中国科大超级计算中心用户使用文档

发行版本 *2021-03*

中国科大超级计算中心

2022 年 12 月 17 日

Contents

CHAPTER 1

前言

本用户使用指南主要将对相关指示做一基本介绍，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：

```
QUEUE_NAME      PRIO STATUS          MAX JL/U JL/P JL/H NJOBS  PEND   RUN  SUSP
serial           50  Open:Active       -   16     -     -     0     0     0     0
long             40  Open:Active       -    -     -     -     0     0     0     0
normal           30  Open:Active       -    -     -     -     0     0     0     0
```

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，我们将尽力完善。

CHAPTER 2

现有超级计算系统

2.1 瀚海 22 超级计算系统

瀚海 22 超级计算系统，含有 2 个管理节点、2 个用户登录节点、25 个 8 卡 GPU 计算节点（单节点 64 颗 Intel Xeon Scale 8358 CPU 核（2.6GHz，48MB L3 Cache）、1TB 内存、8 颗 NVIDIA A100 Tensor Core GPU（80GB 显存、SXM4、600GB/s NVLink 卡间互联）），11PB 可用容量高性能存储，采用 Mellanox HDR 200Gbps InfiniBand 高速互联。纯计算节点共 1600 颗 CPU 核及 200 颗 A100 GPU，总双精度浮点计算能力：2.07PFLOPS（千万亿次/秒，CPU：0.13PFLOPS，GPU：1.94PFLOPS）；Tensor Float 32(TF32)AI 算力：62.4PFLOPS。

- 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	高速网络	型号
admin22-[01 - 02]	2*Intel Xeon Scale 8358（2.6GHz，32 核，L3 Cache 48MB），64 核/节点	256GB DDR4 3200MHz	2*1.6TB NVMe	HDR 100Gbps InfiniBand	浪潮 NF5280M6

- 用户登录节点（2 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	高速网络	型号
hanhai22-[01 - 02]	2*Intel Xeon Scale 8358（2.6GHz，32 核，L3 Cache 48MB），64 核/节点	256GB DDR4 3200MHz	2*1.6TB NVMe	HDR 100Gbps InfiniBand	浪潮 NF5280M6

- GPU 计算节点（25 个）：

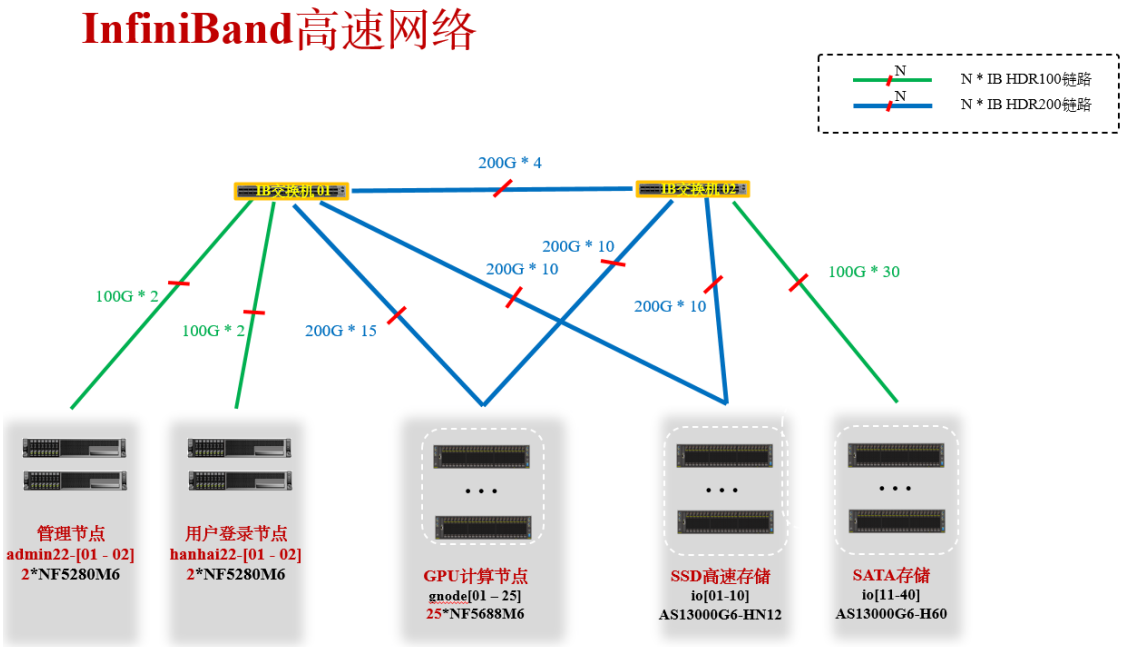
  适合 GPU 应用，加速性能：https://developer.nvidia.com/hpc-application-performance。

节点名	CPU	内存	GPU	硬盘	高速网络	型号
gn-ode[01 - 25]	2*Intel Xeon Scale 8358（2.6GHz，48MB L3 Cache），64 核/节点	1TB DDR4 3200MHz	8*NVIDIA A100（SXM4，80GB 显存）	3.84TB NVMe	HDR 200Gbps InfiniBand	浪潮 NF5688M6

表 1: 单颗 NVIDIA A100 Tensor Core GPU 参数

指标	数值
FP64 峰值性能	9.7TFLOPS
FP64 Tensor Core 峰值性能	19.5TFLOPS
FP32 峰值性能	19.5TFLOPS
FP32 Tensor Core 峰值性能	312TFLOPS
BFLOAT16 Tensor Core 峰值性能	624TFLOPS
FP16 Tensor Core 峰值性能	624TFLOPS
INT8 Tensor Core 峰值性能	1248TOPS
GPU 显存	80GB
GPU 显存带宽	1935GB/s
互联	NVIDIA NVLink 600GB/s
多实例 GPU	最大 7 个 MIG，每个 10GB

- 存储系统：
  - 10 台浪潮 AS13000G6-HN12，每台 8 块 3.2TB NVMe SSD 硬盘，总可用容量 194.6TB
  - 30 台浪潮 AS13000G6-HN60，每台 59 块 8TB NL-SAS 7.2K RPM 硬盘，总可用容量 11.56PB
  - 文件系统：Spectrum Scale Advanced
  - 默认用户磁盘配额：500GB
  - IO 性能（通过 IOR 工具）：
  - 1MB 块读带宽：单流 ≥6GB/s，聚合 ≥160GB/s
  - 1MB 块写带宽：单流 ≥6GB/s，聚合 ≥100GB/s
  - 8KB 块 IOPS，随机读 ≥205 万，随机写 ≥80 万
- 计算网络：Mellonax HDR 200Gbps InfiniBand
- 管理网络：千兆以太网
- 操作系统：Ubuntu Server 22.04 LTS
- 编译器：Intel、NVIDIA HPC SDK 和 GNU 等 C/C++ Fortran、GPU 编译器
- 数值函数库：Intel MKL
- 并行环境：HPC-X、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 22.05.3
- 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等配置文件中设置后才可以使用。



2.2 瀚海 20 超级计算系统

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
admin[01-02]	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
login[01 - 02]	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
cnode[001 - 720]	2*Intel Xeon Scale 6248(2.5GHz，20 核，27.5MB)，40 核/节点	192GB DDR4 2933MHz	1*240GB SSD	华为 FusionServer 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
anode[01 - 08]	2*Intel Xeon Scale 6248(2.5GHz，20 核，27.5MB)，40 核/节点	384GB DDR4 2933MHz	2TB(8*256GB)	4TB 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	内存	GPU	硬盘	型号
gnode[01 - 10]	2*Intel Xeon Scale 6248 (2.5GHz，20 核，27.5MB)，40 核/节点	384GB DDR4 2933MHz	2*NVIDIA Tesla V100	1TB NVMe	华为 FusionServer G530 V5
gnode-a100-[1 - 2]	2*AMD Rome 7742 (2.25GHz，64 核)，128 核/节点	1TB DDR4 3200MHz	8*NVIDIA A100 Tensor Core, 40GB，NVLink	2*1.92TB+4*3.84TB NVME	浪科联创 RW-4124GO-NART

表 2: 单颗 NVIDIA Tesla V100 GPU 参数

GPU 单元	显存	主频	核数		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 3: 单颗 NVIDIA A100 Tensor Core GPU 参数

FP64 峰值性能	9.7TFLOPS
FP64 Tensor Core 峰值性能	19.5TFLOPS
FP32 峰值性能	19.5TFLOPS
FP32 Tensor Core 峰值性能	312TF*LOPS
BFLOAT16 Tensor Core 峰值性能	624TF*LOPS
FP16 Tensor Core 峰值性能	624TF*LOPS
INT8 Tensor Core 峰值性能	1248TOPS*
INT4 Tensor Core 峰值性能	2496TOPS*
GPU 内存	40GB
GPU 内存带宽	1555GB/s
互联	NVIDIA NVLink 600GB/s
多实例 GPU	各种实例大小（最大为 7 MIG @10 GB）

- 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
rnode[01 - 09], rnode[11 - 21]	2* 鲲鹏 920 5250（48 核, 2.6GHz），96 核/节点	256GB DDR4 2666MHz	1*300GB SAS	100Gbps 以太网（支持 RoCE）	华为 TaiShan 2280 V2

其中：rnode[12-21] 每台配置 6 颗 Atlas 300 AI 卡，rnode[01-11] 未配置。

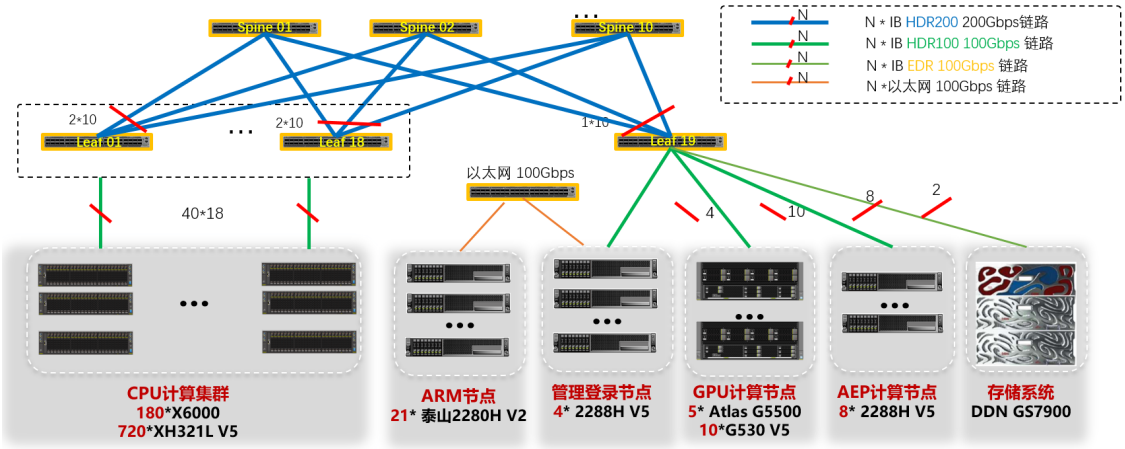
表 4: 单颗 Atlas 300 AI 卡参数

内存	AI 算力	编解码能力
LPDDR4x 32 GB，3200 Mbps	64TOPS INT8，256TFLOPS FP16，256TFLOPS 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
  - 并行文件系统：GRID Scaler
  - 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：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等配置文件中设置后才可以使用。

