实验1 并行计算：从PC并行集群到MPI编程与应用

2012-11-19

由于如二代测序、免疫共沉淀、酵母双杂交等各种高通量技术在现代生命科学研究中的广泛应用，海量数据的大量积累，高性能计算成为生命科学数据分析中不可或缺的一部分。本实验在建立PC 并行集群的基础上，熟悉并行计算的模型，并采用MPI进行并行开发。

1 并行集群的网络配置

首先我们将分成3组，每组5台PC

- 第1组：bio11-15: 192.168.2.11-15，其中节点bio11设为服务器；
- 第2组：bio21-25: 192.168.2.21-25，其中节点bio21设为服务器；
- 第3组：bio31-35: 192.168.2.31-35，其中节点bio31设为服务器。

我们必须对并行集群的网络进行配置，使得集群内的节点之间的通讯成为可能。第一步需要配置的是机器的IP、节点名称，需要修改的文件是`/etc/hosts`，将本集群的节点名称和对应的IP加入该文件后，运行`service network restart`重启网络服务。

为了配置的方便，我们输入命令`chkconfig --levels 35 iptables off`关闭防火墙。当然这在实际应用中并不推荐。

2 NFS服务配置

NFS是网络文件系统（Network File System）的缩写，顾名思义就是在集群内部创建共享文件系统。

2.1 服务器端

1. 修改文件/etc/exports，将目录/home和/usr/local 设为共享。

```
/home       192.168.2.0/255.255.255.0(rw,no_root_squash)
/usr/local  192.168.2.0/255.255.255.0(rw,no_root_squash)
```

注意第二栏的字符串中间是没有空格的。

2. 启动NFS守护进程

```
chkconfig --levels 35 nfs on
service nfs restart
```

2.2 客户端配置

1. 编辑文件/etc/fstab，将NFS加入本地路径中

```
bio11:/home         /home       nfs     default     0 0
bio11:/usr/local    /usr/local  nfs     default     0 0
```

2. 将原有的/home目录更名并创建新的/home，并加载NFS

```
mv /home /home2
mkdir /home
mount -a
```

3. 输入命令df -k，查看NFS是否成功加载。

3 NIS服务配置

NIS的主要功能是可以让服务器端添加的用户被所有客户端所共享和识别，因此需要在服务器端安装ypserv程序，在客户端安装ypbind程序。具体的配置信息在这里就不赘述了，请参见本实验的slides。

4 SSH服务配置

在我们的机器上安装了OpenSSH，这是开源的SSH软件。我们配置SSH的目的是为了让集群内节点之间相互访问时，不需要输入密码进行访问，其原理是利用公钥/私钥进行加密/解密的机制。

1. 在所有节点上启动sshd守护进程

```
chkconfig --levels 35 sshd on
service sshd restart
```

2. 在不同帐号下，分别运行命令产生公钥/私钥对：

   ssh-keygen -t rsa

   过程中不输入密码，这样就在~目录下产生了一个文件夹.ssh，其下有几个文件：id_rsa，id_rsa.pub，分别存放私钥和公钥。然后运行

```
cd /home/<user>/.ssh/
cp id_rsa.pub authorized_keys
ssh <client_machine>
```

   这样就可以不使用密码互相访问了。

5 安装和配置MPICH

MPICH是并行开发的一个重要库程序和编译器等的组合。

5.1 在server端安装MPICH

编译和安装的故事我们已经讲的太多了，在这里就不多赘述了，这里唯一要说的就是需要编辑MPICH安装目录util/machines下的文件machines.LINUX，指定所有的集群节点。不要忘记的是，还必须将MPI命令的路径加入环境变量PATH。

5.2 MPI程序测试

安装配置完成以后，你需要测试一下你的并行环境是否已经配置完成了，可以执行一下example目录下的测试脚本。

```
which mpicc
which mpirun
mpicc -o MPIcode MPIcode.c -lmpi -L/PATH/TO/MPI_LIB -I/PATH/TO/MPI_INCLUDE
mpirun -np 2 MPIcode
```

