北京超级云计算中心 N32-H 分区

GPU 资源用户使用手册

2023 年 3 月

目录

1 系统资源简介

1.1 计算资源（节点配置）

登陆节点：主机名 paraai-n32-h-01-ccs-cli-1（8 核 CPU，30G 内存），用于提供用户登陆服务。请不要在登录节点直接运行作业（日常查看、编辑、编译等除外），以免影响其余用户的正常使用。

GPU 计算节点：主机命名规则：scx6000-scxafff，可参考 2.3 节内容申请 GPU 计算资源进行计算任务提交。

BSCC-N32-H 分区的 GPU 资源，每台服务器的配置如下：

Architecture：aarch64

OS：Kylin Linux Advanced Server release V10 (Sword)

CPU：2 * 鲲鹏-920@3.0GHz

GPU：4* NVIDIA A100　PCIe 40GB

可用内存：220GB

节点互联：4 * 100G RoCE 高速互联(RDMA 协议)

登录节点可以联网，计算节点也能联网。

BSCC-N32-H 的 gpu 队列配置为：每台机器配置 4 块型号为 NVIDIA Tesla A100-PCIe 40GB 显存的 GPU，每块 GPU 卡默认分配 32CPU 核及 55GB 内

存，即 GPU：CPU：内存配比为 1GPU 卡：32CPU 核：55GB 内存。如需更多 CPU 和内存，可以通过申请更多 GPU 卡方式获得。

1.2 存储资源（文件系统）

用户登录上后即在家目录（home），默认配额为 300GB。

如果数据量超出磁盘配额，可以联系客户经理增加配额。

2 应用及软件管理

该章节将对算例管理的全生命周期进行详细叙述，包括算例运行前的环境搭建、代码编译、算例提交、算例取消和查看算例状态。

2.1 使用 BSCC-N32-H 已部署的应用软件及开发环境

2.1.1 简介

“BSCC-N32-H”已部署多款应用软件及软件开发运行环境。

由于不同用户在“BSCC-N32-H”上可能需要使用不同的软件环境，配置不同的环境变量，软件之间可能会相互影响，因而在“BSCC-N32-H”上安装了 module 工具用于统一管理应用软件。module 工具主要用来帮助用户在使用软件前设置必要的环境变量。用户使用 module 加载相应版本的软件后，即可直接调用超算上已安装的软件。

2.1.2 基本命令

常用命令如下：

命令	功能	示例
module avail	查看可用的软件列表	
module load [modulesfile]	加载需要使用的软件	module load compilers/cuda/11.1
module show [modulesfile]	查看对应软件的环境（安装路径、库路径等）	module show compilers/cuda/11.1
module list	查看当前已加载的所有软件	
module unload [modulesfile]	移除使用 module 加载的软件环境	module unload compilers/cuda/11.1

module 其它用法，可使用 module --help 中查询。module 加载的软件环境只在当前登陆窗口有效，退出登陆后软件环境就会失效。用户如果需要经常使用一个软件，可以把 load 命令放在~/.bashrc 或者提交脚本里面。

2.1.3 已部署软件及开发环境列表

已安装软件开发运行环境包括但不限于下面所列：

软件名称	版本
Anaconda	2021.11
gcc	9.3
毕昇编译器	2.1.0、2.5.0

CUDA	11.1、11.3、11.6
其它软件	持续更新中

2.1.4 示例说明

用户需要使用 cuda11.1 环境

操作步骤:

- 执行 module avail 查看系统中可用软件，查询到 cuda11.1 的 module 环境名称为 compilers/cuda/11.1

- 执行 module load compilers/cuda/11.1 加载 cuda11.1 环境
- 执行 module list 查看已加载的环境

```
$ module list
Currently Loaded Modulefiles:
 1) compilers/cuda/11.1
```

2.2 软件安装/环境搭建

如果执行 module avail 没有查询到所需要的软件，可以上传软件安装包，在 home 目录下自定义安装所需软件。

软件安装基本流程：

1. 上传软件安装包。

2. 打开【SSH】命令行窗口，cd 到对应目录下，进行软件、工具、库等安装部署和算例运行环境搭建。

2.2.1通过 Anaconda 进行自定义 python 环境安装

Anaconda 是一个开源的 Python 发行版本，其包含了 conda、Python 等 180 多个科学包及其依赖项。支持 Linux, Mac, Windows 系统，利用 conda 工具/命令来进行包管理与环境管理，可以很方便地解决多版本 python 并存、切换以及各种第三方包安装问题。并且已经包含了 Python 和相关的配套工具。

