“神威•太湖之光”系统
快速使用指南

国家超级计算无锡中心

2016 年 8 月 15 日

1. 系统简介

“神威·太湖之光”计算机系统是由国家并行计算机工程技术研究中心在国家863计划支持下研制的新一代超级计算机系统。根据2016年6月20日国际TOP500排名公布的最新数据，全系统峰值运算速度为每秒12.54亿亿次，持续运算速度为每秒9.3亿亿次，均位居世界第一。性能功耗比每瓦60.5亿次，与其他相同量级计算机相比节能60%以上。“神威·太湖之光”是世界上首台峰值运算性能超过十亿亿次量级的超级计算机，也是我国第一台全部采用国产处理器构建的世界第一的超级计算机。

“神威·太湖之光”超级计算机采用了由国家“核高基”重大专项支持的“申威26010”众核处理器，该处理器由国家高性能集成电路设计中心采用自主核心技术成功研制，采用64位自主申威指令系统，260核，核心工作频率1.45GHz，峰值性能每秒超过3万次浮点结果，性能指标世界领先。

1.1. 用户可用资源

1.1.1. 国产高速计算系统：

- 申威26010处理器个数（计算节点数）：40960
- 申威26010处理器核组数：163840（基本收费单位）
- 每个计算节点32GB内存，每个核组可使用8GB内存
- 峰值性能125PFlops，适合有源代码的科学与工程计算

1.1.2. 商用辅助计算系统：

- 980台普通计算节点，每个计算节点24核心128GB内存
- 32台胖节点，每个节点8路16核心1TB内存
- 峰值性能1PFlops，提供工业设计等商用软件计算服务

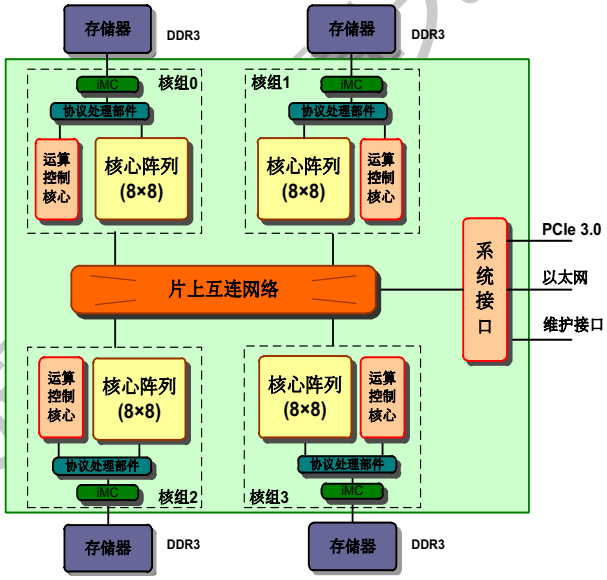
1.1.3. 通信网络带宽：双向 14GB/s

- 国产高速计算系统：14GB/s
- 商用普通计算节点：14GB/s
- 商用胖节点：28GB/s

1.1.4. 存储磁盘空间：

- 国产高速计算系统：10PB
- 商用辅助计算系统：5PB + 10PB
- 云存储系统：5PB

1.2. “申威 26010”众核处理器



申威众核处理器采用片上融合的异构体系结构，由 4 个核组构成，每个核组包括 1 个主核（运算控制核心）和 64 个从核（核心阵列）。整芯片共 260 个计算核心。

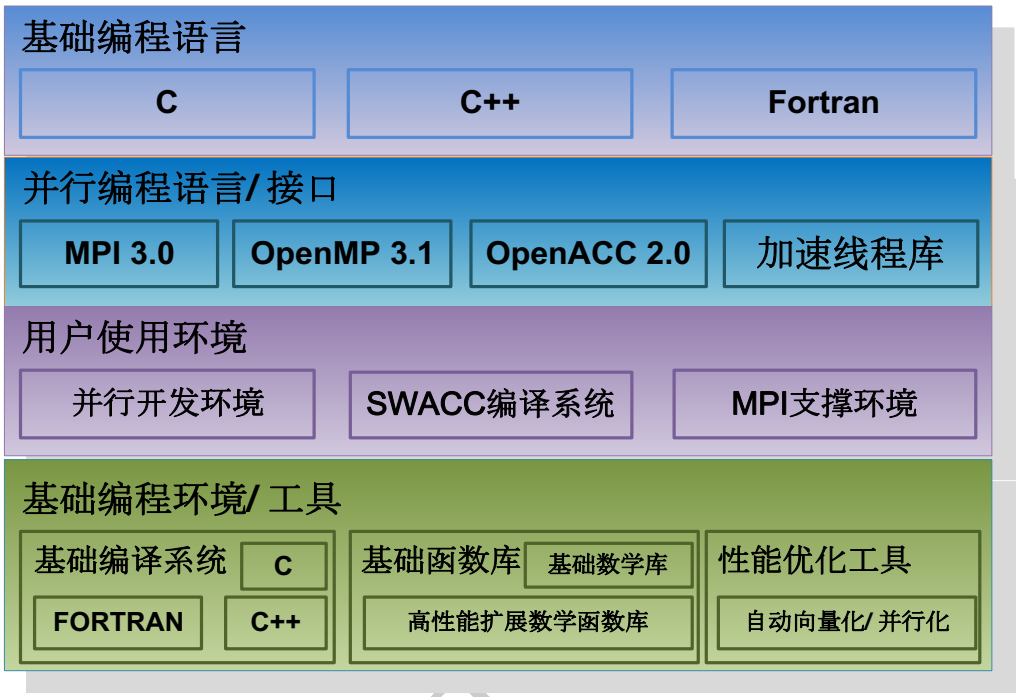
主核主频为 1.45GHz，每核组内存 8GB，L1 CACHE 大小为 32KB，L2 CACHE（数据 CACHE 和指令 CACHE 混合）大小为 256KB。