瀚海20超级计算系统计算网拓扑



2.3 曙光 TC4600 百万亿次超级计算系统

曙光 TC4600 百万亿次超级计算系统（519 万亿次/秒），兼为安徽省教育科研网高性能计算平台，也为入网用户提供高性能计算服务。此套系统采用 100Gbps 和 56Gbps 高速互联，具有 Intel Xeon E5 2600 v3、E5 2600 v4、E7 8800 v4、E3 1240 v5 等不同类型 CPU 及 Intel Xeon Phi 和 NVIDIA Tesla K80 GPU 等处理器，共计 1 个管理节点、2 个用户登录节点、7 个存储节点、2 个 Lustre LNet 路由节点及 506 个计算节点构成。计算节点共 12200 颗 CPU 核心，512 颗 Intel Xeon Phi 融核（MIC）KNL 核心和 39936 颗 NVIDIA CUDA 核心，总双精度峰值计算能力为每秒 519 万亿次（CPU：482.82 万亿次/秒，GPU：14.96 万亿次/秒，Intel Xeon Phi 融核：21.28 万亿次/秒）。

- 管理节点：

  用于系统管理，普通用户无权登录。

节点名	CPU	内存	硬盘	计算网络	型号
tcad-min	2*Intel Xeon E5-2620 v3，共 12 核	32GB DDR4 2133MHz	2*600GB SAS	56Gbps FDR InfiniBand	曙光 I620-G10

- 用户登录节点（2 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。
  - Intel Xeon E5 2600 v3、E5 2600 v4 和 E7 8800 v4 是同一架构 CPU，编译时可以通用。
  - 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	计算网络	型号
tc4600v3	2*Intel Xeon E5-2620 v3，共 12 核	32GB DDR4 2133MHz	2*600GB SAS	56Gbps FDR InfiniBand	曙光 I620-G10
tc4600v4	2*Intel Xeon E5-2620 v4，共 16 核	64GB DDR4 2133MHz	2*240GB SSD	100Gbps Intel OPA	曙光 I620-G10

- Intel Xeon CPU 计算节点（494 个）：

节点名	数量	CPU	内存	硬盘	计算网络	备注
node[1 - 300]	300	2*Intel Xeon E5-2680v3 (2.5GHz，30MB L3 Cache)，共 24 核	64GB DDR4 2133MHz	1*300GB SAS	56Gbps FDR InfiniBand	曙光 CX50-G20
node[301 - 450]	150	2*Intel Xeon E5-2680 v4 (2.4GHz, L3 Cache)，共 28 核	128GB DDR4 2400MHz	1*240GB SSD	100Gbps Intel OPA	曙光 CX50-G20 X50-G20
node[451 - 490]	140	1*Intel Xeon E3-1240 v5 (3.5GHz, 8MB L3 Cache)，共 4 核	32GB DDR4 2400MHz	1*240GB SSD	25Gbps EDR Infiniband	曙光 CX50-G20，高主频，每四个节点通过一块 Mellonax Multi-Host 互联
node[491 - 494]	4	8*Intel Xeon E7-8860 v4 (2.2GHz, 45MB L3 Cache)，共 144 核	1TB DDR4 2400MHz	2*240GB SSD	100Gbps Intel OPA	曙光 I980-G20，大共享内存

- Intel Xeon 融核计算节点（8 个）：

节点名	数量	CPU	内存	硬盘	计算网络	备注
knl[1 - 8]	8	1*Intel Xeon Phi 7210 (64 核，1.3GHz 16GB MCDRAM，2.66 万亿次/秒)	96GB DDR4 2133MHz	1*160GB SSD	100Gbps Intel OPA	曙光 I620-T25

- GPU 计算节点（4 个）：

节点名	数量	GPU	CPU	内存	硬盘	计算网络	备注
k80[1 - 4]	4	2*NVIDIA Tesla K80	2*Intel Xeon E5-2680 v4 (2.4GHz, 30MB L3 Cache)，共 28 核	128GB DDR4 2400MHz	2*240GB SSD	100Gbps Intel OPA	曙光 W740-G20

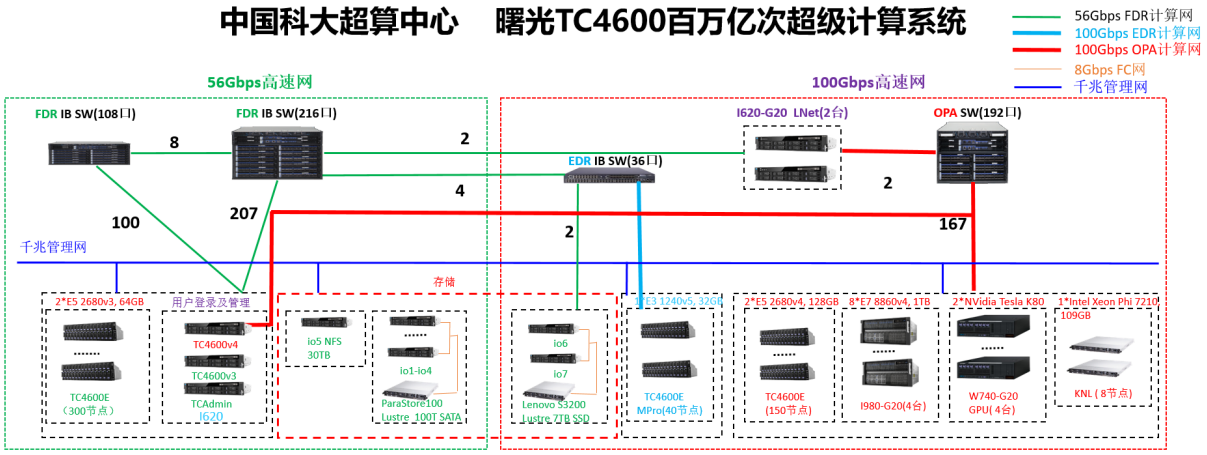
单颗 NVIDIA Tesla K80 GPU 参数

GPU 单元	显存	主频	CUDA 计算核心数	单精度计算能力	双精度计算能力
2*GK210	24GB GDDR5	562MHz, GPU Boost 875MHz	4992	5.6 万亿次/秒	1.87 万亿次/秒

- 存储系统：
  - Lustre SATA 并行存储系统：
    - 曙光 DS800-G10 机架式存储，34 块 4TB SATA 硬盘，实际可用空间 102TB
    - Lustre 并行 IO 节点（io1-io4，曙光 I620-G10 机架式服务器 4 台）：
      - 两颗 64 位主频 2.1GHz 的 Intel Xeon E5-2620 v2 x86_64 6 核 CPU，共 12 核
      - 32GB DDR3 1600MHz 内存
      - 两块 600GB SAS 硬盘

        - 一块 56Gbps FDR InfiniBand 卡
        - 一块 8Gbps FC 卡
    - Lustre SSD 闪存并行存储系统：
        - 高 IO 读写性能，仅提供有需要的特殊用户使用，需特殊申请
        - 联想 S3200 机架式存储，24 块 4GB SSD 硬盘，实际可用空间 7TB
        - Lustre 并行 IO 节点（io6、io7，联想 System x3650 M5 机架式服务器 2 台）：
            - 两颗 64 位主频 2.2GHz 的 Intel Xeon E5-2630 v4 x86_64 10 核 CPU，共 20 核
            - 64GB DDR4 2400MHz ECC REG 内存
            - 两块 240GB SSD 硬盘
            - 一块 56Gbps FDR InfiniBand 卡
            - 两块 16Gbps 双口 FC 卡
    - Lustre LNet 路由节点：
        - 用途：Intel OPA 高速计算网络节点通过此路由节点访问基于 InifiniBand 高速网络的 Lustre 存储
        - 节点名：lnet1、lnet2
        - 曙光 I620-G20 机架式服务器 2 台
        - 各节点配置：
            - 两颗 64 位主频 2.1GHz 的 Intel Xeon E5-2620 v4 x86_64 8 核 CPU，共 16 核
            - 64GB DDR4 2400MHz ECC REG 内存
            - 两块 240GB SSD 硬盘
            - 一块 100Gbps Intel OFI 卡
            - 一块 56Gbps FDR InfiniBand 卡
    - NFS 普通 IO 节点：
        - 节点名：io5
        - 曙光 I620-G20 机架式服务器，实际可用空间 30TB：
            - 两颗主频 2.6GHz 的 Intel Xeon E5-2640 v3 x86_64 8 核 CPU，共 16 核
            - 64GB DDR4 2133MHz ECC REG 内存
            - 12 块 4TB 3.5 寸 7.2K NL-SAS 硬盘
            - 1 块 2GB Cache RAID 卡
            - 1 块 56Gbps FDR InfiniBand 双端口 HCA 卡
- 高速计算网络：
    - 100Gbps Intel Omni Path 全线速高速网，Intel 100SWD06 192 口交换机一台
    - 100Gbps EDR InfiniBand 高速网，Mellanox SB7800 36 口交换机一台
    - 56Gbps FDR InfiniBand 准全线速高速网，Mellanox SX6512 216 口、SX6506 108 口交换机各一台
- 管理网络：千兆以太网
- 操作系统：CentOS 7.3 Linux

- 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：IBM Spectrum LSF
- 常用公用软件安装目录:。请自己查看有什么软件，有些软件需要在自己等配置文件中设置后才可以使用。



CHAPTER 3

用户登录与文件传输

本超算系统的操作系统为 64 位 **CentOS 7.x** 和 **Ubuntu Server 22.04 LTS 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]，Windows 的命令行也支持 ssh 及 sftp 命令）登录到用户登录节点后进行编译、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。如希望使用英文或 GBK 中文，可以在自己的中添加 `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` 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；Ubuntu 基于 Debian 开发而成。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，详细信息可参考 CentOS、Red Hat Enterprise Linux、Ubuntu、Debian 手册或通用 Linux 手册。

[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

CHAPTER 4

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 或 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 是必需的。

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`

  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`。

CHAPTER 5

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[1] 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[module]），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] 目前尚未配置，等以后配置上

5.1 串行 C/C++ 程序的编译

5.1.1 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，见下表。

表 1: 输入文件后缀与类型的关系

文件名	解释	动作
filename.c filename.C filename.CC	C 源文件	传给编译器
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

编译器默认将输出按照文件类型与后缀相对应，见下表。

表 2: 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

5.1.2 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
    ```

5.2 串行 Fortran 程序的编译

5.2.1 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，见下表。

表 3: 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

编译器默认将输出按照文件类型与后缀相对应，见下表。

表 4: 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

5.2.2 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。

5.3 OpenMP 程序的编译与运行

5.3.1 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
    ```

5.3.2 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。

CHAPTER 6

Intel、PGI 及 GNU C/C++ Fortran 编译器介绍

6.1 Intel C/C++ Fortran 编译器

6.1.1 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，其下有多种年份版本。官方手册目录为 `/opt/intel` 。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 (Intelmath) 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，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 = 3.14159265358979323 8/4.0;

    // pi/4 radians is about 45 degrees
    fp32bits = (float) pi_by_four; // float approximation to pi/4
```

(续下页)

(接上页)

