MPI Programming

实验内容

- 尝试在单机上安装并运行MPI环境。（MPICH或者OpenMPI等）
- 编程
  - 1. 用MPI_Reduce接口改写大数组各元素开平方求和(data[N], data[i]=i*(i+1))的代码（可通过命令行传入N的值，比如1000，10000，100000）；
  - 2. 用MPI_Send和MPI_Receive接口计算积分：$y=x^3$，求其在[10,100]区间的积分。
- 尝试在多个节点上运行上述MPI程序，可设置不同的进程数对结果进行比较，并评估所需时间。

回顾：MPI并行程序设计接口

- 基本编程接口：MPI提供了6个最基本的编程接口，理论上任何并行程序都可以通过这6个基本API实现
  - 1. MPI_Init (argc, argv)： 初始化MPI，开始MPI并行计算程序体
  - 2. MPI_Finalize()： 终止MPI并行计算
  - 3. MPI_Comm_Size(comm, size)：确定指定范围内处理器/进程数目
  - 4. MPI_Comm_Rank(comm, rank)：确定一个处理器/进程的标识号
  - 5. MPI_Send (buf, count, datatype, dest, tag, comm)：发送一个消息
  - 6. MPI_Recv (buf, count, datatype, source, tag, comm, status)：接受消息
  - size：进程数，rank：指定进程的ID
  - comm：指定一个通信组(communicator)
  - dest：目标进程号，source：源进程标识号，tag：消息标签

MPI并行程序设计接口

- MPI并行计算初始化与结束
  - 任何一个MPI程序都要用MPI_Init和MPI_Finalize来指定并行计算开始和结束的地方；同时在运行时，这两个函数将完成MPI计算环境的初始化设置以及结束清理工作。处于两者之间的程序即被认为是并行化的，将在每个机器上被执行。

```
#include <mpi.h>
#include <stdio.h>
main(int argc, char **argv)
{
   int numtasks, rank;
   MPI_Init(&argc, &argv);
   printf("Hello parallel
           world!\n");
   MPI_Finalize();
   exit(0);
}
```

在一个有5个处理器的系统中，输出为：

```
Hello parallel world!
Hello parallel world!
Hello parallel world!
Hello parallel world!
Hello parallel world!
```

MPI并行程序设计接口

- 通信组（Communicator）
  - 为了在指定的范围内进行通信，可以将系统中的处理器划分为不同的通信组；一个处理器可以同时参加多个通信组；MPI定义了一个最大的缺省通信组：MPI_COMM_WORLD，指明系统中所有的进程都参与通信。一个通信组中的总进程数可以由MPI_Comm_Size调用来确定。
- 进程标识
  - 为了在通信时能准确指定一个特定的进程，需要为每个进程分配一个进程标识，一个通信组中每个进程标识号由系统自动编号（从0开始）；进程标识号可以由MPI_Comm_Rank调用来确定。

MPI并行程序设计接口

- 点对点通信
  - 同步通信：阻塞式通信，等待通信操作完成后才返回
    - *MPI_Send (buf, count, datatype, dest, tag, comm)*：发送一个消息
    - *MPI_Recv (buf, count, datatype, source, tag, comm, status)*：接受消息
  - 同步通信时一定要等到通信操作完成，这会造成处理器空闲，因而可能导致系统效率下降，为此MPI提供异步通信功能
  - 异步通信：非阻塞式通信，不等待通信操作完成即返回
    - *MPI_ISend (buf, count, datatype, dest, tag, comm, request)*：异步发送
    - *MPI_IRecv (buf, count, datatype, source, tag, comm, status, request)* 异步接受消息
    - *MPI_Wait (request, status)*：等待非阻塞数据传输完成
    - *MPI_Test (request, flag, status)*：检查是否异步数据传输确实完成

MPI编程示例

- 计算大数组元素的开平方之和
- 设系统中共有5个进程，进程号：0，1，2，3，4
- 0号进程作主节点，负责分发数据，不参加子任务计算
- 1-4号进程作为子节点从主进程接受数组数据：
  - #1: data[0,4,8,...]
  - #2: data[1,5,9,...]
  - #3: data[2,6,10,...]
  - #4: data[3,7,11,...]