从核主频为 1.45GHz，可以通过 gld/gst 直接离散访问主存，也可以通过 DMA 方式批量访问主存，从核阵列之间可以采用寄存器通信方式进行通信。从核局部

存储空间大小为 64KB，指令存储空间为 16KB。

1.3. 语言环境

“神威•太湖之光”系统语言环境主要包括四个组成部分，如下图所示。



1) 基础编程语言

提供主流的基础编程语言支持，包括：C 语言，支持 C99 标准。C++语言，支持 C++03 标准，并提供支持 C++11 标准的 SWGCC 编译环境(从核不支持 C++)。Fortran 语言，支持 Fortran2003 标准中主要的功能，满足实际课题需求。

2) 并行编程语言/接口

提供与国际接轨的并行编程标准支持，包括 MPI3.0、OpenMP3.1、Pthreads、OpenACC2.0，支持消息并行模型、共享编程模型、加速编程模型，满足科学计算课题移植和开发的多样性需要。同时提供自主设计的加速线程库编程接口，满足部分追求极致性能的课题开发需求。

3) 用户使用环境

提供并行开发环境，以图形界面的方式提供编辑、编译、调试、性能监测于一体的使用环境。同时支持以字符界面的方式使用 Sunway OpenACC 编译系统、MPI 支撑环境等完成基本的开发过程，满足不同用户的使用习惯。

4) 基础编程环境

基础编程环境是所有上层语言及工具的基础，提供基础语言、主从异构编程、基础函数库、自动向量化/并行化等支持，提供丰富高效的编译优化功能。

1.4. 用户使用模式

系统所提供的主要的用户使用模式共两种：

1) 核组私有模式：“神威•太湖之光”上主要使用的模式。

2) 全片共享模式：满足部分课题的大内存需求。

每种模式中可支持多种具体的使用方式，具体如下表所示。

(注 L1：第一级并行；L2：第二级并行；L3：第三级并行)

序号	分类	使用方式	应用场景
1	核组私有模式	**L1:** 4 个 MPI 进程	纯主核应用，每个主核运行一个 MPI 进程
		L1: 4 个 MPI 进程 **L2:** 4 个从核组(OpenACC/Athread)	主要的使用方式；主核和从核组一对一使用
2	全片共享模式	**L1:** 1/2 个 MPI 进程	纯主核应用，MPI 私有空间内存需求大； 全片或 2 个核组的内存供 1 个 MPI 进程使用
		L1: 1 个 MPI 进程 **L2:** 4 个线程(OMP/Pthreads)	纯主核应用，两级并行
		L1: 2 个 MPI **L2:** 2 个线程(OMP/Pthreads)	纯主核应用，两级并行； MPI 进程私有空间内存需求大
		L1: 1/2 个 MPI 进程 **L2:** 1/2 个核组(OpenACC/Athread)	MPI 私有空间内存需求大，使用从核； 全片的内存或 2 个核组的内存供 1 个 MPI 进程使用
		L1: 1/2 个 MPI **L2:** OMP/pthreads **L3:** Athread	三级并行课题 满足少部分课题大规模扩展效率和大内存需求
		master 模式	模式 2 的上述用法中均包含一种 MPI 的 master 使用模式，即 0 号进程独占一个节点(内存需求大)，其他节点为正常使用。

