目录

[MPI Tutorial Introduction 2](#_Toc56531127)

[MPI Hello World 4](#_Toc56531128)

[MPI Send and Receive 8](#_Toc56531129)

[Dynamic Receiving with MPI Probe (and MPI Status) 13](#_Toc56531130)

[Point-to-Point Communication Application - Random Walk 17](#_Toc56531131)

[MPI Broadcast and Collective Communication 25](#_Toc56531132)

[MPI Scatter, Gather, and Allgather 29](#_Toc56531133)

[Performing Parallel Rank with MPI 33](#_Toc56531134)

[MPI Reduce and Allreduce 39](#_Toc56531135)

[Introduction to Groups and Communicators 44](#_Toc56531136)

MPI Tutorial Introduction

分布式计算现在对于我们来说，就跟日常生活里的手机和电脑一样普及。你很明显应该认同这个观点，因为你发现了这个了不起的 MPI 教程网站！不管你是出于什么原因想学习并行编程（parallel programming），或者说分布式编程、并行编程，也许是因为课程需要，或者是工作，或者单纯地觉得好玩，我觉得你都应该选择一项在未来几年依然十分有价值的技术去学习。我觉得「消息传递接口」（Message Passing Interface, MPI）就是这样一项技术，而且学习它确实可以让你的并行编程知识变得更深厚。尽管 MPI 比大多数并行框架要更底层（比如 Hadoop），但是学习 MPI 会为你的并行编程打下良好的基础。

在我开始介绍 MPI 之前，我想要解释下我为什么做这个教程。当我在读研究生的时候，我大量的用到了 MPI。当我在 Argonne National Laboratory 实习的时候，我很幸运地可以跟 MPI 社区里很厉害的一些人一起工作，并且使用 MPI 在庞大的超级计算（supercomputing）集群上面做了很多疯狂的事情。然而，即使有这些资源和懂行的人可以问，我还是觉得学习 MPI 是件苦差事。

对我来说学习 MPI 很难主要是因为以下三个方面。第一，网上关于 MPI 的资料几乎都是过时的，或者不那么全的。第二，我想要自己简单地搭建一个可以运行 MPI 的集群环境，但是找不到这样的教程。最后，我读研究生的时候能买到的最便宜的关于 MPI 的书要60美元 - 对研究生来说太贵了。就目前分布式编程对我们生活的重要性来说，我觉得提供一个更好的教程能让别人学习 MPI 这样一个并行编程最重要的协议同等重要。

尽管我不敢自称是 MPI 专家，我觉得以简单易读的教程形式传播这些我在研究生阶段学习到的知识还是很有意义的一件事，你可以根据教程在你自己的集群上运行 MPI 程序！我希望这个教程能对你所有帮助，也许是事业上的，也许是学习上的，或者可能是生活上的帮助 - 因为分布式编程不仅仅意味着现在，它还是未来！

MPI 的历史简介

在 90 年代之前，程序员可没我们这么幸运。对于不同的计算架构写并发程序是一件困难而且冗长的事情。当时，很多软件库可以帮助写并发程序，但是没有一个大家都接受的标准来做这个事情。

在当时，大多数的并发程序只出现在科学和研究的领域。最广为接受的模型就是消息传递模型。什么是消息传递模型？它其实只是指程序通过在进程间传递消息（消息可以理解成带有一些信息和数据的一个数据结构）来完成某些任务。在实践中，并发程序用这个模型去实现特别容易。举例来说，主进程（manager process）可以通过对从进程（worker process）发送一个描述工作的消息来把这个工作分配给它。另一个例子就是一个并发的排序程序可以在当前进程中对当前进程可见的（我们称作本地的，locally）数据进行排序，然后把排好序的数据发送的邻居进程上面来进行合并的操作。几乎所有的并行程序可以使用消息传递模型来描述。

由于当时很多软件库都用到了这个消息传递模型，但是在定义上有些微小的差异，这些库的作者以及一些其他人为了解决这个问题就在 Supercomputing 1992 大会上定义了一个消息传递接口的标准- 也就是 MPI。这个标准接口使得程序员写的并发程序可以在所有主流的并发框架中运行。并且允许他们可以使用当时已经在使用的一些流行库的特性和模型。

到 1994 年的时候，一个完整的接口标准定义好了（MPI-1）。我们要记住 MPI 只是一个接口的定义而已。然后需要程序员去根据不同的架构去实现这个接口。很幸运的是，仅仅一年之后，一个完整的 MPI 实现就已经出现了。在第一个实现之后，MPI 就被大量地使用在消息传递应用程序中，并且依然是写这类程序的标准（de-facto）。

An accurate representation of the first MPI programmers.第一批 MPI 程序员的一个真实写照

MPI 对于消息传递模型的设计

在开始教程之前，我会先解释一下 MPI 在消息传递模型设计上的一些经典概念。第一个概念是通讯器（communicator）。通讯器定义了一组能够互相发消息的进程。在这组进程中，每个进程会被分配一个序号，称作秩（rank），进程间显性地通过指定秩来进行通信。

通信的基础建立在不同进程间发送和接收操作。一个进程可以通过指定另一个进程的秩以及一个独一无二的消息标签（tag）来发送消息给另一个进程。接受者可以发送一个接收特定标签标记的消息的请求（或者也可以完全不管标签，接收任何消息），然后依次处理接收到的数据。类似这样的涉及一个发送者以及一个接受者的通信被称作点对点（point-to-point）通信。

当然在很多情况下，某个进程可能需要跟所有其他进程通信。比如主进程想发一个广播给所有的从进程。在这种情况下，手动去写一个个进程点对点的信息传递就显得很笨拙。而且事实上这样会导致网络利用率低下。MPI 有专门的接口来帮我们处理这类所有进程间的集体性（collective）通信。

把点对点通信和集体性通信这两个机制合在一起已经可以创造十分复杂的并发程序了。事实上，这两个功能已经强大到我现在不需要再介绍任何 MPI 高级的特性了，我会把那些放到后面的教程中。现在，我们可以从在单机上安装 MPI或 启动一个 Amazon EC2 MPI 集群 开始我们的 MPI 旅途了！如果你已经把 MPI 装好了，那太好了，直接开始这个MPI Hello World 课程吧。

MPI Hello World

在这个课程里，在展示一个基础的 MPI Hello World 程序的同时我会介绍一下该如何运行 MPI 程序。这节课会涵盖如何初始化 MPI 的基础内容以及让 MPI 任务跑在几个不同的进程上。这节课程的代码是在 MPICH2（当时是1.4版本）上面运行通过的。（译者在 MPCH-3.2.1 上运行程序也没有问题）。如果你还没装 MPICH2，你参考MPICH2 安装指南

注意 - 这个网站的提到的所有代码都在 GitHub 上面。这篇教程的代码在 tutorials/mpi-hello-world/code。

Hello world 代码案例

让我们来看一下这节课的代码吧，完整的代码在 mpi\_hello\_world.c。 下面是一些重点内容的摘录。

#include <mpi.h>

#include <stdio.h>

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

// 初始化 MPI 环境

MPI\_Init(NULL, NULL);

// 通过调用以下方法来得到所有可以工作的进程数量

int world\_size;

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

// 得到当前进程的秩

int world\_rank;

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

// 得到当前进程的名字

char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

int name\_len;

MPI\_Get\_processor\_name(processor\_name, &name\_len);

// 打印一条带有当前进程名字，秩以及

// 整个 communicator 的大小的 hello world 消息。

printf("Hello world from processor %s, rank %d out of %d processors\n",

processor\_name, world\_rank, world\_size);

// 释放 MPI 的一些资源

MPI\_Finalize();

}

你应该已经注意到搭建一个 MPI 程序的第一步是引入 #include <mpi.h> 这个头文件。然后 MPI 环境必须以以下代码来初始化：

MPI\_Init(

int\* argc,

char\*\*\* argv)

在 MPI\_Init 的过程中，所有 MPI 的全局变量或者内部变量都会被创建。举例来说，一个通讯器 communicator 会根据所有可用的进程被创建出来（进程是我们通过 mpi 运行时的参数指定的），然后每个进程会被分配独一无二的秩 rank。当前来说，MPI\_Init 接受的两个参数是没有用处的，不过参数的位置保留着，可能以后的实现会需要用到。

在 MPI\_Init 之后，有两个主要的函数被调用到了。这两个函数是几乎所有 MPI 程序都会用到的。

MPI\_Comm\_size(

MPI\_Comm communicator,

int\* size)

MPI\_Comm\_size 会返回 communicator 的大小，也就是 communicator 中可用的进程数量。在我们的例子中，MPI\_COMM\_WORLD（这个 communicator 是 MPI 帮我们生成的）这个变量包含了当前 MPI 任务中所有的进程，因此在我们的代码里的这个调用会返回所有的可用的进程数目。

MPI\_Comm\_rank(

MPI\_Comm communicator,

int\* rank)

MPI\_Comm\_rank 这个函数会返回 communicator 中当前进程的 rank。 communicator 中每个进程会以此得到一个从0开始递增的数字作为 rank 值。rank 值主要是用来指定发送或者接受信息时对应的进程。

我们代码中使用到的一个不太常见的方法是：

MPI\_Get\_processor\_name(

char\* name,

int\* name\_length)

MPI\_Get\_processor\_name 会得到当前进程实际跑的时候所在的处理器名字。 代码中最后一个调用是：

MPI\_Finalize()

MPI\_Finalize 是用来清理 MPI 环境的。这个调用之后就没有 MPI 函数可以被调用了。

运行 MPI hello world 程序

现在查看以下代码文件以及代码所在的文件夹，你会看到一个 makefile。

>>> git clone https://github.com/mpitutorial/mpitutorial

>>> cd mpitutorial/tutorials/mpi-hello-world/code

>>> cat makefile

EXECS=mpi\_hello\_world

MPICC?=mpicc

all: ${EXECS}

mpi\_hello\_world: mpi\_hello\_world.c

${MPICC} -o mpi\_hello\_world mpi\_hello\_world.c

clean:

rm ${EXECS}

我的 makefile 会去找 MPICC 这个环境变量。如果你把 MPICH2 装在了本地文件夹里面而不是全局 PATH 下面, 手动设置一下 MPICC 这个环境变量，把它指向你的 mpicc 二进制程序。mpicc 二进制程序其实只是对 gcc 做了一层封装，使得编译和链接所有的 MPI 程序更方便。

>>> export MPICC=/home/kendall/bin/mpicc

>>> make

/home/kendall/bin/mpicc -o mpi\_hello\_world mpi\_hello\_world.c

当你的程序编译好之后，它就可以被执行了。不过执行之前你也许会需要一些额外配置。比如如果你想要在好几个节点的集群上面跑这个 MPI 程序的话，你需要配置一个 host 文件（不是 /etc/hosts）。如果你在笔记本或者单机上运行的话，可以跳过下面这一段。

需要配置的 host 文件会包含你想要运行的所有节点的名称。为了运行方便，你需要确认一下所有这些节点之间能通过 SSH 通信，并且需要根据[设置认证文件这个教程]((http://www.eng.cam.ac.uk/help/jpmg/ssh/authorized\_keys\_howto.html)配置不需要密码的 SSH 访问。 我的 host 文件看起来像这样：

>>> cat host\_file

cetus1

cetus2

cetus3

cetus4

为了用我提供的脚本来运行这个程序，你应该设置一个叫 MPI\_HOSTS 的环境变量，把它指向 host 文件所在的位置。我的脚本会自动把这个 host 文件的配置项加到 MPI 启动命令里。如果单机跑的话就不用设这个环境变量。另外如果你的 MPI 没有装到全局环境的话，你还需要指定 MPIRUN 这个环境变量指向你的 mpirun 二进制程序。

