开源地址：https://github.com/Vincent-Huang-2000/DATA301
感觉不错的话，可以在 Github 上给个 Star

这是 Term 2 的内容，大部分可以参考：
https://python-parallel-programmning-cookbook.readthedocs.io/zh_CN/latest/index.html

本文档未完结，最新修订日期：2023年6月19日

基本概念

MPI（消息传递接口）

自学必看：https://nyu-cds.github.io/python-mpi/

以下是关于MPI（消息传递接口）的一些关键信息和概念的总结：

MPI是一种并行计算方法，它允许多个计算资源同时被用来解决一个计算问题。这包括将一个问题分解成可以并发解决的离散部分，每个部分进一步被分解成一系列指令，这些指令在不同的处理器上同时执行1。

MPI的主要优点是它可以运行在共享内存或分布式内存架构上，可以应用于OpenMP无法解决的更广泛的问题，每个进程都有自己的局部变量，分布式内存计算机比大型共享内存计算机便宜1。

MPI的缺点包括从串行版本到并行版本需要更多的编程更改，可能更难调试，性能受节点间通信网络的限制1。

在MPI中，一个进程有时被称为一个任务，但这两个术语可以互换使用。此外，一个线程（或轻量级进程）是进程内的一个或多个控制线程，线程共享相同的地址空间1。

MPI是通过消息传递接口实现的，这是一个函数库（用于C/C++）或子程序库（用于Fortran）。MPI的历史可以追溯到1980年代的Argonne的P4和Oak Ridge PVM。MPI-1在1994年完成，MPI-2在1998年完成，MPI-3在2012年完成，MPI-3的特性逐渐添加到MPI实现中1。

MPI的主要编程方法是SPMD（单程序，多数据流），这意味着每个处理器都在不同的数据上执行相同的程序1。

各种MPI实现有一些共同点和差异。例如，它们在源代码兼容性（除并行I/O外）、程序应编译并按原样运行、对异构并行架构的支持等方面都有相似之处。而它们的差异主要在于编译和链接的命令、如何启动MPI程序、并行I/O（来自MPI-2）和调试1。

并行计算的目标是减少执行时间，包括计算时间、空闲时间（等待其他处理器的数据）和通信时间（处理器发送和接收消息所需的时间）1。

目前，我尚未总结完整个MPI使用教程的所有部分。我尚未涵盖如何在Python中使用MPI，以及MPI的具体函数和操作。如果你需要更多关于这些主题的信息，我会很乐意继续为你提供帮助。

Data dependencies

数据依赖（Data dependencies）是指程序中指令之间存在的数据相关性或依赖关系。它表示一个指令的执行结果作为另一个指令的输入数据，从而导致这两个指令之间存在依赖关系。

数据依赖可以通过对指令的操作数进行分析来确定。当一个指令需要使用另一个指令的执行结果作为自己的输入数据时，就存在数据依赖。数据依赖可以分为以下几种类型：

1. 写后读（Write-after-Read，WAR）依赖：一个指令在写入某个位置后，紧接着另一个指令需要读取该位置的值。也称为反相关（Anti-dependence）。

2. 写后写（Write-after-Write，WAW）依赖：两个指令都试图写入同一个位置，其中一个指令先写入，后面的指令再试图写入该位置。也称为输出相关（Output dependence）。

3. 读后写（Read-after-Write，RAW）依赖：一个指令在读取某个位置的值后，另一个指令需要写入该位置。也称为真相关（True dependence）或输入相关（Input dependence）。

数据依赖关系对于指令的并行执行和流水线处理是非常重要的。如果存在数据依赖关系，需要确保被依赖的指令先于依赖的指令执行，并且保证数据的正确传递。处理器和编译器通常会使用各种技术和优化策略来管理和解决数据依赖关系，以提高程序的性能和并行度。

