中国科学技术大学超级计算中心

瀚海20超级计算系统使用指南

李会民

2023年6月7日

目录

Part I

前言

本用户使用指南主要将对在中国科学技术大学超级计算中心瀚海20超级计算系统上进行编译以及运行作业做一基本介绍，详细信息请参看相应的文档。

为了便于查看，主要排版约定如下：

- 文件名：*/path/file*
- 环境变量：*MKLROOT*
- 命令：*command parameters*
- 脚本文件内容或长命令：

```
export OPENMPI=/opt/openmpi/1.8.2_intel-compiler-2015.1.133
export PATH=$OPENMPI/bin:$PATH
export MANPATH=$MANPATH:$OPENMPI/share/man
```

- 命令输出：

```
PARTITION  AVAIL TIMELIMIT NODES STATE NODELIST
CPU-Large*    up   infinite   720  alloc  cnode[001-720]
GPU-V100      up   infinite     8    mix  gnode[01-02,05-10]
GPU-V100      up   infinite     2   idle  gnode[03-04]
2TB-AEP-Mem   up   infinite     8   idle  anode[01-08]
```

由于受水平和时间所限，错误和不妥之处在所难免，欢迎指出错误和改进意见，本人将尽力完善。本指南会经常更新，请从超算中心主页上下载更新后的手册。

Part II

瀚海20超级计算系统简介

中国科学技术大学超级计算中心瀚海20超级计算系统采用Mellonax HDR 100Gbps高速互联，具有Intel Xeon Scale 6248、华为鲲鹏920 5250等不同类型CPU及NVIDIA Tesla V100 GPU和华为Atlas 300 AI卡等协处理器，共计2个管理节点、2个用户登录节点、720个普通CPU计算节点（采用高效节能的板级液冷技术）、10个双V100 GPU计算节点、8个2TB Intel AEP大内存节点、20个华为鲲鹏CPU计算节点构成（其中10个各含6颗华为Atlas 300 AI加速卡），计算节点共30480颗CPU核心和20块NVIDIA V100 GPU卡，总双精度浮点计算能力：2.51PFlops（千万亿次/秒，CPU：2.37PFlops，GPU：0.14PFlops），Atlas计算能力：3840 TOPS INT8 + 15360T FLOPS FP16。

- 管理节点（2个）：

  用于系统管理，普通用户无权登录。

节点名	CPU	内存	硬盘	型号
admin01 - admin02	2*Intel Xeon Scale 6248 (2.5GHz，20核，27.5MB)，40核/节点	192GB DDR4 2933MHz	2*1TB NVMe	华为FusionServer 2288H V5

- 用户登录节点（3个）：
  - 用于用户登录、编译与通过作业调度系统提交管理作业等。
  - 禁止在此节点上不通过作业调度系统直接运行作业。

节点名	CPU	内存	硬盘	型号
login01 - login02	2*Intel Xeon Scale 6248 (2.5GHz，20核，27.5MB)，40核/节点	192GB DDR4 2933MHz	2*1TB NVMe	华为FusionServer 2288H V5
Taishan-Login	16*Hi1620 ARM CPU (2.6GHz)	64GB DDR4 2666MHz	50GB	华为泰山 2280H V2

- Intel Xeon CPU普通计算节点（720个）：

  用于多数作业。

节点名	CPU	内存	硬盘	型号
cnode001 - cnode720	2*Intel Xeon Scale 6248 (2.5GHz，20核，27.5MB)，40核/节点	192GB DDR4 2933MHz	1*240GB SSD	华为FusionServ XH321L V5

- Intel Xeon CPU 2TB AEP内存计算节点（8个）：

  AEP内存性能低于普通内存，性价比高，适合大内存应用。

节点名	CPU	普通内存	AEP内存	硬盘	型号
anode01 - anode08	2*Intel Xeon Scale 6248(2.5GHz，20核，27.5MB)，40核/节点	384GB DDR4 2933MHz	2TB (8*256GB)	1*1TB NVMe	华为FusionServer 2288H V5

- GPU计算节点（12个）：

  适合GPU应用，加速性能：https://developer.nvidia.com/hpc-application-performance。

节点名	CPU	GPU	内存	硬盘	型号
gnode01 - gnode10	2*Intel Xeon Scale 6248 (2.5GHz，20核，27.5MB)，40核/节点	2*NVIDIA Tesla V100	384GB DDR4 2933MHz	1*1TB NVMe	华为FusionServer G530 V5
gnode-a100-1 - gnode-a100-2	2*AMD Rome 7742 (2.25GHz, 64核)，128核/节点	8*NVIDIA Tesla A100	1TB DDR4 3200MHz	2*1.92TB+ 4*3.84TB NVMe	融科联创 RW-4124GO-NART

表 1: 单颗NVIDIA Tesla V100 GPU参数

GPU单元	显存	主频	核心数		计算能力(TFlops)			
			Tensor	CUDA	深度学习	半精度	单精度	双精度
GV100	32GB HBM2	基准1230MHz, 加速1370MHz	640	5120	112	28	14	7

表 2: 单颗NVIDIA Tesla A100 GPU参数

峰值性能(TF)		Tensor Core峰值性能						GPU内存	GPU内存带宽	互联
FP64	FP32	FP64(TF)	FP32(TF)	BFLOAT16(TF)	FP16(TF)	INT8(TOPS)	INT4(TOPS)			
9.7	19.5	19.5	156 | 312*	312| 624*	312 | 624*	624 | 1248*	1248 | 2496*	40GBa	1555 GB/s	NVIDIA NVLink 600 GB/s

- 鲲鹏计算节点（20个）：

- 华为ARM V8 CPU，参见：
https://developer.nvidia.com/hpc-application-performance。

- 华为Atlas AI卡，主要提供推理能力，参见：https://support.huawei.com/enterprise/zh/ai-computing-platform/atlas-300-pid-23464095

- 注：使用华为Atlas卡，需特殊申请，加入HwHiAiUser组才可以（运行*id*可以查看自己所在组）。

节点名	CPU	内存	硬盘	计算网络	型号
rnode02 - rnode21	2*鲲鹏920 5250 (48核, 2.6GHz)，共96核	256GB DDR4 2666MHz	1*300GB SAS	25Gbps 以太网	华为TaiShan 2280 V2

其中：rnode[12-21]每台配置6颗Atlas 300 AI卡，rnode[02-11]未配置。

表 3: 单颗Atlas 300 AI卡参数

内存	AI算力	编解码能力
LPDDR4x 32 GB，3200 Mbps	64 TOPS INT8，256T FLOPS FP16	– 支持H.264硬件解码，64路1080P 30FPS（2路 3840*2160 60FPS） – 支持H.265硬件解码，64路1080P 30FPS（2路 3840*2160 60FPS） – 支持H.264硬件编码，4路1080P 30FPS – 支持H.265硬件编码，4路1080P 30FPS – JPEG解码能力4x 1080P 256FPS，编码能力4x 1080P 64FPS – PNG解码能力4x 1080P 48FPS

- 存储系统:
  - 1台长虹DDN GS7990 GRID Scaler及2台DDN SS9012磁盘扩展柜,配置280块8TB SATA硬盘
  - 文件系统：GRID Scaler并行存储
  - 实际可用空间1.5PB
  - 默认用户磁盘配额：100GB
- 计算网络：Mellonax HDR 100Gbps
- 管理网络：千兆以太网

- 操作系统：x86_64架构的64位CentOS Linux 7.7.1908
- 编译器：Intel、PGI和GNU等C/C++ Fortran编译器
- 数值函数库：Intel MKL
- 并行环境：Intel MPI和Open MPI等，支持MPI并行程序；各节点内的CPU共享内存，节点内既支持分布式内存的MPI并行方式，也支持共享内存的OpenMP并行方式；同时支持在节点内部共享内存，节点间分布式内存的混合并行模式。
- 资源管理和作业调度：Slurm 19.05.5
- 常用公用软件安装目录：*/opt*。请自己查看有什么软件，有些软件需要在自己*~/.bashrc*等配置文件中设置后才可以使用。

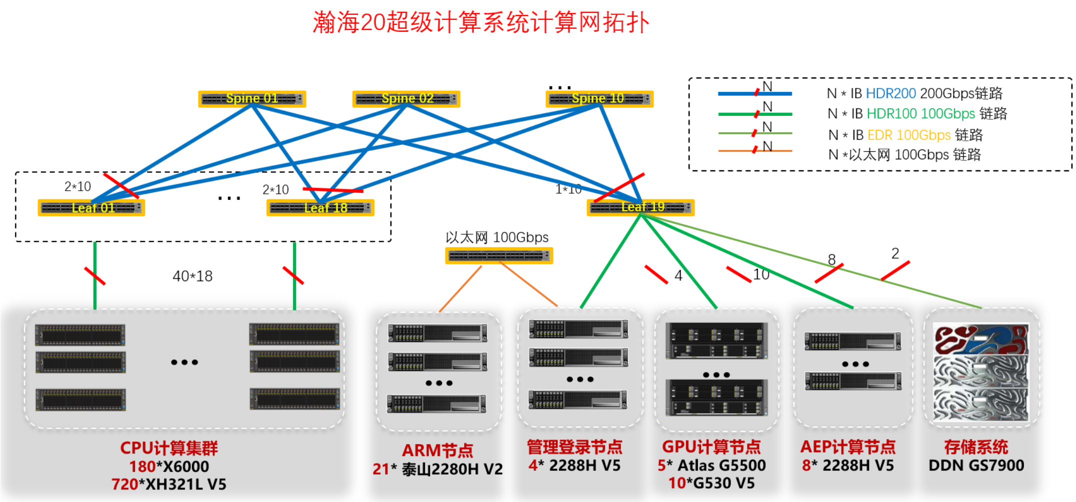

图 1: 瀚海20超级计算系统拓扑

Part III

用户登录与文件传输

本超算系统的操作系统为64位CentOS 7.7 Linux，用户需以SSH方式（在MS Windows下可利用PuTTY、Xshell等支持SSH协议的客户端软件[1]，Win10的命令行也支持ssh及sftp命令）登录到用户登录节点（X86_64 CPU节点：login01、login02，ARM CPU节点：Taishan-Login）后进行编译、提交作业等操作。用户数据可以利用SFTP（不支持FTP）协议进行数据传输。

本超算系统禁止SSH密钥登录，并且采用google authenticator二次验证，用法参见：`http://scc.ustc.edu.cn/2018/0926/c409a339006/page.htm`。

用户若10分钟内5次密码错误登录，那么登录时所使用IP将被自动封锁10分钟禁止登录，之后自动解封，可以等待10分钟后再尝试，或换个IP登录，或联系超算中心老师解封。

本超算系统可从校内IP登录，校外一般无法直接访问。如果需要从校外等登录，可以使用学校的VPN（教师的网络通带有此功能，学生的不带），或者申请校超算中心VPN（`http://scc.ustc.edu.cn/vpn/`）。

用户修改登录密码及shell，只能通过`http://scc.ustc.edu.cn/user/chpasswd.php`修改密码，而不能通过*passwd*和*chsh*修改。请不要设置简单密码和向无关人员泄漏密码，以免给用户造成损失。如果忘记密码，请邮件（sccadmin@ustc.edu.cn）联系中心老师申请重置，并需提供帐号名、所在超算系统名称等必要信息。

用户登录进来的默认语言环境为zh_CN.UTF-8中文[2]，以方便查看登录后的中文提示。如果希望使用英文或GBK中文，可以在自己的*~/.bashrc*中添加export LC_ALL=C或export LC_ALL=zh_CN.GBK。

登录进来后请注意登录后的中文提示，或运行*cat /etc/motd*查看登录提示，也可以运行*faq*命令查看常见问题的回答。

您可运行*du -hs 目录*可以查看目录占用的空间。请及时清除不需要的文件，以释放空间。如需更大存储空间，请与超算中心老师联系，并说明充分理由及所需大小。

超算中心不提供数据备份服务，数据一但丢失或误删将无法恢复，请务必及时下载保存自己的数据。

CentOS(Community ENTerprise Operating System)是Linux主流发行版之一，它来自于Red Hat Enterprise Linux依照开放源代码规定释出的源代码所编译而成。一般来说可以用*man命令*或命令加-h或–help等选项来查看该命令的详细用法，详细信息可参

[1]客户端下载：`http://scc.ustc.edu.cn/411/list.htm`
[2]SSH Secure Shell Client不支持UTF-8中文，不建议使用

考CentOS、Red Hat Enterprise Linux手册或通用Linux手册。

Part IV

设置编译及运行环境

本超算系统安装了多种编译环境及应用等，为方便用户使用，采用Environment Modules工具对其进行了封装，用户可以利用*module*命令设置、查看所需要的环境等。编译和运行程序时在命令行可用*module load modulefile*加载对应的模块（仅对该次登录生效），如不想每次都手动设置，可将其设置在 *~/.bashrc*或 *~/.modulerc*文件中：

- *~/.bashrc*，只Bash启动时设置：

```
module load intel/2020
```

- *~/.modulerc*，每次module命令启动时都设置：

```
#%Module1.0
module load intel/2020
```

注意第一行#%Module1.0是需要的。

module基本语法为：*module [switches] [sub-command] [sub-command-args]*

常用开关参数(switches)：

- –help, -H：显示帮助。
- –force, -f：强制激活依赖解决。
- –terse, -t：以短格式显示。
- –long, -l：以长格式显示。
- –human, -h：以易读方式显示。
- –verbose, -v：显示module命令执行时的详细信息。
- –silent, -s：静默模式，不显示出错信息等。
- –icase, -i：搜索时不区分大小写。

常用子命令(sub-command)：

- *avail [path...]*：显示MODULEPATH环境变量中设置的目录中的某个目录下可用的模块，如有参数指定，则显示MODULEPATH中符合这个参数的路径。如*module avail*：

```
---------------------------------------------------------/opt/Modules/app ------------------------------------------------------
gaussian/g16.C01                   vasp/5.4.4/intel-2020              vasp/5.4.4/vtst/intel-2020
matlab/2019b                       vasp/5.4.4/intelmpimkl2018u4
vasp/5.4.4/hpcx-intel-2019.update5-novtst vasp/5.4.4/vtst/hpcx-intel-2019.update5

------------------------------------------------------/opt/Modules/compiler ---------------------------------------------------
cuda/10.2.89    gcc/7.5.0      gcc/8.3.0      gcc/9.2.0      intel/2018.update4 intel/2019.update5 intel/2020

---------------------------------------------------------/opt/Modules/lib -----------------------------------------------------
mkl/2018.update4 mkl/2019.update5 mkl/2020

---------------------------------------------------------/opt/Modules/mpi ------------------------------------------------------
hpcx/hpcx                hpcx/hpcx-mt             hpcx/hpcx-prof-ompi      intelmpi/2020            openmpi/4.0.2/intel/2020
hpcx/hpcx-debug          hpcx/hpcx-mt-ompi        hpcx/hpcx-stack          openmpi/3.0.5/gcc/9.2.0
hpcx/hpcx-debug-ompi     hpcx/hpcx-ompi           intelmpi/2018.update4    openmpi/3.0.5/intel/2020
hpcx/hpcx-intel-2019.update5 hpcx/hpcx-prof       intelmpi/2019.update5    openmpi/4.0.2/gcc/9.2.0

------------------------------------------------------/opt/Modules/python -----------------------------------------------------
anaconda3      python/3.8.1
```

上面输出：

- /opt/Modules/mpi：模块所在的目录，由*MODULEPATH*环境变量中设定。
- openmpi/3.0.5/intel/2020：模块名或模块文件modulefile，此表示此为3.0.5版本Open MPI，而且是采用2020版本Intel编译器编译的。

- *help [modulefile...]*:显示每个子命令的用法,如给定modulefile参数,则显示modulefile中的帮助信息。
- *add|load modulefile...*:加载modulefile中设定的环境,如*module load openmpi/3.0.5/intel/2020*。
- *rm|unload modulefile...*:卸载已加载的环境modulefile,如*module unload openmpi/3.0.5/intel/2020*。
- *swap|switch [modulefile1] modulefile2*:用modulefile2替换当前已加载的modulefile1,如modulefile1没指定,则交换与modulefile2同样根目录下的当前已加载modulefile。
- *show|display modulefile...*：显示modulefile环境变量信息。如 *module show openmpi/3.0.5/intel/2020*

```
-------------------------------------------------------------------
/opt/Modules/mpi/openmpi/3.0.5/intel/2020:

module-whatis   Open MPI 3.0.5 utilitiesr work with Intel compiler 2020(module load intel/2020)
conflict        openmpi
conflict        intelmpi
conflict        mvapich2
prepend-path    PATH /opt/openmpi/3.0.5/intel/2020/bin
prepend-path    LD_LIBRARY_PATH /opt/openmpi/3.0.5/intel/2020/lib
prepend-path    INCLUDE /opt/openmpi/3.0.5/intel/2020/include
prepend-path    MANPATH /opt/openmpi/3.0.5/intel/2020/share/man
```

上面输出：

- 第一行是modulefile具体路径。
- module-whatis：模块说明，后面可用子命令whatis、apropos、keyword等显示或搜索。
- module load：表示自动加载的模块。
- prepend-path：表示将对应目录加到对应环境变量的前面。

- *clear*：强制module软件相信当前没有加载任何modulefiles。
- *purge*：卸载所有加载的modulefiles。
- *refresh*：强制刷新所有当前加载的不安定的组件。一般用于aliases需要重新初始化，但环境比那两已经被当前加载的模块设置了的派生shell中。
- *whatis [modulefile...]*:显示modulefile中module-whatis命令指明的关于此modulefile的说明，如果没有指定modulefile，则显示所有modulefile的。
- *apropos|keyword string*:在modulefile中module-whatis命令指明的关于此modulefile的说明中搜索关键字，显示符合的modulefile。

其它一些不常用命令及参数，请*man module*查看。

用户也可自己生成自己所需要的modulefile文件，用*MODULEFILE*和环境变量来指该modulefile文件所在目录，用*module*来设置，具体请*man module*及*man modulefile*。

Part V

串行及OpenMP程序编译及运行

在本超算系统上可运行C/C++、Fortran的串行程序，以及与OpenMP和MPI结合的并行程序。编译程序时，用户只需在登录节点(login01、login02)上以相应的编译命令和选项进行编译即可（用户不应到其余节点上进行编译，以免影响系统效率。其它节点一般只设置了运行作业所需要的库路径等，未必设置了编译环境）。当前安装的编译环境主要为：

- C/C++、Fortran编译器：Intel、PGI[3]和GNU编译器，支持OpenMP并行。
- MPI并行环境：HPC-X、Open MPI和Intel MPI并行环境。

安装目录为*/opt*等，系统设置采用module进行管理（参见IV），用户可以采用下述方式之一等需设置自己所需的编译环境运行，如：

- *module load intel/2020*
- 在*~/.bashrc*中设置（设置完成后需要*source ~/.bashrc*或重新登录以便设置生效）：

```
. /opt/intel/2020/bin/compilervars.sh intel64
```

注意：在*~/.bashrc*中设置的级别有可能要高于使用*module load*设置的，可以运行*icc -v*或*which icc*等命令查看实际使用的编译环境。

建议采用对一般程序来说性能较好的Intel编译器，用户也可以选择适合自己程序的编译器，以取得更好的性能。

本部分主要介绍串行C/C++ Fortran源程序和OpenMP并行程序的编译，MPI并行程序的编译将在后面介绍。

1 串行C/C++程序的编译

1.1 输入输出文件后缀与类型的关系

编译器默认将按照输入文件的后缀判断文件类型，见表4。

编译器默认将输出按照文件类型与后缀相对应，见表5。

[3] 目前尚未配置，等以后配置上

表 4: 输入文件后缀与类型的关系

文件名	解释	动作
filename.c	C源文件	传给编译器
filename.C filename.CC filename.cc filename.cpp filename.cxx	C++源文件	传给编译器
filename.a filename.so	静态链接库文件 动态链接库文件	传递给链接器
filename.i	已预处理的文件	传递给标准输出
filename.o	目标文件	传递给链接器
filename.s	汇编文件	传递给汇编器

表 5: 输出文件后缀与文件类型的关系

文件名	解释
filename.i	已预处理的文件，一般使用-p选项生成
filename.o	目标文件，一般使用-c选项生成
filename.s	汇编文件，一般使用-s选项生成
a.out	默认生成的可执行文件

1.2 串行C/C++程序编译举例

- Intel C/C++编译器：
  - 将C程序yourprog.c编译为可执行文件yourprog：
    icc -o yourprog yourprog.c
  - 将C++程序yourprog.cpp编译为可执行文件yourprog：
    icpc -o yourprog yourprog.cpp
  - 将C程序yourprog.c编译为对象文件yourprog.o而不是可执行文件：
    icc -c yourprog.c
  - 将C程序yourprog.c编译为汇编文件yourprog.s而不是可执行文件：
    icc -S yourprog.c
  - 生成带有调试信息的可执行文件以用于调试：
    icc -g yourprog.c -o yourprog
  - 指定头文件路径编译：
    icc -I/alt/include -o yourprog yourprog.c
  - 指定库文件路径及库名编译：
    icc -L/alt/lib -lxyz -o yourprog yourprog.c
- PGI C/C++编译器：
  - 将C程序yourprog.c编译为可执行文件yourprog：
    pgcc -o yourprog yourprog.c
  - 将C++程序yourprog.cpp编译为可执行文件yourprog：
    pgCC -o yourprog yourprog.cpp
  - 将C程序yourprog.c编译为对象文件yourprog.o而不是可执行文件：
    pgcc -c yourprog.c
  - 将C程序yourprog.c编译为汇编文件yourprog.s而不是可执行文件：
    pgcc -S yourprog.c
  - 生成带有调试信息的可执行文件以用于调试：
    pgcc -g yourprog.c -o yourprog
  - 指定头文件路径编译：
    pgcc -I/alt/include -o yourprog yourprog.c
  - 指定库文件路径及库名(-l)编译：
    pgcc -L/alt/lib -lxyz -o yourprog yourprog.c

- GNU C/C++编译器：
  - 将C程序yourprog.c编译为可执行文件yourprog：*gcc -o yourprog yourprog.c*
  - 将C++程序yourprog.cpp编译为可执行文件yourprog：
    g++ -o yourprog yourprog.cpp
  - 将C程序yourprog.c编译为对象文件yourprog.o而不是可执行文件：
    gcc -c yourprog.c
  - 将C程序yourprog.c编译为汇编文件yourprog.s而不是可执行文件：
    gcc -S yourprog.c
  - 生成带有调试信息的可执行文件以用于调试：
    gcc -g yourprog.c -o yourprog
  - 指定头文件路径编译：
    gcc -I/alt/include -o yourprog yourprog.c
  - 指定库文件路径及库名编译：
    gcc -L/alt/lib -lxyz -o yourprog yourprog.c

2 串行Fortran程序的编译

2.1 输入输出文件后缀与类型的关系

编译器默认将按照输入文件的后缀判断文件类型，见表6。

编译器默认将输出按照文件类型与后缀相对应，见表7。

表 6: 输入文件后缀与文件类型的关系

文件名	解释	动作
filename.a	静态链接库文件，多个.o文件的打包集合	传给编译器
filename.f filename.for filename.ftn filename.i	固定格式的Fortran源文件	被Fortran编译器编译
filename.fpp filename.FPP filename.F filename.FOR filename.FTN	固定格式的Fortran源文件	自动被Fortran编译器预处理后再被编译
filename.f90 filename.i90	自由格式的Fortran源文件	被Fortran编译器编译
filename.F90	自由格式的Fortran源文件	自动被Fortran编译器预处理后再被编译
filename.s	汇编文件	传递给汇编器
filename.so	动态链接库文件，多个.o文件的打包集合	传递给链接器
filename.o	目标文件	传递给链接器

表 7: 输出文件后缀与类型的关系

文件名	解释	生成方式
filename.o	目标文件	编译时添加-c选项生成
filename.so	共享库文件	编译时指定为共享型，如添加-shared，并不含-c
filename.mod	模块文件	编译含有MODULE声明时的源文件生成
filename.s	汇编文件	编译时添加-S选项生成
a.out	默认生成的可执行文件	编译时没有指定-c时生成

2.2 串行Fortran程序编译举例

- Intel Fortran编译器：
  - 将Fortran 77程序yourprog.for编译为可执行文件yourprog：
    ifort -o yourprog yourprog.for
  - 将Fortran 90程序yourprog.f90编译为可执行文件yourprog：
    ifort -o yourprog yourprog.f90
  - 将Fortran 90程序yourprog.90编译为对象文件yourprog.o而不是可执行文件：
    ifort -c yourprog.f90
  - 将Fortran程序yourprog.f90编译为汇编文件yourprog.s而不是可执行文件：
    ifort -S yourprog.f90
  - 生成带有调试信息的可执行文件以用于调试：
    ifort -g yourprog.f90 -o yourprog
  - 指定头文件路径编译：
    ifort -I/alt/include -o yourprog yourprog.f90
  - 指定库文件路径及库名编译：
    ifort -L/alt/lib -lxyz -o yourprog yourprog.f90
- PGI Fortran编译器：
  - 将Fortran 77程序yourprog.for编译为可执行文件yourprog：
    pgf77 -o yourprog yourprog.for
  - 将Fortran 90程序yourprog.f90编译为可执行文件yourprog：
    pgf90 -o yourprog yourprog.f90
  - 将Fortran程序yourprog.f90编译为对象文件yourprog.o而不是可执行文件：
    pgf90 -c yourprog.f90
  - 将Fortran程序yourprog.f90编译为汇编文件yourprog.s而不是可执行文件：
    pgf90 -S yourprog.f90
  - 生成带有调试信息的可执行文件以用于调试：
    pgf90 -g yourprog.f90 -o yourprog
  - 指定头文件路径编译：
    pgf90 -I/alt/include -o yourprog yourprog.f90
  - 指定库文件路径及库名编译：
    pgf90 -L/alt/lib -lxyz -o yourprog yourprog.f90

- GNU Fortran编译器：
  - 将Fortran 77程序yourprog.for编译为可执行文件yourprog：
    - gcc 4.x系列：*gfortran -o yourprog yourprog.for*
    - gcc 3.x系列：*g77 -o yourprog yourprog.for*
  - 将Fortran 90程序yourprog.f90编译为可执行文件yourprog：
    gfortran -o yourprog yourprog.f90
  - 将Fortran程序yourprog.f90编译为对象文件yourprog.o而不是可执行文件：
    gfortran -c yourprog.f90
  - 将Fortran程序yourprog.f90编译为汇编文件yourprog.s而不是可执行文件：
    gfortran -S yourprog.f90
  - 生成带有调试信息的可执行文件以用于调试：
    gfortran -g yourprog.f90 -o yourprog
  - 指定头文件路径编译：
    gfortran -I/alt/include -o yourprog yourprog.f90
  - 指定库文件路径及库名编译：
    gfortran -L/alt/lib -lxyz -o yourprog yourprog.f90

注意：*g77*既不支持OpenMP，也不支持Fortran 90及之后的标准。

3　OpenMP程序的编译与运行

3.1　OpenMP程序的编译

Intel、PGI和GNU编译器都支持OpenMP并行，只需利用相关编译命令结合必要的OpenMP编译选项编译即可。对应此三种编译器的OpenMP编译选项：

- Intel编译器：-qopenmp（2015及之后版）
- PGI编译器：-mp
- GNU编译器：-fopenmp

采用这三种编译器的OpenMP源程序编译例子如下：

- Intel编译器：
  - 将C程序yourprog-omp.c编译为可执行文件yourprog-omp：
    icc -qopenmp -o yourprog-omp yourprog.c
  - 将Fortran 90程序yourprog-omp.f90编译为可执行文件yourprog-omp：
    ifort -qopenmp -o yourprog-omp yourprog.f90
- PGI编译器：
  - 将C程序yourprog-omp.c编译为可执行文件yourprog-omp：
    pgcc -mp -o yourprog-omp yourprog.c
  - 将Fortran 90程序yourprog-omp.f90编译为可执行文件yourprog-omp：
    pgf90 -mp -o yourprog-omp yourprog.f90
- GNU编译器：
  - 将C程序yourprog-omp.c编译为可执行文件yourprog-omp：
    gcc -fopenmp -o yourprog-omp yourprog.c
  - 将Fortran 90程序yourprog-omp.f90编译为可执行文件yourprog-omp：
    gfortran -fopenmp -o yourprog-omp yourprog.f90

3.2 OpenMP程序的运行

OpenMP程序的运行一般是通过在运行前设置环境变量*OMP_NUM_THREADS*来控制线程数，比如在BASH中利用*export OMP_NUM_THREADS=40*设置使用40个线程运行。

注意，本系统为节点内共享内存节点间分布式内存的架构，因此只能在一个节点上的CPU之间运行同一个OpenMP程序作业，在提交作业时需要使用相应选项以保证在同一个节点运行。

Part VI

Intel、PGI及GNU C/C++ Fortran编译器介绍

4 Intel C/C++ Fortran编译器

4.1 Intel C/C++ Fortran编译器简介

Intel Parallel Studio XE Cluster版C/C++ Fortran编译器，是一种主要针对Inetl平台的高性能编译器，可用于开发复杂且要进行大量计算的C/C++、Fortran程序。

系统当前安装目录为*/opt/intel*，其下有多种年份版本。官方手册目录为*/opt/intel/版本/documentation_年份*。用户可以采用molule命令来设置所需的环境，请参看设置编译及运行环境。

Intel编译器编译C和C++源程序的编译命令分别为*icc*和*icpc*；编译Fortran源程序的命令为*ifort*。*icpc*命令使用与*icc*命令相同的编译器选项，利用*icpc*编译时将后缀为.c和.i的文件看作为C++文件；而利用*icc*编译时将后缀为.c和.i的文件则看作为C文件。用*icpc*编译时，总会链接C++库；而用*icc*编译时，只有在编译命令行中包含C++源文件时才链接C++库。

编译命令格式为:*command [options] [@response_file] file1 [file2...]*,其中*response_file*为文件名，此文件包含一些编译选项，请注意调用时前面有个@。

在Intel数学库(Intel®math)中的许多函数针对Intel微处理器相比针对非Intel微处理器做了非常大的优化处理。

为了使用Intel数学库中的函数，需要在程序源文件中包含头文件*mathimf.h*，例如使用实函数:

```
// real_math.c
#include <stdio.h>
#include <mathimf.h>

int main() {
 float fp32bits;
 double fp64bits;
 long double fp80bits;
 long double pi_by_four = 3.141592653589793238/4.0;
```

```
// pi/4 radians is about 45 degrees
 fp32bits = (float) pi_by_four; // float approximation to pi/4
 fp64bits = (double) pi_by_four; // double approximation to pi/4
 fp80bits = pi_by_four; // long double (extended) approximation to pi/4

// The sin(pi/4) is known to be 1/sqrt(2) or approximately .7071067
 printf("When x = %8.8f, sinf(x) = %8.8f \n", fp32bits, sinf(fp32bits));
 printf("When x = %16.16f, sin(x) = %16.16f \n", fp64bits, sin(fp64bits));
 printf("When x = %20.20Lf, sinl(x) = %20.20f \n", fp80bits, sinl(fp80bits));

 return 0;
}
```

编译：*icc real_math.c*

4.2 编译错误

C/C++程序编译时的出错信息类似以下：

```
netlog.c(140): error: identifier "hhh" is undefined
                          for(int hhh=domain_cnt+1;hhh>TMP;hhh--){
                                                   ^

netlog.c(156): error: expected an expression
    for(int i=0;i<32;i++)for(int j=0;j<256;j++)if(ip1[i][j]!=0)fprintf(fin);
        ^
```

Fortran程序编译时的出错信息类似以下：

```
NOlihm.f90(146): error #6404: This name does not have a type, and must have
an explicit type. [NPR]
  n2nd=0; npr=0
-----------^
NOlihm.f90(542): remark #8290: Recommended relationship between field width
'W' and the number of fractional digits 'D' in this edit descriptor is 'W>=D+3'.
6060 format(/i2,'-th layer',i2,'-th element: z=',i3,' a=',f9.5/' Ef=',f7.5)
----------------------------------------------------------------------^
```

编译错误的格式为：

- 源文件名(行数): 错误类型:具体说明
- 源代码，^指示出错位置

错误类型可以为：

- Warning：警告，报告对编译有效但也许存在问题的语法，请根据信息及程序本身判断，不一定需要处理。
- Error：存在语法或语义问题，必须要处理。
- Fatal Error：报告环境错误，如磁盘空间没有了。

4.3 Fortran程序运行错误

根据运行时错误代码可以在官方手册中查找对应错误解释。

4.4 Intel Parallel Studio XE版重要编译选项

Intel编译器选项分为几类，可以用*icc -help 类别*查看对应的选项，类别与选项对应关系如下：

advanced - Advanced Optimizations，高级优化

codegen - Code Generation，代码生成

compatibility - Compatibility，兼容性

component - Component Control，组件控制

data - Data，数据

deprecated - Deprecated Options，过时选项

diagnostics - Compiler Diagnostics，编译器诊断

float - Floating Point，浮点

help - Help，帮助

inline - Inlining，内联

ipo - Interprocedural Optimization (IPO)，过程间优化

language - Language，语言

link - Linking/Linker，链接/链接器

misc - Miscellaneous，杂项

opt - Optimization，优化

output - Output，输出

pgo - Profile Guided Optimization (PGO)，概要导向优化

preproc - Preprocessor，预处理

reports - Optimization Reports，优化报告

openmp - OpenMP and Parallel Processing，OpenMP和并行处理

可以运行*icc -help help*查看选项分类情况。

4.4.1 优化选项

- -fast：最大化整个程序的速度，相当于设置-ipo、-O3、-no-prec-div、-static、-fp-model fast=2和-xHost。这里是所谓的最大化，还是需要结合程序本身使用合适的选项，默认不使用此选项。
- -nolib-inline：取消标准库和内在函数的内联展开。
- -O*n*：设定优化级别，默认为O2。O与O2相同，推荐使用；O3为在O2基础之上增加更激进的优化，比如包含循环和内存读取转换和预取等，但在有些情况下速度反而慢，建议在具有大量浮点计算和大数据处理的循环时的程序使用。
- -Ofast：设定一定的优化选项提高程序性能，设定-O3, -no-prec-div和-fp-model fast=2。在Linux系统上提供与gcc的兼容。
- -Os：启用优化，但不增加代码大小，并且产生比-O2优化小的代码。它取消了一些优化不明显却增大了代码的优化选项。

4.4.2 代码生成选项

- -ax*code*：在有性能提高时，生成针对Intel®处理器的多特征面向的自动调度代码路径。*code*可为：
  - COMMON-AVX512：生成Intel(R) Advanced Vector Extensions 512 (Intel(R) AVX-512)基础指令。
  - CORE-AVX2:生成Intel®Advanced Vector Extensions 2 (Intel®AVX2)、Intel®AVX、SSE4.2、SSE4.1、SSE3、SSE2、SSE和SSSE3指令。

  - CORE-AVX-I:生成Float-16转换指令和RDRND(随机数)指令、Intel®Advanced Vector Extensions (Intel®AVX)、Intel®SSE4.2、SSE4.1、SSE3、SSE2、SSE和SSSE3指令。
  - AVX:生成Intel®Advanced Vector Extensions (Intel®AVX)、Intel®SSE4.2、SSE4.1、SSE3、SSE2、SSE和SSSE3指令。
  - SSE4.2：生成Intel®SSE4.2、SSE4.1、SSE3、SSE2、SSE和SSSE3指令。
  - SSE4.1：生成Intel®SSE4.1、SSE3、SSE2、SSE和SSSE3指令
  - SSSE3：生成SSSE3指令和Intel®SSE3、SSE2和SSE指令。
  - SSE3：生成Intel®SSE3、SSE2和SSE指令。
  - SSE2：生成Intel®SSE2和SSE指令。

- -fexceptions、-fno-exceptions：是否生成异常处理表。
- -x*code*：设置启用编译目标的特征，包含采取何种指令集和优化。
  - COMMON-AVX512
  - CORE-AVX2
  - CORE-AVX-I
  - AVX
  - SSE4.2
  - SSE4.1
  - SSSE3
  - SSE3
  - SSE2
- -m*code*：需要生成目标特征的指令集。*code*可为：
  - avx:生成Intel®Advanced Vector Extensions (Intel®AVX)、Intel®SSE4.2、SSE4.1、SSE3、SSE2、SSE和SSSE3指令。
  - sse4.2：生成Intel®SSE4.2、SSE4.1、SSE3、SSE2、SSE和SSSE3指令。
  - sse4.1：生成Intel®SSE4.1、SSE3、SSE2、SSE和SSSE3指令。
  - ssse3：生成SSSE3指令和Intel®SSE3、SSE2和SSE指令。
  - sse3：生成Intel®SSE3、SSE2和SSE指令。
  - sse2：生成Intel®SSE2和SSE指令。
  - sse：已过时，现在与ia32一样。

  - ia32：生成与IA-32架构兼容的x86/x87通用代码。取消任何默认扩展指令集，任何之前的扩展指令集。并且取消所有面向特征的优化及指令。此值仅在Linux系统上使用IA-32架构时有效。