准备就绪之后你就可以使用这个项目的我提供的 python 脚本来执行程序。脚本在 tutorials 目录下面，这个脚本可以用来跑我们这个教程里面提到的所有程序（而且它会帮你先编译一下程序）。你可以在 mpitutorial 这个文件夹的根目录下执行以下命令：

>>> export MPIRUN=/home/kendall/bin/mpirun

>>> export MPI\_HOSTS=host\_file

>>> cd tutorials

>>> ./run.py mpi\_hello\_world

/home/kendall/bin/mpirun -n 4 -f host\_file ./mpi\_hello\_world

Hello world from processor cetus2, rank 1 out of 4 processors

Hello world from processor cetus1, rank 0 out of 4 processors

Hello world from processor cetus4, rank 3 out of 4 processors

Hello world from processor cetus3, rank 2 out of 4 processors

跟预想的一样，这个 MPI 程序运行在了我提供的所有节点上面。每个进程都被分配了一个单独的 rank，跟进程的名字一起打印出来了。你可以看到，在我们的输出的结果里，进程之间的打印顺序是任意的，因为我们的代码里并没有涉及到同步的操作。

我们可以在打印的内容上面那条看到脚本是如何调用 mpirun 这个程序的。mpirun 是 MPI 的实现用来启动任务的一个程序。进程会在 host 文件里指定的所有机器上面生成，MPI 程序就会在所有进程上面运行。我的脚步自定地提供了一个 -n 参数告诉 MPI 程序我要运行 4 个进程。你可以试着修改脚步来使用更多进程运行 MPI 程序。当心别把你的操作系统玩蹦了。:-)

你可能会问，“我的节点都都是双核的机器，我怎么样可以让 MPI 先在每个节点上的每个核上生成进程，再去其他的机器？ 其实方案很简单。修改一下你的 host 文件，在每个节点名字的后面加一个冒号和每个处理器有的核数就行了。比如，我在 host 文件里指定我的每个节点有2个核。

>>> cat host\_file

cetus1:2

cetus2:2

cetus3:2

cetus4:2

当我再次运行我的脚本，哇!，MPI 任务只在我的两个节点上生成了4个进程。

>>> ./run.py mpi\_hello\_world

/home/kendall/bin/mpirun -n 4 -f host\_file ./mpi\_hello\_world

Hello world from processor cetus1, rank 0 out of 4 processors

Hello world from processor cetus2, rank 2 out of 4 processors

Hello world from processor cetus2, rank 3 out of 4 processors

Hello world from processor cetus1, rank 1 out of 4 processors

接下来

现在你对 MPI 程序有了基本的了解。接下来可以学习基础的 点对点 （point-to-point）通信方法了。在下节课里，我讲解了 MPI 里基础的发送和接收函数。你也可以再去 MPI tutorials 首页查看所有其他的教程。

MPI Send and Receive

发送和接收是 MPI 里面两个基础的概念。MPI 里面几乎所有单个的方法都可以使用基础的发送和接收 API 来实现。在这节课里，我会介绍怎么使用 MPI 的同步的（或阻塞的，原文是 blocking）发送和接收方法，以及另外的一些跟使用 MPI 进行数据传输的基础概念。

注意 - 这个网站的提到的所有代码都在 GitHub 上面。这篇教程的代码在 tutorials/mpi-send-and-receive/code。

MPI 的发送和接收简介

MPI 的发送和接收方法是按以下方式进行的：开始的时候，A 进程决定要发送一些消息给 B 进程。A进程就会把需要发送给B进程的所有数据打包好，放到一个缓存里面。因为所有数据会被打包到一个大的信息里面，因此缓存常常会被比作信封（就像我们把好多信纸打包到一个信封里面然后再寄去邮局）。数据打包进缓存之后，通信设备（通常是网络）就需要负责把信息传递到正确的地方。这个正确的地方也就是根据特定秩确定的那个进程。

尽管数据已经被送达到 B 了，但是进程 B 依然需要确认它想要接收 A 的数据。一旦它确定了这点，数据就被传输成功了。进程 A 会接收到数据传递成功的信息，然后去干其他事情。

有时候 A 需要传递很多不同的消息给 B。为了让 B 能比较方便地区分不同的消息，MPI 运行发送者和接受者额外地指定一些信息 ID (正式名称是标签, tags)。当 B 只要求接收某种特定标签的信息的时候，其他的不是这个标签的信息会先被缓存起来，等到 B 需要的时候才会给 B。

把这些概念记在心里的同时，让我们来看一下 MPI 发送和接收方法的定义。

MPI\_Send(

void\* data,

int count,

MPI\_Datatype datatype,

int destination,

int tag,

MPI\_Comm communicator)

MPI\_Recv(

void\* data,

int count,

MPI\_Datatype datatype,

int source,

int tag,

MPI\_Comm communicator,

MPI\_Status\* status)

尽管一开始看起来参数有点多，慢慢地你会发现其实这些参数还是很好记忆的，因为大多数的 MPI 方法定义是类似的。第一个参数是数据缓存。第二个和第三个参数分别描述了数据的数量和类型。MPI\_send 会精确地发送 count 指定的数量个元素，MPI\_Recv 会最多接受 count 个元素（之后会详细讲）。第四个和第五个参数指定了发送方/接受方进程的秩以及信息的标签。第六个参数指定了使用的 communicator。MPI\_Recv 方法特有的最后一个参数提供了接受到的信息的状态。

基础 MPI 数据结构

MPI\_send 和 MPI\_Recv 方法使用了 MPI 的数据结构作为一种在更高层次指定消息结构的方法。举例来说，如果一个进程想要发送一个整数给另一个进程，它会指定 count 为 1，数据结构为 MPI\_INT。其他的 MPI 数据结构以及它们在 C 语言里对应的结构如下：

MPI datatype C equivalent

MPI\_SHORT short int

MPI\_INT int

MPI\_LONG long int

MPI\_LONG\_LONG long long int

MPI\_UNSIGNED\_CHAR unsigned char

MPI\_UNSIGNED\_SHORT unsigned short int

MPI\_UNSIGNED unsigned int

MPI\_UNSIGNED\_LONG unsigned long int

MPI\_UNSIGNED\_LONG\_LONG unsigned long long int

MPI\_FLOAT float

MPI\_DOUBLE double

MPI\_LONG\_DOUBLE long double

MPI\_BYTE char

目前来说，我们在 beginner 栏目里面只会使用到这些基础的数据结构。当我们有了足够多的基础知识之后，你会学习到如何创建自己的 MPI 数据类型来构建更复杂的消息类型。

MPI 发送 / 接收 程序

跟开头说的一样，所有代码会在 GitHub 上, 这节课的代码在 tutorials/mpi-send-and-receive/code。

第一个例子的代码在 send\_recv.c. 我们来看一下主要的部分：

// 得到当前进程的 rank 以及整个 communicator 的大小

int world\_rank;

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

int world\_size;

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

int number;

if (world\_rank == 0) {

number = -1;

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

} else if (world\_rank == 1) {

MPI\_Recv(&number, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

printf("Process 1 received number %d from process 0\n",

number);

}

MPI\_Comm\_rank 和 MPI\_Comm\_size 一开始是用来得到整个 communicator 空间的大小（也就是所有进程的数量）以及当前进程的秩。然后如果当前进程是 0 进程，那么我们就初始化一个数字 -1 然后把它发送给 1 进程。然后你可以看到 else if 条件语句里的话题，进程 1 会调用 MPI\_Recv 去接受这个数字。然后会将接收到的数字打印出来。由于我们明确地发送接收了一个整数，因此 MPI\_INT 数据类型被使用了。每个进程还使用了 0 作为消息标签来指定消息。由于我们这里只有一种类型的信息被传递了，因此进程也可以使用预先定义好的常量 MPI\_ANY\_TAG 来作为标签数字。

你可以把代码从GitHub下载下来并运行 run.py 脚本.

>>> git clone https://github.com/mpitutorial/mpitutorial

>>> cd mpitutorial/tutorials

>>> ./run.py send\_recv

mpirun -n 2 ./send\_recv

Process 1 received number -1 from process 0

可以看到跟我们预想的一样，进程一收到了来自进程零传递的数字 -1。

MPI 乒乓程序

接下来的程序比较有趣，是一个乒乓游戏。两个进程会一直使用 MPI\_Send 和 MPI\_Recv 方法来“推挡”消息，直到他们决定不玩了。 你可以看一眼代码ping\_pong.c。主要部分如下所示。

int ping\_pong\_count = 0;

int partner\_rank = (world\_rank + 1) % 2;

while (ping\_pong\_count < PING\_PONG\_LIMIT) {

if (world\_rank == ping\_pong\_count % 2) {

// Increment the ping pong count before you send it

ping\_pong\_count++;

MPI\_Send(&ping\_pong\_count, 1, MPI\_INT, partner\_rank, 0, MPI\_COMM\_WORLD);

printf("%d sent and incremented ping\_pong\_count %d to %d\n",

world\_rank, ping\_pong\_count,

partner\_rank);

} else {

MPI\_Recv(&ping\_pong\_count, 1, MPI\_INT, partner\_rank, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("%d received ping\_pong\_count %d from %d\n",

world\_rank, ping\_pong\_count, partner\_rank);

}

}

这个程序是为2个进程执行而设计的。这两个进程一开始会根据我们写的一个简单的求余算法来确定各自的对手。ping\_pong\_count 一开始被初始化为0，然后每次发送消息之后会递增1。随着 ping\_pong\_count 的递增，两个进程会轮流成为发送者和接受者。最后，当我们设定的 limit 被触发的时候（我的代码里设定为10），进程就停止了发送和接收。程序的输出如下。

>>> ./run.py ping\_pong

0 sent and incremented ping\_pong\_count 1 to 1

0 received ping\_pong\_count 2 from 1

0 sent and incremented ping\_pong\_count 3 to 1

0 received ping\_pong\_count 4 from 1

0 sent and incremented ping\_pong\_count 5 to 1

0 received ping\_pong\_count 6 from 1

0 sent and incremented ping\_pong\_count 7 to 1

0 received ping\_pong\_count 8 from 1

0 sent and incremented ping\_pong\_count 9 to 1

0 received ping\_pong\_count 10 from 1

1 received ping\_pong\_count 1 from 0

1 sent and incremented ping\_pong\_count 2 to 0

1 received ping\_pong\_count 3 from 0

1 sent and incremented ping\_pong\_count 4 to 0

1 received ping\_pong\_count 5 from 0

1 sent and incremented ping\_pong\_count 6 to 0

1 received ping\_pong\_count 7 from 0

1 sent and incremented ping\_pong\_count 8 to 0

1 received ping\_pong\_count 9 from 0

1 sent and incremented ping\_pong\_count 10 to 0

这个程序在其他机器上运行的输出可能会由于进程调度的不同跟上面的不一样。不管怎么样，你可以看到，进程0和进程1在轮流发送和接收 ping\_pong\_count。

环程序

我还添加了另一个使用 MPI\_Send 和 MPI\_Recv 的样例程序，这个程序使用到了多个进程。在这个例子里，一个值会在各个进程之间以一个环的形式传递。代码在 ring.c。主要的部分如下。

int token;

if (world\_rank != 0) {

MPI\_Recv(&token, 1, MPI\_INT, world\_rank - 1, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n",

world\_rank, token, world\_rank - 1);

} else {

// Set the token's value if you are process 0

token = -1;

}

MPI\_Send(&token, 1, MPI\_INT, (world\_rank + 1) % world\_size,

0, MPI\_COMM\_WORLD);

// Now process 0 can receive from the last process.

if (world\_rank == 0) {

MPI\_Recv(&token, 1, MPI\_INT, world\_size - 1, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n",

world\_rank, token, world\_size - 1);

}

这个环程序在进程0上面初始化了一个值-1，赋值给 token。然后这个值会依次传递给每个进程。程序会在进程0从最后一个进程接收到值之后结束。如你所见，我们的逻辑避免了死锁的发生。具体来说，进程0保证了在想要接受数据之前发送了 token。所有其他的进程只是简单的调用 MPI\_Recv (从他们的邻居进程接收数据)，然后调用 MPI\_Send (发送数据到他们的邻居进程)把数据从环上传递下去。 MPI\_Send 和 MPI\_Recv 会阻塞直到数据传递完成。因为这个特性，打印出来的数据是跟数据传递的次序一样的。用5个进程的话，输出应该是这样的：

>>> ./run.py ring

Process 1 received token -1 from process 0

Process 2 received token -1 from process 1

Process 3 received token -1 from process 2

Process 4 received token -1 from process 3

Process 0 received token -1 from process 4

如你所见，进程0先把-1这个值传递给了进程1。然后数据会在环里一直传递到进程0。

接下来

现在你有了对于 MPI\_Send 和 MPI\_Recv 的基础理解，是时候对这些方法进行一些深入研究了。下节课，我会讲解如何预估和动态地接受信息。你也可以再去 MPI tutorials 首页查看所有其他的教程。

Dynamic Receiving with MPI Probe (and MPI Status)

In the previous lesson, I discussed how to use MPI\_Send and MPI\_Recv to perform standard point-to-point communication. I only covered how to send messages in which the length of the message was known beforehand. Although it is possible to send the length of the message as a separate send / recv operation, MPI natively supports dynamic messages with just a few additional function calls. I will be going over how to use these functions in this lesson.