加载 anaconda/2021.11 环境后，默认的 python 是 3.9.13（环境名称为 base）。

假设我们需要安装 python3.8，此时，我们需要做的操作如下：

1) 加载 anaconda/2021.11 环境

```
module load anaconda/2021.11
```

2) 创建名为 python38 的环境，指定 Python 版本是 3.8

```
conda create --name py38 python=3.8
```

注：

a）不用管是 3.8.x，conda 会为我们自动寻找 3.8.x 中的最新版本

b）conda 环境会默认安装到~/.conda 文件夹中，安装过程的安装包也会下载到此文件夹下，安装过程中会在~/.cache 目录下产生临时文件。

c）由于 N32-H 分区是 aarch64 架构，安装支持 GPU 的 torch 需要采用源码编译，目前提供 torch_1.11.0+cu113 的包（如果需要其他版本的 torch，联系客服人员处理），安装方法如下：

```
# 1.11.0 是 torch 版本，cu113 是 cuda/11.3.0，cp38 是 python 3.8。
pip install
/home/bingxing2/apps/package/pytorch/1.11.0+cu113_cp38/*.whl


# 上述 torch 安装完成之后，可在对应的目录下有 env.sh 中查看加载
cuda 信息。
cat /home/bingxing2/apps/package/pytorch/1.11.0+cu113_cp38/env.sh
# 输出：
#!/bin/bash
module load compilers/cuda/11.3
module load compilers/gcc/9.3.0
```

3) 查看已安装的环境

```
$ conda env list
```

```
# conda environments:
#
base                  *  /home/bingxing2/apps/anaconda
test                     /home/bingxing2/apps/anaconda/envs/test
tf2.5-py38               /home/bingxing2/apps/anaconda/envs/tf2.5-
py38
torch1.12.1-py38
/home/bingxing2/apps/anaconda/envs/torch1.12.1-py38
torch1.8-py37
/home/bingxing2/apps/anaconda/envs/torch1.8-py37
py38                     /home/bingxing2/scx6002/.conda/envs/py38
```

输出如下,*代表目前激活的环境

```
# conda environments:
#
# conda environments:
#
base                  *  /home/bingxing2/apps/anaconda
```

4) 安装好后，使用 activate 激活某个环境

```
[username@cli01 ~]$ source activate py38

(py38) [username@cli01 ~]$ which python

~/.conda/envs/py38/bin/python

(py38) [username@cli01 ~]$ python --version

Python 3.8.16
```

可以看到 Python 已切换到 3.8 的环境下。

5) 取消当前加载的环境，可执行：

```
$ source deactivate py38
```

6) 使用 conda 的包管理：

如需安装 pytorch 和 tensorflow 等需要 GPU 的应用不能直接从 PIP 源或 conda 源在线安装。可以联系我们获取安装包，或让我们协助安装。

7) 查看 conda 已经安装的 packages

```
$ conda list
```

最新版的 conda 是从 site-packages 文件夹中搜索已经安装的包，不依赖于 pip，因此可以显示出通过各种方式安装的包

注：如需客服协助安装、部署相关软件，可直接在您专属的微信服务群里@客服。

2.3 提交计算任务

本章节针对程序测试完成后的正式计算任务提交。推荐软件调试完毕后按照如下流程提交计算任务到计算节点进行计算。

编写提交脚本过程遇到困难时，可将程序测试能够成功运行的命令在群里发给客服，客服协助编写脚本。

使用本节方法提交计算任务后，调度系统会自动申请 GPU 资源，随后自动在计算节点执行用户所编辑的脚本内的命令，直到脚本执行结束作业自动退出（或者在作业运行时执行 scancel 命令取消作业后作业自动停止）。申请到资源至作业结束，这段时间按卡时进行计费（费用=卡数 x 时间（精确到秒）x 单价)。申请到资源到运行结束这段时间按卡时进行计费。

gpu 队列计费公式：费用=卡数 x 时间（精确到秒）x 单价。

每台机器配置 4 块型号为 NVIDIA Tesla A100-PCIe 40GB 显存的 GPU，每块 GPU 卡默认分配 32CPU 核及 55GB 内存，即: GPU\CPU\内存 配比为 1GPU 卡\32CPU 核\55GB 内存。提交作业必须按照该比例进行作业提交。