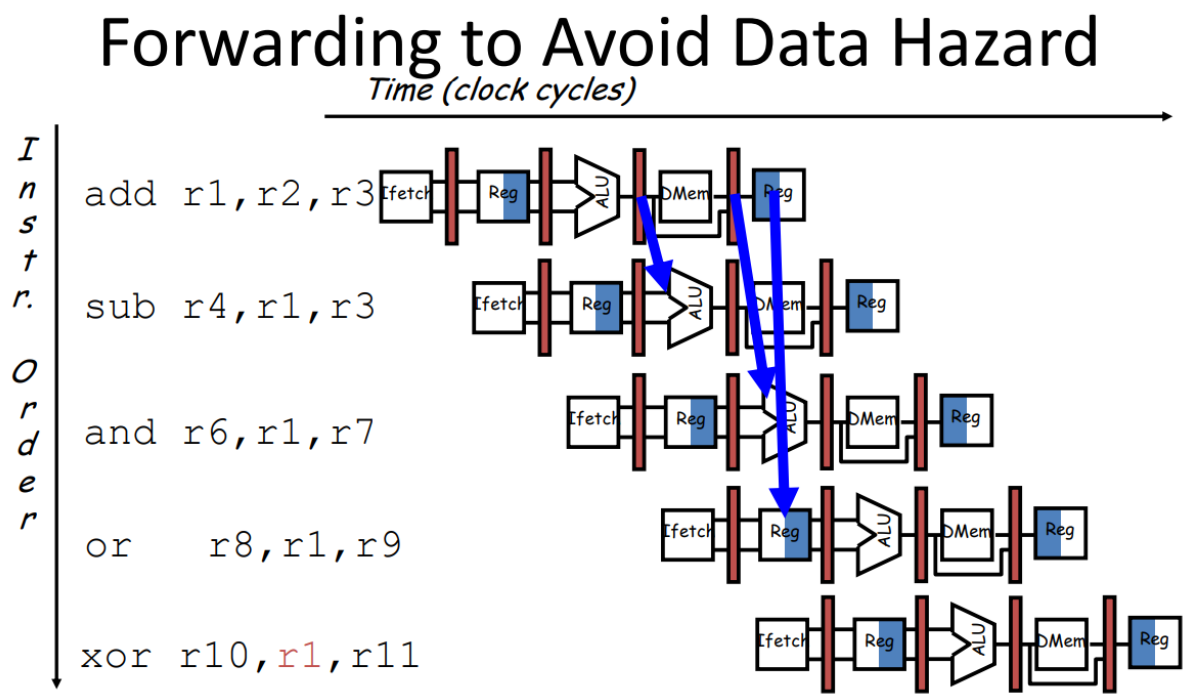
lecture note 题目：

What are the data dependencies in this code? (listed as out, in1, in2)
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11

listed as out, in1, in2 是说：例如 add r1,r2,r3 中，r1 就是输出寄存器，r2和r3为输入寄存器。

对于指令之间的数据依赖关系，主要关注输出寄存器是否被后续指令作为输入寄存器使用。显然，以上代码只有 r1 被后续的其他指令作为输入了，所以指令1是所有其他指令的数据依赖源，而其他指令之间没有直接的数据依赖关系。

Forwarding to Avoid Data Hazard



Forwarding（也称为数据前递、数据旁路）是一种用于避免数据冲突（Data Hazard）的技术。数据冲突指的是在流水线中由于数据依赖关系而导致的指令之间的数据相关性问题。

在流水线处理器中，指令的执行被划分为多个阶段，每个阶段执行不同的操作。当一个指令需要使用前面指令的执行结果时，如果前面指令的结果尚未写回到寄存器文件中，就会产生数据冲突。这可能导致后续指令需要等待前面指令的结果，从而引起流水线停顿，降低流水线的效率。

为了解决数据冲突问题，使用 Forwarding 将指令执行结果直接传递给需要使用该结果的指令，而无需等待结果写回到寄存器文件。这样，后续指令就可以立即使用最新的结果，避免流水线停顿。

Forwarding 可以通过在流水线的不同阶段之间建立临时数据路径来实现。当检测到数据依赖关系时，流水线会将最新的计算结果转发给需要使用该结果的指令，而不是等待结果写回到寄存器文件后再获取。这样，指令的执行可以继续进行，减少流水线停顿的次数。

通过使用数据前递技术，流水线处理器可以更有效地处理数据冲突，提高指令级并行度和流水线的吞吐量，从而提高计算机系统的性能。

Out of order execution

乱序执行是一种处理器的执行策略，它允许指令按照其可执行条件和数据依赖性的顺序进行执行，而不是按照它们在程序中的顺序执行。这种策略可以提高指令级并行度和流水线的利用率。

因为，让所有指令在相同的周期内执行可能不是最优的选择，不同的指令可能具有不同的性能要求或资源需求。下面是一个例子：

假设有一个程序包含两个指令：
1. 指令A：执行一次复杂的数学运算，需要较长的执行时间。
2. 指令B：执行简单的数据传输操作，需要较短的执行时间。

在一个要求所有指令在相同周期内执行的情况下，如果指令B被强制等待指令A的完成，那么指令B将浪费大量的时间在等待阶段，造成资源的浪费和效率的降低。

然而，如果允许乱序执行，处理器可以在指令A执行时，提前执行指令B，利用空闲的资源完成更多的工作。这样，指令B可以在指令A执行的过程中，并行地进行，最大限度地提高资源利用率和整体性能。

Extension for FP Pipelining

- FP operations are long and cannot be completed in 5 cycles (EX lasts more than 1 cycle)
  - Simply imagine that EX stage is duplicated for FP
  - There are multiple FP functional units

	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
FP Instruction *i+1*		IF	ID	EX	EX	EX	EX	MEM	WB
Instruction *i+2*			IF	ID	EX	MEM	WB		

在浮点数流水线中，由于浮点数操作的执行时间较长，无法在规定的5个周期内完成。为了解决这个问题，我们可以简单地想象，在执行阶段（EX stage）为浮点数操作进行复制，即增加多个相同的执行阶段。

扩展后的浮点数流水线中，可以同时执行多个浮点数操作，每个操作都有自己的执行阶段（EX stage）。这样，在每个时钟周期内，可以有多个浮点数操作同时进行，提高浮点数操作的并行度和整体执行效率。

通过多个浮点数功能单元的并行执行，可以充分利用处理器的资源，加速浮点数操作的执行。扩展后的浮点数流水线可以提高计算机系统中浮点数操作的处理能力，使其能够更高效地执行长时间的浮点数操作。

FP Pipelining（浮点数流水线）：指的是对浮点数操作进行流水线化，将其分解为多个阶段并依次执行。这样可以提高浮点数操作的执行效率和吞吐量。

Extension（扩展）：在此上下文中，指的是对浮点数流水线进行扩展，以适应执行时间较长的浮点数操作。

EX stage（执行阶段）：在流水线中，EX（Execute）阶段是指令执行的阶段，其中实际执行浮点数操作的计算发生。

Functional units（功能单元）：在处理器中，功能单元是执行特定操作的硬件单元。在此处，指的是用于执行浮点数操作的多个功能单元。

Extension for FP Pipelining（针对浮点数流水线的扩展）的概念是指在浮点数流水线中，为了适应执行时间较长的浮点数操作而进行的扩展。

Floating-Point Pipelining

Floating-Point Pipelining（浮点数流水线）：就是上面的 FP Pipelining

EXE cycle（执行周期）：在流水线中，EXE（Execute）周期是指令执行的阶段，其中实际执行指令操作的计算发生。

Floating-Point Pipelining（浮点数流水线）是指对浮点数指令进行流水线化处理的技术。简单地让所有指令在相同的周期内执行不是一种可接受的解决方案，特别是对于执行时间较长的指令，如浮点数操作。