Note - All of the code for this site is on GitHub. This tutorial’s code is under tutorials/dynamic-receiving-with-mpi-probe-and-mpi-status/code.

The MPI\_Status structure

As covered in the previous lesson, the MPI\_Recv operation takes the address of an MPI\_Status structure as an argument (which can be ignored with MPI\_STATUS\_IGNORE). If we pass an MPI\_Status structure to the MPI\_Recv function, it will be populated with additional information about the receive operation after it completes. The three primary pieces of information include:

The rank of the sender. The rank of the sender is stored in the MPI\_SOURCE element of the structure. That is, if we declare an MPI\_Status stat variable, the rank can be accessed with stat.MPI\_SOURCE.

The tag of the message. The tag of the message can be accessed by the MPI\_TAG element of the structure (similar to MPI\_SOURCE).

The length of the message. The length of the message does not have a predefined element in the status structure. Instead, we have to find out the length of the message with MPI\_Get\_count.

MPI\_Get\_count(

MPI\_Status\* status,

MPI\_Datatype datatype,

int\* count)

In MPI\_Get\_count, the user passes the MPI\_Status structure, the datatype of the message, and count is returned. The count variable is the total number of datatype elements that were received.

Why would any of this information be necessary? It turns out that MPI\_Recv can take MPI\_ANY\_SOURCE for the rank of the sender and MPI\_ANY\_TAG for the tag of the message. For this case, the MPI\_Status structure is the only way to find out the actual sender and tag of the message. Furthermore, MPI\_Recv is not guaranteed to receive the entire amount of elements passed as the argument to the function call. Instead, it receives the amount of elements that were sent to it (and returns an error if more elements were sent than the desired receive amount). The MPI\_Get\_count function is used to determine the actual receive amount.

An example of querying the MPI\_Status structure

The program that queries the MPI\_Status structure is in check\_status.c. The program sends a random amount of numbers to a receiver, and the receiver then finds out how many numbers were sent. The main part of the code looks like this.

const int MAX\_NUMBERS = 100;

int numbers[MAX\_NUMBERS];

int number\_amount;

if (world\_rank == 0) {

// Pick a random amount of integers to send to process one

srand(time(NULL));

number\_amount = (rand() / (float)RAND\_MAX) \* MAX\_NUMBERS;

// Send the amount of integers to process one

MPI\_Send(numbers, number\_amount, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

printf("0 sent %d numbers to 1\n", number\_amount);

} else if (world\_rank == 1) {

MPI\_Status status;

// Receive at most MAX\_NUMBERS from process zero

MPI\_Recv(numbers, MAX\_NUMBERS, MPI\_INT, 0, 0, MPI\_COMM\_WORLD,

&status);

// After receiving the message, check the status to determine

// how many numbers were actually received

MPI\_Get\_count(&status, MPI\_INT, &number\_amount);

// Print off the amount of numbers, and also print additional

// information in the status object

printf("1 received %d numbers from 0. Message source = %d, "

"tag = %d\n",

number\_amount, status.MPI\_SOURCE, status.MPI\_TAG);

}

As we can see, process zero randomly sends up to MAX\_NUMBERS integers to process one. Process one then calls MPI\_Recv for a total of MAX\_NUMBERS integers. Although process one is passing MAX\_NUMBERS as the argument to MPI\_Recv, process one will receive at most this amount of numbers. In the code, process one calls MPI\_Get\_count with MPI\_INT as the datatype to find out how many integers were actually received. Along with printing off the size of the received message, process one also prints off the source and tag of the message by accessing the MPI\_SOURCE and MPI\_TAG elements of the status structure.

As a clarification, the return value from MPI\_Get\_count is relative to the datatype which is passed. If the user were to use MPI\_CHAR as the datatype, the returned amount would be four times as large (assuming an integer is four bytes and a char is one byte). If you run the check\_status program from the tutorials directory of the repo, the output should look similar to this.

>>> cd tutorials

>>> ./run.py check\_status

mpirun -n 2 ./check\_status

0 sent 92 numbers to 1

1 received 92 numbers from 0. Message source = 0, tag = 0

As expected, process zero sends a random amount of integers to process one, which prints off information about the received message.

Using MPI\_Probe to find out the message size

Now that you understand how the MPI\_Status object works, we can now use it to our advantage a little bit more. Instead of posting a receive and simply providing a really large buffer to handle all possible sizes of messages (as we did in the last example), you can use MPI\_Probe to query the message size before actually receiving it. The function prototype looks like this.

MPI\_Probe(

int source,

int tag,

MPI\_Comm comm,

MPI\_Status\* status)

MPI\_Probe looks quite similar to MPI\_Recv. In fact, you can think of MPI\_Probe as an MPI\_Recv that does everything but receive the message. Similar to MPI\_Recv, MPI\_Probe will block for a message with a matching tag and sender. When the message is available, it will fill the status structure with information. The user can then use MPI\_Recv to receive the actual message.

The lesson code has an example of this in probe.c. Here’s what the main source code looks like.

int number\_amount;

if (world\_rank == 0) {

const int MAX\_NUMBERS = 100;

int numbers[MAX\_NUMBERS];

// Pick a random amount of integers to send to process one

srand(time(NULL));

number\_amount = (rand() / (float)RAND\_MAX) \* MAX\_NUMBERS;

// Send the random amount of integers to process one

MPI\_Send(numbers, number\_amount, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

printf("0 sent %d numbers to 1\n", number\_amount);

} else if (world\_rank == 1) {

MPI\_Status status;

// Probe for an incoming message from process zero

MPI\_Probe(0, 0, MPI\_COMM\_WORLD, &status);

// When probe returns, the status object has the size and other

// attributes of the incoming message. Get the message size

MPI\_Get\_count(&status, MPI\_INT, &number\_amount);

// Allocate a buffer to hold the incoming numbers

int\* number\_buf = (int\*)malloc(sizeof(int) \* number\_amount);

// Now receive the message with the allocated buffer

MPI\_Recv(number\_buf, number\_amount, MPI\_INT, 0, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("1 dynamically received %d numbers from 0.\n",

number\_amount);

free(number\_buf);

}

Similar to the last example, process zero picks a random amount of numbers to send to process one. What is different in this example is that process one now calls MPI\_Probe to find out how many elements process zero is trying to send (using MPI\_Get\_count). Process one then allocates a buffer of the proper size and receives the numbers. Running the code will look similar to this.

>>> ./run.py probe

mpirun -n 2 ./probe

0 sent 93 numbers to 1

1 dynamically received 93 numbers from 0

Although this example is trivial, MPI\_Probe forms the basis of many dynamic MPI applications. For example, manager/worker programs will often make heavy use of MPI\_Probe when exchanging variable-sized worker messages. As an exercise, make a wrapper around MPI\_Recv that uses MPI\_Probe for any dynamic applications you might write. It makes the code look much nicer :-)

Up next

Do you feel comfortable using the standard blocking point-to-point communication routines? If so, then you already have the ability to write endless amounts of parallel applications! Let’s look at a more advanced example of using the routines you have learned. Check out the application example using MPI\_Send, MPI\_Recv, and MPI\_Probe.

Point-to-Point Communication Application - Random Walk

It’s time to go through an application example using some of the concepts introduced in the sending and receiving tutorial and the MPI\_Probe and MPI\_Status lesson. The application simulates a process which I refer to as “random walking.”

Note - All of the code for this site is on GitHub. This tutorial’s code is under tutorials/point-to-point-communication-application-random-walk/code.

The basic problem definition of a random walk is as follows. Given a Min, Max, and random walker W, make walker W take S random walks of arbitrary length to the right. If the process goes out of bounds, it wraps back around. W can only move one unit to the right or left at a time.

Random walk illustration

Although the application in itself is very basic, the parallelization of random walking can simulate the behavior of a wide variety of parallel applications. More on that later. For now, let’s overview how to parallelize the random walk problem.

Parallelization of the random walking problem

Our first task, which is pertinent to many parallel programs, is splitting the domain across processes. The random walk problem has a one-dimensional domain of size Max - Min + 1 (since Max and Min are inclusive to the walker). Assuming that walkers can only take integer-sized steps, we can easily partition the domain into near-equal-sized chunks across processes. For example, if Min is 0 and Max is 20 and we have four processes, the domain would be split like this.

Domain decomposition example

The first three processes own five units of the domain while the last process takes the last five units plus the one remaining unit. Once the domain has been partitioned, the application will initialize walkers. As explained earlier, a walker will take S walks with a random total walk size. For example, if the walker takes a walk of size six on process zero (using the previous domain decomposition), the execution of the walker will go like this:

The walker starts taking incremental steps. When it hits value four, however, it has reached the end of the bounds of process zero. Process zero now has to communicate the walker to process one.

Process one receives the walker and continues walking until it has reached its total walk size of six. The walker can then proceed on a new random walk.

Random walk, step one

In this example, W only had to be communicated one time from process zero to process one. If W had to take a longer walk, however, it may have needed to be passed through more processes along its path through the domain.

Coding the application using MPI\_Send and MPI\_Recv

This application can be coded using MPI\_Send and MPI\_Recv. Before we begin looking at code, let’s establish some preliminary characteristics and functions of the program:

Each process determines their part of the domain.

Each process initializes exactly N walkers, all which start at the first value of their local domain.

Each walker has two associated integer values: the current position of the walker and the number of steps left to take.

Walkers start traversing through the domain and are passed to other processes until they have completed their walk.

The processes terminate when all walkers have finished.

Let’s begin by writing code for the domain decomposition. The function will take in the total domain size and find the appropriate subdomain for the MPI process. It will also give any remainder of the domain to the final process. For simplicity, I just call MPI\_Abort for any errors that are found. The function, called decompose\_domain, looks like this:

void decompose\_domain(int domain\_size, int world\_rank,

int world\_size, int\* subdomain\_start,

int\* subdomain\_size) {

if (world\_size > domain\_size) {

// Don't worry about this special case. Assume the domain

// size is greater than the world size.

