


# CA\_Assignment10

 Assign	
 tag	homework
 姓名	周鹏宇
 学号	2019K8009929039

1. 请介绍 MPI 中阻塞发送 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。

**在阻塞方式中**，它必须等到消息从本地送出之后才可以执行后续的语句，保证了缓冲区等资源可再用;**对于非阻塞方式**，它无须等到消息从本地送出就可执行后续的语句，从而允许通信和计算的重叠，但非阻塞调用的返回并不保证资源的可再用性。

2. 请介绍什么是归约(Reduce)操作，MPI 和 OpenMP 中分别采用何种函数或者子句来实现归约操作。

**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

MPI：

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
//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.输入缓冲区由参数sendbuf·count和datatype定义;
//输出缓冲区由参数recvbuf·count和datatype定义;两者的元素数目和类型都相同.
//所有组成员都用同样的参数count·datatype·op·root和comm来调用此例程,
//因此所有进程都提供长度相同、元素类型相同的输入和输出缓冲区.
//每个进程可能提供一个元素或一系列元素,组合操作针对每个元素进行
```

OpenMP：

```
//reduction子句：用来指定一个或多个变量是私有的，并且在并行处理结束后
//这些变量要执行指定的归约运算，并将结果返回给主线程同名变量；
reduction(+:x,y,z)
reduction(+:array[:])
```

3. 请介绍什么是栅障(Barrier)操作, MPI 和 OpenMP 中分别采用何种函数或者命令来实现

栅障。

栅障是用于线程同步的一种方法, 线程遇到栅障时必须等到直到所有的组成员都到达栅障后才能继续运行。

MPI 中使用函数 MPI\_Barrier 实现栅障。而 OpenMP 中使用 barrier 编译直到语句实现栅障。

4. 下面的 MPI 程序片段是否正确?请说明理由。假定只有 2 个进程正在运行且 mypid 为每个进程的进程号。

```
If(mypid==0) {  
    MPI_Bcast(buf0,count,type,0,comm,ierr);  
    MPI_Send(buf1,count,type,1,tag,comm,ierr);  
} else {  
    MPI_Recv(buf1,count,type,0,tag,comm,ierr);  
    MPI_Bcast(buf0,count,type,0,comm,ierr);  
}
```

不正确。

首先, MPI\_Bcast 是阻塞式进程, 在if语句中, 由pid=0的进程发出广播并将自身阻塞, 等待其他进程收到广播后解锁, 在else语句中, 则由另一个进程 (pid!=0) 接收来自 pid=0的进程的广播, 换言之, 只有else块的第二句执行后, if块的第一句才会解除阻塞, 但else块的第一句MPI\_Recv依赖于if块第二句MPI\_Send语句发送的信息, 但由于 0进程被阻塞, 故无法执行到MPI\_Send, 形成了**死锁**。

5. 矩阵乘是数值计算中的重要运算。假设有一个  $m \times p$  的矩阵 A, 还有一个  $p \times n$  的矩阵 B。令

C 为矩阵 A 与 B 的乘积, 即  $C=AB$ 。表示矩阵在(i,j)位置处的值, 则  $0 \leq i \leq m-1, 0 \leq j \leq n-1$ 。请

采用 OpenMP, 将矩阵 C 的计算并行化。假设矩阵在存储器中按行存放。

```
#include <stdio.h>  
#include <stdlib.h>  
#define M 100  
#define N 200  
#define O 300  
  
int main() {  
    int A[M][N];  
    int B[N][O];  
    int C[O][M];  
    for (int i = 0; i < M; ++i) {  
        for (int j = 0; j < N; ++j) {
```

```

        *A[i * N + j] = rand()%10;
    }
}
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < O; ++j) {
        *B[i * O + j] = rand()%10;
    }
}
#pragma omp parallel for shared(A, B, C) private(i, j, k)
for (int i = 0; i < M; ++i) {
    for (int j = 0; j < O; ++j) {
        for (int k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
return 0;
}

```

6. 请采用 MPI 将上题中矩阵 C 的计算并行化，并比较 OpenMP 与 MPI 并程序的特点。

```

#include <string.h>
#include <iostream>
#include <chrono>
#include <random>
#include <Windows.h>
#include <mpi.h>
#include <omp.h>

using namespace std;
using namespace chrono;

#define MAT_SIZE 2000
#define MAX_NUM 1000
#define NUM_THREADS 8

enum tags {
    init_mat_a,
    init_mat_b,
    send_mat_a,
    send_mat_b,
    calc_mat_c
};

void MainProc(const int proc_cnt) {
    MPI_Status status;

    const auto timeNow = time(nullptr);
    default_random_engine rand(timeNow);
    const uniform_int_distribution<unsigned> dist(1, MAX_NUM);

    cout << "Threads per process: " << omp_get_max_threads() << endl;

    const auto start = system_clock::now();

    const auto a = new int*[MAT_SIZE], b = new int*[MAT_SIZE], c = new int*[MAT_SIZE];

```