浮点数指令与整数指令使用相同的流水线结构，但有一个关键区别。在浮点数流水线中，执行阶段（EXE cycle）会根据指令的类型（如MUL、ADD、DIV）重复多次，以完成相应的操作。

这意味着浮点数指令在执行阶段（EXE cycle）上可以重复多次，以便根据操作类型完成适当的计算。不同的浮点数操作可能需要不同的执行周期数量来完成。

此外，浮点数流水线可能会具有多个浮点数功能单元。这些功能单元是专门用于执行浮点数操作的硬件单元。多个功能单元的存在使得浮点数指令能够并行执行，从而提高浮点数操作的并行度和整体执行效率。

通过浮点数流水线的设计和浮点数功能单元的并行执行，计算机系统可以更高效地处理浮点数操作，提高系统性能和浮点数运算的吞吐量。

"EXE cycle is repeated as many times as needed to complete the operations based on the type of instructions, e.g., MUL, ADD, DIV" 这句话的含义是根据指令的类型（如MUL、ADD、DIV），执行阶段（EXE cycle）会根据需要重复多次，直到完成相应的操作。

在浮点数流水线中，不同的浮点数操作可能需要不同的执行时间。例如，乘法（MUL）操作可能需要更长的时间来完成，而加法（ADD）和除法（DIV）操作可能需要相对较短的时间。
为了适应这种差异，执行阶段（EXE cycle）会根据指令的类型进行动态调整。当遇到需要更多周期才能完成的操作时，流水线会重复执行执行阶段（EXE cycle），直到该操作完成。

举个例子，假设有一个乘法指令（MUL）在执行阶段（EXE cycle）需要3个周期才能完成。当该指令进入执行阶段时，流水线会重复执行执行阶段3次，以确保乘法操作得到正确的结果。

类似地，对于其他类型的指令（如加法或除法），流水线会根据需要重复执行执行阶段，以保证每个操作都能在正确的周期内完成。

这种动态地根据指令类型重复执行执行阶段，使得浮点数指令能够适应不同的操作时间要求，提高了浮点数操作的并行度和整体执行效率。

引入"执行阶段"这个概念是为了描述指令在流水线中的执行过程，并将其划分为不同的阶段。它有助于组织和管理指令的执行，以实现更高的并行度和更高的处理器性能。

在流水线处理器中，指令的执行被划分为多个阶段，每个阶段负责完成特定的操作。常见的流水线阶段包括：

- 取指令（Instruction Fetch，IF）、
- 译码（Instruction Decode，ID）、
- 执行（Execute，EXE）、
- 访存（Memory Access，MEM）和
- 写回（Write Back，WB）等。

引入执行阶段的目的有以下几点：

1. 分解指令执行过程：将指令执行过程划分为多个阶段，有助于将复杂的指令处理过程分解成可管理和并行执行的部分。每个阶段可以专注于特定的操作，提高了流水线的效率。

2. 实现并行执行：通过将指令的不同阶段并行执行，可以同时处理多条指令，提高处理器的并行度。执行阶段的存在允许不同的指令在同一时钟周期内进行不同的操作，提高了指令级并行度和整体性能。

3. 资源利用：执行阶段可以帮助优化处理器资源的利用。例如，当一个指令的执行阶段需要更长的时间时，可以让其他指令在该阶段之间并行执行，以充分利用处理器的资源。

4. 引入流水线停顿：执行阶段的引入还使得检测和处理数据相关性以及其他冲突变得更加方便。当检测到相关冲突时，可以通过在执行阶段添加流水线停顿（stall）来解决冲突，保证数据的正确传递和指令的顺序执行。

总之，引入执行阶段的概念有助于划分和管理指令的执行过程，实现流水线的并行执行，并提高处理器的效率和性能。执行阶段的存在使得指令级并行度成为可能，并为处理器设计者提供了更多的优化和调整机会。

Flynn’s Taxonomy

弗林分类（Flynn's Taxonomy）是一种用于描述计算机体系结构的分类系统，由迈克尔·弗林（Michael Flynn）在1966年提出。该分类基于指令流和数据流的数量，将计算机体系结构划分为四个主要类别。

1. 单指令流单数据流（Single Instruction Stream, Single Data Stream，SISD）：这种体系结构中，只有一个指令流和一个数据流。传统的单处理器计算机就是典型的SISD结构。它执行单个指令并处理单个数据元素。

2. 单指令流多数据流（Single Instruction Stream, Multiple Data Streams，SIMD）：这种体系结构中，只有一个指令流，但有多个数据流。SIMD结构常用于处理并行计算任务，其中多个处理单元同时执行相同的指令，但对不同的数据元素进行操作。SIMD体系结构可以进一步细分为以下几种类型：
   - 向量架构（Vector Architectures）：使用向量处理器执行同一指令的并行向量操作。
   - 多媒体扩展（Multimedia Extensions）：增加了对多媒体数据处理的专用指令和硬件支持。
   - 图形处理器单元（Graphics Processor Units，GPU）：专门用于图形渲染和并行计算的处理器。

3. 多指令流单数据流（Multiple Instruction Streams, Single Data Stream，MISD）：这种体系结构中，存在多个指令流，但只有一个数据流。尽管理论上存在这种分类，但目前没有商业实现。