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

\*subdomain\_start = domain\_size / world\_size \* world\_rank;

\*subdomain\_size = domain\_size / world\_size;

if (world\_rank == world\_size - 1) {

// Give remainder to last process

\*subdomain\_size += domain\_size % world\_size;

}

}

As you can see, the function splits the domain in even chunks, taking care of the case when a remainder is present. The function returns a subdomain start and a subdomain size.

Next, we need to create a function that initializes walkers. We first define a walker structure that looks like this:

typedef struct {

int location;

int num\_steps\_left\_in\_walk;

} Walker;

Our initialization function, called initialize\_walkers, takes the subdomain bounds and adds walkers to an incoming\_walkers vector (by the way, this application is in C++).

void initialize\_walkers(int num\_walkers\_per\_proc, int max\_walk\_size,

int subdomain\_start, int subdomain\_size,

vector<Walker>\* incoming\_walkers) {

Walker walker;

for (int i = 0; i < num\_walkers\_per\_proc; i++) {

// Initialize walkers in the middle of the subdomain

walker.location = subdomain\_start;

walker.num\_steps\_left\_in\_walk =

(rand() / (float)RAND\_MAX) \* max\_walk\_size;

incoming\_walkers->push\_back(walker);

}

}

After initialization, it is time to progress the walkers. Let’s start off by making a walking function. This function is responsible for progressing the walker until it has finished its walk. If it goes out of local bounds, it is added to the outgoing\_walkers vector.

void walk(Walker\* walker, int subdomain\_start, int subdomain\_size,

int domain\_size, vector<Walker>\* outgoing\_walkers) {

while (walker->num\_steps\_left\_in\_walk > 0) {

if (walker->location == subdomain\_start + subdomain\_size) {

// Take care of the case when the walker is at the end

// of the domain by wrapping it around to the beginning

if (walker->location == domain\_size) {

walker->location = 0;

}

outgoing\_walkers->push\_back(\*walker);

break;

} else {

walker->num\_steps\_left\_in\_walk--;

walker->location++;

}

}

}

Now that we have established an initialization function (that populates an incoming walker list) and a walking function (that populates an outgoing walker list), we only need two more functions: a function that sends outgoing walkers and a function that receives incoming walkers. The sending function looks like this:

void send\_outgoing\_walkers(vector<Walker>\* outgoing\_walkers,

int world\_rank, int world\_size) {

// Send the data as an array of MPI\_BYTEs to the next process.

// The last process sends to process zero.

MPI\_Send((void\*)outgoing\_walkers->data(),

outgoing\_walkers->size() \* sizeof(Walker), MPI\_BYTE,

(world\_rank + 1) % world\_size, 0, MPI\_COMM\_WORLD);

// Clear the outgoing walkers

outgoing\_walkers->clear();

}

The function that receives incoming walkers should use MPI\_Probe since it does not know beforehand how many walkers it will receive. This is what it looks like:

void receive\_incoming\_walkers(vector<Walker>\* incoming\_walkers,

int world\_rank, int world\_size) {

MPI\_Status status;

// Receive from the process before you. If you are process zero,

// receive from the last process

int incoming\_rank =

(world\_rank == 0) ? world\_size - 1 : world\_rank - 1;

MPI\_Probe(incoming\_rank, 0, MPI\_COMM\_WORLD, &status);

// Resize your incoming walker buffer based on how much data is

// being received

int incoming\_walkers\_size;

MPI\_Get\_count(&status, MPI\_BYTE, &incoming\_walkers\_size);

incoming\_walkers->resize(

incoming\_walkers\_size / sizeof(Walker));

MPI\_Recv((void\*)incoming\_walkers->data(), incoming\_walkers\_size,

MPI\_BYTE, incoming\_rank, 0, MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

}

Now we have established the main functions of the program. We have to tie all these function together as follows:

Initialize the walkers.

Progress the walkers with the walk function.

Send out any walkers in the outgoing\_walkers vector.

Receive new walkers and put them in the incoming\_walkers vector.

Repeat steps two through four until all walkers have finished.

The first attempt at writing this program is below. For now, we will not worry about how to determine when all walkers have finished. Before you look at the code, I must warn you - this code is incorrect! With this in mind, lets look at my code and hopefully you can see what might be wrong with it.

// Find your part of the domain

decompose\_domain(domain\_size, world\_rank, world\_size,

&subdomain\_start, &subdomain\_size);

// Initialize walkers in your subdomain

initialize\_walkers(num\_walkers\_per\_proc, max\_walk\_size,

subdomain\_start, subdomain\_size,

&incoming\_walkers);

while (!all\_walkers\_finished) { // Determine walker completion later

// Process all incoming walkers

for (int i = 0; i < incoming\_walkers.size(); i++) {

walk(&incoming\_walkers[i], subdomain\_start, subdomain\_size,

domain\_size, &outgoing\_walkers);

}

// Send all outgoing walkers to the next process.

send\_outgoing\_walkers(&outgoing\_walkers, world\_rank,

world\_size);

// Receive all the new incoming walkers

receive\_incoming\_walkers(&incoming\_walkers, world\_rank,

world\_size);

}

Everything looks normal, but the order of function calls has introduced a very likely scenario - deadlock.

Deadlock and prevention

According to Wikipedia, deadlock “refers to a specific condition when two or more processes are each waiting for the other to release a resource, or more than two processes are waiting for resources in a circular chain.” In our case, the above code will result in a circular chain of MPI\_Send calls.

Deadlock

It is worth noting that the above code will actually not deadlock most of the time. Although MPI\_Send is a blocking call, the MPI specification says that MPI\_Send blocks until the send buffer can be reclaimed. This means that MPI\_Send will return when the network can buffer the message. If the sends eventually can’t be buffered by the network, they will block until a matching receive is posted. In our case, there are enough small sends and frequent matching receives to not worry about deadlock, however, a big enough network buffer should never be assumed.

Since we are only focusing on MPI\_Send and MPI\_Recv in this lesson, the best way to avoid the possible sending and receiving deadlock is to order the messaging such that sends will have matching receives and vice versa. One easy way to do this is to change our loop around such that even-numbered processes send outgoing walkers before receiving walkers and odd-numbered processes do the opposite. Given two stages of execution, the sending and receiving will now look like this:

Deadlock prevention

Note - Executing this with one process can still deadlock. To avoid this, simply don’t perform sends and receives when using one process.

You may be asking, does this still work with an odd number of processes? We can go through a similar diagram again with three processes:

Deadlock solution

As you can see, at all three stages, there is at least one posted MPI\_Send that matches a posted MPI\_Recv, so we don’t have to worry about the occurrence of deadlock.

Determining completion of all walkers

Now comes the final step of the program - determining when every single walker has finished. Since walkers can walk for a random length, they can finish their journey on any process. Because of this, it is difficult for all processes to know when all walkers have finished without some sort of additional communication. One possible solution is to have process zero keep track of all of the walkers that have finished and then tell all the other processes when to terminate. This solution, however, is quite cumbersome since each process would have to report any completed walkers to process zero and then also handle different types of incoming messages.

For this lesson, we will keep things simple. Since we know the maximum distance that any walker can travel and the smallest total size it can travel for each pair of sends and receives (the subdomain size), we can figure out the amount of sends and receives each process should do before termination. Using this characteristic of the program along with our strategy to avoid deadlock, the final main part of the program looks like this:

// Find your part of the domain

decompose\_domain(domain\_size, world\_rank, world\_size,

&subdomain\_start, &subdomain\_size);

// Initialize walkers in your subdomain

initialize\_walkers(num\_walkers\_per\_proc, max\_walk\_size,

subdomain\_start, subdomain\_size,

&incoming\_walkers);

// Determine the maximum amount of sends and receives needed to

// complete all walkers

int maximum\_sends\_recvs =

max\_walk\_size / (domain\_size / world\_size) + 1;

for (int m = 0; m < maximum\_sends\_recvs; m++) {

// Process all incoming walkers

for (int i = 0; i < incoming\_walkers.size(); i++) {

walk(&incoming\_walkers[i], subdomain\_start, subdomain\_size,

domain\_size, &outgoing\_walkers);

}

// Send and receive if you are even and vice versa for odd

if (world\_rank % 2 == 0) {

send\_outgoing\_walkers(&outgoing\_walkers, world\_rank,

world\_size);

receive\_incoming\_walkers(&incoming\_walkers, world\_rank,

world\_size);

} else {

receive\_incoming\_walkers(&incoming\_walkers, world\_rank,

world\_size);

send\_outgoing\_walkers(&outgoing\_walkers, world\_rank,

world\_size);

}

}

Running the application

The lesson code is viewable here. In contrast to the other lessons, this code uses C++. When installing MPICH2, you also installed the C++ MPI compiler (unless you explicitly configured it otherwise). If you installed MPICH2 in a local directory, make sure that you have set your MPICXX environment variable to point to the correct mpicxx compiler in order to use my makefile.

In my code, I have set up the application run script to provide default values for the program: 100 for the domain size, 500 for the maximum walk size, and 20 for the number of walkers per process. If you run the random\_walk program from the tutorials directory of the repo, it should spawn 5 processes and produce output similar to this.

>>> cd tutorials

>>> ./run.py random\_walk

mpirun -n 5 ./random\_walk 100 500 20

Process 2 initiated 20 walkers in subdomain 40 - 59

Process 2 sending 18 outgoing walkers to process 3

Process 3 initiated 20 walkers in subdomain 60 - 79

Process 3 sending 20 outgoing walkers to process 4

Process 3 received 18 incoming walkers

Process 3 sending 18 outgoing walkers to process 4

Process 4 initiated 20 walkers in subdomain 80 - 99

Process 4 sending 18 outgoing walkers to process 0

Process 0 initiated 20 walkers in subdomain 0 - 19

Process 0 sending 17 outgoing walkers to process 1

Process 0 received 18 incoming walkers

Process 0 sending 16 outgoing walkers to process 1

Process 0 received 20 incoming walkers

The output continues until processes finish all sending and receiving of all walkers.

MPI Broadcast and Collective Communication

MPI 教程 到目前为止，我们讲解了点对点的通信，这种通信只会同时涉及两个不同的进程。这节课是我们 MPI 集体通信（collective communication）的第一节课。集体通信指的是一个涉及 communicator 里面所有进程的一个方法。这节课我们会解释集体通信以及一个标准的方法 - broadcasting (广播)。

注意 - 这个网站的提到的所有代码都在 GitHub 上面。这篇教程的代码在 tutorials/mpi-broadcast-and-collective-communication/code。

集体通信以及同步点

关于集体通信需要记住的一点是它在进程间引入了同步点的概念。这意味着所有的进程在执行代码的时候必须首先都到达一个同步点才能继续执行后面的代码。

在看具体的集体通信方法之前，让我们更仔细地看一下同步这个概念。事实上，MPI 有一个特殊的函数来做同步进程的这个操作。

MPI\_Barrier(MPI\_Comm communicator)

这个函数的名字十分贴切（Barrier，屏障）- 这个方法会构建一个屏障，任何进程都没法跨越屏障，直到所有的进程都到达屏障。这边有一个示意图。假设水平的轴代表的是程序的执行，小圆圈代表不同的进程。

MPI\_Barrier example

进程0在时间点 (T 1) 首先调用 MPI\_Barrier。然后进程0就一直等在屏障之前，之后进程1和进程3在 (T 2) 时间点到达屏障。当进程2最终在时间点 (T 3) 到达屏障的时候，其他的进程就可以在 (T 4) 时间点再次开始运行。

MPI\_Barrier 在很多时候很有用。其中一个用途是用来同步一个程序，使得分布式代码中的某一部分可以被精确的计时。

想知道 MPI\_Barrier 是怎么实现的么？我知道你当然想 :-) 还记得我们之前的在发送和接收教程 里的环程序么？帮你回忆一下，我们当时写了一个在所有进程里以环的形式传递一个令牌（token）的程序。这种形式的程序是最简单的一种实现屏障的方式，因为令牌只有在所有程序都完成之后才能被传递回第一个进程。