如果在此过程中一切顺利的话，那么恭喜你，你已经往MPI的路上迈进了一大步了。

6 OpenPBS的安装与配置

PBS是由NAS开发的面向作业调度和系统资源管理的软件包，用于同构或异构的集群系统。OpenPBS是PBS系统的开源实现，遵循开源软件的相关约定。

6.1 OpenPBS的组成

OpenPBS主要由三个部件组成：

进程名称	进程描述
pbs_server	PBS服务守护进程，主要负责作业提交，安装在服务节点上
pbs_sched	PBS调度守护进程，主要负责作业调度，安装在服务节点上
pbs_mom	PBS-MOM守护进程，负责监控本机并执行作业，安装在所有计算节点上

为了安装和变换节点角色方便，在所有节点上均安装这三个组件，只是通过启动不同的组件来达到变换角色的需要：启动OpenPBS时，服务节点三个组件全部启动，而计算节点只启动MOM进程。

6.2 安装OpenPBS

在所有的节点上执行安装：

1. 解压下载的OpenPBS安装包并进入相关目录。
2. 执行configure脚本命令完成初始配置并完成编译安装:

```
./configure --disable-gui --set-default-server=<serverName> --set-
      server-home=/var/spool/PBS
make
make install
```

6.3 配置计算节点

1. 配置MOM进程配置文件/var/spool/PBS/mom_priv/config，写入下列内容

```
$logevent 0x1ff
$clienthost <server_host>
```

其中<server_host>是集群服务节点的机器名称。

2. 编辑/var/spool/PBS/server_name文件，设置server_name，写入集群服务节点名。

3. 执行pbs_mom，启动各个计算节点的MOM守护进程。

6.4 配置服务节点

1. 编辑spool/PBS/server_priv/nodes文件，写入所有节点名。

2. 执行命令pbs_server -t create，启动PBS服务器进程（仅首次启动运行）

3. 执行命令pbs_sched，启动PBS调度进程

4. 用qmgr创建并设置作业队列:

```
qmgr -c "c q bio"  # create queue bio‘
qmgr -c "s q bio queue_type=Execution"  # set queue
qmgr -c "s q bio resources_max.cput=24:00:00"  # set max resource
qmgr -c "s q bio resources_min.cput=1"   # set min resource
qmgr -c "s q bio resources_default.cput=12:00" # set default resource
qmgr -c "s q bio enabled=true"   # set status
qmgr -c "s q bio started=true"   # set status
qmgr -c "s s default_queue=bio"  # set default
```

6.5 将PBS加入系统服务

1. 编辑/etc/pbs.conf文件，加入以下内容

```
pbs_home=/usr/spool/PBS # PBS所在路径
pbs_exec=/usr/local
start_server=1          # server_node: 1, compute node: 0
start_sched=1           # server_node: 1, compute node: 0
start_mom=1             # all 1
```

2. 将OpenPBS目录中的src/tools/init.d/pbs文件复制到/etc/init.d/

3. 运行chkconfig -add pbs将其加入启动。

6.6 检查安装是否成功

进入一个非root帐户，执行命令

```
echo hostname > test.pbs # job scripts
qsub test.pbs            # qsub to submit job
```

7 MPI入门

MPI是Message Passing Interface的缩写，是一种信息传递库标准（规范），是一种分布式内存平台的模型，而非一种编译器规范，可用于同构或异构的并行计算机、集群和异构网络。是专为用户、库编写者、工具开发者设计的，其接口标准已被C/C++/Fortran 程序描述。因其标准化、跨平台性、函数化和开源性而被广为接受。

我们在这里并不会涉及到太多关于并行计算理论的内容，重要的是如何利用这些理论，更好的开发并行程序帮助我们去解决生物信息学里面的问题。

7.1 MPI的编程方式

对于基本应用，MPI如同其他消息传递系统一样易于使用。下面是一个简单的基于C程序的MPI样本代码，它是SPMD（single program, multiple data）方式的，也就是每个进程都执行相同的程序，通过返回的进程编号来区分不同的进程。该程序完成由进程0向进程1发送数据buf。

```c
/* first.c */
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv)
{
    int rank, size, tag=333;
    int buf[20];
    MPI_Status status;
    MPI_Init(&argc, &argv); /* Initialization function */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* process rank */
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* number of processes */
    if (rank == 0)
        MPI_Send(buf, 20, MPI_Int, 1, tag, MPI_COMM_WORLD); /* send data */
    if (rank == 1)
        MPI_Recv(buf, 20, MPI_Int, 0, tag, MPI_COMM_WORLD, &status); /*
            receive data */
    MPI_Finalize(); /* end of MPI */

    return 0;
}
```