4. 多指令流多数据流（Multiple Instruction Streams, Multiple Data Streams，MIMD）：这种体系结构中，同时存在多个指令流和多个数据流。MIMD结构允许多个处理单元独立地执行不同的指令，并处理不同的数据元素。MIMD体系结构可以进一步细分为以下两种类型：
   - 紧耦合MIMD（Tightly-Coupled MIMD）：多个处理单元共享内存，彼此之间可以相互访问和通信。
   - 松耦合MIMD（Loosely-Coupled MIMD）：多个处理单元具有独立的内存和指令流，彼此之间通常通过消息传递进行通信。

弗林分类提供了对不同计算机体系结构的有用框架，可以用于理解和比较不同类型的并行计算系统。

Memory Hierarchy

内存层次结构（Memory Hierarchy）是一种结构，它根据访问速度、容量和成本将内存组织成多个层次。这些层次从最近的CPU（最快速、最小容量、最昂贵）开始，逐步扩展到最远的存储设备（最慢、最大容量、最便宜）。理想的情况下，该结构使得计算机系统能够提供接近最快内存速度的性能，同时仍然能够利用大容量存储设备的成本效益。

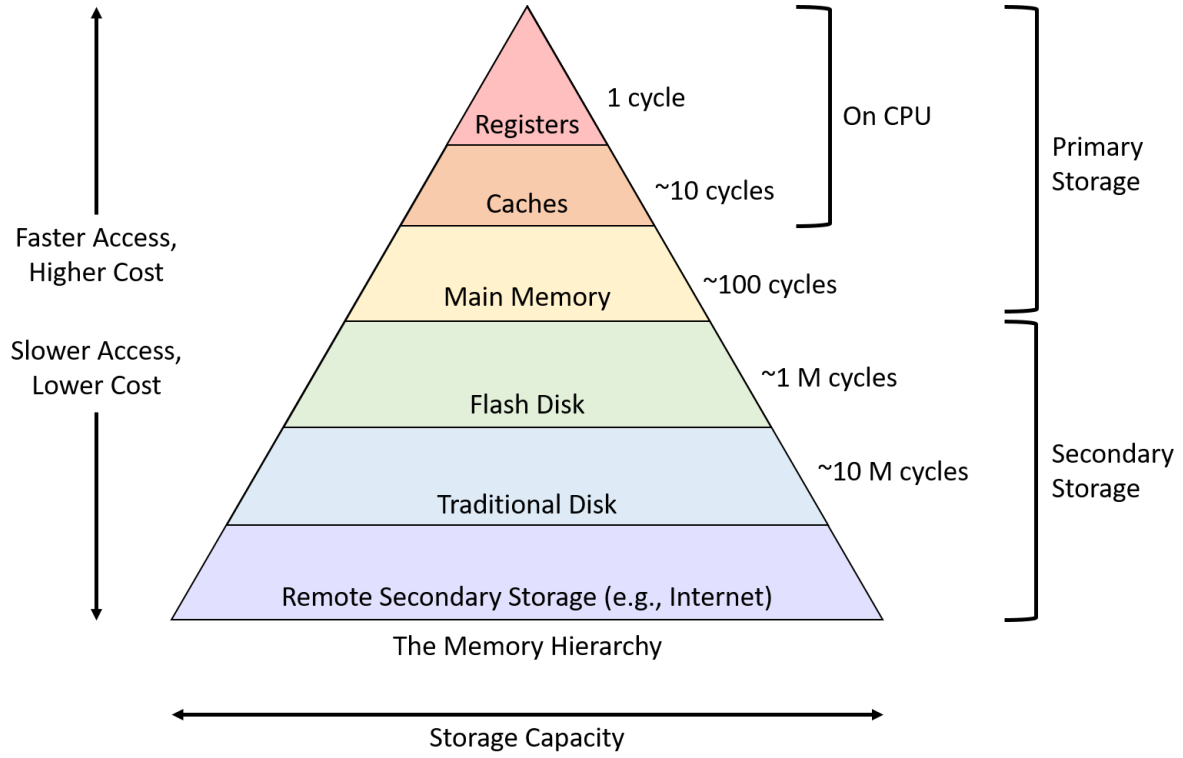
下面是一种常见的内存层次结构：

寄存器（Registers）：这是位于CPU内部的内存，它们有极限的容量，但访问速度最快。寄存器存储的是当前正在执行的指令需要的数据。

高速缓存（Cache）：高速缓存是一种小而快速的内存，位于CPU和主内存之间。它的目的是暂时存储那些最近或频繁访问的数据，以减少从主内存中获取数据的时间。

主内存（Main Memory / RAM）：主内存也称为随机访问内存（RAM），它比高速缓存有更大的容量，但访问速度较慢。主内存存储的是当前正在运行的程序和数据。

辅助存储（Secondary Storage）：这是存储设备中最慢和最大的存储层次，包括硬盘驱动器（HDD）、固态驱动器（SSD）和光盘等。它用于长期存储数据，即使电源关闭也不会丢失数据。



Distributed Memory Model

在 Distributed Memory Model（分布式内存模型）中，每个处理器都有自己的私有内存。处理器之间必须通过一种通信协议（如消息传递）来交换数据。这种模型的优点是可以很好地扩展到大规模系统，并且避免了共享内存模型中的并发控制问题。然而，编程会更复杂，因为处理器之间的数据交换需要通过明确的通信操作来完成。

总的来说，选择哪种模型取决于你的具体需求。如果你的任务可以轻易地分解成许多相互独立的子任务，那么分布式内存模型可能更合适。如果你的任务需要频繁的数据共享和交互，那么共享内存模型可能更合适。在实际应用中，许多系统实际上使用的是这两种模型的混合形式。

