## 高性能计算实验作业4

### 第一题

* 题目要求：

基于某种编程语言测试MPI至少4种通信模式代码并解释其效果（比如C、C++、PythonMATLAB等语言，通信模式比如sactter、broadcast、reduce、send等）；

采用C++语言，在Ubuntu22.04上进行实验。

#### Send方式：

采用点对点通信方式,包括发送和接收线程,一个进程将消息发送到另一个特定的线程。

* 代码：

#include <iostream>

#include <mpi.h>

#include <sstream> // 使用 stringstream 来格式化输出

int main() {

int rank, size;

int data = 0;

MPI\_Init(NULL, NULL);

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

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

if (rank == 0) {

data = 123;

MPI\_Send(&data, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

// 使用 stringstream 格式化输出

std::stringstream ss;

ss << "Rank " << rank << ": sent " << data << " to rank 1\n";

std::cout << ss.str();

}

if (rank == 1) {

MPI\_Recv(&data, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

// 使用 stringstream 格式化输出

std::stringstream ss;

ss << "Rank " << rank << ": received " << data << " from rank 0\n";

std::cout << ss.str();

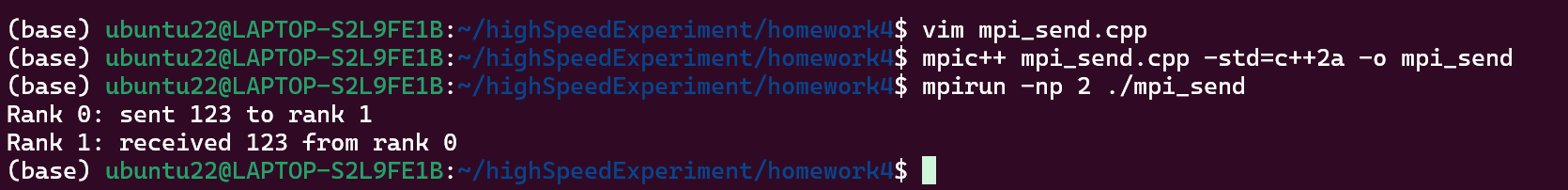
}

MPI\_Finalize();

return 0;

}

* 实验结果及其分析



上述结果表明,通过mpirun指定进程数量为2,通过 MPI\_Comm\_rank 获取进程的rank,进程0初始化data为123,并通过Send方式发送给进程1,进程1获取消息并打印。

#### Broadcast方式：

广播通信指的是将一个消息从一个进程发送到所有其他进程,通常用于让一个进程将数据分发给所有进程。

* 代码:

#include <iostream>

#include <mpi.h>

#include <sstream> // 使用 stringstream 来格式化输出

int main() {

int rank, size;

int data = 0;

int root = 0;

MPI\_Init(NULL, NULL);

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

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

// Root 进程发送数据

if (rank == root) {

data = 123;

// 使用 stringstream 格式化输出

std::stringstream ss;

ss << "Rank " << rank << ": send " << data << " to rank 1\n";

std::cout << ss.str();

}

// 广播数据到所有进程

MPI\_Bcast(&data, 1, MPI\_INT, root, MPI\_COMM\_WORLD);

// 所有进程打印接收到的数据

std::stringstream ss;

ss << "Rank " << rank << ": received " << data << " from rank 0\n";

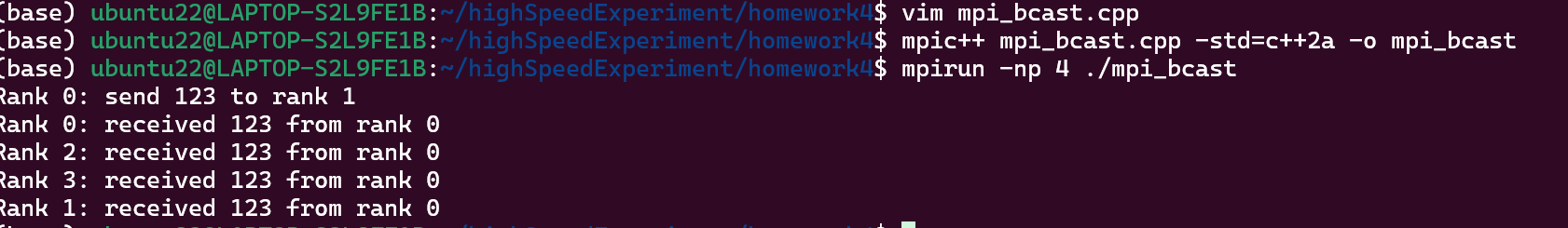
std::cout << ss.str();