关于同步最后一个要注意的地方是：始终记得每一个你调用的集体通信方法都是同步的。也就是说，如果你没法让所有进程都完成 MPI\_Barrier，那么你也没法完成任何集体调用。如果你在没有确保所有进程都调用 MPI\_Barrier 的情况下调用了它，那么程序会空闲下来。这对初学者来说会很迷惑，所以小心这类问题。

使用 MPI\_Bcast 来进行广播

广播 (broadcast) 是标准的集体通信技术之一。一个广播发生的时候，一个进程会把同样一份数据传递给一个 communicator 里的所有其他进程。广播的主要用途之一是把用户输入传递给一个分布式程序，或者把一些配置参数传递给所有的进程。

广播的通信模式看起来像这样：

MPI\_Bcast 模式

在这个例子里，进程0是我们的根进程，它持有一开始的数据。其他所有的进程都会从它这里接受到一份数据的副本。

在 MPI 里面，广播可以使用 MPI\_Bcast 来做到。函数签名看起来像这样：

MPI\_Bcast(

void\* data,

int count,

MPI\_Datatype datatype,

int root,

MPI\_Comm communicator)

尽管根节点和接收节点做不同的事情，它们都是调用同样的这个 MPI\_Bcast 函数来实现广播。当根节点(在我们的例子是节点0)调用 MPI\_Bcast 函数的时候，data 变量里的值会被发送到其他的节点上。当其他的节点调用 MPI\_Bcast 的时候，data 变量会被赋值成从根节点接受到的数据。

使用 MPI\_Send 和 MPI\_Recv 来做广播

粗略看的话，似乎 MPI\_Bcast 仅仅是在 MPI\_Send 和 MPI\_Recv 基础上进行了一层包装。事实上，我们现在就可以自己来做这层封装。我们的函数叫做 my\_bcast，在这里可以看到: bcast.c。它跟 MPI\_Bcast 接受一样的参数，看起来像这样：

void my\_bcast(void\* data, int count, MPI\_Datatype datatype, int root,

MPI\_Comm communicator) {

int world\_rank;

MPI\_Comm\_rank(communicator, &world\_rank);

int world\_size;

MPI\_Comm\_size(communicator, &world\_size);

if (world\_rank == root) {

// If we are the root process, send our data to everyone

int i;

for (i = 0; i < world\_size; i++) {

if (i != world\_rank) {

MPI\_Send(data, count, datatype, i, 0, communicator);

}

}

} else {

// If we are a receiver process, receive the data from the root

MPI\_Recv(data, count, datatype, root, 0, communicator,

MPI\_STATUS\_IGNORE);

}

}

根节点把数据传递给所有其他的节点，其他的节点接收根节点的数据。很简单对吧？如果你从这个 repo tutorials 目录下面运行这个程序的话，输出看起来应该像这样：

>>> cd tutorials

>>> ./run.py my\_bcast

mpirun -n 4 ./my\_bcast

Process 0 broadcasting data 100

Process 2 received data 100 from root process

Process 3 received data 100 from root process

Process 1 received data 100 from root process

不管你信不信，其实我们的函数效率特别低！假设每个进程都只有一个「输出/输入」网络连接。我们的方法只是使用了进程0的一个输出连接来书传递数据。比较聪明的方法是使用一个基于树的沟通算法对网络进行更好的利用。比如这样：

MPI\_Bcast tree

在示意图里，进程0一开始传递数据给进程1。跟我们之前的例子类似，第二个阶段的时候进程0依旧会把数据传递给进程2。这个例子中不同的是进程1在第二阶段也会传递数据给进程3。在第二阶段，两个网络连接在同时发生了。在这个树形算法里，能够利用的网络连接每个阶段都会比前一阶段翻番，直到所有的进程接受到数据为止。

你觉得你能用代码把这个算法实现么？实现这个算法有点超出我们这个课的主要目的了，如果你觉得你足够勇敢的话，可以去看这本超酷的书：Parallel Programming with MPI 这本书里面有完整的代码。

MPI\_Bcast 和 MPI\_Send 以及 MPI\_Recv 的比较

MPI\_Bcast 的实现使用了一个类似的树形广播算法来获得比较好的网络利用率。我们的实现跟 MPI\_Bcast 比起来怎么样呢？我们可以运行 compare\_bcast，在课程代码里我们提供了这个程序 (compare\_bcast.c)。在看代码之前，先让我们看一个 MPI 跟时间相关的函数 - MPI\_Wtime。MPI\_Wtime 不接收参数，它仅仅返回以浮点数形式展示的从1970-01-01到现在为止进过的秒数，跟 C 语言的 time 函数类似。我们可以多次调用 MPI\_Wtime 函数，并去差值，来计算我们的代码运行的时间。

让我们看一下我们的比较代码：

for (i = 0; i < num\_trials; i++) {

// Time my\_bcast

// Synchronize before starting timing

MPI\_Barrier(MPI\_COMM\_WORLD);

total\_my\_bcast\_time -= MPI\_Wtime();

my\_bcast(data, num\_elements, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Synchronize again before obtaining final time

MPI\_Barrier(MPI\_COMM\_WORLD);

total\_my\_bcast\_time += MPI\_Wtime();

// Time MPI\_Bcast

MPI\_Barrier(MPI\_COMM\_WORLD);

total\_mpi\_bcast\_time -= MPI\_Wtime();

MPI\_Bcast(data, num\_elements, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

total\_mpi\_bcast\_time += MPI\_Wtime();

}

代码里的 num\_trials 是一个指明一共要运行多少次实验的变量。我们分别记录两个函数运行所需的累加时间，平均的时间会在程序结束的时候打印出来。完整的代码在 compare\_bcast.c

如果你从这个 repo tutorials 目录下面运行这个程序的话，输出看起来应该像这样：

>>> cd tutorials

>>> ./run.py compare\_bcast

/home/kendall/bin/mpirun -n 16 -machinefile hosts ./compare\_bcast 100000 10

Data size = 400000, Trials = 10

Avg my\_bcast time = 0.510873

Avg MPI\_Bcast time = 0.126835

我们指定了16个进程来运行代码，每次广播发送 100,000 个整数，然后每次运行跑10个循环。如你所见，我的实验使用了通过网络连接起来的16个进程，结果显示运行我们的实现和 MPI 官方的实现体现了明显的时间差异。这里是一些不同进程数目运行时候的时间差异：

Processors my\_bcast MPI\_Bcast

2 0.0344 0.0344

4 0.1025 0.0817

8 0.2385 0.1084

16 0.5109 0.1296

可以看到，2个进程运行的时候是没有时间差异的。这是因为 MPI\_Bcast 的树算法在使用两个进程的时候并没有提供额外的网络利用率。然而，进程数量稍微增加到即使只有16个的时候我们也可以看到明显的差异。

试着自己运行一下代码，用更多的进程试试！

MPI Scatter, Gather, and Allgather

在之前的课程里，我们讲述了集体通信的必要知识点。我们讲了基础的广播通信机制 - MPI\_Bcast。在这节课里，我们会讲述两个额外的机制来补充集体通信的知识 - MPI\_Scatter 以及 MPI\_Gather。我们还会讲一个 MPI\_Gather 的变体：MPI\_Allgather。

注意 - 这个网站的提到的所有代码都在 GitHub 上面。这篇教程的代码在 tutorials/mpi-scatter-gather-and-allgather/code。

MPI\_Scatter 的介绍

MPI\_Scatter 是一个跟 MPI\_Bcast 类似的集体通信机制（如果你对这些词汇不熟悉的话，请阅读上一节课。MPI\_Scatter 的操作会设计一个指定的根进程，根进程会将数据发送到 communicator 里面的所有进程。MPI\_Bcast 和 MPI\_Scatter 的主要区别很小但是很重要。MPI\_Bcast 给每个进程发送的是同样的数据，然而 MPI\_Scatter 给每个进程发送的是一个数组的一部分数据。下图进一步展示了这个区别。

MPI\_Bcast vs MPI\_Scatter

在图中我们可以看到，MPI\_Bcast 在根进程上接收一个单独的数据元素（红色的方块），然后把它复制到所有其他的进程。MPI\_Scatter 接收一个数组，并把元素按进程的秩分发出去。第一个元素（红色方块）发往进程0，第二个元素（绿色方块）发往进程1，以此类推。尽管根进程（进程0）拥有整个数组的所有元素，MPI\_Scatter 还是会把正确的属于进程0的元素放到这个进程的接收缓存中。下面的 MPI\_Scatter 函数的原型。

MPI\_Scatter(

void\* send\_data,

int send\_count,

MPI\_Datatype send\_datatype,

void\* recv\_data,

int recv\_count,

MPI\_Datatype recv\_datatype,

int root,

MPI\_Comm communicator)

这个函数看起来确实很大很吓人，别怕，我们来详细解释一下。第一个参数，send\_data，是在根进程上的一个数据数组。第二个和第三个参数，send\_count 和 send\_datatype 分别描述了发送给每个进程的数据数量和数据类型。如果 send\_count 是1，send\_datatype 是 MPI\_INT的话，进程0会得到数据里的第一个整数，以此类推。如果send\_count是2的话，进程0会得到前两个整数，进程1会得到第三个和第四个整数，以此类推。在实践中，一般来说send\_count会等于数组的长度除以进程的数量。除不尽怎么办？我们会在后面的课程中讲这个问题 :-)。

函数定义里面接收数据的参数跟发送的参数几乎相同。recv\_data 参数是一个缓存，它里面存了recv\_count个recv\_datatype数据类型的元素。最后两个参数，root 和 communicator 分别指定开始分发数组的根进程以及对应的communicator。

MPI\_Gather 的介绍

MPI\_Gather 跟 MPI\_Scatter 是相反的。MPI\_Gather 从好多进程里面收集数据到一个进程上面而不是从一个进程分发数据到多个进程。这个机制对很多平行算法很有用，比如并行的排序和搜索。下图是这个算法的一个示例。

MPI\_Gather

跟MPI\_Scatter类似，MPI\_Gather从其他进程收集元素到根进程上面。元素是根据接收到的进程的秩排序的。MPI\_Gather的函数原型跟MPI\_Scatter长的一样。

MPI\_Gather(

void\* send\_data,

int send\_count,

MPI\_Datatype send\_datatype,

void\* recv\_data,

int recv\_count,

MPI\_Datatype recv\_datatype,

int root,

MPI\_Comm communicator)

在MPI\_Gather中，只有根进程需要一个有效的接收缓存。所有其他的调用进程可以传递NULL给recv\_data。另外，别忘记recv\_count参数是从每个进程接收到的数据数量，而不是所有进程的数据总量之和。这一点对MPI初学者来说经常容易搞错。

使用 MPI\_Scatter 和 MPI\_Gather 来计算平均数

在这节课的代码里，我提供了一个用来计算数组里面所有数字的平均数的样例程序（avg.c）。尽管这个程序十分简单，但是它展示了我们如何使用MPI来把工作拆分到不同的进程上，每个进程对一部分数据进行计算，然后再把每个部分计算出来的结果汇集成最终的答案。这个程序有以下几个步骤：

在根进程（进程0）上生成一个充满随机数字的数组。

把所有数字用MPI\_Scatter分发给每个进程，每个进程得到的同样多的数字。

每个进程计算它们各自得到的数字的平均数。

根进程收集所有的平均数，然后计算这个平均数的平均数，得出最后结果。

代码里面有 MPI 调用的主要部分如下所示：

if (world\_rank == 0) {

rand\_nums = create\_rand\_nums(elements\_per\_proc \* world\_size);