2.3.1单节点任务提交

作业提交命令：

sbatch --gpus=GPU 卡数 程序运行脚本

每个作业可以申请的 GPU 卡数范围为 1~4，不需要增加其它参数，每卡默认分配 32 核及 55GB 内存。

申请到资源后，程序必须按运行程序所需的参数指定进程数、线程数、卡数调用申请到的资源。注意：如果使用 srun 运行作业必须使用-n 参数指定进程数，或配合使用-c 参数指定每进程的线程数 （进程数 x 每进程线程数<申请的核数）。

作业提交命令示例：

```
$ sbatch --gpus=1 ./run.sh
```

执行此命令后即申请到 1GPU 卡、32CPU 核、55GB 内存资源。作业显示为 R（Runing）状态（parajobs 命令查看作业状态）后即开始执行 run.sh 脚本中的内容。

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/bash 等指定该脚本的解释程序，/bin/bash 可以变为 /bin/sh、/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。

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 run.sh 示例 1，python 程序运行脚本示例：

```
#!/bin/bash
#SBATCH --gpus=1
```

```
#参数在脚本中可以加上前缀“#SBATCH”指定，和在命令参数中指定功能一致，如果脚本中的参数和命令指定的参数冲突，则命令中指定的参数优先级更高。在此处指定后可以直接 sbatch ./run.sh 提交。
#加载环境，此处加载 anaconda 环境以及通过 anaconda 创建的名为 pytorch 的环境(pytorch 环境需要自己部署)
module load anaconda/2021.11
source activate pytorch

#python 程序运行，需在.py 文件指定调用 GPU，并设置合适的线程数，batch_size 大小等
python train.py
```

如果申请 2 卡 64 核 110GB 内存，则修改如下参数为，依此类推：

```
#sbatch --gpus=2 ./run.sh
```

- **脚本 run.sh 示例 2，gromacs 程序运行脚本示例：**

```
#!/bin/bash
#SBATCH --gpus=2
#加载环境，此处加载 gromacs 环境
module load gromacs/2022.5_hmpi1.2.1_cuda11.6_bs2.1.0

#运行 gromacs 命令
```

```
gmx_mpi grompp -f pme.mdp


mpirun -np 2 gmx_mpi mdrun -dlb yes -v -nsteps 10000 -noconfout -nb
gpu -pme gpu -pin on -ntomp 32
```

2.3.2 跨节点任务提交

2.3.2.1 跨节点任务提交参数解析

跨节点主要提交参数解析：

-N，–nodes=<node>：采用特定节点数运行作业，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。

--ntasks-per-node=<ntasks>：每个节点运行<ntasks>个进程，需与-N 配合使用。

-n <number> ，--ntasks=<number> ： 此作业申请<number>个进程数。如果不设置默认为 1。

-c <ncpus> ，--cpus-per-task=<ncpus> ： 每个进程需 ncpus 颗 CPU 核，一般运行 OpenMP 等多线程程序时需要设置。如果不设置默认为 1。

--gres=gpu:卡数 ： 每节点申请的 GPU 卡数，多机任务建议用此参数。

--qos=gpugpu ： 跨节点任务必须指定此参数，否则没有跨节点提交权限。

提交命令示例 1：

```
sbatch -N 2 --gres=gpu:4 --qos=gpugpu 脚本名
```

-N 2 申请 2 台机器

--gres=gpu:4 每台机器申请 4 卡。--gres=gpu:卡数代表每台机器申请多少卡，--gpus=卡数 代表总共申请多少卡，并且在多机场景--gpus=申请的卡数是随机分布的，不建议使用此参数。