- -m32和-m64：生成IA-32或Intel®64位代码，默认由主机系统设定。
- -march=*processor*：生成支持某种处理器特定特征的代码。*processor*可为：
  - generic
  - core-avx2
  - core-avx-i
- -mtune=*processor*：针对特定处理器优化。*processor*可为：
  - generic（默认）
  - core-avx2
  - core-avx-i
- -xHost：生成编译主机处理器能支持的最高指令集。建议使用。

4.4.3 过程间优化(IPO)选项

- -ip：在单个文件中进行过程间优化(Interprocedural Optimizations-IPO)。
- -ip-no-inlining：禁止过程间优化时启用的全部和部分内联。
- -ip-no-pinlining：禁止过程间优化时启用的部分内联。
- -ipo*[n]*、-no-ipo：是否在多文件中进行过程间优化，非负整数*n*为可生成的对象文件数。
- -ipo-c：在多文件中进行过程间优化，并生成一个对象文件。
- -ipo-jobs*n*：指定在过程间优化的链接阶段时的命令（作业）数。
- -ipo-S：在多文件中进行过程间优化，并生成一个汇编文件。
- -ipo-separate：在多文件中进行过程间优化，并为每个文件分别生成一个对象文件。

4.4.4 高级优化选项

- -funroll-all-loops：即使在循环次数不确定的情况下也展开所有循环。默认为否。
- -guide*[=n]*：设置自动向量化、自动并行及数据变换的指导级别。*n*为1到4，1为标准指导，4为最高指导，如果*n*忽略，则默认为4。默认为不启用。
- -guide-data-trans*[=n]*：设置数据变换时的指导级别。*n*为1到4，1为标准指导，4为最高指导，如果*n*忽略，则默认为4。默认为不启用。
- -guide-file*[=filename]*：将自动并行的结果输出到文件*filename*中。
- -guide-file-append*[=filename]*：将自动并行的结果追加到文件*filename*中。
- -guide-par*[=n]*：设置自动并行的指导级别。*n*为1到4，1为标准指导，4为最高指导，如果*n*忽略，则默认为4。默认为不启用。
- -guide-vec*[=n]*：设置自动向量化的指导级别。*n*为1到4，1为标准指导，4为最高指导，如果*n*忽略，则默认为4。默认为不启用。
- -mkl*[=lib]*：链接时自动链接Intel MKL库，默认为不启用。*lib*可以为：
  - parallel：采用线程化部分的MKL库链接，此为*lib*如果没指明时的默认选项。
  - sequential：采用未线程化的串行MKL库链接。
  - cluster：采用集群部分和串行部分MKL链接。
- -simd、-no-simd：是否启用SIMD编译指示的编译器解释。
- -unroll*[=n]*：设置循环展开的最大层级。
- -unroll-aggressive、-no-unroll-aggressive：设置对某些循环执行激进展开。默认不启用。
- -vec、-no-vec：是否启用向量化。默认启用。

4.4.5 概要导向优化(PGO)选项

- -p：使用gprof编译和链接函数。
- -prof-dir *dir*：设定存储概要导向优化信息的文件目录。
- -prof-file *filename*：设定概要摘要文件名。

4.4.6 优化报告选项

- -qopt-report*[=n]*：设定显示优化报告信息的级别，为每个对象文件生成一个对应的文件。*n*为0（不显示）到5（最详细）。
- -qopqopt-report-file=*keyword*：设定报告文件名。*keyword*可以为：
  - filename：保存输出的文件名。
  - stderr：输出到标准错误输出。
  - stdout：输出到标准输出。
- -qopt-report-filter=*string*:设置报告的过滤器。*string*可以为filename、routine、range等。
- -qopt-report-format=*keyword*：设置报告的格式。*keyword*可以为text和vs，分别对应纯文本和Visual Studio格式。
- -qopt-report-help：显示使用-qopt-report-phase选项时可用于报告生成的各优化阶段，并显示各级别报告的简短描述。
- -qopt-report-per-object：为各对象文件生成独立的报告文件。
- -qopt-report-phase：对生成的优化报告指明一个或多个优化阶段。*phase*可以为：cg、ipo、loop、openmp、par、pgo、tcollect、vec和all等。
- -qopt-report-routine=*substring*：让编译器对含有*substring*的子程序生成优化报告。
- -qopt-report-names=*keyword*：是否在优化报告中显示重整的或未重整的名字。*keyword*可以为：mangled和unmangled。
- -tcheck：对线程应用启用分析。
- -tcollect*[lib]*：插入测试探测调用Intel Trace Collector API。*lib*为一种Intel Trace Collector库，例如：VT、VTcs、VTmc或VTfs。
- -tcollect-filter*filename*：对特定的函数启用或禁止测试。
- -vec-report*[=n]*：设置向量化诊断信息详细程度。*n*为0（不显示）到7（最详细）。

4.4.7 OpenMP和并行处理选项

- -fmpc-privatize、-fno-mpc-privatize：是否启用针对多处理器计算环境(MPC)所有静态数据私有。
- -par-affinity=*[modifier,...]type[,permute][,offset]*：设定线程亲和性。

- modifier:可以为以下值之一:granularity=fine|thread|core、[no]respect、[no]verbose、[no]warnings、proclist=proc_list。默认为granularity=core, respect, noverbose。
- type:指示线程亲和性。此选项是必需的,并且需为以下之一:compact、disabled、explicit、none、scatter、logical、physical。默认为none。logical和physical已经过时。分别使用compact和scatter,并且没有permute值。
- permute:非负整数。当type设置为explicit、none或disabled时,不能使用此选项。默认为0。
- offset:非负整数。当type设置为explicit、none或disabled时,不能使用此选项。默认为0。

- -par-num-threads=*n*:设定并行区域内的线程数。
- -par-report*n*:设定自动并行时诊断信息的显示级别。*n*可以为0到5。
- -par-runtime-control*n*、-no-par-runtime-control:设定是否对符号循环边界的循环执行运行时检查代码。
- -par-schedule-*keyword[=n]*:设定循环迭代的调度算法。*keyword*可以为:
  - auto:由编译器或者运行时系统设定调度算法。
  - static:将迭代分割成连续块。
  - static-balanced:将迭代分割成偶数大小的块。
  - static-steal:将迭代分割成偶数大小的块,但允许线程从临近线程窃取部分块。
  - dynamic:动态获取迭代集。
  - guided:设定迭代的最小值。
  - guided-analytical:使用指数分布或动态分布分割迭代。
  - runtime:直到运行时才设定调度分割。

  *n*为每个迭代数或块大小。此设置,只能配合static、dynamic和guided使用。
- -par-threshold*n*:设定针对循环自动并行的阈值。*n*为一个在0到100间的整数,限定针对循环自动并行的阈值:
  - 如*n*为0,则循环总会被并行。
  - 如*n*为100,则循环只有在基于编译器分析应用的数据能达到预期收益时才并行。
  - 1到99为预期可能的循环加速百分比。

- -parallel：让自动并行器针对可以安全并行执行的循环生成多线程代码。
- -parallel-source-info=*n*、-no-parallel-source-info：设定当生成OpenMP或自动并行代码是否显示源位置。*n*为显示级别：
  – 0：禁止显示源位置信息。
  – 1：显示子程序名和行信息。
  – 2：显示路径、文件名、子程序名和行信息。
- -qopenmp：编译OpenMP程序。注意：在一般只能在同一个节点内的CPU上运行OpenMP程序。
- -qopenmp-lib=*type*：设定链接时使用的OpenMP运行时库。当前*type*只能设定为compat。
- -qopenmp-link=*library*：设定采用动态还是静态链接OpenMP运行时库。*library*可以为static和dynamic，分别表示静态和动态链接OpenMP运行时库。
- -qopenmp-report*n*：设定OpenMP并行器的诊断信息的显示级别。*n*可以为0、1和2。
- -qopenmp-simd、-no-qopenmp-simd：设定是否启用OpenMP SIMD编译。
- -qopenmp-stubs：使用串行模式编译OpenMP程序。
- -qopenmp-task=*model*：设定OpenMP的任务模型。*model*可以为：
  – intel：让编译接受Intel任务序列指导指令（#pragma intel_omp_taskq和#pragma intel_omp_task）。OpenMP API 3.0将被忽略。
  – omp：让编译接受OpenMP API 3.0任务序列指导指令（#pragma omp_task）。Intel任务序列指导指令将被忽略。
- -qopenmp-threadprivate=*type*：设定OpenMP线程私有的实现。*type*可以为：
  – legacy：让编译器继承使用以前Intel编译器使用的OpenMP线程私有实现。
  – compat：让编译器使用基于对每个私有线程变量应用__declspec(thread)属性的兼容OpenMP线程私有实现。

4.4.8 浮点选项

- -fast-transcendentals：让编译器使用超越函数代替，超越函数是较快但精度较低的实现。
- -fimf-absolute-error=*value[:funclist]*：定义对于数学函数返回值允许的最大绝对误差的值。*value*为正浮点数，*funclist*为函数名列表。如:-fimf-absolute-error=0.00001:sin,sinf。

- -fimf-accuracy-bits=*bits[:funclist]*：定义数学函数返回值的相对误差，包含除法及开方。*bits*为正浮点数，指明编译器应该使用的正确位数，*funclist*为函数名列表。如：-fimf-accuracy-bits=23:sin,sinf。bits与ulps之间的变换关系为：$ulps = 2^{p-1-bits}$，其中p为目标格式尾数$bits$的位数（对应单精度、双精度和长双精度分别为23、53和64）。

- -fimf-max-error=*ulps[:funclist]*：定义对于数学函数返回值的最大允许相对误差，包含除法及开方。*value*为正浮点数，指定编译器可以使用的最大相对误差，*funclist*为函数名列表，如：-fimf-max-error=4.0:sin,sinf。

- -fimf-precision*[=value[:funclist]]*：当设定使用何种数学库函数时，定义编译器应该使用的精度。*value*可以为：
  - high：等价于max-error = 0.6
  - medium：等价于max-error = 4
  - low：等价于accuracy-bits = 11（对单精度）和accuracy-bits = 26（对双精度）

  *funclist*为函数名列表，如：-fimf-precision=high:sin,sinf。

- -fma、-no-fma：是否对存在融合乘加(fused multiply-add-FMA)的目标处理器启用融合乘加。此选项只有在-x或-march参数设定CORE-AVX2或更高时才有效。

- -fp-model *keyword*：控制浮点计算的语义，*keyword*可以为：
  - precise：取消浮点数据的非值安全优化。
  - fast[=1|2]：对浮点数据启用更加激进的优化。
  - strict：启用精度和异常，禁止收缩，启用编译指示stdc和fenv_access。
  - source：四舍五入中间结果到源定义精度。
  - double：四舍五入中间结果到53-bit（双）精度。
  - extended：四舍五入中间结果到64-bit（扩展）精度。
  - [no-]except：定义严格浮点异常编译指令是否启用。

  *keyword*可以分成以下三组使用：
  - precise, fast, strict
  - source, double, extended
  - except

- -fp-port、-no-fp-port：是否对浮点操作启用四舍五入。

- -fp-speculation=*mode*：设定推测浮点操作时使用的模式。*mode*可以为：

  - fast：让编译器推测浮点操作。
  - safe：让编译器在推测浮点操作有可能存在浮点异常时停止推测。
  - strict：让编译器禁止浮点操作时推测。
  - off：与strict相同。

- -fp-trap=*mode[,mode,...]*：设置主函数的浮点异常捕获模式。*mode*可以为：
  - [no]divzero：是否启用被0除时的IEEE捕获。
  - [no]inexact：是否启用不精确结果时的IEEE捕获。
  - [no]invalid：是否启用无效操作时的IEEE捕获。
  - [no]overflow：是否启用上溢时的IEEE捕获。
  - [no]underflow：是否启用下溢时的IEEE捕获。
  - [no]denormal：是否启用非正规时的IEEE捕获。
  - all：启用上述所有的IEEE捕获。
  - none：禁止启用上述所有的IEEE捕获。
  - common：启用最常见的IEEE捕获：被0除、无效操作和上溢。

- -fp-trap-all=*mode[,mode,...]*：设置所有函数的浮点异常捕获模式。*mode*可以为：
  - [no]divzero：是否启用被0除时的IEEE捕获。
  - [no]inexact：是否启用不精确结果时的IEEE捕获。
  - [no]invalid：是否启用无效操作时的IEEE捕获。
  - [no]overflow：是否启用上溢时的IEEE捕获。
  - [no]underflow：是否启用下溢时的IEEE捕获。
  - [no]denormal：是否启用非正规时的IEEE捕获。
  - all：启用上述所有的IEEE捕获。
  - none：禁止启用上述所有的IEEE捕获。
  - common：启用最常见的IEEE捕获：被0除、无效操作和上溢。

- -ftz：赋值非常规操作结果为0。

- -mp1：提高浮点操作的精度和一致性。

- -pc*n*：设定浮点尾数精度。*n*可以为：
  - 32：四舍五入尾数到24位（单精度）。
  - 64：四舍五入尾数到53位（双精度）。

  – 80：四舍五入尾数到64位（扩展精度）。

- -prec-div、-no-prec-div：是否提高浮点除的精度。
- -prec-sqrt、-no-prec-sqrt：是否提高开根的精度。
- -rcd：启用快速浮点数到整数转换。

4.4.9 内联选项

- -gnu89-inline：设定编译器在C99模式时使用C89定义处理内联函数。
- -finline、-fno-inline：是否对__inline声明的函数进行内联，并执行C++内联。
- -finline-functions、-fno-inline-functions：对单个文件编译时启用函数内联。
- -finline-limit=*n*：设定内联函数的最大数。*n*为非负整数。
- -inline-calloc、-no-inline-calloc:是否设定编译器内联调用calloc()为调用malloc()和memset()。
- -inline-factor、-no-inline-factor：是否设定适用于所有内联选项定义的上限的比例乘法器。
- -inline-level=*n*：设定内联函数的展开级别。*n*可以为0、1、2.

4.4.10 输出、调试及预编译头文件(PCH)选项

- -c：仅编译成对象文件（.o文件）。
- -debug *[keyword]*：设定是否生成调试信息。*keyword*可以为：
  – none：不生成调试信息。
  – full或all：生成完全调试信息。
  – minimal：生成最少调试信息。
  – [no]emit_column：设定是否针对调试生成列号信息。
  – [no]expr-source-pos：设定是否在表达式粒度级别生成源位置信息。
  – [no]inline-debug-info：设定是否针对内联代码生成增强调试信息。
  – [no]macros：设定是否针对C/C++宏生成调试信息。
  – [no]pubnames：设定是否生成DWARF debug_pubnames节。
  – [no]semantic-stepping：设定是否生成针对断点和单步的增强调试信息。

  - [no]variable-locations：设定是否编译器生成有助于寻找标量局部变量的增强型调试信息。
  - extended：设定关键字值semantic-stepping和variable-locations。
  - [no]parallel：设定是否编译器生成并行调试代码指令以有助于线程数据共享和可重入调用探测。

- -g：包含调试信息。
- -g0：禁止生成符号调试信息。
- -gdwarf-*n*：设定生成调试信息时的DWARF版本，*n*可以为2、3、4。
- -o file：指定生成的文件名。
- -pch：设定编译器使用适当的预编译头文件。
- -pch-create *filename*：设定生成预编译头文件。
- -pch-dir *dir*：设定搜索预编译头文件的目录。
- -pch-use *filename*：设定使用的预编译头文件。
- -print-multi-lib：打印哪里系统库文件应该被发现。
- -S：设定编译器只是生成汇编文件但并不进行链接。

4.4.11 预处理选项

- -B*dir*：设定头文件、库文件及可执行文件的搜索路径。
- -D*name[=value]*：设定编译时的宏及其值。
- -dD：输出预处理的源文件中的#define指令。
- -dM：输出预处理后的宏定义。
- -dN：与-dD类似，但只输出的#define指令的宏名。
- -E：设定预处理时输出到标注输出。
- -EP：设定预处理时输出到标注输出，忽略#line指令。
- -gcc、-no-gcc、-gcc-sys：判定确定的GNU宏(__GNUC__、__GNUC_MINOR__和__GNUC_PATCHLEVEL__)是否定义。

- -gcc-include-dir、-no-gcc-include-dir：设定是否将gcc设定的头文件路径加入到头文件路径中。
- -H：编译时显示头文件顺序并继续编译。
- -I：设定头文件附加搜索路径。
- -icc、-no-icc：设定Intel宏(__INTEL_COMPILER)是否定义。
- -idirafter*dir*：设定*dir*路径到第二个头文件搜索路径中。
- -imacros *filename*：允许一个头文件在编译时在其它头文件前面。
- -iprefix *prefix*：指定包含头文件的参考目录的前缀。
- -iquote *dir*：在搜索的头文件路径前面增加*dir*目录以供那些使用引号而不是尖括号的文件使用。
- -isystem*dir*：附加*dir*目录到系统头文件的开始。
- -iwithprefix*dir*：附加*dir*目录到通过-iprefix引入的前缀后，并将其放在头文件目录末尾的头文件搜索路径中。
- -iwithprefixbeforex*dir*:除头文件目录*dir*放置的位置与-I声明的一样外,与-iwithprefix类似。
- -M：让编译器针对各源文件生成makefile依赖行。
- -MD：预处理和编译，生成后缀为.d包含依赖关系的输出文件。
- -MF*filename*：让编译器在一个文件中生成makefile依赖信息。
- -MG：让编译器针对各源文件生成makefile依赖行。与-M类似，但将缺失的头文件作为生成的文件。
- -MM：让编译器针对各源文件生成makefile依赖行。与-M类似，但不包含系统头文件。
- -MMD：预处理和编译，生成后缀为.d包含依赖关系的输出文件。与-M类似，但不包含系统头文件。
- -MP：让编译器对每个依赖生成伪目标。
- -MQ*target*：对依赖生成改变默认目标规则。*target*是要使用的目标规则。与-MT类似，但引用特定Make字符。
- -MT*target*：对依赖生成改变默认目标规则。*target*是要使用的目标规则。

- -nostdinc++：对C++不搜索标准目录下的头文件，而搜索其它标准目录。
- -P：停止编译处理，并将结果写入文件。
- -pragma-optimization-level=*interpretation*：指定如没有前缀指定时，采用何种优化级别编译指令解释。*interpretation*可以为：
  - Intel：Intel解释。
  - GCC：GCC解释。
- -U*name*：取消某个宏的预定义。
- -undef：取消所有宏的预定义。
- -X：从搜索路径中去除标准搜索路径。

4.4.12 C/C++语言选项

- -ansi：与gcc的-ansi选项兼容。
- -check=*keyword[, keyword...]*：设定在运行时检查某些条件。*keyword*可以为：
  - [no]conversions：设定是否在转换成较小类型时进行检查。
  - [no]stack：设定是否在堆栈帧检查。
  - [no]uninit：设定是否对未初始化变量进行检查。
- -fno-gnu-keywords：让编译器不将typeof作为一个关键字。
- -fpermissive：让编译器允许非一致性代码。
- -fsyntax-only：让编译器仅作语法检查，不生成目标代码。
- -funsigned-char：将默认字符类型变为无符号类型。
- -help-pragma：显示所有支持的编译指令。
- -intel-extensions、-no-intel-extensions：是否启用Intel C和C++语言扩展。
- -restrict、-no-restrict：设定是否采用约束限定进行指针消岐。
- -std=*val*：*val*可以为c89、c99、gnu89、gnu++89或c++0x，分别对应相应标准。
- -stdlib*[=keyword]*：设定链接时使用的C++库。*keyword*可以为：
  - libstdc++：链接使用GNU libstdc++库。

  - libc++：链接使用libc++库。

- -strict-ansi：让编译器采用严格的ANSI一致性语法。
- -x *type*：*type*可以为c、c++、c-header、cpp-output、c++-cpp-output、assembler、assembler-with-cpp或none，分别表示c源文件等，以使所有源文件都被认为是此类型的。
- -Zp*[n]*：设定结构体在字节边界的对齐。n是字节大小边界，可以为1、2、4、8和16。

4.4.13 Fortran语言选项

- -auto-scalar：INTEGER、REAL、COMPLEX和LOGICAL内在类型变量，如未声明有SAVE属性，将分配到运行时堆栈中，下次调用此函数时变量赋值。
- -allow *keyword*：设定编译器是否允许某些行为。*keyword*可以为[no]fpp_comments，声明fpp预处理器如何处理在预处理指令行中的Fortran行尾注释。
- -altparam、-noaltparam：设定是否允许不同的语法（不带括号）PARAMETER声明。
- -assume *keyword[, keyword...]*：设定某些假设。*keyword*可以为：none、[no]bscc、[no]buffered_io、[no]buffered_stdout、[no]byterecl、[no]cc_omp、[no]dummy_aliases、[no]fpe_summary、[no]ieee_fpe_flags、[no]minus0、[no]old_boz、[no]old_ldout_format、[no]old_logical_ldio、[no]old_maxminloc、[no]old_unit_star、[no]old_xor、[no]protect_constants、[no]protect_parens、[no]realloc_lhs、[no]source_include、[no]std_intent_in、[no]std_minus0_rounding、[no]std_mod_proc_name、[no]std_value、[no]underscore、[no]2underscores、[no]writeable-strings等
- -ccdefault *keyword*：设置文件显示在终端上时的回车类型。*keyword*可以为：
  - none：设定编译器使用无回车控制预处理。
  - default：设定编译器使用默认回车控制设定。
  - fortran：设定编译器使用通常的第一个字符的Fortran解释。如字符0使得在输出一个记录时先输出一个空行。
  - list：设定编译器记录之间出输出换行。
- -check=*keyword[, keyword...]*：设定在运行时检查某些条件。*keyword*可以为：
  - none：禁止所有检查。
  - [no]arg_temp_created：设定是否在子函数调用前检查实参。

  - [no]assume：设定是否在测试在ASSUME指令中的标量布尔表达式为真，或在ASSUME_ALIGNED指令中的地址对齐声明的类型边界时进行检查。
  - [no]bounds：设定是否对数组下标和字符子字符串表达式进行检查。
  - [no]format：设定是否对格式化输出的数据类型进行检查。
  - [no]output_conversion：设定是否对在指定的格式描述域内的数据拟合进行检查。
  - [no]pointers：设定是否对存在一些分离的或未初始化的指针或为分配的可分配目标时进行检查。
  - [no]stack：设定是否在堆栈帧检查。
  - [no]uninit：设定是否对未初始化变量进行检查。
  - all：启用所有检查。

- -cpp：对源代码进行预处理，等价于-fpp。
- -extend-source*[size]*：指明固定格式的Fortran源代码宽度，*size*可为72、80和132。也可直接用-72、-80和-132指定，默认为72字符。
- -fixed：指明Fortran源代码为固定格式，默认由文件后缀设定格式类别。
- -free：指明Fortran源程序为自由格式，默认由文件后缀设定格式类别。
- -nofree：指明Fortran源程序为固定格式。
- -implicitnone：指明默认变量名为未定义。建议在写程序时添加implicit none语句，以避免出现由于默认类型造成的错误。
- -names *keyword*：设定如何解释源代码的标志符和外部名。*keyword*可以为：
  - lowercase：让编译器忽略标识符的大小写不同，并转换外部名为小写。
  - uppercase：让编译器忽略标识符的大小写不同，并转换外部名为大写。
  - as_is：让编译器区分标识符的大小写，并保留外部名的大小写。
- -pad-source、-nopad-source：对固定格式的源码记录是否采用空白填充行尾。
- -stand *keyword*：以指定Fortran标准进行编译，编译时显示源文件中不符合此标准的信息。*keyword*可为f03、f90、f95和none，分别对应显示不符合Fortran 2003、90、95的代码信息和不显示任何非标准的代码信息，也可写为-std*keyword*，此时*keyword*不带f，可为03、90、95。
- -standard-semantics：设定编译器的当前Fortran标准行为是否完全实现。
- -syntax-only：仅仅检查代码的语法错误，并不进行其它操作。

- -wrap-margin、-no-wrap-margin：提供一种在Fortran列表输出时禁止右边缘包装。
- -us：编译时给外部用户定义的函数名添加一个下划线，等价于-assume underscore，如果编译时显示_函数找不到时也许添加此选项即可解决。

4.4.14 数据选项

- 共有选项
  - -fcommon、-fno-common：设定编译器是否将common符号作为全局定义。
  - -fpic、-fno-pic：是否生成位置无关代码。
  - -fpie：类似-fpic生成位置无关代码，但生成的代码只能链接到可执行程序。
    - -gcc：定义GNU宏。
    - -no-gcc：取消定义GNU宏。
    - -gcc-sys：只有在编译系统头文件时定义GNU宏。
  - -mcmodel=*mem_model*:设定生成代码和存储数据时的内存模型。*mem_model*可以为：
    - small：让编译器限制代码和数据使用最开始的2GB地址空间。对所有代码和数据的访问可以使用指令指针(IP)相对地址。
    - medium：让编译器限制代码使用最开始的2GB地址空间，对数据没有内存限制。对所有代码的访问可以使用指令指针（IP）相对地址，但对数据的访问必须采用绝对地址。
    - large：对代码和数据不做内存限制。所有访问都得使用绝对地址。
  - -mlong-double-*n*：覆盖掉默认的长双精度数据类型配置。*n*可以为：
    - 64：设定长双精度数据为64位。
    - 80：设定长双精度数据为80位。
- C/C++专有选项
  - -auto-ilp32：让编译器分析程序设定能否将64位指针缩成32位指针，能否将64位长整数缩成32位长整数。
  - -auto-p32：让编译器分析程序设定能否将64位指针缩成32位指针。
  - -check-pointers=*keyword*：设定编译器是否检查使用指针访问的内存边界。*keyword*可以为：
    - none：禁止检查，此为默认选项。
    - rw：检查通过指针读写的内存边界。
    - write：只检查通过内存写的内存边界。

  - -check-pointers-dangling*keyword*：设定编译器是否对悬挂（dangling）指针参考进行检查。*keyword*可以为：
    * none：禁止检查悬挂指针参考，此为默认选项。
    * heap：检查heap的悬挂指针参考。
    * stack：检查stack的悬挂指针参考。
    * all：检查上述所有的悬挂指针参考。
  - -fkeep-static-consts、-fno-keep-static-consts：设定编译器是否保留在源文件中没有参考的变量分配。

- Fortran专有选项
  - -convert *[keyword]*:转换无格式数据的类型,比如*keyword*为big_endian和little_endian时,分别表示无格式的输入输出为big_endian和little_endian格式，更多格式类型，请看编译器手册。
  - -double-size *size*：设定DOUBLE PRECISION和DOUBLE COMPLEX声明、常数、函数和内部函数的默认KIND。*size*可以为64或128,分别对应KIND=8和KIND=16。
  - -dyncom *"common1,common2,..."*：对指定的common块启用运行时动态分配。
  - -fzero-initialized-in-bss、-fno-zero-initialized-in-bss：设定编译器是否将数据显式赋值为0的变量放置在DATA块内。
  - -intconstant、-nointconstant：让编译器使用FORTRAN 77语法设定整型常数的KIND参数。
  - -integer-size *size*：设定整型和逻辑变量的默认KIND。*size*可以为16、32或64，分别对应KIND=2、KIND=4或KIND=8。
  - -no-bss-init：让编译器将任何未初始化变量和显式初始化为0的变量放置在DATA块。默认不启用，放置在BSS块。
  - -real-size *size*：设定实型变量的默认KIND。*size*可以为32、64或18，分别对应KIND=4、KIND=8或KIND=16。
  - -save:强制变量值存储在静态内存中。此选项保存递归函数和用AUTOMATIC声明的所有变量（除本地变量外）在静态分配中，下次调用时可继续用。默认为-auto-scalar，内在类型INTEGER、REAL、COMPLEX和LOGICAL变量分配到运行时堆栈中。
  - -zero、-nozero：是否将所有保存的但未初始化的内在类型INTEGER、REAL、COMPLEX或LOGICAL的局部变量值初始为0。

4.4.15 编译器诊断选项

- -diag-*type=diag-list*：控制显示的诊断信息。*type*可以为：

- enable：启用一个或一组诊断信息。
- disable：禁用一个或一组诊断信息。
- error：让编译器将诊断信息变为错误。
- warning：让编译器将诊断信息变成警告
- remark：让编译器将诊断信息变为备注。

*diag-list*可为：driver、port-win、thread、vec、par 、openmp 、warn 、error 、remark 、cpu-dispatch 、id[,id,...] 、tag[,tag,...]等。

- -traceback、-notraceback：编译时在对象文件中生成额外的信息使得在运行出错时可以提供源文件回朔信息。
- -w：编译时不显示任何警告，只显示错误。
- -w*n*：设置编译器生成的诊断信息级别。*n*可以为：
  - 0：对错误生成诊断信息，屏蔽掉警告信息。
  - 1：对错误和警告生成诊断信息。此为默认选项。
  - 2：对错误和警告生成诊断信息，并增加些额外的警告信息。
  - 3：对备注、错误和警告生成诊断信息，并在级别2的基础上再增加额外警告信息。建议对产品使用此级别。
  - 4：在级别3的基础上再增加一些警告和备注信息，这些增加的信息一般可以安全忽略。
- -Wabi、-Wno-abi：设定生成的代码不是C++ ABI兼容时是否显示警告信息。
- -Wall：编译时显示警告和错误信息。
- -Wbrief：采用简短方式显示诊断信息。
- -Wcheck：让编译器在对特定代码在编译时进行检查。
- -Werror：将所有警告信息变为错误信息。
- -Werror-all将所有警告和备注信息变为错误信息。
- -Winline：设定编译器显示哪些函数被内联，哪些未被内联。
- -Wunused-function、-Wno-unused-functio：设定是否在声明的函数未使用时显示警告信息。
- -Wunused-variable、-Wno-unused-variable：设定是否在声明的变量未使用时显示警告信息。

4.4.16 兼容性选项

- -f66：FORTRAN程序特有。使用FORTRAN 66标准，默认为使用 Fortran 95标准。
- -f77rtl、-nof77rtl：FORTRAN程序特有。 是否使用FORTRAN 77运行时行为，默认为使用Intel Fortran运行时行为。控制以下行为：
  - 当unit没有与一个文件对应时，一些INQUIRE说明符将返回不同的值：
    * NUMBER= 返回0；
    * ACCESS= 返回'UNKNOWN'；
    * BLANK= 返回'UNKNOWN'；
    * FORM= 返回'UNKNOWN'。
  - PAD= 对格式化输入默认为"NO'。
  - NAMELIST和列表输入的字符串必需用单引号或双引号分隔。
  - 当处理NAMELIST输入时：
    * 每个记录的第一列被忽略。
    * 出现在组名前的'$'或'&'必须在输入记录的第二列。
  - -fpscomp [keyword[, keyword...]]、-nofpscomp：FORTRAN程序特有。设定是否某些特征与Intel®Fortran或Microsoft* Fortran PowerStation兼容。*keyword*可以为：
    * none：没有选项需要用于兼容性。
    * [no]filesfromcmd：设定当OPEN声明中FILE=说明符为空时的兼容性。
    * [no]general：设定当Fortran PowerStation和Intel®Fortran语法存在不同时的兼容性。
    * [no]ioformat：设定列表格式和无格式IO时的兼容性。
    * [no]libs：设定可移植性库是否传递给链接器。
    * [no]ldio_spacing：设定是否在运行时在数值量后字符值前插入一个空白。
    * [no]logicals：设定代表LOGICAL值的兼容性。
    * all：设定所有选项用于兼容性。
- -fabi-version=*n*：设定使用指定版本的ABI实现。*n*可以为：
  - 0：使用最新的ABI实现。
  - 1：使用gcc 3.2和gcc 3.3使用的ABI实现。
  - 2：使用gcc 3.4及更高的gcc中使用的ABI实现。
- -gcc-name=*name*：设定使用的gcc编译器的名字。
- -gxx-name*name*：设定使用的g++编译器的名字。

4.4.17 链接和链接器选项

- -Bdynamic：在运行时动态链接所需要的库。
- -Bstatic：静态链接用户生成的库。
- -cxxlib*[=dir]*、-cxxlib-nostd、-no-cxxlib：设定是否使用gcc提供的C++运行时库及头文件。*dir*为gcc二进制及库文件的顶层目录。
- -I*dir*：指明头文件的搜索路径。
- -L*dir*：指明库的搜索路径。
- -l*string*：指明所需链接的库名，如库名为libxyz.a，则可用-lxyz指定。
- -no-libgcc：禁止使用特定gcc库链接。
- -nodefaultlibs：禁止使用默认库链接。
- -nostartfiles：禁止使用标准启动文件链接。
- -nostdlib：禁止使用标准启动和库文件链接。
- -pie、-no-pie：设定编译器是否生成需要链接进可执行程序的位置独立代码
- -pthread：对多线程启用pthreads库。
- -shared：生成共享对象文件而不是可执行文件，必须在编译每个对象文件时使用-fpic选项。
- -shared-intel：动态链接Intel库。
- -shared-libgcc：动态链接GNU libgcc库。
- -static：静态链接所有库。
- -static-intel：静态链接Intel库。
- -static-libgcc：静态链接GNU libgcc库。
- -static-libstdc++：静态链接GNU libstdc++库。
- -u *symbol*：设定指定的符号未定义。
- -v：显示驱动工具编译信息。
- -Wa,*option1[,option2,...]*：传递参数给汇编器进行处理。
- -Wl,*option1[,option2,...]*：传递参数给链接器进行处理。

- -Wp,*option1[,option2,...]*：传递参数给预处理器。
- -Xlinker *option*：将option信息传递给链接器。

4.4.18 其它选项

- -help *[category]*：显示帮助。
- -sox*[=keyword[,keyword]]*、-no-sox：设定是否让编译时在生成的可执行文件中保存编译选项和版本等信息，也可以指定是否保存子程序等信息。
  - inline：包含在各目标文件中的内联子程序名。
  - profile：包含编译时采用-prof-use的子程序列表，以及存储概要信息的.dpi文件名和指明使用的和忽略的概要信息。

  存储的信息可以使用以下方法查看：
  - *objdump -sj .comment a.out*
  - *strings -a a.out | grep comment:*
- -V：显示版本信息。
- –version：显示版本信息。
- -watch*[=keyword[, keyword...]]*、-nowatch：设定是否在控制台显示特定信息。*keyword*可以为：
  - none：禁止cmd和source。
  - [no]cmd：设定是否显示驱动工具命令及执行。
  - [no]source：设定是否显示编译的文件名。
  - all：启用cmd和source。

5 PGI C/C++ Fortran编译器

目前尚未真正配置，等以后配置

5.1 PGI C/C++ Fortran编译器简介

PGI C/C++ Fortran编译器是一种针对多种CPU与操作系统的高性能编译器，可用于开发复杂且要进行大量计算的程序。当前安装的版本为2016.7和2014.10，分别安装在*/opt/pgi-16.7/linux86-64/16.7*、*/opt/pgi/linux86-64/14.10*。安装在*/opt/intel*，可用*module avail*查看，用*moudle load 模块名*使用，或在自己的*~/.bashrc*之类环境设置文件中添加以下代码设置：

```
PATH=/opt/pgi/linux86-64/14.10/bin:$PATH
MANPATH=$MANPATH:/opt/pgi/linux86-64/14.10/man
export PATH MANPATH
```

PGI编译器编译C、C++、Fortran 77源程序的命令分别为*pgcc*、*pgCC|pgc++*[4]和*pgf77*，编译Fortran 90（为了描述方便，本手册中将Fortran 90、95、2003、2008标准统称为Fortran 90）的源程序的命令有*pgf90*、*pgf901*、*pgf902*、*pgf90_ex*、*pgf95*和*pgfortran*。

编译工具	语言或函数	命令
PGF77	ANSI FORTRAN 77	pgf77
PGFORTRAN	ISO/ANSI Fortran 2003	pgfortran、pgf90、pgf95
PGCC	ISO/ANSI C11 and K&R C	pgcc
PGC++	ISO/ANSI C++14 with GNU compatibility	pgc++
PGDBG	Source code debugger	pgdbg
PGPROF	Performance profiler	pgprof

官方手册目录：*安装目录/doc*。

5.2 编译错误

编译时的出错信息类似以下：

```
PGF90-S-0034-Syntax error at or near * (NOlihm.f90: 2)
```

编译错误的格式为：

[4]2016版为pgc++，之前版本为pgCC

大写的编译命令-严重级别-错误编号-解释，含指明位置(文件名: 行号)

错误严重级别分为：

- I：信息
- W：警告
- S：严重
- F：致命
- V：其它

Fortran程序编译错误解释，参见：PGI Compiler Reference Guide->Chapter 9. MESSAGES->9.3. Fortran Compiler Error Messages->9.3.2. Message List

5.3 Fortran程序运行错误

Fortran程序运行时错误解释，参见：PGI Compiler Reference Guide->Chapter 9. MESSAGES->9.4. Fortran Run-time Error Messages->9.4.2. Message List