```
    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`

6.1.2 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。

6.1.3 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。

6.1.4 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。

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。

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 IntelAdvanced Vector Extensions 2 (IntelAVX2)、IntelAVX、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、IntelAdvanced Vector Extensions (IntelAVX)、IntelSSE4.2、SSE4.1、SSE3、SSE2、SSE 和 SSSE3 指令。
  - AVX：生成 IntelAdvanced Vector Extensions (IntelAVX)、IntelSSE4.2、SSE4.1、SSE3、SSE2、SSE 和 SSSE3 指令。
  - SSE4.2：生成 IntelSSE4.2、SSE4.1、SSE3、SSE2、SSE 和 SSSE3 指令。
  - SSE4.1：生成 IntelSSE4.1、SSE3、SSE2、SSE 和 SSSE3 指令
  - SSSE3：生成 SSSE3 指令和 IntelSSE3、SSE2 和 SSE 指令。
  - SSE3：生成 IntelSSE3、SSE2 和 SSE 指令。
  - SSE2：生成 IntelSSE2 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 IntelAdvanced Vector Extensions (IntelAVX)、IntelSSE4.2、SSE4.1、SSE3、SSE2、SSE 和 SSSE3 指令。
  - sse4.2：生成 IntelSSE4.2、SSE4.1、SSE3、SSE2、SSE 和 SSSE3 指令。
  - sse4.1：生成 IntelSSE4.1、SSE3、SSE2、SSE 和 SSSE3 指令。
  - ssse3：生成 SSSE3 指令和 IntelSSE3、SSE2 和 SSE 指令。
  - sse3：生成 IntelSSE3、SSE2 和 SSE 指令。
  - sse2：生成 IntelSSE2 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 Intel64 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。

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。

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。

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。

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）。

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。

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。

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.

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。

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。

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歧。
- -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。

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。

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。

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。

兼容性选项

- -f66：使用 FORTRAN 66 标准，默认为使用 Fortran 95 标准。
- -f77rtl、-nof77rtl：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：设定是否某些特征与 IntelFortran 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 IntelFortran 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。

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。

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。

6.2 PGI C/C++ Fortran 编译器

6.2.1 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、。安装在，可用 `module avail` 查看，用 `moudle load 模块名`使用，或在自己的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++` [1]_ 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：。

6.2.2 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：

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

6.2.3 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 Runtime Error Messages->9.4.2. Message List

6.2.4 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。

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*n* 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。

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。

调试选项

- -g：包含调试信息。

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。

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。

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。

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。

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:math:1'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。

6.3 GNU C/C++ Fortran 编译器

6.3.1 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++`；`gfortran` 可以直接编译 Fortran 77、90 源程序。

6.3.2 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

6.3.3 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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

CHAPTER 7

GPU 异构计算和 CUDA 程序简介

Author
吴超，中国科大超级计算中心

7.1 背景

近年来，随着人工智能、高性能数据分析和金融分析等计算密集型领域的兴起，传统通用计算已经无法满足对计算性能的需求，异构计算越来越引起学术界和产业界的重视。

异构计算是指采用不同类型的指令集和体系架构的计算单元组成系统的计算方式。相比传统 CPU，异构计算可以实现更高的效率和更低的延迟。目前的异构计算引擎主要有图形处理器（GPU，Graphics Processing Unit）、现场可编程门阵列（FPGA，Field Programming Gate Array）、专用集成电路（ASIC）等。

当前的通用 CPU 设计得已经很复杂，配有几十个核心，运行频率高达几 GHz，每个核心有自己的独立缓存。通常 CPU 已具备一级、二级、三级缓存。而 GPU 是目前科研领域比较常用的硬件计算工具。GPU 的计算核心数通常是 CPU 的上百倍，运行频率尽管比 CPU 的低，但是核心数量多，整体性能好。所以，GPU 比较适合计算密集型应用，比如视频处理、人工智能等，现在传统的科学计算、工程计算等也开始越来越适合在 GPU 上运行。相比来说，CPU 的缺点就是太通用了，数据读写、计算、逻辑等各种功能都得照顾，反而影响了计算性能。

7.2 GPGPU

通用图形处理器（GPGPU，General Purpose Graphics Processing Unit）最早由 NVIDIA 公司的 Mark J. Harris 于 2002 年提出。基于图形渲染管线的流水线特征，GPU 本质上是一个可同时处理多个计算任务的硬件加速器。由于 GPU 中包含了大量的计算资源，Mark J. Harris 自 2002 年就开始尝试在 GPU 上做通用并行计算方面的研究。在此阶段，由于架构及编程平台的限制，研究人员采用将目标计算算法转换为图形运算算法的方式，使用 GPU 来实现通用并行计算需求。

NVIDIA 公司提出 Tesla 统一渲染架构以及 CUDA（Compute Unified Device Architecture，计算统一设备架构）编程模型后，NVIDIA 公司的 GPU 开始了对通用并行计算的全面支持。在 CUDA 提出近两年之后，开放计算语言标准 OpenCL 1.0 发布，这标志着利用 GPU 进行通用并行计算已基本成熟。目前市场上应用甚广的

GPU 芯片除了完成高质量的图形渲染之外，通用并行计算也已经成为一个主流应用。GPGPU 在各个方面得到了不同 GPU 厂家为 GPU 通用计算提供的编程模型与平台，如 CUDA 和 OpenCL，这些编程模型在 C/C++ 基础之上做了面向大规模通用并行计算的语法扩展，为程序员提供了更好的、面向 GPU 的编程接口。

GPGPU 通常由成百上千个架构相对简易的基本运算单元组成。在这些基本运算单元中，一般不提供复杂的诸如分支预测、寄存器重命名、乱序执行等处理器设计技术来提高单个处理单元性能，而是采用极简的流水线进行设计。每个基本运算单元可同时执行一至多个线程，并由 GPGPU 中相应的调度器控制。GPGPU 作为一个通用的众核处理器，凭借着丰富的高性能计算资源以及高带宽的数据传输能力在通用计算领域占据了重要的席位。虽然各个 GPGPU 厂商的芯片架构各不相同，但几乎都是采用众核处理器阵列架构，在一个 GPU 芯片中包含成百上千个处理核心，以获得更高的计算性能和更大的数据带宽。

GPU 中执行的线程对应的程序通常成为内核（kernel），这与操作系统中的内核是完全不同的两个概念。除此之外，GPU 中执行的线程与 CPU 或者操作系统中定义的线程也有所区别，GPU 中的线程相对而言更为简单，所包含的内容也更为简洁。在 GPU 众核架构中，多个处理核心通常被组织成一个线程组调度执行单位，线程以组的方式被调度在执行单元中执行，如 NVIDIA 的流多处理器、AMD 的 SIMD 执行单元。同一个线程组中的线程执行相同的程序指令，并以同步的方式执行，每个线程处理不同的数据，实现数据级并行处理。不同 GPU 架构对线程组的命名各不一样，如 NVIDIA 将线程组称为 warp，AMD 将线程组称为 wavefront。线程组中包含的线程数量各不相同，从 4 个到 128 个不等。除此以为，线程组的组织执行模式也各不相同，常见的执行模式有 SIMT（Single Instruction Multiple Threads，单指令多线程）执行模式和 SIMD（Single Instruction Multiple Data，单指令流多数据流）执行模式两种。

在一个 GPU 程序中，避免不了对数据的加载和存储，同时也避免不了条件分支跳转指令。这两类指令通常会引起程序以不可预测的情况执行。对于前者，在第一级高速缓存命中缺失的情况下，指令的执行周期将不可预测。为了避免执行单元因为数据加载或者存储原因而造成运算资源的浪费，GPU 的每个执行单元通常设置线程组缓冲区，以支持同时执行多个线程组。线程组之间的调度由线程组硬件调度器承担，与软件调度器不同的是，硬件调度过程一般为零负载调度。在执行单元中，即将执行的线程组首先被调度到缓冲区中，以队列的方式组织，当线程组被调度执行时，调度器从线程组队列中选择一个准备好的线程组启动执行。采用这种线程调度执行方式，可有效解决指令之间由于长延时操作所引起的停顿问题，更高效的应用执行单元中的计算资源。对于后者，在线程级并行执行过程中，条件分支指令的执行特点决定了程序执行的实际效率。无论是 SIMD 执行模式或是 SIMT 执行模式，当一组线程均执行相同的代码路径时可获得最佳性能。若一组线程中的每个线程各自执行不同的代码路径，为了确保所有线程执行的正确性，线程组中的多线程指令发送单元将串行地发送所有的指令代码，代码的执行效率将受到严重的影响。GPU 架构采用各种控制方法来提高条件分支指令的执行效率。

背景知识大部分内容引自[1]。

7.3 GPU 异构计算

CPU-GPU 协同是实现高性能计算的必要条件，称为 CPU-GPU 异构计算（HC，Heterogeneous Computing）。它通过将应用程序的计算密集型部分卸载到 GPU 来提供更高的性能，而其余代码仍然在 CPU 上运行，能智能地结合 CPU 和 GPU 的最佳特性以实现高计算增益，旨在将每个应用程序的需求与 CPU/GPU 架构的优势相匹配，并避免两个处理单元的空闲时间。需要新的优化技术来充分发挥 HC 的潜力并朝着百亿级性能的目标迈进。

了解 CPU 和 GPU 之间差异的一种简单方法是比较它们处理任务的方式。CPU 由几个针对顺序串行处理优化的内核组成，而 GPU 具有大规模并行架构，由数千个更小、更高效的内核组成，旨在同时处理多个任务。

在 GPU 上解决计算问题原则上类似于使用多个 CPU 解决问题。手头的任务必须拆分为小任务，其中每个任务由单个 GPU 内核执行。GPU 内核之间的通信由 GPU 芯片上的内部寄存器和内存处理。CUDA 或 OpenCL 等特殊编程语言不是使用消息传递进行编程，而是提供主机 CPU 之间的数据交换和同步 GPU 内核的机制。

一个现代超级计算系统实际上可能由大量节点组成，每个节点包含 2 到 32 颗常规 CPU 以及 1 到 16 个 GPU。通常还会有一个高速网络和一个数据存储系统。该系统的软件可以使用传统编程语言（如 C/C++、Fortran

[1] 陈国良, 吴俊敏. 并行计算机体系结构（第 2 版）[M]. 北京: 高等教育出版社, 2021.