--qos=gpugpu 申请开通跨节点权限后可以添加此参数进行跨节点任务提交。

CPU 默认每卡分配 32 核。

提交命令示例 2：

```
sbatch -N 2 --gres=gpu:4 --ntasks-per-node=128 --qos=gpugpu 脚本名
```

每节点申请 128 核（srun 时申请 128 进程）

提交命令示例 3：

```
sbatch -N 2 --gres=gpu:4 --ntasks-per-node=4 -c 32 --qos=gpugpu 脚本名
```

每节点申请 128 核（srun 时申请 4 进程,每进程 32 核）

2. 通信参数：跨节点通常使用 IB 卡配合 NCCL 通信。需要在脚本中添加如下参数启动 IB 通信

```
export NCCL_ALGO=Ring
export NCCL_MAX_NCHANNELS=16
export NCCL_MIN_NCHANNELS=16
export NCCL_DEBUG=INFO
export NCCL_TOPO_FILE=/home/bingxing2/apps/nccl/conf/dump.xml
```

```
export NCCL_IB_HCA=mlx5_0,mlx5_2
export NCCL_IB_GID_INDEX=3
```

openmpi 跨节点：

不需要添加额外参数。

Intelmpi 跨节点：

不需要添加额外参数。

2.3.2.2 python 跨节点任务提交示例

程序配置参数，以下是 torch 提供的分布式跨节点启动脚本，以此为例进行说明（不同程序下面参数会有差异，此处仅供参考，脚本需根据实际情况进行）：

--nnodes：节点数量，跟上述作业参数 -N 一致。

--node_rank：节点的角色。主节点的角色为 0，其他节点角色依次为 1、2、3 等...

--nproc_per_node：每个节点运行的进程数量。通常跟申请的 GPU 卡数一致。

--master_addr：主节点的地址

--master_port：主节点的端口

在 torch(1.11.0) 之前的版本中（不包括 torch1.11.0），跨节点的实现主要是通过 torch.distributed.launch 的方式启动，参考如下：

```
python -m torch.distributed.launch \
        --nnodes=2 \
```

```
        --node_rank=0 \
        --nproc_per_node=4 \
        --master_addr="gpu01" \
        --master_port="29501" \
        train.py \
        --epochs 3 \
        --data coco.yaml \
        --cfg yolov5s.yaml \
        --weights '' \
        --batch-size 768 \
        --workers 64 \
        --device 0,1,2,3 >> train_rank0_"${SLURM_JOB_ID}".log 2>&1
```

新版本的 torch (1.11.0)，启动方式改成使用 torchrun，参考如下：

```
torchrun \
      --nnodes=2 \
      --node_rank=0 \
      --nproc_per_node=4 \
      --master_addr="gpu01" \
      --master_port="29501" \
      --max_restarts=3 \
      train.py --epochs 30 \
      --data coco128.yaml \
```

```
    --cfg yolov5s.yaml \
    --weights '' \
    --batch-size "${BATCH_SIZE}" \
    --device 0,1,2,3 >> "${OUTPUT_LOG}" 2>&1
```

提交作业示例：

以下是两个节点运行的示例，提供参考，如有困难可寻求客服帮助。

1. 先准备作业的运行脚本，如下 run.sh

```
#!/bin/bash
module load anaconda/2021.11
module load compilers/cuda/11.3
module load cudnn/8.4.0.27_cuda11.x
source activate yolov5

### 启用 IB 通信
export NCCL_ALGO=Ring
export NCCL_MAX_NCHANNELS=16
export NCCL_MIN_NCHANNELS=16
export NCCL_DEBUG=INFO
export NCCL_TOPO_FILE=/home/bingxing2/apps/nccl/conf/dump.xml
export NCCL_IB_HCA=mlx5_0,mlx5_2
export NCCL_IB_GID_INDEX=3
```

```
export NCCL_IB_TIMEOUT=23
export NCCL_IB_RETRY_CNT=7
### 获取每个节点的 hostname
for i in `scontrol show hostnames`
do
   let k=k+1
   host[$k]=$i
   echo ${host[$k]}
done

### 主节点运行
python -m torch.distributed.launch \
            --nnodes=2 \
            --node_rank=0 \
            --nproc_per_node=4 \
            --master_addr="${host[1]}" \
            --master_port="29501" \
            train.py --epochs 3 --data coco128.yaml --cfg yolov5s.yaml --
weights " --batch-size 128 --workers 16 --device 0,1,2,3 >>
train_rank0_${SLURM_JOB_ID}.log 2>&1 &

### 使用 srun 运行第二个节点
```

```
srun -N 1 --gres=gpu:4 -w ${host[2]} \
        python -m torch.distributed.launch \
        --nnodes=2 \
        --node_rank=1 \
        --nproc_per_node=4 \
        --master_addr="${host[1]}" \
        --master_port="29501" \
        train.py --epochs 3 --data coco128.yaml --cfg yolov5s.yaml --
weights '' --batch-size 128 --workers 16 --device 0,1,2,3 >>
train_rank1_${SLURM_JOB_ID}.log 2>&1 &
wait
```