假设我们正在设计一个大规模的天气预测系统。在这个系统中，我们需要处理来自全球各地的大量气象数据。每个地理区域的数据可以在一个单独的处理节点上进行处理，然后通过消息传递来交换数据和结果。这样，每个处理节点只需要处理一部分数据，大大减轻了单个节点的负担，并且可以并行处理多个地区的数据。这就是一个典型的分布式内存模型的应用。在这种模型中，每个处理节点都有自己的内存，而且只能访问自己的内存。处理节点之间的通信需要通过消息传递的方式进行。

Shared Memory Model

在 Shared Memory Model（共享内存模型）中，所有处理器或线程共享同一块内存空间。当一个处理器在内存中修改数据时，这些更改对所有其他处理器都是可见的。这种模型的优点是编程相对简单，因为所有处理器都可以直接访问内存中的任何位置。然而，这也带来了并发控制的问题，因为必须小心处理并发访问共享数据的情况。

假设我们正在设计一个多线程的搜索引擎。在这个搜索引擎中，我们有一份巨大的文档库需要被搜索。每次用户提交一个查询，我们会派发多个线程去文档库中搜索符合条件的文档。在这种情况下，共享内存模型非常适用。所有的线程都能够直接访问到文档库（即共享内存），并且可以并行地执行搜索操作。需要注意的是，我们需要确保线程之间正确地管理对共享资源（文档库）的访问，避免数据竞争等问题。

Shared Memory Model or Distributed Memory Model

选择使用共享内存模型（Shared Memory Model）还是分布式内存模型（Distributed Memory Model）主要取决于几个因素：

1. 任务的性质：如果任务需要大量的数据交换和通信，那么共享内存模型可能更合适，因为在这种模型中，所有的线程或进程都可以直接访问内存。但是，如果任务可以被划分为许多相对独立的子任务，并且每个子任务可以在自己的数据上独立工作，那么分布式内存模型可能更合适。

2. 硬件和系统的规模：共享内存模型在小到中等规模的系统中比较常见，因为所有的处理器都需要能够直接访问内存，这在大规模系统中可能难以实现。相反，分布式内存模型可以更好地扩展到大规模的系统，因为每个处理器都有自己的内存。

3. 编程的复杂性：共享内存模型通常更容易编程，因为所有的线程或进程都可以直接访问内存。然而，这也带来了并发控制的挑战，因为我们需要确保对共享数据的访问是正确的。分布式内存模型在编程上可能更复杂，因为处理器之间需要通过消息传递来交换数据。

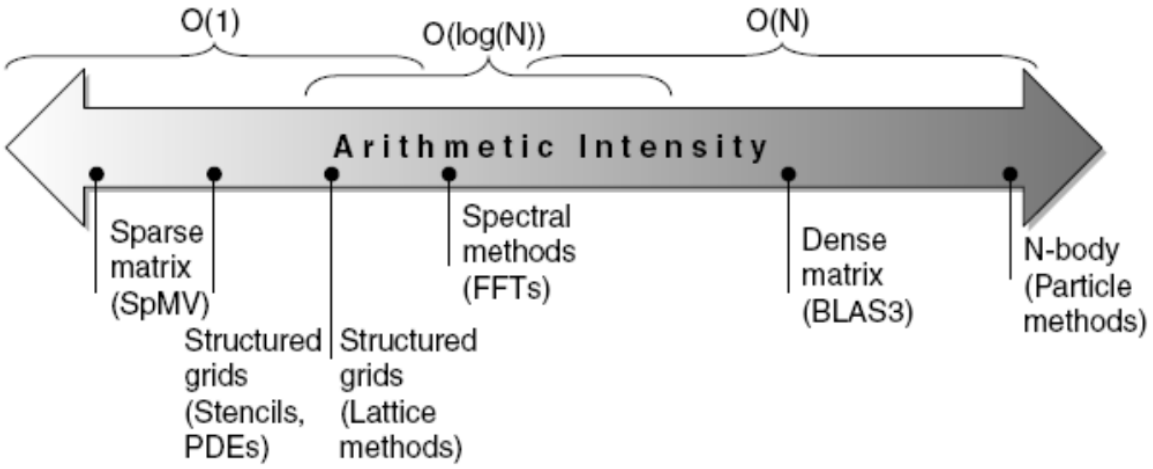
总的来说，没有一种模型是在所有情况下都是最好的。你需要根据你的具体需求和环境来选择最合适的模型。在实践中，许多系统实际上使用的是这两种模型的混合形式。

Roofline Performance Model

基本思想：Arithmetic intensity（算术强度）
– 每字节读取的浮点运算
– 量化计算和通信之间的关系

我们的 Spark 框架在计算和通信顺序相同时运行良好，即算术强度 O(1)

> O(1) 为使用共享内存模型的低级数据并行提供了机会



Roofline模型是一种视觉化的工具，它可以帮助研究人员和开发人员理解并优化并行计算的性能。该模型在2009年由Williams等人首次提出，用于分析和理解计算机系统的峰值性能和内存带宽限制。

Roofline模型图的X轴表示算术密度，即每个字节的内存带宽所能执行的浮点操作次数（FLOPs/Byte）。Y轴表示性能，即每秒的浮点操作次数（FLOPs/second）。

图中有两个主要的部分：roofline和ridge line。Roofline是系统的性能上限，它由两部分组成：一部分是由处理器的峰值性能决定的，另一部分是由内存带宽决定的。当算术密度较低时，性能主要受到内存带宽的限制；而当算术密度较高时，性能主要受到处理器峰值性能的限制。