  } 各自求开平方后累加=>本地SqrtSum

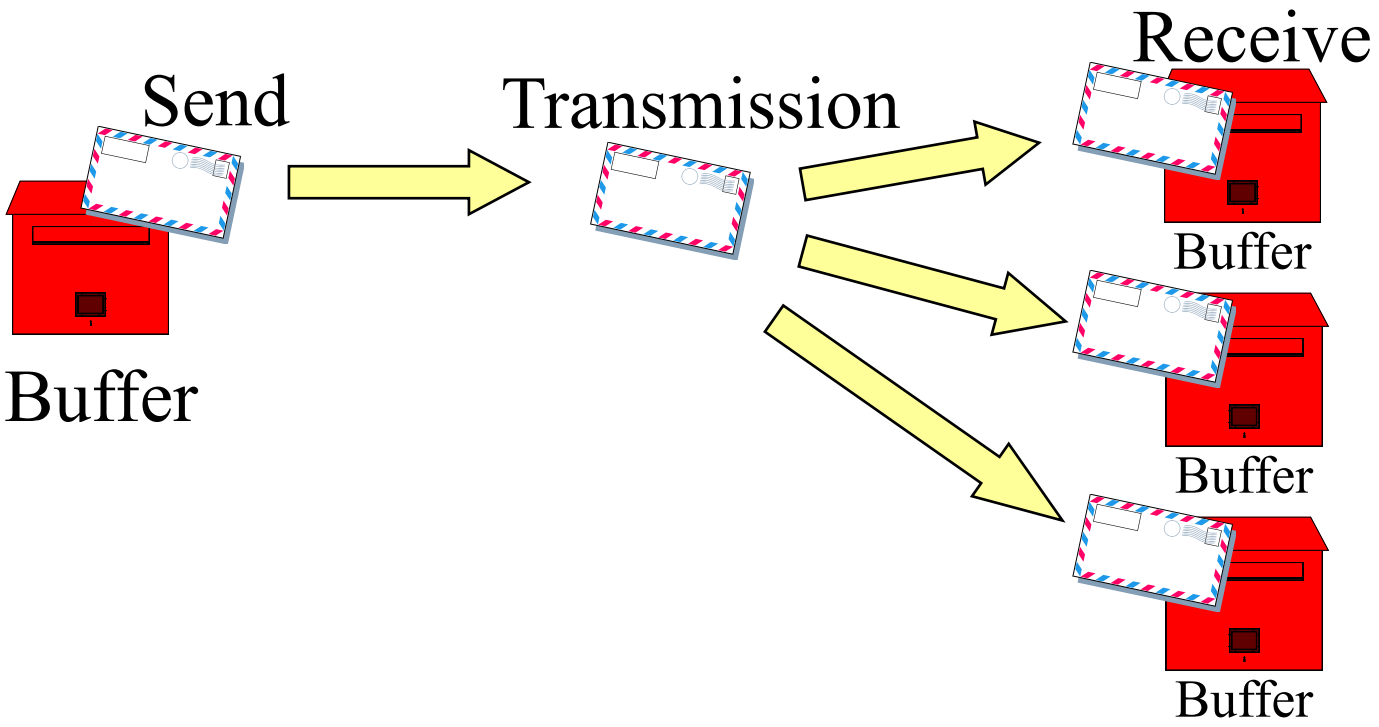
- #0: Sqrt Sum = ∑ 各子进程的SqrtSum

MPI编程示例


```
#include <stdio.h>   #include <mpi.h>   #include <math.h>   #define N 1002
int main(int argc, char** argv)
{
int myid, P, source, C=0; double  data[N],  SqrtSum=0.0;
MPI_Status status;   char message[100];
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);   - -numprocs; /*数据分配时除去0号主节点*/
if (myid== 0){   /*0号主节点，主要负责数据分发和结果收集*/
   for (int  i = 0;  i < N; ++i )  /*数据分发: 0,*/
        MPI_Send(data[i], 1, MPI_DOUBLE, i%numprocs+1, 1 ,MPI_COMM_WORLD);
   for (int  source = 1;  source <= numprocs;  ++source)  /*结果收集*/
   {  MPI_Recv(&d, 1, MPI_DOUBLE, source, 99, MPI_COMM_WORLD, &status);  SqrtSum += d; }
}else
{   for (i = myid-1; i < N; i=i+numprocs)  /*各子节点接受数据计算开平方，本地累加*/
    {    MPI_Recv(&d, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &status); SqrtSum+=sqrt(d); }
     MPI_Send(SqrtSum, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD); /*本地累加结果送回主节点*/
}
printf("I am process %d. I recv total %d from process 0, and SqrtSum=%f.\n", myid, C, SqrtSum);
MPI_Finalize();
}
```

节点集合通信接口

- 提供一个进程与多个进程间同时通信的功能



节点集合通信接口

- 三种类型的集合通信功能

1.同步(Barrier)