}

// Create a buffer that will hold a subset of the random numbers

float \*sub\_rand\_nums = malloc(sizeof(float) \* elements\_per\_proc);

// Scatter the random numbers to all processes

MPI\_Scatter(rand\_nums, elements\_per\_proc, MPI\_FLOAT, sub\_rand\_nums,

elements\_per\_proc, MPI\_FLOAT, 0, MPI\_COMM\_WORLD);

// Compute the average of your subset

float sub\_avg = compute\_avg(sub\_rand\_nums, elements\_per\_proc);

// Gather all partial averages down to the root process

float \*sub\_avgs = NULL;

if (world\_rank == 0) {

sub\_avgs = malloc(sizeof(float) \* world\_size);

}

MPI\_Gather(&sub\_avg, 1, MPI\_FLOAT, sub\_avgs, 1, MPI\_FLOAT, 0,

MPI\_COMM\_WORLD);

// Compute the total average of all numbers.

if (world\_rank == 0) {

float avg = compute\_avg(sub\_avgs, world\_size);

}

代码开头根进程创建里一个随机数的数组。当MPI\_Scatter被调用的时候，每个进程现在都持有elements\_per\_proc个原始数据里面的元素。每个进程计算子数组的平均数，然后根进程收集这些平均数。然后总的平均数就可以在这个小的多的平均数数组里面被计算出来。

如果你运行这个repo下面tutorials目录下的代码，输出应该跟下面的类似。注意因为数字是随机生成的，所以你的最终结果可能跟我的不一样。

>>> cd tutorials

>>> ./run.py avg

/home/kendall/bin/mpirun -n 4 ./avg 100

Avg of all elements is 0.478699

Avg computed across original data is 0.478699

MPI\_Allgather 以及修改后的平均程序

到目前为止，我们讲解了两个用来操作多对一或者一对多通信模式的MPI方法，也就是说多个进程要么向一个进程发送数据，要么从一个进程接收数据。很多时候发送多个元素到多个进程也很有用（也就是多对多通信模式）。MPI\_Allgather就是这个作用。

对于分发在所有进程上的一组数据来说，MPI\_Allgather会收集所有数据到所有进程上。从最基础的角度来看，MPI\_Allgather相当于一个MPI\_Gather操作之后跟着一个MPI\_Bcast操作。下面的示意图显示了MPI\_Allgather调用之后数据是如何分布的。

MPI\_Allgather

就跟MPI\_Gather一样，每个进程上的元素是根据他们的秩为顺序被收集起来的，只不过这次是收集到了所有进程上面。很简单吧？MPI\_Allgather的方法定义跟MPI\_Gather几乎一样，只不过MPI\_Allgather不需要root这个参数来指定根节点。

MPI\_Allgather(

void\* send\_data,

int send\_count,

MPI\_Datatype send\_datatype,

void\* recv\_data,

int recv\_count,

MPI\_Datatype recv\_datatype,

MPI\_Comm communicator)

我把计算平均数的代码修改成了使用MPI\_Allgather来计算。你可以在all\_avg.c这个文件里看到源代码。主要的不同点如下所示。

// Gather all partial averages down to all the processes

float \*sub\_avgs = (float \*)malloc(sizeof(float) \* world\_size);

MPI\_Allgather(&sub\_avg, 1, MPI\_FLOAT, sub\_avgs, 1, MPI\_FLOAT,

MPI\_COMM\_WORLD);

// Compute the total average of all numbers.

float avg = compute\_avg(sub\_avgs, world\_size);

现在每个子平均数被MPI\_Allgather收集到了所有进程上面。最终平均数在每个进程上面都打印出来了。样例运行之后应该跟下面的输出结果类似。

>>> ./run.py all\_avg

/home/kendall/bin/mpirun -n 4 ./all\_avg 100

Avg of all elements from proc 1 is 0.479736

Avg of all elements from proc 3 is 0.479736

Avg of all elements from proc 0 is 0.479736

Avg of all elements from proc 2 is 0.479736

跟你注意到的一样，all\_avg.c 和 avg.c 之间的唯一的区别就是 all\_avg.c 使用MPI\_Allgather把平均数在每个进程上都打印出来了。

Performing Parallel Rank with MPI

In the previous lesson, we went over MPI\_Scatter, MPI\_Gather, and MPI\_Allgather. We are going to expand on basic collectives in this lesson by coding a useful function for your MPI toolkit - parallel rank.

Note - All of the code for this site is on GitHub. This tutorial’s code is under tutorials/performing-parallel-rank-with-mpi/code.

Parallel rank - problem overview

When processes all have a single number stored in their local memory, it can be useful to know what order their number is in respect to the entire set of numbers contained by all processes. For example, a user might be benchmarking the processors in an MPI cluster and want to know the order of how fast each processor is relative to the others. This information can be used for scheduling tasks and so on. As you can imagine, it is rather difficult to find out a number’s order in the context of all other numbers if they are spread across processes. This problem - the parallel rank problem - is what we are going to solve in this lesson.

An illustration of the input and output of parallel rank is below:

Parallel Rank

The processes in the illustration (labeled 0 through 3) start with four numbers - 5, 2, 7, and 4. The parallel rank algorithm then computes that process 1 has rank 0 in the set of numbers (i.e. the first number), process 3 has rank 1, process 0 has rank 2, and process 2 has the last rank in the set of numbers. Pretty simple, right?

Parallel rank API definition

Before we dive into solving the parallel rank problem, let’s first decide on how our function is going to behave. Our function needs to take a number on each process and return its associated rank with respect to all of the other numbers across all processes. Along with this, we will need other miscellaneous information, such as the communicator that is being used, and the datatype of the number being ranked. Given this function definition, our prototype for the rank function looks like this:

TMPI\_Rank(

void \*send\_data,

void \*recv\_data,

MPI\_Datatype datatype,

MPI\_Comm comm)

TMPI\_Rank takes a send\_data buffer that contains one number of datatype type. The recv\_data receives exactly one integer on each process that contains the rank value for send\_data. The comm variable is the communicator in which ranking is taking place.

Note - The MPI standard explicitly says that users should not name their own functions MPI\_<something> to avoid confusing user functions with functions in the MPI standard itself. Thus, we will prefix functions in these tutorials with T.

Solving the parallel rank problem

Now that we have our API definition, we can dive into how the parallel rank problem is solved. The first step in solving the parallel rank problem is ordering all of the numbers across all of the processes. This has to be accomplished so that we can find the rank of each number in the entire set of numbers. There are quite a few ways how we could accomplish this. The easiest way is gathering all of the numbers to one process and sorting the numbers. In the example code (tmpi\_rank.c), the gather\_numbers\_to\_root function is responsible for gathering all of the numbers to the root process.

// Gathers numbers for TMPI\_Rank to process zero. Allocates space for

// the MPI datatype and returns a void \* buffer to process 0.

// It returns NULL to all other processes.

void \*gather\_numbers\_to\_root(void \*number, MPI\_Datatype datatype,

MPI\_Comm comm) {

int comm\_rank, comm\_size;

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

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

// Allocate an array on the root process of a size depending

// on the MPI datatype being used.

int datatype\_size;

MPI\_Type\_size(datatype, &datatype\_size);

void \*gathered\_numbers;

if (comm\_rank == 0) {

gathered\_numbers = malloc(datatype\_size \* comm\_size);

}

// Gather all of the numbers on the root process

MPI\_Gather(number, 1, datatype, gathered\_numbers, 1,

datatype, 0, comm);

return gathered\_numbers;

}

The gather\_numbers\_to\_root function takes the number (i.e. the send\_data variable) to be gathered, the datatype of the number, and the comm communicator. The root process must gather comm\_size numbers in this function, so it mallocs an array of datatype\_size \* comm\_size length. The datatype\_size variable is gathered by using a new MPI function in this tutorial - MPI\_Type\_size. Although our code only supports MPI\_INT and MPI\_FLOAT as the datatype, this code could be extended to support datatypes of varying sizes. After the numbers have been gathered on the root process with MPI\_Gather, the numbers must be sorted on the root process so their rank can be determined.

Sorting numbers and maintaining ownership

Sorting numbers is not necessarily a difficult problem in our ranking function. The C standard library provides us with popular sorting algorithms like qsort. The difficulty in sorting with our parallel rank problem is that we must maintain the ranks that sent the numbers to the root process. If we were to sort the list of numbers gathered to the root process without attaching additional information to the numbers, the root process would have no idea how to send the numbers’ ranks back to the requesting processes!

In order to facilitate attaching the owning process to the numbers, we create a struct in the code that holds this information. Our struct definition is as follows:

// Holds the communicator rank of a process along with the

// corresponding number. This struct is used for sorting

// the values and keeping the owning process information

// intact.

typedef struct {

int comm\_rank;

union {

float f;

int i;

} number;

} CommRankNumber;

The CommRankNumber struct holds the number we are going to sort (remember that it can be a float or an int, so we use a union) and it holds the communicator rank of the process that owns the number. The next part of the code, the get\_ranks function, is responsible for creating these structs and sorting them.

// This function sorts the gathered numbers on the root process and

// returns an array of ordered by the process's rank in its

// communicator. Note - this function is only executed on the root

// process.

int \*get\_ranks(void \*gathered\_numbers, int gathered\_number\_count,

MPI\_Datatype datatype) {

int datatype\_size;

MPI\_Type\_size(datatype, &datatype\_size);

// Convert the gathered number array to an array of CommRankNumbers.

// This allows us to sort the numbers and also keep the information

// of the processes that own the numbers intact.

CommRankNumber \*comm\_rank\_numbers = malloc(

gathered\_number\_count \* sizeof(CommRankNumber));

int i;

for (i = 0; i < gathered\_number\_count; i++) {

comm\_rank\_numbers[i].comm\_rank = i;

memcpy(&(comm\_rank\_numbers[i].number),

gathered\_numbers + (i \* datatype\_size),

datatype\_size);

}

// Sort the comm rank numbers based on the datatype

if (datatype == MPI\_FLOAT) {

qsort(comm\_rank\_numbers, gathered\_number\_count,

sizeof(CommRankNumber), &compare\_float\_comm\_rank\_number);

} else {

qsort(comm\_rank\_numbers, gathered\_number\_count,

sizeof(CommRankNumber), &compare\_int\_comm\_rank\_number);

}

// Now that the comm\_rank\_numbers are sorted, make an array of rank

// values for each process. The ith element of this array contains

// the rank value for the number sent by process i.

int \*ranks = (int \*)malloc(sizeof(int) \* gathered\_number\_count);

for (i = 0; i < gathered\_number\_count; i++) {

ranks[comm\_rank\_numbers[i].comm\_rank] = i;

}

// Clean up and return the rank array

free(comm\_rank\_numbers);

return ranks;

}

The get\_ranks function first creates an array of CommRankNumber structs and attaches the communicator rank of the process that owns the number. If the datatype is MPI\_FLOAT, qsort is called with a special sorting function for our array of structs (see tmpi\_rank.c for the code). Likewise, we use a different sorting function if the datatype is MPI\_INT.

After the numbers are sorted, we must create an array of ranks in the proper order so that they can be scattered back to the requesting processes. This is accomplished by making the ranks array and filling in the proper rank values for each of the sorted CommRankNumber structs.