Ridge line则表示一种特定类型的算法或应用程序在不同的算术密度下的性能。通过将ridge line和roofline进行比较，我们可以理解一个程序的性能瓶颈在哪里，以及如何改进代码以达到更好的性能。

Roofline模型提供了一种直观的方式来理解并行计算的性能，它可以帮助我们识别性能瓶颈，以及理解如何通过改进算法和优化代码来提高性能。

SIMD Parallelism

SIMD代表"Single Instruction, Multiple Data"，即单指令流，多数据流。这是一种并行计算的形式，其中一个操作可以同时应用于多个数据点。

在SIMD并行中，一组数据可以同时进行相同的操作。例如，你可能有一个数据向量，你想要对向量中的每个元素都加上一个常数。在传统的序列计算中，你需要一个一个地处理每个元素。但在SIMD并行中，你可以同时对所有元素进行操作。

SIMD并行被广泛应用在各种硬件和软件中。例如，许多现代处理器都包含SIMD指令集，如Intel的AVX指令集，或ARM的NEON指令集。这些指令集允许程序员编写可以并行处理多个数据的代码。在软件中，一些针对并行计算优化的编程语言和库，如OpenMP和CUDA，也支持SIMD并行。

需要注意的是，SIMD并行对于某些类型的问题非常有效，特别是那些可以被划分为许多相互独立，需要执行相同操作的子任务的问题。然而，如果问题无法被有效地分解为这种形式，或者不同的数据需要执行不同的操作，那么SIMD并行可能不是最好的选择。

应用场景：

- Vector architectures 向量运算：向量和矩阵运算是SIMD并行的主要应用场景，例如在物理模拟、计算机图形学和机器学习中广泛使用的矩阵乘法和矩阵向量乘法。
- SIMD extensions
- Graphics Processor Units (GPUs)
- 数字信号处理：在处理音频、图像或视频数据时，SIMD并行可以提高效率。例如，许多图像和视频编解码器使用SIMD指令来加速像素级的操作。
- 数据压缩和加密：数据压缩和加密算法往往需要对大量数据进行相同的操作，这使得它们可以从SIMD并行中受益。

MIMD Parallelism

MIMD代表"Multiple Instruction, Multiple Data"，即多指令流，多数据流。这也是一种并行计算的形式，在MIMD并行中，多个处理器可以同时执行不同的指令序列，且每个指令序列可以在不同的数据上进行操作。

这与SIMD（单指令流，多数据流）并行形成了对比。在SIMD并行中，所有的数据都执行相同的操作，而在MIMD并行中，不同的处理器可以执行不同的操作。MIMD并行的一个典型例子就是多核处理器中的并行计算：每个处理器核心都可以执行自己的指令流，并且在自己的数据上进行操作。

在实际应用中，SIMD和MIMD并行通常是同时使用的。例如，一个多核处理器（MIMD）可能会在每个核心上执行SIMD操作。这样，我们可以在处理器核心之间（MIMD）以及核心内部（SIMD）实现并行性，从而得到更高的性能。

选择使用SIMD并行还是MIMD并行，或者它们的组合，主要取决于你的具体任务和硬件。有些任务可能更适合使用SIMD并行（例如，对数组中的所有元素执行相同的操作），而有些任务可能更适合使用MIMD并行（例如，每个处理器核心执行不同的任务）。

应用场景：

- 并行算法：许多并行算法可以利用MIMD并行来提高效率，例如并行排序和搜索算法。
- 分布式计算：在分布式系统中，每个节点可以执行不同的任务，这是MIMD并行的一个典型应用。例如，在MapReduce框架中，map阶段和reduce阶段的任务可以在不同的节点上并行执行。
- 多线程编程：在多线程编程中，每个线程可以执行不同的任务，这也是MIMD并行的一个应用。

MPI 编程指南

参考文档：
https://mpi4py.readthedocs.io/en/stable/
https://pypi.org/project/mpi4py/
https://youtu.be/M9kKJtq5twk 英文视频教程（20分钟）
https://zhuanlan.zhihu.com/p/25332041 Python多进程并行编程实践
https://python-parallel-programmning-cookbook.readthedocs.io/zh_CN/latest/chapter3/11_Using_the_mpipy_Python_module.html Python并行编程 中文版

简介

Python MPI 是一个用于 Python 编程语言的消息传递接口（MPI）标准的绑定，允许任何 Python 程序利用多个处理器。

MPI（Message Passing Interface）是一个函数库，早在 1980 年代就开始了早期的消息传递。MPI 的主要目标是通过并行计算来减少执行时间，包括计算时间、空闲时间（等待其他处理器的数据）以及通信时间（处理器发送和接收消息所花费的时间）。
有两种主要的并行编程方法：共享内存计算机（多核 CPU 共享全局内存空间）和分布式内存（例如计算集群，每个节点使用其本地内存，节点之间通过消息进行通信）。

安装与使用

以下的代码适用于 Colab

安装 mpi4py 模块

```
!pip install mpi4py
```

注意的是，我们在 Colab 中使用 mpi4py 需要将 Python 代码写到文件里，然后再通过指令来运行 Python 文件。我们需要将 Python 代码放到单独一个 Colab 代码框，在第一行添加：

```
%%writefile mpi1.py
```

其中 `mpi1.py` 是需要保存为的 Python 文件名

```
%%writefile mpi1.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
print('size=%d, rank=%d' % (size, rank))
```

```
Overwriting mpi1.py
```

```
!mpiexec --allow-run-as-root --oversubscribe -n 4 python mpi1.py
```