```

#pragma omp parallel for shared(a) shared(b)
for (auto i = MAT_SIZE / proc_cnt * (proc_cnt - 1); i < MAT_SIZE; ++i) {
    a[i] = new int[MAT_SIZE];
    b[i] = new int[MAT_SIZE];
    c[i] = new int[MAT_SIZE]{0};
    for (auto j = 0; j < MAT_SIZE; ++j) {
        a[i][j] = dist(rand);
        b[i][j] = dist(rand);
    }
}

for (auto pid = 0; pid < proc_cnt - 1; ++pid) {
    for (auto i = MAT_SIZE / proc_cnt * pid; i < MAT_SIZE / proc_cnt * (pid + 1); ++i) {
        a[i] = new int[MAT_SIZE];
        b[i] = new int[MAT_SIZE];
        MPI_Recv(a[i], MAT_SIZE, MPI_INT, pid, init_mat_a, MPI_COMM_WORLD, &status);
        MPI_Recv(b[i], MAT_SIZE, MPI_INT, pid, init_mat_b, MPI_COMM_WORLD, &status);
    }
}

for (auto pid = 0; pid < proc_cnt - 1; ++pid) {
    for (auto i = 0; i < MAT_SIZE; ++i) {
        MPI_Send(a[i], MAT_SIZE, MPI_INT, pid, send_mat_a, MPI_COMM_WORLD);
        MPI_Send(b[i], MAT_SIZE, MPI_INT, pid, send_mat_b, MPI_COMM_WORLD);
    }
}

#pragma omp parallel for shared(a) shared(b) shared(c)
for (auto i = MAT_SIZE / proc_cnt * (proc_cnt - 1); i < MAT_SIZE; ++i) {
    for (auto j = 0; j < MAT_SIZE; ++j) {
        for (auto k = 0; k < MAT_SIZE; ++k) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}

for (auto pid = 0; pid < proc_cnt - 1; ++pid) {
    for (auto i = MAT_SIZE / proc_cnt * pid; i < MAT_SIZE / proc_cnt * (pid + 1); ++i) {
        c[i] = new int[MAT_SIZE];
        MPI_Recv(c[i], MAT_SIZE, MPI_INT, pid, calc_mat_c, MPI_COMM_WORLD, &status);
    }
}

cout << "Processed in " << proc_cnt << " process(es) uses "
    << duration_cast<milliseconds>(system_clock::now() - start).count()
    << " milliseconds." << endl;

for (auto i = 0; i < MAT_SIZE; ++i) {
    delete[] a[i];
    delete[] b[i];
    delete[] c[i];
}
delete[] a;
delete[] b;
delete[] c;
}

void SubProc(const int proc_cnt) {
    MPI_Status status;

```

```

const auto timeNow = time(nullptr);
default_random_engine rand(timeNow);
const uniform_int_distribution<unsigned> dist(1, MAX_NUM);

auto a = new int*[MAT_SIZE / proc_cnt];
auto b = new int*[MAT_SIZE / proc_cnt];
#pragma omp parallel for shared(a) shared(b)
for (auto i = 0; i < MAT_SIZE / proc_cnt; ++i) {
    a[i] = new int[MAT_SIZE];
    b[i] = new int[MAT_SIZE];
    for (auto j = 0; j < MAT_SIZE; ++j) {
        a[i][j] = dist(rand);
        b[i][j] = dist(rand);
    }
}

for (auto i = 0; i < MAT_SIZE / proc_cnt; ++i) {
    MPI_Send(a[i], MAT_SIZE, MPI_INT, proc_cnt - 1, init_mat_a, MPI_COMM_WORLD);
    MPI_Send(b[i], MAT_SIZE, MPI_INT, proc_cnt - 1, init_mat_b, MPI_COMM_WORLD);
}

for (auto i = 0; i < MAT_SIZE / proc_cnt; ++i) {
    delete[] a[i];
    delete[] b[i];
}
delete[] a;
delete[] b;

a = new int*[MAT_SIZE];
b = new int*[MAT_SIZE];
const auto c = new int*[MAT_SIZE / proc_cnt];
for (auto i = 0; i < MAT_SIZE; ++i) {
    a[i] = new int[MAT_SIZE];
    b[i] = new int[MAT_SIZE];
    MPI_Recv(a[i], MAT_SIZE, MPI_INT, proc_cnt - 1, send_mat_a, MPI_COMM_WORLD, &status);
    MPI_Recv(b[i], MAT_SIZE, MPI_INT, proc_cnt - 1, send_mat_b, MPI_COMM_WORLD, &status);
}

#pragma omp parallel for shared(a) shared(b) shared(c)
for (auto i = 0; i < MAT_SIZE / proc_cnt; ++i) {
    c[i] = new int[MAT_SIZE]{0};
    for (auto j = 0; j < MAT_SIZE; ++j) {
        for (auto k = 0; k < MAT_SIZE; ++k) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}

for (auto i = 0; i < MAT_SIZE / proc_cnt; ++i) {
    MPI_Send(c[i], MAT_SIZE, MPI_INT, proc_cnt - 1, calc_mat_c, MPI_COMM_WORLD);
}

for (auto i = 0; i < MAT_SIZE; ++i) {
    delete[] a[i];
    delete[] b[i];
    if (i < MAT_SIZE / proc_cnt) {
        delete[] c[i];
    }
}
delete[] a;
delete[] b;

```