3. 提交作业。使用 sbatch 提交主脚本。

```
$ sbatch -N 2 --gres=gpu:4 --qos=gpugpu run.sh
```

相关的日志输出文件可以在当前目录下看到。上述提交的作业，将产生以下日志:

slurm-[作业 ID].out 整体运行情况的输出。

train_rank0_[作业 ID].log 节点 1 的计算日志输出。

train_rank1_[作业 ID].log 节点 2 的计算日志输出。

2.3.3 包节点用户提交作业到专属 vip 队列

申请包节点后会根据用户包的节点数量创建一个专属队列，队列命名规则:

vip_gpu_用户名,例如：vip_gpu_scx6002

提交任务时必须添加下面参数指定提交到专属 vip 队列:

```
-p vip_gpu_用户名
```

单机任务提交举例（提交一个 4 卡任务，脚本名为 run.sh）：

```
sbatch --gpus=4 -p vip_gpu_用户名  ./run.sh
```

跨节点(多机)任务提交参考 2.3.2 ,注意添加参数：-p vip_gpu_用户名

注意：为了防止误提交到公共队列产生费用，包节点用户会将公共队列提交权限关闭。请注意提交到 vip_gpu_用户名 队列才能正常运行。有额外的资源需求也可申请开通公共队列。

2.3.4 高级功能：将数据集放到内存文件系统进行计算

BSCC-N32-H 区提供内存文件系统供用户使用，用户可以将数据集放到内存中直接读取，减少数据读取时间，最大程度降低 IO 瓶颈，发挥出 GPU 最大性能。内存文件系统使用的是每台机器（节点）上的内存，最大容量为 128GB，同时受用户提交任务申请的内存大小限制，例如：申请 4 卡，最多能用 128GB。

操作步骤：

（1）打包数据集

打包后解压到内存会比直接拷贝快很多,cd 到数据集目录所在路径，假如 datasets 是数据集的目录，则输入如下命令,也可以是其它名称。-cf 是打包不压缩，不建议加-z 参数压缩，压缩后解压时间会增加很多。

```
tar -cf datasets.tar datasets
```

注意 cd 到数据集的目录下打包，打包文件路径不建议写绝对路径，绝对路径打包解压后也会加上绝对路径。

（2）在提交脚本中运行程序之前加上下面代码，此代码是解压数据集到内存中

```
date
tar -xf datasets.tar -C /dev/shm
date
```

要注意解压后的文件夹路径，按照上述打包方式解压出的路径为 /dev/shm/datasets/

（3）将 python 代码中的读取数据集的路径改为/dev/shm/datasets （如果目录是其它名称按实际情况来写，注意要写解压到内存后的路径/dev/shm)

（4）sbatch 提交任务

如果有疑问可联系技术支持工程师协助。

2.4 查看作业状态

- 查看已提交的作业

```
$ parajobs
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
1292 gpu run.sh scx6002 R 1:31 1 paraai-n32-h-01-agent-21
paraai-n32-h-01-agent-21: index, utilization.gpu [%], utilization.memory [%], memory.total [MiB], memory.free [MiB], memory.used [MiB]
paraai-n32-h-01-agent-21: 3, 100 %, 55 %, 40960 MiB, 3206 MiB, 37147 MiB
```

```
paraai-n32-h-01-agent-21: 2, 100 %, 29 %, 40960 MiB, 3206 MiB, 37147 MiB
paraai-n32-h-01-agent-21: 1, 100 %, 56 %, 40960 MiB, 3206 MiB, 37147 MiB
paraai-n32-h-01-agent-21: 0, 100 %, 32 %, 40960 MiB, 3206 MiB, 37147 MiB
```