这个命令是使用 mpiexec 来启动 MPI 程序。以下是命令中各部分的解释：

- mpiexec：这是用于启动 MPI 程序的命令。它会启动指定数量的进程，并让它们运行指定的程序。
- --allow-run-as-root：这个参数允许你以 root 用户身份运行 MPI 程序。在某些环境（例如 Docker 容器或某些云服务）中，你可能需要以 root 用户身份运行程序。
- --oversubscribe：这个参数允许你在每个节点上启动的进程数超过该节点的物理核心数。这对于一些并行任务（例如，那些在等待 I/O 操作时会阻塞的任务）可能很有用，因为在一个核心上运行多个进程可以让其他进程在一个进程阻塞时继续运行。
- -n 4：这个参数指定你想要启动的进程数。在这个例子中，你将启动 4 个进程。
- python mpi1.py：这是你想要运行的程序。mpiexec 将会在每个进程上运行这个程序。

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
  randNum = numpy.random.random_sample(1)
  print("Process", rank, "drew the number", randNum[0])
  comm.Send(randNum, dest=0)

if rank == 0:
  print("Process", rank, "before receiving has the number", randNum[0])
  comm.Recv(randNum, source=1)
  print("Process", rank, "received the number", randNum[0])
```

这是一个非常基础的 MPI 程序，使用 mpi4py 库在 Python 中实现。这个程序创建了两个进程，它们会交互并共享一些数据。让我们逐行解释这段代码：

- `import numpy`:这行代码导入了 NumPy 库，一个常用的 Python 数学库。在这个程序中，它被用来创建和操作数组。
- `from mpi4py import MPI`:这行代码导入了 mpi4py 库中的 MPI 模块，这是一个 Python 中使用 MPI 的接口。
- `comm = MPI.COMM_WORLD`:创建了一个 MPI 通信器对象。MPI 通信器是进程间通信的上下文，COMM_WORLD 是所有启动的进程的默认通信器。
- `rank = comm.Get_rank()`:每个进程都有一个唯一的标识符，称为它的 "rank"。这行代码获取并存储了当前进程的 rank。
- `randNum = numpy.zeros(1)`:创建了一个包含一个元素(这个元素的值为0)的 NumPy 数组。
- `if rank == 1:` 这个条件判断语句是说，如果当前进程的 rank 等于 1，那么执行下面的代码块。
    - `randNum = numpy.random.random_sample(1)`:生成一个随机数，并存入 `randNum` 数组。
    - `print("Process", rank, "drew the number", randNum[0])`:打印出当前进程的 rank 和生成的随机数。
    - `comm.Send(randNum, dest=0)`:使用 MPI 的 `Send` 函数将 `randNum` 数组发送到 rank 为 0 的进程。
- `if rank == 0:` 这个条件判断语句是说，如果当前进程的 rank 等于 0，那么执行下面的代码块。
    - `print("Process", rank, "before receiving has the number", randNum[0])`:打印出当前进程的 rank 和接收前的 `randNum` 数组的值。
    - `comm.Recv(randNum, source=1)`:使用 MPI 的 `Recv` 函数从 rank 为 1 的进程接收数据，并将接收到的数据存入 `randNum` 数组。
    - `print("Process", rank, "received the number", randNum[0])`:打印出当前进程的 rank 和接收后的 `randNum` 数组的值。

注意上面的代码，我们需要通过以下命令来运行：

```
!mpiexec --allow-run-as-root --oversubscribe -n 2 python mpi3.py
```

`mpi3.py` **是我们当前的文件，后面不再赘述。**

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

status = MPI.Status()

randNum = numpy.zeros(1)

if rank != 0:
  randNum[0] = numpy.random.random_sample(1)
  print("Process", rank, "drew the number", randNum[0])
  comm.Send(randNum, dest=0)
  comm.Recv(randNum, source=0)
  print("Process", rank, "received the number", randNum[0])

if rank == 0:
  for i in range(comm.Get_size() - 1):
    print("Process", rank, "before receiving has the number",
randNum[0])
    comm.Recv(randNum, source=MPI.ANY_SOURCE, status=status)
    print("Process", rank, "received the number", randNum[0])
    randNum *= 2
    comm.Send(randNum, dest=status.source)
```

这个程序和前一个例子类似，只不过它使用了更多的 MPI 功能，包括动态接收（从任何源接收消息）和状态对象。它也支持任意数量的进程，而不仅仅是两个。下面是这段代码的逐行解释：

- `status = MPI.Status()`：这行代码创建了一个 MPI 状态对象，可以用来查询接收到的消息的源和标签。
- `randNum = numpy.zeros(1)`：这行代码创建了一个包含一个元素（这个元素的值为0）的 NumPy 数组。
- `if rank != 0:` 这个条件判断语句是说，如果当前进程的 rank 不等于 0（即，对于所有的非零 rank 的进程），那么执行下面的代码块。
  - `randNum[0] = numpy.random.random_sample(1)`：生成一个随机数，并存入 `randNum` 数组。
  - `print("Process", rank, "drew the number", randNum[0])`：打印出当前进程的 rank 和生成的随机数。
  - `comm.Send(randNum, dest=0)`：使用 MPI 的 `Send` 函数将 `randNum` 数组发送到 rank 为 0 的进程。
  - `comm.Recv(randNum, source=0)`：使用 MPI 的 `Recv` 函数从 rank 为 0 的进程接收数据，并将接收到的数据存入 `randNum` 数组。