MPI_Barrier： 设置同步障使所有进程的执行同时完成

2. 数据移动(Data movement)

MPI_BCAST: 一对多的广播式发送

MPI_GATHER： 多个进程的消息以某种次序收集到一个进程

MPI_SCATTER： 将一个信息划分为等长的段依次发送给其它进程

3. 数据规约(Reduction)

MPI_Reduce： 将一组进程的数据按照指定的操作方式规约到一起并传送给一个进程

MPI Routines and Constants：https://www.mpich.org/static/docs/latest/

节点集合通信接口

- 数据规约操作

  将一组进程的数据按照指定的操作方式规约到一起并传送给一个进程

  MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)

- 其中规约操作op可设为下表定义的操作之一：

MPI_MAX	求最大值	MPI_MIN	求最小值
MPI_SUM	求和	MPI_PROD	求积
MPI_LAND	逻辑与	MPI_BAND	按位与
MPI_LOR	逻辑或	MPI_BOR	按位或
MPI_LXOR	逻辑异或	MPI_BXOR	按位异或
MPI_MAXLOC	最大值和位置	MPI_MINLOC	最小值和位置

MPI编程示例

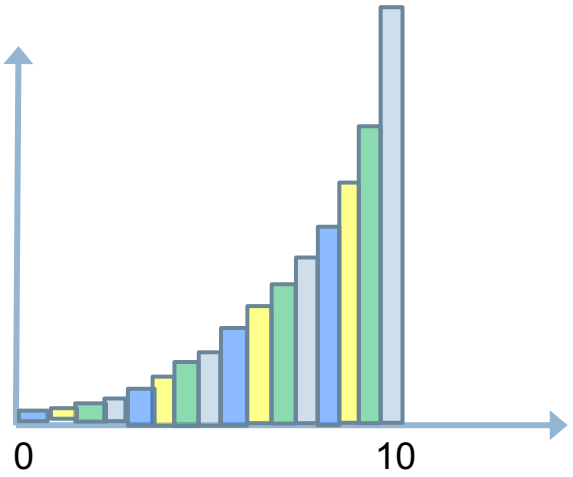
规约操作编程示例-计算积分

根据微积分原理，任一函数f(x)在区间[a,b]上的积分是由各个x处的y值为高构成的N个小矩形(当N趋向无穷大时的）面积之和构成。因此，选取足够大的N可近似计算积分。

设$y=x^2$，求其在[0,10]区间的积分。

先把[0,10]分为N个小区间，则对每个x取值对应小矩形面积为：y*10/N

求和所有矩形面积，当N足够大时即为近似积分值。



我们用n个节点来分工计算N个区间的面积。如图所示，根据总结点数目，每个节点将求和一个颜色的小矩形块。

MPI编程示例


```
#define N 100000000
#define  a 0
#define  b 10
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include “mpi.h”
int main(int argc, char** argv)
{
  int myid,numprocs;
  int i;
  double local=0.0, dx=(double)(b-a)/N; /* 小矩形宽度 */
  double inte, x;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
```