Putting it all together

Now that we have our two primary functions, we can put them all together into our TMPI\_Rank function. This function gathers the numbers to the root process, sorts the numbers to determine their ranks, and then scatters the ranks back to the requesting processes. The code is shown below:

// Gets the rank of the recv\_data, which is of type datatype. The rank

// is returned in send\_data and is of type datatype.

int TMPI\_Rank(void \*send\_data, void \*recv\_data, MPI\_Datatype datatype,

MPI\_Comm comm) {

// Check base cases first - Only support MPI\_INT and MPI\_FLOAT for

// this function.

if (datatype != MPI\_INT && datatype != MPI\_FLOAT) {

return MPI\_ERR\_TYPE;

}

int comm\_size, comm\_rank;

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

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

// To calculate the rank, we must gather the numbers to one

// process, sort the numbers, and then scatter the resulting rank

// values. Start by gathering the numbers on process 0 of comm.

void \*gathered\_numbers = gather\_numbers\_to\_root(send\_data, datatype,

comm);

// Get the ranks of each process

int \*ranks = NULL;

if (comm\_rank == 0) {

ranks = get\_ranks(gathered\_numbers, comm\_size, datatype);

}

// Scatter the rank results

MPI\_Scatter(ranks, 1, MPI\_INT, recv\_data, 1, MPI\_INT, 0, comm);

// Do clean up

if (comm\_rank == 0) {

free(gathered\_numbers);

free(ranks);

}

}

The TMPI\_Rank function uses the two functions we just created, gather\_numbers\_to\_root and get\_ranks, to get the ranks of the numbers. The function then performs the final MPI\_Scatter to scatter the resulting ranks back to the processes.

If you have had trouble following the solution to the parallel rank problem, I have included an illustration of the entire data flow of our problem using an example set of data:

Parallel Rank

Have any questions about how the parallel rank algorithm works? Leave them below!

Running our parallel rank algorithm

I have included a small program in the example code to help test out our parallel rank algorithm. The code can be viewed in the random\_rank.c file file in the lesson code.

The example application simply creates a random number on each process and calls TMPI\_Rank to get the rank of each number. If you run the random\_rank program from the tutorials directory of the repo, the output should look similar to this.

>>> cd tutorials

>>> ./run.py random\_rank

mpirun -n 4 ./random\_rank 100

Rank for 0.242578 on process 0 - 0

Rank for 0.894732 on process 1 - 3

Rank for 0.789463 on process 2 - 2

Rank for 0.684195 on process 3 - 1

MPI Reduce and Allreduce

In the previous lesson, we went over an application example of using MPI\_Scatter and MPI\_Gather to perform parallel rank computation with MPI. We are going to expand on collective communication routines even more in this lesson by going over MPI\_Reduce and MPI\_Allreduce.

Note - All of the code for this site is on GitHub. This tutorial’s code is under tutorials/mpi-reduce-and-allreduce/code.

An introduction to reduce

Reduce is a classic concept from functional programming. Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. For example, let’s say we have a list of numbers [1, 2, 3, 4, 5]. Reducing this list of numbers with the sum function would produce sum([1, 2, 3, 4, 5]) = 15. Similarly, the multiplication reduction would yield multiply([1, 2, 3, 4, 5]) = 120.

As you might have imagined, it can be very cumbersome to apply reduction functions across a set of distributed numbers. Along with that, it is difficult to efficiently program non-commutative reductions, i.e. reductions that must occur in a set order. Luckily, MPI has a handy function called MPI\_Reduce that will handle almost all of the common reductions that a programmer needs to do in a parallel application.

MPI\_Reduce

Similar to MPI\_Gather, MPI\_Reduce takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype for MPI\_Reduce looks like this:

MPI\_Reduce(

void\* send\_data,

void\* recv\_data,

int count,

MPI\_Datatype datatype,

MPI\_Op op,

int root,

MPI\_Comm communicator)

The send\_data parameter is an array of elements of type datatype that each process wants to reduce. The recv\_data is only relevant on the process with a rank of root. The recv\_data array contains the reduced result and has a size of sizeof(datatype) \* count. The op parameter is the operation that you wish to apply to your data. MPI contains a set of common reduction operations that can be used. Although custom reduction operations can be defined, it is beyond the scope of this lesson. The reduction operations defined by MPI include:

MPI\_MAX - Returns the maximum element.

MPI\_MIN - Returns the minimum element.

MPI\_SUM - Sums the elements.

MPI\_PROD - Multiplies all elements.

MPI\_LAND - Performs a logical and across the elements.

MPI\_LOR - Performs a logical or across the elements.

MPI\_BAND - Performs a bitwise and across the bits of the elements.

MPI\_BOR - Performs a bitwise or across the bits of the elements.

MPI\_MAXLOC - Returns the maximum value and the rank of the process that owns it.

MPI\_MINLOC - Returns the minimum value and the rank of the process that owns it.

Below is an illustration of the communication pattern of MPI\_Reduce.

MPI\_Reduce

In the above, each process contains one integer. MPI\_Reduce is called with a root process of 0 and using MPI\_SUM as the reduction operation. The four numbers are summed to the result and stored on the root process.

It is also useful to see what happens when processes contain multiple elements. The illustration below shows reduction of multiple numbers per process.

MPI\_Reduce

The processes from the above illustration each have two elements. The resulting summation happens on a per-element basis. In other words, instead of summing all of the elements from all the arrays into one element, the ith element from each array are summed into the ith element in result array of process 0.

Now that you understand how MPI\_Reduce looks, we can jump into some code examples.

Computing average of numbers with MPI\_Reduce

In the previous lesson, I showed you how to compute average using MPI\_Scatter and MPI\_Gather. Using MPI\_Reduce simplifies the code from the last lesson quite a bit. Below is an excerpt from reduce\_avg.c in the example code from this lesson.

float \*rand\_nums = NULL;

rand\_nums = create\_rand\_nums(num\_elements\_per\_proc);

// Sum the numbers locally

float local\_sum = 0;

int i;

for (i = 0; i < num\_elements\_per\_proc; i++) {

local\_sum += rand\_nums[i];

}

// Print the random numbers on each process

printf("Local sum for process %d - %f, avg = %f\n",

world\_rank, local\_sum, local\_sum / num\_elements\_per\_proc);

// Reduce all of the local sums into the global sum

float global\_sum;

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_FLOAT, MPI\_SUM, 0,

MPI\_COMM\_WORLD);

// Print the result

if (world\_rank == 0) {

printf("Total sum = %f, avg = %f\n", global\_sum,

global\_sum / (world\_size \* num\_elements\_per\_proc));

}

In the code above, each process creates random numbers and makes a local\_sum calculation. The local\_sum is then reduced to the root process using MPI\_SUM. The global average is then global\_sum / (world\_size \* num\_elements\_per\_proc). If you run the reduce\_avg program from the tutorials directory of the repo, the output should look similar to this.

>>> cd tutorials

>>> ./run.py reduce\_avg

mpirun -n 4 ./reduce\_avg 100

Local sum for process 0 - 51.385098, avg = 0.513851

Local sum for process 1 - 51.842468, avg = 0.518425

Local sum for process 2 - 49.684948, avg = 0.496849

Local sum for process 3 - 47.527420, avg = 0.475274

Total sum = 200.439941, avg = 0.501100

Now it is time to move on to the sibling of MPI\_Reduce - MPI\_Allreduce.

MPI\_Allreduce

Many parallel applications will require accessing the reduced results across all processes rather than the root process. In a similar complementary style of MPI\_Allgather to MPI\_Gather, MPI\_Allreduce will reduce the values and distribute the results to all processes. The function prototype is the following:

MPI\_Allreduce(

void\* send\_data,

void\* recv\_data,

int count,

MPI\_Datatype datatype,

MPI\_Op op,

MPI\_Comm communicator)

As you might have noticed, MPI\_Allreduce is identical to MPI\_Reduce with the exception that it does not need a root process id (since the results are distributed to all processes). The following illustrates the communication pattern of MPI\_Allreduce:

MPI\_Allreduce

MPI\_Allreduce is the equivalent of doing MPI\_Reduce followed by an MPI\_Bcast. Pretty simple, right?

Computing standard deviation with MPI\_Allreduce

Many computational problems require doing multiple reductions to solve problems. One such problem is finding the standard deviation of a distributed set of numbers. For those that may have forgotten, standard deviation is a measure of the dispersion of numbers from their mean. A lower standard deviation means that the numbers are closer together and vice versa for higher standard deviations.

To find the standard deviation, one must first compute the average of all numbers. After the average is computed, the sums of the squared difference from the mean are computed. The square root of the average of the sums is the final result. Given the problem description, we know there will be at least two sums of all the numbers, translating into two reductions. An excerpt from reduce\_stddev.c in the lesson code shows what this looks like in MPI.

rand\_nums = create\_rand\_nums(num\_elements\_per\_proc);

// Sum the numbers locally

float local\_sum = 0;

int i;

for (i = 0; i < num\_elements\_per\_proc; i++) {

local\_sum += rand\_nums[i];

}

// Reduce all of the local sums into the global sum in order to

// calculate the mean

float global\_sum;

MPI\_Allreduce(&local\_sum, &global\_sum, 1, MPI\_FLOAT, MPI\_SUM,

MPI\_COMM\_WORLD);

float mean = global\_sum / (num\_elements\_per\_proc \* world\_size);

// Compute the local sum of the squared differences from the mean

float local\_sq\_diff = 0;

for (i = 0; i < num\_elements\_per\_proc; i++) {

local\_sq\_diff += (rand\_nums[i] - mean) \* (rand\_nums[i] - mean);

}

// Reduce the global sum of the squared differences to the root

// process and print off the answer

float global\_sq\_diff;

MPI\_Reduce(&local\_sq\_diff, &global\_sq\_diff, 1, MPI\_FLOAT, MPI\_SUM, 0,

MPI\_COMM\_WORLD);

// The standard deviation is the square root of the mean of the

// squared differences.

if (world\_rank == 0) {

float stddev = sqrt(global\_sq\_diff /

(num\_elements\_per\_proc \* world\_size));

printf("Mean - %f, Standard deviation = %f\n", mean, stddev);

}

In the above code, each process computes the local\_sum of elements and sums them using MPI\_Allreduce. After the global sum is available on all processes, the mean is computed so that local\_sq\_diff can be computed. Once all of the local squared differences are computed, global\_sq\_diff is found by using MPI\_Reduce. The root process can then compute the standard deviation by taking the square root of the mean of the global squared differences.

Running the example code with the run script produces output that looks like the following:

>>> ./run.py reduce\_stddev

mpirun -n 4 ./reduce\_stddev 100

Mean - 0.501100, Standard deviation = 0.301126

Introduction to Groups and Communicators

In all previous tutorials, we have used the communicator MPI\_COMM\_WORLD. For simple applications, this is sufficient as we have a relatively small number of processes and we usually either want to talk to one of them at a time or all of them at a time. When applications start to get bigger, this becomes less practical and we may only want to talk to a few processes at once. In this lesson, we show how to create new communicators to communicate with a subset of the original group of processes at once.

Note - All of the code for this site is on GitHub. This tutorial’s code is under tutorials/introduction-to-groups-and-communicators/code.

Overview of communicators

As we have seen when learning about collective routines, MPI allows you to talk to all processes in a communicator at once to do things like distribute data from one process to many processes using MPI\_Scatter or perform a data reduction using MPI\_Reduce. However, up to now, we have only used the default communicator, MPI\_COMM\_WORLD.

For simple applications, it’s not unusual to do everything using MPI\_COMM\_WORLD, but for more complex use cases, it might be helpful to have more communicators. An example might be if you wanted to perform calculations on a subset of the processes in a grid. For instance, all processes in each row might want to sum a value. This brings us to the first and most common function used to create new communicators:

MPI\_Comm\_split(

MPI\_Comm comm,

int color,

int key,

MPI\_Comm\* newcomm)

As the name implies, MPI\_Comm\_split creates new communicators by “splitting” a communicator into a group of sub-communicators based on the input values color and key. It’s important to note here that the original communicator doesn’t go away, but a new communicator is created on each process. The first argument, comm, is the communicator that will be used as the basis for the new communicators. This could be MPI\_COMM\_WORLD, but it could be any other communicator as well. The second argument, color, determines to which new communicator each processes will belong. All processes which pass in the same value for color are assigned to the same communicator. If the color is MPI\_UNDEFINED, that process won’t be included in any of the new communicators. The third argument, key, determines the ordering (rank) within each new communicator. The process which passes in the smallest value for key will be rank 0, the next smallest will be rank 1, and so on. If there is a tie, the process that had the lower rank in the original communicator will be first. The final argument, newcomm is how MPI returns the new communicator back to the user.

