) $(MFLAGS) "OBJ = $(OBJLIB)" "INC = $(INC)" "SHFLAGS =" \
    "LMPLIB = $(ARLIB)" "ARLIB = $(ARLIB)" "SHLIB = $(SHLIB)" \
    "LMPLINK = $(LMPLINK)" "EXE = ../$(EXE)" ../$(EXE)
  @ln -s $(ARLIB) $(ARLINK)
endif
ifeq ($(mode),shared)
  @cd $(objdir); \
  $(MAKE) $(MFLAGS) "OBJ = $(OBJLIB)" "INC = $(INC)" \
    "LMPLIB = $(SHLIB)" "ARLIB = $(ARLIB)" "SHLIB = $(SHLIB)" \
    "LMPLINK = $(LMPLINK)" "EXE = ../$(EXE)" ../$(EXE)
  @ln -s $(SHLIB) $(SHLINK)
endif
# backward compatibility
ifeq ($(mode),exe)
  $(MAKE) $(MFLAGS) mode=static $@
endif
ifeq ($(mode),lib)
  $(MAKE) $(MFLAGS) mode=static $@
endif
ifeq ($(mode),shexe)
  $(MAKE) $(MFLAGS) mode=shared $@
endif
ifeq ($(mode),shlib)
  $(MAKE) $(MFLAGS) mode=shared $@
endif

ifeq ($(mode),print)
  @cd $(objdir); \
  $(MAKE) $(MFLAGS) "OBJ = $(OBJLIB)" "INC = $(INC)" \
    "EXE = ../$(ARLIB)" -f ../Makefile.print
```

```
endif
```

在makefile中若没找到target对应的规则，且没有其他隐含规则，那么按照.DEFAULTtarget指定的规则进行。