MPI编程示例


```
for(i=myid;i<N;i=i+numprocs)  /* 根据节点数目将N个矩形分为图示的多个颜色组 */
  {  /* 每个节点计算一个颜色组的矩形面积并累加*/
     x = a + i*dx +dx/2; /*  以每个矩形的中心点x值计算矩形高度  */
     local +=x*x*dx;  /* 矩形面积 = 高度x宽度=y*dx  */
  }
  MPI_Reduce(&local,&inte,1,MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
  if(myid==0)                          /* 规约所有节点上的累加和并送到主节点0 */
  { /* 主节点打印累加和*/
     printf("The integal of x*x in region [%d,%d] =%16.15f\n", a, b, inte);
  }
  MPI_Finalize();
}
```

The integal of x*x in region[0, 10] = 333.33345

单机运行：安装MPI

- 操作系统：Ubuntu 14.04（仅供参考）
- 安装MPI编译和运行环境：OpenMPI

```
1   $ sudo apt-get install libopenmpi-dev
```

之后如果不能使用`mpirun`命令，则还需要安装：

```
1   $ sudo apt-get install openmpi-bin
```

- 测试运行

多机运行：搭建集群

- 准备工作
  - 配置IP地址和Host Names

修改每个节点的 `/etc/hosts` 文件，添加两个节点的IP映射，查看IP使用 `ifconfig` 命令，如果两个节点的主机名分别为host1和host2，IP分别为 `192.168.1.1` 和 `192.168.1.2`，在该文件中添加;

```
1  192.168.1.1 host1
2  192.168.1.2 host2
```

  - 配置NFS建立共享磁盘空间（可选，略）
  - 关闭防火墙
  - 安装并配置sshd

初始机器可能没有安装ssh server服务，需要手动添加：

```
1  $ sudo apt-get install openssh-server
```

多机运行：搭建集群

- ssh免密访问

$ ssh-keygen -t rsa

在运行过程中，会提示你输入这个输入那个，不用管那么多，直接回车就完了。等运行结束后，进入.ssh目录，你会看到公钥和私钥文件：

```
$ cd ~/.ssh
$ ls
id_rsa   id_rsa.pub   (可能还有其他乱七八糟的东西)
```

其中id_rsa就是私钥，id_rsa.pub就是公钥，现在我们需要把各个节点上的公钥都集中发送到一个节点上，来制作授权文件（authorized_keys）：

```
$ scp id_rsa.pub User@MainNode:/path/to/yours/id_rsa.pub-X
```

其中有色字体需要你根据实际情况进行调整，这里我们讲所有的公钥发送到了MainNode，然后我们在MainNode上用这些公钥合成一个authorized_keys：

```
$ cp ~/.ssh/id_rsa.pub authorized_keys
$ cat /path/to/yours/id_rsa.pub-X >> authorized_keys
```

其中第2条语句需要反复多次，讲所有的公钥都放入authorized_keys中，这样就做好了authorized_keys文件。然后我们分别把他拷到各个机器的.ssh目录中：

```
$ cp authorized_keys ~/.ssh/
$ scp authorized_keys User@SomeNode:/home/User/.ssh/
```

如果你现在试一下ssh很可能发现他们仍旧不好用，别急，我的话还没有说完呢。这个东西对权限的要求很严格，所以我们需要更改一下必要的文件的权限（所有节点都要更改）：

```
$ chmod 755 ~
$ chmod 755 ~/.ssh
$ chmod 600 ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/id_rsa
$ chmod 644 ~/.ssh/id_rsa.pub
```

多机运行

- 测试 hellompi

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define  MASTER     0
5
6  int main (int argc, char *argv[])
7  {
8          int   numtasks, taskid, len;
9      char hostname[MPI_MAX_PROCESSOR_NAME];
10
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
13     MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
14     MPI_Get_processor_name(hostname, &len);
15     printf ("Hello from task %d on %s!\n", taskid, hostname);
16     if (taskid == MASTER)
17        printf("MASTER: Number of MPI tasks is: %d\n",numtasks);
18     MPI_Finalize();
19     return 0 ;
20 }
```

多机运行

编译及运行

使用`mpicc`命令编译：