等）的组合编写，结合用于 CPU 并行化的消息传递系统以及用于 GPU 的 CUDA 或 OpenCL。所有这些组件都必须进行调整和优化，以实现整个系统的最佳性能。

7.4 CUDA 编程框架

CUDA 是由 NVIDIA 公司推行的一套并行编程框架，目前只有 NVIDIA 的 GPU 支持该框架，其开发语言主要为 CUDA C。作为一种 GPU 的并行开发语言，CUDA 的 API 涉及设备管理、存储管理、数据传输、线程管理、事件管理等功能。

CUDA 的存储模型主要分为全局存储（global memory）、局部存储（local memory）、共享存储（shared memory）、常量存储（constant memory）和纹理存储（texture memory）等存储类型。不同的存储类型，其存储容量、可见程度、读写速度差异巨大，需要在程序设计中根据各自特点和应用问题的需求合理调配。

CUDA 的编程模型和执行模型按照层次结构分层设计。CUDA 的执行模型由 3 个层级组成，最基础的执行单位是线程（thread），多个线程组成一个线程块（block），多个线程块形成线程网格（grid）。

CUDA 编程模型作为一个异构模型，其中使用了 CPU 和 GPU。在 CUDA 中，主机（host）指的是 CPU 及其存储器，设备（device）是指 GPU 及其存储器。在主机上运行的代码可以管理主机和设备上的内存，还可以启动在设备上执行的内核函数（kernel）。这些内核由许多 GPU 线程并行执行。

鉴于 CUDA 编程模型的异构性，CUDA C 程序的典型操作序列是：

1. 声明并分配主机和设备内存。
2. 初始化主机数据。
3. 将数据从主机传输到设备。
4. 执行一个或多个内核。
5. 将结果从设备传输到主机。

对于更多的 CUDA 编程细节可以在 NVIDIA 官网 CUDA 开发者页面[2] 找到详细的资料，读者可以自行查阅。

7.5 NVCC 编译引擎

CUDA 应用程序的源代码由传统的 C++ 主机代码和 GPU 设备函数混合组成。CUDA 编译过程将设备函数与主机代码分开，使用专有的 NVIDIA 编译器和汇编器编译设备函数，使用可用的 C++ 主机编译器编译主机代码，之后将编译过的 GPU 函数嵌入到主机对象（object）文件。在链接阶段，向最终生成的可运行二进制文件添加特定的 CUDA 运行时库函数，如支持远程 SPMD 过程调用的运行时库函数、显式 GPU 操作的运行时库函数（如分配 GPU 内存缓冲区，主机与设备之间的数据传输等）。

编译过程涉及每个 CUDA 源文件的拆分、编译、预处理和合并步骤。为了将上述复杂的编译过程向开发人员隐藏，NVIDIA 公司设计了 CUDA 编译器引擎程序 nvcc。nvcc 接受一系列常规编译器选项，例如宏定义和头文件、函数库路径设置，支持编译过程的组合。所有非 CUDA 编译步骤都被转发到 nvcc 支持的主机 C++ 编译器，编译选项到主机 C++ 编译选项的转换也由 nvcc 自动完成。

[2] cuda-toolkit 网址：https://developer.nvidia.com/cuda-toolkit

7.5.1 nvcc 预定义宏

预定义宏	含义
__NVCC__	编译 C/C++/CUDA 源文件时预定义
__CUDACC__	编译 CUDA 源文件时预定义
__CUDACC_VER_MAJOR__	NVCC 主版本号
__CUDACC_VER_MINOR__	NVCC 次版本号
__CUDACC_VER_BUILD__	NVCC 编译版本号

7.5.2 支持的输入文件后缀

输入文件后缀	描述
.cu	CUDA 源文件，包含主机代码和设备函数
.c	C 源文件
.cc, .cxx, .cpp	C++ 源文件
.o, .obj	目标文件（object file）
.a, .lib	库文件（library file）
.res	资源文件（resource file）
.so	共享目标文件（shared object file）

7.5.3 常用编译选项

1. 文件和路径配置

选项	描述
-o file	配置输出文件名和路径
-l library,⋯	配置链接阶段链接的库文件
-D def,⋯	定义预处理阶段使用的宏
-U def,⋯	取消宏定义
-I path,⋯	配置头文件搜索路径
-L path,⋯	配置库文件搜索路径
-cudart {none|shared|static}	配置 CUDA 运行时库的类型，默认使用静态（static）
-cudadevrt {none|static}	配置 CUDA 设备运行时库的类型，默认使用静态（static）

2. 编译器/链接器选项

选项	描述
-pg	生成供 gprof 使用的可执行代码
-g	编译带调试信息的主机代码
-G	编译带调试信息的设备代码
-O level	指定主机代码的优化级别
-dopt kind	允许设备端代码优化。当不指定-G 选项时，设备端代码优化是默认的行为
-shared	生成共享库
-x {c|c++|cu}	显式指定待编译的输入文件的编程语言，而不是由编译器根据文件后缀自动判断
-std {c++03|c++11|c++14|c++17}	指定 c++ 标准的版本

3. 特定阶段编译选项

下表列举了可以直接传递给 nvcc 封装的内部编译工具的编译选项。通过这些选项的应用，nvcc 不需要具备对内部编译工具的过多细节的了解。

选项	描述
-Xcompiler options,···	指定直接传递给编译器/预处理器的编译选项
-Xlinker options,···	指定直接传递给链接器的编译选项
-Xarchive options,···	指定直接传递给库管理器的编译选项
-Xptxas options,···	指定直接传递给 ptxas（PTX 优化汇编器）的编译选项
-Xnvlink options,···	指定直接传递给 nvlink（设备链接器）的编译选项

4. GPU 代码生成选项

选项	描述
-arch {arch|native|all|allmajor}	指定编译阶段使用的虚拟 GPU 类型
-code code,···	指定汇编优化使用的具体 GPU 类型
-use_fast_math	使用快速数学计算库

为了实现架构演进，NVIDIA 的 GPU 以不同的世代（generation）发布。新一代产品在功能和/或芯片架构方面进行重大改进，同时同一代产品中的 GPU 型号仅在配置方面存在次要差别，对功能、性能的影响适中。不同代的 GPU 其应用程序的二进制兼容性是没有保证的。例如，为 Fermi GPU 编译的 CUDA 应用程序很可能无法在 Kepler GPU 上运行（反之亦然）。这是因为每一代的指令集和指令编码与其他世代的指令编码都不相同。同一代的 GPU 由于共享相同的指令集，在满足特定条件下其二进制兼容性可以得到保证。特定条件通常是指两个没有功能差异的 GPU 版本之间的情况（例如，当一个版本是另一个版本的缩减版），或者当一个版本在功能上完全包含在另一个版本中。后者的一个例子是基础 Maxwell 版本 sm_52，其功能是所有其他 Maxwell 版本的一个子集：任何为 sm_52 编译的代码将可以在所有 Maxwell GPU 上运行。

nvcc 编译命令总是使用两个架构：一个虚拟的中间架构，加上一个真实的 GPU 架构（指定代码将运行的平台）。要使 nvcc 命令有效，真实架构必须是虚拟架构的实现。

虚拟 GPU 完全由提供给应用程序的能力和特征定义。虚拟架构提供了一个通用的指令集合，并且不涉及二进制编码格式。虚拟架构列表如下：

虚拟架构（-arch 参数）	特征描述
compute_35 compute_37	Kepler 架构支持 统一内存编程 支持动态并行
compute_50 compute_52 compute_53	+Maxwell 架构支持
compute_60 compute_61 compute_62	+Pascal 架构支持
compute_70 compute_72	+Volta 架构支持
compute_75	+Turing 架构支持
compute_80 compute_86 compute_87	+Ampere 架构支持

在 CUDA 的命名方案中，GPU 被命名为 sm_xy，其中 x 表示 GPU 的世代编号，y 表示该世代的版本。为了便于比较 GPU 的能力，执行特定的命名设计规则，如果 x1y1 <= x2y2，那么 sm_x1y1 的所有非 ISA 相关能力都包括在 sm_x2y2 中。由此可见，sm_52 确实是基础麦克斯韦模型，这也解释了为什么表格中的高条目总是对低条目的功能扩展（表格中用加号表示）。

真实架构（-code 参数）	特征描述
sm_35 sm_37	Kepler 架构支持 统一内存编程 支持动态并行
sm_50 sm_52 sm_53	+Maxwell 架构支持
sm_60 sm_61 sm_62	+Pascal 架构支持
sm_70 sm_72	+Volta 架构支持
sm_75	+Turing 架构支持
sm_80 sm_86 sm_87	+Ampere 架构支持

本节介绍了 CUDA 的 nvcc 编译引擎，并列举了 nvcc 一些基础和常用的编译选项。关于 nvcc 完整的介绍请参考官方指南 NVIDIA CUDA Compiler Driver NVCC[3]。

7.6 一个简单的例子

7.6.1 代码示例

下面介绍一个简单的 CUDA C 程序例子，演示如何在瀚海 22 上编译运行 CUDA 代码。

例子展示的是两个向量相加的 CUDA 代码 `add.cu` 。

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

(续下页)

[3] NVIDIA CUDA Compiler Driver NVCC 网址：https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

(接上页)

```
__global__
void add(int n, float *x, float *y, float *z)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) z[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *z, *d_x, *d_y, *d_z;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));
  z = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
  cudaMalloc(&d_z, N*sizeof(float));

  for (int i = 0; i < N; i++) {
        x[i] = rand()*1.0/RAND_MAX;
        y[i] = rand()*1.0/RAND_MAX;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  add<<<(N+255)/256, 256>>>(N, d_x, d_y, d_z);

  cudaMemcpy(z, d_z, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
        maxError = max(maxError, fabs(z[i]-x[i]-y[i]));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
  cudaFree(d_y);
  cudaFree(d_z);
  free(x);
  free(y);
  free(z);

  return 0;
}
```

函数 `add` 是在 GPU 上并行运行的内核，`main` 函数是宿主代码。

7.6.2 程序解读

main 函数声明 3 个数组。

```
float *x, *y, *z, *d_x, *d_y, *d_z;
x = (float*)malloc(N*sizeof(float));
y = (float*)malloc(N*sizeof(float));
z = (float*)malloc(N*sizeof(float));

cudaMalloc(&d_x, N*sizeof(float));
cudaMalloc(&d_y, N*sizeof(float));
cudaMalloc(&d_z, N*sizeof(float));
```

指针 x、y 和 z 分别指向使用 **malloc** 分配的主机内存空间，d_x、d_y 和 d_z 指针分别指向使用 CUDA 运行时 API **cudaMalloc** 函数分配的设备存储空间。CUDA 中的主机和设备有独立的内存空间，这两个空间都可以从主机代码进行管理。

为了初始化设备数组，使用 **cudaMemcpy** 将数据从 x 和 y 复制到相应的设备数组 d_x 和 d_y，它的工作方式与标准的 C **memcpy** 函数一样，只是增加了第四个参数，指定拷贝的方向。在这里，我们使用 *cudaMemcpyHostToDevice* 指定第一个（目标）参数是设备指针，第二个（源）参数是主机指针。

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

add 内核由以下语句启动：

```
add<<<(N+256-1)/256, 256>>>(N, d_x, d_y, d_z);
```

<< 和 **>>** 符号之间的信息是执行配置，指示有多少设备线程并行执行内核。在 CUDA 中，软件中有一个线程层次结构，它模仿线程处理器在 GPU 上的分组方式。执行配置中的第一个参数指定网格中线程块的数量，第二个参数指定线程块中的线程数。线程块和网格可以通过为这些参数传递 dim3（一个由 CUDA 用 x、y 和 z 成员定义的简单结构）值来生成一维、二维或三维的线程块和网格。对于 add 这个示例，只需要一维线程组，所以我们只传递整数。在本例中，我们使用包含 256 个线程的线程块启动内核，并使用“上整计算”来确定处理数组全部 N 个元素所需的线程块数（(N+256-1)/256）。