7.2 MPI基本语句

1. MPI_COMM_WORLD
   进程由唯一的标识数（rank）表示，其取值为$0-(N-1)$。MPI_COMM_WORLD表示“MPI应用中所有的进程”，被称为通信子，提供消息传递所需的全部信息。

2. 进入和退出MPI
   MPICH提供了两个函数MPI_Init()和MPI_Finalize()。

3. 相互识别
   并行程序中的进程的相互通信是通过标识来进行的，因此需要相互能够识别。一个进程通过调用MPI_Comm_rank()来发现自身的标识，而通过调用MPI_Comm_size()来返回进程总数。

4. 发送消息
   消息是一组给定数据类型的元素。通过指定目标标识，消息被发送给一个指定的进程，其函数原型是

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
      int tag, MPI_Comm comm);
```

5. 接收消息
   接收方指定消息的标识和发送进程的标识。MPI_ANY_TAG和MPI_ANY_SOURCE可用于接收任意标识和从任意发送进程发送而来的消息。

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
      int tag, MPI_Comm comm, MPI_Status *status);
```

   有关接收消息的信息在一个状态变量status中返回。

通过上述这些函数，你可以编写任意应用程序。在MPI中也许有很多其他函数，但迄今为止绝大部分这些函数都可以在上述这些函数的基础上构造得到。

7.3 MPI实例

下面是应用MPI实现环的一种简单C程序。

```
/* cycle.c */
#include "mpi.h"
#include <stdio.h>

#define T_SIZE 2000

int main(int argc, char **argv)
{
    int ierr, prev, next, tag, rank, size;
    MPI_Status status;
    double send_buf[T_SIZE], recv_buf[T_SIZE];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    next = rank + 1;
```

```
17      if (next > size) next = 0;
18      prev = rank - 1;
19      if (prev < 0) prev = size - 1;
20
21      if (rank == 0) {
22          MPI_Send(send_buf, T_SIZE, MPI_DOUBLE, next, tag, MPI_COMM_WORLD);
23          MPI_Recv(recv_buf, T_SIZE, MPI_DOUBLE, prev, tag+1, MPI_COMM_WORLD,
                &status);
24      } else {
25          MPI_Recv(recv_buf, T_SIZE, MPI_DOUBLE, prev, tag, MPI_COMM_WORLD, &
                status);
26          MPI_Send(recv_buf, T_SIZE, MPI_DOUBLE, next, tag+1, MPI_COMM_WORLD);
27      }
28
29      MPI_Finalize(void);
30
31      return 0;
32  }
```

7.4 计算π的程序

```
1   #include "mpi.h"
2   #include <stdio.h>
3   #include <math.h>
4
5   double f(double a)
6   {
7       return (4.0 / (1.0 + a*a));
8   }
9   int main(int argc, char **argv)
10  {
11    int done = 0, n=100, myid, numprocs, i;
12    double PI25DT = 3.14159265358979323846643;
13    double mypi, pi, h, sum, x, a, startwtime, endwtime;
14    MPI_Init(&argc, &argv);
15    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
16    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
17
18    if (myid == 0)
19      startwtime = MPI_Wtime();
20
21    h = 1.0 / (double) n;
22    sum = 0.0;
23    for (i = myid + 1; i <= n; i += numprocs)
24    {
25      x = h * ((double)i - 0.5);
26      sum += f(x);
27    }
28
29    mypi = h * sum;
30    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
31    if (myid == 0) {
32       printf("pi is %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
33       endwtime = MPI_Wtime();
34       printf("wall clock time = %f\n",     endwtime - startwtime);
35    }
36
37    MPI_Finalize();
```

```
38
39    return 0;
40 }
```

8 MPI进阶篇

8.1 基本概念

1. 消息包
   在消息传递过程中，发送和接收的消息除了数据部分外，还带有识别消息和选择接收消息的信息。这个信息由确定的值组成的，我们称之为信封，这些值包括：source(源)、destination(目的)、tag(标识)、communicator(通信子)。

2. 组和通信子
   一个组(group)是进程的有序集合。组内的每个进程与一个整数rank相联系，序列是连续的并从0开始。组用模糊的组对象来描述，因此不能直接从一个进程到另一个进程传送。可在一个通信子中使用组来描述通信空间中的参与者并对这些参与者进行分级(rank)。