2. 并行模式

2.1. 主从加速并行

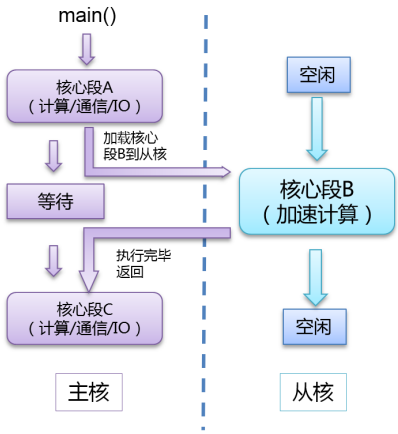


图 2-1 主从加速并行示意图

大部分课题采用的两级并行方式，根据实际应用课题的计算核心进行众核加速，主核主要完成不可众核并行部分的计算以及通信，在从核进行任务计算时，主核等待。

2.2. 主从协同并行

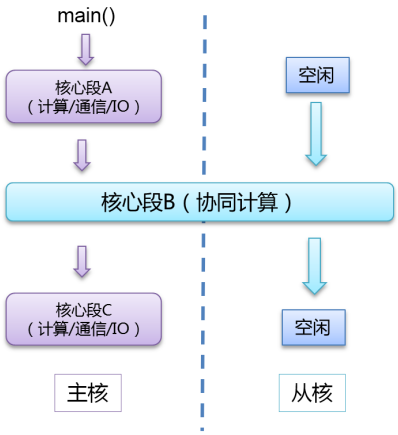


图 2-2 主从协同并行示意图

主核和从核作为对等的个体进行并行计算，根据各自计算能力进行负载分配，共同完成核心段的计算。如果课题对 LDM 需求不高，在 LDM 范围内从核可以

完成整个核心的计算，则该方式可以减少访存开销，并能够获得较好的并行效果。

2.3. 主从异步并行

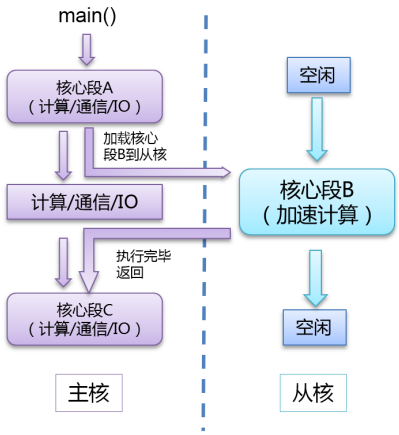


图 2-3 主从异步并行示意图

在从核进行加速计算的同时，主核完成其他计算、通信或 I/O 等操作，提高主从协作的并行效率。

2.4. 主从动态并行

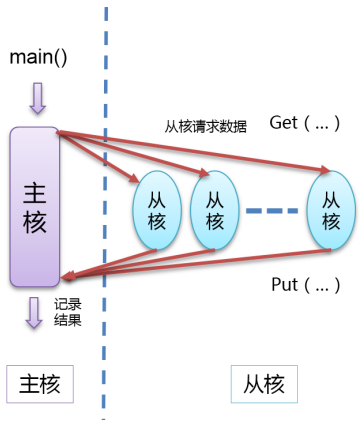


图 2-4 主从动态并行示意图

主核负责任务分配，从核负责取得新计算任务、完成计算、写回计算结果，该方法适合从核计算任务的计算时间不固定情况，可以考虑采用两级主从并行方式进行大规模并行计算，第一级是进程级的主从并行，第二级为异构众核处理器核组内部的主从并行。

3. 众核并行实例（编译、提交、运行）

3.1. 使用加速编程库

3.1.1. Fortran 示例

在本小节中，将给出一个小算例——针对两个数组赋值并求解对应元素的方差。

```
program main
implicit none
real ,dimension(imin:imax,jmin:jmax):: a，b，c
integer :: i,j,k
！ init part
do j= jmin,jmax
do i= imin,imax
   a(i,j)=i+j-0.8888
   b(i,j)=i+j+7.7777
end do
end do
！ excute
do j= jmin,jmax
do i= imin,imax
   c(i,j)=a(i,j)*b(i,j)- b(i,j)*b(i,j)
end do
end do
end
```

该段代码众核并行后将分为两个部分，即主核代码和从核代码。主核代码如下：

```
program main
        implicit none
        integer:： i,j
        real,dimension(imin:imax,jmin:jmax)::a,b,c
```

```
        integer,external :: slave_fun                    ！指定从核函数名
        common /shared_g1/ a,b,c                         ！共享数组
        do j= jmin,jmax
        do i= imin,imax
            a(i,j)=i+j-0.8888
            b(i,j)=i+j+7.7777
        end do
        end do
        call athread_init()                              ！初始化从核
        call athread_spawn(slave_fun, 1)                 ！从核并行任务开始
        call athread_join()                              ！等待从核任务结束
        call athread_halt()                              ！结束从核
    end
```

从核部分代码如下：

```
subroutine fun
        implicit none
        real,dimension(imin:imax,jmin:jmax)::a,b,c
        common /shared_g1/ a,b,c                         ！共享变量
        integer,dimension(imin:imax):: a_slave,b_slave,c_slave！从核局存变量申请
        integer i,j
        integer slavecore_id,reply
!$omp threadprivate (/local_g1/)                         ！从核编译引导语句
        call get_myid(slavecore_id)                      ！获得从核逻辑 id
        do j= jmin,jmax
        if(mod(j,corenum)+1.eq.slavecore_id)then         ！从核计算任务绑定
          reply=0
          call athread_get(0,a(imin,j),a_slave(imin), (imax-imin+1)*4,reply,0,0,0)！读入数据
          call athread_get(0,b(imin,j),b_slave(imin), (imax-imin+1)*4,reply,0,0,0)
          do while (reply.ne.2)
          end do
          do i= imin,imax
            c_slave(i)=a_slave(i)*a_slave(i)-b_slave(i)*b_slave(i)
          end do
```

```
            reply=0
            call athread_put(0,c_slave(imin),c(imin,j), (imax-imin+1)*4,reply,0,0)！写回数据
            do while (reply.ne.1)
            end do
          end if
          end do
          return
end
```

主从核分开编译：

sw5f90 -slave -c slave.f90

sw5f90 -host –c master.f90

sw5f90 -O3 -hybrid *.o -o test

从核代码用 sw5f90 -slave 编译，主核代码用使用 sw5f90 -host 编译，使用 sw5f90 -hybrid 进行链接生成最终的可执行文件。

提交作业命令如下：

bsub -I -b -q queue_name -n 1 -cgsp 64 -share_size 4096 -host_stack 128 ./test

其中：

- “-I”选项表示提交交互式作业，使作业输出在作业提交窗口；
- “-b”表示从核函数栈变量放在从核局部存储上，该选项为获取加速性能必须的提交选项；
- “-q”向指定的队列中提交作业；
- “-n”指定需要的所有主核数；
- “-cgsp”指定每个核组内需要的从核个数，指定时该参数必须<=64；
- “-share_size”指定核组共享空间大小，一般最大可以用到 7600MB；
- “-host_stack”指定主核栈空间大小，默认为 8M，一般设置为 128MB 以上。

从上面的典型众核代码可以看出，众核并行时需要将核心计算相关的数据通

过 common 进行共享。主核程序中首先初始化加速线程库，然后调用需要执行的从核程序的接口，等待从核计算结束，最后结束加速线程库。从核程序中主要实现共享数据的读入、计算和回写。

