Q1.介绍阻塞发送MPI\_SEND/阻塞接收MPI\_RECV与非阻塞发送MPI\_ISEND/非阻塞接收MPI\_IRECV的区别

阻塞的消息传递

阻塞发送（MPI\_SEND）：在消息确实已经发送出去（到消息缓冲区）后，才允许进程继续执行下一语句。

阻塞接收(MPI\_RECV)：进程处于挂起状态，直到可以在消息缓冲区确切地接收到消息后，才允许进程继续执行下一语句。

非阻塞的消息传递

非阻塞发送(MPI\_ISEND)：发送原语通知系统将要发送的消息在消息缓冲区中后，即可返回。发送进程可继续执行后续工作， 无须等待系统真正发送消息。

非阻塞接收(MPI\_IRECV)：接收原语不管消息缓冲区中是否已有发送原语发送的消息，都将返回。

也就是说，在非阻塞的消息传递中，当消息被确切地发出或收到时，系统将用中断信号通知发送方或接受方。在此之前，它们可以周期性地查询、暂时挂起或执行其它计算，以实现计算与通信的重叠。

Q2.介绍什么是归约操作。

归约操作是指在分布在不同进程中的数据间进行交互的运算，常用的运算有求和、求最大或最小值等。

在MPI中，归约函数如下定义：

int MPI\_Reduce(void\* sendbuf, void\* recvbuf, int count,

MPI\_Datatype datatype, MPI\_Op op, int root,

MPI\_Comm comm)

参数说明如下：

MPI\_REDUCE(sendbuf,recvbuf,count,datatype,op,root,comm)

sendbuf 发送消息缓冲区的起始地址(可变)

recvbuf 接收消息缓冲区中的地址(可变,仅对于根进程)

count 发送消息缓冲区中的数据个数(整型)

datatype 发送消息缓冲区的元素类型(句柄)

op 归约操作符(句柄)

root 根进程序列号(整型)

comm 通信子(句柄)

MPI\_REDUCE将组内每个进程输入缓冲区中的数据按op操作组合起来,并将其结果返回到序列号为root的进程的输出缓冲区中。

常用的op操作有：

名字 含义

MPI\_MAX 最大值

MPI\_MIN 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 最大值且相应位置

MPI\_MINLOC 最小值且相应位置

对于OpenMP，归约过程由一个子句Reduction完成：

reduction(operator : val1,val2…… )      每个线程根据reduction（op: sum）的声明算出自己的sum，然后再将每个线程的sum进行op运算，最后将数据归约（含初始值）。

其中operator以及约定变量的初始值如下：

运算符            数据类型                  默认初始值

+                   整数、浮点               0

-                    整数、浮点               0

\*                   整数、浮点               1

&                   整数                        所有位均为1

|                    整数                        0

^                   整数                        0

&&                 整数                        1

||                   整数                        0

一个例子：

#pragma omp parallel for reduction(+:sum)

for (int i = 1; i <= 40000; i++)

{

ff(&sum);

}

Q3.介绍什么是栅障操作。

并行计算中，进程之间独立执行大量的计算块，然后他们在同步屏障的地方互相等待，执行发送/接收消息，等到所有相关进程都到达栅障，处理完信息后，继续他们的程序执行。

在MPI中，barrier操作如下函数实现：

int MPI\_Barrier(MPI\_Comm);

表示阻止调用直到communicator中所有进程完成调用。用于一个通信子中所有进程的同步，调用函数时进程将处于等待状态直到通信子中所有进程都调用了该函数后才继续执行。

例子：

MPI\_Init(&agc,&agv);

MPI\_Comm\_size(MPI\_COMM\_ WORLD,&comm\_size);

MPI\_Comm\_rank(MPI\_COMM\_ WORLD,&my\_rank);

……

MPI\_Barrier(MPI\_COMM\_WORLD);

……

MPI\_Finalize()；

在OpenMP中，barrier的实现由编译指导完成：

#pragma omp barrier

子进程执行到该语句时陷入等待，直到所有的进程都执行到该步时释放。

例子：

#pragma omp parallel

    {

……

        #pragma om barrier

……

    }

Q4.