由于数组的元素数有不能被线程块大小整除的可能，内核代码必须检查内存访问是否越界。

在运行内核之后，使用 **cudaMemcpy**（拷贝方向：*cudaMemcpyDeviceToHost*），从 d_z 指向的设备数组复制到 z 指向的主机数组，将结果返回给主机。

```
cudaMemcpy(z, d_z, N*sizeof(float), cudaMemcpyDeviceToHost);
```

程序的最后，使用 **cudaFree()** 和 **free()** 分别清理设备端和主机端申请的内存。

```
cudaFree(d_x);
cudaFree(d_y);
cudaFree(d_z);
free(x);
free(y);
free(z);
```

7.6.3 编译运行

接下来以瀚海 22 超级计算系统使用 Slurm 作业调度系统的交互式任务为例，演示 CUDA 程序在超算系统上编译运行的一般方法。

- 首先在 **登录节点**，利用 **module** 命令载入合适的 CUDA 版本。本例中，在 **登录节点**运行命令 **module avail** 确认系统中已安装的 CUDA 版本，选择符合自身需求的版本，如 cuda/11.7.1_515.65.01 装载（**module load cuda/x.x.x** , x.x.x 指版本号）：

```
scc@hanhai22-02:~$ module avail
------------------------------------------------------------------------------ /opt/MODULES/app ------------------------------------------------------------------------------
gaussian/16/g16.c02  matlab/R2022a  QuantumESPRESSO/7.1/nvhpc_22.7  relion/oneapi_latest/cuda_11.7.1  vasp/6.3.2/openacc_nvhpc21.11

---------------------------------------------------------------------------- /opt/MODULES/compiler ----------------------------------------------------------------------------
cuda/11.7.1_515.65.01  hpcx/2.12/hpcx-debug       hpcx/2.12/hpcx-mt-ompi  hpcx/2.12/hpcx-prof-ompi  nvhpc-nompi/22.7      nvhpc/22.7
gcc/10.2.0             hpcx/2.12/hpcx-debug-ompi  hpcx/2.12/hpcx-ompi     hpcx/2.12/hpcx-stack      nvhpc/21.11           openmpi/4.1.4/gcc/11.2.0
hpcx/2.12/hpcx         hpcx/2.12/hpcx-mt          hpcx/2.12/hpcx-prof     nvhpc-byo-compiler/22.7   nvhpc/21.11_more_env

------------------------------------------------------------------------------ /opt/MODULES/lib ------------------------------------------------------------------------------
fftw/3.3.10

---------------------------------------------------------------------------- /opt/MODULES/oneapi ----------------------------------------------------------------------------
advisor/2022.1.0        compiler-rt32/latest  dev-utilities/2021.6.0  dnnl/latest     init_opencl/2022.1.0        intel_ippcp_ia32/latest       mpi/2021.6.0      vpl/latest
advisor/latest          compiler/2022.1.0     dev-utilities/latest    dpct/2022.1.0   init_opencl/latest          intel_ippcp_intel64/2021.6.0  mpi/latest        vtune/2022.2.0
ccl/2021.6.0            compiler/latest       dnnl-cpu-gomp/2022.1.0  dpct/latest     inspector/2022.1.0          intel_ippcp_intel64/latest    oclfpga/2022.1.0  vtune/latest
ccl/latest              compiler32/2022.1.0   dnnl-cpu-gomp/latest    dpl/2021.7.0    inspector/latest            itac/2021.6.0                 oclfpga/latest
clck/2021.6.0           compiler32/latest     dnnl-cpu-iomp/2022.1.0  dpl/latest      intel_ipp_ia32/2021.6.0     itac/latest                   tbb/2021.6.0
clck/latest             dal/2021.6.0          dnnl-cpu-iomp/latest    icc/2022.1.0    intel_ipp_ia32/latest       mkl/2022.1.0                  tbb/latest
compiler-rt/2022.1.0    dal/latest            dnnl-cpu-tbb/2022.1.0   icc/latest      intel_ipp_intel64/2021.6.0  mkl/latest                    tbb32/2021.6.0
compiler-rt/latest      debugger/2021.6.0     dnnl-cpu-tbb/latest     icc32/2022.1.0  intel_ipp_intel64/latest    mkl32/2022.1.0                tbb32/latest
compiler-rt32/2022.1.0  debugger/latest       dnnl/2022.1.0           icc32/latest    intel_ippcp_ia32/2021.6.0   mkl32/latest                  vpl/2022.1.0

---------------------------------------------------------------------------- /opt/MODULES/python ----------------------------------------------------------------------------
anaconda/3/22.05  cell2location  miniconda/3

------------------------------------------------------------------------------ /opt/MODULES/tool ------------------------------------------------------------------------------
cmake/3.19.0

Key:
modulepath
scc@hanhai22-02:~$ module load cuda/11.7.1_515.65.01
```

- （可选操作）使用 **nvcc –version** 命令可以查看确认当前环境载入的 CUDA 版本：

```
scc@hanhai22-02:~$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0
```

- 接着，使用 **nvcc** 命令编译 CUDA 程序，编译选项的含义可参考上节 “NVCC 编译引擎”：

```
scc@hanhai22-02:~$ nvcc -O2 -arch compute_80 -code sm_80 test/add.cu -o cuadd
scc@hanhai22-02:~$
```

- 然后，使用 **salloc** 命令向 Slurm 系统申请交互式任务。**salloc** 命令选项及 Slurm 系统可参考本手册的 “Slurm 作业调度系统”[4]：

```
scc@hanhai22-02:~$ salloc -N1 -p GPU-8A100 --time=1:00:00 --exclusive
salloc: Granted job allocation 3725
salloc: Waiting for resource configuration
salloc: Nodes gnode03 are ready for job
scc@hanhai22-02:~$
```

- 至此，已完成在 **登录节点**的工作（CUDA 程序的编译和 Slurm 交互式任务的提交）。我们转入 **计算节点**。
- 使用 SSH 命令免密登入 **计算节点**（计算节点的主机名称由 salloc 命令的输出确定）。

```
scc@hanhai22-02:~$ ssh gnode03
Last login: Sun Oct 16 15:55:17 2022 from hanhai22-02
scc@gnode03:~$
```

备注： 由于瀚海 22 系统的登录节点和计算节点的 */home* 目录都挂载共享存储对应目录，所以两者的 */home* 目录内容相同。

- 在 **计算节点**使用 **module load** 载入与登录节点一致的 CUDA 版本。然后运行已编译好的程序，完成计算。

```
scc@gnode03:~$ module load cuda/11.7.1_515.65.01
scc@gnode03:~$ ~/test/cuadd
Max error: 0.000000
scc@gnode03:~$
```

[4] Slurm 作业调度系统网址：http://scc.ustc.edu.cn/zlsc/user_doc/html/slurm/index.html

7.6.4 小结

本小节的简单例子用于展示瀚海 22 超级计算系统上 CUDA 程序的编译运行（交互式）的一般流程。

备注： 使用 **salloc** （交互式）提交任务是为了向初学者展示登录节点和计算节点的区别，并不是瀚海 22 的常规作业调度方式。瀚海 22 超级计算系统上的正式作业请通过编写计算脚本并以 **sbatch** 提交的方式实现资源的申请和计算的执行。

Naive 加法计算没有考虑如何充分利用 GPU 的计算资源和数据带宽。对性能优化感兴趣，希望充分利用 GPU 的计算资源的读者可以进一步阅读 NVIDIA 关于 CUDA 编程的进阶读物[5]。

7.7 One More Thing

瀚海 22 超级计算系统系统预装了 NVIDIA HPC SDK（nvhpc），方便用户充分利用 GPU 计算资源，实现科学计算应用和性能优化等工作。

NVIDIA HPC SDK 是 NVIDIA 公司提供的一个包含编译器、函数库和软件工具的软件包，包含了用于方便用户开发、增强程序性能和可移植性的一系列工具。NVIDIA HPC SDK C、C++ 和 Fortran 编译器支持使用标准 C++ 和 Fortran、OpenACC 指令和 CUDA 对 HPC 建模和模拟应用程序进行 GPU 加速。GPU 加速的数学库使普通 HPC 算法的性能最大化，优化的通信库使基于标准的多 GPU 和可扩展系统编程成为可能。性能剖析和调试工具简化了 HPC 应用程序的移植和优化，而容器化工具则使得在企业内部或在云端的部署变得容易。HPC SDK 支持 NVIDIA GPU 和运行 Linux 的 Arm、OpenPOWER 或 x86-64 CPU，为用户提供了构建 NVIDIA GPU 加速的 HPC 应用程序所需的工具。

7.7.1 编译器

- nvc：nvc 是一个用于 NVIDIA GPU 和 AMD、Intel、OpenPOWER 和 Arm CPU 的 C 语言编译器。nvc 支持 ISO C11，支持用 OpenACC 进行 GPU 编程，并支持用 OpenACC 和 OpenMP 进行多核 CPU 编程。
- nvc++：nvc++ 是一个针对 NVIDIA GPU 和 AMD、Intel、OpenPOWER 和 Arm CPU 的 C++ 语言编译器。它为目标处理器调用 C++ 编译器、汇编器和链接器，选项来自其命令行参数。nvc++ 支持 ISO C++17，支持使用 C++17 并行算法、OpenACC 和 OpenMP 的 GPU 和多核 CPU 编程。
- nvfortran：nvfortran 是一个用于 NVIDIA GPU 和 AMD、Intel、OpenPOWER 和 Arm CPU 的 Fortran 编译器。nvfortran 支持 ISO Fortran 2003/2008 的许多特性，支持使用 CUDA Fortran 进行 GPU 编程，以及使用 ISO Fortran 并行语言特性、OpenACC 和 OpenMP 进行 GPU 和多核 CPU 编程。
- nvcc：nvcc 是用于 NVIDIA GPU 的 CUDA C 和 CUDA C++ 编译器驱动程序。nvcc 接受一系列传统的编译器选项，例如用于定义宏和 include/library 路径，以及用于引导编译过程。nvcc 为 NVIDIA GPU 生成优化代码，并驱动支持 AMD、Intel、OpenPOWER 和 Arm CPU 的主机编译器。

[5] CUDA C++ Programming Guide 网址：https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

7.7.2 数学库

- cuBLAS：cuBLAS 库提供了基本线性代数子程序（BLAS，Basic Linear Algebra Subprograms）的 GPU 加速实现。cuBLAS 利用针对 NVIDIA GPU 高度优化的行业标准 BLAS API 加速 AI 和 HPC 应用。cuBLAS 库包含用于分批操作、跨多个 GPU 执行以及混合和低精度执行的扩展。
- cuTENSOR：cuTENSOR 库是第一个由 GPU 加速的张量线性代数库，提供张量收缩、还原和元素运算。cuTENSOR 用于加速深度学习训练和推理、计算机视觉、量子化学和计算物理等领域的应用。
- cuSPARSE：cuSPARSE 库为稀疏矩阵提供了 GPU 加速的基本线性代数子程序，其功能可用于建立 GPU 加速的求解器。cuSPARSE 被从事机器学习、计算流体力学、地震勘探和计算科学等应用的工程师和科学家广泛使用。
- cuSOLVER：cuSOLVER 库提供了针对 NVIDIA GPU 高度优化的密集和稀疏因式分解、线性求解器和 eigensolvers。cuSOLVER 用于加速科学计算和数据科学等不同领域的应用，并拥有针对混合精度张量加速和跨多个 GPU 执行的扩展。
- cuFFT：cuFFT 库提供了针对 NVIDIA GPU 高度优化的快速傅里叶变换（FFT，Fast Fourier Transform）实现。cuFFT 被用于建立跨学科的商业和研究应用，如深度学习、计算机视觉、计算物理、分子动力学、量子化学以及地震和医学成像，并具有跨多个 GPU 执行的扩展。
- cuRAND：cuRAND 库是一个随机数发生器的 GPU 设备端实现。