5.4 PGI C/C++编译器重要编译选项

PGI编译器选项非常多，下面仅仅是列出一些本人认为常用的关于编译C程序的*pgcc*命令的重要选项。编译C++程序的*pgc++|pgCC*命令有稍微不同，建议仔细查看PGI相关资料。建议仔细查看编译器手册中关于程序优化的部分，多加测试，选择适合自己程序的编译选项以提高性能。

5.4.1 一般选项

- -#：显示编译器、汇编器、链接器的调用信息。
- -c：仅编译成对象文件（.o文件）。
- -defaultoptions和-nodefaultoptions：是否使用默认选项，默认为使用。
- -flags：显示所有可用的编译选项。
- -help*[=option]*：显示帮助信息，option可以为groups、asm、debug、language、linker、opt、other、overall、phase、phase、prepro、suffix、switch、target和variable。

- -Minform=*level*：控制编译时错误信息的显示级别。level可以为fatal、file、severe、warn、inform，默认为-Minform=warn。
- -noswitcherror：显示警告信息后，忽略未知命令行参数并继续进行编译。默认显示错误信息并且终止编译。
- -o file：指定生成的文件名。
- -show：显示现有pgcc命令的配置信息。
- -silent：不显示警告信息，与-Minform=severe等同。
- -v：详细模式，在每个命令执行前显示其命令行。
- -V：显示编译器版本信息。
- -w：编译时不显示任何警告，只显示错误。

5.4.2 优化选项

- -fast：编译时选择针对目标平台的普通优化选项。用*pgcc -fast -help*可以查看等价的开关。优化级别至少为O2，参看-O选项。
- -fastsse：对支持SSE和SSE2指令的CPU（如Intel Xeon CPU）编译时选择针对目标平台的优化选项。用*pgcc -fastsse -help*可以查看等价的开关，优化级别至少为O2，参看-O选项。
- -fpic或-fPIC：编译器生成地址无关代码，以便可用于生成共享对象文件（动态链接库）。
- -Kpic或-KPIC：与-fpic或-fPIC相同，为了与其余编译器兼容。
- -Minfo *[=option[,option,...]]*：显示有用信息到标准错误输出，选项可为all、autoinline、inline、ipa、loop或opt、mp、time或stat。
- -Mipa*[=option[,option,...]]*和-Mnoipa：启用指定选项的过程间分析优化，默认为-Mnoipa。
- -Mneginfo=*option[,option...]*：使编译器显示为什么特定优化没有实现的信息。选项包括concur、loop和all。
- -Mnoopenmp：当使用-mp选项时，忽略OpenMP并行指令。
- -Mnosgimp：当使用-mp选项时，忽略SGI并行指令。

- -Mpfi：生成概要导向工具，此时将会包含特殊代码收集运行时的统计信息以用于子序列编译。-Mpfi必须在链接时也得使用。当程序运行时，会生成概要导向文件pgfi.out。
- -Mpfo：启用概要导向优化，此时必须在当前目录下有概要文件pgfi.out。
- -Mprof*[=option[,option,...]]*：设置性能功能概要选项。此选项可使得结果执行生成性能概要，以便PGPROF性能概要器分析。
- -mp*[=option]*：打开对源程序中的OpenMP并行指令的支持。
- -O*[level]*：设置优化级别。level可设为0、1、2、3、4，其中4与3相同。
- -pg：使用gprof风格的基于抽样的概要刨析。

5.4.3 调试选项

- -g：包含调试信息。

5.4.4 预处理选项

- -C：预处理时保留C源文件中的注释。
- -D*name[=def]*：预处理时定义宏name为def。
- -dD：打印源文件中已定义的宏及其值到标准输出。
- -dI：打印预处理中包含的所有文件信息，含文件名和定义时的行号。
- -dM：打印预处理时源文件已定义的宏及其值，含定义时的文件名和行号。
- -dN：与-dD类似，但只打印源文件已定义的宏，而不打印宏值。
- -E：预处理每个.c文件，将结果发送给标准输出，但不进行编译、汇编或链接等操作。
- -I*dir*：指明头文件的搜索路径。
- -M：打印make的依赖关系到标准输出。
- -MD：打印make的依赖关系到文件file.d，其中file是编译文件的根名字。
- -MM：打印make的依赖关系到标准输出，但忽略系统头文件。
- -MMD：打印make的依赖关系到文件file.d，其中file是编译的文件的根名字，但忽略系统头文件。

- -P：预处理每个文件，并保留每个file.c文件预处理后的结果到file.i。
- -U*name*：去除预处理中的任何name的初始定义。

5.4.5 链接选项

- -Bdynamic：在运行时动态链接所需的库。
- -Bstatic：静态链接所需的库。
- -Bstatic_pgi：动态链接系统库时静态链接PGI库。
- -g77libs：允许链接GNU *g77*或*gcc*命令生成的库。
- -l*string*：指明所需链接的库名。如库为libxyz.a，则可用-lxyz指定。
- -L*dir*：指明库的搜索路径。
- -m：显示链接拓扑。
- -Mrpath和-Mnorpath:默认为-rpath,以给出包含PGI共享对象的路径。用-Mnorpath可以去除此路径。
- -pgf77libs：链接时添加pgf77运行库，以允许混合编程。
- -r：生成可以重新链接的对象文件。
- -R*directory*：对共享对象文件总搜索directory目录。
- -pgf90libs：链接时添加pgf90运行库，以允许混合编程。
- -shared:生成共享对象而不是可执行文件,必须在编译每个对象文件时使用-fpic选项。
- -soname*name*：生成共享对象时，用内在的DT_SONAME代替指定的name。
- -u*name*：传递给链接器，以生成未定义的引用。

5.4.6 C/C++语言选项

- -B：源文件中允许C++风格的注释，指的是以//开始到行尾内容为注释。除非指定-C选项，否则这些注释被去除。
- -c8x或-c89：对C源文件采用C89标准。
- -c9x或-c99：对C源文件采用C99标准。

5.4.7 Fortran语言选项

- -byteswapio或-Mbyteswapio：对无格式Fortran数据文件在输入输出时从大端（big-endian)到小端(little-endian)交换比特，或者相反。此选项可以用于读写Sun或SGI等系统中的无格式的Fortran数据文件。
- -i2：将INTEGER变量按照2比特处理。
- -i4：将INTEGER变量按照4比特处理。
- -i8：将默认的INTEGER和LOGICAL变量按照4比特处理。
- -i8storage：对INTEGER和LOGICAL变量分配8比特。
- -Mallocatable*[=95|03]*：按照Fortran 95或2003标准分配数组。
- -Mbackslash和-Mnobackslash：将反斜线(\)当作正常字符（非转义符）处理，默认为-Mnobackslash。-Mnobackslash导致标准的C反斜线转义序列在引号包含的字串中重新解析。-Mbackslash则导致反斜线被认为和其它字符一样。
- -Mextend：设置源代码的行宽为132列。
- -Mfixed、-Mnofree和-Mnofreeform：强制对源文件按照固定格式进行语法分析，默认.f或.F文件被认为固定格式。
- -Mfree和-Mfreeform：强制对源文件按照自由格式进行语法分析，默认.f90、.F90、.f95或.F95文件被认为自由格式。
- -Mi4和-Mnoi4:将INTEGER看作INTEGER*4。-Mnoi4将INTEGER看作INTEGER*2。
- -Mnomain：当链接时，不包含调用Fortran主程序的对象文件。
- -Mr8和-Mnor8：将REAL看作DOUBLE PRECISION，将实(REAL)常数看作双精度(DOUBLE PRECISION)常数。默认为否。
- -Mr8intrinsics *[=float]*和-Mnor8intrinsics:将CMPLX看作DCMPLX,将REAL看作DBLE。添加float选项时，将FLOAT看作DBLE。
- -Msave和-Mnosave：是否将所有局部变量添加SAVE声明，默认为否。
- -Mupcase和-Mnoupcase：是否保留名字的大小写。-Mnoupcase导致所有名字转换成小写。注意，如果使用-Mupcase，那么变量名X与变量名x不同，并且关键字必须为小写。
- -Mcray=*pointer*：支持Cray指针扩展。

- -module *directory*：指定编译时保存生成的模块文件的目录。
- -r4：将DOUBLE PRECISION变量看作REAL。
- -r8：将REAL变量看作DOUBLE PRECISION。

5.4.8 平台相关选项

- -Kieee和-Knoieee：浮点操作是否严格按照IEEE 754标准。使用-Kieee时一些优化处理将被禁止，并且使用更精确的数值库。默认为-Knoieee，将使用更快的但精确性低的方式。
- -Ktrap=*[option,[option]...]*：控制异常发生时CPU的操作。选项可为divz、fp、align、denorm、inexact、inv、none、ovf、unf，默认为none。
- -Msecond_underscore和-Mnosecond_underscore：是否对已有_的Fortran函数名添加第二个_。与*g77*编译命令兼容时使用，因为*g77*默认符号后添加第二个_。
- -mcmodel=*small|medium*：使内存模型是否限制对象小于2GB(small)或允许数据块大于2GB(medium)。medium时暗含-Mlarge_arrays选项。
- -tp *target*：target可以为haswell等，默认与编译时的平台一致。

6 GNU C/C++ Fortran编译器

6.1 GNU C/C++ Fortran编译器简介

GNU C/C++ Fortran(GCC)编译器为系统自带的编译器，当前自带的版本为4.8.5，还装有9.2.0等。默认为4.8.5版本，用户无需特殊设置即可使用。GNU编译器编译C、C++源程序的命令分别*gcc*和*g++*；LSgfortran可以直接编译Fortran 77、90源程序。

6.2 编译错误

C/C++程序编译时错误信息类似：

```
netlog.c: In function 'main':
netlog.c:84:7: error: 'for' loop initial declarations are only allowed in C99 mode
netlog.c:84:7: note: use option -std=c99 or -std=gnu99 to compile your code
```

编译错误的格式为：

- 源文件名: 函数中
- 源文件名:行数:列数:错误类型:具体说明
- 源文件名:行数:列数:注解:解决办法

Fortran程序编译时错误信息类似：

```
NOlihm.f90:146.14:

  n2nd=0; npr=0
              1
Error: Symbol 'npr' at (1) has no IMPLICIT type
```

编译错误的格式为：

- 源文件名:行数:列数:
- 源文件代码
- 1指示出错位置
- 错误类型: 具体说明

6.3 GNU C/C++编译器GCC重要编译选项

GNU编译器GCC是Linux系统自带的编译器，系统自带的版本为4.8.5，另外还装有9.2.0等，选项非常多，下面仅仅是列出一些针对4.8.5本人认为常用的重要选项，建议仔细看GCC相关资料。

建议仔细查看编译器手册中关于程序优化的部分，多加测试，选择适合自己程序的编译选项以提高性能。

6.3.1 控制文件类型的选项

- -x language：明确指定而非让编译器判断输入文件的类型。language可为：
  - c、c-header、c-cpp-output
  - c++、c++-header、c++-cpp-output
  - objective-c、objective-c-header、objective-c-cpp-output
  - objective-c++、objective-c++-header、objective-c++-cpp-output
  - assembler、assembler-with-cpp

  - ada
  - f95、f95-cpp-input
  - java
  - treelang

  当language为none时，禁止任何明确指定的类型，其类型由文件名后缀设定。

- -c：仅编译成对象文件（.o文件），并不进行链接。
- -o file：指定生成的文件名。
- -v：详细模式，显示在每个命令执行前显示其命令行。
- -###：显示编译器、汇编器、链接器的调用信息但并不进行实际编译，在脚本中可以用于捕获驱动器生成的命令行。
- –help：显示帮助信息。
- –target-help：显示目标平台的帮助信息。
- –version：显示编译器版本信息。

6.3.2 C/C++语言选项

- -ansi：C模式时，支持所有ISO C90指令。在C++模式时，去除与ISO C++冲突的GNU扩展。
- -std=*val*：控制语言标准，可以为c89、iso9899:1990、iso9899:199409、c99、c9x、iso9899:1999、iso9899:199x、gnu89、gnu99、gnu9x、c++98、gnu++98。
- -B：在源文件中允许C++风格的注释，指的是以//开始到行尾内容为注释。除非指定-C选项，否则这些注释被去除。
- -c8x或-c89：对C源文件采用C89标准。
- -c9x或-c99：对C源文件采用C99标准。

6.3.3 Fortran语言选项

- -fconvert=*conversion*：指定对无格式Fortran数据文件表示方式，其值可以为：native，默认值；swap，在输入输出时从大端（big-endian）到小端（little-endian）交换比特，或者相反；big-endian，用大端方式读写；little-endian，用小端方式读写。
- -ff2c：与*g77*和*f2c*命令生成的代码兼容。

- -ffree-form和-ffixed-form：声明源文件是自由格式还是固定格式，默认从Fortran 90起的源文件为自由格式，之前的Fortran 77等的源文件为固定格式。
- -fdefault-double-8：设置DOUBLE PRECISION类型为8比特。
- -fdefault-integer-8：设置INTEGER和LOGICAL类型为8比特。
- -fdefault-real-8：设置REAL类型为8比特。
- -fno-backslash：将反斜线(\)当作正常字符（非转义符）处理。
- -fno-underscoring：不在名字后添加_。注意：*gfortran*默认行为与*g77*和*f2c*不兼容，为了兼容需要加-ff2c选项。除非使用者了解与现有系统环境的集成，否则不建议使用-fno-underscoring选项。
- -ffixed-line-length-*n*：设置固定格式源代码的行宽为n。
- -ffree-line-length-*n*：设置自由格式源代码的行宽为n。
- -fimplicit-none：禁止变量的隐式声明，所有变量都需要显式声明。
- -fmax-identifier-length=*n*：设置名称的最大字符长度为n，Fortran 95和200x的长度分别为31和65。
- -fno-automatic：将每个程序单元的本地变量和数组声明具有SAVE属性。
- -fcray-pointer：支持Cray指针扩展。
- -fopenmp：编译OpenMP并行程序。
- -M*dir*和-J*dir*：指定编译时保存生成的模块文件目录。
- -fsecond-underscore：默认*gfortran*对外部函数名添加一个_，如果使用此选项，那么将添加两个_。此选项当使用-fno-underscoring选项时无效。此选项当使用-ff2c时默认启用。
- -std=*val*：指明Fortran标准，val可以为f95、f2003、legacy。
- -funderscoring：对外部函数名没有_的加_，以与一些Fortran编译器兼容。

6.3.4 警告选项

- -fsyntax-only：仅仅检查代码的语法错误，并不进行其它操作。
- -w：编译时不显示任何警告，只显示错误。
- -Wfatal-errors：遇到第一个错误就停止，而不尝试继续运行显示更多错误信息。

6.3.5 调试选项

- -g：包含调试信息。
- -ggdb：包含利用gbd调试时所需要的信息。

6.3.6 优化选项

- -O*[level]*：设置优化级别。优化级别level可以设置为0、1、2、3、s。

6.3.7 预处理选项

- -C：预处理时保留C源文件中的注释。
- -D *name*：预处理时定义宏name的值为1。
- -D *name=def*：预处理时定义name为def。
- -U *name*：预处理时去除的任何name初始定义。
- -undef：不预定义系统或GCC声明的宏，但标准预定义的宏仍旧被定义。
- -dD：显示源文件中定义的宏及其值到标准输出。
- -dI：显示预处理中包含的所有文件，包括文件名和定义时的行号信息。
- -dM：显示预处理时源文件中定义的宏及其值，包括定义时文件名和行号。
- -dN：与-dD类似，但只显示源文件中定义的宏，而不显示宏值。
- -E：预处理各.c文件，将结果发给标准输出，不进行编译、汇编或链接。
- -I*dir*：指明头文件的搜索路径。
- -M：打印make的依赖关系到标准输出。
- -MD：打印make的依赖关系到文件file.d，其中file是编译文件的根名字。
- -MM：打印make的依赖关系到标准输出，但忽略系统头文件。
- -MMD：打印make的依赖关系到文件file.d，其中file是编译的文件的根名字，但忽略系统头文件。
- -P：预处理每个文件，并保留每个file.c文件预处理后的结果到file.i。

6.3.8 链接选项

- -pie：在支持的目标上生成地址无关的可执行文件。
- -s：从可执行文件中去除所有符号表。
- -rdynamic：添加所有符号表到动态符号表中。
- -static：静态链接所需的库。
- -shared：生成共享对象文件而不是可执行文件，必须在编译每个对象文件时使用-fpic选项。
- -shared-libgcc：使用共享libgcc库。
- -static-libgcc：使用静态libgcc库。
- -u *symbol*：确保符号symbol未定义，强制链接一个库模块来定义它。
- -I*dir*：指明头文件的搜索路径。
- -l*string*：指明所需链接的库名，如库为libxyz.a，则可用-lxyz指定。
- -L*dir*：指明库的搜索路径。
- -B*dir*：设置寻找可执行文件、库、头文件、数据文件等路径。

6.3.9 i386和x86-64平台相关选项

- -mtune=*cpu-type*:设置优化针对的CPU类型,可为:generic、core2、opteron、opteron-sse3、bdver1、bdver2等，bdver1为针对本系统AMD Opteron CPU的。
- -march=*cpu-type*：设置指令针对的CPU类型，CPU类型与上行中一样。
- -mieee-fp和-mno-ieee-fp：浮点操作是否严格按照IEEE标准。

6.3.10 约定成俗选项

- -fpic：生成地址无关的代码以用于共享库。
- -fPIC：如果目标机器支持，将生成地址无关的代码。
- -fopenmp：编译OpenMP并行程序。
- -fpie和-fPIE：与-fpic和-fPIC类似，但生成的地址无关代码，只能链接到可执行文件中。

Part VII

MPI并行程序编译及运行

本系统的通信网络由Mellanox QM8700 HDR200交换机和ConnectX-6 HDR100网卡组成的100Gbps高速计算网络及1Gbps千兆以太网两套网络组成。InfiniBand网络相比千兆以太网具有高带宽、低延迟的特点，通信性能比千兆以太网要高很多，建议使用。

本系统安装有多种MPI实现，主要有：HPC-X（Mellanox官方推荐）、Intel MPI（不建议使用，特别是2019版）和Open MPI，并可与不同编译器相互配合使用，安装目录分别在*/opt/hpcx*、*/opt/intel*和*/opt/openmpi*[5]。

用户可以运行*module apropos MPI*或*module avail*查看可用MPI环境，可用类似命令设置所需的MPI环境：*module load hpcx/hpcx-intel-2019.update5*，使用此命令有时需要手动加载对应的编译器等版本，比如报：

```
error while loading shared libraries: libimf.so: cannot open shared object
file: No such file or directory
```

则需要加载对应的Intel编译器，比如*module load intel/2019.update5*。

注意：Mellanox HDR是比较新的高速计算网络，老的MPI环境对其支持不好。

7 MPI并行程序的编译

7.1 HPC-X ScalableHPC工具集

Mellanox HPC-X ScalableHPC工具集是综合的软件包，含有MPI及SHMEM/PGAS通讯库。HPC-X ScalableHPC还包含这些库之上的用于提升性能和扩展性的多种加速包，包括加速点对点通信的UCX(Unified Communication X)、加速MPI/PGAS中集合操作的FCA(Fabric Collectives Accelerations)。这些全特性的、经完备测试的及打包好的工具集使得MPI和SHMEM/PGAS程序获得高性能、扩展性和效率，且保证了在Mellanox互连系统中这些通信库经过了全优化。

Mellanox HPC-X ScalableHPC工具集利用了基于Mellanox硬件的加速引擎，可以最大化基于MPI和SHMEM/PGAS的应用性能。这些应用引擎是Mellanox网卡（CORE-Direct引擎，硬件标记匹配(Tag Matching)等）和交换机（如Mellanox SHARP加速引擎）解决方案的一部分。Mellanox可扩展的分层聚合和归约协议(Scalable Hierarchical Aggregation and Reduction Protocol, SHARP)技术通过将集合操作从CPU端卸载到交换机网络端，通过去除在端到端之间发送多次数据的的需要，大幅提升了MPI操作性能。

[5] 具有不同版本的Open MPI与编译器的组合。

这种创新性科技显著降低了MPI操作时间，释放了重要的CPU资源使其用于计算而不是通信，且降低了到达聚合节点时通过网络的数据量。

HPC-X主要特性如下：

- 完整的MPI、PGAS/SHMEM包，且含有Mellanox UCX和FCA加速引擎
- 兼容MPI 3.2标准
- 兼容OpenSHMEM 1.4标准
- 从MPI进程将集合通信从CPU卸载到Mellanox网络硬件上
- 利用底层硬件体系结构最大化提升应用程序性能
- 针对Mellanox解决方案进行了全优化
- 提升应用的可扩展性和资源效率
- 支持RC、DC和UD等多种传输
- 节点内共享内存通信
- 带消息条带的多轨支持
- 支持GPU-direct的CUDA

HPC-X环境：

- HPC-X CUDA支持：
  – HPC-X默认是对于单线程模式优化的，这支持GPU和无GPU模式。
  – HPC-X是基于CUDA 10.1编译的，由于CUDA 10.1不支持比GCC v8新的，因此对于基于v8之后的GCC编译的，不支持CUDA。
- HPC-X多线程支持 - hpcx-mt：
  – 该选项启用所有多线程支持。

HPC-X MPI是Open MPI的一种高性能实现，利用Mellanox加速能力且无缝结合了业界领先的商业和开源应用软件包进行了优化。很多用法可参考Open MPI库，该部分主要介绍其不同的参数设置等。

7.1.1 Mellanox Fabric集合通信加速(Fabric Collective Accelerator, FCA)

集合通信执行全体工薪操作占用系统中所有进程/节点，因此必须执行越快越高效越好。很多应用里面都含有大量的集合通讯，普通的MPI实现那会占用大量的CPU资源及产生系统噪声。Mellanox将很多类似通信从CPU卸载到Mellanox硬件网卡适配器(HCA)和交换机上及降低噪声，这种技术称之为CORE-Direct® (Collectives Offload Resource Engine)。

FCA 4.4当前支持阻塞和非阻塞的集合通信：Allgather、Allgatherv、Allreduce、AlltoAll、AlltoAllv、Barrier和Bcast。

采用FCA v4.x (hcoll)运行MPI

HPC-X默认启用FCA v4.3。

- 采用默认FCA配置参数运行：

```
mpirun -mca coll_hcoll_enable 1 -x HCOLL_MAIN_IB=mlx5_0:1 <...>
```

- 采用FCA运行：

```
oshrun -mca scoll_mpi_enable 1 -mca scoll basic,mpi -mca coll_hcoll_enable 1 <...>
```

Open MPI中启用FCA

在Open MPI中启用FCA v4.4，通过下述方法显式设定模块化组件架构模块化组件架构MCA(Modular Component Architecture)参数：

```
mpirun -np 32 -mca coll_hcoll_enable 1 -x coll_hcoll_np=0 \
        -x HCOLL_MAIN_IB=<device_name>:<port_num> ./a.out
```

调整FCA v4.4配置

显示当前信息：

```
/opt/mellanox/hcoll/bin/hcoll_info --all
```

FCA v4.4的参数是简单的环境变量，可以通过以下方式之一设置：

- 通过mpirun命令设置：

```
mpirun ... -x HCOLL_ML_BUFFER_SIZE=65536
```

- 从SHELL设置：

```
export HCOLL_ML_BUFFER_SIZE=65536
mpirun ...
```

选择端口及设备

-x HCOLL_MAIN_IB=<device_name>:<port_num>

启用卸载MPI非阻塞集合

-x HCOLL_ENABLE_NBC=1

支持以下非阻塞MPI集合：

- MPI_Ibarrier
- MPI_Ibcast
- MPI_Iallgather
- MPI_Iallreduce (4b, 8b, SUM, MIN, PROD, AND, OR, LAND, LOR)

注意：启用非阻塞MPI集合将在阻塞MPI集合中禁止多播聚合。

启用Mellanox SHARP软件加速集合

HPC-X支持Mellanox SHARP软件加速集合，这些集合默认是启用的。

- 启用Mellanox SHARP加速：

  -x HCOLL_ENABLE_SHARP=1

- 禁止Mellanox SHARP加速

  -x HCOLL_ENABLE_SHARP=0

- 更改Mellanox SHARP消息阈值（默认为256）：

  -x HCOLL_BCOL_P2P_ALLREDUCE_SHARP_MAX=<threshold>

HCOLL v4.4中的GPU缓存支持

如果CUDA运行时(runtime)是有效的，则HCOLL自动启用GPU支持。以下集合操作支持GPU缓存：

- MPI_Allreduce
- MPI_Bcast
- MPI_Allgather

如果libhcoll的其它聚合操作API被启用GPU缓存调用，则会检查缓存类型后返回错误HCOLL_ERROR。

控制参数为*HCOLL_GPU_ENABLE*，其值可为0、1和-1：

值	含义
0	禁止GPU支持。不会检查用户缓存指针。此情形下，如用户提供在GPU上分配缓存，则这种行为是未定义的。
1	启用GPU支持。将检查缓存指针，且启用HCOLL GPU聚合，这是CUDA运行时有效时的默认行为。
-1	部分GPU支持。将检查缓存指针，且HCOLL回退到GPU缓存情形下的运行时。

局限性

对于MPI_Allreduce的GPU缓存支持，不是所有(OP, DTYPE)的组合都支持：

- 支持的操作
  - SUM
  - PROD
  - MIN
  - MAX
- 支持的类型
  - INT8、INT16、INT32、INt64
  - UINT8、UINT16、UINT32、UINT64
  - FLOAT16、FLOAT32、FLOAT64

局限性

环境变量*HCOLL_ALLREDUCE_ZCOPY_TUNE=<static/dynamic>*（默认为dynamic）用于设置HCOLL的大数据全归约操作算法的自动化运行优化级别。如为Static，则对运行时不优化；如是dynamic，则允许HCOLL基于性能的运行时抽样自动调节算法的基数和zero-copy[6]阈值。

注：由于dynamic模式可能会导致浮点数归约结果有变化，因此不应该用于要求是数值可再现的情形下，导致该问题的原因在于非固定的归约顺序。

[6]zero copy技术就是减少不必要的内核缓冲区跟用户缓冲区间的拷贝，从而减少CPU的开销和内核态切换开销，达到性能的提升

7.1.2 统一通信-X架构(Unified Communication - X Framework, UCX)

UCX是一种新的加速库，并且被集成到OpenMPI（作为pml层）和OpenSHMEM（作为spml层）中作为HPC-X的一部分。这是种开源的通信库，被设计为HPC应用能获得最高的性能。UCX含有广泛范围的优化，能实现在通信方面接近低级别软件开销，接近原生级别的性能。

UCX支持接收端标记匹配、单边通信语义、有效内存注册和各种增强，能有效提高HPC应用的缩放性和性能。

UCX支持：

- InfiniBand传输：
  - 不可信数据报(Unreliable Datagram, UD)
  - 可信连接(Reliable Connected, RC)
  - 动态连接(Dynamically Connected, DC)
  - 加速verbs(Accelerated verbs)
- Shared Memory communication with support for KNEM, CMA and XPMEM
- RoCE
- TCP
- CUDA

更多信息，请参见：`https://github.com/openucx/ucx`、`http://www.openucx.org/`

OpenMPI中使用UCX

UCX在Open MPI是默认的pml，在OpenSHMEM中是默认的spml，一般安装好设置号后无需用户自己设置就可使用，用户也可利用下面方式显式指定：

- 在Open MPI中显式指定采用UCX：

  mpirun --mca pml ucx --mca osc ucx ...

- 在OpenSHMEM显示指定采用UCX：

  oshrun --mca spml ucx ...

调整UCX

检查UCX的版本：

$HPCX_UCX_DIR/bin/ucx_info -v

UCX的参数可通过下述方法之一设置：

- 通过mpirun设置：

  mpirun -x UCX_RC_VERBS_RX_MAX_BUFS=128000 <...>

- 通过SHELL设置：

  export UCX_RC_VERBS_RX_MAX_BUFS=128000

  mpirun <...>

- 从命令行选择采用的传输：

  mpirun -mca pml ucx -x UCX_TLS=sm,rc_x ...

  上述命令设置了采用pml ucx和设定其用于使用、共享内存和加速传输verbs。

- 为了提高缩放性能，可以加大DC传输时使用的网卡的DC发起者(DCI)的QPs数

  mpirun -mca pml ucx -x UCX_TLS=sm,dc_x -x UCX_DC_MLX5_NUM_DCI=16

  对于大规模系统，当DC传输不可用或者被禁用时，UCX将回退到UD传输。

  在256个连接建立后，RC传输将被禁用，该值可以利用*UCX_RC_MAX_NUM_EPS*环境变量加大。

- 设置UCX使用zero-copy时的阈值

  mpirun -mca pml ucx -x UCX_ZCOPY_THRESH=16384

  默认UCX会自己计算优化的该阈值，需要时可利用上面环境变量覆盖掉。

- 利用*UCX_DC_MLX5_TX_POLICY=<policy>*环境变量设定端点如何选择DC。策略<policy>可以为：

  – dcs：端点或者采用已指定的DCI或DCI是LIFO顺序分配的，且在不在有操作需要时释放。

  – dcs_quota：类似dcs。另外，该DCI将在发送超过一定配额时，且有端点在等待DCI时被释放。该DCI一旦完成其所有需要的操作后就被释放。该策略确保了在端点间没有饥荒。

  – rand：每个端点被赋予一个随机选择的DCI。多个端点有可能共享相同的DCI。

- 利用UCX CUDA内存钩子也许在静态编译CUDA应用时不会生效，作为一个工作区，可利用下面选项扩展配置：

  *-x UCX_MEMTYPE_CACHE=0 -x HCOLL_GPU_CUDA_MEMTYPE_CACHE_ENABLE=0 *

  -x HCOLL_GPU_ENABLE=1

- GPUDirectRDMA性能问题可以通过分离协议禁止：

  -x UCX_RNDV_SCHEME=get_zcopy

- 共享内存新传输协议命名为：TBD

  可用的共享内存传输名是：posix、sysv和xpmem

  sm和mm将被包含咋以上三种方法中。

  设备‘device’名对于共享内存传输是‘memory’（在*UCX_SHM_DEVICES*中使用）

UCX特性

硬件标识符匹配(Tag Matching)

从ConnectX-5起，在UCX中之前由软件负责的标识符匹配工作可以卸载到HCA。对于MPI应用发送消息时附带的数值标志符的加速对收到消息的处理，可以提高CPU利用率和降低等待消息的延迟。在标识符匹配中，由软件控制的匹配入口表称为匹配表。每个匹配入口含有一个标志符及对应一个应用缓存的指针。匹配表被用于根据消息标志引导到达的消息到特定的缓存。该传输匹配表和寻找匹配入口的动作被称为标识符匹配，该动作由HCA而不再由CPU实现。在当收到的消息被不按照到达顺序而是基于与发送者相关的数值标记使用时，非常有用。

硬件标识符匹配使得省下的CPU可供其它应用使用。当前硬件标识符匹配对于加速的RC和DC传输(RC_X和DC_X)是支持的，且可以在UCX中利用下面环境参数启用：

- 对RC_X传输：*UCX_RC_MLX5_TM_ENABLE=y*
- 对DC_X传输：*UCX_DC_MLX5_TM_ENABLE=y*

默认，只有消息大于一定阈值时才卸载到传输。该阈值由*UCXTM_THRESH*环境变量控制，默认是1024比特。

对于硬件标识符匹配，特定阈值时，UCX也许用回弹缓冲区(bounce buffer)卸载内部预注册缓存代替用户缓存。该阈值由*UCX_TM_MAX_BB_SIZE*环境变量控制，该值等于或小于分片大小，且必须大于*UCX_TM_THRESH*才能生效（默认为1024比特，即默认优化是被禁止的）。

CUDA GPU

HPC-X中的CUDA环境支持,使得HPC-X在针对点对点和集合函数的UCX和HCOLL通信库中使用各自NVIDIA GPU显存。

系统已安装了NVIDIA peer memory，支持GPUDirect RDMA。

片上内存(MEMIC)

片上内存允许从UCX层发送消息时使用设备上的内存，该特性默认启用。它仅支持UCX中的rc_x和dc_x传输。

控制这些特性的环境变量为：

- *UCX_RC_MLX5_DM_SIZE*
- *UCX_RC_MLX5_DM_COUNT*
- *UCX_DC_MLX5_DM_SIZE*
- *UCX_DC_MLX5_DM_COUNT*

针对这些参数的更多信息,可以运行ucx_info工具查看:*$HPCX_UCX_DIR/bin/ucx_info -f*。

生成Open MPI/OpenSHMEM的UCX统计信息

为生成统计信息，需设定统计目的及触发器，它们可被选择性过滤或/和格式化。

- 统计目的可以利用*UCX_STATS_DEST*环境变量设置，其值可以为下列之一：

空字符串	不会生成统计信息
file:<filename>	存到一个文件中,具有以下替换:%h: host, %p:pid, %c:cpu, %t: time, %e:exe，如文件名有%h，则自动替换为节点名
stderr	显示标准错误信息
stdout	显示标准输出
udp:<host>[:<port>]	通过UDP协议发送到host:port

比如：

export UCX_STATS_DEST=”file:ucx_%h_%e_%p.stats”

export UCX_STATS_DEST=”stdout”

- 触发器通过*UCX_STATS_TRIGGER*环境变量设置，其值可以为下述之一：

exit	在程序退出前存储统计信息
timer:<interval>	每隔一定时间存储统计信息
signal:<signo>	当进程收到信号时存储统计信息

比如：

export UCX_STATS_TRIGGER=exit export UCX_STATS_TRIGGER=timer:3.5

- 利用*UCX_STATS_FILTER*环境变量可以过滤报告中的计数器。它接受以,分割的一组匹配项以指定显示的计数器，统计概要将包含匹配的计数器，批配项的顺序是没关系的。列表中的每个表达式可以包含任何以下选项：

*	匹配任意字符，包含没有（显示全部报告）
?	匹配任意单子字符
[abc]	匹配在括号中的一个字符
[a-z]	匹配从括号中一定范围的字符

关于该参数的更多信息可以参见：https://github.com/openucx/ucx/wiki/Statistics。

- 利用*UCX_STATS_FORMAT*环境参数可以控制统计的格式：

full	每个计数器都将被在一个单行中显示
agg	每个计数器将在一个单行中显示，但是将会聚合类似的计数器
summary	所有计数器将显示在同一行

注意：统计特性只有当编译安装UCX库时打开启用统计标记的时候才生效。默认为No，即不启用。因此为了使用统计特性，请重新采用*contrib/configure-prof*文件编译UCX，或采用debug版本的UCX，可以在$HPCX_UCX_DIR/debug中找到：

mpirun -mca pml ucx -x LD_PRELOAD=$HPCX_UCX_DIR/debug/lib/libucp.so ...