片段是错误的。原因在于将MPI\_Bcast放在了if(mtpid == 0)的循环内部，导致Bcast执行的唯一条件是mypid == 0，这样的话只执行了数据的发送，当mypid！=0时，Bcast中数据的接收始终没有被操作，从而导致错误（死锁）。

Q5.用OpenMP实现m\*p 的矩阵A和p\*n的矩阵B的乘法。

#include<stdio.h>

#include<omp.h>

#define n 1000

#define m 1000

#define p 500

double A[m][p],B[p][n],C[m][n];

int main()

{

int i,j,k;

for (int i = 0; i < m; i++)

for (int j = 0; j < p; j++)

{

A[i\*p + j] = i + j; //A[i][j]

}

for (int i = 0; i < p; i++)

for (int j = 0; j < n; j++)

{

B[i\*n + j] = i + j; //B[i][j]

}

#pragma omp parallel for shared(A,B,C) private(i,j,k)

for(i = 0;i < m; i++)

for(j = 0; j <n; j++)

{

C[i][j] = 0;

for(k = 0; k< p; k++)

C[i][j] += A[i][k]\*B[k][j];

}

return 0;

}

Q6.用MPI实现m\*p 的矩阵A和p\*n的矩阵B的乘法。

#include<mpi.h>

#include<stdio.h>

#include <iostream>

#include<math.h>

#pragma comment(lib,"mpi.lib")

#define n 1000

using namespace std;

int main(int argv, char \*argc[])

{

int rank, p, a.m.p;

MPI\_Init(&argv, &argc);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);

MPI\_Status status;

if (p!=1)

a = m / (p - 1);

if (rank == 0)

{

int\* A = new int[m\*p];

int\* B = new int[p\*n];

int\* C = new int[m\*n];

for (int i = 0; i < m; i++)

for (int j = 0; j < p; j++)

{

A[i\*p + j] = i + j; //A[i][j]

}

for (int i = 0; i < p; i++)

for (int j = 0; j < n; j++)

{

B[i\*n + j] = i + j; //B[i][j]

}

if (p == 1)//只有一个进程

{

double tb, te;

for (int i = 0; i < m; i++)

for (int j = 0; j < n; j++)

{

C[i\*n + j] = 0; //C[i][j]

for (int k = 0; k < n; k++)

{

C[i\*n + j] = A[i\*n + k] \* B[k\*n + j];

}

}

}

if (p != 1)//有多个从进程

{

double tb, te;

for (int i = 0; i < p-1; i++)

{

MPI\_Send(&A[0+0], m\*p, MPI\_INT, i+1, 1, MPI\_COMM\_WORLD);//每个发送 a行，a\*n大小的数据

MPI\_Send(&B[0+0], p\*n, MPI\_INT, i+1,2, MPI\_COMM\_WORLD);

}

for (int i =0; i < p-1; i++)

MPI\_Recv(&C[i\*a+0], a\*n, MPI\_INT, i+1,3, MPI\_COMM\_WORLD, &status);//每个接受 a行，a\*n大小的数据

}

delete[] A;

delete[] B;

delete[] C;

}

if (p != 1)

if (rank != 0){

int\* A = new int[n\*n];

int\* B = new int[n\*n];

int\* C = new int[n\*n];

MPI\_Recv(&A[0+0], m\*p, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, &status);//从A[0][0]和B[0][0]开始接受

MPI\_Recv(&B[0+0], p\*n, MPI\_INT,0, 2, MPI\_COMM\_WORLD, &status);

for (int i =a\*(rank-1); i < (a\*(rank)); i++)//按照行间隔分，每个cpu计算自己的a行

for (int j = 0; j < n; j++)