7.7.3 常用工具

- CUDA-GDB：用于调试 CUDA 应用程序的 NVIDIA 工具软件。
- Nsight Compute：NVIDIA Nsight Compute 是面向 CUDA 应用程序的下一代交互式内核分析器。它通过一个用户界面和命令行工具提供详细的性能指标和 API 调试。
- Nsight System：NVIDIA Nsight System 是一款全系统性能分析工具，旨在实现应用程序算法的可视化。有助于识别优化和调整机会，以便在 CPU 和 GPU 上高效扩展应用程序。

NVIDIA HPC SDK 高性能计算工具包的完整介绍请参考[6]。

[6] NVIDIA HPC SDK 网址：https://developer.nvidia.com/hpc-sdk

CHAPTER 8

MPI 并行程序编译及运行

8.1 简介

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` ，且具有不同版本的组合。

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`。

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

8.2 MPI 并行程序的编译

8.2.1 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。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，该部分主要介绍其不同的参数设置等。

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[1] 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。

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)

[1] 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

  - 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
	匹配在括号中的一个字符
	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。

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`

：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`

8.2.2 Open MPI 库

Open MPI[2] 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 `/opt/opnempi` 目录在下。

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`[3]

`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
  ```
- 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：`\ [-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。

[2] 主页：http://www.open-mpi.org/
[3] 注意为 mpifort，而不是 Intel MPI 的 mpifort

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

8.2.3 Intel MPI 库

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[4] 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。

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。

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

表 1: 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
mpicc	gcc, cc	C	32/64 bit
mpicxx	g++	C/C++	32/64 bit
mpifc	gfortran	Fortran77*/Fortran 95*	32/64 bit
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
mpiicc	icc	C	32/64 bit
mpiicpc	icpc	C++	32/64 bit
mpiifort	ifort	Fortran77/Fortran 95	32/64 bit

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。

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：
    - -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。

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。

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`

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。

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。

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。

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。

8.2.4 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。

8.3 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。

CHAPTER 9

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。

9.1 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。

9.2 基本启动方式 [gdbstart]

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
```

9.2.1 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[1] 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

[1] 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。

9.2.2 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`

9.3 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>`

9.4 准备所需要调试的程序

9.4.1 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。

9.4.2 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。

9.4.3 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。

9.4.4 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。

9.4.5 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!");
    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
```

9.5 开始调试程序

启动调试：`gdb helloworld`。更多启动方式参见 *[gdbstart]*。

9.5.1 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。

9.5.2 运行程序

在命令行中输入 `run`，将开始运行程序。

9.5.3 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。

9.5.4 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。

9.5.5 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。

9.5.6 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。

9.5.7 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。

9.5.8 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`

9.6 传递命令给调试器

9.6.1 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。

9.6.2 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

9.7 调试并行程序

9.7.1 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。

9.7.2 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。

CHAPTER 10

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。

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。在 BASH 下可以通过运行 `module load` 选择 Intel 编译器时设置，或者在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
```

10.1 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。

10.2 Intel MKL 目录内容

Intel MKL 的主要目录内容见下表。

表 1: 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 Intel64 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[3] 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
<mk _dir>/include/mic/lp64l	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 Fortran90 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 MPIFFTW(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

[3] ILP64 接口和 LP64 接口的区别参见 *4.3.3* 使用接口库链接。

10.3 链接 Intel MKL

10.3.1 快速入门

利用-mkl 编译器参数

Intel Composer XE 编译器支持采用-mkl[4] 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。

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 *4.3.2* 动态选择接口和线程层部分。

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
Intel64 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 *4.3.2* 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

[4] 是-mkl，不是-lmkl，其它编译器未必支持此-mkl 选项。

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。

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，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。

10.3.2 链接举例

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
```

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
```

- 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
```

10.3.3 链接细节

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。

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。

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。

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。

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 IntelMKL 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 或库 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	

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。

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。

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）。

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。

10.4 性能优化等

请参见 Intel MKL 官方手册。

CHAPTER 11

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[1]，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[2]。
  - 源代码编译时经常用到的编译命令为 `make`，编译配置文件为，请查看 `make` 命令用法及文件说明。

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。

11.1 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
- 解压缩：`tar xvf gaussian09.tar.gz`[3]
- 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
```

[1] Gaussian 主页：http://www.gaussian.com/
[2] VASP 主页：http://www.vasp.at/
[3] 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。

- 刷新环境设置：`. ~/.bashrc` 或重新登录下。

11.2 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，而是提供了几个针对不同系统和编译器的 makefile 模板，可以复制成
  - 其它程序也许需要`./configure` 生成所需要的，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：

    ```
    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
- 修改文件配置，设定编译环境等：
  - 对做如下修改：
    - 设定编译 Fortran 的编译器命令为 Intel Fortran 编译器命令：FC=ifort
  - 对做如下修改：

    - 设定 BLAS 库使用 Intel MKL 中的 BLAS：BLAS=-mkl[4]
    - 打开 FFT3D 支持：去掉 FFT3D = fft3dfurth.o fft3dlib.o 前的 #[5]
    - 设定 MPI Fortran 编译器为 Intel MPI 编译器：FC=mpiifort
- 编译：`make`
  - 先在目录中执行 `make`
  - 如未出错，则再在目录中执行 `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
```

[4] 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。
[5] 在 Makefile 中 # 表示注释

CHAPTER 12

Slurm 作业调度系统

12.1 简介

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。

该文档的针对的 Slurm 版本为 19.05.5（瀚海 20 超级计算系统使用）、22.05.3（瀚海 22 超级计算系统使用）。

12.2 基本概念

12.2.1 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。

12.2.2 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。

12.2.3 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。

12.2.4 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。

12.3 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]
```

12.3.1 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。

12.3.2 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。

12.4 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)
```

12.4.1 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。
- QOSUsageThreshold：所需的 QOS 阈值被违反。
- WaitingForScheduling：等待被调度中。

12.4.2 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）。

12.5 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
```

12.5.1 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。

12.6 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
```

12.6.1 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。

12.7 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=gnode01
   BatchHost=gnode01
   NumNodes=1 NumCPUs=1 NumTasks=1 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   TRES=cpu=1,node=1,billing=1
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=1 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
```

(续下页)

(接上页)

```
   StdOut=/home/nic/hmli/job-77.log
   Power=
```

12.7.1 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。

12.8 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。

12.9 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 " -e: show error log.\n -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
```

(续下页)

(接上页)

```
    echo "Job $JOBID has no $STD file or you have no authority to access."
fi
```

12.10 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。

12.10.1 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。
  - 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。

12.10.2 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、…。

12.11 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...]`

12.11.1 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。

12.11.2 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）。

12.11.3 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
```

12.11.4 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
wait
```

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"
```

(续下页)

(接上页)

```
        ;;
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 &
wait

 -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
```

12.12 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。

1. 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 *9.1*。

12.12.1 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。

12.12.2 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。

12.12.3 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
```

12.12.4 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
```

12.12.5 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
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
```

12.12.6 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
wait
```

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。

12.12.7 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，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。

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/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
```

(续下页)

(接上页)

```
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
```

12.13 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 &`。

12.13.1 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
```

(续下页)

(接上页)

```
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
```

(续下页)

(接上页)

```
    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
```

12.13.2 主要选项

常见主要选项参见 *10.1*。

12.13.3 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。

12.13.4 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”。

12.13.5 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
```

12.14 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。

12.14.1 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。

12.14.2 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。

12.14.3 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
```

12.15 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>`

12.15.1 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。

12.15.2 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。

12.15.3 例子

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

12.16 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。

12.17 其它常用作业管理命令

12.17.1 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
```

12.17.2 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
```

12.17.3 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
```

12.17.4 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`

12.17.5 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
```

12.17.6 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
```

12.17.7 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
```

12.17.8 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
```

CHAPTER 13

LSF 作业调度系统

曙光 TC4600 百万亿次超级计算系统利用 IBM Spectrum LSF 10.1.0 进行资源和作业调度管理，所有需要运行的作业均必须通过作业提交命令 `bsub` 提交，提交后可利用相关命令查询作业状态等。为了利用 `bsub` 提交作业，需要在 `bsub` 中指定各选项和需要执行的程序。注意：

- 不要在登录节点 (tc4600) 上不通过作业调度管理系统直接运行作业（编译等日常操作除外），以免影响其余用户的正常使用。
- 如果不通过作业调度管理系统直接在计算节点上运行将会被监护进程直接杀掉。

13.1 作业运行的条件

作业提交后需要一段时间等待作业调度系统调度运行，一般为先提交的先运行，并且作业运行需要满足多个基本条件：

- 系统有空闲资源，满足程序运行需要。可以利用 `bhosts` 命令查看，ok 状态的才可以接受作业运行。
- 用户作业没有超过系统设置的允许用户运行的作业数，可以利用 `busers` 命令查看。
- 用户作业没有超过所使用的作业队列的允许作业核数的限制，可以利用 `bqueues` 命令查看。
- 用户作业没有被挂起等。利用 `bjobs` 命令查看。
- 作业调度管理系统工作正常。

当系统作业繁忙时，如果提交需要核数很多的作业，也许需要长时间才可以运行甚至根本无法获取到足够资源来运行，请考虑选择合适的并行规模。

作业如不运行，请先请运行 `bjobs -l JobID` 或 `bjobs -p JobID` 查看输出信息中 PENDING REASONS 部分及提交时设置的参数等，并结合运行上述几个命令查看原因。

如果上述条件都符合，也许作业调度管理系统存在问题，请与超算中心工作人员联系处理。

13.2 查看队列情况：bqueues

用户在使用时，首先需要了解哪些队列可以使用，利用 `bqueues` 可以查看现有队列信息。

具体队列会根据需要更改，请注意登录系统后的提示，或运行 `bqueues -l` 查看。

```
bqueues
```

将输出：

```
QUEUE_NAME PRIO STATUS       MAX  JL/U  JL/P JL/H NJOBS  PEND   RUN  SUSP
long         59  Open:Active  1200 1200    -    -     0     0     0     0
normal       58  Open:Inact   1440  192    -    -   360     0   360     0
small        57  Closed:Active 720  120    -    -  1976  1476   500     0
```

其中，主要列的含义为：

- QUEUE_NAME：队列名
- PRIO：优先级，数字越大优先级越高
- STATUS：状态
  - Open：队列开放，可以接受提交新作业
  - Active：队列已激活，队列中未开始运行的作业可以开始运行
  - Closed：队列已关闭，不接受提交新作业
  - Inact：队列未激活，可接受提交新作业，但队列中的等待运行的作业不会开始运行
- MAX：队列对应的最大作业槽数（Job Slot，一般与 CPU 核数一致，以下通称 CPU 核数），-表示无限
- JL/U：单个用户同时可以使用的 CPU 核数
- JL/P：每个处理器可以接受的 CPU 核数
- JL/H：每个节点可以接受的 CPU 核数
- NJOBS：排队、运行和被挂起的总作业所占 CPU 核数
- PEND：排队中的作业所需 CPU 核数
- RUN：运行中的作业所占 CPU 核数
- SUSP：被挂起的作业所占 CPU 核数
- RSV：为排队作业预留的 CPU 核数

13.2.1 查看队列详细情况：bqueues -l

队列策略也许会调整，利用 `bqueues -l [队列名]` 查看各队列的详细情况：