   一个通信子(communicator)指定一个通信操作的上下文。每个通信上下文提供一个单独的"'通信全域"；消息总是在一个上下文内被发送和接收，不同的上下文发送的消息互不干涉。通信子也指定共享这个通信上下文的进程组。这个进程组的编号，并且由这个组中的进程号标识。

   一个内建的通信子MPI_COMM_WORLD在MPI初始化后建立，它允许和可存取的所有进程通讯，进程是由它们在MPI_COMM_WORLD的标志号所标识。

3. 基本数据类型
   MPI系统描述了消息的数据类型，从最简单的原始机器类型到复杂的结构、数组和下标。以下是与C描述的MPI基本数据类型：

8.2 点到点通信

MPI点到点通信有两种消息传递的机制：阻塞的和非阻塞的。对于阻塞方式，必须等到消息从本地送出以后，才能执行后续的语句，保证了消息缓存等资源的可再用性；而非阻塞的方式不需等到消息从本地发出，就可执行后续语句，从而允许通信和计算的重叠，利用合适的硬件使得计算和通信同步执行，但是非阻塞调用的返回并不保证资源的可再用性。

点到点通信的发送和接收语句必须是匹配的，为了区分不同进程或同一进程发送来的不同消息，可以采用通讯体（source-dest）和标志位

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

（tag）实现相应语句的匹配，也可以接收不确定source和tag的消息，例如MPI_ANY_TAG和MPI_ANY_SOURCE。

1. 阻塞通信
   对于阻塞类型的通信，MPI_Send和MPI_Recv函数的原型为：

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
      int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src, int
       tag, MPI_Comm comm, MPI_Status *status);
```

   发送的含义是将包含count个datatype类型的首地址为buf的消息发送到dest 进程，该消息是与标识tag和通信体comm封装在一起的；接收的含义是接收标识为tag和通信体为comm的消息，并将该消息写入首地址为buf的缓冲区，返回值status是一种结构体，包含两个域，分别为MPI_SOURCE和MPI_TAG。该结构体还可包含附加域。

2. 非阻塞通信
   对于非阻塞通信，也有两个函数：

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
      int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int src,
      int tag, MPI_Comm comm, MPI_Request *request);
```

   可用MPI_Wait和MPI_Test来结束非阻塞通信：

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

其中MPI_Test用来检测非阻塞操作提交的任务是否结束并立即返回，而MPI_Wait则一直等到非阻塞提交的任务结束才返回。因此可以认为：

阻塞通信 = 非阻塞通信 + MPI_Wait

3. 消息的检测
函数MPI_Probe和MPI_Iprobe允许在没有实际输入消息的情况下对通信进行检测，用户可以根据返回的信息决定如何接收（该信息一般用status返回），或根据被检查消息的长度分配接收缓冲区大小，或进行条件分支等，其格式如下：

```
int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_Status *status);
int MPI_Iprobe(int src, int tag, MPI_Comm comm, int *flag, MPI_Status
      *status);
```

两者也是阻塞和非阻塞的差别，如果MPI_Iprobe返回flag=true，则状态目标（status）才能获取source，tag以及消息长度；而MPI_Probe是一个阻塞的语句，必须等到相匹配的消息（src，tag）到达才完成。

8.3 点到点通信实例

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv)
{
    int locId, data[100], tag=8888;
    MPI_Status status;
    MPI_Init(&argc, &argv) ;
    MPI_Comm_rank(MPI_COMM_WORLD, &locId) ;
    if(locId == 0) {
        MPI_Request events;
        MPI_Isend(data, 100, MPI_INT, 1, tag , MPI_COMM_WORLD, &events) ;
        MPI_Wait(&events, &status) ;
    }
    if (locId == 1) {
        MPI_Probe(MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
        if (status.MPI_SOURCE==0)
            MPI_Recv(data, 100, MPI_INT, 0, tag, MPI_COMM_WORLD, &status) ;
    }
    MPI_Finalize();

    return 0;
}
```

8.4 群体通信

群体通信意味着一个通信子中的所有进程调用同一例程，所有的群体操作都是阻塞的，主要包括：

1. 同步(barrier)

```
int MPI_Barrier(MPI_Comm comm);
```

   该函数使得调用者阻塞，直到该通信子内所有进程都调用它，起到同步的功能。

2. 从一个进程到组内所有进程的广播(broadcast)