Example of using multiple communicators

Now let’s look at a simple example where we attempt to split a single global communicator into a set of smaller communicators. In this example, we’ll imagine that we’ve logically laid out our original communicator into a 4x4 grid of 16 processes and we want to divide the grid by row. To do this, each row will get its own color. In the image below, you can see how each group of processes with the same color on the left ends up in its own communicator on the right.

MPI\_Comm\_split example

Let’s look at the code for this.

// Get the rank and size in the original communicator

int world\_rank, world\_size;

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

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

int color = world\_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the

// original rank for ordering

MPI\_Comm row\_comm;

MPI\_Comm\_split(MPI\_COMM\_WORLD, color, world\_rank, &row\_comm);

int row\_rank, row\_size;

MPI\_Comm\_rank(row\_comm, &row\_rank);

MPI\_Comm\_size(row\_comm, &row\_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",

world\_rank, world\_size, row\_rank, row\_size);

MPI\_Comm\_free(&row\_comm);

The first few lines get the rank and size for the original communicator, MPI\_COMM\_WORLD. The next line does the important operation of determining the “color” of the local process. Remember that color decides to which communicator the process will belong after the split. Next, we see the all important split operation. The new thing here is that we’re using the orignal rank (world\_rank) as the key for the split operation. Since we want all of the processes in the new communicator to be in the same order that they were in the original communicator, using the original rank value makes the most sense here as it will already be ordered correctly. After that, we print out the new rank and size just to make sure it works. Your output should look something like this:

WORLD RANK/SIZE: 0/16 ROW RANK/SIZE: 0/4

WORLD RANK/SIZE: 1/16 ROW RANK/SIZE: 1/4

WORLD RANK/SIZE: 2/16 ROW RANK/SIZE: 2/4

WORLD RANK/SIZE: 3/16 ROW RANK/SIZE: 3/4

WORLD RANK/SIZE: 4/16 ROW RANK/SIZE: 0/4

WORLD RANK/SIZE: 5/16 ROW RANK/SIZE: 1/4

WORLD RANK/SIZE: 6/16 ROW RANK/SIZE: 2/4

WORLD RANK/SIZE: 7/16 ROW RANK/SIZE: 3/4

WORLD RANK/SIZE: 8/16 ROW RANK/SIZE: 0/4

WORLD RANK/SIZE: 9/16 ROW RANK/SIZE: 1/4

WORLD RANK/SIZE: 10/16 ROW RANK/SIZE: 2/4

WORLD RANK/SIZE: 11/16 ROW RANK/SIZE: 3/4

WORLD RANK/SIZE: 12/16 ROW RANK/SIZE: 0/4

WORLD RANK/SIZE: 13/16 ROW RANK/SIZE: 1/4

WORLD RANK/SIZE: 14/16 ROW RANK/SIZE: 2/4

WORLD RANK/SIZE: 15/16 ROW RANK/SIZE: 3/4

Don’t be alarmed if yours isn’t in the right order. When you print things out in an MPI program, each process has to send its output back to the place where you launched your MPI job before it can be printed to the screen. This tends to mean that the ordering gets jumbled so you can’t ever assume that just because you print things in a specific rank order, that the output will actually end up in the same order you expect. The output was just rearranged here to look nice.

Finally, we free the communicator with MPI\_Comm\_free. This seems like it’s not an important step, but it’s just as important as freeing your memory when you’re done with it in any other program. When an MPI object will no longer be used, it should be freed so it can be reused later. MPI has a limited number of objects that it can create at a time and not freeing your objects could result in a runtime error if MPI runs out of allocatable objects.

Other communicator creation functions

While MPI\_Comm\_split is the most common communicator creation function, there are many others. MPI\_Comm\_dup is the most basic and creates a duplicate of a communicator. It may seem odd that there would exist a function that only creates a copy, but this is very useful for applications which use libraries to perform specialized functions, such as mathematical libraries. In these kinds of applications, it’s important that user codes and library codes do not interfere with each other. To avoid this, the first thing every application should do is to create a duplicate of MPI\_COMM\_WORLD, which will avoid the problem of other libraries also using MPI\_COMM\_WORLD. The libraries themselves should also make duplicates of MPI\_COMM\_WORLD to avoid the same problem.

Another function is MPI\_Comm\_create. At first glance, this function looks very similar to MPI\_Comm\_create\_group. Its signature is almost identical:

MPI\_Comm\_create(

MPI\_Comm comm,

MPI\_Group group,

MPI\_Comm\* newcomm)

The key difference however (besides the lack of the tag argument), is that MPI\_Comm\_create\_group is only collective over the group of processes contained in group, where MPI\_Comm\_create is collective over every process in comm. This is an important distinction as the size of communicators grows very large. If trying to create a subset of MPI\_COMM\_WORLD when running with 1,000,000 processes, it’s important to perform the operation with as few processes as possible as the collective becomes very expensive at large sizes.

There are other more advanced features of communicators that we do not cover here, such as the differences between inter-communicators and intra-communicators and other advanced communicator creation functions. These are only used in very specific kinds of applications which may be covered in a future tutorial.

Overview of groups

While MPI\_Comm\_split is the simplest way to create a new communicator, it isn’t the only way to do so. There are more flexible ways to create communicators, but they use a new kind of MPI object, MPI\_Group. Before going into lots of detail about groups, let’s look a little more at what a communicator actually is. Internally, MPI has to keep up with (among other things) two major parts of a communicator, the context (or ID) that differentiates one communicator from another and the group of processes contained by the communicator. The context is what prevents an operation on one communicator from matching with a similar operation on another communicator. MPI keeps an ID for each communicator internally to prevent the mixups. The group is a little simpler to understand since it is just the set of all processes in the communicator. For MPI\_COMM\_WORLD, this is all of the processes that were started by mpiexec. For other communicators, the group will be different. In the example code above, the group is all of the processes which passed in the same color to MPI\_Comm\_split.

MPI uses these groups in the same way that set theory generally works. You don’t have to be familiar with all of set theory to understand things, but it’s helpful to know what two operations mean. Here, instead of referring to “sets”, we’ll use the term “groups” as it applies to MPI. First, the union operation creates a new, (potentially) bigger set from two other sets. The new set includes all of the members of the first two sets (without duplicates). Second, the intersection operation creates a new, (potentially) smaller set from two other sets. The new set includes all of the members that are present in both of the original sets. You can see examples of both of these operations graphically below.

Group Operation Examples

In the first example, the union of the two groups {0, 1, 2, 3} and {2, 3, 4, 5} is {0, 1, 2, 3, 4, 5} because each of those items appears in each group. In the second example, the intersection of the two groups {0, 1, 2, 3}, and {2, 3, 4, 5} is {2, 3} because only those items appear in each group.

Using MPI groups

Now that we understand the fundamentals of how groups work, let’s see how they can be applied to MPI operations. In MPI, it’s easy to get the group of processes in a communicator with the API call, MPI\_Comm\_group.

MPI\_Comm\_group(

MPI\_Comm comm,

MPI\_Group\* group)

As mentioned above, a communicator contains a context, or ID, and a group. Calling MPI\_Comm\_group gets a reference to that group object. The group object works the same way as a communicator object except that you can’t use it to communicate with other ranks (because it doesn’t have that context attached). You can still get the rank and size for the group (MPI\_Group\_rank and MPI\_Group\_size, respectively). However, what you can do with groups that you can’t do with communicators is use it to construct new groups locally. It’s important to remember here the difference between a local operation and a remote one. A remote operation involves communication with other ranks where a local operation does not. Creating a new communicator is a remote operation because all processes need to decide on the same context and group, where creating a group is local because it isn’t used for communication and therefore doesn’t need to have the same context for each process. You can manipulate a group all you like without performing any communication at all.

Once you have a group or two, performing operations on them is straightforward. Getting the union looks like this:

MPI\_Group\_union(

MPI\_Group group1,

MPI\_Group group2,

MPI\_Group\* newgroup)

And you can probably guess that the intersection looks like this:

MPI\_Group\_intersection(

MPI\_Group group1,

MPI\_Group group2,

MPI\_Group\* newgroup)

In both cases, the operation is performed on group1 and group2 and the result is stored in newgroup.

There are many uses of groups in MPI. You can compare groups to see if they are the same, subtract one group from another, exclude specific ranks from a group, or use a group to translate the ranks of one group to another group. However, one of the recent additions to MPI that tends to be most useful is MPI\_Comm\_create\_group. This is a function to create a new communicator, but instead of doing calculations on the fly to decide the makeup, like MPI\_Comm\_split, this function takes an MPI\_Group object and creates a new communicator that has all of the same processes as the group.

MPI\_Comm\_create\_group(

MPI\_Comm comm,

MPI\_Group group,

int tag,

MPI\_Comm\* newcomm)

)

Example of using groups

Let’s look at a quick example of what using groups looks like. Here, we’ll use another new function which allows you to pick specific ranks in a group and construct a new group containing only those ranks, MPI\_Group\_incl.

MPI\_Group\_incl(

MPI\_Group group,

int n,

const int ranks[],

MPI\_Group\* newgroup)

With this function, newgroup contains the processes in group with ranks contained in ranks, which is of size n. Want to see how that works? Let’s try creating a communicator which contains the prime ranks from MPI\_COMM\_WORLD.

// Get the rank and size in the original communicator

int world\_rank, world\_size;

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

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

// Get the group of processes in MPI\_COMM\_WORLD

MPI\_Group world\_group;

MPI\_Comm\_group(MPI\_COMM\_WORLD, &world\_group);

int n = 7;

const int ranks[7] = {1, 2, 3, 5, 7, 11, 13};

// Construct a group containing all of the prime ranks in world\_group

MPI\_Group prime\_group;

MPI\_Group\_incl(world\_group, 7, ranks, &prime\_group);

// Create a new communicator based on the group

MPI\_Comm prime\_comm;

MPI\_Comm\_create\_group(MPI\_COMM\_WORLD, prime\_group, 0, &prime\_comm);

int prime\_rank = -1, prime\_size = -1;

// If this rank isn't in the new communicator, it will be

// MPI\_COMM\_NULL. Using MPI\_COMM\_NULL for MPI\_Comm\_rank or

// MPI\_Comm\_size is erroneous

if (MPI\_COMM\_NULL != prime\_comm) {

MPI\_Comm\_rank(prime\_comm, &prime\_rank);

MPI\_Comm\_size(prime\_comm, &prime\_size);

}

printf("WORLD RANK/SIZE: %d/%d \t PRIME RANK/SIZE: %d/%d\n",

world\_rank, world\_size, prime\_rank, prime\_size);

MPI\_Group\_free(&world\_group);

MPI\_Group\_free(&prime\_group);

MPI\_Comm\_free(&prime\_comm);

In this example, we construct a communicator by selecting only the prime ranks in MPI\_COMM\_WORLD. This is done with MPI\_Group\_incl and results in prime\_group. Next, we pass that group to MPI\_Comm\_create\_group to create prime\_comm. At the end, we have to be careful to not use prime\_comm on processes which don’t have it, therefore we check to ensure that the communicator is not MPI\_COMM\_NULL, which is returned from MPI\_Comm\_create\_group on the ranks not included in ranks.