对主从核而言，共享存储空间都是可见的，这也意味着主从核都可以直接访问和修改共享存储空间的数据。采用 Fortran 编写程序时，数据共享通常采用 common 方式实现，因此一定要保持相应 commnon 名字和数组大小的一致性。通过 common 的方式实现数据的共享数组时，对数组的大小是有要求的——即数组的大小必须是固定的。对于动态大小的数组，可直接 common 共享数组指针，或者 common 指向该数组的 Cray 指针。下面给出三种 common 例子。

… n=10;m=10 real v(n,m) common/fun_v / v …	… real,pointer v(:,:) common/fun_v / v …	… real v_host(n,m) pointer(v_p,v_host) common/fun_v / v_p v_p=loc(v(1,1)) …

三种 common 方式的 DMA 通信性能存在差异，实验结果表明，数组指针的 DMA 性能最差，Cray 指针次之，静态数组最优。

3.1.2. C 示例

主核代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <athread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

extern SLAVE_FUN(func)();
static inline unsigned long rpcc()
{
```

```
        unsigned long time;
        asm("rtc %0": "=r" (time) : );
        return time;
}

#define J 64
#define I 1000
double a[J][I],b[J][I],c[J][I],cc[J][I];
double check[J];
unsigned long counter[J];
int main(void)
{
        int i,j;
        double checksum;
        double checksum2;
        unsigned long st,ed;

        printf("!!!!!!!!!! BEGIN INIT !!!!!!!!!!\n");fflush(NULL);
        for(j=0;j<J;j++)
        for(i=0;i<I;i++){
                a[j][i]=(i+j+0.5);
                b[j][i]=(i+j+1.0);
        }
        st=rpcc();
        for(j=0;j<J;j++)
        for(i=0;i<I;i++){
                cc[j][i]=(a[j][i])/(b[j][i]);
        }
        ed=rpcc();
        printf("the host counter=%ld\n",ed-st);
        checksum=0.0;
        checksum2=0.0;
        athread_init();

        st=rpcc();
```

```
        athread_spawn(func,0);//fflush(NULL);
        athread_join();
        ed=rpcc();
        printf("the manycore counter=%ld\n",ed-st);
        printf("!!!!!!!!!! END JOIN !!!!!!!!!\n");fflush(NULL);
        for(j=0;j<J;j++)
        for(i=0;i<I;i++){
                checksum=checksum+c[j][i];
                checksum2=checksum2+cc[j][i];
        }
        printf("the master value is %f!\n",checksum2);
        printf("the manycore value is %f!\n",checksum);
        athread_halt();
        printf("!!!!!!!!!! END HALT !!!!!!!!!\n");fflush(NULL);
    }
```

从核代码如下：

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "slave.h"

#define J 64
#define I 1000

__thread_local volatile unsigned long get_reply,put_reply;
__thread_local volatile unsigned long start,end;
__thread_local int my_id;
__thread_local double a_slave[I],b_slave[I],c_slave[I];
extern double a[J][I],b[J][I],c[J][I];
extern unsigned long counter[64];

void func()
{
```

```
        int i,j;
        my_id = athread_get_id(-1);
        get_reply = 0;
        athread_get(PE_MODE,&a[my_id][0],&a_slave[0],I*8,&get_reply,0,0,0);
        athread_get(PE_MODE,&b[my_id][0],&b_slave[0],I*8,&get_reply,0,0,0);
        while(get_reply!=2);

        for(i=0;i<I;i++){
                c_slave[i]=a_slave[i]/b_slave[i];
        }
        put_reply=0;
        athread_put(PE_MODE,&c_slave[0],&c[my_id][0],I*8,&put_reply,0,0);
        while(put_reply!=1);
}
```

编译链接过程：

```
sw5cc -host -c master.c
sw5cc -slave -c slave.c
sw5cc -hybrid master.o slave.o -o test
```

提交选项同 Fortran 示例，在此不再赘述。

3.2. 使用 OpenACC

下面介绍如何使用 OpenACC 将上面的程序移植到众核平台。

```
program main
        implicit none
        real ,dimension(1:512,1:64):: a, b, c
        integer :: i,j
        !init part
        do j= 1,64
        do i= 1,512
              a(i,j)=i+j-0.8888
```

```
            b(i,j)=i+j+7.7777
        end do
        end do
        !excute
        !$ACC PARALLEL LOOP COPYIN(a, b) COPYOUT(c) LOCAL(i)
        do j= 1,64
        do i= 1,512
            c(i,j)=a(i,j)*b(i,j)- b(i,j)*b(i,j)
        do
        end do
        !$ACC END PARALLEL LOOP
end program
```

将需要使用众核加速执行的执行(execute)部分的循环的前后使用 OpenACC 加速编译指示进行标注，如代码中加粗部分所示，其中 PARALLEL 表明该部分代码是需要加速执行的并行代码；LOOP 表示下面紧跟着的 j 循环需要在加速线程间进行并行划分；COPYIN(a,b)表明 a、b 数组需要拷贝到 LDM，因为 a、b 数组是只读，不需要更新回主存；COPYOUT(c)表明 c 数组在计算结束后需要拷贝回主存；LOCAL(i)表明变量 i 是加速线程私有的并且放入 LDM 中。

使用 swafort 进行编译，运行方法与 3.1 中类似。该程序的执行效果是，如果以 64 个加速线程运行，则每个线程会执行 64 个 j 循环中的一个，相应的会将 a、b、c 数组对应的数据在 LDM 中申请空间，并执行对应的数据拷贝和计算操作。