```
int MPI_Bcast(void *buf, int cout, MPI_Datatype datatype, int root,
      MPI_Comm comm);
```

   所有进程使用同一计数、数据类型、根和通信子。操作前，根进程缓冲区包含一个消息；操作后，所有进程缓冲区均包含该消息。

3. 从一个进程分散数据到本组内所有进程(scatter)

```
int MPI_Scatter(void *sndbuf, int sndcnt, MPI_Datatype sndtype, void *
      rcvbuf, int rcvcnt, MPI_Datatype rcvtype, int root, MPI_Comm
      comm);
```

   也称为散播，所有进程使用同一计数、数据类型、根和通信子。在操作前，根发送缓冲区包含长度为$N * sndcnt$的消息，N为进程数目；操作后，相等划分消息，并且分散到随后标识数序的所有进程（包括根）的接收缓冲区中。

4. 从本组所有进程手机数据到一个进程(gather)，与scatter刚好相反的过程。

```
int MPI_Gather(void *sndbuf, int sndcnt, MPI_Datatype sndtype, void *
      rcvbuf, int rcvcnt, MPI_Datatype rcvtype, int root, MPI_Comm
      comm);
```

5. 将gather得到的数据发送到组内所有进程(allgather)

6. 组内的多对多分散/收集(alltoall)

7. 求和、最大值、最小值及用户定义的函数等的汇总操作(reduce)

```
int MPI_Reduce(void *sndbuf, void *rcvbuf, int count, MPI_Datatype
      datatype, MPI_Op op, int root, MPI_Comm comm);
```

所有进程均使用同一计数、数据类型、根和通信子。操作后，根进程在其接收缓冲区中保存所有进程的发送缓冲区的汇总结果，这些常用操作包括：MPI_MAX，MPI_MIN，MPI_SUM，MPI_PROD，MPI_LAND，MPI_BAND，MPI_LOR，MPI_BOR，MPI_LXOR，MPI_BXOR，或者是用户自定义的汇总函数。

8. scan或prefix操作。

8.5 群体通信实例

```
#include "mpi.h"

int main(int argc, char **argv)
{
    int i, myrank, size, root, full_domain_length, sub_domain_length;
    double global_max, local_max, *full_domain, *sub_domain;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    root = 0;
    if (myrank == 0)
        get_full_domain(&full_domain, &full_domain_length);
    MPI_Bcast(&full_domain_length, 1, MPI_INT, root, MPI_COMM_WORLD);
    sub_domain_length = full_domain_length / size;
    sub_domain = (double*)malloc(sub_domain_length * sizeof(double));
    MPI_Scatter(full_domain, sub_domain_length, MPI_DOUBLE, sub_domain,
          sub_domain_length, MPI_DOUBLE, root, MPI_COMM_WORLD);
    compute(sub_domain, sub_domain_length, &local_max);
    MPI_Reduce(&local_ax, &global_max, 1, MPI_DOUBLE, MPI_MAX, root,
          MPI_COMM_WORLD);
    MPI_Gather(sub_domain, sub_domain_length, MPI_DOUBLE, full_domain,
          sub_domain_length, MPI_DOUBLE, root, MPI_COMM_WORLD);

    return 0;
}
```

本程序并不完整，尝试去完善这个程序，尤其是几个用户定义的函数。

8.6 其它函数

这里简要说一个函数，就是关于时钟的函数，这函数对于性能调试是很重要的。