注意：采用上面提到的重新编译的UCX将会影响性能。

7.1.3 PGAS共享内存访问(OpenSHMEM)

共享内存(SHMEM)子程序为高级并行扩展程序提供了低延迟、高带宽的通信。这些子程序在SHMEM API中提供了用于在协作并行进程间交换数据的编程模型。SHMEM API可在同一个并行程序中独自或与MPI子程序一起使用。

SHMEM并行编程库是一种非常简单易用的编程模型，可以使用高效的单边通讯API为共享或分布式内存系统提供直观的全局观点接口。

SHMEM程序是单程序多数据(SPMD)类型的。所有的SHMEM进程，被引用为进程单元(PEs)，同时启动且运行相同程序。通常，PEs在它们大程序中自己的子域进行计算，并且周期性与其它下次通讯依赖的PEs进行通信实现数据交换。

SHMEM子程序最小化数据传输请求、最大化带宽以及最小化数据延迟（从一个PE初始化数据传输到结束时的时间周期差）。

SHMEM子程序通过以下支持远程数据传输：

- put操作：传递数据给一个不同PE；
- get操作：从一个不同PE和远程指针获取数据，允许直接访问属于其它PE的数据。

其它支持的操作是集合广播和归约、栅栏同步和原子内存操作(atomic memory operation)。原子内存操作指的是原子（不允许多个进程同时）读-更新操作，比如对远程或本地数据的获取-增加。

SHMEM库实现激活消息。源处理器将数据传输到目的处理器时仅涉及一个CPU，例如，一个处理器从另外处理器内存读取数据而无需中断远程CPU，除非编程者实现了一种机制去告知这些，否则远程处理器察觉不到其内存被读写。

HPC-X Open MPI/OpenSHMEM

HPC-X Open MPI/OpenSHMEM编程库是单边通信库，支持唯一的并行编程特性集合，包括并行程序应用进程间使用的点对点和集合子程序、同步、原子操作、和共享内存范式。

HPC-X OpenSHMEM基于OpenSHMEM.org协会定义的API，该库可在OFED(OpenFabrics RDMA for Linux stack)上运行，并可使用UCX和Mellanox FCA，为运行在InfiniBand上的SHMEM程序提供了史无前例的可扩展性级别。

运行HPC-X OpenSHMEM

采用UCX运行HPC-X OpenSHMEM

对于HPC-X，采用spml对程序提供服务。v2.1及之后的版本的HPC-X，ucx已是默认的spml，无需特殊指定，或者也可在oshrun命令行添加*-mca spml ucx*显式指定。

所有的UCX环境参数，oshrun使用时与mpirun一样，完整的列表可运行下面命令获取：

$HPCX_UCX_DIR/bin/ucx_info -f

采用HPC-X OpenSHMEM与MPI一起开发应用

SHMEM编程模型提供了一种提高延迟敏感性部分的性能的方法。通常，要求采用调用shmem_put/shmem_get和shmem_barrier来代替调用MPI中的send/recv。

SHMEM模型对于短消息来说可以相比传统的MPI调用能显著降低延时。对于MPI-2 MPI_Put/MPI_Get函数，也可以考虑替换为shmem_get/shmem_put调用。

HPC-X OpenSHMEM调整参数

HPC-X OpenSHMEM采用MCA参数来调整设置用户应用运行时环境。每个参数对应一种特定函数，以下为可以改变用于应用函数的参数：

- memheap：控制内存分配策略及阈值
- scoll：控制HPC-X OpenSHMEM集合API阈值及算法
- spml：控制HPC-X OpenSHMEM点对点传输逻辑及阈值
- atomic：控制HPC-X OpenSHMEM原子操作逻辑及阈值
- shmem：控制普通HPC-X OpenSHMEM API行为

显示HPC-X OpenSHMEM参数：

- 显示所有可用参数：
  - *oshmem_info -a*

- 显示HPC-X OpenSHMEM特定参数：
  - *oshmem_info --param shmem all*
  - *oshmem_info --param memheap all*
  - *oshmem_info --param scoll all*
  - *oshmem_info --param spml all*
  - *oshmem_info --param atomic all*

注意：在所有节点上运行OpenSHMEM应用或性能测试时，需要执行以下命令以释放内存：

echo 3 > /proc/sys/vm/drop_caches

针对对称堆(Symmetric Heap)应用的OpenSHMEM MCA参数

SHMEM memheap大小可以通过对oshrun命令添加*SHMEM_SYMMETRIC_HEAP_SIZE*参数来设置，默认为256M。

例如，采用64M memheap来运行SHMEM：

*oshrun -x SHMEM_SYMMETRIC_HEAP_SIZE=64M -np 512 -mca mpi_paffinity_alone 1 *

--map-by node -display-map -hostfile myhostfile example.exe

memheap可以采用下述方法分配：

- sysv：system V共享内存API，目前不支持采用大页面(hugepages)分配。
- verbs：采用IB verbs分配子。
- mmap：采用mmap()分配内存。
- ucx：通过UCX库分配和注册内存

默认，HPC-X OpenSHMEM会自己寻找最好的分配子，优先级为verbs、sysv、mmap和ucx，也可以采用-mca sshmem <name>指定分配方法。

用于强制连接生成的参数

通常，SHMEM会在PE间消极地生成连接，一般是在第一个通信发生时。

- 开始时就强制连接生成，设定MCA参数：

  -mca shmem_preconnect_all 1

  内存注册器（如，infiniband rkeys）信息启动时会在进程间交换。

- 启用按需内存密钥(key)交换，可设置MCA参数：

  -mca shmalloc_use_modex 0

7.2 Open MPI库

Open MPI[7]库是另一种非常优秀MPI实现，用户如需使用可以自己通过运行*module load*选择加载与openmpi相关的项自己设置即可。

Open MPI的安装目录在*/opt/openmpi*下。

Open MPI的编译命令主要为：

- C程序：*mpicc*
- C++程序：*mpic++*、*mpicxx*、*mpiCC*
- Fortran 77程序：*mpif77*、*mpif90*、*mpifort*
- Fortran 90程序：*mpif90*
- Fortran程序：*mpifort*[8]

*mpifort*为1.8系列引入的编译Fortran程序的命令。

*mpif77*和*mpif90*为1.6系列和1.8系列的编译Fortran程序的命令。

对于MPI并行程序，对应不同类型源文件的编译命令如下：

- 将C语言的MPI并行程序yourprog-mpi.c编译为可执行文件yourprog-mpi：
  mpicc -o yourprog-mpi yourprog-mpi.c

- 将C++语言的MPI并行程序yourprog-mpi.cpp编译为可执行文件yourprog-mpi，也可换为*mpic++*或*mpiCC*：
  mpicxx -o yourprog-mpi yourprog-mpi.cpp

- 将Fortran 90语言的MPI并行程序yourprog-mpi.f90编译为可执行文件yourprog-mpi：
  mpifort -o yourprog-mpi yourprog-mpi.f90

[7]主页：`http://www.open-mpi.org/`

[8]注意为mpifort，而不是Intel MPI的mpiifort

- 将Fortran 77语言的MPI并行程序yourprog-mpi.f编译为可执行文件yourprog-mpi：
  mpif77 -o yourprog-mpi yourprog-mpi.f
- 将Fortran 90语言的MPI并行程序yourprog-mpi.f90编译为可执行文件yourprog-mpi：
  mpif90 -o yourprog-mpi yourprog-mpi.f90

编译命令的基本语法为：编译命令*[-showme|-showme:compile|-showme:link] ...*

编译参数可以为：

- –showme：显示所调用的编译器所调用编译参数等信息。
- –showme:compile：显示调用的编译器的参数
- –showme:link：显示调用的链接器的参数
- –showme:command：显示调用的编译命令
- –showme:incdirs：显示调用的编译器所使用的头文件目录，以空格分隔。
- –showme:libdirs：显示调用的编译器所使用的库文件目录，以空格分隔。
- –showme:libs：显示调用的编译器所使用的库名，以空格分隔。
- –showme:version：显示Open MPI的版本号。

默认使用配置Open MPI时所用的编译器及其参数，可以利用环境变量来改变。环境变量格式为*OMPI_value*，其*value*可以为：

- *CPPFLAGS*：调用C或C++预处理器时的参数
- *LDFLAGS*：调用链接器时的参数
- *LIBS*：调用链接器时所添加的库
- *CC*：C编译器
- *CFLAGS*：C编译器参数
- *CXX*：C++编译器
- *CXXFLAGS*：C++编译器参数
- *F77*：Fortran 77编译器
- *FFLAGS*：Fortran 77编译器参数
- *FC*：Fortran 90编译器
- *FCFLAGS*：Fortran 90编译器参数

7.3 Intel MPI库

特别提醒：Intel MPI针对最新的Mellanox HDR有问题，不建议使用；如您的应用运行起来没问题，也可以使用。

Intel MPI库[9]是一种多模消息传递接口(MPI)库，所安装的5.0版本Intel MPI库实现了MPI V3.0标准。Intel MPI库可以使开发者采用新技术改变或升级其处理器和互联网络而无需改编软件或操作环境成为可能。主要包含以下内容：

- Intel MPI库运行时环境(RTO)：具有运行程序所需要的工具，包含多功能守护进程(MPD)、Hydra及支持的工具、共享库(.so)和文档。
- Intel MPI库开发套件(SDK)：包含所有运行时环境组件和编译工具，含编译器命令，如*mpiicc*、头文件和模块、静态库(.a)、调试库、追踪库和测试代码。

7.3.1 编译命令

请注意，Intel MPI与Open MPI等MPI实现不同，*mpicc*、*mpif90*和*mpifc*命令默认使用GNU编译器，如需指定使用Intel编译器等，请使用对应的*mpiicc*、*mpiicpc*和*mpiifort*命令。下表为Intel MPI编译命令及其对应关系。

其中：

- ia32：IA-32架构。
- intel64：Intel 64(x86_64, amd64)架构。
- 移植现有的MPI程序到Intel MPI库时，请重新编译所有源代码。
- 如需显示某命令的简要帮助，可以不带任何参数直接运行该命令。

7.3.2 编译命令参数

- -mt_mpi：采用以下级别链接线程安全的MPI库：MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED或MPI_THREAD_MULTIPLE。

  Intel MPI库默认使用MPI_THREAD_FUNNELED级别线程安全库。

  注意：

  – 如使用Intel C编译器编译时添加了-openmp、-qopenmp或-parallel参数，那么使用线程安全库。
  – 如果用Intel Fortran编译器编译时添加了如下参数，那么使用线程安全库：

[9]主页：http://software.intel.com/en-us/intel-mpi-library/

表 8: Intel MPI编译命令及其对应关系

编译命令	调用的默认编译器命令	支持的语言	支持的应用二进制接口
通用编译器			
mpicc	gcc, cc	C	32/64 bit
mpicxx	g++	C/C++	32/64 bit
mpifc	gfortran	Fortran77*/Fortran 95*	32/64 bit
GNU* Compilers Versions 3 and Higher			
mpigcc	gcc	C	32/64 bit
mpigxx	g++	C/C++	32/64 bit
mpif77	g77	Fortran 77	32/64 bit
mpif90	gfortran	Fortran 95	32/64 bit
Intel Fortran, C++ Compilers Versions 11.1 and Higher			
mpiicc	icc	C	32/64 bit
mpiicpc	icpc	C++	32/64 bit
mpiifort	ifort	Fortran77/Fortran 95	32/64 bit

  * -openmp
  * -qopenmp
  * -parallel
  * -threads
  * -reentrancy
  * -reentrancy threaded

- -static_mpi：静态链接Intel MPI库，并不影响其它库的链接方式。
- -static：静态链接Intel MPI库，并将其传递给编译器，作为编译器参数。
- -config=*name*：使用的配置文件。
- -profile=*profile_name*：使用的MPI分析库文件。
- -t或-trace：链接Intel Trace Collector库。
- -check_mpi：链接Intel Trace Collector正确性检查库。
- -ilp64：打开局部ILP64支持。对于Fortran程序编译时如果使用-i8选项，那么也需要此ILP64选项。

- -dynamic_log：与-t组合使用链接Intel Trace Collector库。不影响其它库链接方式。
- -g：采用调试模式编译程序，并针对Intel MPI调试版本生成可执行程序。可查看官方手册Environment variables部分*I_MPI_DEBUG*变量查看-g参数添加的调试信息。采用调试模式时不对程序进行优化，可查看*I_MPI_LINK*获取Intel MPI调试版本信息。
- -link_mpi=*arg*：指定链接MPI的具体版本，具体请查看*I_MPI_LINK*获取Intel MPI版本信息。此参数将覆盖掉其它参数，如-mt_mpi、-t=log、-trace=log和-g。
- -O：启用编译器优化。
- -fast：对整个程序进行最大化速度优化。此参数强制使用静态方法链接Intel MPI库。*mpiicc*、*mpiicpc*和*mpiifort*编译命令支持此参数。
- -echo：显示所有编译命令脚本做的信息。
- -show：仅显示编译器如何链接，但不实际执行。
- -{cc,cxx,fc,f77,f90}=*compiler*：选择使用的编译器。如：*mpicc -cc=icc -c test.c*。
- -gcc-version=*nnn*，设置编译命令*mpicxx*和*mpiicpc*编译时采用部分GNU C++环境的版本，如nnn的值为340，表示对应GNU C++ 3.4.x。

<nnn>值	GNU* C++版本
320	3.2.x
330	3.3.x
340	3.4.x
400	4.0.x
410	4.1.x
420	4.2.x
430	4.3.x
440	4.4.x
450	4.5.x
460	4.6.x
470	4.7.x

- -compchk：启用编译器设置检查，以保证调用的编译器配置正确。
- -v：显示版本信息。

7.3.3 环境变量

- *I_MPI_{CC,CXX,FC,F77,F90}_PROFILE*和*MPI{CC,CXX,FC,F77,F90}_PROFILE*：
  - 默认分析库。
  - 语法：*I_MPI_{CC,CXX,FC,F77,F90}_PROFILE=<profile_name>*。
  - 过时语法：*MPI{CC,CXX,FC,F77,F90}_PROFILE=<profile_name>*。
- *I_MPI_TRACE_PROFILE*：
  - 设定-trace参数使用的默认分析文件。
  - 语法：*I_MPI_TRACE_PROFILE=<profile_name>*
  - *I_MPI_{CC,CXX,F77,F90}_PROFILE*环境变量将覆盖掉*I_MPI_TRACE_PROFILE*。
- *I_MPI_CHECK_PROFILE*：
  - 设定-check_mpi参数使用的默认分析。
  - 语法：*I_MPI_CHECK_PROFILE=<profile_name>*。
- *I_MPI_CHECK_COMPILER*：
  - 设定启用或禁用编译器兼容性检查。
  - 语法：*I_MPI_CHECK_COMPILER=<arg>*。
    - <arg>为enable | yes | on | 1时打开兼容性检查。
    - <arg>为disable | no | off | 0时，关闭编译器兼容性检查，为默认值。
- *I_MPI_{CC,CXX,FC,F77,F90}*和*MPICH_{CC,CXX,FC,F77,F90}*：
  - 语法：*I_MPI_{CC,CXX,FC,F77,F90}=<compiler>*。
  - 过时语法：*MPICH_{CC,CXX,FC,F77,F90}=<compiler>*。
  - <compiler>为编译器的编译命令名或路径。
- *I_MPI_ROOT*：
  - 设置Intel MPI库的安装目录路径。
  - 语法：*I_MPI_ROOT=<path>*。
  - <path>为Intel MPI库的安装后的目录。
- *VT_ROOT*：
  - 设置Intel Trace Collector的安装目录路径。

  - 语法：*VT_ROOT=<path>*。
  - <path>为Intel Trace Collector的安装后的目录。

- *I_MPI_COMPILER_CONFIG_DIR*：
  - 设置编译器配置目录路径。
  - 语法：*I_MPI_COMPILER_CONFIG_DIR=<path>*。
  - <path>为编译器安装后的配置目录，默认值为<installdir>/<arch>/etc。

- *I_MPI_LINK*：
  - 设置链接MPI库版本。
  - 语法：*I_MPI_LINK=<arg>*。
  - <arg>可为：
    * opt：优化的单线程版本Intel MPI库；
    * opt_mt：优化的多线程版本Intel MPI库；
    * dbg：调试的单线程版本Intel MPI库；
    * dbg_mt：调试的多线程版本Intel MPI库；
    * log：日志的单线程版本Intel MPI库；
    * log_mt：日志的多线程版本Intel MPI库。

7.3.4 编译举例

对于MPI并行程序，对应不同类型源文件的编译命令如下：

- 调用默认C编译器将C语言的MPI并行程序yourprog-mpi.c编译为可执行文件yourprog-mpi：
  mpicc -o yourprog-mpi yourprog-mpi.c

- 调用Intel C编译器将C语言的MPI并行程序yourprog-mpi.c编译为可执行文件yourprog-mpi：
  mpiicc -o yourprog-mpi yourprog-mpi.cpp

- 调用Intel C++编译器将C++语言的MPI并行程序yourprog-mpi.cpp编译为可执行文件yourprog-mpi：
  mpiicxx -o yourprog-mpi yourprog-mpi.cpp

- 调用GNU Forttan编译器将Fortran 77语言的MPI并行程序yourprog-mpi.f编译为可执行文件yourprog-mpi：
  mpif90 -o yourprog-mpi yourprog-mpi.f

- 调用Intel Fortran编译器将Fortran 90语言的MPI并行程序yourprog-mpi.f90编译为可执行文件yourprog-mpi：
  mpiifort -o yourprog-mpi yourprog-mpi.f90

7.3.5 调试

使用以下命令对Intel MPI库调用GDB调试器：*mpirun -gdb -n 4 ./testc*

可以像使用GDB调试串行程序一样调试。

也可以使用以下命令附着在一个运行中的作业上：

mpirun -n 4 -gdba <pid>

其中<pid>为运行中的MPI作业进程号。

环境变量I_MPI_DEBUG提供一种获得MPI应用运行时信息的方式。可以设置此变量的值从0到1000，值越大，信息量越大。

mpirun -genv I_MPI_DEBUG 5 -n 8 ./my_application

更多信息参见程序调试章节。

7.3.6 追踪

使用-t或-trace选项链接调用Intel Trace Collector库生成可执行程序。此与当在mpiicc或其它编译脚本中使用-profile=vt时具有相同的效果。

mpiicc -trace test.c -o testc

在环境变量VT_ROOT中包含用Intel Trace Collector库路径以便使用此选项。设置I_MPI_TRACE_PROFILE为<profile_name>环境变量指定另一个概要库。如设置I_MPI_TRACE_PROFILE为vtfs，以链接fail-safe版本的Intel Trace Collector库。

7.3.7 正确性检查

使用-check_mpi选项调用Intel Trace Collector正确性检查库生成可执行程序。此与当在mpiicc或其它编译脚本中使用-profile=vtmc时具有相同的效果。

mpiicc -profile=vtmc test.c -o testc

或

mpiicc -check_mpi test.c -o testc

在环境变量VT_ROOT中包含用Intel Trace Collector库路径以便使用此选项。设置I_MPI_CHECK_PROFILE为<profile_name>环境变量指定另一个概要库。

7.3.8 统计收集

如果想收集在应用使用的MPI函数统计，可以设置I_MPI_STATS环境变量的值为1到10。设置好后再运行MPI程序，则在stats.txt文件中存储统计信息。

7.4 与编译器相关的编译选项

MPI编译环境的编译命令实际上是调用Intel、PGI或GCC编译器进行编译，具体优化选项等，请参看Intel MPI、Open MPI以及Intel、PGI和GCC编译器手册。

8 MPI并行程序的运行

MPI程序最常见的并行方式类似为：*mpirun -n 40 yourmpi-prog*。

在本超算系统上，MPI并行程序需结合Slurm作业调度系统的作业提交命令*sbatch*、*srun*或*salloc*等来调用作业脚本运行，请参看Slurm作业管理系统。

Part VIII

程序调试

从Intel C/C++ Fortran编译器2015版开始，采用的是Intel改造的GDB调试器，命令为*gdb-ia*。PGI调试器很多调试命令类似GDB调试器，请自己查看相关资料。

9 GDB调试器简介

GDB调试器可以让使用者查看其它程序运行时内部发生了什么或查看其它程序崩溃时程序在做什么。主要包括以下四项功能以便帮助找出bug：

- 启动程序，并指定任何可能影响行为的东西。
- 使程序在特定条件下停止。
- 当程序停止时，检查发生了什么。
- 修改程序中的一些东西，以便能用正确的东西影响bug，并获得进一步信息。

GDB调试可用于调试采用C/C++、Fortran、D、Modula-2、OpenCL C、Pascal、Objective-C等编写的程序。

注：GDB内部命令支持缩写，在不引起歧义的情况下，允许只输入命令的前几个字符，如*l*与*list*等同。

10 基本启动方式

GDB调试器在Linux系统上可以采用命令行（command line）和图形界面（GUI，借助Eclipse* IDE或xxgdb）两种方式进行调试。

图形界面的GDB调试器相对简单，本手册主要介绍基于命令行的GDB调试器。基于命令行的启动方式主要有如下几种：

- 最常用的方式是只跟程序名为参数，启动调试程序：

  gdb program

- 启动应用程序及以前其出错时生成的core文件：

  gdb program core

- 利用运行程序的进程号吸附到运行中程序进行调试：

  gdb program 1234

- 如果需要调试的程序有参数，那么需要添加–args参数：

  gdb --args gcc -O2 -c foo.c

- 采用静默方式启动，不打印启动后的版权信息等：

  gdb --silent

- 仅显示帮助信息等：

  gdb --help

10.1 选择启动时文件

当GDB启动的时候，它读取除选项之外的任何参数用于指定可执行程序文件和core文件（或进程号），这与分别采用-se和-c（或-p）参数类似（gdb读取参数时，如第一个参数没有关联选项标记，那么等价于跟着-se选项之后的参数，如第二个参数没有关联选项标记，那么等价于跟着-c/-p选项之后的参数）。如果第二个参数以十进制数字开始，那么gdb尝试将其作为进程号并进行吸附，如果失败，则尝试作为core文件打开。如果以数字开始的core文件，那么可以在此之前添加./以防止被认为是进程号，例如./12345。很多选项同时具有长格式和短格式两种格式，如果采用了截断的长格式选项，且长度足够避免歧义，那么也可以被重新辨认为长格式。（如果你喜欢，可以采用–而不是-来标记选项参数）。

- -symbols file、-s file：从文件file中读取符号表。
- -exec file、-e file：适当时采用文件file作为可执行程序，并且与core dump文件关联时用于检查纯数据。
- -se file：从文件file中读取符号表，并且将其作为可执行文件。
- -core file、-c file：将文件file作为core dump文件进行检查。
- -pid number、-p number：将附带的命令吸附到进程号number。
- -command file、-x file：指定启动后执行的命令文件file，文件file中保存一系列命令，启动后会顺序执行。
- -eval-command command、-ex command：执行单个gdb命令，此选项可以多次使用以多次调用命令。需要时，此选项也许与-command交替，如：

  gdb -ex 'target sim' -ex 'load' -x setbreakpoints -ex 'run' a.out

- -init-command file、-ix file：从文件file中加载并执行命令，此过程在加载inferior[10]之前（但在加载gdbinit文件之后）。
- -init-eval-command command、-iex command：在加载inferior之前（但在加载gdbinit文件之后）执行命令command。
- -directory directory、-d directory：添加目录directory到源文件和脚本文件的搜索目录

10.2 记录日志

可以采用以下方式记录日志等，启动GDB后执行：

- 启用日志：*set logging on*
- 关闭日志：*set logging off*
- 记录日志到文件file（默认为gdb.txt）：*set logging file file*
- 设定日志是否覆盖原有文件（默认为追加）：*set logging overwrite [on|off]*
- 设定日志是否重定向（默认为显示在终端及文件中）：*set logging redirect [on|off]*
- 显示当前日志设置：*show logging*

11 退出GDB

退出调试器，在GDB内部命令执行完后的命令行，输入以下两者之一：

- *quit*
- *<ctrl+d>*

[10]GDB采用对象表示每个程序执行状态，这个对象被称为inferior。典型的，一个inferior对应一个进程，但是更通常的是对应一个没有进程的目标。inferior有可能在进程执行之前生成，并且可以在进程停止后驻留。inferior具有独有的与进程号不同的标志符。尽管一些嵌入的目标也许具有多个运行在单个地址空间内不同部分的多个inferior，但通常每个inferior具有自己隔离的地址空间。反过来，每个inferior又有多个线程运行它。在GDB中可以用info inferiors查看。

12 准备所需要调试的程序

12.1 准备调试代码源代码

调试程序时，一般无需修改程序源代码，但是在程序中建议做如下改变：

- 如果程序运行后，利用调试器难于终止，请设置一个初始停止点；
- 在源代码增加一些断言，以便帮助定位错误。

12.2 准备编译器和链接器环境

调试信息被编译器存储在.o文件。信息的级别和格式由编译器选项控制。

对于Intel C/C++ Fortran编译器，采用-g或-debug选项，例如：

- *icc -g hello.c*
- *icpc -g hello.cpp*
- *ifort -g hello.f90*

对于GCC编译器，采用-g选项。对于一些较老版本的GCC，此选项也许会产生DWARF-1标准的调试信息，如果这样，请使用-gdwarf-2选项，例如：

- *gcc -gdwarf-2 hello.c*
- *g++ -gdwarf-2 hello.cpp*
- *gfortran -gdwarf-2 hello.f90*

调试信息将通过*ld*命令导入到a.out（可执行程序）或.so（共享库）文件中。

如果是在调试优化编译的代码，采用-g选项将自动增加-O0选项。

请参看调试优化编译的代码部分中关于-g和相关扩展调试选项及它们的与优化之间的关系。

12.3 调试优化编译的代码

GDB调试器可以通过使用-g参数帮助调试优化编译的程序。但是关于此程序的信息也许并不准确，尤其是变量的地址和值经常没有被正确报告，这是因为通用调试信息模式无法全部表示-O1、-O2、-O3及其它优化选项的复杂性。

为了避免此限制，采用Intel编译器编译程序时在所需的-O1、-O2或-O3优化选项同时指明-g和-debug扩展选项。这会产生具有更多高级但更少通用支持的调试信息，主要激活以下：

- 给出变量的正确地址和值，不管其是在寄存器或不同时间在不同地址时。注意：
  - 在程序中，一些变量可能被优化掉或转换成不同类型的数据，或其地址没有在所有点都被记录。在这些情形下，打印变量时将显示无值。
  - 否则，这些值和地址将正确，但这些寄存器没有地址，调试器中*print &i*命令将打印一条警告。
  - 尽管*break main*命令通常将在程序开始处理后停止，但程序大多数变量和参数在程序的开始处理和结束处理时是未定义的。
- 在堆栈追踪中显示内联函数，这通过使用inline关键词识别。注意：
  - 只有在堆栈顶端和通常（非内联）调用的函数显示指令指针，其原因在于其它函数与其调用的内联函数共享硬件定义的堆栈帧。
  - 返回指令将只返回对那些采用调用指令时是非内联调用函数的控制，其原因在于内联调用没有定义返回地址。
  - *up*、*down*和*call*命令以通常方式工作。
- 允许在内联函数中设置断点。

12.4 准备所需要调试的并行程序

编译时必须用-g等调试参数编译源代码才可以使用GDB调试器特性，比如分析共享数据或在重入函数调用中停止。

为了使用并行调试特性，需要：

- 如果存在makefile编译配置文件，请对它进行编辑。
- 在命令行添加编译器选项-debug parallel（Intel编译器针对OpenMP多线程）。
- 重编译程序。

12.5 编译所要调试的程序

下面以常做为例子的hello程序为例介绍。

- hello.c例子：

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

编译：

icc -g helloworld.c -o helloworld

- hello.f90例子：

```
program main
print *,"Hello World!"
end program main
```

编译：

ifort -debug -O0 helloworld.f90 -o helloworld

13 开始调试程序

启动调试：*gdb helloworld*。更多启动方式参见10。

13.1 显示源代码

在调试器启动后的命令行中输入*list*命令可以显示源代码，如输入*list main*，将显示main函数的代码。

13.2 运行程序

在命令行中输入*run*，将开始运行程序。

13.3 设置和删除断点

- 设置断点：
  - 输入以下命令：*break main*
    此时在程序main处设置了一个断点。

  - 输入*run*再次运行程序
    应用将停止在设置的断点处。

- 删除断点：
  - 列出所有设置的断点ID号：*info breakpoints*
    调试器将显示所有存在的断点。
  - 指明所要删除的断点ID号。如果从开始调试后没有设置其它断点，那么只有1个断点，其ID号为1。
  - 删除此断点：*delete breakpoint 1*
    那么将删除设置断点1。
  - 重新运行程序。
    那么程序将运行并显示“Hello World!”，并退出程序。

13.4 控制进程环境

用户可以：1、对进程的环境变量进行设置或者取消设置以便在将来使用；2、设置与当前调试器环境和启动调试器的shell不同的环境。设置的变量将影响后续调试的新进程。环境命令不影响当前运行进程。设置的环境变量不改变或显示调试器的环境变量，它们只影响新产生的进程。

- 显示当前集的所有环境变量：*show environment*
- 增加或改变环境变量：*set environment*
- 取消一个环境变量：*unset environment*

注意：GDB调试器没有命令可以简单回到调试器启动时的环境变量的初始状态，用户必须正确设置和取消环境变量。

13.5 执行一行代码

如果源代码当前行是函数调用，那么可以步入(step into)或者跨越(step over)此函数。

1. *step*命令：应用程序执行一行代码，如果此行是函数调用，那么应用程序步入到函数中，即不执行完此函数调用。
2. *next*命令：应用程序执行一行代码，如果此行是函数调用，那么应用程序跨越此函数，即执行完此函数调用。

13.6 执行代码直到

运行代码直到某行或某个表达式，可用*until*命令。

13.7 执行一行汇编指令

如果应用的当前指令为函数调用，那么可以步入或者跨越此函数。

1. *stepi*命令：应用程序执行一行汇编指令，如果此行指令是函数调用，那么应用程序步入到函数中。
2. *nexti*命令：应用程序执行一行汇编指令，如果此行指令当前行是跳出或调用，那么应用程序跨越过它。

13.8 显示变量或表达式值

利用*print*命令可以显示变量值或表达式的值。如：

- 显示变量val2的当前值：*print val2*
- 显示表达式val2*2的值：$print\ val2*2$

14 传递命令给调试器

14.1 命令、文件名和变量补全

GDB调试器支持命令、文件名和变量的补全。在GDB调试器命令行中开始键入一个命令、文件名或变量名，然后按Tab键。如果有不只一个备选，调试器会发出铃声。再一次按Tab键，将列出备选。

利用单引号和双引号影响可能备选集。利用单引号填充C++名字，包含特殊字符“:”、“<”、“>”、“(”、“)”等。利用双引号告诉调试器在文件名中查看备选。

14.2 自定义命令

GDB调试器支持用户自定义命令。

用户定义的命令支持在定义体内包含if、while、loop_break和loop_continue命令。用户定义的命令最多可有10个参数，以空白分割。参数名依次为$arg0、$arg1、$arg2、. . .、$arg9。参数总数存储在$argc中。

其步骤为：

- 输入define commandname
- 每行输入一个命令
- 输入end

15 调试并行程序

15.1 调试OpenMP等多线程程序

一个单独的程序可以有不止一个线程执行，但一般来说，一个程序的线程除了它们共享一个地址空间外，还类似于多个进程。另一方面，每个县城具有自己的寄存器和执行堆栈，也许还占有私有内存。

线程是进程内部单个、串行控制流。每个线程包含单个执行点。线程在单个地址空间中（共享）执行；因此，进程的线程可以读写相同的内存地址。

多个进程执行时，当用户需要关注某个进程时，它却恼人地或不切实际地枚举所有进程。

当为了设置代码断点而定义停止线程和线程过滤器时，用户需要定义线程集。

用户可以以紧凑方式指定进程或线程集，集可包含一个或多个范围。用户可以对每个进程集执行普通操作，调试器变量既可以存储集也可以存储范围以便操作、引用和查看。

- *info threas*：查看线程集
- *thread*：在线程间进行切换，如thread 2
- *thread apply*：对线程应用特定命令，如thread apply 2 break 164
- *thread apply all*：对所有线程应用特定命令
- *thread find*：发现满足某些特定条件的线程
- *thread name*：给当前线程设定名字

注意：线程与当前执行到多线程程序中的位置有关系，在单线程执行的地方只显示一个线程，在多线程执行的地方会显示多线程。

对各线程就可采用普通GDB命令对单个进程分别进行调试。

15.2 调试MPI并行应用

采用Intel MPI时，可以采用类似下面命令调用GDB调试器：

mpirun -gdb -n 4 ./tmissem-dbg

之后可以像单进程程序一样调试程序。

也可以吸附到一个运行中的程序：

mpirun -n 4 -gdba <pid>

其中<pid>为MPI进程的进程号。

如：*mpirun -gdb -n 4 ./tmissem-dbg*显示：

```
mpigdb: np = 4
mpigdb: attaching to 13526 ./tmissem-dbg tc4600v4
mpigdb: attaching to 13527 ./tmissem-dbg tc4600v4
mpigdb: attaching to 13528 ./tmissem-dbg tc4600v4
mpigdb: attaching to 13529 ./tmissem-dbg tc4600v4
```

上面np=4显示使用了4个进程启动MPI程序，13526之类的为系统MPI程序进程号（不是MPI rank号），./tmissem-dbg为应用程序，tc4600v4为对应节点。

查看源码，执行*list*：

```
[2,3]␣␣200␣␣␣␣implicit none
[0,1]␣␣200␣␣␣␣implicit none
[2,3]␣␣201␣␣␣␣include 'mpif.h'
[0,1]␣␣201␣␣␣␣include 'mpif.h'
[2,3]␣␣202␣␣␣␣integer nmstep,ik,NStep,jk,i,PNum
[0,1]␣␣202␣␣␣␣integer nmstep,ik,NStep,jk,i,PNum
[2,3]␣␣203␣␣␣␣!real pathxyz(3,100000),t_p(3) !path
[0,1]␣␣203␣␣␣␣!real pathxyz(3,100000),t_p(3) !path
[2,3]␣␣204␣␣␣␣real␣␣␣Time_S
[0,1]␣␣204␣␣␣␣real␣␣␣Time_S
[2,3]␣␣205␣␣␣␣real(8) T3
[0,1]␣␣205␣␣␣␣real(8) T3
[2,3]␣␣206␣␣␣␣character*2 resf
[0,1]␣␣206␣␣␣␣character*2 resf
[2,3]␣␣207
[0,1]␣␣207
[2,3]␣␣208␣␣␣␣call MPI_Init(ierr)
[0,1]␣␣208␣␣␣␣call MPI_Init(ierr)
```

上面[0-3]、[0,1]之类的为MPI进程编号，表示改行后面显示的内容为这些进程的。

*z*命令可设置对某MPI进程进行操作，如*z 0,1,3*命令设置当前进程集包含进程0、1、3：

```
mpigdb: set active processes to 0 1 3
```

*z all*切换到全部进程。

之后对各进程就可采用普通GDB命令对单个进程分别进行调试。

Part IX

Intel MKL数值函数库

本系统上安装的数值函数库主要有Intel核心数学库(Math Kernel Library, MKL)，用户可以直接调用，以提高性能、加快开发。

16 Intel MKL

当前安装的Intel MKL版本为Intel Parallel Studio XE 2018、2019和2020版编译器自带的，安装在*/opt/intel*。在BASH下可以通过运行*module load*选择Intel编译器时设置，或者在*~/.bashrc*之类的环境变量设置文件中添加类似下面代码设置Intel MKL所需的环境变量*INCLUDE*、*LD_LIBRARY_PATH*和*MANPATH*等：

```
. /opt/intel/2020/mkl/bin/mklvars.sh intel64
```

17 Intel MKL主要内容

Intel MKL主要包含如下内容：

- 基本线性代数子系统库（BLAS, level 1, 2, 3）和线性代数库（LAPACK）：提供向量、向量-矩阵、矩阵-矩阵操作。
- ScaLAPACK分布式线性代数库：含基础线性代数通信子程序（Basic Linear Algebra Communications Subprograms, BLACS）和并行基础线性代数子程序（Parallel Basic Linear Algebra Subprograms, PBLAS）
- PARDISO直接离散算子：一种迭代离散算子，支持用于求解方程的离散系统的离散BLAS (level 1, 2, and 3)子函数，并提供可用于集群系统的分布式版本的PARDISO。
- 快速傅立叶变换方程（Fast Fourier transform, FFT）：支持1、2或3维，支持混合基数（不局限与2的次方），并有分布式版本。
- 向量数学库（Vector Math Library, VML）：提供针对向量优化的数学操作。
- 向量统计库（Vector Statistical Library, VSL）：提供高性能的向量化随机数生成算子，可用于一些几率分布、剪辑和相关例程和汇总统计功能。

- 数据拟合库（Data Fitting Library）：提供基于样条函数逼近、函数的导数和积分，及搜索。
- 扩展本征解算子（Extended Eigensolver）：基于FEAST的本征值解算子的共享内存版本的本征解算子。

18 Intel MKL目录内容

Intel MKL的主要目录内容见表9。

19 链接Intel MKL

19.1 快速入门

19.1.1 利用-mkl编译器参数

Intel Composer XE编译器支持采用-mkl[12]参数链接Intel MKL：

- -mkl或-mkl=parallel：采用标准线程Intel MKL库链接；
- -mkl=sequential：采用串行Intel MKL库链接；
- -mkl=cluster：采用Intel MPI和串行MKL库链接；
- 对Intel 64架构的系统，默认使用LP64接口链接程序。