初始化部分的代码也可以参照该方法进行移植。详细用法请参见 OpenACC 用户手册。

3.3. 全片共享模式示例

全片共享模式支持主从混合编译的课题，在混合链接时需要使用选项 -allshare。根据用户的需求，全片共享模式分为 master 模式和非 master 模式，应用程序运行时，作业管理根据选项给用户应用程序分配内存。

3.3.1. 全片共享 master 模式

如果用户应用程序在实际运行时，主进程（一般规定为 0 进程）的内存需求较大（大于 7500M），其他进程的内存需求相对较小（小于 7500M），这种情况下，建议选择使用 master 模式提交作业，下面通过实例进行说明。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mpi.h"
#define DIM_I 1024
#define DIM_J 1024
#define DIM_K 256

int main(int argc, char *argv[])
{
   unsigned long *a,*b,*c,*d;
   unsigned long sum_total,size1;
   int myid, numprocs,i;
   extern void slave_func();

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   sum_total=0;
   if(myid==0){
      size1=DIM_I*DIM_J*DIM_K;
      a=(unsigned long*)malloc(size1*sizeof(unsigned long));
      b=(unsigned long*)malloc(size1*sizeof(unsigned long));
      c=(unsigned long*)malloc(size1*sizeof(unsigned long));
      d=(unsigned long*)malloc(size1*sizeof(unsigned long));
      for(i=0;i<size1;i++){
                  a[i] = 1; b[i] = 2; c[i] = 3; d[i] = 4;
                  sum_total = sum_total + a[i] + b[i]/2 + c[i]/3 + d[i]/4;
      }
```

```
    sum_total = sum_total/4;
    if(sum_total==size1){
        size1=4*8*size1/(1024*1024);
        printf("Process %d Memery size : %dMB and sum = %d \n",myid,size1,sum_total);
    }
  } else{
    size1=DIM_I*DIM_J;
    a=(unsigned long*)malloc(size1*sizeof(unsigned long));
    b=(unsigned long*)malloc(size1*sizeof(unsigned long));
    c=(unsigned long*)malloc(size1*sizeof(unsigned long));
    d=(unsigned long*)malloc(size1*sizeof(unsigned long));
    for(i=0;i<size1;i++){
        a[i] = 1; b[i] = 2; c[i] = 3; d[i] = 4;
        sum_total = sum_total + a[i] + b[i]/2 + c[i]/3 + d[i]/4;
    }
    sum_total = sum_total/4;
    if (sum_total==size1){
        size1=4*8*size1/(1024*1024);
        printf("Process %d Memery size : %dMB and sum = %d \n",myid,size1,sum_total);
    }
  }
  MPI_Finalize();
}
```

在程序中，数组数据类型为 long，在实际运行时，0 进程需要的总内存需求是 8G，其他进程内存需求较小。用户在提交程序时，为了满足内存需求，0 进程需要使用全片共享模式，而其他进程内存需求小，建议使用 master 模式提交作业：

bsub -b -I -pr -q q_sw_yyz -n 5 -cgsp 64 -allmaster -host_stack0 256 -cross_size0 10000 -host_stack 256 -cross_size 2500 ./a.out

上述提交命令行，-allmaster 说明程序需要使用 master 全片共享模式，-cross_size0 指定 0 进程所需内存大小，-cross_size 指定其他进程所需内存大小。

3.3.2. 全片共享非 master 模式

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mpi.h"
#define DIM_I 1024
#define DIM_J 1024
#define DIM_K 256

int main(int argc, char *argv[])
{
   unsigned long *a,*b,*c,*d;
   unsigned long sum_total,size1;
   int myid, numprocs,i;
   extern void slave_func();

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   sum_total=0;
   size1=DIM_I*DIM_J*DIM_K;
   a=(unsigned long*)malloc(size1*sizeof(unsigned long));
   b=(unsigned long*)malloc(size1*sizeof(unsigned long));
   c=(unsigned long*)malloc(size1*sizeof(unsigned long));
   d=(unsigned long*)malloc(size1*sizeof(unsigned long));
   for(i=0;i<size1;i++){
            a[i] = 1; b[i] = 2; c[i] = 3; d[i] = 4;
            sum_total = sum_total + a[i] + b[i]/2 + c[i]/3 + d[i]/4;
     }
   sum_total = sum_total/4;
   if(sum_total==size1){
   size1=4*8*size1/(1024*1024);
   printf("Process %d Memery size : %dMB and sum = %d \n",myid,size1,sum_total);
   }
```

```
    MPI_Finalize();
}
```

程序中主要数组的数据类型为 long，每个进程的内存需求是 8G。用户在提交程序时，为了满足内存需求，每个进程都需要使用全片共享模式，建议使用非 master 模式提交作业：

bsub -b -I -pr -q q_sw_yyz -n 5 -np 1 -cgsp 64 -sw3runarg "-a 1" -host_stack 256 -cross_size 10000 ./a.out

上述提交命令行，通过-sw3runarg "-a 1" -np 1 实现全片共享，-cross_size 指定进程所需内存大小。

4. 编译环境

4.1. 国产高速计算系统

4.1.1. 基础编译器

1）编译命令：

语言/ 目标核心	运算控制核心	运算核心	混合链接
C	sw5cc -host	sw5cc -slave	sw5cc -hybrid
C++	sw5CC -host	不支持	sw5CC -hybrid
Fortran	sw5f90 -host	sw5f90 -slave	sw5f90 -hybrid