其中，

第一列 JOBID 是作业号，作业号是唯一的。

第二列 PARTITION 是作业运行使用的队列名。

第三列 NAME 是作业名。

第四列 USER 是超算账号名。

第五列 ST 是作业状态，R（RUNNING）表示正常运行，PD（PENDING）表示在排队，CG（COMPLETING）表示正在退出，S 是管理员暂时挂起，CD（COMPLETED）已完成，F（FAILED）作业已失败。只有 R 状态会计费。

第六列 TIME 是作业运行时间。

第七列 NODES 是作业使用的节点数。

第八列 NODELIST(REASON)对于运行作业（R 状态）显示作业使用的节点列表；对于排队作业（PD 状态），显示排队的原因。

2.5 取消作业

执行 scancel 作业 ID 取消作业

```
$ scancel 1292
```

2.6 输出 GPU 的利用率

本章节针对 GPU 利用率的采集和输出。如果不想 ssh 到计算节点，但是又想查看程序的 GPU 利用率情况，可以通过在作业脚本中添加如下脚本模板：

```
#!/bin/bash
#假设以 python 程序为例
module load  模块名称
source activate  环境名称

# 后台循环采集，每间隔 1s 采集一次 GPU 数据。
# 采集的数据将输出到本地 log_[作业 ID]/gpu.log 文件中
X_LOG_DIR="log_${SLURM_JOB_ID}"
X_GPU_LOG="${X_LOG_DIR}/gpu.log"
mkdir "${X_LOG_DIR}"
function gpus_collection(){
    sleep 15
    process=`ps -ef | grep python | grep $USER | grep -v "grep" | wc -l`
    while [[ "${process}" > "0" ]]; do
        sleep 1
        nvidia-smi >> "${X_GPU_LOG}" 2>&1
        echo "process num:${process}" >> "${X_GPU_LOG}" 2>&1
        process=`ps -ef | grep python | grep $USER | grep -v "grep" | wc -l`
    done
```

```
}
gpus_collection &


# 算例执行文件
python xxx.py
```

2.7 程序测试流程（正式提交任务请参考 2.3）

本章节针对程序部署完成后的验证性测试、调试。推荐在不确定程序是否能够正常运行时使用此流程进行测试。此方法终端中断后程序作业即退出运行，正式提交作业请不要使用此方法。测试程序成功运行后正式提交计算任务请参考 2.3 提交计算任务

此方法提交有如下弊端:

1、salloc 方式申请资源后资源使用时长和任务实际运行时长没有关联，容易出现由于网络不稳定等问题导致的中断，或计算完毕后忘记取消作业等问题导致的费用浪费。建议一般不要使用此方式!

2、如果用 salloc 方法申请资源运行程序，程序运行完成后作业不会自动停止，需使用 scancel 方式停止，才会终止计费。故推荐使用 sbatch 命令提交，作业正常结束后，会自动终止计费。

3、终端断开可能会导致运行的任务中断，如果作业未运行完成时终端断开会导致任务运行中断。sbatch 命令提交作业终端断开连接作业依然在后台运行直到结束。

4、如果多个作业提交到相同机器，登录机器上只能看到最后一个提交作业申请的卡。

使用下面方法可以申请 GPU 资源，然后直接登录到计算节点上进行计算。成功申请到资源后即开始计费。

1. 使用 salloc 命令申请 GPU。

```
$ salloc --gpus=1
salloc: Granted job allocation 1293
salloc: Waiting for resource configuration
salloc: Nodes paraai-n32-h-01-agent-13 are ready for job
```

如上所示，输出提示 paraai-n32-h-01-agent-13 为我们申请的主机，之后可以 ssh paraai-n32-h-01-agent-13 登录到节点进行计算。

2. 执行 scancel 作业 ID 取消该作业（释放资源并停止计费）

```
$ scancel 1293
```

3. 也可以执行 parajobs 命令查看申请到的节点名称。

执行：

```
parajobs
```