19.1.2 使用单一动态库

可以通过使用Intel MKL Single Dynamic Library(SDL)来简化链接行。

为了使用SDL库，请在链接行上添加libmkl_rt.so。例如

icc application.c -lmkl_rt

SDL使得可以在运行时选择Intel MKL的接口和线程。默认使用SDL链接时提供：

- 对Intel 64架构的系统，使用LP64接口链接程序；
- Intel线程。

如需要使用其它接口或改变线程性质，含使用串行版本Intel MKL等，需要使用函数或环境变量来指定选择，参见19.3.2动态选择接口和线程层部分。

[12]是-mkl，不是-lmkl，其它编译器未必支持此-mkl选项。

表 9: Intel MKL目录内容

目录	内容
<mkl_dir>	MKL主目录，如*/opt/intel/2020/mkl*
<mkl_dir>/benchmarks/linpack	包含OpenMP版的LINPACK的基准程序
<mkl_dir>/benchmarks/mp_linpack	包含MPI版的LINPACK的基准程序
<mkl_dir>/bin	包含设置MKL环境变量的脚本
<mkl_dir>/bin/ia32	包含针对IA-32架构设置MKL环境变量的脚本
<mkl_dir>/bin/intel64	包含针对Intel 64架构设置MKL环境变量的脚本
<mkl_dir>/examples	一些例子，可以参考学习
<mkl_dir>/include	含有INCLUDE文件
<mkl_dir>/include/ia32	含有针对ia32 Intel编译器的Fortran 95 .mod文件
<mkl_dir>/include/intel64/ilp64	含有针对Intel64 Intel编译器ILP64接口[11]的Fortran 95 .mod文件
<mkl_dir>/include/intel64/lp64	含有针对Intel64 Intel编译器LP64接口的Fortran 95 .mod文件
<mkl_dir>/include/mic/ilp64	含针对MIC架构ILP64接口的Fortran 95 .mod文件，本系统未配置MIC
<mkl_dir>/include/mic/lp64	含针对MIC架构LP64接口的Fortran 95 .mod文件，本系统未配置MIC
<mkl_dir>/include/fftw	含有FFTW2和3的INCLUDE文件
<mkl_dir>/interfaces/blas95	包含BLAS的Fortran 90封装及用于编译成库的makefile
<mkl_dir>/interfaces/LAPACK95	包含LAPACK的Fortran 90封装及用于编译成库的makefile
<mkl_dir>/interfaces/fftw2xc	包含2.x版FFTW(C接口)封装及用于编译成库的makefile
<mkl_dir>/interfaces/fftw2xf	包含2.x版FFTW(Fortran接口)封装及用于编译成库的makefile
<mkl_dir>/interfaces/fftw2x_cdft	包含2.x版集群FFTW(MPI接口)封装及用于编译成库的makefile
<mkl_dir>/interfaces/fftw3xc	包含3.x版FFTW(C接口)封装及用于编译成库的makefile
<mkl_dir>/interfaces/fftw3xf	包含3.x版FFTW(Fortran接口)封装及用于编译成库的makefile
<mkl_dir>/interfaces/fftw3x_cdft	包含3.x版集群FFTW(MPI接口)封装及用于编译成库的makefile
<mkl_dir>/interfaces/fftw2x_cdft	包含2.x版MPI FFTW(集群FFT)封装及用于编译成库的makefile
<mkl_dir>/lib/ia32	包含IA32架构的静态库和共享目标文件
<mkl_dir>/lib/intel64	包含EM64T架构的静态库和共享目标文件
<mkl_dir>/lib/mic	用于MIC协处理器，本系统未配置MIC
<mkl_dir>/tests	一些测试文件
<mkl_dir>/tools	工具及插件
<mkl_dir>/tools/builder	包含用于生成定制动态可链接库的工具
<mkl_dir>/../Documentation/en_US/mkl	MKL文档目录

19.1.3 选择所需库进行链接

选择所需库进行链接，一般需要：

- 从接口层(Interface layer)和线程层(Threading layer)各选择一个库；
- 从计算层(Computational layer)和运行时库(run-time libraries, RTL)添加仅需的库。

链接应用程序时的对应库参见下表。

	接口层	线程层	计算层	运行库
IA-32架构，静态链接	libmkl_intel.a	libmkl_intel_thread.a	libmkl_core.a	libiomp5.so
IA-32架构，动态链接	libmkl_intel.so	libmkl_intel_thread.so	libmkl_core.so	libiomp5.so
Intel 64架构，静态链接	libmkl_intel_lp64.a	libmkl_intel_thread.a	libmkl_core.a	libiomp5.so
Intel 64架构，动态链接	libmkl_intel_lp64.so	libmkl_intel_thread.so	libmkl_core.so	libiomp5.so

SDL会自动链接接口、线程和计算库，简化了链接处理。下表列出的是采用SDL动态链接时的Intel MKL库。参见19.3.2动态选择接口和线程层部分，了解如何在运行时利用函数调用或环境变量设置接口和线程层。

	SDL	运行时库
IA-32和Intel 64架构	libmkl_rt.so	libiomp5.so

19.1.4 使用链接行顾问

Intel提供了网页方式的链接行顾问帮助用户设置所需要的MKL链接参数。访问`http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor`,按照提示输入所需要信息即可获取链接Intel MKL时所需要的参数。

19.1.5 使用命令行链接工具

使用Intel MKL提供的命令行链接工具可以简化使用Intel MKL编译程序。本工具不仅可以给出所需的选项、库和环境变量，还可执行编译和生成可执行程序。*mkl_link_tool*命令安装在*<mkl directory>/tools*，主要有三种模式：

- 查询模式：返回所需的编译器参数、库或环境变量等：
  - 获取Intel MKL库：*mkl_link_tool -libs [Intel MKL Link Tool options]*

  - 获取编译参数：*mkl_link_tool -opts [Intel MKL Link Tool options]*
  - 获取编译环境变量：*mkl_link_tool -env [Intel MKL Link Tool options]*

- 编译模式：可编译程序。
  - 用法：*mkl_link_tool [options] <compiler> [options2] file1 [file2 ...]*

- 交互模式：采用交互式获取所需要的参数等。
  - 用法：*mkl_link_tool -interactive*

参见http://software.intel.com/en-us/articles/mkl-command-line-link-tool。

19.2 链接举例

19.2.1 在Intel 64架构上链接

在这些例子中：

- MKLPATH=$MKLROOT/lib/intel64
- MKLINCLUDE=$MKLROOT/include

如果已经设置好环境变量，那么在所有例子中可以略去-I$MKLINCLUDE，在所有动态链接的例子中可以略去-L$MKLPATH。

- 使用LP64接口的并行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5 -lpthread -lm
```

- 使用LP64接口的并行Intel MKL库动态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm
```

- 使用LP64接口的串行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lpthread -lm
```

- 使用LP64接口的串行Intel MKL库动态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm
```

- 使用ILP64接口的并行Intel MKL库动态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5 -lpthread -lm
```

- 使用ILP64接口的并行Intel MKL库动态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_ilp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm
```

- 使用串行或并行（调用函数或设置环境变量选择线程或串行模式，并设置接口）Intel MKL库动态链接myprog.f:

```
ifort myprog.f -lmkl_rt
```

- 使用Fortran 95 LAPACK接口和LP64接口的并行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/intel64/lp64
-lmkl_lapack95_lp64 -Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a
-Wl,--end-group -liomp5 -lpthread -lm
```

- 使用Fortran 95 BLAS接口和LP64接口的并行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/intel64/lp64
-lmkl_blas95_lp64 -Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a
-Wl,--end-group -liomp5 -lpthread -lm
```

19.2.2 在IA-32架构上链接

在这些例子中:

- MKLPATH=$MKLROOT/lib/ia32
- MKLINCLUDE=$MKLROOT/include

如果已经设置好环境变量，那么在所有例子中可以略去-I$MKLINCLUDE，在所有动态链接的例子中可以略去-L$MKLPATH。

- 使用并行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5 -lpthread -lm
```

- 使用并行Intel MKL库动态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm
```

- 使用串行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lpthread -lm
```

- 使用串行Intel MKL库动态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel -lmkl_sequential -lmkl_core -lpthread -lm
```

- 使用串行或并行（调用mkl_set_threading_layer函数或设置环境变量MKL_THREADING_LAYER选择线程或串行模式）Intel MKL库动态链接myprog.f:

```
ifort myprog.f -lmkl_rt
```

- 使用Fortran 95 LAPACK接口和并行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/ia32
-lmkl_lapack95
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5 -lpthread -lm
```

- 使用Fortran 95 BLAS接口和并行Intel MKL库静态链接myprog.f:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/ia32
-lmkl_blas95
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5 -lpthread -lm
```

19.3 链接细节

19.3.1 在命令行上列出所需库链接

注意：下面是动态链接的命令，如果想静态链接，需要将含有-l的库名用含有库文件的路径来代替，比如用$MKLPATH/libmkl_core.a代替-lmkl_core，其中$MKLPATH为用户定义的指向MKL库目录的环境变量。

```
<files to link>

-L<MKL path> -I<MKL include>
[-I<MKL include>/{ia32|intel64|{ilp64|lp64}}]
[-lmkl_blas{95|95_ilp64|95_lp64}]
[-lmkl_lapack{95|95_ilp64|95_lp64}]
[ <cluster components> ]
-lmkl_{intel|intel_ilp64|intel_lp64|intel_sp2dp|gf|gf_ilp64|gf_lp64}
-lmkl_{intel_thread|gnu_thread|pgi_thread|sequential}
-lmkl_core
-liomp5 [-lpthread] [-lm] [-ldl]
```

注：*[]*内的表示可选，*|*表示其中之一、*{}*表示含有。在静态链接时，在分组符号（如，*-Wl,--start-group $MKLPATH/libmkl_cdft_core.a $MKLPATH/libmkl_blacs_intelmpi_ilp64.a $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group*）封装集群组件、接口、线程和计算库。

列出库的顺序是有要求的，除非是封装在上面分组符号中的。

19.3.2 动态选择接口和线程层链接

SDL接口使得用户可以动态选择Intel MKL的接口和线程层。

- 设置接口层

  可用的接口与系统架构有关，对于Intel 64架构，可使用LP64和ILP64接口。在运行时设置接口，可调用mkl_set_interface_layer函数或设置*MKL_INTERFACE_LAYER*环境变量。下表为可用的接口层的值。

接口层	*MKL_INTERFACE_LAYER*的值	mkl_set_interface_layer的参数值
LP64	LP64	MKL_INTERFACE_LP64
ILP64	ILP64	MKL_INTERFACE_ILP64

如果调用了mkl_set_interface_layer函数,那么环境变量*MKL_INTERFACE_LAYER*的值将被忽略。默认使用LP64接口。

- 设置线程层

  在运行时设置线程层，可以调用mkl_set_threading_layer函数或者设置环境变量*MKL_THREADING_LAYER*。下表为可用的线程层的值。

线程层	*MKL_INTERFACE_LAYER*的值	mkl_set_interface_layer的参数值
Intel线程	INTEL	MKL_THREADING_INTEL
串行线程	SEQUENTIAL	MKL_THREADING_SEQUENTIAL
GNU线程	GNU	MKL_THREADING_GNU
PGI线程	PGI	MKL_THREADING_PGI

如果调用了mkl_set_threading_layer函数,那么环境变量MKL_THREADING_LAYER的值被忽略。默认使用Intel线程。

19.3.3 使用接口库链接

- 使用ILP64接口 vs. LP64接口

  Intel MKL ILP64库采用64-bit整数（索引超过含有$2^{31}-1$个元素的大数组时使用），而LP64库采用32-bit整数索引数组。

  LP64和ILP64接口在接口层实现，分别采用下面接口层链接使用LP64或ILP64：

  – 静态链接：libmkl_intel_lp64.a或libmkl_intel_ilp64.a
  – 动态链接：libmkl_intel_lp64.so或libmkl_intel_ilp64.so

  ILP64接口提供以下功能：

  – 支持大数据数组（具有超过$2^{31}-1$个元素）；
  – 添加-i8编译器参数编译Fortran程序。

  LP64接口提供与以前Intel MKL版本的兼容，因为LP64对于仅提供一种接口的版本低于9.1的Intel MKL来说是一个新名字。如果用户的应用采用Intel MKL计算大数据数组或此库也许在将来会用到时请选择使用ILP64接口。

  Intel MKL提供的ILP64和LP64头文件路径是相同的。

  – 采用LP64/ILP64编译

    下面显示如何采用ILP64和LP64接口进行编译：

- Fortran:
  ILP64: *ifort -i8 -I<mkl directory>/include ...*
  LP64: *ifort -I<mkl directory>/include ...*
- C/C++:
  ILP64: *icc -DMKL_ILP64 -I<mkl directory>/include ...*
  LP64: *icc -I<mkl directory>/include ...*

注意，采用-i8或-DMKL_ILP64选项链接LP64接口库时也许将会产生预想不到的错误。

– 编写代码

如果不使用ILP64接口，无需修改代码。

为了移植或者新写代码使用ILP64接口，需要使用正确的Intel MKL函数和子程序的参数类型：

整数类型	Fortran	C/C++
32-bit整数	INTEGER*4或INTEGER(KIND=4)	int
针对ILP64/ LP64的通用整数（ILP64使用64-bit，其余32-bit）	INTEGER，不指明KIND	MKL_INT
针对ILP64/ LP64的通用整数（64-bit整数）	INTEGER*8或INTEGER(KIND=8)	MKL_INT64
针对ILP64/LP64的FFT接口	INTEGER，不指明KIND	MKL_LONG

– 局限性

所有Intel MKL函数都支持ILP64编程，但是针对Intel MKL的FFTW接口：

- FFTW 2.x封装不支持ILP64；
- FFTW 3.2封装通过专用功能函数plan_guru64支持ILP64。

• 使用Fortran 95接口库

libmkl_blas95*.a和libmkl_lapack95*.a库分别含有BLAS和LAPACK所需的Fortran 95接口，并且是与编译器无关。在Intel MKL包中，已经为Intel Fortran编译器预编译了，如果使用其它编译器，请在使用前先编译。

19.3.4 使用线程库链接

• 串行库模式

采用Intel MKL串行（非线程化）模式时，Intel MKL运行非线程化代码。它是线程安全的（除了LAPACK已过时的子程序?lacon），即可以在用户程序

的OpenMP代码部分使用。串行模式不要求与OpenMP运行时库的兼容，环境变量*OMP_NUM_THREADS*或其Intel MKL等价变量对其也无影响。

只有在不需要使用Intel MKL线程时才应使用串行模式。当使用一些非Intel编译器线程化程序或在需要非线程化库（比如使用MPI的一些情况时）的情形使用Intel MKL时，串行模式也许有用。为了使用串行模式，请选择*sequential.*库。

对于串行模式，由于*sequential.*依赖于pthread，请在链接行添加POSIX线程库(pthread)。

- 选择线程库层

  一些Intel MKL支持的编译器使用OpenMP线程技术。Intel MKL支持这些编译器提供OpenMP技术实现，为了使用这些支持，需要采用正确的线程层和编译器支持运行库进行链接。

  – 线程层

    每个Intel MKL线程库包含针对同样的代码采用不同编译器(Intel、GNU和PGI编译器）分别编译的库。

  – 运行时库

    此层包含Intel编译器兼容的OpenMP运行时库libiomp。在Intel编译器之外，libiomp提供在Linux操作系统上对更多线程编译器的支持。即，采用GNU编译器线程化的程序可以安全地采用intel MKL和libiomp链接。

    下表有助于解释在不同情形下使用Intel MKL时选择线程库和运行时库（仅静态链接情形）：

编译器	应用是否线程化	线程层	推荐的运行时库	备注
Intel	无所谓	libmkl_intel_thread.a	libiomp5.so	
PGI	Yes	libmkl_pgi_thread.a 或libmkl_sequential.a	由PGI*提供	使 用libmkl_sequential.a从Intel MKL调用中去除线程化
PGI	No	libmkl_intel_thread.a	libiomp5.so	
PGI	No	libmkl_pgi_thread.a	由PGI*提供	
PGI	No	libmkl_sequential.a	None	
GNU	Yes	libmkl_gnu_thread.a	libiomp5.so或GNU OpenMP运行时库	libiomp5提供监控缩放性能
GNU	Yes	libmkl_sequential.a	None	
GNU	No	libmkl_intel_thread.a	libiomp5.so	
other	Yes	libmkl_sequential.a	None	
other	No	libmkl_intel_thread.a	libiomp5.so	

19.3.5 使用计算库链接

- 如不使用Intel MKL集群软件在链接应用程序时只需要一个计算库即可，其依赖于链接方式：
  - 静态链接：libmkl_core.a
  - 动态链接：libmkl_core.so
- 采用Intel MKL集群软件的计算库

  ScaLAPACK和集群Fourier变换函数(Cluster FFTs)要求更多的计算库，其也许依赖于架构。下表为列出的针对Intel 64架构的使用ScaLAPACK或集群FFTs的计算库：

函数域	静态链接	动态链接
ScaLAPACK，LP64接口	libmkl_scalapack_lp64.a和libmkl_core.a	libmkl_scalapack_lp64.so和libmkl_core.so
ScaLAPACK，ILP64接口	libmkl_scalapack_ilp64.a和libmkl_core.a	libmkl_scalapack_ilp64.so和libmkl_core.so
集群FFTs	libmkl_cdft_core.a和libmkl_core.a	libmkl_cdft_core.so和libmkl_core.so

下表为列出的针对IA-32架构的使用 ScaLAPACK或集群FFTs的计算库：

函数域	静态链接	动态链接
ScaLAPACK	libmkl_scalapack_core.a和libmkl_core.a	libmkl_scalapack_core.so和libmkl_core.so
集群FFTs	libmkl_cdft_core.a和libmkl_core.a	libmkl_cdft_core.so和libmkl_core.so

注意：对于ScaLAPACK和集群FFTs，当在MPI程序中使用时，还需要添加BLACS库。

19.3.6 使用编译器运行库链接

甚至在静态链接其它库时，也可动态链接libiomp5、兼容的OpenMP运行时库。

静态链接libiomp5也许会存在问题，其原因由于操作环境或应用越复杂，将会包含更多多余的库的复本。这将不仅会导致性能问题，甚至导致不正确的结果。

动态链接libiomp5时，需确保*LD_LIBRARY_PATH*环境变量设置正确。

19.3.7 使用系统库链接

使用Intel MKL的FFT、Trigonometric Transforn或Poisson、Laplace和Helmholtz求解程序时，需要通过在链接行添加-lm参数链接数学支持系统库。

在Linux系统上，由于多线程libiomp5库依赖于原生的pthread库，因此，在任何时候，libiomp5要求在链接行随后添加-lpthread参数（列出的库的顺序非常重要）。

19.3.8 冗长（Verbose）启用模式链接

如果应用调用了MKL函数，您也许希望知道调用了哪些计算函数，传递给它们什么参数，并且花费多久执行这些函数。当启用Intel MKL冗长（Verbose）模式时，您的应用可以打印出这些信息。可以打印出这些信息的函数称为冗长启用函数。并不是所有Intel MKL函数都是冗长启用的，请查看Intel MKL发布说明。

在冗长模式下，每个冗长启用函数的调用都将打印人性化可读行描述此调用。如果此应用在此函数调用中终止，不会有针对此函数的信息打印出来。第一个冗长启用函数调用将打印一版本信息行。

为了对应用启用Intel MKL冗长模式，需要执行以下两者之一：

- 设置环境变量MKL_VERBOSE为1，在bash下可以执行*export MKL_VERBOSE=1*
- 调用支持函数mkl_verbose(1)

函数调用mkl_verbose(0)将停止冗长模式。调用启用或禁止冗长模式的函数将覆盖掉环境变量设置。关于mkl_verbose函数，请参看Intel MKL Reference Manual。

Intel MKL冗长模式不是线程局域的，而是全局状态。这意味着如果一个应用从多线程中改变模式，其结果将是未定义的。

20 性能优化等

请参见Intel MKL官方手册。

Part X

应用程序的编译与安装

应用程序一般有两种方式发布：

- 二进制方式:用户无需编译,只要解压缩后设置相关环境变量等即可。如Gaussian[13]，国内用户只能购买到已经编译好的二进制可执行文件，有些国家和地区能购买到源代码。
- 源代码方式：
  - 用户需要自己编译，并且可以按照需要修改编译参数以编译成最适合自己的可执行程序，之后再设置环境变量等使用，如VASP[14]。
  - 源代码编译时经常用到的编译命令为*make*，编译配置文件为*Makefile*，请查看*make*命令用法及*Makefile*文件说明。

应用程序一般都有官方的安装说明，建议在安装前，首先仔细查看一下，比如到其主页或者查看解压缩后的目录中的类似：install*、readme*等文件。

21 二进制程序的安装

以二进制方式发布的程序，安装相对简单，一般只要解压缩后设置好环境变量即可，以Gaussian09为例：

- 将压缩包复制到某个地方，如*/opt*
- 解压缩：*tar xvf gaussian09.tar.gz*[15]
- 设置环境变量：修改*~/.bashrc*，添加：

```
##Add for g09
export g09root="/opt"
export GAUSS_SCRDIR="/tmp"
. $g09root/g09/bsd/g09.profile
##End for g09
```

- 刷新环境设置：*. ~/.bashrc*或重新登录下。

[13]Gaussian主页：http://www.gaussian.com/
[14]VASP主页：http://www.vasp.at/
[15]当前主流Linux系统，*tar*命令已经能自动识别.gz和.bz压缩，无需再明确添加z或j参数来指定。

22 源代码程序的安装

以源代码发布的程序安装相对复杂，需了解所采用的编译环境，并对配置等做相应修改（主要修改编译命令、库、头文件等编译参数等）。以VASP为例：

- 查看安装说明:其主页上的文档:`http://www.vasp.at/index.php/documentation`
- 解压缩文件：
  - *tar xvf vasp.5.lib.tar.gz*
  - *tar xvf vasp.5.2.tar.gz*
- 查看说明：install、readme文件等，VASP解压后的目录中未含有，可以参见上述主页文档安装。
- 生成默认配置文件：*./configure*。
  - VASP不需要*./configure*命令生成*Makefile*，而是提供了几个针对不同系统和编译器的makefile模板，可以复制*makefile.linux_ifc_P4*成*Makefile*
  - 其它程序也许需要*./configure*生成所需要的*Makefile*，在运行*./configure*之前，一般可以运行*./configure -h*查看其选项。如对Open MPI 1.6.4，可以运行以下命令生成*Makefile*：

    F77=ifort FC=ifort CC=icc CXX=icpc ./configure --prefix=/opt/openmpi-1.6.4

    其中：
    * F77：编译Fortran77源文件的编译器命令
    * FC：编译Fortran90源文件的编译器命令
    * CC：编译C源文件的编译器命令
    * CXX：编译C++源文件的编译器命令
    * –prefix：安装到的目录前缀

    另外一些在Makefile中常见变量为：
    * CPP：预处理参数
    * CLAGS：C程序编译参数
    * CXXFLAGS：C程序编译参数
    * F90：编译Fortran90及以后源文件的编译器命令
    * FFLAGS：Fortran编译参数
    * OFLAG：优化参数
    * INCLUDE：头文件参数
    * LIB：库文件参数

        * LINK：链接参数

- 修改*Makefile*文件配置，设定编译环境等：
    - 对*vasp.5.lib/Makefile*做如下修改：
        * 设定编译Fortran的编译器命令为Intel Fortran编译器命令：FC=ifort
    - 对*vasp.5/Makefile*做如下修改：
        * 设定BLAS库使用Intel MKL中的BLAS：BLAS=-mkl[16]
        * 打开FFT3D支持：去掉FFT3D = fft3dfurth.o fft3dlib.o前的#[17]
        * 设定MPI Fortran编译器为Intel MPI编译器：FC=mpiifort
- 编译：*make*
    - 先在*vasp.5.lib*目录中执行*make*
    - 如未出错，则再在*vasp.5.2*目录中执行*make*
- 安装：*make install*。VASP不需要，有些程序需要执行此步。
- 设置环境变量：比如在*~/.bashrc*中设置安装后的可执行程序目录在环境变量*PATH*中：

```
export PATH=$PATH:/opt/vasp.5.2
```

[16]因为2013版本的Intel编译器支持-mkl选项自动Intel MKL库，因此可以这么设置。
[17]在Makefile中#表示注释

Part XI

Slurm作业管理系统

Slurm(Simple Linux Utility for Resource Management, http://slurm.schedmd.com/)是开源的、具有容错性和高度可扩展大型和小型Linux集群资源管理和作业调度系统。超级计算系统可利用Slurm进行资源和作业管理，以避免相互干扰，提高运行效率。所有需运行的作业无论是用于程序调试还是业务计算均必须通过交互式并行*srun*、批处理式*sbatch*或分配式*salloc*等命令提交，提交后可以利用相关命令查询作业状态等。请不要在登录节点直接运行作业（编译除外），以免影响其余用户的正常使用。

本系统安装的Slurm版本为19.05.5，安装在*/usr*等系统默认目录，用户无需自己设置即可使用。

23 基本概念

23.1 三种模式区别

- 批处理作业（采用*sbatch*命令提交，最常用方式）:

  对于批处理作业（提交后立即返回该命令行终端，用户可进行其它操作）使用*sbatch*命令提交作业脚本，作业被调度运行后，在所分配的首个节点上执行作业脚本。在作业脚本中也可使用*srun*命令加载作业任务。提交时采用的命令行终端终止，也不影响作业运行。

- 交互式作业提交（采用*srun*命令提交）:

  资源分配与任务加载两步均通过*srun*命令进行：当在登录shell中执行*srun*命令时，*srun*首先向系统提交作业请求并等待资源分配，然后在所分配的节点上加载作业任务。采用该模式，用户在该终端需等待任务结束才能继续其它操作，在作业结束前，如果提交时的命令行终端断开，则任务终止。一般用于短时间小作业测试。

- 实时分配模式作业（采用*salloc*命令提交）:

  分配作业模式类似于交互式作业模式和批处理作业模式的融合。用户需指定所需要的资源条件，向资源管理器提出作业的资源分配请求。提交后，作业处于排队，当用户请求资源被满足时，将在用户提交作业的节点上执行用户所指定的命令，指定的命令执行结束后，运行结束，用户申请的资源被释放。在作业结束前，如果提交时的命令行终端断开，则任务终止。典型用途是分配资源并启动一个shell，然后在这个shell中利用*srun*运行并行作业。

*salloc*后面如果没有跟定相应的脚本或可执行文件，则默认选择/bin/sh，用户获得了一个合适环境变量的shell环境。

*salloc*和*sbatch*最主要的区别是*salloc*命令资源请求被满足时，直接在提交作业的节点执行相应任务，而*sbatch*则当资源请求被满足时，在分配的第一个节点上执行相应任务。

*salloc*在分配资源后，再执行相应的任务，很适合需要指定运行节点和其它资源限制，并有特定命令的作业。

23.2 基本用户命令

- *sacct*：显示激活的或已完成作业或作业步的记账（对应需缴纳的机时费）信息。
- *salloc*：为需实时处理的作业分配资源，典型场景为分配资源并启动一个shell，然后用此shell执行*srun*命令去执行并行任务。
- *sattach*：吸附到运行中的作业步的标准输入、输出及出错，通过吸附，使得有能力监控运行中的作业步的IO等。
- *sbatch*：提交作业脚本使其运行。此脚本一般也可含有一个或多个*srun*命令启动并行任务。
- *sbcast*：将本地存储中的文件传递分配给作业的节点上，比如*/tmp*等本地目录；对于*/home*等共享目录，因各节点已经是同样文件，无需使用。
- *scancel*：取消排队或运行中的作业或作业步，还可用于发送任意信号到运行中的作业或作业步中的所有进程。
- *scontrol*：显示或设定Slurm作业、队列、节点等状态。
- *sinfo*：显示队列或节点状态，具有非常多过滤、排序和格式化等选项。
- *speek*：查看作业屏幕输出。注：该命令是本人写的，不是slurm官方命令，在其它系统上不一定有。
- *squeue*：显示队列中的作业及作业步状态，含非常多过滤、排序和格式化等选项。
- *srun*：实时交互式运行并行作业，一般用于段时间测试，或者与*sallcoc*及*sbatch*结合。

23.3 基本术语

- socket：CPU插槽，可以简单理解为CPU。
- core：CPU核，单颗CPU可以具有多颗CPU核。
- job：作业。
- job step：作业步，单个作业（job）可以有个多作业步。
- tasks：任务数，单个作业或作业步可有多个任务，一般一个任务需一个CPU核，可理解为所需的CPU核数。
- rank：秩，如MPI进程号。
- partition：队列、分区。作业需在特定队列中运行，一般不同队列允许的资源不一样，比如单作业核数等。
- stdin：标准输入文件，一般指可以通过屏幕输入或采用<文件名方式传递给程序的文件，对应C程序中的文件描述符0。
- stdout：标准输出文件，程序运行正常时输出信息到的文件，一般指输出到屏幕的，并可采用>文件名定向到的文件，对应C程序中的文件描述符1。
- stderr：标准出错文件，程序运行出错时输出信息到的文件，一般指也输出到屏幕，并可采用2>定向到的文件（注意这里的2），对应C程序中的文件描述符2。

23.4 常用参考

- 作业提交：
  - *salloc*：为需实时处理的作业分配资源，提交后等获得作业分配的资源后运行，作业结束后返回命令行终端。
  - *sbatch*：批处理提交，提交后无需等待立即返回命令行终端。
  - *srun*：运行并行作业，等获得作业分配的资源并运行，作业结束后返回命令行终端。

  常用参数：

  - --begin=<time>：设定作业开始运行时间，如--begin="18:00:00"。
  - --constraints<features>：设定需要的节点特性。
  - --cpu-per-task：需要的CPU核数。
  - --error=<filename>：设定存储出错信息的文件名。

 - --exclude=<names>：设定不采用（即排除）运行的节点。
 - --dependency=<state:jobid>：设定只有当作业号的作业达到某状态时才运行。
 - --exclusive[=user|mcs]：设定排它性运行，不允许该节点有它人或某user用户或mcs的作业同时运行。
 - --export=<name[=value]>：输出环境变量给作业。
 - --gres=<name[:count]>：设定需要的通用资源。
 - --input=<filename>：设定输入文件名。
 - --job-name=<name>：设定作业名。
 - --label：设定输出时前面有标记（*仅限srun*）。
 - --mem=<size[unit]>：设定每个节点需要的内存。
 - --mem-per-cpu=<size[unit]>：设定每个分配的CPU所需的内存。
 - -N<minnodes[-maxnodes]>：设定所需要的节点数。
 - -n：设定启动的任务数。
 - --nodelist=<names>：设定需要的特定节点名，格式类似node[1-10,11,13-28]。
 - --output=<filename>：设定存储标准输出信息的文件名。
 - --partition=<name>：设定采用的队列。
 - --qos=<name>：设定采用的服务质量(QOS)。
 - --signal=[B:]<num>[@time]：设定当时间到时发送给作业的信号。
 - --time=<time>：设定作业运行时的墙上时钟限制。
 - --wrap=<command_strings>:将命令封装在一个简单的sh shell中运行(*仅限sbatch*)。

- 服务质量(QoS)：*sacctmgr*
 - *sacctmgr list qos*或*sacctmgr show qos*：显示QOS

- 记账信息：*sacct*
 - --endtime=<time>：设定显示的截止时间之前的作业。
 - --format=<spec>：格式化输出。
 - --name=<jobname>：设定显示作业名的信息。
 - --partition=<name>：设定采用队列的作业信息。
 - --state=<state_list>：显示特定状态的作业信息。

- 作业管理

- *scancel*：取消作业
  - jobid<job_id_list>：设定作业号。
  - --name=<name>：设定作业名。
  - --partition=<name>：设定采用队列的作业。
  - --qos=<name>：设定采用的服务质量(QOS)的作业。
  - --reservation=<name>：设定采用了预留测略的作业。
  - --nodelist=<name>：设定采用特定节点名的作业，格式类似node[1-10,11,13-28]。
- *squeue*：查看作业信息
  - --format=<spec>：格式化输出。
  - --jobid<job_id_list>：设定作业号。
  - --name=<name>：设定作业名。
  - --partition=<name>：设定采用队列的作业。
  - --qos=<name>：设定采用的服务质量(QOS)的作业。
  - --start：显示作业开始时间。
  - --state=<state_list>：显示特定状态的作业信息。
- *scontrol*：查看作业、节点和队列等信息
  - --details：显示更详细信息。
  - --oneline：所有信息显示在同一行。
  - show ENTITY ID：显示特定入口信息，ENTITY可为：job、node、partition等，ID可为作业号、节点名、队列名等。
  - update SPECIFICATION：修改特定信息，用户一般只能修改作业的。

24 显示队列、节点信息：sinfo

*sinfo*可以查看系统存在什么队列、节点及其状态。如*sinfo -l*：

```
PARTITION  AVAIL TIMELIMIT  JOB_SIZE ROOT OVERSUBS GROUPS NODES      STATE NODELIST
CPU-Large*    up   infinite 1-infinite no      NO       all   720       idle cnode[001-720]
GPU-V100      up   infinite 1-infinite no      NO       all    10       idle gnode[01-10]
2TB-AEP-Mem   up   infinite 1-infinite no      NO       all     8      mixed anode[01-08]
ARM-CPU       up   infinite 1-infinite no      NO       all     2      down* rnode[01,09]
ARM-CPU       up   infinite 1-infinite no      NO       all     2  allocated rnode[02-03]
ARM-CPU       up   infinite 1-infinite no      NO       all     5       idle rnode[04-08]
```

注：系统队列根据需要会调整，请根据上述命令确定可用队列，该文档后面部分采用batch作为队列名，并不是真正的队列名。

24.1 主要输出项

- AVAIL：up表示可用，down表示不可用。
- CPUS：各节点上的CPU数。
- S:C:T：各节点上的CPU插口sockets(S)数（CPU颗数，一颗CPU含有多颗CPU核，以下类似）、CPU核cores(C)数和线程threads(T)数。
- SOCKETS：各节点CPU插口数，CPU颗数。
- CORES：各节点CPU核数。
- THREADS：各节点线程数。
- GROUPS：可使用的用户组，all表示所有组都可以用。
- JOB_SIZE：可供用户作业使用的最小和最大节点数，如果只有1个值，则表示最大和最小一样，infinite表示无限制。
- TIMELIMIT：作业运行墙上时间（walltime，指的是用计时器，如手表或挂钟，度量的实际时间）限制，infinite表示没限制，如有限制的话，其格式为“days-hours:minutes:seconds”。
- MEMORY：实际内存大小，单位为MB。
- NODELIST：节点名列表，格式类似node[1-10,11,13-28]。
- NODES：节点数。
- NODES(A/I)：节点数，状态格式为“available/idle”。
- NODES(A/I/O/T)：节点数，状态格式为“available/idle/other/total”。
- PARTITION：队列名，后面带有*的，表示此队列为默认队列。
- ROOT：是否限制资源只能分配给root账户。
- OVERSUBSCRIBE：是否允许作业分配的资源超过计算资源（如CPU数）：
  - no：不允许超额。
  - exclusive：排他的，只能给这些作业用（等价于*srun --exclusive*）。
  - force：资源总被超额。
  - yes：资源可以被超额。

- STATE：节点状态，可能的状态包括：
  - allocated、alloc：已分配。
  - completing、comp：完成中。
  - down：宕机。
  - drained、drain：已失去活力。
  - draining、drng：失去活力中。
  - fail：失效。
  - failing、failg：失效中。
  - future、futr：将来可用。
  - idle：空闲，可以接收新作业。
  - maint：保持。
  - mixed：混合，节点在运行作业，但有些空闲CPU核，可接受新作业。
  - perfctrs、npc：因网络性能计数器使用中导致无法使用。
  - power_down、pow_dn：已关机。
  - power_up、pow_up：正在开机中。
  - reserved、resv：预留。
  - unknown、unk：未知原因。

  注意，如果状态带有后缀*，表示节点没响应。
- TMP_DISK：/tmp所在分区空间大小，单位为MB。

24.2 主要参数

- -a、--all：显示全部队列信息，如显示隐藏队列或本组没有使用权的队列。
- -d、--dead：仅显示无响应或已宕机节点。
- -e、--exact：精确而不是分组显示显示各节点。
- --help：显示帮助。
- -i <seconds>、--iterate=<seconds>：以<seconds>秒间隔持续自动更新显示信息。
- -l、--long：显示详细信息。
- -n <nodes>、--nodes=<nodes>：显示<nodes>节点信息。

- -N, --Node：以每行一个节点方式显示信息，即显示各节点信息。
- -p <partition>、--partition=<partition>：显示<partition>队列信息。
- -r、--responding：仅显示响应的节点信息。
- -R、--list-reasons：显示不响应（down、drained、fail或failing状态）节点的原因。
- -s：显示摘要信息。
- -S <sort_list>、--sort=<sort_list>：设定显示信息的排序方式。排序字段参见后面输出格式部分，多个排序字段采用,分隔，字段前面的+和-分表表示升序（默认）或降序。队列字段P前面如有#，表示以Slurm配置文件slurm.conf中的顺序显示。例如：*sinfo -S +P,-m*表示以队列名升序及内存大小降序排序。
- -t <states>、--states=<states>：仅显示<states>状态的信息。<states>状态可以为（不区分大小写）：ALLOC、ALLOCATED、COMP、COMPLETING、DOWN、DRAIN、DRAINED、DRAINING、ERR、ERROR、FAIL、FUTURE、FUTR、IDLE、MAINT、MIX、MIXED、NO_RESPOND、NPC、PERFCTRS、POWER_DOWN、POWER_UP、RESV、RESERVED、UNK和UNKNOWN。
- -T, --reservation：仅显示预留资源信息。
- --usage：显示用法。
- -v、--verbose：显示冗余信息，即详细信息。
- -V：显示版本信息。
- -o <output_format>、--format=<output_format>：按照<output_format>格式输出信息，默认为“%#P %.5a %.10l %.6D %.6t %N”：
  - %all：所有字段信息。
  - %a：队列的状态及是否可用。
  - %A：以“allocated/idle”格式显示状态对应的节点数。
  - %b：激活的特性，参见%f。
  - %B：队列中每个节点可分配给作业的CPU数。
  - %c：各节点CPU数。
  - %C：以“allocated/idle/other/total”格式状态显示CPU数。
  - %d：各节点临时磁盘空间大小，单位为MB。
  - %D：节点数。