MPI\_Finalize();

return 0;

}

* 实验结果及其分析



通常用于让一个进程将数据分发给所有的四个进程。上述代码调用了 MPI\_Bcast 进行广播,并且设定参数中的root为0,即进程0作为根节点,当进程0调用 MPI\_Bcast 时,会向其他进程广播数据,其他进程也通过 MPI\_Bcast 函数获取数据。 运行结果如上图所示,每个进程都收到了进程0发送的整型数字123。

#### Scatter方式：

在 MPI（Message Passing Interface）中，Scatter 是一种数据分发通信方式，用于将一个数据数组分成多个部分，并将每部分发送到不同的进程。通常由根进程（如 rank 0）执行分发操作，将数据分散到所有其他进程中。

* 代码

#include <iostream>

#include <mpi.h>

#include <sstream> // 使用 stringstream 来格式化输出

int main() {

int rank, size;

int data[12]; // 数据数组，大小为 12

int recv\_data[2]; // 每个进程接收的数据，大小为 2

MPI\_Init(NULL, NULL);

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

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

// Rank 0 进程初始化数据

if (rank == 0) {

for (int i = 0; i < 12; i++) {

data[i] = i;

}

}

// 将数据分发到所有进程

MPI\_Scatter(data, 2, MPI\_INT, recv\_data, 2, MPI\_INT, 0, MPI\_COMM\_WORLD);

// 使用 stringstream 格式化输出

std::stringstream ss;

ss << "Rank: " << rank << ", data: " << recv\_data[0] << ", " << recv\_data[1] << "\n";

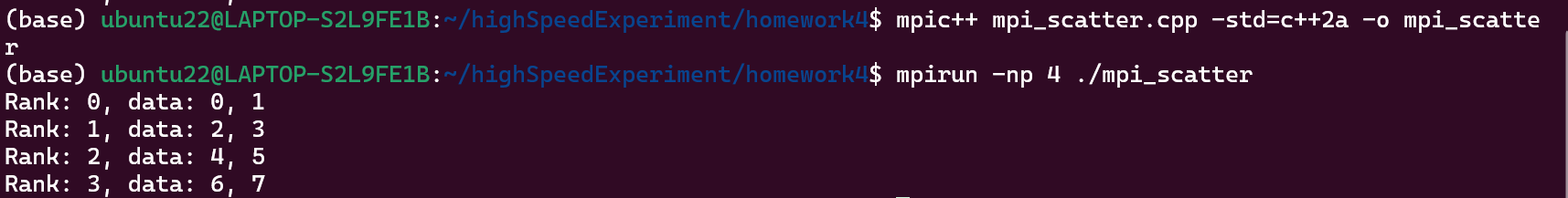
std::cout << ss.str();

MPI\_Finalize();

return 0;

}

* 实验结果及其分析



在 rank == 0 的进程中，初始化数组 data，将其填充为 0 到 11 的整数。 使用 MPI\_Scatter 将 data 数组中的数据分发到每个进程。 每个进程接收 data 的一部分，并将其存储在 recv\_data 数组中。这里每个进程接收 2 个整数。使用 std::stringstream 来构建格式化的字符串，然后打印每个进程接收到的数据。

#### Reduce通信方式:

在 MPI（Message Passing Interface）中，Reduce 是一种数据聚合通信方式，用于将多个进程的数据根据指定的操作（如求和、求最大值等）进行归约，并将结果汇总到指定的根进程。