{

C[i\*n + j] = 0; //C[i][j]

for (int k = 0; k < p; k++)

{

C[i\*n + j] = A[i\*n + k] \* B[k\*n + j];

}

}

{//向rank=0发送自己的那a行C,大小是a\*n

//int \* sendptr = &(C[a\*(rank - 1)+0]);

MPI\_Send(&C[a\*(rank - 1) + 0], a\*n, MPI\_INT, 0,3, MPI\_COMM\_WORLD);//起始地址是C[rank-1][0],大小是a\*n

}

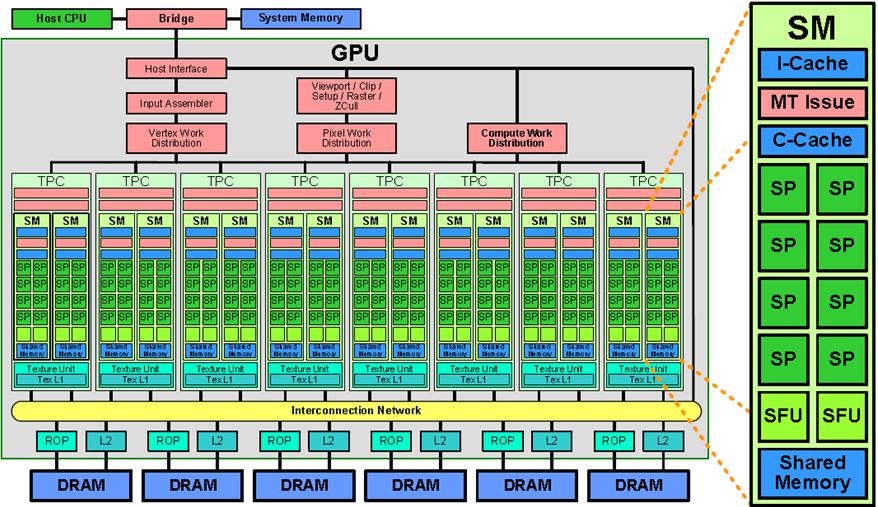
}

MPI\_Finalize();

return 0;

}

Q7.分析一款GPU的存储层次



GPU的存储系统包括 register，shared memory，texture memory， local memory， global memory



每个thread  都有自己的一份register 和local memory  的空间。同一个block 中的每个thread 则有共享的一份share memory。此外，所有的thread（包括不同block 的thread）都共享一份global memory、constant memory、和texture memory。不同的grid 则有各自的global memory、constant memory 和texture memory。

#寄存器

与CPU不同，GPU的每个SM（流多处理器）有成千上万个寄存器，在GPU技术简介中已经提到，SM类似于CPU的核，每个SM拥有多个SP（流处理器），所有的工作都是在SP上处理的，GPU的每个SM可能有8~192个SP，这就意味着，SM可同时运行这些数目的线程。

寄存器是每个线程私有的，并且GPU没有使用寄存器重命名机制，而是致力于为每一个线程都分配真实的寄存器，CUDA上下文切换机制非常高效，几乎是零开销。当然，这些细节对程序员是完全透明的。

#Local memory\*\*\*

Local memory和寄存器类似，也是线程私有的，访问速度比寄存器稍微慢一点。

事实上，是由编译器在寄存器全部使用完的时候自动分配的。

#共享内存

每个线程块都有一个共享内存，该线程块中的线程都可以读取该内存，其他线程块的线程无法访问该共享内存。共享内存帮助同一线程块的线程通信与协作。并且共享内存缓存区实在物理GPU上，而不是其他与GPU相连的设备上，访问效率更高。

#常量内存

常量内存通过\_\_constant\_\_来修饰变量，被限制为只读。

1.对常量内存的单次读操作可以广播到该线程的半线程数中，减少15读操作。

2.常量内存的数据将缓存起来，因此对相同地址的连续读操作不会产生额外通信量。