2）常用编译选项：

选项	作用
-c	为每个源文件生成一个中间目标文件，但是不进行链接
-g	为之后的调试生成调试符号信息
-I<dir>	为预处理头文件添加查找<路径>
-l<library>	在链接阶段指定需要链的库文件
-L<dir>	为链接阶段添加查找<路径>
-lm	在链接阶段使用 libm 数学库，在 C 程序中若使用了 exp、log、sin 及 cos 等函数则需要调用该库
-o <filename>	指定生成的可执行(库)文件的名字
-O1~-O3	生成高级优化的可执行代码
-pg	为 gprof 分析程序生成反馈信息
-msimd	打开 simd 功能模块

-OPT： IEEE_arch=1	浮点异常处理（Fortran）
-mieee	浮点异常处理（C++）
-convert big_endian	按照大端读写文件（Fortran）
-freeform -fixedform	Fortran 自由书写格式 Fortran 固定书写格式

3）C 程序编译链接过程：

sw5cc -host -c master.c

sw5cc -slave -c slave.c

sw5cc -hybrid master.o slave.o -o test

4）Fortran 编译链接过程：

sw5f90 -host -c master.f90

sw5f90 -slave -c slave.f90

sw5f90 -hybrid master.o slave.o -o test

4.1.2. OpenACC 编译器

1）编译命令：

- FORTRAN 编译器：swafort[选项] 文件名
- FORTRAN 编译指示格式：添加!$acc 编译指示名 [子语[[,]子语]...]
- C 语言编译器：swacc[选项] 文件名
- C 语言编译指示格式：添加#pragma acc 编译指示名 [子句[[,] 子句]...]

2）编译选项：

编译选项	说明
-|--(h|help)	显示帮助
-SCFlags	指定的选项将被传递到编译 device 程序的串行编译器，若同时传递多个选项需使用逗号进行分割，中间不能有空格，例如： swafort -SCFlags -extend_source,-O3 hello.c

-HCFlags	指定的选项将被传递到 host 程序的基础编译器，若同时传递多个选项需使用逗号进行分割，中间不能有空格。
-LFlags	指定的选项将被传递到链接器，若同时传递多个选项需使用逗号进行分割，中间不能有空格。
-priv	**辅助选项：**对加速区进行私有化变量分析，给出变量的读写属性，遇到无法识别的函数调用则分析终止
-priv:ignore_call	**辅助选项：**对加速区进行私有化变量分析，给出变量的读写属性，忽略加速区内可能存在的函数调用的影响
-arrayAnalyse	**辅助选项：**数组访问模式分析，可以给出可 copy 的数组的建议
-ldmAnalyse	**辅助选项：**设备内存使用情况分析，编译之后在运行时会反馈各加速计算区内对设备内存的使用情况及相关建议
-preinline	**辅助选项：**对含有#inline 指示的函数调用语句中的函数进行内联，不对其他的加速指示进行处理，主要用于辅助函数内联
-preinline:all	**辅助选项：**在-preinline 的基础上，对同一个函数内的所有同名的被调用函数进行内联。
-dmaReuse	针对 data copy 指示，自动进行数据重用优化
-autoSwap	自动对适合的数组进行转置处理和优化
-vl-version	显示编译器和运行时库的版本号
-Minfo	显示编译器关于程序的分析信息
-keep	保留编译的中间文件
-dumpcommand [dumpfile]	将对中间文件的编译命令输出到 dumpfile 中

4.1.3. MPI

1）编译命令：

- FORTRAN 编译器：mpif90 [选项] 文件名。
- C 语言编译器：mpicc [选项] 文件名。
- C++语言编译器：mpiCC [选项] 文件名。

2）常用编译选项：

参考 4.1.1 节基础编译器选项。

4.2. 商用辅助计算系统(X86 集群)

4.2.1. intel 基础编译器

1）编译命令：

- FORTRAN 编译器：ifort [选项] 文件名
- C 语言编译器： icc [选项] 文件名
- C++语言编译：icpc [选项] 文件名

2）编译选项：

编译选项	说明
-O1	最大优化速度，但是关闭一些会增大文件大小而对速度提升很小的选项
-O2	最大优化速度 (默认)
-O3	最大优化速度，并打开更多激进的但不是对所有程序都能提高性能的选项
-O	同-O2
-Os	打开优化选项，但是关闭一些会增大文件大小而对速度提升很小的选项
-O0	关闭优化选项
-fast	等同打开 -xHOST -O3 -ipo -no-prec-div -static
-Ofast	等同打开-O3 -no-prec-div optimizations
-fno-alias	不采用代码混淆(用于保护代码，防反编译)
-fno-fnalias	不采用内部函数混淆，只混淆外部调用
-nolib-inline	禁用固有函数在内部扩展
-ftz	将非正常数值置为 0
-fp-model precise	保证数值不变优化
-convert big_endian	按照大端读写文件
-assume byterecl	确定文件读写按照字节或长字单位
-mcmodel=large	编译所需内存大于 2GB 时使用该选项
-free -fixed	Fortran 自由书写格式 Fortran 固定书写格式

4.2.2. intel 并行编译器

1）编译命令：

- FORTRAN 编译器：mpiifort [选项] 文件名
- C 语言编译器：mpiicc [选项] 文件名
- C++语言编译器：mpiicpc [选项] 文件名