* 代码

#include <iostream>

#include <mpi.h>

#include <sstream> // 使用 stringstream 来格式化输出

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

int rank, size;

int local\_value, global\_sum;

MPI\_Init(&argc, &argv);

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

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

// 每个进程的局部值设为它的 rank

local\_value = rank;

// 使用 MPI\_Reduce 求所有进程的局部值之和，并将结果存储在根进程 (rank 0) 中

MPI\_Reduce(&local\_value, &global\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// 仅根进程打印结果

if (rank == 0) {

std::stringstream ss;

ss << "Sum of ranks across all processes: " << global\_sum << "\n";

std::cout << ss.str();

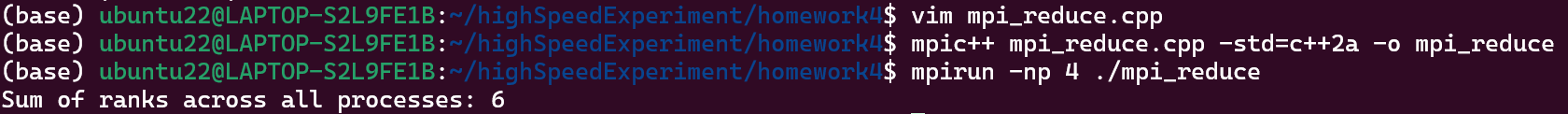
}

MPI\_Finalize();

return 0;

}

* 实验结果及其分析



每个进程的 local\_value 设为它的 rank 值。使用 MPI\_Reduce 对所有进程的 local\_value 进行求和操作。将结果存储在根进程（rank 0）的 global\_sum 中。 使用 std::stringstream 来格式化输出，仅根进程打印总和。 使用 MPI\_Reduce 对每个进程的 rank 值求和,指定了 4 个进程运行该程序，因此每个进程的 rank 分别为 0、1、2 和 3。因此，根进程（rank 0）输出的结果是 Sum of ranks across all processes: 6。

### 第二题

对比Pthread、OpenMP和MPI三种并行编程模式

Pthread (POSIX Threads)：

* 提供了在共享内存系统上创建和管理线程的底层接口。 适用于细粒度控制的多线程编程，特别是需要对线程管理（创建、销毁、同步）有精细控制的程序。常用于多核 CPU 的并行任务，通常在系统编程和性能敏感的应用中使用。基于共享内存模型，所有线程共享同一个地址空间。线程可以直接访问共享变量，通常需要手动同步（如使用互斥锁）来避免数据竞争。细粒度并行，能够控制每个线程的执行和同步，适合性能调优。 因为是低层次库，能提供较高的性能，但是也可能增加编程复杂度。

OpenMP (Open Multi-Processing)：

是用于共享内存系统的 API，基于多线程设计，利用编译器指令（pragma）帮助程序员方便地并行化代码。 使用简单，适合在多核处理器上进行并行计算，不需要深入的线程管理知识。 广泛应用于科学计算和大规模数据处理领域，特别适合循环并行化和数据并行。也是基于共享内存模型。线程之间通过共享变量进行通信，但提供了更高层的抽象和自动并行化机制，使用同步工具（如 critical、barrier 等）管理数据访问。提供了中高粒度并行控制，通常通过编译器指令在循环或特定代码段中实现并行化。 简单易用，易于集成到现有代码中，常用于多核 CPU 的并行计算。对于内存共享的多核系统，它能带来接近线性的加速比。

MPI (Message Passing Interface)：

* 基于消息传递的编程模型，适用于分布式内存系统（如计算集群）。 提供了不同计算节点之间通信的接口，适合跨节点的高性能并行计算。 常用于大规模并行计算和高性能计算（HPC），例如超级计算机上的模拟、建模和科学计算。基于分布式内存模型，每个进程有独立的内存空间。进程之间通过消息传递（如发送和接收）来交换数据，程序员需要显式管理通信。更适合粗粒度并行，需要显式控制数据的分割和通信。 对于跨节点的分布式计算，MPI 通常可以提供更高的扩展性和性能，尤其在跨多个节点或数百上千个 CPU 核心时。