- %e：节点空闲内存。
- %E：节点无效的原因（down、draine或ddraining状态）。
- %f：节点可用特性，参见%b。
- %F：以“allocated/idle/other/total”格式状态的节点数。
- %g：可以使用此节点的用户组。
- %G：与节点关联的通用资源（gres）。
- %h：作业是否能超用计算资源（如CPUs），显示结果可以为yes、no、exclusive或force。
- %H：节点不可用信息的时间戳。
- %I：队列作业权重因子。
- %l：以“days-hours:minutes:seconds”格式显示作业可最长运行时间。
- %L：以“days-hours:minutes:seconds”格式显示作业默认时间。
- %m：节点内存，单位MB。
- %M：抢占模式，可以为no或yes。
- %n：节点主机名。
- %N：节点名。
- %o：节点IP地址。
- %O：节点负载。
- %p：队列调度优先级。
- %P：队列名，带有*为默认队列，参见%R。
- %R：队列名，不在默认队列后附加*，参见%P。
- %s：节点最大作业大小。
- %S：允许分配的节点数。
- %t：以紧凑格式显示节点状态。
- %T：以扩展格式显示节点状态。
- %v：slurmd守护进程版本。
- %w：节点调度权重。
- %X：单节点socket数。
- %Y：单节点CPU核数。
- %Z：单核进程数。
- %z：扩展方式显示单节点处理器信息：sockets、cores、threads（S:C:T）数。

- -O <output_format>, --Format=<output_format>：按照<output_format>格式输出信息，类似-o <output_format>、--format=<output_format>。

  每个字段的格式为“type[:[.]size]”：

  – size：最小字段大小，如没指明，则最大为20个字符。
  – .：指明为右对齐，默认为左对齐。
  – 可用type：
    * all：所有字段信息。
    * allocmem：节点上分配的内存总数，单位MB。
    * allocnodes：允许分配的节点。
    * available：队列的State/availability状态。
    * cpus：各节点CPU数。
    * cpusload：节点负载。
    * freemem：节点可用内存，单位MB。
    * cpusstate：以“allocated/idle/other/total”格式状态的CPU数。
    * cores：单CPU颗CPU核数。
    * disk：各节点临时磁盘空间大小，单位为MB。
    * features：节点可用特性，参见features_act。
    * features_act：激活的特性，参见features。
    * groups：可以使用此节点的用户组。
    * gres：与节点关联的通用资源（gres）。
    * maxcpuspernode：队列中各节点最大可用CPU数。
    * memory：节点内存，单位MB。
    * nodeai：以“allocated/idle”格式显示状态对应的节点数。
    * nodes：节点数。
    * nodeaiot：以“allocated/idle/other/total”格式状态的节点数。
    * nodehost：节点主机名。
    * nodelist：节点名，格式类似node[1-10,11,13-28]。
    * oversubscribe：作业是否能超用计算资源（如CPUs），显示结果可以为yes、no、exclusive或force。
    * partition：队列名，带有*为默认队列，参见%R。
    * partitionname：队列名，默认队列不附加*，参见%P。
    * preemptmode：抢占模式，可以为no或yes。
    * priorityjobfactor：队列作业权重因子。
    * prioritytier或priority：队列调度优先级。

  * reason：节点无效的原因（down、draine或ddraining状态）。
  * size：节点最大作业数。
  * statecompact：紧凑格式节点状态。
  * statelong：扩展格式节点状态。
  * sockets：各节点CPU颗数。
  * socketcorethread:扩展方式显示单节点处理器信息:sockets、cores、threads（S:C:T）数。
  * time：以“days-hours:minutes:seconds”格式显示作业可最长运行时间。
  * timestamp：节点不可用信息的时间戳。
  * threads：CPU核线程数。
  * weight：节点调度权重。
  * version：slurmd守护进程版本。

25 查看队列中的作业信息：squeue

显示队列中的作业信息。如*squeue*显示：

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
   75   ARM-CPU  arm_job     hmli  R       2:27      2 rnode[02-03]
   76  GPU-V100 gpu.slurm    hmli PD       0:00      5 (Resources)
```

25.1 主要输出项

- JOBID：作业号。
- PARTITION：队列名（分区名）。
- NAME：作业名。
- USER：用户名。
- ST：状态。
  - PD：排队中，PENDING。
  - R：运行中，RUNNING。
  - CA：已取消，CANCELLED。
  - CF：配置中，CONFIGURING。

  - CG：完成中，COMPLETING
  - CD：已完成，COMPLETED。
  - F：已失败，FAILED。
  - TO：超时，TIMEOUT。
  - NF：节点失效，NODE FAILURE。
  - SE：特殊退出状态，SPECIAL EXIT STATE。

- TIME：已运行时间。
- NODELIST(REASON)：分配给的节点名列表（原因）：
  - AssociationCpuLimit：作业指定的关联CPU已在用，作业最终会运行。
  - AssociationMaxJobsLimit：作业关联的最大作业数已到，作业最终会运行。
  - AssociationNodeLimit：作业指定的关联节点已在用，作业最终会运行。
  - AssociationJobLimit：作业达到其最大允许的作业数限制。
  - AssociationResourceLimit：作业达到其最大允许的资源限制。
  - AssociationTimeLimit：作业达到时间限制。
  - BadConstraints：作业含有无法满足的约束。
  - BeginTime：作业最早开始时间尚未达到。
  - Cleaning：作业被重新排入队列，并且仍旧在执行之前运行的清理工作。
  - Dependency：作业等待一个依赖的作业结束后才能运行。
  - FrontEndDown：没有前端节点可用于执行此作业。
  - InactiveLimit：作业达到系统非激活限制。
  - InvalidAccount：作业用户无效，建议取消该作业重新采用正确账户提交。
  - InvalidQOS：作业QOS无效，建议取消该作业重新采用正确QoS提交。
  - JobHeldAdmin：作业被系统管理员挂起。
  - JobHeldUser：作业被用户自己挂起。
  - JobLaunchFailure：作业无法被启动，有可能因为文件系统故障、无效程序名等。
  - Licenses：作业等待相应的授权。
  - NodeDown：作业所需的节点宕机。
  - NonZeroExitCode：作业停止时退出代码非零。
  - PartitionDown：作业所需的队列出于DOWN状态。

- PartitionInactive：作业所需的队列处于Inactive状态。
- PartitionNodeLimit：作业所需的节点超过所用队列当前限制。
- PartitionTimeLimit：作业所需的队列达到时间限制。
- PartitionCpuLimit：该作业使用的队列对应的CPU已经被使用，作业最终会运行。
- PartitionMaxJobsLimit：该作业使用的队列的最大作业数已到，作业最终会运行。
- PartitionNodeLimit：该作业使用的队列对指定的节点都已在用，作业最终会运行。
- Priority：作业所需的队列存在高等级作业或预留。
- Prolog：作业的PrologSlurmctld前处理程序仍旧在运行。
- QOSJobLimit：作业的QOS达到其最大作业数限制。
- QOSResourceLimit：作业的QOS达到其最大资源限制。
- QOSGrpCpuLimit：作业的QoS的指定所有CPU已被占用，作业最终会运行。
- QOSGrpMaxJobsLimit：作业的QoS的最大作业数已到，作业最终会运行。
- QOSGrpNodeLimit：作业的QoS指定的节点都已经被占用，作业最终会运行。
- QOSTimeLimit：作业的QOS达到其时间限制。
- QOSUsageThreshold：所需的QOS阈值被违反。
- ReqNodeNotAvail：作业所需的节点无效，如节点宕机。
- Reservation：作业等待其预留的资源可用。
- Resources：作业将要等待所需要的资源满足后才运行。
- SystemFailure：Slurm系统失效，如文件系统、网络失效等。
- TimeLimit：作业超过去时间限制。
- WaitingForScheduling：等待被调度中。

25.2 主要参数

- -A <account_list>, --account=<account_list>：显示用户<account_list>的作业信息，用户以,分隔。
- -a, --all：显示所有队列中的作业及作业步信息，也显示被配置为对用户组隐藏队列的信息。
- -r, --array：以每行一个作业元素方式显示。

- -h, --noheader：不显示头信息，即不显示第一行“PARTITION AVAIL TIMELIMIT NODES STATE NODELIST”。
- --help：显示帮助信息。
- --hide：不显示隐藏队列中的作业和作业步信息。此为默认行为，不显示配置为对用户组隐藏队列的信息。
- -i <seconds>, --iterate=<seconds>：以间隔<seconds>秒方式循环显示信息。
- -j <job_id_list>, --jobs=<job_id_list>：显示作业号<job_id_list>的作业，作业号以,分隔。--jobs=<job_id_list>可与--steps选项结合显示特定作业的步信息。作业号格式为“job_id[_array_id]”，默认为64字节，可以用环境变量SLURM_BITSTR_LEN设定更大的字段大小。
- -l, --long：显示更多的作业信息。
- -L, --licenses=<license_list>：指定使用授权文件<license_list>，以,分隔。
- -n, --name=<name_list>：显示具有特定<name_list>名字的作业，以,分隔。
- --noconvert：不对原始单位做转换，如2048M不转换为2G。
- -p <part_list>, --partition=<part_list>：显示特定队列<part_list>信息，<part_list>以,分隔。
- -P, --priority：对于提交到多个队列的作业，按照各队列显示其信息。如果作业要按照优先级排序时，需考虑队列和作业优先级。
- -q <qos_list>, --qos=<qos_list>：显示特定qos的作业和作业步，<qos_list>以,分隔。
- -R, --reservation=reservation_name：显示特定预留信息作业。
- -s, --steps：显示特定作业步。作业步格式为“job_id[_array_id].step_id”。
- -S <sort_list>, --sort=<sort_list>：按照显示特定字段排序显示，<sort_list>以,分隔。如-S P,U。
- --start：显示排队中的作业的预期执行时间。
- -t <state_list>, --states=<state_list>：显示特定状态<state_list>的作业信息。<state_list>以,分隔，有效的可为：PENDING(PD)、RUNNING(R)、SUSPENDED(S)、STOPPED(ST)、COMPLETING(CG)、COMPLETED(CD)、CONFIGURING(CF)、CANCELLED(CA)、FAILED(F)、TIMEOUT(TO)、PREEMPTED(PR)、BOOT_FAIL(BF)、NODE_FAIL(NF)和SPECIAL_EXIT(SE)，注意是不区分大小写的，如“pd”和“PD”是等效的。

- -u <user_list>, --user=<user_list>:显示特定用户<user_list>的作业信息,<user_list>以,分隔。
- --usage：显示帮助信息。
- -v, --verbose：显示squeue命令详细动作信息。
- -V, --version：显示版本信息。
- -w <hostlist>, --nodelist=<hostlist>：显示特定节点<hostlist>信息，<hostlist>以,分隔。
- -o <output_format>, --format=<output_format>：以特定格式<output_format>显示信息。参见 -O <output_format>, --Format=<output_format>，采用不同参数的默认格式为：
  - default：“%.18i %.9P %.8j %.8u %.2t %.10M %.6D %R”
  - -l, --long：“%.18i %.9P %.8j %.8u %.8T %.10M %.9l %.6D %R”
  - -s, --steps：“%.15i %.8j %.9P %.8u %.9M %N”

  每个字段的格式为“%[[.]size]type”：
  - size：字段最小尺寸，如果没有指定size，则按照所需长度显示。
  - .：右对齐显示，默认为左对齐。
  - type：类型，一些类型仅对作业有效，而有些仅对作业步有效，有效的类型为：
    * %all：显示所有字段。
    * %a：显示记帐信息（仅对作业有效）。
    * %A：作业步生成的任务数（仅适用于作业步）。
    * %A：作业号（仅适用于作业）。
    * %b：作业或作业步所需的普通资源（gres）。
    * %B：执行作业的节点。
    * %c：作业每个节点所需的最小CPU数（仅适用于作业）。
    * %C：如果作业还在运行，显示作业所需的CPU数；如果作业正在完成，显示当前分配给此作业的CPU数（仅适用于作业）。
    * %d：作业所需的最小临时磁盘空间，单位MB（仅适用于作业）。
    * %D：作业所需的节点（仅适用于作业）。
    * %e：作业结束或预期结束时间（基于其时间限制）（仅适用于作业）。
    * %E：作业依赖剩余情况。作业只有依赖的作业完成才运行，如显示NULL，则无依赖（仅适用于作业）。

* %f：作业所需的特性（仅适用于作业）。
* %F：作业组作业号（仅适用于作业）。
* %g：作业用户组（仅适用于作业）。
* %G：作业用户组ID（仅适用于作业）。
* %h：分配给此作业的计算资源能否被其它作业预约（仅适用于作业）。可被预约的资源包含节点、CPU颗、CPU核或超线程。值可以为：
  - YES：如果作业提交时含有oversubscribe选项或队列被配置含有OverSubscribe=Force。
  - NO：如果作业所需排他性运行。
  - USER：如果分配的计算节点设定为单个用户。
  - MCS:如果分配的计算节点设定为单个安全类(参看MCSPlugin和MCSParameters配置参数，Multi-Category Security)。
  - OK：其它（典型的分配给专用的CPU）（仅适用于作业）。
* %H：作业所需的单节点CPU数，显示srun --sockets-per-node提交选项，如--sockets-per-node未设定，则显示*（仅适用于作业）。
* %i:作业或作业步号,在作业组中,作业号格式为“<base_job_id>_<index>”，默认作业组索引字段限制到64字节,可以用环境变量*SLURM_BITSTR_LEN*设定为更大的字段大小。
* %I：作业所需的每颗CPU的CPU核数，显示的是srun --cores-per-socket设定的值，如--cores-per-socket未设定，则显示*（仅适用于作业）。
* %j：作业或作业步名。
* %J：作业所需的每个CPU核的线程数，显示的是srun --threads-per-core设定的值，如--threads-per-core未被设置则显示*（仅适用于作业）。
* %k：作业说明（仅适用于作业）。
* %K：作业组索引默认作业组索引字段限制到64字节，可以用环境变量*SLURM_BITSTR_LEN*设定为更大的字段大小（仅适用于作业）。
* %l:作业或作业步时间限制,格式为“days-hours:minutes:seconds”:NOT_SET表示没有建立；UNLIMITED表示没有限制。
* %L：作业剩余时间，格式为“days-hours:minutes:seconds”，此值由作业的时间限制减去已用时间得到：NOT_SET表示没有建立；UNLIMITED表示没有限制（仅适用于作业）。
* %m：作业所需的最小内存，单位为MB（仅适用于作业）。
* %M:作业或作业步已经使用的时间,格式为“days-hours:minutes:seconds”。
* %n：作业所需的节点名（仅适用于作业）。
* %N：作业或作业步分配的节点名，对于正在完成的作业，仅显示尚未释放资源回归服务的节点。

* %o：执行的命令。
* %O：作业是否需连续节点（仅适用于作业）。
* %p：作业的优先级（0.0到1.0之间），参见%Q（仅适用于作业）。
* %P：作业或作业步的队列。
* %q：作业关联服务的品质（仅适用于作业）。
* %Q：作业优先级（通常为非常大的一个无符号整数），参见%p（仅适用于作业）。
* %r：作业在当前状态的原因，参见JOB REASON CODES（仅适用于作业）。
* %R：参见JOB REASON CODES（仅适用于作业）：
  · 对于排队中的作业：作业没有执行的原因。
  · 对于出错终止的作业：作业出错的解释。
  · 对于其他作业状态：分配的节点。
* %S：作业或作业步实际或预期的开始时间。
* %t：作业状态，以紧凑格式显示：PD（排队pending）、R（运行running）、CA（取消cancelled）、CF(配置中configuring)、CG（完成中completing）、CD（已完成completed）、F（失败failed）、TO（超时timeout）、NF（节点失效node failure)和SE（特殊退出状态special exit state），参见JOB STATE CODES（仅适用于作业）。
* %T：作业状态，以扩展格式显示：PENDING、RUNNING、SUSPENDED、CANCELLED、COMPLETING、COMPLETED、CONFIGURING、FAILED、TIMEOUT、PREEMPTED、NODE_FAIL和SPECIAL_EXIT，参见JOB STATE CODES（仅适用于作业）。
* %u：作业或作业步的用户名。
* %U：作业或作业步的用户ID。
* %v：作业的预留资源（仅适用于作业）。
* %V：作业的提交时间。
* %w：工程量特性关键Workload Characterization Key（wckey）（仅适用于作业）。
* %W：作业预留的授权（仅适用于作业）。
* %x：作业排他性节点名（仅适用于作业）。
* %X：系统使用需每个节点预留的CPU核数（仅适用于作业）。
* %y：Nice值（调整作业调动优先级）（仅适用于作业）。
* %Y：对于排队中作业，显示其开始运行时期望的节点名。
* %z：作业所需的每个节点的CPU颗数、CPU核数和线程数（S:C:T），如（S:C:T）未设置，则显示*（仅适用于作业）。

    * %Z：作业的工作目录。

- -O <output_format>, --Format=<output_format>：以特定格式<output_format>显示信息，参见-o <output_format>, --format=<output_format> 每个字段的格式为"%[[.]size]type"：
  - size：字段最小尺寸，如果没有指定size，则最长显示20个字符。
  - .：右对齐显示，默认为左对齐。
  - type：类型，一些类型仅对作业有效，而有些仅对作业步有效，有效的类型为：
    * account：作业记账信息（仅适用于作业）。
    * allocnodes：作业分配的节点（仅适用于作业）。
    * allocsid：用于提交作业的会话ID（仅适用于作业）。
    * arrayjobid：作业组中的作业ID。
    * arraytaskid：作业组中的任务ID。
    * associd：作业关联ID（仅适用于作业）。
    * batchflag：是否批处理设定了标记（仅适用于作业）。
    * batchhost：执行节点（仅适用于作业）：
      · 对于分配的会话：显示的是会话执行的节点（如，srun或salloc命令执行的节点）。
      · 对于批处理作业：显示的执行批处理的节点。
    * chptdir：作业checkpoint的写目录（仅适用于作业步）。
    * chptinter：作业checkpoint时间间隔（仅适用于作业步）。
    * command：作业执行的命令（仅适用于作业）。
    * comment：作业关联的说明（仅适用于作业）。
    * contiguous：作业是否要求连续节点（仅适用于作业）。
    * cores:作业所需的每颗CPU的CPU核数,显示的是srun --cores-per-socket设定的值，如--cores-per-socket未设定，则显示*（仅适用于作业）。
    * corespec：为了系统使用所预留的CPU核数（仅适用于作业）。
    * cpufreq：分配的CPU主频（仅适用于作业步）。
    * cpuspertask：作业分配的每个任务的CPU颗数（仅适用于作业）。
    * deadline：作业的截止时间（仅适用于作业）。
    * dependency：作业依赖剩余。作业只有依赖的作业完成才运行，如显示NULL，则无依赖（仅适用于作业）。
    * derivedec：作业的起源退出码，对任意作业步是最高退出码（仅适用于作业）。

* eligibletime：预计作业开始运行时间（仅适用于作业）。
* endtime：作业实际或预期的终止时间（仅适用于作业）。
* exit_code：作业退出码（仅适用于作业）。
* feature：作业所需的特性（仅适用于作业）。
* gres：作业或作业步需的通用资源（gres）。
* groupid：作业用户组ID（仅适用于作业）。
* groupname：作业用户组名（仅适用于作业）。
* jobarrayid：作业组作业ID（仅适用于作业）。
* jobid：作业号（仅适用于作业）。
* licenses：作业预留的授权（仅适用于作业）。
* maxcpus：分配给作业的最大CPU颗数（仅适用于作业）。
* maxnodes：分配给作业的最大节点数（仅适用于作业）。
* mcslabel：作业的MCS_label（仅适用于作业）。
* minmemory：作业所需的最小内存大小，单位MB（仅适用于作业）。
* mintime：作业的最小时间限制（仅适用于作业）。
* mintmpdisk：作业所需的临时磁盘空间，单位MB（仅适用于作业）。
* mincpus：作业所需的各节点最小CPU颗数，显示的是srun --mincpus设定的值（仅适用于作业）。
* name：作业或作业步名。
* network：作业运行的网络。
* nice Nice值(调整作业调度优先值)（仅适用于作业）。
* nodes：作业或作业步分配的节点名，对于正在完成的作业，仅显示尚未释放资源回归服务的节点。
* nodelist：作业或作业步分配的节点，对于正在完成的作业，仅显示尚未释放资源回归服务的节点，格式类似node[1-10,11,13-28]。
* ntpercore：作业每个CPU核分配的任务数（仅适用于作业）。
* ntpernode：作业每个节点分配的任务数（仅适用于作业）。
* ntpersocket：作业每颗CPU分配的任务数（仅适用于作业）。
* numcpus：作业所需的或分配的CPU颗数。
* numnodes：作业所需的或分配的最小节点数（仅适用于作业）。
* numtask：作业或作业号需的任务数，显示的--ntasks设定的。
* oversubscribe：分配给此作业的计算资源能否被其它作业预约（仅适用于作业）。可被预约的资源包含节点、CPU颗、CPU核或超线程。值可以为：

    - YES：如果作业提交时含有oversubscribe选项或队列被配置含有OverSubscribe=Force。
    - NO：如果作业所需排他性运行。
    - USER：如果分配的计算节点设定为单个用户。
    - MCS：如果分配的计算节点设定为单个安全类（参看MCSPlugin和MCSParameters配置参数）。
    - OK：其它(典型分配给指定CPU)。

* partition：作业或作业步的队列。
* priority：作业的优先级（0.0到1.0之间），参见%Q（仅适用于作业）。
* prioritylong：作业优先级（通常为非常大的一个无符号整数），参见%p（仅适用于作业）。
* profile：作业特征（仅适用于作业）。
* preemptime：作业抢占时间（仅适用于作业）。
* qos：作业的服务质量（仅适用于作业）。
* reason：作业在当前的原因，参见JOB REASON CODES（仅适用于作业）。
* reasonlist：参见JOB REASON CODES（仅适用于作业）。
    - 对于排队中的作业：作业没有执行的原因。
    - 对于出错终止的作业：作业出错的解释。
    - 对于其他作业状态：分配的节点。
* reqnodes：作业所需的节点名（仅适用于作业）。
* requeue：作业失败时是否需重新排队运行（仅适用于作业）。
* reservation：预留资源（仅适用于作业）。
* resizetime：运行作业的变化时间总和（仅适用于作业）。
* restartcnt：作业的重启checkpoint数（仅适用于作业）。
* resvport：作业的预留端口（仅适用于作业步）。
* schednodes：排队中的作业开始运行时预期将被用的节点列表（仅适用于作业）。
* sct：各节点作业所需的CPU数、CPU核数和线程数（S:C:T），如（S:C:T）未设置，则显示*（仅适用于作业）。
* selectjobinfo：节点选择插件针对作业指定的数据，可能的数据包含：资源分配的几何维度（X、Y、Z维度）、连接类型（TORUS、MESH或NAV == torus else mesh），是否允许几何旋转（yes或no），节点使用（VIRTUAL或COPROCESSOR）等（仅适用于作业）。
* sockets：作业每个节点需的CPU数，显示srun时的--sockets-per-node选项，如--sockets-per-node未设置，则显示*（仅适用于作业）。

* sperboard：每个主板分配给作业的CPU数（仅适用于作业）。
* starttime：作业或作业布实际或预期开始时间。
* state：扩展格式作业状态：排队中PENDING、运行中RUNNING、已停止STOPPED、被挂起SUSPENDED、被取消CANCELLED、完成中COMPLETING、已完成COMPLETED、配置中CONFIGURING、已失败FAILED、超时TIMEOUT、预取PREEMPTED、节点失效NODE_FAIL、特定退出SPECIAL_EXIT，参见JOB STATE CODES部分（仅适用于作业）。
* statecompact：紧凑格式作业状态：PD（排队中pending）、R（运行中running）、CA（已取消cancelled）、CF(配置中configuring）、CG（完成中completing）、CD（已完成completed）、F（已失败failed）、TO（超时timeout）、NF（节点失效node failure）和SE（特定退出状态special exit state），参见JOB STATE CODES部分（仅适用于作业）。
* stderr：标准出错输出目录（仅适用于作业）。
* stdin：标准输入目录（仅适用于作业）。
* stdout：标准输出目录（仅适用于作业）。
* stepid：作业或作业步号。在作业组中，作业号格式为“<base_job_id>_<index>”（仅适用于作业步）。
* stepname：作业步名（仅适用于作业步）。
* stepstate：作业步状态（仅适用于作业步）。
* submittime：作业提交时间（仅适用于作业）。
* threads：作业所需的每颗CPU核的线程数，显示srun的--threads-per-core参数，如--threads-per-core未设置，则显示*（仅适用于作业）。
* timeleft：作业剩余时间，格式为“days-hours:minutes:seconds”，此值是通过其时间限制减去已运行时间得出的：如未建立则显示“NOT_SET”；如无限制则显示“UNLIMITED”（仅适用于作业）。
* timelimit：作业或作业步的时间限制。
* timeused：作业或作业步以使用时间，格式为“days-hours:minutes:seconds”，days和hours只有需要时才显示。对于作业步，显示从执行开始经过的时间，因此对于被曾挂起的作业并不准确。节点间的时间差也会导致时间不准确。如时间不对（如，负值），将显示“INVALID”。
* tres：显示分配给作业的可被追踪的资源。
* userid：作业或作业步的用户ID。
* username：作业或作业步的用户名。
* wait4switch：需满足转轨器数目的总等待时间（仅适用于作业）。
* wckey：工作负荷特征关键（wckey）（仅适用于作业）。
* workdir：作业工作目录（仅适用于作业）。

26 查看详细队列信息：scontrol show partition

*scontrol show partition*显示全部队列信息，*scontrol show partition PartitionName*或*scontrol show partition=PartitionName*显示队列名PartitionName的队列信息，输出类似：

```
PartitionName=CPU-Large
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=YES QoS=N/A
   DefaultTime=NONE DisableRootJobs=YES ExclusiveUser=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=0 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=cnode[001-720]
   PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO OverSubscribe=NO
   OverTimeLimit=NONE PreemptMode=OFF
   State=UP TotalCPUs=28800 TotalNodes=720 SelectTypeParameters=NONE
   JobDefaults=(null)
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED

PartitionName=GPU-V100
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=NO QoS=N/A
   DefaultTime=NONE DisableRootJobs=YES ExclusiveUser=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=0 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=gnode[01-10]
   PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO OverSubscribe=NO
   OverTimeLimit=NONE PreemptMode=OFF
   State=UP TotalCPUs=400 TotalNodes=10 SelectTypeParameters=NONE
   JobDefaults=(null)
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED

PartitionName=2TB-AEP-Mem
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=NO QoS=N/A
   DefaultTime=NONE DisableRootJobs=YES ExclusiveUser=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=0 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=anode[01-08]
   PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO OverSubscribe=NO
   OverTimeLimit=NONE PreemptMode=OFF
   State=UP TotalCPUs=320 TotalNodes=8 SelectTypeParameters=NONE
   JobDefaults=(null)
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED

PartitionName=ARM-CPU
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
```

```
AllocNodes=ALL Default=NO QoS=N/A
DefaultTime=NONE DisableRootJobs=YES ExclusiveUser=NO GraceTime=0 Hidden=NO
MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=0 LLN=NO MaxCPUsPerNode=UNLIMITED
Nodes=rnode[01-09]
PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO OverSubscribe=NO
OverTimeLimit=NONE PreemptMode=OFF
State=UP TotalCPUs=864 TotalNodes=9 SelectTypeParameters=NONE
JobDefaults=(null)
DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
```

26.1 主要输出项

- PartitionName：队列名。
- AllowGroups：允许的用户组。
- AllowAccounts：允许的用户。
- AllowQos：允许的QoS。
- AllocNodes：允许的节点。
- Default：是否为默认队列。
- QoS：服务质量。
- DefaultTime：默认时间。
- DisableRootJobs：是否禁止root用户提交作业。
- ExclusiveUser：排除的用户。
- GraceTime：抢占的款显时间，单位秒。
- Hidden：是否为隐藏队列。
- MaxNodes：最大节点数。
- MaxTime：最大运行时间。
- MinNodes：最小节点数。
- LLN：是否按照最小负载节点调度。
- MaxCPUsPerNode：每个节点的最大CPU颗数。

- Nodes：节点名。
- PriorityJobFactor：作业因子优先级。
- PriorityTier：调度优先级。
- RootOnly：是否只允许Root。
- ReqResv：要求预留的资源。
- OverSubscribe：是否允许超用。
- PreemptMode：是否为抢占模式。
- State：状态：
  - UP：可用，作业可以提交到此队列，并将运行。
  - DOWN：作业可以提交到此队列，但作业也许不会获得分配开始运行。已运行的作业还将继续运行。
  - DRAIN：不接受新作业，已接受的作业可以被运行。
  - INACTIVE：不接受新作业，已接受的作业未开始运行的也不运行。
- TotalCPUs：总CPU核数。
- TotalNodes：总节点数。
- SelectTypeParameters：资源选择类型参数。
- DefMemPerNode：每个节点默认分配的内存大小，单位MB。
- MaxMemPerNode：每个节点最大内存大小，单位MB。

27 查看详细节点信息：scontrol show node

*scontrol show node*显示全部节点信息，*scontrol show node NODENAME*或*scontrol show node=NODENAME*显示节点名NODENAME的节点信息，输出类似：

```
NodeName=anode01 Arch=x86_64 CoresPerSocket=20
   CPUAlloc=0 CPUTot=40 CPULoad=0.01
   AvailableFeatures=(null)
   ActiveFeatures=(null)
   Gres=(null)
   NodeAddr=anode01 NodeHostName=anode01 Version=19.05.4
   OS=Linux 3.10.0-1062.el7.x86_64 #1 SMP Wed Aug 7 18:08:02 UTC 2019
```

```
  RealMemory=2031623 AllocMem=0 FreeMem=1989520 Sockets=2 Boards=1
  State=IDLE ThreadsPerCore=1 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
  Partitions=2TB-AEP-Mem
  BootTime=2019-11-09T15:47:56 SlurmdStartTime=2019-12-01T19:01:59
  CfgTRES=cpu=40,mem=2031623M,billing=40
  AllocTRES=
  CapWatts=n/a
  CurrentWatts=0 AveWatts=0
  ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s

NodeName=gnode01 Arch=x86_64 CoresPerSocket=20
  CPUAlloc=0 CPUTot=40 CPULoad=0.01
  AvailableFeatures=(null)
  ActiveFeatures=(null)
  Gres=gpu:v100:2
  NodeAddr=gnode01 NodeHostName=gnode01 Version=19.05.4
  OS=Linux 3.10.0-1062.el7.x86_64 #1 SMP Wed Aug 7 18:08:02 UTC 2019
  RealMemory=385560 AllocMem=0 FreeMem=368966 Sockets=2 Boards=1
  State=IDLE ThreadsPerCore=1 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
  Partitions=GPU-V100
  BootTime=2019-11-13T16:51:31 SlurmdStartTime=2019-12-01T19:54:55
  CfgTRES=cpu=40,mem=385560M,billing=40,gres/gpu=2
  AllocTRES=
  CapWatts=n/a
  CurrentWatts=0 AveWatts=0
  ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
```

27.1 主要输出项

- NodeName：节点名。
- Arch：系统架构。
- CoresPerSocket：12。
- CPUAlloc：分配给的CPU核数。
- CPUErr：出错的CPU核数。
- CPUTot：总CPU核数。
- CPULoad：CPU负载。
- AvailableFeatures：可用特性。

- ActiveFeatures：激活的特性。
- Gres：通用资源。如上面Gres=gpu:v100:2指明了有两块V100 GPU。
- NodeAddr：节点IP地址。
- NodeHostName：节点名。
- Version：Slurm版本。
- OS：操作系统。
- RealMemory：实际物理内存，单位GB。
- AllocMem：已分配内存，单位GB。
- FreeMem：可用内存，单位GB。
- Sockets：CPU颗数。
- Boards：主板数。
- State：状态。
- ThreadsPerCore：每颗CPU核线程数。
- TmpDisk：临时存盘硬盘大小。
- Weight：权重。
- BootTime：开机时间。
- SlurmdStartTime：Slurmd守护进程启动时间。

28 查看详细作业信息：scontrol show job

*scontrol show job*显示全部作业信息,*scontrol show job JOBID*或*scontrol show job=JOBID*显示作业号为JOBID的作业信息，输出类似下面：

```
JobId=77 JobName=gres_test.bash
   UserId=hmli(10001) GroupId=nic(10001) MCS_label=N/A
   Priority=4294901755 Nice=0 Account=(null) QOS=normal
   JobState=RUNNING Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   RunTime=00:00:11 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2019-12-01T20:10:15 EligibleTime=2019-12-01T20:10:15
```

```
AccrueTime=2019-12-01T20:10:15
StartTime=2019-12-01T20:10:16 EndTime=Unknown Deadline=N/A
SuspendTime=None SecsPreSuspend=0 LastSchedEval=2019-12-01T20:10:16
Partition=GPU-V100 AllocNode:Sid=login01:1016
ReqNodeList=(null) ExcNodeList=(null)
NodeList=cnode[188-189]
BatchHost=cnode188
NumNodes=2 NumCPUs=80 NumTasks=80 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
TRES=cpu=80,node=2,billing=80
Socks/Node=* NtasksPerN:B:S:C=40:0:*:* CoreSpec=*
MinCPUsNode=40 MinMemoryNode=0 MinTmpDiskNode=0
Features=(null) DelayBoot=00:00:00
OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
Command=/home/nic/hmli/gres_test.bash
WorkDir=/home/nic/hmli
StdErr=/home/nic/hmli/job-77.err
StdIn=/dev/null
StdOut=/home/nic/hmli/job-77.log
Power=
```

注：billing部分显示的是记账信息，上述作业表示按照80核计算费用。

28.1 主要输出项

- JobId：作业号。
- JobName：作业名。
- UserId：用户名（用户ID）。
- GroupId：用户组（组ID）。
- MCS_label：。
- Priority：优先级，越大越优先，如果为0则表示被管理员挂起，不允许运行。
- Nice：Nice值，越小越优先，-20到19。
- Account：记账用户名。
- QOS：作业的服务质量。
- JobState：作业状态。

  - PENDING：排队中。
  - RUNNING：运行中。
  - CANCELLED：已取消。
  - CONFIGURING：配置中。
  - COMPLETING：完成中。
  - COMPLETED：已完成。
  - FAILED：已失败。
  - TIMEOUT：超时。
  - NODE FAILURE：节点失效。
  - SPECIAL EXIT STATE：特殊退出状态。
- Reason：原因。
- Dependency：依赖关系。
- Requeue：节点失效时，是否重排队，0为否，1为是。
- Restarts：失败时，是否重运行，0为否，1为是。
- BatchFlag：是否为批处理作业，0为否，1为是。
- Reboot：节点空闲时是否重启节点，0为否，1为是。
- ExitCode：作业退出代码。
- RunTime：已运行时间。
- TimeLimit：作业允许的剩余运行时间。
- TimeMin：最小时间。
- SubmitTime：提交时间。
- EligibleTime：获得认可时间。
- StartTime：开始运行时间。
- EndTime：预计结束时间。
- Deadline：截止时间。
- PreemptTime：先占时间。
- SuspendTime：挂起时间。

- SecsPreSuspend：0。
- Partition：对列名。
- AllocNode：Sid：分配的节点：系统ID号。
- ReqNodeList：需要的节点列表，格式类似node[1-10,11,13-28]。
- ExcNodeList：排除的节点列表，格式类似node[1-10,11,13-28]。
- NodeList：实际运行节点列表，格式类似node[1-10,11,13-28]。
- BatchHost：批处理节点名。
- NumNodes：节点数。
- NumCPUs：CPU核数。
- NumTasks：任务数。
- CPUs/Task：CPU核数/任务数。
- ReqB:S:C:T：所需的主板数:每主板CPU颗数:每颗CPU核数:每颗CPU核的线程数，<baseboard_count>:<socket_per_baseboard_count>:<core_per_socket_count>:<thread_per_core_count>。
- TRES：显示分配给作业的可被追踪的资源。
- Socks/Node：每节点CPU颗数。
- NtasksPerN:B:S:C：每主板数:每主板CPU颗数:每颗CPU的核数:每颗CPU核的线程数启动的作业数，<tasks_per_node>:<tasks_per_baseboard>:<tasks_per_socket>:<tasks_per_core>。
- CoreSpec：各节点系统预留的CPU核数，如未包含，则显示*。
- MinCPUsNode：每节点最小CPU核数。
- MinMemoryNode：每节点最小内存大小，0表示未限制。
- MinTmpDiskNode：每节点最小临时存盘硬盘大小，0表示未限制。
- Features：特性。
- Gres：通用资源。
- Reservation：预留资源。
- OverSubscribe：是否允许与其它作业共享资源，OK允许，NO不允许。

- Contiguous：是否要求分配连续节点，OK是，NO否。
- Licenses：软件授权。
- Network：网络。
- Command：作业命令。
- WorkDir：工作目录。
- StdErr：标准出错输出文件。
- StdIn：标准输入文件。
- StdOut：标准输出文件。

29 查看服务质量(QoS)

服务质量(Quality of Service-QoS)，或者理解为资源限制或者优先级，只有达到QoS要求时作业才能运行，QoS将在以下三个方面影响作业运行：

- 作业调度优先级
- 作业抢占
- 作业限制

可以用*sacctmgr show|list qos*查看。

30 查看作业屏幕输出：speek

查看作业屏幕输出的命令*speek*（类似LSF的*bpeek*），基本用法*speek [-e] [-f] 作业号*。默认显示正常屏幕输出，如加-f参数，则连续监测输出；如加-e参数，则监测错误日志。

注：该*speek*命令是本人写的，不是slurm官方命令，在其它系统上不一定有。