```
double MPI_Wtime(void);
double MPI_Wtick();
```

其中MPI_Wtime()返回当前系统的Wall time，是一个浮点的秒数；而MPI_Wtick()则返回上一函数返回值的精度。

9 MPI编程习题

1. 在图论中，图G的一个连通分量(connected components)是G的一个最大连通子图，该子图中每对顶点间均有一条路径。根据图G，如何找出其所有连通分量的问题称为连通分量问题。解决该问题的方法有三种：(1) 使用某种形式的图搜索技术；(2) 通过图的布尔连接矩阵计算传递闭包；(3) 顶点合并算法。下面的源程序就是顶点合并算法的并行实现。

   顶点合并算法中，开始途中的N个顶点被看作N个孤立的超顶点（super vertex），算法执行过程中，有边连通的超顶点相继合并，直到形成最后的整个连通分量。每个顶点属于且属于一个超顶点，超顶点中标号最小的称为该超顶点的根。

   该算法的流程由一系列的循环组成。每次循环分为3步：(1) 发现每个顶点的最小标号邻接超顶点；(2) 把每个超顶点的根连到最小标号邻接超顶点的根上；(3) 所有在第(2)步连接在一起的超顶点合并成为一个更大的超顶点。

   图G的顶点总数为N，因为超顶点的个数每次循环后至少减少一半，所以把每个连通分量连接成单个超顶点至多需要$\log N$次循环。顶点i的超顶点的根为$D(i)$，刚开始时$D(i) = i$。算法运行后，所有处于同一连通分量的顶点具有相同的$D(i)$。

   算法中为顶点设置数组变量D和C，其中$D(i)$为顶点i所在的超顶点号，$C(i)$和为顶点i或超顶点i相连的超顶点号等，根据程序运行的阶段不同，意义也有相应的变化。算法主要分为5个步骤：(1) 各处理器并行为每个顶点找出相应的$C(i)$；(2) 各处理器并行为每个超顶点找出最小邻接超顶点，编号放入$C(i)$中；(3) 修改所有$D(i) = C(i)$；(4) 修改所有$C(i) = C(C(i))$，运行$\log N$次；(5) 修改所有$D(i) = min(C(i), D(C(i)))$。

```c
/* connect.c */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <mpi.h>
#define A(i,j) A[i*N+j]

int N;
int n;
int p;
int *D,*C;
```

```
13 int *A;
14 int temp;
15 int myid;
16 MPI_Status status;
17
18 void print(int *P)
19 {
20     int i;
21     if(myid==0)
22     {
23         for(i=0;i<N;i++)
24             printf("%d ",P[i]);
25         printf("\n");
26     }
27 }
28
29 void readA()
30 {
31     char *filename;
32     int i,j;
33     printf("\n");
34     printf("Input the vertex num:\n");
35     scanf("%d",&N);
36     n=N/p;
37     if(N%p!=0) n++;
38     A=(int*)malloc(sizeof(int)*(n*p)*N);
39     if(A==NULL){
40     printf("Error when allocating memory\n");
41     exit(0);
42     }
43     printf("Input the adjacent matrix:\n");
44     for(i=0;i<N;i++)
45         for(j=0;j<N;j++)
46             scanf("%d",&A(i,j));
47     for(i=N;i<n*p;i++)
48         for(j=0;j<N;j++)
49             A(i,j)=0;
50 }
51
52 void bcast(int *P)
53 {
54     MPI_Bcast(P,N,MPI_INT,0,MPI_COMM_WORLD);
55 }
56
57 int min(int a,int b)
58 {
59     return(a<b?a:b);
60 }
61
62 void D_to_C()
63 {
64     int i,j;
65     for(i=0;i<n;i++){
66         C[n*myid+i]=N+1;
67         for(j=0;j<N;j++)
68         if((A(i,j)==1)&&(D[j]!=D[n*myid+i])&&(D[j]<C[n*myid+i])){
69             C[n*myid+i]=D[j];
70         }
71         if(C[n*myid+i]==N+1)
72             C[n*myid+i]=D[n*myid+i];
73     }
74 }
```

```
75
76
77  void C_to_C()
78  {
79      int i,j;
80      for(i=0;i<n;i++){
81      temp = N+1;
82      for(j=0;j<N;j++)
83          if((D[j]==n*myid+i)&&(C[j]!=n*myid+i)&&(C[j]<temp)){
84              temp=C[j];
85          }
86          if(temp==N+1) temp=D[n*myid+i];
87          C[myid*n+i]=temp;
88      }
89  }
90
91  void CC_to_C()
92  {
93      int i;
94      for(i=0;i<n;i++)
95          C[myid*n+i]=C[C[myid*n+i]];
96  }
97
98  void CD_to_D()
99  {
100     int i;
101     for(i=0;i<n;i++)
102         D[myid*n+i] = min(C[myid*n+i],D[C[myid*n+i]]);
103 }
104
105 void freeall()
106 {
107     free(A);
108     free(D);
109     free(C);
110 }
111
112 int main(int argc,char **argv)
113 {
114     int i,j,k;
115     double l;
116     int group_size;
117
118     double starttime, endtime;
119     MPI_Init(&argc, &argv);
120     MPI_Comm_size(MPI_COMM_WORLD, &group_size);
121     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
122     p = group_size;
123     MPI_Barrier(MPI_COMM_WORLD);
124     if(myid==0)
125         starttime=MPI_Wtime();
126
127     if(myid==0) readA();
128     MPI_Barrier(MPI_COMM_WORLD);
129     MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
130     if(myid!=0){
131         n=N/p;
132         if(N%p != 0) n++;
133     }
134     D = (int*)malloc(sizeof(int)*(n*p));
135     C = (int*)malloc(sizeof(int)*(n*p));
136     if(myid != 0)
```