```
QUEUE: small
 -- Maximum number of job slots that each user can use in this queue is 720.
 on nodes: node1~node67

PARAMETERS/STATISTICS
PRIO NICE STATUS          MAX JL/U JL/P JL/H NJOBS  PEND   RUN SSUSP USUSP  RSV
 57   20  Open:Active     720  120    -    -  1964  1464   500     0     0    0
Interval for a host to accept two jobs is 0 seconds
```

(续下页)

(接上页)

```
 PROCLIMIT
 48

SCHEDULING PARAMETERS
           r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp    mem
 loadSched   -     -     -     -       -     -    -     -     -      -      -
 loadStop    -     -     -     -       -     -    -     -     -      -      -

          adapter_windows     poe nrt_windows
 loadSched               -       -           -
 loadStop                -       -           -

SCHEDULING POLICIES:  EXCLUSIVE

USERS: paiduigroup/ scc_group/
HOSTS:  node1 node2 node3 node4 node5 node6 node7 node8 node9 node10 node11 node12
RES_REQ:  span[ptile=24]
```

- QUEUE：队列名，跟着的下一行是描述
- PRIO：优先值，越大越优先
- NICE：作业运行时的 nice 值，即作业运行时的操作系统调度优先值，从-20 到 19，越小越优先
- RUNLIMIT：作业单 CPU 运行时间限制，以系统中的某个节点为基准，如为 1440.0 min 则表示可以运行一天（60*24）
- CPULIMIT：单个作业运行时间限制，以系统中的某个节点 E5410 作为参考，运行机时（核数 * 墙上时间）为 345600.0 CPU 分钟，即如用 8 CPU 核计算，允许运行 30 天
- PROCLIMIT：单个作业核数限制，`2 4 8`，表示使用此队列时，最少使用 2 个核，最多使用 8 核，如提交时没用-n 指定具体核数，则使用默认 4 核
- PROCESSLIMIT：单个作业最大核数限制，为 8
- MEMLIMIT：每个进程能使用的内存数，默认以 KB 为单位
- THREADLIMIT：作业最大线程数
- USERS：有权使用的用户
- HOSTS：对应的节点

13.3 查看各节点的运行情况：lsload

利用 `lsload` 命令可查看当前各节点的运行情况，例如：

`lsload`

```
HOST_NAME      status  r15s   r1m  r15m    ut   pg   ls  it   tmp   swp   mem
node1          ok       0.0   0.0   0.0   0%   0.5   0   25  227G   32G   62G
node2          locku    0.1   0.0   0.0   0%   0.4   0   60  227G   32G   62G
node4          unavail   -     -     -     -     -    -    -     -     -     -
```

- HOST_NAME：节点名。
- status：状态。
- status 列：

  - ok：表示可以接受新作业，只有这种状态可以接受新作业
  - closed：表示系统在运行，但已被调度系统关闭，不接受新作业
  - locku：表示在进行排他性运行
  - busy：表示负载超过限定
  - unavail、-ok：作业调度系统服务有问题
- r15s、r1m、r15m 列：分别表示 15 秒、1 分钟、15 分钟平均负载
- ut：利用率
- tmp：目录大小
- swp：swp 虚拟内存大小
- mem：内存大小
- io：硬盘读写（-l 选项时才出现）

查看 node2 节点：`lsload node2`

13.4 查看各节点的空闲情况：bhosts

利用 `bhosts` 命令可查看当前各节点的空闲情况，例如：

`bhosts`

```
HOST_NAME  STATUS  JL/U  MAX  NJOBS  RUN  SSUSP  USUSP  RSV
node12          closed   -    16       2    2      0      0    0
node10          ok       -    16       2    1      0      0    0
node14          ok       -    16       2    1      0      0    0
```

- HOST_NAME：节点名。
- STATUS：状态。
  - ok：表示可以接收新作业，只有这种状态可以接受新作业
  - closed：表示已被作业占满，不接受新作业
  - unavail 和 unreach：系统停机或作业调度系统服务有问题
- JL/U：允许每个用户的作业核数。-表示未限制。
- MAX：允许最大作业核数。
- NJOBS：当前运行的作业数。
- RUN：当前运行作业占据的核数。
- SSUSP：被系统挂起的作业占据的核数。
- USUSP：被用户挂起的作业占据的核数。
- RSV：预留的核数。

查看 node1 节点：`bhosts node1`

`bhosts -l` 会显示节点详细信息，其中 slots 表示目前最大可以接受作业槽数（默认一般与 CPU 核数一致）即使有节点状态为 ok 状态，也不一定表示您的作业可以运行，具体运行条件参见 *[run]* 作业运行条件。

13.5 查看用户信息：busers

利用 busers 可以查看用户信息，例如：

```
busers hmli
```

```
USER/GROUP JL/P MAX NJOBS PEND RUN SSUSP USUSP RSV
hmli           -     320     40     32    8      0     0    0
```

其中：

- USER/GROUP：用户或组名。
- JL/U：允许每个用户的作业核数。-表示未限制。
- MAX：允许最大作业核数。
- NJOBS：当前运行的作业数。
- PEND：当前挂起作业占据的核数。
- RUN：当前运行作业占据的核数。
- SSUSP：被系统挂起的作业占据的核数。
- USUSP：被用户挂起的作业占据的核数。
- RSV：预留的核数。

13.6 提交作业：bsub

用户需要利用 bsub 提交作业，其基本格式为 bsub [options] command [arguments]。其中 options 设置队列、CPU 核数等的选项，必须在 command 之前，否则将作为 command 的参数；arguments 为设置作业的可执行程序本身所需要的参数，必须在 command 之后，否则将作为设置队列等的选项。下面将给出常用的几种提交方式。

注意：

- 作业提交后，应经常检查一下作业的 CPU、内存等利用率，判断实际运行效率：
  - 可以 ssh 到对应运行作业的节点运行 top 命令；
  - 查看 Ganglia 系统监控：http://scc.ustc.edu.cn/ganglia。
- 请不要 ssh 到节点后直接运行作业，以免影响作业调度系统分配到此节点的作业。

13.6.1 提交到特定队列：bsub -q

利用-q 选项可以指定提交到哪个队列，，请注意参看登录后的提示或运行 bqueues -l 命令查看，现有的队列为：

- normal：所需要的 CPU 核数大于 1 个且不超过 16 个时
- long：所需要的 CPU 核数超过 32 个时
- serial：所需要的 CPU 核数为一个时

比如想提交到 normal 队列使用 2 个 CPU 核运行程序 executable1，可以：

```
bsub -q normal -n 2 executable1
```

如果提交成功，将显示类似下面的输出：

Job <79722> is submitted to queue <normal>.

其中 79722 为此作业的作业号，以后可利用此作业号来进行查询及终止等操作。

如提交不成功，则显示相应提示。

13.6.2 提交串行作业：bsub -n 1

我校超算系统鼓励运行并行作业，允许运行的串行作业资源较少，有些系统不允许运行串行作业。

运行串行作业，请使用支持串行队列的队列（假如 serial 队列支持串行），比如：

```
bsub -q serial -n 1 executable-serial
```

13.6.3 指明所需要的 CPU 核数：bsub -n

利用`-n min\_proc[,max\_proc]`选项指定所需要的 CPU 核数（一般来说核数和进程数一致），比如下面指定利用 24 个 CPU 核（由`-n 24`指定）运行 MPI（由 `mpijob` 指明为运行 MPI 程序）程序：

```
bsub -q small -n 24 mpijob executable-mpi1
```

最少采用 24 最大采用 72 CPU 核数运行：`bsub -n 24,72`

当作业调度尝试开始运行时：

- 如空闲资源满足 72 CPU 核，那么将采用 72 CPU 核运行；
- 如空闲资源不满足 72 CPU 核，但满足 24 CPU 核，则用 24 CPU 核运行。

由于每个节点的 CPU 核数为 24，建议单个作业所使用的核数最好为 24 或 12 的整数倍，以尽量保证自己的程序占据独立的节点或半个节点，尽量避免相互影响。以上仅仅是建议，具体申请核数应考虑作业实际情况。即使同一个计算软件，在计算不同条件时，也有可能不一样，请务必仔细研究自己所使用的软件。

13.6.4 提交到特定节点：bsub -m “host_name”

利用`-m`选项可以指定作业在特定节点上运行，如：

```
bsub -m ``node1 node3''
```

如非必要，建议不要添加此选项，以免导致作业无法及时运行。

13.6.5 提交 MPI 作业：bsub -n NUM mpijob

如果需要运行 MPI 作业，需要利用 `mpijob` 调用 MPI 可执行程序，并用-n 选项指定所需的 CPU 核数，比如下面指定利用 64 颗 CPU 核运行 MPI 程序 executable-mpi1：

```
bsub -q long -n 64 mpijob executable-mpi1
```

注意：

- `mpijob` 命令已经封装了 Intel MPI 和 Open MPI 的运行 MPI 作业的脚本，用户一般无需再特别按照 Intel MPI 和 Open MPI 的原始 `mpirun`、`mpiexec` 等命令使用。
- 提交作业时在 `mpijob` 之前的参数将传递给 LSF，之后的参数将传递给原始的 `mpirun` 或 `mpiexec` 等。
- 如用户对 LSF 和 Intel MPI、Open MPI 等不熟悉，请勿擅自添加其它参数，如有需要请与超算中心联系。

- 如您了解所使用的 MPI 环境与 LSF 的配合，也可不用 `mpijob` 提交，如直接用 `mpirun` 等命令运行。

13.6.6 指明需要某种资源作业提交：bsub -R

`-R ``res_req'' [-R ``res_req'' ...]` 可以使得作业在需要满足某种条件的节点上运行，如：

- `-R ``span[hosts=1]''`：指定需要在同一个节点内运行；
- `-R ``span[ptile=8]''`：指定需要在每一个节点内运行多少核，如 `bsub -n 16 -R ``span[ptile=8]''`
  则会分给 2 个节点，每个节点 8 核；
- `-R ``1*{mem>5000} + 9*{mem>1000}''`：一个 CPU 核需要至少 5GB 内存，另外 9 个每个至少需要 1GB 内存。

13.6.7 提交 OpenMP 等共享内存作业：bsub -R "span[hosts=1]" OMP_NUM_THREADS=

由于只能在同一个节点内部运行 OpenMP 共享内存的作业，如 Gaussian 程序，此时需要添加利用`-R ``span[hosts=1]''`参数指定使用一个节点，并用 OMP_NUM_THREADS 设定指定的线程数，一般应与申请的核数一致[1]：

指定利用 8 CPU 核运行 OpenMP 程序：