```
#!/bin/bash
#Author: HM Li <hmli@ustc.edu.cn>
if [ $# -lt 1 ] ; then
    echo "Usage: speek [-e] [-f] jobid"
    echo "-e: show error log."
    echo -e "-f: output appended data as the file grows.\n\nYour jobs are:"
    if [ $USER != 'root' ]; then
```

```
        squeue -u $USER -t r -o "%.8i %10P %12j %19S %.12M %.7C %.5D %R"
    else
        squeue -t r -o "%.8i %10u %10P %12j %19S %.12M  %.7C %.5D %R"
    fi
    exit
fi
NO=1
STD=StdOut
while getopts 'ef' OPT; do
    case $OPT in
    e)
        STD=StdErr
        ;;
    f)
        T='-f'
        ;;
    esac
done
JOBID=${!#}
F=`scontrol show job $JOBID 2>/dev/null | awk -v STD=$STD -F= '{if($1~'STD') print $2}'`
if [ -f "$F" ]; then
    tail $T $F
else
    echo "Job $JOBID has no $STD file or you have no authority to access."
fi
```

31 提交作业命令共同说明

提交作业的命令主要有*salloc*、*sbatch*与*srun*，其多数参数、输入输出变量等都是一样的。

31.1 主要参数

- -A, --account=<account>：指定此作业的责任资源为账户<account>，即账单（与计算费对应）记哪个名下，只有账户属于多个账单组才有权指定。
- --accel-bind=<options>：*srun特有*，控制如何绑定作业到GPU、网络等特定资源，支持同时多个选项，支持的选项如下：
  - g：绑定到离分配的CPU最近的GPU

  - m：绑定到离分配的CPU最近的MIC
  - n：绑定到离分配的CPU最近的网卡
  - v：详细模式，显示如何绑定GPU和网卡等等信息

- --acctg-freq:指定作业记账和剖面信息采样间隔。支持的格式为--acctg-freq=<datatype>=<interval>其中<datatype>=<interval>指定了任务抽样间隔或剖面抽样间隔。多个<datatype>=<interval>可以采用,分隔（默认为30秒）：
  - task=<interval>：以秒为单位的任务抽样（需要jobacct_gather插件启用）和任务剖面（需要acct_gather_profile插件启用）间隔。
  - energy=<interval>:以秒为单位的能源剖面抽样间隔,需要acct_gather_energy插件启用。
  - network=<interval>:以秒为单位的InfiniBand网络剖面抽样间隔,需要acct_gather_infiniband插件启用。
  - filesystem=<interval>:以秒为单位的文件系统剖面抽样间隔,需要acct_gather_filesystem插件启用。
- -B --extra-node-info=<sockets[:cores[:threads]]>:选择满足<sockets[:cores[:threads]]>的节点，*表示对应选项不做限制。对应限制可以采用下面对应选项：
  - --sockets-per-node=<sockets>
  - --cores-per-socket=<cores>
  - --threads-per-core=<threads>
- --bcast[=<dest_path>]:*srun特有*,复制可执行程序到分配的计算节点的[<dest_path>]目录。如指定了<dest_path>，则复制可执行程序到此；如没指定则复制到当前工作目录下的"slurm_bcast_<job_id>.<step_id>"。如*srun --bcast=/tmp/mine -N3 a.out*将从当前目录复制a.out到每个分配的节点的/tmp/min并执行。
- --begin=<time>：设定开始分配资源运行的时间。时间格式可为HH:MM:SS，或添加AM、PM等，也可采用MMDDYY、MM/DD/YY或YYYY-MM-DD格式指定日期，含有日期及时间的格式为：YYYY-MM-DD[THH:MM[:SS]]，也可以采用类似now+时间单位的方式，时间单位可以为seconds（默认）、minutes、hours、days和weeks、today、tomorrow等，例如：
  - --begin=16:00：16:00开始。
  - --begin=now+1hour：1小时后开始。
  - --begin=now+60：60秒后开始（默认单位为秒）。
  - --begin=2017-02-20T12:34:00：2017-02-20T12:34:00开始。

- --bell：分配资源时终端响铃，参见--no-bell。
- --cpu-bind=[quiet,verbose,]type：*srun特有*，设定CPU绑定模式。
- --comment=<string>：作业说明。
- --contiguous：需分配到连续节点，一般来说连续节点之间网络会快一点，如在同一个IB交换机内，但有可能导致开始运行时间推迟（需等待足够多的连续节点）。
- --cores-per-socket=<cores>：分配的节点需要每颗CPU至少<cores>CPU核。
- --cpus-per-gpu=<ncpus>：每颗GPU需<ncpus>个CPU核，与--cpus-per-task不兼容。
- -c, --cpus-per-task=<ncpus>：每个进程需<ncpus>颗CPU核，一般运行OpenMP等多线程程序时需，普通MPI程序不需。
- --deadline=<OPT>：如果在此deadline（start > (deadline - time[-min]）之前没有结束，那么移除此作业。默认没有deadline，有效的时间格式为：
  - HH:MM[:SS] [AM|PM]
  - MMDD[YY]或MM/DD[/YY]或MM.DD[.YY]
  - MM/DD[/YY]-HH:MM[:SS]
  - YYYY-MM-DD[THH:MM[:SS]]]
- -d, --dependency=<dependency_list>：满足依赖条件<dependency_list>后开始分配。<dependency_list>可以为<type:job_id[:job_id][,type:job_id[:job_id]]>或<type:job_id[:job_id][?type:job_id[:job_id]]>。依赖条件如果用,分隔，则各依赖条件都需要满足；如果采用?分隔，那么只要任意条件满足即可。可以为：
  - after:job_id[:jobid...]：当指定作业号的作业结束后开始运行。
  - afterany:job_id[:jobid...]：当指定作业号的任意作业结束后开始运行。
  - aftercorr:job_id[:jobid...]：当相应任务号任务结束后，此作业组中的开始运行。
  - afternotok:job_id[:jobid...]：当指定作业号的作业结束时具有异常状态（非零退出码、节点失效、超时等）时。
  - afterok:job_id[:jobid...]：当指定的作业正常结束（退出码为0）时开始运行。
  - expand:job_id：分配给此作业的资源将扩展给指定作业。
  - singleton：等任意通账户的相同作业名的前置作业结束时。
- -D, --chdir=<path>：在切换到<path>工作目录后执行命令。

- -e, --error=<mode>：设定标准错误如何重定向。非交互模式下，默认srun重定向标准错误到与标准输出同样的文件（如指定）。此参数可以指定重定向到不同文件。如果指定的文件已经存在，那么将被覆盖。参见IO重定向。*salloc*无此选项。
- --epilog=<executable>：*srun特有*，作业结束后执行<executable>程序做相应处理。
- -E, --preserve-env：将环境变量*SLURM_NNODES*和*SLURM_NTASKS*传递给可执行文件，而无需通过计算命令行参数。
- --exclusive[=user|mcs]：排他性运行，独占性运行，此节点不允许其他[user]用户或mcs选项的作业共享运行作业。
- --export=<[ALL,]environment variables|ALL|NONE>：*sbatch与srun特有*，将环境变量传递给应用程序
  - ALL：复制所有提交节点的环境变量，为默认选项。
  - NONE：所有环境变量都不被传递，可执行程序必须采用绝对路径。一般用于当提交时使用的集群与运行集群不同时。
  - [ALL,]environment variables：复制全部环境变量及特定的环境变量及其值，可以有多个以,分隔的变量。如：“--export=EDITOR,ARG1=test”。
- --export-file=<filename | fd>：*sbatch特有*，将特定文件中的变量设置传递到计算节点，这允许在定义环境变量时有特殊字符。
- -F, --nodefile=<node file>：类似--nodelist指定需要运行的节点，但在一个文件中含有节点列表。
- -G, --gpus=[<type>:]<number>：设定使用的GPU类型及数目，如--gpus=v100:2。
- --gpus-per-node=[<type>:]<number>：设定单个节点使用的GPU类型及数目。
- --gpus-per-socket=[<type>:]<number>：设定每个socket需要的GPU类型及数目。
- --gpus-per-task=[<type>:]<number>：设定每个任务需要的GPU类型及数目。
- --gres=<list>:设定通用消费资源,可以以,分隔。每个<list>格式为“name[[:type]:count]”。name是可消费资源；count是资源个数，默认为1；
- -H, --hold：设定作业将被提交为挂起状态。挂起的作业可以利用scontrol release <job_id>使其排队运行。
- -h, --help：显示帮助信息。
- --hint=<type>：绑定任务到应用提示：

  - compute_bound：选择设定计算边界应用：采用每个socket的所有CPU核，每颗CPU核一个进程。
  - memory_bound：选择设定内存边界应用：仅采用每个socket的1颗CPU核，每颗CPU核一个进程。

  no multithread：在in-core multi-threading是否采用额外的线程，对通信密集型应用有益。仅当task/affinity插件启用时。

  - help：显示帮助信息

- -I, --immediate[=<seconds>]：*salloc与srun特有*，在<seconds>秒内资源未满足的话立即退出。格式可以为"-I60"，但不能之间有空格是"-I 60"。
- --ignore-pbs：*sbatch特有*，忽略批处理脚本中的"#PBS"选项。
- -i, --input=<mode>：*sbatch与srun特有*，指定标准输入如何重定向。默认，srun对所有任务重定向标准输入为从终端。参见IO重定向。
- -J, --job-name=<jobname>：设定作业名<jobname>，默认为命令名。
- --jobid=<jobid>：*srun特有*，初始作业步到某个已分配的作业号<jobid>下的作业下，类似设置了*SLURM_JOB_ID*环境变量。仅对作业步申请有效。
- -K, --kill-command[=signal]：*salloc特有*，设定需要终止时的signal，默认，如没指定，则对于交互式作业为SIGHUP，对于非交互式作业为SIGTERM。格式类似可以为"-K1"，但不能包含空格为"-K 1"。
- -K, –kill-on-bad-exit[=0|1]：*srun特有*，设定是否任何一个任务退出码为非0时，是否终止作业步。
- -k, --no-kill：如果分配的节点失效，那么不会自动终止。
- -L, --licenses=<license>：设定使用的<license>。
- -l, --label：*srun特有*，在标注正常输出或标准错误输出的行前面添加作业号。
- --mem=<size[units]>：设定每个节点的内存大小，后缀可以为[K|M|G|T]，默认为MB。
- --mem-per-cpu=<size[units]>：设定分配的每颗CPU对应最小内存，后缀可以为[K|M|G|T]，默认为MB。
- --mem-per-gpu=<size[units]>：设定分配的每颗GPU对应最小内存，后缀可以为[K|M|G|T]，默认为MB。
- --mincpus=<n>：设定每个节点最小的逻辑CPU核/处理器。

- --mpi=<mpi_type>：*srun特有*，指定使用的MPI环境，<mpi_type>可以主要为：
  - list：列出可用的MPI以便选择。
  - pmi2：启用PMI2支持
  - pmix：启用PMIx支持
  - none：默认选项，多种其它MPI实现有效。
- --multi-prog：*srun特有*，让不同任务运行不同的程序及参数，需指定一个配置文件，参见MULTIPLE PROGRAM CONFIGURATION。
- -N, --nodes=<minnodes[-maxnodes]>:采用特定节点数运行作业,如没指定maxnodes则需特定节点数，注意，这里是节点数，不是CPU核数，实际分配的是节点数×每节点CPU核数。
- --nice[=adjustment]：设定NICE调整值。负值提高优先级，正值降低优先级。调整范围为：+/- 2147483645。
- -n, --ntasks=<number>：设定所需要的任务总数。默认是每个节点1个任务，注意是节点，不是CPU核。仅对作业起作用，不对作业步起作用。--cpus-per-task选项可以改变此默认选项。
- --ntasks-per-core=<ntasks>:每颗CPU核运行<ntasks>个任务,需与-n, --ntasks=<number>配合，并自动绑定<ntasks>个任务到每个CPU核。仅对作业起作用，不对作业步起作用。
- --ntasks-per-node=<ntasks>:每个节点运行<ntasks>个任务,需与-n, --ntasks=<number>配合。仅对作业起作用，不对作业步起作用。
- --ntasks-per-socket=<ntasks>:每颗CPU运行<ntasks>个任务,需与-n, --ntasks=<number>配合，并绑定<ntasks>个任务到每颗CPU。仅对作业起作用，不对作业步起作用。
- --no-bell：*salloc特有*，资源分配时不终端响铃。参见--bell。
- --no-shell：*salloc特有*，分配资源后立即退出，而不运行命令。但Slurm作业仍旧被生成，在其激活期间，且保留这些激活的资源。用户会获得一个没有附带进程和任务的作业号，用户可以采用提交srun命令到这些资源。
- -o, --output=<mode>：*sbatch与srun特有*，指定标准输出重定向。在非交互模式中，默认srun收集各任务的标准输出，并发送到吸附的终端上。采用--output可以将其重定向到同一个文件、每个任务一个文件或/dev/null等。参见IO重定向。
- --open-mode=<append|truncate>：*sbtach与srun特有*，对标准输出和标准错误输出采用追加模式还是覆盖模式。

- -O, --overcommit：采用此选项可以使得每颗CPU运行不止一个任务。
- --open-mode=<append|truncate>：标准输出和标准错误输出打开文件的方式：
  - append：追加。
  - truncate：截断覆盖。
- -p, --partition=<partition_names>：使用<partition_names>队列
- --prolog=<executable>：*srun特有*，作业开始运行前执行<executable>程序，做相应处理。
- -Q, --quiet：采用安静模式运行，一般信息将不显示，但错误信息仍将被显示。
- --qos=<qos>：需要特定的服务质量(QS)。
- --quit-on-interrupt：*srun特有*，当SIGINT (Ctrl-C)时立即退出。
- -r, --relative=<n>：*srun特有*，在当前分配的第n节点上运行作业步。该选项可用于分配一些作业步到当前作业占用的节点外的节点，节点号从0开始。-r选项不能与-w或-x同时使用。仅对作业步有效。
- --reservation=<name>：从<name>预留资源分配。
- –requeue：*sbtach特有*，当非配的节点失效或被更高级作业抢占资源后，重新运行该作业。相当于重新运行批处理脚本，小心已运行的结果被覆盖等。
- --no-requeue：任何情况下都不重新运行。
- -S, --core-spec=<num>：指定预留的不被作业使用的各节点CPU核数。但也会被记入费用。
- --signal=<sig_num>[@<sig_time>]：设定到其终止时间前信号时间<sig_time>秒时的信号。由于Slurm事件处理的时间精度，信号有可能比设定时间早60秒。信号可以为10或USER1，信号时间sig_time必须在0到65535之间，如没指定，则默认为60秒。
- --sockets-per-node=<sockets>：设定每个节点的CPU颗数。
- -T, --threads=<nthreads>：*srun特有*，限制从srun进程发送到分配节点上的并发线程数。
- -t, --time=<time>:作业最大运行总时间<time>,到时间后将被终止掉。时间<time>的格式可以为：分钟、分钟:秒、小时:分钟:秒、天-小时、天-小时:分钟、天-小时:分钟:秒

- --task-epilog=<executable>：*srun特有*，任务终止后立即执行<executable>，对应于作业步分配。
- --task-prolog=<executable>：*srun特有*，任务开始前立即执行<executable>，对应于作业步分配。
- --test-only：*sbatch与srun特有*，测试批处理脚本，并预计将被执行的时间，但并不实际执行脚本。
- --thread-spec=<num>：设定指定预留的不被作业使用的各节点线程数。
- --threads-per-core=<threads>：每颗CPU核运行<threads>个线程。
- --time-min=<time>：设定作业分配的最小时间，设定后作业的运行时间将使得--time设定的时间不少于--time-min设定的。时间格式为：minutes、minutes:seconds、hours:minutes:seconds、days-hours、days-hours:minutes和days-hours:minutes:seconds。
- --usage：显示简略帮助信息
- --tmp=<size[units]>：设定/tmp目录最小磁盘空间，后缀可以为[K|M|G|T]，默认为MB。
- -u, --usage：显示简要帮助信息。
- -u, –unbuffered：*srun特有*，该选项使得输出可以不被缓存立即显示出来。默认应用的标准输出被glibc缓存，除非被刷新(flush)或输出被设定为步缓存。
- --use-min-nodes：设定如果给了一个节点数范围，分配时，选择较小的数。
- -V, --version：显示版本信息。
- -v, --verbose：显示详细信息，多个v会显示更详细的详细。
- -W, --wait=<seconds>：设定在第一个任务结束后多久结束全部任务。
- -w, --nodelist=<host1,host2,... or filename>：在特定<host1,host2>节点或filename文件中指定的节点上运行。
- --wait-all-nodes=<value>：*salloc与sbatch特有*，控制当节点准备好时何时运行命令。默认，当分配的资源准备好后*salloc*命令立即返回。<value>可以为：
  - 0：当分配的资源可以分配时立即执行，比如有节点以重启好。
  - 1：只有当分配的所有节点都准备好时才执行
- -X, --disable-status：*srun特有*，禁止在srun收到SIGINT (Ctrl-C)时显示任务状态。
- -x, --exclude=<host1,host2,... or filename>：在特定<host1,host2>节点或filename文件中指定的节点之外的节点上运行。

31.2 IO重定向

默认标准输出文件和标准出错文件将从所有任务中被重定向到*sbatch和srun（salloc不存在IO重定向）*的标准输出文件和标准出错文件，标准输入文件从srun的标准输入文件重定向到所有任务。如果标准输入仅仅是几个任务需要，建议采用读文件方式而不是重定向方式，以免输入错误数据。

以上行为可以通过 --output、--error和--input(-o、-e、-i)等选项改变，有效的格式为：

- all：标准输出和标准出错从所有任务定向到srun，标准输入文件从srun的标准输入文件重定向到所有任务（默认）。
- none：标准输出和标准出错不从任何任务定向到srun，标准输入文件不从srun定向到任何任务。
- taskid：标准输出和/或标准出错仅从任务号为taskid的任务定向到srun，标准输入文件仅从srun定向到任务号为taskid任务。
- filename: srun将所有任务的标准输出和标准出错重定向到filename文件，标准输入文件将从filename文件重定向到全部任务。
- 格式化字符：srun允许生成采用格式化字符命名的上述IO文件，如可以结合作业号、作业步、节点或任务等。
  - \\：不处理任何代替符。
  - %%：字符“%”。
  - %A：作业组的主作业分配号。
  - %a：作业组ID号。
  - %J：运行作业的作业号.步号（如128.0）。
  - %j：运行作业的作业号
  - %s：运行作业的作业步号。
  - %N：短格式节点名，每个节点将生成的不同的IO文件。
  - %n：当前作业相关的节点标记（如“0”是运行作业的第一个节点），每个节点将生成的不同的IO文件。
  - %t：与当前作业相关的任务标记(rank)，每个rank将生成一个不同的IO文件。
  - %u：用户名。

  在%与格式化标记符之间的数字可以用于生成前导零，如：

  - job%J.out：job128.0.out。
  - job%4j.out：job0128.out。
  - job%j-%2t.out：job128-00.out、job128-01.out、...。

32 交互式提交并行作业：srun

*srun*可以交互式提交运行并行作业，提交后，作业等待运行，等运行完毕后，才返回终端。语法为：*srun [OPTIONS...] executable [args...]*

32.1 主要输入环境变量

一些提交选项可通过环境变量来设置，命令行的选项优先级高于设置的环境变量，将覆盖掉环境变量的设置。环境变量与对应的参数如下：

- *SLURM_ACCOUNT*：类似-A, --account。
- *SLURM_ACCTG_FREQ*：类似--acctg-freq。
- *SLURM_BCAST*：类似--bcast。
- *SLURM_COMPRESS*：类似--compress。
- *SLURM_CORE_SPEC*：类似--core-spec。
- *SLURM_CPU_BIND*：类似--cpu-bind。
- *SLURM_CPUS_PER_GPU*：类似-c, --cpus-per-gpu。
- *SLURM_CPUS_PER_TASK*：类似-c, --cpus-per-task。
- *SLURM_DEBUG*：类似-v, --verbose。
- *SLURM_DEPENDENCY*：类似-P, --dependency=<jobid>。
- *SLURM_DISABLE_STATUS*：类似-X, --disable-status。
- *SLURM_DIST_PLANESIZE*：类似-m plane。
- *SLURM_DISTRIBUTION*：类似-m, --distribution。
- *SLURM_EPILOG*：类似--epilog。
- *SLURM_EXCLUSIVE*：类似--exclusive。
- *SLURM_EXIT_ERROR*：Slurm出错时的退出码。
- *SLURM_EXIT_IMMEDIATE*：当--immediate使用时且资源当前无效时的Slurm退出码。
- *SLURM_GEOMETRY*：类似-g, --geometry。

- *SLURM_GPUS*：类似-G, --gpus。
- *SLURM_GPU_BIND*：类似--gpu-bind。
- *SLURM_GPU_FREQ*：类似--gpu-freq。
- *SLURM_GPUS_PER_NODE*：类似--gpus-per-node。
- *SLURM_GPUS_PER_TASK*：类似--gpus-per-task。
- *SLURM_GRES*：类似--gres，参见*SLURM_STEP_GRES*。
- *SLURM_HINT*：类似--hint。
- *SLURM_IMMEDIATE*：类似-I, --immediate。
- *SLURM_JOB_ID*：类似--jobid。
- *SLURM_JOB_NAME*：类似-J, --job-name。
- *SLURM_JOB_NODELIST*：类似-w, –nodelist=<host1,host2,... or filename>，格式类似node[1-10,11,13-28]。
- *SLURM_JOB_NUM_NODES*：分配的总节点数。
- *SLURM_KILL_BAD_EXIT*：类似-K, --kill-on-bad-exit。
- *SLURM_LABELIO*：类似-l, --label。
- *SLURM_LINUX_IMAGE*：类似--linux-image。
- *SLURM_MEM_BIND*：类似--mem-bind。
- *SLURM_MEM_PER_CPU*：类似--mem-per-cpu。
- *SLURM_MEM_PER_NODE*：类似--mem。
- *SLURM_MPI_TYPE*：类似--mpi。
- *SLURM_NETWORK*：类似--network。
- *SLURM_NNODES*：类似-N, --nodes，即将废弃。
- *SLURM_NO_KILL*：类似-k, --no-kill。
- *SLURM_NTASKS*：类似-n, --ntasks。
- *SLURM_NTASKS_PER_CORE*：类似--ntasks-per-core。

- *SLURM_NTASKS_PER_SOCKET*：类似--ntasks-per-socket。
- *SLURM_NTASKS_PER_NODE*：类似--ntasks-per-node。
- *SLURM_OPEN_MODE*：类似--open-mode。
- *SLURM_OVERCOMMIT*：类似-O, --overcommit。
- *SLURM_PARTITION*：类似-p, --partition。
- *SLURM_PROFILE*：类似--profile。
- *SLURM_PROLOG*：类似--prolog，*仅限srun*。
- *SLURM_QOS*：类似--qos。
- *SLURM_REMOTE_CWD*：类似-D, --chdir=。
- *SLURM_RESERVATION*：类似--reservation。
- *SLURM_RESV_PORTS*：类似--resv-ports。
- *SLURM_SIGNAL*：类似--signal。
- *SLURM_STDERRMODE*：类似-e, --error。
- *SLURM_STDINMODE*：类似-i, --input。
- *SLURM_SRUN_REDUCE_TASK_EXIT_MSG*：如被设置，并且非0,那么具有相同退出码的连续的任务退出消息只显示一次。
- *SLURM_STEP_GRES*：类似--gres（仅对作业步有效，不影响作业分配），参见*SLURM_GRES*。
- *SLURM_STEP_KILLED_MSG_NODE_ID=ID*：如被设置，当作业或作业步被信号终止时只特定ID的节点下显示信息。
- *SLURM_STDOUTMODE*：类似-o, --output。
- *SLURM_TASK_EPILOG*：类似--task-epilog。
- *SLURM_TASK_PROLOG*：类似--task-prolog。
- *SLURM_TEST_EXEC*：如被定义，在计算节点执行之前先在本地节点上测试可执行程序。
- *SLURM_THREAD_SPEC*：类似--thread-spec。

- *SLURM_THREADS*：类似-T, --threads。
- *SLURM_TIMELIMIT*：类似-t, --time。
- *SLURM_UNBUFFEREDIO*：类似-u, --unbuffered。
- *SLURM_USE_MIN_NODES*：类似--use-min-nodes。
- *SLURM_WAIT*：类似-W, --wait。
- *SLURM_WORKING_DIR*：类似-D, --chdir。
- *SRUN_EXPORT_ENV*：类似--export，将覆盖掉*SLURM_EXPORT_ENV*。

32.2 主要输出环境变量

*srun*会在执行的节点上设置如下环境变量：

- *SLURM_CLUSTER_NAME*：集群名。
- *SLURM_CPU_BIND_VERBOSE*：--cpu-bind详细情况(quiet、verbose)。
- *SLURM_CPU_BIND_TYPE*：--cpu-bind类型(none、rank、map-cpu:、mask-cpu:)。
- *SLURM_CPU_BIND_LIST*：--cpu-bind映射或掩码列表。
- *SLURM_CPU_FREQ_REQ*：需要的CPU频率资源，参见--cpu-freq和输入环境变量*SLURM_CPU_FREQ_REQ*。
- *SLURM_CPUS_ON_NODE*：节点上的CPU颗数。
- *SLURM_CPUS_PER_GPU*：每颗GPU对应的CPU颗数，参见--cpus-per-gpu选项指定。
- *SLURM_CPUS_PER_TASK*：每作业的CPU颗数，参见--cpus-per-task选项指定。
- *SLURM_DISTRIBUTION*：分配的作业的分布类型，参见-m, --distribution。
- *SLURM_GPUS*：需要的GPU颗数，仅提交时有-G, --gpus时。
- *SLURM_GPU_BIND*：指定绑定任务到GPU，仅提交时具有--gpu-bind参数时。
- *SLURM_GPU_FREQ*：需求的GPU频率，仅提交时具有--gpu-freq参数时。
- *SLURM_GPUS_PER_NODE*：需要的每个节点的GPU颗数，仅提交时具有--gpus-per-node参数时。

- *SLURM_GPUS_PER_SOCKET*：需要的每个socket的GPU颗数，仅提交时具有--gpus-per-socket参数时。
- *SLURM_GPUS_PER_TASK*：需要的每个任务的GPU颗数，仅提交时具有--gpus-per-task参数时。
- *SLURM_GTIDS*：此节点上分布的全局任务号，从0开始，以,分隔。
- *SLURM_JOB_ACCOUNT*：作业的记账名。
- *SLURM_JOB_CPUS_PER_NODE*：每个节点的CPU颗数，格式类似40(x3),3，顺序对应*SLURM_JOB_NODELIST*节点名顺序。
- *SLURM_JOB_DEPENDENCY*：依赖关系，参见--dependency选项。
- *SLURM_JOB_ID*：作业号。
- *SLURM_JOB_NAME*：作业名，参见--job-name选项或srun启动的命令名。
- *SLURM_JOB_PARTITION*：作业使用的队列名。
- *SLURM_JOB_QOS*：作业的服务质量QOS。
- *SLURM_JOB_RESERVATION*：作业的高级资源预留。
- *SLURM_LAUNCH_NODE_IPADDR*：任务初始启动节点的IP地址。
- *SLURM_LOCALID*：节点本地任务号。
- *SLURM_MEM_BIND_LIST*：--mem-bind映射或掩码列表（<list of IDs or masks for this node>）。
- *SLURM_MEM_BIND_PREFER*：--mem-bin prefer优先权。
- *SLURM_MEM_BIND_TYPE*:--mem-bind类型(none、rank、map-mem:、mask-mem:)。
- *SLURM_MEM_BIND_VERBOSE*：内存绑定详细情况，参见--mem-bind verbosity（quiet、verbose）。
- *SLURM_MEM_PER_GPU*：每颗GPU需求的内存，参见--mem-per-gpu。
- *SLURM_NODE_ALIASES*：分配的节点名、通信IP地址和节点名，每组内采用:分隔，组间通过,分隔，如：*SLURM_NODE_ALIASES*=0:1.2.3.4:foo,ec1:1.2.3.5:bar。
- *SLURM_NODEID*：当前节点的相对节点号。
- *SLURM_NODELIST*：分配的节点列表，格式类似node[1-10,11,13-28]。

- *SLURM_NTASKS*：任务总数。
- *SLURM_PRIO_PROCESS*：作业提交时的调度优先级值（nice值）。
- *SLURM_PROCID*：当前MPI秩号。
- *SLURM_SRUN_COMM_HOST*：节点的通信IP。
- *SLURM_SRUN_COMM_PORT*：srun的通信端口。
- *SLURM_STEP_LAUNCHER_PORT*：作业步启动端口。
- *SLURM_STEP_NODELIST*：作业步节点列表，格式类似node[1-10,11,13-28]。
- *SLURM_STEP_NUM_NODES*：作业步的节点总数。
- *SLURM_STEP_NUM_TASKS*：作业步的任务总数。
- *SLURM_STEP_TASKS_PER_NODE*：作业步在每个节点上的任务总数，格式类似40(x3),3，顺序对应*SLURM_JOB_NODELIST*节点名顺序。
- *SLURM_STEP_ID*：当前作业的作业步号。
- *SLURM_SUBMIT_DIR*：提交作业的目录，或有可能由-D, –chdir参数指定。
- *SLURM_SUBMIT_HOST*：提交作业的节点名。
- *SLURM_TASK_PID*：任务启动的进程号。
- *SLURM_TASKS_PER_NODE*：每个节点上启动的任务数，以*SLURM_NODELIST*中的节点顺序显示，以,分隔。如果两个或多个连续节点上的任务数相同，数后跟着(x#)，其中#是对应的节点数，如*SLURM_TASKS_PER_NODE*=2(x3),1”表示，前三个节点上的作业数为3，第四个节点上的任务数为1。
- *SLURM_UMASK*：作业提交时的umask掩码。
- *SLURMD_NODENAME*：任务运行的节点名。
- *SRUN_DEBUG*：srun命令的调试详细信息级别，默认为3（info级）。

32.3 多程序运行配置

Slurm支持一次申请多个节点，在不同节点上同时启动执行不同任务。为实现次功能，需要生成一个配置文件，在配置文件中做相应设置。

配置文件中的注释必需第一列为#，配置文件包含以空格分隔的以下域（字段）：

- 任务范围(Task rank)：一个或多个任务秩，多个值的话可以用逗号,分隔。范围可以用两个用-分隔的整数表示，小数在前，大数在后。如果最后一行为*，则表示全部其余未在前面声明的秩。如没有指明可执行程序，则会显示错误信息："No executable program specified for this task"。
- 需要执行的可执行程序(Executable)：也许需要绝对路径指明。
- 可执行程序的参数(Arguments)："%t"将被替换为任务号；"%o"将被替换为任务号偏移（如配置的秩为"1-5"，则偏移值为"0-4"）。单引号可以防止内部的字符被解释。此域为可选项，任何在命令行中需要添加的程序参数都将加在配置文件中的此部分。

例如，配置文件silly.conf内容为：

```
###################################################################
# srun multiple program configuration file
#
# srun -n8 -l --multi-prog silly.conf
###################################################################
4-6 hostname
1,7 echo task:%t
0,2-3 echo offset:%o
```

运行：*srun -n8 -l --multi-prog silly.conf*

输出结果：

```
0: offset:0
1: task:1
2: offset:1
3: offset:2
4: node1
5: node2
6: node4
7: task:7
```

32.4 常见例子

- 使用8个CPU核(-n8)运行作业，并在标准输出上显示任务号(-l)：

  srun -n8 -l hostname

  输出结果：

```
0: node0
1: node0
2: node1
3: node1
4: node2
5: node2
6: node3
7: node3
```

- 在脚本中使用-r2参数使其在第2号（分配的节点号从0开始）开始的两个节点上运行，并采用实时分配模式而不是批处理模式运行：

  脚本*test.sh*内容：

```
#!/bin/sh
echo $SLURM_NODELIST
srun -lN2 -r2 hostname
srun -lN2 hostname
```

  运行：*salloc -N4 test.sh*

  输出结果：

```
dev[7-10]
0: node9
1: node10
0: node7
1: node8
```

- 在分配的节点上并行运行两个作业步：

  脚本*test.sh*内容：

```
#!/bin/bash
srun -lN2 -n4 -r 2 sleep 60 &
srun -lN2 -r 0 sleep 60 &
sleep 1
squeue
squeue -s
```

  运行：*salloc -N4 test.sh*

  输出结果：

```
  JOBID PARTITION   NAME    USER  ST     TIME  NODES NODELIST
  65641     batch test.sh  grondo  R     0:01      4 dev[7-10]

STEPID     PARTITION    USER     TIME NODELIST
65641.0        batch  grondo     0:01 dev[7-8]
65641.1        batch  grondo     0:01 dev[9-10]
```

- 运行MPICH作业：

  脚本*test.sh*内容：

```
#!/bin/sh
MACHINEFILE="nodes.$SLURM_JOB_ID"

# 生成MPICH所需的包含节点名的machinfile文件
srun -l /bin/hostname | sort -n | awk '{print $2}' > $MACHINEFILE

# 运行MPICH作业
mpirun -np $SLURM_NTASKS -machinefile $MACHINEFILE mpi-app

rm $MACHINEFILE
```

  采用2个节点（-N2）共4个CPU核（-n4）运行：*salloc -N2 -n4 test.sh*

- 利用不同节点号（*SLURM_NODEID*）运行不同作业，节点号从0开始：

  脚本*test.sh*内容：

```
case $SLURM_NODEID in
    0) echo "I am running on "
       hostname ;;
    1) hostname
       echo "is where I am running" ;;
esac
```

  运行：*srun -N2 test.sh*

  输出：

```
dev0
is where I am running
I am running on
ev1
```

- 利用多核选项控制任务执行：

  采用2个节点（-N2），每节点4颗CPU每颗CPU 2颗CPU核（-B 4-4:2-2），运行作业：

  srun -N2 -B 4-4:2-2 a.out

- 运行GPU作业：脚本*gpu.script*内容：

```
#!/bin/bash
srun -n1 -p GPU-V100 --gres=gpu:v100:2 prog1 &
```

  -p GPU-V100 指定采用GPU队列GPU-V100，--gres=gpu:v100:2指明每个节点使用2块NVIDIA V100 GPU卡。

- 排它性独占运行作业：

  脚本*my.script*内容：

```
#!/bin/bash
srun --exclusive -n4 prog1 &
srun --exclusive -n3 prog2 &
srun --exclusive -n1 prog3 &
srun --exclusive -n1 prog4 &
wait
```

33　批处理方式提交作业：sbatch

Slurm支持利用*sbatch*命令采用批处理方式运行作业，*sbatch*命令在脚本正确传递给作业调度系统后立即退出，同时获取到一个作业号。作业等所需资源满足后开始运行。

*sbatch*提交一个批处理作业脚本到Slurm。批处理脚本名可以在命令行上通过传递给*sbatch*，如没有指定文件名，则*sbatch*从标准输入中获取脚本内容。

脚本文件基本格式：

- 第一行以#!/bin/sh等指定该脚本的解释程序，/bin/sh可以变为/bin/bash、/bin/csh等。
- 在可执行命令之前的每行"#SBATCH"前缀后跟的参数作为作业调度系统参数。在任何非注释及空白之后的"#SBATCH"将不再作为Slurm参数处理。

默认，标准输出和标准出错都定向到同一个文件*slurm-%j.out*，"%j"将被作业号代替。

脚本*myscript*内容：

```
#!/bin/sh
#SBATCH --time=1
#SBATCH -p serial
srun hostname |sort
```

采用4个节点（-N4）运行：*sbatch -p batch -N4 myscript*

在这里，虽然脚本中利用“#SBATCH -p serial”指定了使用serial队列，但命令行中的*-p batch*优先级更高，因此实际提交到batch队列。

提交成功后有类似输出：

```
salloc: Granted job allocation 65537
```

其中65537为分配的作业号。

程序结束后的作业日志文件*slurm-65537.out*显示：

```
node1
node2
node3
node4
```

从标准输入获取脚本内容，可采用以下两种方式之一：

1. 运行*sbatch -N4*，显示等待后输入：

   ```
   #!/bin/sh
   srun hostname |sort
   ```

   输入以上内容后按CTRL+D终止输入。

2. 运行*sbatch -N4 <<EOF*，

   ```
   > #!/bin/sh
   > srun hostname |sort
   > EOF
   ```

   - 第一个EOF表示输入内容的开始标识符
   - 最后的EOF表示输入内容的终止标识符，在两个EOF之间的内容为实际执行的内容。
   - >实际上是每行输入回车后自动在下一行出现的提示符。

以上两种方式输入结束后将显示：