```
1  $ mpicc hellompi.c -o hellompi
```

运行：

```
1  $ mpirun -np 10 -host host1,host2 hellompi
```

运行之前需要保证每台机器上都有`hellompi`，一般拷贝到HOME目录下。

基于Docker构建分布式环境

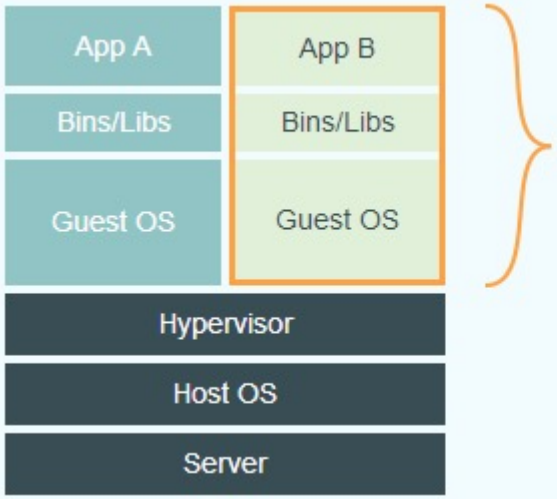
- 建立合适的Image
  - 拉取基础系统镜像
  - 安装必备软件包
    - gcc, g++, gfortran, mpich, ssh…
    - vim, cat…
  - 生成ssh密钥
- 生成若干Container
  - 挂载宿主机目录
  - ssh互通
  - 对外端口

Docker简介

- Docker的英文本意是码头工人，也就是搬运工，这种搬运工搬运的是集装箱（Container）
- 集装箱里面装的是任意类型的App，Docker把App（叫Payload）装在Container内，通过Linux Container技术的包装将App变成一种标准化的、可移植的、自管理的组件，这种组件可以在你的laptop上开发、调试、运行，最终非常方便和一致地运行在production环境下。

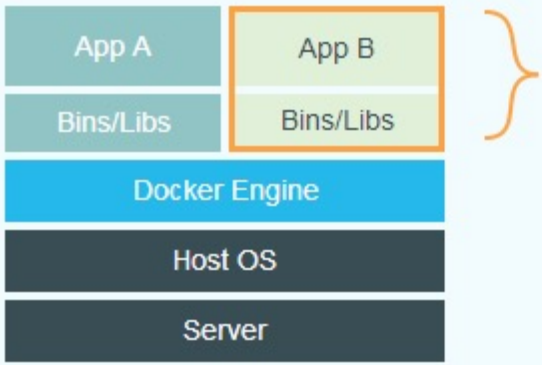
Docker简易教程：https://yeasy.gitbooks.io/docker_practice/（仅供参考）

Docker



Virtual Machines

Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.



Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

为什么用Docker?

- 更高效地利用系统资源
- 更快速的启动时间
- 一致的运行环境
- 持续交付和部署
- 更轻松的迁移
- 更轻松的维护和扩展

Docker VS. VM

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

Docker基本概念

- 镜像（image）
  - Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。
- 容器（container）
  - 容器是镜像运行时的实体，容器的实质是进程，运行于属于自己的独立命名空间
- 仓库（repository）
  - Docker Registry：包含多个仓库，每个仓库可有多个标签（tag），每个tag对应一个镜像

Docker基本概念

- 优势
  - 一生万物的image-container关系
  - 虚拟内网环境，所有容器处于同一局域网内
  - 宿主机目录挂载，不需要NFS
  - 空间占用相对较小
- 劣势
  - 对Kernel（内核）的操作受限（禁止/重置）

实验报告要求

- 1. 代码
- 2. 运行截图
- 3. 问题总结及解决方案
- 4. 其它思考

- 建议：使用Git仓库（Github或Gitee）托管代码