        - `print("Process", rank, "received the number", randNum[0])`:打印出当前进程的 rank 和接收后的 `randNum` 数组的值。
- `if rank == 0:` 这个条件判断语句是说，如果当前进程的 rank 等于 0，那么执行下面的代码块。
    - `for i in range(comm.Get_size() - 1):` 这个循环将会执行 `comm.Get_size() - 1` 次，其中 `comm.Get_size()` 返回的是通信器中进程的总数量。
        - `print("Process", rank, "before receiving has the number", randNum[0])`:打印出当前进程的 rank 和接收前的 `randNum` 数组的值。
        - `comm.Recv(randNum, source=MPI.ANY_SOURCE, status=status)`:使用 MPI 的 `Recv` 函数从任何进程接收数据，并将接收到的数据存入 `randNum` 数组。这里 `source=MPI.ANY_SOURCE` 表示接收消息的来源可以是任何进程。
        - `print("Process", rank, "received the number", randNum[0])`:打印出当前进程的 rank以及接收到的 `randNum` 数组的值。
- `randNum *= 2`:将 `randNum` 数组的值乘以 2。
- `comm.Send(randNum, dest=status.source)`:使用 MPI 的 `Send` 函数将 `randNum` 数组发送回给发送随机数的进程。这里 `dest=status.source` 表示发送的目标进程是接收消息的源进程。

总的来说，这个程序的主要目标是让所有非零 rank 的进程生成一个随机数，然后将这个随机数发送给 rank 为 0 的进程。然后，rank 为 0 的进程接收到这个随机数，将它乘以 2，然后再发送回原来的进程。这个过程会对所有的非零 rank 的进程进行，从而实现了一个简单的并行计算过程。

这里，`comm.Send(randNum, dest=0)` 和 `comm.Recv(randNum, source=0)` 的确会使得 rank 为 0 的进程的 `randNum` 变量被覆盖，但是这正是我们想要的行为。

在 rank 不为 0 的进程中，首先生成一个随机数并保存到 `randNum` 中，然后将 `randNum` 发送给 rank 为 0 的进程。然后，它会从 rank 为 0 的进程接收一个数(这个数是 rank 为 0 的进程接收到的随机数乘以 2 后的结果)，并保存到 `randNum` 中，覆盖原来的随机数。

在 rank 为 0 的进程中，它会从其他进程接收一个随机数并保存到 `randNum` 中，覆盖原来的值(注意，这个值在初次使用之前并没有被赋值，所以我们不用担心有任何重要的数据被覆盖)。然后，它会将这个接收到的数乘以 2，然后再发送回原来的进程。

总的来说，这个程序是通过覆盖 `randNum` 的值来传递数据的

Point-to-Point Communication

需要先创建 default communicator

```
comm = MPI.COMM_WORLD
```

然后调用 comm 并通过下面两个函数来进行【点对点通信】(这是一个进程间通信)

```
comm.Send(randNum, dest=0)
comm.Recv(randNum, source=0)
```

对于 Send() 函数，我们要设置 dest（目标进程）的 rank
对于 Recv() 函数，我们要设置 source(源进程)的 rank

Send() 函数与 Recv() 函数都可以设置 tag 参数，这是可选的，用于筛选消息的标签。默认值是0。你也可以使用 MPI.ANY_TAG 来表示接收任何标签的消息。

```
comm.Send(buf, dest, tag=0)
comm.Recv(buf, source, tag=0, status=None)
```

我们看到 Recv() 函数还有一个 status 参数，这个是可选的。这是一个 MPI.Status 对象，用于保存接收操作的状态信息。这个对象可以提供如源进程的 rank，消息的标签，接收到的数据的数量等信息。如果这个参数被省略，那么这些信息就会被丢弃。

官方文档：
https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Status.html#mpi4py.MPI.Status

这里有个例子：

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

status = MPI.Status()

randNum = numpy.zeros(1)

if rank != 0:
  randNum[0] = numpy.random.random_sample(1)
  print("Process", rank, "drew the number", randNum[0])
```

```
  comm.Send(randNum, dest=0)
  comm.Recv(randNum, source=0)
  print("Process", rank, "received the number", randNum[0])

if rank == 0:
  for i in range(comm.Get_size() - 1):
    print("Process", rank, "before receiving has the number",
randNum[0])
    comm.Recv(randNum, source=MPI.ANY_SOURCE, status=status)
    print("Process", rank, "received the number", randNum[0])
    randNum *= 2
    comm.Send(randNum, dest=status.source)
```

在这个上面的例子中，rank == 0 的进程在循环中重复接收消息，直到它已经从所有其他进程接收到消息。

这里有一行代码很重要，

```
comm.Recv(randNum, source=MPI.ANY_SOURCE, status=status)
```

status 参数的设置在这里非常有用，因为它允许 rank == 0 的进程知道每个接收到的消息来自哪个进程。这是通过 status.source 属性来实现的（见下面），它记录了发送最后一条接收到的消息的进程的排名。然后这个信息被用于把消息发送回原来的进程。

```
comm.Send(randNum, dest=status.source)
```

估计有人会好奇上面的例子是否会导致消息被重复接收？

答案是不会。尽管 rank == 0 的进程被设定为从任何进程接收消息，但 MPI 的 Recv 函数是阻塞的，这意味着每次调用 Recv 会阻塞进程，直到它接收到一个新的消息。一旦消息被接收，它就不会再被接收第二次。

在给定的代码中，rank == 0 的进程会在循环中多次调用 Recv，每次从不同的进程接收一个消息。然后，它使用 status 对象来确定消息的来源，将接收到的数值乘以 2，然后将结果发送回原来的进程。这个过程对每个非零进程都会进行一次，所以每个进程都会发送和接收一次消息，没有消息会被接收两次。

所以，虽然 MPI.ANY_SOURCE 使得 rank == 0 的进程可以从任何进程接收消息，但这并不会导致消息被重复接收。