```

    delete[] c;
}

int main(int argc, char* argv[]) {
    int procCnt, procId;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &procId);
    MPI_Comm_size(MPI_COMM_WORLD, &procCnt);

    omp_set_num_threads(NUM_THREADS);

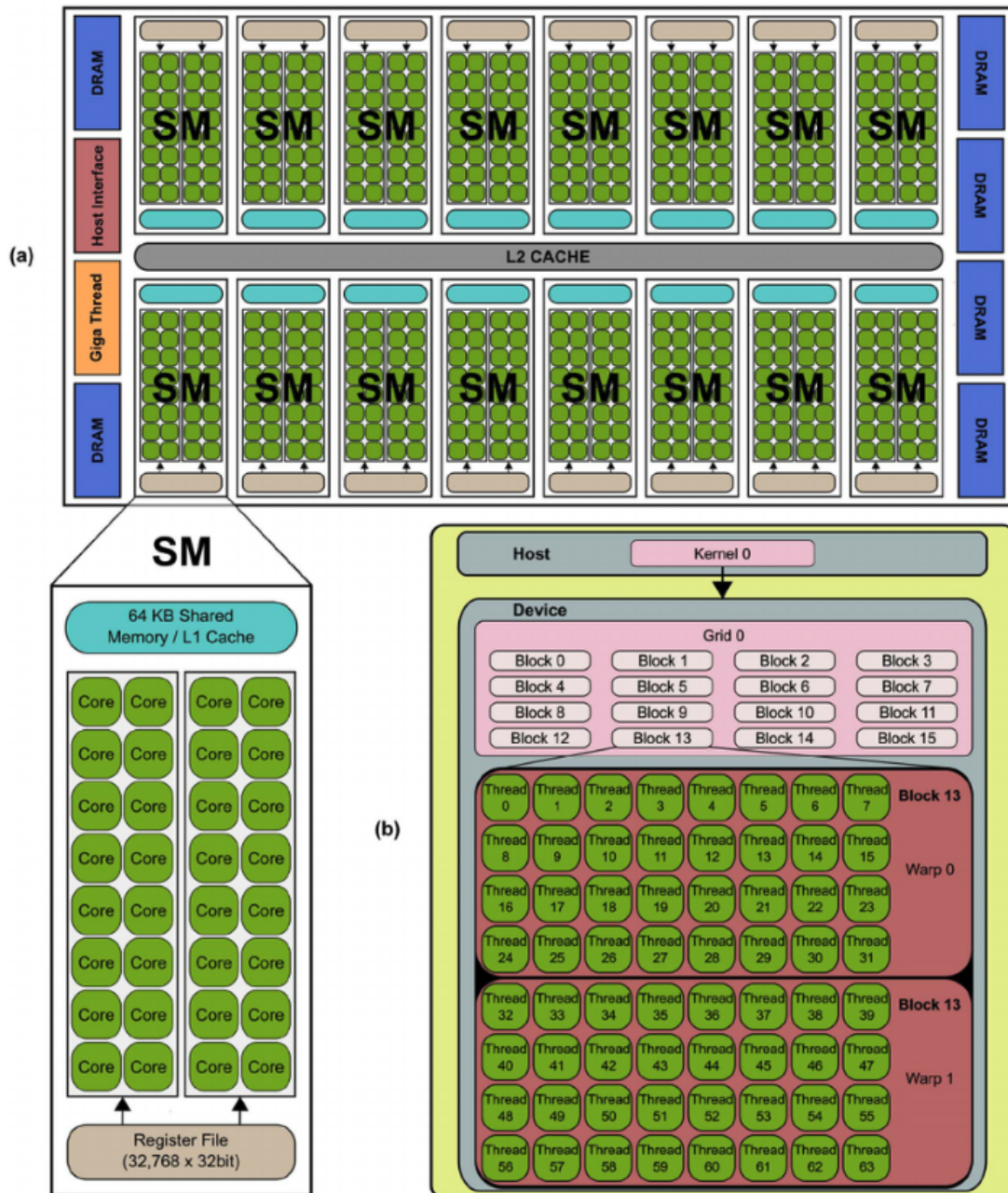
    if (procId == procCnt - 1) {
        MainProc(procCnt);
    }
    else {
        SubProc(procCnt);
    }

    MPI_Finalize();
}

```

---

## 7. 分析一款 GPU 的存储层次。



Typical NVIDIA GPU architecture.

The GPU is comprised of a set of Streaming MultiProcessors (SM). Each SM is comprised of several Stream Processor (SP) cores, as shown for the NVIDIA's Fermi architecture (a). The GPU resources are controlled by the programmer through the CUDA programming model, shown in (b).