这四种通信方式都是属于点对点通信中标准模式，MPI\_Recv和MPI\_Send为阻塞机制，必须等待消息从本地发送出去之后才会继续执行后续的语句，可以保证缓冲区等资源可再用；而MPI\_Irecv和MPI\_Isend为非阻塞机制，无须等到消息从本地送出就可以执行后续的语句，这样可以允许通信和计算的重叠，但与此同时非阻塞调用的返回并不保证资源的可再用性。

归约（Reduce）是一种并行计算的通信操作，用于在多个进程之间合并数据。MPI的归约操作允许多个进程将各自的数据合并为一个全局结果，这对于并行计算中的全局汇总和聚合操作非常有用。

MPI的归约操作通常涉及一个运算，如求和、求积、找最大值等。这个运算在每个进程上独立地执行，然后通过MPI的归约操作将各个进程的局部结果合并成一个全局结果。这有助于减少通信的开销。

在MPI中，归约操作通常使用 MPI\_Reduce 和MPI\_Allreduce函数来实现，在OpenMP中，归约操作通常使用 reduction 子句来实现。OpenMP会自动处理并行线程之间的同步和归约操作。

3、

栅障（Barrier）操作是一种同步机制，它用于确保在多线程或多进程并行计算中，所有参与的线程或进程都达到了某个同步点，然后再一起继续执行。栅障操作能够协调不同线程或进程之间的执行顺序，防止并发执行引起的竞态条件问题。

在 MPI 中，栅障操作通常使用 MPI\_Barrier 函数来实现。MPI\_Barrier 会阻塞调用该函数的所有进程，直到所有参与的进程都调用了该函数，然后所有进程一起解除阻塞，继续执行。

在 OpenMP 中，栅障操作通常使用 omp barrier 命令来实现。omp barrier 会阻塞调用该命令的所有线程，直到所有参与的线程都调用了该命令，然后所有线程一起解除阻塞，继续执行。

4、

这个程序会发生死锁，原因是进程0需要等进程1执行完MPI\_Bcast后才会进行MPI\_Send,但是进程1因为阻塞不会进行发送，与此同时进程0因为收不到进程1发送的数据，也会进行阻塞从而也不能进行MPI\_Bcast。可以将else中的两个语句进行交换就可以解决这个问题。

5、

#include <omp.h>

void matrix\_multiply(int m, int p, int n, double \*A, double \*B, double \*C) {

#pragma omp parallel for collapse(2)

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

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

C[i \* n + j] = 0.0;

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

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

}

}

}

}

6、

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

void matrix\_multiply(int m, int p, int n, double \*A, double \*B, double \*C, MPI\_Comm comm) {

int rank, num\_procs, line;

MPI\_Comm\_rank(comm, &rank);

MPI\_Comm\_size(comm, &num\_procs);

line = m / num\_procs;

double \*local\_A = (double \*)malloc(sizeof(double) \* line \* p);

double \*local\_C = (double \*)malloc(sizeof(double) \* line \* n);

// Scatter matrix A to all processes

MPI\_Scatter(A, line \* p, MPI\_DOUBLE, local\_A, line \* p, MPI\_DOUBLE, 0, comm);

// Broadcast matrix B to all processes

MPI\_Bcast(B, p \* n, MPI\_DOUBLE, 0, comm);

// Perform local matrix multiplication

for (int i = 0; i < line; i++) {

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

local\_C[i \* n + j] = 0;

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

local\_C[i \* n + j] += local\_A[i \* p + k] \* B[k \* n + j];

}

}

}

// Gather local results back to the root process

MPI\_Gather(local\_C, line \* n, MPI\_DOUBLE, C, line \* n, MPI\_DOUBLE, 0, comm);

free(local\_A);

free(local\_C);

}

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

int m = 1000;

int p = 1000;

int n = 1000;

double \*A, \*B, \*C;

int ID;

MPI\_Init(&argc, &argv);

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

A = (double \*)malloc(sizeof(double) \* m \* p);

B = (double \*)malloc(sizeof(double) \* p \* n);

C = (double \*)malloc(sizeof(double) \* m \* n);

if (ID == 0) {

// Initialize matrices A and B

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

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

A[i \* p + j] = 1.0;

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

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

B[i \* n + j] = 1.0;

}

MPI\_Barrier(MPI\_COMM\_WORLD);

double start\_time = MPI\_Wtime();

// Perform matrix multiplication

matrix\_multiply(m, p, n, A, B, C, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

double end\_time = MPI\_Wtime();

if (ID == 0) {

printf("Matrix multiplication took %f seconds.\n", end\_time - start\_time);

}

free(A);

free(B);

free(C);

MPI\_Finalize();

return 0;

}

这个程序中，每个MPI进程都负责计算部分矩阵乘法。MPI\_Comm\_rank和MPI\_Comm\_size用于获取进程的排名和总数。并且使用了MPI\_Scatter和MPI\_Gather来更有效地分发和收集数据，以及MPI\_Bcast来广播矩阵B。这样可以避免使用显式的发送和接收。同时，通过使用本地矩阵 local\_A 和 local\_C，避免了在每个进程中为整个矩阵分配内存。这样，每个进程只负责自己分配的局部内存块的计算，减少了数据传输的开销。

MPI更适用于在分布式系统中处理大规模问题，因为它允许进程在不同的计算节点上执行。MPI的通信模型适用于在不同节点之间传递数据，每个进程有自己的内存空间，MPI使用显式的消息传递来在不同的进程之间传递数据。OpenMP更适合共享内存架构，其中多个线程可以同时访问相同的内存空间。在矩阵乘法中，可以通过并行化循环来利用共享内存并行性。

7、

以英伟达的费米架构的GPU显卡为例：

存储层次依次为：

寄存器（Register）：位于每个CUDA核心内，是最快的存储层次，用于存储局部变量和线程的寄存器文件中的其他数据。

共享内存（Shared Memory）：位于每个多处理器（SM）上，被所有线程块共享，用于线程块之间的通信和协作，以及提高数据局部性。

本地内存（Local Memory）：在每个CUDA核心上都有，相对于寄存器和共享内存而言较慢，用于存储未被寄存器或共享内存容纳的局部变量。

全局内存（Global Memory）：位于设备内存中，对所有SM和CUDA核心可见， 主要用于存储全局变量，可以在核函数调用之间保持数据的持久性。

常量缓存（Constant Cache）：存储在设备内存中，用于缓存常量内存的读取， 用于存储不会被修改的常量数据，对读取访问具有一定的优势。

纹理缓存（Texture Cache）：存储在设备内存中，用于缓存纹理内存的读取， 用于提高对图像纹理的访问速度。

L1 Cache 和 L2 Cache：位于SM中，用于缓存全局内存读取和写入， 提供对全局内存的更高效访问，减少内存访问延迟。

存储器控制器（Memory Controller）： 用于协调和管理存储器层次结构中的数据传输， 确保数据能够有效地从不同存储层次中传输到CUDA核心。

Femi架构CUDA线程层次结构实例：