```
bsub -q normal -n 8 -R ``span[hosts=1]'' OMP_NUM_THREADS=8 executable-omp1
```

13.6.8 MPI 和 OpenMP 共享内存混合并行作业

需要针对需求利用 LSF 环境变量特殊处理，如果自己不清楚怎么处理，请联系管理人员。

13.6.9 给作业起个名字：bsub -P project_name

提交时可以利用`-P` 选项给作业起个名字方便查看，如：

```
bsub -P VASPJOB
```

13.6.10 运行排他性作业：bsub -x

如需要独占节点运行，此时需要添加-x 选项：

```
bsub -x -q normal -n 8 executable-omp1
```

注意：

- 排他性运行在运行期间，不允许其余的作业提交到运行此作业的节点，并且只有在某节点没有任何其余的作业在运行时才会提交到此节点上运行；
- 如果不需要采用排他性运行，请不要使用此选项，否则将导致作业必须等待完全空闲的节点才会运行，也许将增加等待时间；
- 另外使用排他性运行时，哪怕只使用某节点内的一个 CPU 核，也将按照此节点内的所有 CPU 核数进行机时计算。

[1] 有些程序可以在输入文件中设置，那么可以不填加 OMP_NUM_THREADS

13.6.11 指明输出、输出文件运行：bsub -i -o -e

作业的正常屏幕输入文件（指的是类似方式的文件）、正常屏幕输出到的文件和错误屏幕输出的文件可以利用-i、-o 和-e 选项来分别指定，运行后可以通过查看指定的这些输出文件来查看运行状态，文件名可利用%J 与作业号挂钩。比如指定 executable1 的输入、正常和错误屏幕输出文件分别为:、和:

```
bsub -i executable1.input -o executable1-\%J.log -e executable1-\%J.err
executable1
```

-o 和-e 及变种-oo 和-eo:

- -o：如日志原文件存在，正常屏幕输出将追加到原文件后
- -oo：如日志原文件存在，正常屏幕输出将覆盖原文件
- -e：如日志原文件存在，出错时屏幕输出将追加到原文件后
- -eo：如日志原文件存在，出错时屏幕输出将覆盖原文件

建议打开-o 和-e 参数，以便查看作业为什么出问题等。如果需要管理人员协助解决，请告知这些输出，以及运行目录，怎么运行的等，以便管理人员能获取足够的信息及时处理。

13.6.12 指明输出目录提交：bsub -outdir output_directory

默认情况下，屏幕输出等存放在提交作业时的目录下，如想将其放到其它目录可以采用 `bsub -outdir output\_directory` 方式，如:

bsub -outdir %U/%J_%I myprog

结合以下常用变量，可以方便结合作业号等设置输出目录及日志文件名等:

- %J：作业号
- %JG：作业组
- %I：作业组中的索引
- %EJ：执行作业号
- %EI：作业组中的执行作业索引
- %P：作业名
- %U：用户名
- %G：用户组名

13.6.13 交互式运行作业：bsub -I

如果需要运行交互式的作业（如在运行期间需要手动输入参数或利用调试器手动调试程序等需要进行交互时），需要结合-I 参数。建议只是在调试期间使用，一般作业还是尽量不要使用此选项，类似选项还有-Ip 和-Is:

```
bsub -I executable1
```

13.6.14 满足依赖关系运行作业：bsub -w

利用-w 选项可以使得新提交的作业在满足一定条件时才运行，比如与其它作业的关联：

- done(job_ID | “job_name” …)：作业结束时状态为 DONE 时运行
- ended(job_ID | “job_name”)：作业结束时状态为 DONE 或 EXIT 时运行
- exit(job_ID | “job_name” [,[operator] exit_code])：作业结束时状态为 EXIT，且退出代码满足一定条件时运行
- external(job_ID | “job_name” , “status_text”)：作业状态变为某状态时运行，如变为 SUSP
- 支持的条件之间的条件表达式：&&（和）、||（或）、!（否）
- 支持的条件内的条件算子：>、>=、<、<=、==、!=

如：`bsub -w ``done(1456)''`

13.6.15 指定时间运行：bsub -b time

利用`-b [[year:][month:]day:]hour:minute` 可以使得新提交的作业在特定时间运行，系统会预留资源给此作业，如果在时间到达时所需资源满足则会运行，不满足则继续等待。如：

```
bsub -b 2016:01:09:09:20
```

13.6.16 指定运行时长：bsub -W time

利用`-W [hour:]minute` 可以使得提交的作业在运行超过设定时长后终止，如：

```
bsub -W 1:30
```

13.6.17 在运行前执行特定命令：bsub -E “pre_command”

利用`-E ``pre_exec_command [arguments ...]''`可以使得作业在运行前，在所分配的节点上运行特定命令。

如利用 modinput.sh 此修改程序输入参数：`bsub -E `./modinput.sh 10’’ -n 2 exec1‘

13.6.18 在运行后执行特定命令：bsub -Ep “post_command”

利用`-Ep ``post_exec_command [arguments ...]''`可以使得作业在运行结束时，在所分配的节点上运行特定命令。

如利用此删除 core 文件：`bsub -Ep ``bin/rm -f core.*'' -n 2 exec1`

13.6.19 LSF 作业脚本

如果作业比较复杂，还需要设置环境变量，做其它处理等，可用以在 LSF 脚本中设置队列等参数方式提交，如

```
#!/bin/sh
#BSUB -q long
#BSUB -o %J.log -e %J.err
#BSUB -n 64
source my.sh
mpijob ./mympi-prog1
cd newworkdir
mpijob ./mympi-prog2
```

注意，采用此方式时：

- 不得以直接./my_script.lsf 等常规脚本运行方式运行。
- 需要传递给 bsub 命令运行：bsub < my_script.lsf。
- 如果 bsub 后面更-q 等 LSF 参数，将会覆盖掉 LSF 脚本中的设置。
- 一般用户，没必要写此类脚本，直接通过命令行传递 LSF 参数即可。
- 对于当前设置满足不了作业需求，且用户比较了解 LSF 中的各规定，对 shell 脚本编写比较在行，那么用户完全可自己编写脚本提交作业，比如提交特殊需求的 MPI 与 OpenMP 结合的作业。

LSF 作业脚本主要有以下常见变量比较常用，在作业运行后，这些变量存储对应的作业信息，具体的请参看 LSF 官方手册：

- LS_JOBPID：作业进程号
- LSB_HOSTS：存储系统分配的节点名
- LSB_JOBFILENAME：作业脚本文件名
- LSB_JOBID：作业号
- LSB_QUEUE：作业队列
- LSB_JOBPGIDS：作业进程组号组
- LSB_JOBPIDS：作业进程号组

13.7 终止作业：bkill

利用 bkill 命令可以终止某个运行中或者排队中的作业，比如：

```
bkill 79722
```

运行成功后，将显示类似下面的输出：

```
Job <79722> is being terminated
```

13.8 挂起作业：bstop

利用 bstop 命令可临时挂起某个作业以让别的作业先运行，例如：

```
bstop 79727
```

运行成功后，将显示类似下面的输出：

```
Job <79727> is being stopped.
```

此命令可以将排在队列前面的作业临时挂起，以让后面的作业先运行。虽然也可以作用于运行中的作业，但并不会因为此作业被挂起而允许其余作业占用此作业所占用的 CPU 运行，实际资源不会释放，因此建议不要随便对运行中的作业进行挂起操作，如果运行中的作业不再想继续运行，请用 bkill 终止。

13.9 继续运行被挂起的作业：bresume

利用 bresume 命令可继续运行某个挂起某个作业，例如：

```
bresume 79727
```

运行成功后，将显示类似下面的输出：

```
Job <79727> is being resumed.
```

13.10 设置作业最先运行：btop

利用 btop 命令可最先运行排队中的某个作业，例如：

```
btop 79727
```

运行成功后，将显示类似下面的输出：

```
Job <79727> has been moved to position 1 from top.
```

13.11 设置作业最后运行：bbot

利用 bbot 命令可设定最后运行排队中的某个作业，例如：

```
bbot 79727
```

运行成功后，将显示类似下面的输出：

```
Job <79727> has been moved to position 1 from bottom.
```

13.12 修改排队中的作业选项：bmod

利用 bmod 命令可修改排队中的某个作业的选项，比如想将排队中的运行作业号为 79727 的作业的执行命令修改为 executable2 并且换到 long 队列，可以：

```
bmod -Z executable2 -q long 79727
```

```
Parameters of job <79727> are being changed.
```

13.13 查看作业的排队和运行情况：bjobs

利用 bjobs 可以查看作业的运行情况，比如有哪些作业在运行，哪些在排队，某个作业运行在哪个节点上，以及为什么没有运行等，例如：

```
bjobs
```

```
JOBID USER  STAT  QUEUE     FROM_HOST   EXEC_HOST JOB_NAME   SUBMIT_TIME
79726 hmli  RUN   normal    tc4600      2*node31  *executab1 Mar 12 19:20
                                        1*node18
                                        1*node4
79727 hmli  PEND  long      tc4600                *executab2 Mar 12 19:20
```

上面显示作业 79726 在运行，分别在 node31、node18 和 node4 上运行 2、1、1 个进程；而作业 79727 处于排队中尚未运行，查看未运行的原因可以利用：

```
bjobs -l 79727
```

```
Job Id <79727>, tc4600 <hmli>, Project <default>, Status <PEND>,
Queue <long> , Command <executab2>
Sun Mar 12 14:15:07: Submitted from host <tc4600>,
CWD <$HOME>, Requested Resources <type==any && swp>35>;
PENDING REASONS:
SCHEDULING PARAMETERS:
            r15s r1m r15m  ut pg   io  ls   it   tmp   swp   mem
loadSched   -    0.7 1.0   - 4.0   -   -    -    -     -     -
loadStop    -    1.5 2.5   - 8.0   -   -    -    -     -     -
```

以下为另外几个常用参数：

- -u username：查看某用户的作业，如 username 为 all，则查看所有用户的作业。
- -q queuename：查看某队列上的作业。
- -m hostname：查看某节点上的作业。

13.14 查看作业负载：checkjob

利用/opt/bin/monitor/checkjob 作业号可以查看作业各节点负载等信息，例如：

```
/opt/bin/monitor/checkjob 799374
```

```
user: hmli, jobid: 799374
Note: cpu_load near cpu_num and swap_used smaller or zero is better.
node_name   cpu_num   cpu_load   memory(%)   swap_used(GB)
```

(续下页)

(接上页)

```
node491      144      109.03    23.00          0
node493      144      115.52    23.00          0
```

13.15 查看运行中作业的屏幕正常输出：bpeek

利用 bpeek 命令可查看运行中作业的屏幕正常输出，例如：

```
bpeek 79727
```

```
<< output from stdout >>
Energy: 3.0keV
Angles: 13.0, 0.0
```

如果在运行中用-o 和-e 分别指定了正常和错误屏幕输出，也可以通过直接查看指定的文件的内容来查看屏幕输出。

如果想连续查看某个作业的输出，请添加-f 参数。

CHAPTER 14

LaTeX pdf 版（不再更新）

- 瀚海 20 超级计算系统用户使用指南
- 曙光 TC4600 百万亿次超级计算系统用户使用指南

CHAPTER 15

联系方式

- 信箱：sccadmin@ustc.edu.cn
- 超算中心主页：http://scc.ustc.edu.cn