2）编译选项：

参考 4.2.1 节 intel 基础编译器的编译选项

5. 作业管理

5.1. 基本概念

5.1.1. 作业与作业 ID

- **作业：**是用户编写的经过编译后并在主机上运行的用户可执行程序或脚本。用户作业程序的编写可以使用 MPI、OpenMP 等各种编程语言和环境，最终以可执行程序的形式存在，并以作业的形式被提交到主机中运行。任何作业都必须提交到某个指定的队列中调度运行；
- **作业 ID：**每道作业有唯一的作业 ID（整数，JOBID），这也是作业的唯一性标识。
- **计算节点号（nodeid）：**是计算节点的编号，就是每个操作系统核心（一个 IP 地址）所在 CPU 的编号。

5.1.2. 作业队列

- **作业队列：**分配给是指定用户的计算资源，用户在作业提交时要指定作业队列名。作业队列由管理员创建和管理、由用户使用。
- **作业流程：**所有用户运行课题以作业的形式提交到指定的队列中，系统为每道作业分配唯一的作业 ID，作业在队列中排队，接受作业管理系统

的自动调度运行。运行时，根据作业服务环境的队列属性允许普通用户对作业进行人机交互控制，或者由系统自动批处理。

5.1.3. 作业类型

1） **串行作业：**采用 C、Fortran 等语言编写的单进程串行应用程序或者 shell 程序；

2） **并行作业：**采用 MPI、OpenACC 等编程环境编写的多任务并行应用程序。

5.1.4. 作业运行模式

作业服务提供两种作业运行模式：交互作业、批式作业，由用户提交作业时决定。

1） **交互作业：**用户在作业提交命令 bsub 命令中指定-I 参数选择交互作业服务模式。作业提交后，作业运行信息在终端窗口中实时输出，允许用户进行各种人机交互控制。这种运行模式适合于在服务队列中进行课题调试时使用。一旦提交窗口被意外关闭，作业将自动转为脱机状态，不会终止，用户可以通过作业联机命令（bonline）重新在终端窗口接收作业的实时输出。交互运行模式下，如果系统发现当前队列中的空闲资源不能满足该作业的资源需求，作业提交时将自动报错退出。

2） **批式作业：**用户在作业提交命令 bsub 命令中不指定-I 参数提交批式作业。批作业提交方式后，系统根据计算资源和负载情况自动运行，并将结果输出文件保存在文件系统中。这种作业的特点是作业提交运行后，提交窗口可以关闭，作业运行不受任何影响。作业一旦开始运行，作业的标准输出和错误输出都将被自动重定向，如果提交时指定了输出定向的文件，则将输出到对应的文件中，否则系统将进行缓存。作业运行过程中，可以通过相应的命令查询作业输出。批作业运行模式下，即使当前队列中的空闲资源不能满足本作业的需求，作业将排队等待。由于交互作业在实时终端窗口的输入或操作都会对作业产生影响，所以相比之下批式

作业更适合业务课题的运行，建议所有业务运行的课题都使用批作业模式提交。

5.1.5. 作业运行状态

- PEND：作业等待分配计算资源；
- STARTING：作业正在启动运行。是作业已经分配资源后到作业正式启动完成开始运行之间的过渡状态；
- RUN：作业正在运行。作业已经完成调度并分派，作业占用的系统资源，正在运行中；
- DONE：作业正常完成并退出；
- EXIT：作业异常完成并退出；
- CKPT：作业正在系统级全透明保留，一旦保留完成，作业回复到运行状态；

5.2. 队列命名规则

1）国产高速计算系统：q_sw_xxxx

2）商用辅助计算系统：q_x86_xxxx

3）国产高速计算系统缺省队列：q_sw_expr

该队列命名为 q_sw_expr，主要用于用户程序的开发、移植、调试与优化使用，只要用户有系统账号，就可以使用该队列资源，而且免费，但有一定的限制：

- 每个任务的并行规模不能超过 16 个核组
- 每个任务的计算墙钟时间不能超过 1 小时

当用户完成程序的移植和优化后，根据对计算资源的估算，再申请必要的计算资源进行课题的计算。

5.3. 作业提交（bsub）

功能	向系统中提交作业
命令格式	Usage: bsub [-h][-v] bsub [-f sub_script]

bsub [-I]
[-p] //list job's nodename and spe_map
[-q queue_name]
[-n num_procs [-master] | -N num_nodes]
[-np node_mpes]
[-mpecg mpe_cgs]
[-cgsp spe_in_cg | -min_cgsp min_spe_in_cg | -rtp spe_rtp | -asy]
[-exclu | -shared | -cpuexclu]
[-js job_proj]
[-lfs_proj lfs_proj]
[-J jobname]
[-jobtype job_type] // available type: COMM / I/O / COMP / MEM / MIX
[-mpmd]
[-node nodelist]
[-o outfile]
[-k ckpt-period-minutes]
[-cross]
[-switchnode nodenum_in_switch]
[-midnode nodenum_in_mid]
[-cabnode nodenum_in_cab]
[-health health_level]
[-b]
[-parse]
[-PARSE <all | master | slave>]
[-quick]
[-m value]
[-share_size size]
[-priv_size size]
[-cross_size size]
[-ro_size size]
[-pe_stack size]
[-host_stack size]
[command [argument...]]