```
137         A = (int*)malloc(sizeof(int)*n*N);
138
139     for(i=0; i<n; i++) D[myid*n+i] = myid*n+i;
140     MPI_Barrier(MPI_COMM_WORLD);
141     MPI_Gather(&D[myid*n], n, MPI_INT, D, n, MPI_INT, 0,
                MPI_COMM_WORLD);
142     bcast(D);
143     MPI_Barrier(MPI_COMM_WORLD);
144
145     if(myid == 0)
146         for(i=1; i<p; i++)
147             MPI_Send(&A(i*n,0), n*N, MPI_INT, i, i, MPI_COMM_WORLD);
148     else
149         MPI_Recv(A, n*N, MPI_INT, 0, myid, MPI_COMM_WORLD, &status);
150     MPI_Barrier(MPI_COMM_WORLD);
151
152     l=log(N)/log(2);
153
154     for(i=0; i<l; i++){
155         if(myid==0) printf("Stage %d:\n", i+1);
156
157         D_to_C();
158         MPI_Barrier(MPI_COMM_WORLD);
159         MPI_Gather(&C[n*myid], n, MPI_INT, C, n, MPI_INT, 0,
                    MPI_COMM_WORLD);
160         print(C);
161         bcast(C);
162         MPI_Barrier(MPI_COMM_WORLD);
163
164         C_to_C();
165         print(C);
166         MPI_Barrier(MPI_COMM_WORLD);
167         MPI_Gather(&C[n*myid], n, MPI_INT, C, n, MPI_INT, 0,
                    MPI_COMM_WORLD);
168         MPI_Gather(&C[n*myid], n, MPI_INT, D, n, MPI_INT, 0,
                    MPI_COMM_WORLD);
169         MPI_Barrier(MPI_COMM_WORLD);
170
171         if (myid == 0)
172             for (j=0;j<n;j++)
173                 D[j]=C[j];
174         for (k=0; k<l; k++){
175             bcast(C);
176             CC_to_C();
177             MPI_Gather(&C[n*myid], n, MPI_INT, C, n, MPI_INT, 0,
                        MPI_COMM_WORLD);
178         }
179         bcast(C);
180         bcast(D);
181
182         CD_to_D();
183         MPI_Gather(&D[n*myid], n, MPI_INT, D, n, MPI_INT, 0,
                    MPI_COMM_WORLD);
184         print(D);
185         bcast(D);
186     }
187
188     if (myid==0) printf("Result: \n");
189     print(D);
190     if(myid==0){
191         endtime=MPI_Wtime();
192         printf("The running time is %lf\n",endtime - starttime);
```

```
193     }
194     freeall();
195     MPI_Finalize();
196
197     return 0;
198 }
```

- 下面是一个简单的8×8的邻接矩阵，请试着用这个程序找出连通分量出来。

  0 0 0 0 0 0 0 1
  0 0 0 0 0 1 1 0
  0 0 0 0 0 0 0 0
  0 0 0 0 0 1 0 1
  0 0 0 0 0 0 1 1
  0 1 0 1 0 0 0 0
  0 1 0 0 1 0 0 0
  1 0 0 1 1 0 0 0

- 改写该程序，使其能够从文件中获取邻接矩阵，运行时可以是

```
1 mpirun -np N connect -f nj.txt
```

- 编写程序，指定顶点数的情况下，生成随机邻接矩阵，并计算其连通分量。
- 在不同顶点数目下，重复生成1000个随机邻接矩阵，指定不同处理器数目，用上述程序处理，查看分析时间的变化随顶点个数、处理器数目的变化。并画图表示。
- 编写PBS脚本，用qsub提交该脚本，进行运算。

2. 安装BLAST和mpiblast，阅读mpiblast代码，阐述其并行策略。用其自带的示例进行分析，观察加速比。