```
sbatch: Submitted batch job 65541
```

常见主要选项参见31.1。

33.1 主要输入环境变量

一些选项可通过环境变量来设置，命令行的选项优先级高于设置的环境变量，将覆盖掉环境变量的设置。环境变量与对应的参数如下：

- *SBATCH_ACCOUNT*：类似-A、--account。
- *SBATCH_ACCTG_FREQ*：类似--acctg-freq。
- *SBATCH_ARRAY_INX*：类似-a、--array。
- *SBATCH_BATCH*：类似--batch。
- *SBATCH_CLUSTERS* 或 *SLURM_CLUSTERS*：类似--clusters。
- *SBATCH_CONSTRAINT*：类似-C, --constraint。
- *SBATCH_CORE_SPEC*：类似--core-spec。
- *SBATCH_CPUS_PER_GPU*：类似--cpus-per-gpu。
- *SBATCH_DEBUG*：类似-v、--verbose。
- *SBATCH_DISTRIBUTION*：类似-m、--distribution。
- *SBATCH_EXCLUSIVE*：类似--exclusive。
- *SBATCH_EXPORT*：类似--export。
- *SBATCH_GEOMETRY*：类似-g、--geometry。
- *SBATCH_GET_USER_ENV*：类似--get-user-env。
- *SBATCH_GPUS*：类似 -G, --gpus。
- *SBATCH_GPU_BIND*：类似 --gpu-bind。
- *SBATCH_GPU_FREQ*：类似 --gpu-freq。
- *SBATCH_GPUS_PER_NODE*：类似 --gpus-per-node。
- *SBATCH_GPUS_PER_TASK*：类似 --gpus-per-task。

- *SBATCH_GRES*：类似 --gres。
- *SBATCH_GRES_FLAGS*：类似--gres-flags。
- *SBATCH_HINT* 或 *SLURM_HINT*：类似--hint。
- *SBATCH_IGNORE_PBS*：类似--ignore-pbs。
- *SBATCH_JOB_NAME*：类似-J、--job-name。
- *SBATCH_MEM_BIND*：类似--mem-bind。
- *SBATCH_MEM_PER_CPU*：类似--mem-per-cpu。
- *SBATCH_MEM_PER_GPU*：类似--mem-per-gpu。
- *SBATCH_MEM_PER_NODE*：类似--mem。
- *SBATCH_NETWORK*：类似--network。
- *SBATCH_NO_KILL*：类似-k, –no-kill。
- *SBATCH_NO_REQUEUE*：类似--no-requeue。
- *SBATCH_OPEN_MODE*：类似--open-mode。
- *SBATCH_OVERCOMMIT*：类似-O、--overcommit。
- *SBATCH_PARTITION*：类似-p、--partition。
- *SBATCH_PROFILE*：类似--profile。
- *SBATCH_QOS*：类似--qos。
- *SBATCH_RAMDISK_IMAGE*：类似--ramdisk-image。
- *SBATCH_RESERVATION*：类似--reservation。
- *SBATCH_REQUEUE*：类似--requeue。
- *SBATCH_SIGNAL*：类似--signal。
- *SBATCH_THREAD_SPEC*：类似--thread-spec。
- *SBATCH_TIMELIMIT*：类似-t、--time。
- *SBATCH_USE_MIN_NODES*：类似--use-min-nodes。
- *SBATCH_WAIT*：类似-W、--wait。

- *SBATCH_WAIT_ALL_NODES*：类似--wait-all-nodes。
- *SBATCH_WAIT4SWITCH*：需要交换的最大时间，参见See--switches。
- *SLURM_EXIT_ERROR*：设定Slurm出错时的退出码。
- *SLURM_STEP_KILLED_MSG_NODE_ID=ID*：如果设置，当作业或作业步被信号终止时，只有指定ID的节点记录。

33.2 主要输出环境变量

Slurm将在作业脚本中输出以下变量，作业脚本可以使用这些变量：

- *SBATCH_MEM_BIND*：--mem-bind选项设定。
- *SBATCH_MEM_BIND_VERBOSE*：如--mem-bind选项包含verbose选项时，则由其设定。
- *SBATCH_MEM_BIND_LIST*：内存绑定时设定的bit掩码。
- *SBATCH_MEM_BIND_PREFER*：--mem-bin prefer优先权。
- *SBATCH_MEM_BIND_TYPE*：由--mem-bind选项设定。
- *SLURM_ARRAY_TASK_ID*：作业组ID（索引）号。
- *SLURM_ARRAY_TASK_MAX*：作业组最大ID号。
- *SLURM_ARRAY_TASK_MIN*：作业组最小ID号。
- *SLURM_ARRAY_TASK_STEP*：作业组索引步进间隔。
- *SLURM_ARRAY_JOB_ID*：作业组主作业号。
- *SLURM_CLUSTER_NAME*：集群名。
- *SLURM_CPUS_ON_NODE*：分配的节点上的CPU颗数。
- *SLURM_CPUS_PER_GPU*：每个任务的CPU颗数，只有--cpus-per-gpu选项设定时才有。
- *SLURM_CPUS_PER_TASK*：每个任务的CPU颗数，只有--cpus-per-task选项设定时才有。
- *SLURM_DISTRIBUTION*：类似-m, --distribution。
- *SLURM_EXPORT_ENV*：类似-e, --export。

- *SLURM_GPU_BIND*：指定绑定任务到GPU，仅提交时具有--gpu-bind参数时。
- *SLURM_GPU_FREQ*：需求的GPU频率，仅提交时具有--gpu-freq参数时。
- *SLURM_GPUS* Number of GPUs requested. Only set if the -G, –gpus option is specified.
- *SLURM_GPU_BIND*：需要的任务绑定到GPU，仅提交时有–gpu-bind参数时。
- *SLURM_GPUS_PER_NODE*：需要的每个节点的GPU颗数，仅提交时具有--gpus-per-node参数时。
- *SLURM_GPUS_PER_SOCKET*：需要的每个socket的GPU颗数，仅提交时具有--gpus-per-socket参数时。
- *SLURM_GPUS_PER_TASK*：需要的每个任务的GPU颗数，仅提交时具有--gpus-per-task参数时。
- *SLURM_GTIDS*：在此节点上运行的全局任务号。以0开始，逗号,分隔。
- *SLURM_JOB_ACCOUNT*：作业的记账账户名。
- *SLURM_JOB_ID*：作业号。
- *SLURM_JOB_CPUS_PER_NODE*：每个节点上的CPU颗数，格式类似40(x3),3，顺序对应*SLURM_JOB_NODELIST*节点名顺序。
- *SLURM_JOB_DEPENDENCY*：作业依赖信息，由--dependency选项设置。
- *SLURM_JOB_NAME*：作业名。
- *SLURM_JOB_NODELIST*：分配的节点名列表，格式类似node[1-10,11,13-28]。
- *SLURM_JOB_NUM_NODES*：分配的节点总数。
- *SLURM_JOB_PARTITION*：使用的队列名。
- *SLURM_JOB_QOS*：需要的服务质量(QOS)。
- *SLURM_JOB_RESERVATION*：作业预留。
- *SLURM_LOCALID*：节点本地任务号。
- *SLURM_MEM_PER_CPU*：类似--mem-per-cpu，每颗CPU需要的内存。
- *SLURM_MEM_PER_GPU*：类似--mem-per-gpu，每颗GPU需要的内存。
- *SLURM_MEM_PER_NODE*：类似--mem，每个节点的内存。

- *SLURM_NODE_ALIASES*：分配的节点名、通信IP地址和主机名组合，类似*SLURM_NODE_ALIASES*=ec0:1.2.3.4:foo,ec1:1.2.3.5:bar。
- *SLURM_NODEID*：分配的节点号。
- *SLURM_NTASKS*：类似-n, --ntasks，总任务数，CPU核数。
- *SLURM_NTASKS_PER_CORE*：每个CPU核分配的任务数。
- *SLURM_NTASKS_PER_NODE*：每个节点上的任务数。
- *SLURM_NTASKS_PER_SOCKET*：每颗CPU上的任务数，仅--ntasks-per-socket选项设定时设定。
- *SLURM_PRIO_PROCESS*：进程的调度优先级（nice值）。
- *SLURM_PROCID*：当前进程的MPI秩。
- *SLURM_PROFILE*：类似--profile。
- *SLURM_RESTART_COUNT*：因为系统失效等导致的重启次数。
- *SLURM_SUBMIT_DIR*：sbatch启动目录，即提交作业时目录，或提交时由-D, --chdir参数指定的。
- *SLURM_SUBMIT_HOST*：sbatch启动的节点名，即提交作业时节点。
- *SLURM_TASKS_PER_NODE*：每节点上的任务数，以*SLURM_NODELIST*中的节点顺序显示，以,分隔。如果两个或多个连续节点上的任务数相同，数后跟着(x#)，其中#是对应的节点数，如“*SLURM_TASKS_PER_NODE*=2(x3),1”表示前三个节点上的作业数为3，第四个节点上的任务数为1。
- *SLURM_TASK_PID*：任务的进程号PID。
- *SLURMD_NODENAME*：执行作业脚本的节点名。

33.3 串行作业提交

对于串行程序，用户可类似下面两者之一：

1. 直接采用*sbatch -n1 -o job-%j.log -e job-%j.err yourprog*方式运行
2. 编写命名为*serial_job.sh*（此脚本名可以按照用户喜好命名）的串行作业脚本，其内容如下：

```
#!/bin/sh
#An example for serial job.
#SBATCH -J job_name
#SBATCH -o job-%j.log
#SBATCH -e job-%j.err
echo Running on hosts
echo Time is `date`
echo Directory is $PWD
echo This job runs on the following nodes:
echo $SLURM_JOB_NODELIST
module load intel/2019.update5
echo This job has allocated 1 cpu core.
./yourprog
```

必要时需在脚本文件中利用*module*命令设置所需的环境,如上面的*module load intel/2019.update5*

作业脚本编写完成后，可以按照下面命令提交作业：

hmli@login01:~/work$ sbatch -n1 -p serail serial_job.sh

33.4 OpenMP共享内存并行作业提交

对于OpenMP共享内存并行程序，可编写命名为*omp_job.sh*的作业脚本，内容如下：

```
#!/bin/sh
#An example for serial job.
#SBATCH -J job_name
#SBATCH -o job-%j.log
#SBATCH -e job-%j.err
#SBATCH -N 1 -n 8
echo Running on hosts
echo Time is `date`
echo Directory is $PWD
echo This job runs on the following nodes:
echo $SLURM_JOB_NODELIST
module load intel/2016.3.210
echo This job has allocated 1 cpu core.
export OMP_NUM_THREADS=8
./yourprog
```

相对于串行作业脚本，主要增加export OMP_NUM_THREADS=8和#SBATCH -N 1 -n 8，表示使用一个节点内部的八个核，从而保证是在同一个节点内部，需几个核就设置-n为几。-n后跟的不能超过单节点内CPU核数。

作业脚本编写完成后，可以按照下面命令提交作业：

hmli@login01:~/work$ sbatch omp_job.sh

33.5 MPI并行作业提交

与串行作业类似，对于MPI并行作业，则需编写类似下面脚本*mpi_job.sh*：

```
#!/bin/sh
#An example for MPI job.
#SBATCH -J job_name
#SBATCH -o job-%j.log
#SBATCH -e job-%j.err
#SBATCH -N 1 -n 8
echo Time is `date`
echo Directory is $PWD
echo This job runs on the following nodes:
echo $SLURM_JOB_NODELIST
echo This job has allocated $SLURM_JOB_CPUS_PER_NODE cpu cores.
#module load intelmpi/5.1.3.210
#module load mpich/3.2/intel/2016.3.210
#module load openmpi/2.0.0/intel/2016.3.210
module load hpcx/hpcx-intel-2019.update5
MPIRUN=mpirun #Intel mpi and Open MPI
#MPIRUN=mpiexec #MPICH
#MPIOPT="-env I_MPI_FABRICS shm:ofa" #Intel MPI 2018
#MPIOPT="-env I_MPI_FABRICS shm:ofi" #Intel MPI 2019
#MPIOPT="--mca plm_rsh_agent ssh --mca btl self,openib,sm" #Open MPI
#MPIOPT="-iface ib0" #MPICH3
#目前新版的MPI可以自己找寻高速网络配置，可以设置MPIOPT为空，如去掉下行的#
MPIOPT=
$MPIRUN $MPIOPT ./yourprog
```

与串行程序的脚本相比，主要不同之处在于需采用mpirun或mpiexec的命令格式提交并行可执行程序。采用不同MPI提交时，需要打开上述对应的选项。

与串行作业类似，可使用下面方式提交：

hmli@login01:~/work$ sbatch mpi_job.sh

33.6 GPU作业提交

运行GPU作业，需要提交时利用--gres=gpu等指明需要的GPU资源并用-p指明采用等GPU队列。

脚本*gpu_job.sh*内容：

```
#!/bin/sh
#An example for gpu job.
#SBATCH -J job_name
#SBATCH -o job-%j.log
#SBATCH -e job-%j.err
#SBATCH -N 1 -n 8 -p GPU-V100 --gres=gpu:v100:2
./your-gpu-prog
```

上面-p GPU-V100 指定采用GPU队列GPU-V100，--gres=gpu:v100:2指明每个节点使用2块NVIDIA V100 GPU卡。

33.7 作业获取的节点名及对应CPU核数解析

作业调度系统主要负责分配节点及该节点分配的CPU核数等，在Slurm作业脚本中利用环境变量可以获取分配到的节点名(*SLURM_JOB_NODELIST*及对应核数(*SLURM_JOB_CPUS_PER_NODE*)或对应的任务数(*SLURM_TASKS_PER_NODE*)，然后根据自己程序原始的命令在Slurm脚本中进行修改就成。

*SLURM_JOB_NODELIST*及*SLURM_TASKS_PER_NODE*有一定的格式，在Slurm脚本中直接*scontrol show hostnames ${SLURM_JOB_NODELIST%[*}*可以获取每行一个节点名的节点名列表，或者参考下面解析脚本，可以在自己的Slurm脚本中参考获取自己的节点等信息。

```
#!/bin/bash
#Auther HM_Li<hmli@ustc.edu.cn>
#SLURM_JOB_NODELIST=cnode[001-003,005,440-441]
BASENAME=${SLURM_JOB_NODELIST%[*}
LIST=${SLURM_JOB_NODELIST#*[}
LIST=${LIST%]}
IDLIST=''
for i in `echo $LIST | tr ',' ' '`
do
    if [[ $i =~ '-' ]]; then
        IDLIST=$IDLIST' '`seq -w `echo $i| tr '-' ' '``
    else
```

```
        IDLIST=$IDLIST' '$i
    fi
done
NODELIST=''
for i in $IDLIST
do
    NODELIST=$NODELIST" $BASENAME"$i
done
echo -e "Node list: \n\t"$NODELIST

#上述解析或直接scontrol show hostnames ${SLURM_JOB_NODELIST%[*}

#SLURM_TASKS_PER_NODE='40(x3),23,20(x2)'
#SLURM_JOB_CPUS_PER_NODE='40(x3),23,20(x2)'
CORELIST=''
for i in `echo $SLURM_JOB_CUPS_PER_NODE| tr ',' ' '`
do
    if [[ $i =~ 'x' ]]; then
        read CORES NODES <<<`echo $i| tr '(x)' ' '`
        for n in `seq 1 $NODES`
        do
            CORELIST=$CORELIST' '$CORES
        done
    else
        CORELIST=$CORELIST' '$i
    fi
done
echo -e "\nCPU Core list: \n\t$CORELIST"

echo -e "\nNode list with corresponding CPU Cores: "
CARR=(${CORELIST})
i=0
for n in $NODELIST
do
    echo -e "\t"$n: ${CARR[$i]}
    i=$(($i+1))
done
```

上面只是例子，要加在自己的Slurm脚本中，而不是先提交上面这个脚本获取节点名后放到slurm脚本中，其原因在于除非提交时指定节点名，否则每次提交后获取的节

点名等是有可能变的。比如针对star-cmm+软件，原来的方式是：

/STATCMM+PATH/bin/starccm+ -rsh ssh -batch -power -on cnode400:40,cnode432:40 FireAndSmokeResampled.sim >residual.log

则可以改为下面脚本：

```
#!/bin/sh
#An example for MPI job.
#SBATCH -J job_name
#SBATCH -o job-%j.log
#SBATCH -e job-%j.err
echo This job runs on the following nodes:
echo $SLURM_JOB_NODELIST
echo This job has allocated $SLURM_JOB_CPUS_PER_NODE cpu cores.

BASENAME=${SLURM_JOB_NODELIST%[*}
LIST=${SLURM_JOB_NODELIST#*[}
LIST=${LIST%]}
IDLIST=''
for i in `echo $LIST | tr ',' ' '`
do
    if [[ $i =~ '-' ]]; then
        T=`echo $i| tr '-' ' '`
        IDLIST=$IDLIST' '`seq -w $T`
    else
        IDLIST=$IDLIST' '$i
    fi
done
NODEOPT=''
for i in $IDLIST
do
    NODEOPT=$NODEOPT,$BASENAME$i:40
done
NODEOPT=${NODEOPT:1} #trim the first ,

/STARCMM+PATH/bin/starccm+ -rsh ssh -batch -power -on $NODEOPT \
FireAndSmokeResampled.sim > residual.log
```

34 分配式提交作业：salloc

*salloc*将获取作业的分配后执行命令，当命令结束后释放分配的资源。其基本语法为：

salloc [options] [<command> [command args]]

command可以是任何是用户想要用的程序，典型的为xterm或包含*srun*命令的shell。如果后面没有跟命令,那么将执行Slurm系统slurm.conf配置文件中通过SallocDefaultCommand设定的命令。如果SallocDefaultCommand没有设定，那么将执行用户的默认shell。

注意：*salloc*逻辑上包括支持保存和存储终端行设置，并且设计为采用前台方式执行。如果需要后台执行*salloc*,可以设定标准输入为某个文件,如:*salloc -n16 a.out </dev/null &*。

34.1 作业获取的节点名及对应CPU核数解析

作业调度系统主要负责分配节点及该节点分配的CPU核数等，在Slurm作业脚本中利用环境变量可以获取分配到的节点名(*SLURM_JOB_NODELIST*及对应核数(*SLURM_JOB_CPUS_PER_NODE*)或对应的任务数(*SLURM_TASKS_PER_NODE*)，然后根据自己程序原始的命令在Slurm脚本中进行修改就成。

*SLURM_JOB_NODELIST*及*SLURM_TASKS_PER_NODE*有一定的格式，以下为一个参考解析脚本，可以在自己的Slurm脚本中参考获取自己的节点等信息。

```
#!/bin/bash
#Auther HM_Li<hmli@ustc.edu.cn>
#SLURM_JOB_NODELIST=cnode[001-003,005,440-441]
BASENAME=${SLURM_JOB_NODELIST%[*}
LIST=${SLURM_JOB_NODELIST#*[}
LIST=${LIST%]}
IDLIST=''
for i in `echo $LIST | tr ',' ' '`
do
    if [[ $i =~ '-' ]]; then
        IDLIST=$IDLIST' '`seq -w `echo $i| tr '-' ' '``
    else
        IDLIST=$IDLIST' '$i
    fi
done
NODELIST=''
for i in $IDLIST
do
```

```
    NODELIST=$NODELIST" $BASENAME"$i
done
echo -e "Node list: \n\t"$NODELIST

#SLURM_TASKS_PER_NODE='40(x3),23,20(x2)'
#SLURM_JOB_CPUS_PER_NODE='40(x3),23,20(x2)'
CORELIST=''
for i in `echo $SLURM_JOB_CUPS_PER_NODE| tr ',' ' '`
do
    if [[ $i =~ 'x' ]]; then
        read CORES NODES <<<`echo $i| tr '(x)' ' '`
        for n in `seq 1 $NODES`
        do
            CORELIST=$CORELIST' '$CORES
        done
    else
        CORELIST=$CORELIST' '$i
    fi
done
echo -e "\nCPU Core list: \n\t$CORELIST"

echo -e "\nNode list with corresponding CPU Cores: "
CARR=(${CORELIST})
i=0
for n in $NODELIST
do
    echo -e "\t"$n: ${CARR[$i]}
    i=$(($i+1))
done
```

上面只是例子，要加在自己的Slurm脚本中，而不是先提交上面这个脚本获取节点名后放到slurm脚本中，其原因在于除非提交时指定节点名，否则每次提交后获取的节点名等是有可能变的。比如针对star-cmm+软件，原来的方式是：

/STATCMM+PATH/bin/starccm+ -rsh ssh -batch -power -on cnode400:40,cnode432:40 FireAndSmokeResampled.sim >residual.log

则可以改为下面脚本：

```
#!/bin/sh
#An example for MPI job.
#SBATCH -J job_name
```

```
#SBATCH -o job-%j.log
#SBATCH -e job-%j.err
echo This job runs on the following nodes:
echo $SLURM_JOB_NODELIST
echo This job has allocated $SLURM_JOB_CPUS_PER_NODE cpu cores.

BASENAME=${SLURM_JOB_NODELIST%[*}
LIST=${SLURM_JOB_NODELIST#*[}
LIST=${LIST%]}
IDLIST=''
for i in `echo $LIST | tr ',' ' '`
do
    if [[ $i =~ '-' ]]; then
        T=`echo $i| tr '-' ' '`
        IDLIST=$IDLIST' '`seq -w $T`
    else
        IDLIST=$IDLIST' '$i
    fi
done
NODEOPT=''
for i in $IDLIST
do
    NODEOPT=$NODEOPT,$BASENAME$i:40
done
NODEOPT=${NODEOPT:1} #trim the first ,

/STARCMM+PATH/bin/starccm+ -rsh ssh -batch -power -on $NODEOPT \
FireAndSmokeResampled.sim > residual.log
```

34.2 主要选项

常见主要选项参见31.1。

34.3 主要输入环境变量

- *SALLOC_ACCOUNT*：类似-A, --account
- *SALLOC_ACCTG_FREQ*：类似--acctg-freq
- *SALLOC_BELL*：类似--bell

- *SALLOC_CLUSTERS*、*SLURM_CLUSTERS*：类似--clusters。
- *SALLOC_CONSTRAINT*：类似-C, --constraint。
- *SALLOC_CORE_SPEC*：类似 --core-spec。
- *SALLOC_CPUS_PER_GPU*：类似 --cpus-per-gpu。
- *SALLOC_DEBUG*：类似-v, --verbose。
- *SALLOC_EXCLUSIVE*：类似--exclusive。
- *SALLOC_GEOMETRY*：类似-g, --geometry。
- *SALLOC_GPUS*：类似 -G, --gpus。
- *SALLOC_GPU_BIND*：类似 --gpu-bind。
- *SALLOC_GPU_FREQ*：类似 --gpu-freq。
- *SALLOC_GPUS_PER_NODE*：类似 --gpus-per-node。
- *SALLOC_GPUS_PER_TASK*：类似 --gpus-per-task。
- *SALLOC_GRES*：类似 --gres。
- *SALLOC_GRES_FLAGS*：类似--gres-flags。
- *SALLOC_HINT*、*SLURM_HINT*：类似--hint。
- *SALLOC_IMMEDIATE*：类似-I, --immediate。
- *SALLOC_KILL_CMD*：类似-K, --kill-command。
- *SALLOC_MEM_BIND*：类似--mem-bind。
- *SALLOC_MEM_PER_CPU*：类似 --mem-per-cpu。
- *SALLOC_MEM_PER_GPU*：类似 --mem-per-gpu
- *SALLOC_MEM_PER_NODE*：类似 --mem。
- *SALLOC_NETWORK*：类似--network。
- *SALLOC_NO_BELL*：类似--no-bell。
- *SALLOC_OVERCOMMIT*：类似-O, --overcommit。
- *SALLOC_PARTITION*：类似-p, --partition。

- *SALLOC_PROFILE*：类似--profile。
- *SALLOC_QOS*：类似--qos。
- *SALLOC_RESERVATION*：类似--reservation。
- *SALLOC_SIGNAL*：类似--signal。
- *SALLOC_SPREAD_JOB*：类似--spread-job。
- *SALLOC_THREAD_SPEC*：类似--thread-spec。
- *SALLOC_TIMELIMIT*：类似-t, --time。
- *SALLOC_USE_MIN_NODES*：类似--use-min-nodes。
- *SALLOC_WAIT_ALL_NODES*：类似--wait-all-nodes。
- *SLURM_EXIT_ERROR*：错误退出代码。
- *SLURM_EXIT_IMMEDIATE*：当--immediate选项时指定的立即退出代码。

34.4 主要输出环境变量

- *SLURM_CLUSTER_NAME*：集群名。
- *SLURM_CPUS_PER_GPU*：每颗GPU分配的CPU数。
- *SLURM_CPUS_PER_TASK*：每个任务分配的CPU数。
- *SLURM_DISTRIBUTION*：类似-m, --distribution。
- *SLURM_GPUS*：需要的GPU颗数，仅提交时有-G, --gpus时。
- *SLURM_GPU_BIND*：指定绑定任务到GPU，仅提交时具有--gpu-bind参数时。
- *SLURM_GPU_FREQ*：需求的GPU频率，仅提交时具有--gpu-freq参数时。
- *SLURM_GPUS_PER_NODE*：需要的每个节点的GPU颗数，仅提交时具有--gpus-per-node参数时。
- *SLURM_GPUS_PER_SOCKET*：需要的每个socket的GPU颗数，仅提交时具有--gpus-per-socket参数时。
- *SLURM_GPUS_PER_TASK*：需要的每个任务的GPU颗数，仅提交时具有--gpus-per-task参数时。

- *SLURM_JOB_ACCOUNT*：账户名。
- *SLURM_JOB_ID*（*SLURM_JOBID*为向后兼容）：作业号。
- *SLURM_JOB_CPUS_PER_NODE*：分配的每个节点CPU数。
- *SLURM_JOB_NODELIST*（*SLURM_NODELIST*为向后兼容）：分配的节点名列表。
- *SLURM_JOB_NUM_NODES*（*SLURM_NNODES*为向后兼容）：作业分配的节点数。
- *SLURM_JOB_PARTITION*：作业使用的队列名。
- *SLURM_JOB_QOS*：作业的QOS。
- *SLURM_JOB_RESERVATION*：预留的作业资源。
- *SLURM_MEM_BIND*：--mem-bind选项设定。
- *SLURM_MEM_BIND_VERBOSE*：如--mem-bind选项包含verbose选项时，则由其设定。
- *SLURM_MEM_BIND_LIST*：内存绑定时设定的bit掩码。
- *SLURM_MEM_BIND_PREFER*：--mem-bin prefer优先权。
- *SLURM_MEM_BIND_TYPE*：由--mem-bind选项设定。
- *SLURM_MEM_PER_CPU*：类似--mem。
- *SLURM_MEM_PER_GPU*：每颗GPU需要的内存，仅提交时有--mem-per-gpu参数时。
- *SLURM_MEM_PER_NODE*：类似--mem。
- *SLURM_SUBMIT_DIR*：运行*salloc*时的目录，或提交时由-D, --chdir参数指定。
- *SLURM_SUBMIT_HOST*：运行salloc时的节点名。
- *SLURM_NODE_ALIASES*：分配的节点名、通信地址和主机名，格式类似*SLURM_NODE_ALIASES*=ec0:1.2.3.4:foo,ec1:1.2.3.5:bar。
- *SLURM_NTASKS*：类似-n, --ntasks。
- *SLURM_NTASKS_PER_CORE*：--ntasks-per-core选项设定的值。
- *SLURM_NTASKS_PER_NODE*：--ntasks-per-node选项设定的值。
- *SLURM_NTASKS_PER_SOCKET*：--ntasks-per-socket选项设定的值。

- *SLURM_PROFILE*： 类似--profile。
- *SLURM_TASKS_PER_NODE*:每个节点的任务数。值以,分隔,并与*SLURM_NODELIST*顺序一致。如果连续的节点有相同的任务数，那么数后面跟有“(x#)”，其中“#”是重复次数。如：“*SLURM_TASKS_PER_NODE*=2(x3),1”。

34.5 例子

- 获取分配，并打开csh，以便srun可以交互式输入：

  salloc -N16 csh

  将输出：

  ```
  salloc: Granted job allocation 65537
  (在此，将等待用户输入，csh退出后结束）
  salloc: Relinquishing job allocation 65537
  ```

- 获取分配并并行运行应用：

  salloc -N5 srun -n10 myprogram

- 生成三个不同组件的异构作业，每个都是独立节点：

  salloc -w node[2-3] : -w node4 : -w node[5-7] bash

  将输出：

  ```
  salloc: job 32294 queued and waiting for resources
  salloc: job 32294 has been allocated resources
  salloc: Granted job allocation 32294
  ```

35 将文件同步到各节点：sbcast

*sbcast*命令可以将文件同步到各计算节点对应目录。

当前，用户主目录是共享的，一般不需要此命令，如果用户需要将某些文件传递到分配给作业的各节点*/tmp*等非共享目录，那么可以考虑此命令。

*sbcast*命令的基本语法为：*sbcast [-CfFjpstvV] SOURCE DEST*。

此命令仅对批处理作业或在Slurm资源分配后生成的shell中起作用。SOURCE是当前节点上文件名，DEST为分配给此作业的对应节点将要复制到文件全路径。

35.1 主要参数

- -C [library], --compress[=library]：设定采用压缩传递，及其使用的压缩库，[library]可以为lz4（默认）、zlib。
- -f, --force：强制模式，如果目标文件存在，那么将直接覆盖。
- -F number, --fanout=number：设定用于文件传递时的消息扇出，当前最大值为8。
- -j jobID[.stepID], --jobid=jobID[.stepID]：指定使用的作业号。
- -p, --preserve：保留源文件的修改时间、访问时间和模式等。
- -s size, --size=size：设定广播时使用的块大小。size可以具有k或m后缀，默认单位为比特。默认大小为文件大小或8MB。
- -t seconds, fB--timeout=seconds：设定消息的超时时间。
- -v, --verbose：显示冗余信息。
- -V, --version：显示版本信息。

35.2 主要环境变量

- SBCAST_COMPRESS：类似-C, --compress。
- SBCAST_FANOUT：类似-F number, fB--fanout=number。
- SBCAST_FORCE：类似-f, --force。
- SBCAST_PRESERVE：类似-p, --preserve。
- SBCAST_SIZE：类似-s size, --size=size。
- SBCAST_TIMEOUT：类似-t seconds, fB--timeout=seconds。

35.3 例子

将*my.prog*传到*/tmp/my.proc*，且执行：

- 生成脚本*my.job*：

```
#!/bin/bash
sbcast my.prog /tmp/my.prog
srun /tmp/my.prog
```

- 提交：

  sbatch --nodes=8 my.job

36 吸附到作业步：sattach

*sattach*可以吸附到一个运行中的Slurm作业步，通过吸附，可以获取所有任务的IO流等，有时也可用于并行调试器。

基本语法：*sattach [options] <jobid.stepid>*

36.1 主要参数

- -h, --help：显示帮助信息。
- --input-filter[=]<task number>、--output-filter[=]<task number>、--error-filter[=]<task number>：仅传递标准输入到一个单独任务或打印一个单个任务中的标准输出或标准错误输出。
- -l, --label：在每行前显示其对应的任务号。
- --layout：联系slurmctld获得任务层信息，打印层信息后退出吸附作业步。
- --pty：在伪终端上执行0号任务。与--input-filter、--output-filter或--error-filter不兼容。
- -Q, --quiet：安静模式。不显示一般的sattach信息，但错误信息仍旧显示。
- -u, --usage：显示简要帮助信息。
- -V, --version：显示版本信息。
- -v, --verbose：显示冗余信息。

36.2 主要输入环境变量

- SLURM_CONF：Slurm配置文件。
- SLURM_EXIT_ERROR：Slurm退出错误代码。

36.3 例子

- *sattach 15.0*
- *sattach -{}-output-filter 5 65386.15*

37 查看记账信息：sacct

*sacct*显示激活的或已完成作业或作业步的记账（与机时费对应）信息。

主要参数：

- -b, --brief：显示简要信息，主要包含：作业号jobid、状态status和退出码exitcode。
- -c, --completion：显示作业完成信息而非记账信息。
- -e, --helpformat：显示当采用 --format指定格式化输出的可用格式。
- -E end_time, --endtime=end_time：显示在end_time时间之前（不限作业状态）的作业。有效时间格式：
  - HH:MM[:SS] [AM|PM]
  - MMDD[YY] or MM/DD[/YY] or MM.DD[.YY]
  - MM/DD[/YY]-HH:MM[:SS]
  - YYYY-MM-DD[THH:MM[:SS]]
- -i, --nnodes=N：显示在特定节点数上运行的作业(N = min[-max)]。
- -j job(.step) , --jobs=job(.step)：限制特定作业号（步）的信息，作业号（步）可以以,分隔。
- -l, --long：显示详细信息。
- -n, --noheader：不显示信息头（显示出的信息的第一行，表示个列含义）。
- -N node_list, --nodelist=node_list：显示运行在特定节点的作业记账信息。
- --name=jobname_list：显示特定作业名的作业记账信息。
- -o, --format：以特定格式显示作业记账信息，格式间采用,分隔，利用-e, --helpformat可以查看可用的格式。各项格式中%NUMBER可以限定格式占用的字符数，比如format=name%30，显示name列最多30个字符，如数字前有-则该列左对齐（默认时右对齐）。

- -r, --partition：显示特定队列的作业记账信息。
- -R reason_list, --reason=reason_list：显示由于reason_list（以,分隔）原因没有被调度的作业记账信息。
- -s state_list, --state=state_list：显示state_list（以,分隔）状态的作业记账信息。
- -S, --starttime：显示特定时间之后开始运行的作业记账信息，有效时间格式参见前面-E参数。

38 其它常用作业管理命令

38.1 终止作业：scancel job_id

如果想终止一个作业，可利用*scancel job_id*来取消，job_list可以为以,分隔的作业ID，如：

hmli@login01:~$ scancel 7

38.2 挂起排队中尚未运行的作业：scontrol hold job_list

scontrol hold job_list（job_list可以为以,分隔的作业ID或jobname=作业名）命令可使得排队中尚未运行的作业（设置优先级为0）暂停被分配运行，被挂起的作业将不被执行，这样可以让其余作业优先得到资源运行。被挂起的作业在用*squeue*命令查询显示的时NODELIST(REASON)状态标志为JobHeldUser（被用户自己挂起）或JobHeldAdmin（被系统管理员挂起），利用*scontrol release job_list*可取消挂起。下面命令将挂起作业号为7的作业：

hmli@login01:~/work$ scontrol hold 7

38.3 继续排队被挂起的尚未运行作业：scontrol release job_list

被挂起的作业可以利用*scontrol release job_list*来取消挂起，重新进入等待运行状态，job_list可以为以,分隔的作业ID或jobname=作业名。

hmli@login01:~/work$ scontrol release 7

38.4 重新运行作业：scontrol requeue job_list

利用*scontrol requeue job_list*重新使得运行中的、挂起的或停止的作业重新进入排队等待运行，job_list可以为以,分隔的作业ID。*hmli@login01:~/work$ scontrol requeue 7*

38.5 重新挂起作业：scontrol requeuehold job_list

利用*scontrol requeuehold job_list*重新使得运行中的、挂起的或停止的作业重新进入排队，并被挂起等待运行，job_list可以为以,分隔的作业ID。之后可利用*scontrol release job_list*使其运行。

hmli@login01:~/work$ scontrol requeuehold 7

38.6 最优先等待运行作业：scontrol top job_id

利用*scontrol top job_list*可以使得尚未开始运行的job_list作业排到用户自己排队作业的最前面，最优先运行，job_list可以为以,分隔的作业ID。

hmli@login01:~/work$ scontrol top 7

38.7 等待某个作业运行完：scontrol wait_job job_id

利用*scontrol wait_job job_id*可以等待某个job_id结束后开始运行，一般用于脚本中。

hmli@login01:~/work$ scontrol wait_job 7

38.8 更新作业信息：scontrol update SPECIFICATION

利用*scontrol update SPECIFICATION*可以更新作业、作业步等信息，SPECIFICATION格式为*scontaol show job*显示出的，如下面命令将更新作业号为7的作业名为NewJobName：

scontrol update JobId=7 JobName=NewJobName

Part XII

联系方式

- 超级计算中心:
  - 电话：0551-63602248、63601540
  - 信箱：sccadmin@ustc.edu.cn
  - 主页：http://scc.ustc.edu.cn
  - 办公室：中国科大东区新图书馆一楼东侧超级计算中心126、124室
- 李会民:
  - 电话：0551-63600316
  - 信箱：hmli@ustc.edu.cn
  - 主页：http://hmli.ustc.edu.cn
  - 办公室：中国科大东区新科研楼网络信息中心204室