参数说明	-h 显示帮助信息 -I 提交交互式作业，使作业输出在作业提交窗口，无该选项时为批式作业 -q 向指定的队列中提交作业，必选项 -p 在作业输出中打印作业分配的节点列表及位图 -exclu | -shared | -cpuexclu 指定使用 CG 独占/CG 共享/CPU 独占模式 -n 指定需要的所有主核数 -N 指定需要的节点个数 -np 指定每节点内使用的主核数 -cgsp 指定每个 CG 内需要的从核个数，指定时该参数必须<=64。 -asy 指定使用非对称资源，表示各个 CG 内使用的从核的个数可以不同 -js 指定作业对应的课题代号 -lfs_proj 指定作业使用的局部文件代号 -node 指定运行作业的节点（CG 列表） -cross 要求分配全片 CPU（4CG 的 CPU） -health 指定分配资源的健康度级别 -o 将作业的 stdout 和 stderr 的输出定向到指定文件，可选项 -switchnode 指定每个 switch 中分配的节点数 -midnode 指定每个中板中分配的节点数 -cabnode 指定每个机舱中分配的节点数 -b 指定从核栈位于局存 -share_size 指定核组共享空间大小 -priv_size 指定每个核上私有空间大小 -cross_size 指定交叉段大小 -ro_size 指定只读空间大小 -m value 提供从核自陷模式的控制，指定-m 2时，将浮点控制状态寄存器fpcr的最后两位设为01，允许除不精确结果之外的所有浮点算术异常自陷，相当于编译器使用-OPT:IEEE_arith=2选项；指定-m 1时，将fcpr最后两位设为00，允许所有浮点算术异常自陷，相当于编译器使用-OPT:IEEE_arith=1选项；其他所有值将不对默认的fpcr进行修改。 -pe_stack 指定从核栈空间大小，默认为64K -host_stack 指定主核栈空间大小，默认为 8M
使用范例	向队列 queue 中提交互式作业 myjob，该作业共使用 1 个节点，四个主核： bsub –I –q queue –N 1 –np 4 ./myjob 作业提交成功后，将显示一行包含 jobid 的提示信息，其中包括作业 id 号，

	如”Job <102> is submitted to queue <queue>”，此时，jobid 就是 102，它是全局唯一的。一旦作业提交成功，用户对作业的各种操作就可以通过这个 jobid 来实现的。 使用缺省队列 q_sw_expr 可以不使用-q 参数，如： bsub –I –N 1 –np 4 ./myjob
注意事项	1） -I 参数与-o 参数通常不建议放在一起使用。因为使用-o 参数就无法在屏幕输出上看见程序的打印。 2） 每道作业提交成功后，都会有一个 jobid，这是本作业可以区别于其他作业的唯一特征。系统将保证 jobid 的唯一性。作业生命周期中，对作业的任何操作都需要以 jobid 为参数。 3） 本命令的各种参数都需要在用户程序之前。 4） 用户在 SN 上使用。

5.4. 作业终止（bkill）

功能	终止作业
命令格式	bkill [-h] [-J jobName] [-u user] [-q queue] [-f] [jobId]
参数说明	-h 帮助信息 -u 操作指定用户提交的作业 -q 操作指定队列内的作业 -J 操作指定作业名称的作业 -f 当终止批量作业时不需要确认 jobId 表示作业 id 标识符 说明：当 jobId 指定时，忽略-u –q 选项
使用范例	bkill 1234 终止队列中作业 id 号为 1234 的作业 bkill –q q1 终止队列 q1 内的本用户所有的作业

注意事项	1）使用-u 参数时，管理员可以指定所有用户，普通用户只能指定本用户 2）对于普通用户来说，即使使用-q 参数，也只能操作本用户在指定队列中的作业 **3）在 SN 上使用。**

5.5. 作业状态查询（bjobs）

功能	**查询队列中作业的状态信息**
命令格式	bjobs [-h] [-l] [-w] [-a | -d | -e | -p | -r] [-q queue_name] [-u user_name | -u all] [jobId]
参数说明	-q 指定要查询的队列 -l 长格式显示作业详细信息 -w 全长度显示，当列值的长度超过列宽时，不按列宽进行截取 -a 显示一段时间内的所有作业 -u 显示指定用户的作业，all 是特殊的关键字，可显示所有用户的作业信息 -d 显示最近正常完成的作业 -e 显示最近异常退出的作业 -p 显示处于 pend 状态的作业 -r 显示正在运行的作业 jobid 作业 id 号
使用范例	bjobs 1234 查询 id 号为 1234 的作业的作业运行状态信息。
注意事项	1）-u 参数只面向管理员开放 2）普通用户只能查看属于本用户的作业的信息 3）对于处于调度状态（pend）的作业，如果作业长时间处理 pend 状态，可以用 bjobs –l jobid 来查看作业不能调度运行的原因。 4）**用户在 SN 上使用。**

5.6. 作业输出查询（bpeek）

功能	查询作业的 stdout 和 stderr 输出信息
命令格式	bpeek [-h] [-f] jobId
参数说明	-h 显示帮助信息 -f 持续显示作业的输出信息，类似于 tail 的-f 选项 jobId 指定作业的 id 号
使用范例	bpeek 1234 显示 id 号为 1234 的作业的输出信息
注意事项	1） 本命令只能用于查询属于本用户的已经运行的作业的输出信息 2） **用户在 SN 上使用。**

5.7. 作业联机（bonline）

功能	当提交的交互作业窗口关闭后，可在新窗口中通过该命令实时显示该作业输出信息，进行联机操作
命令格式	bonline [-h]　jobId
参数说明	-h 显示帮助信息 jobid 指定作业的 id 号
使用范例	bonline 1234 对作业 id 号为 1234 的正在运行中的交互式作业进行联机操作
注意事项	1）只能普通用户操作本用户的作业， 2）操作的对象只能是本用户的正在运行状态的交互作业。否则都是无效的，会报错退出。如果作业处于联机状态，则作业输出不再发送至原实时输出终端，只发送到本输出终端 3）对作业联机后，可以在当前联机的会话窗口看到作业的实时输出信息 4）用户在 SN 上使用。