并行与分布式计算基础：第五讲

杨超
chao_yang@pku.edu.cn

2019 秋

内容提纲

上次课程回顾

课程基本情况

- 课程名称：并行与分布式计算基础
- 授课教师：杨超（chao_yang@pku.edu.cn，理科 1 号楼 1520）
- 课程助教：尹鹏飞（pengfeiyin@pku.edu.cn）

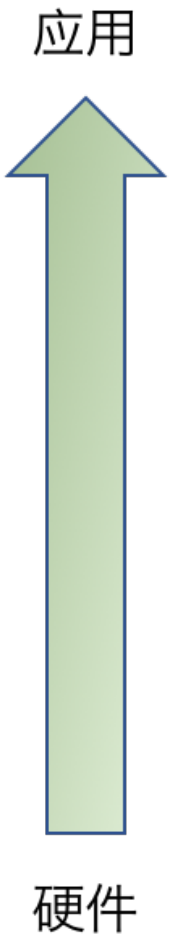
授课内容（暂定）

- 引言
- 硬件体系架构
- 并行计算模型
- 编程与开发环境
- MPI 编程与实践
- OpenMP 编程与实践
- GPU 编程与实践
- 前沿问题选讲

上课时间（地点：二教 211）

上课时间	星期一	星期二	星期三	星期四	星期五
第 1 节 (8:00-8:50)					
第 2 节 (9:00-9:50)					
第 3 节 (10:10-11:00)				单周	
第 4 节 (11:10-12:00)				单周	
第 5 节 (13:00-13:50)		每周			
第 6 节 (14:00-14:50)		每周			
第 7 节 (15:10-16:00)					
第 8 节 (16:10-17:00)					
第 9 节 (17:10-18:00)					
第 10 节 (18:40-19:30)					
第 11 节 (19:40-20:30)					
第 12 节 (20:40-21:30)					

并行计算研究的几个主要视角



- 应用视角：数学建模，算法设计与分析等。
- 算法视角：算法的并行化以及相关的主要原则等。
- 编程视角：采用何种方式编程实现并行算法。
- 性能视角：通过建立并行计算模型，指导算法设计、编程实现、性能优化等。
- 硬件视角：并行计算机体系架构。

课程从应用与硬件切入，通过讨论算法与性能，最终聚焦并行编程。

并行计算三大定律 (1)

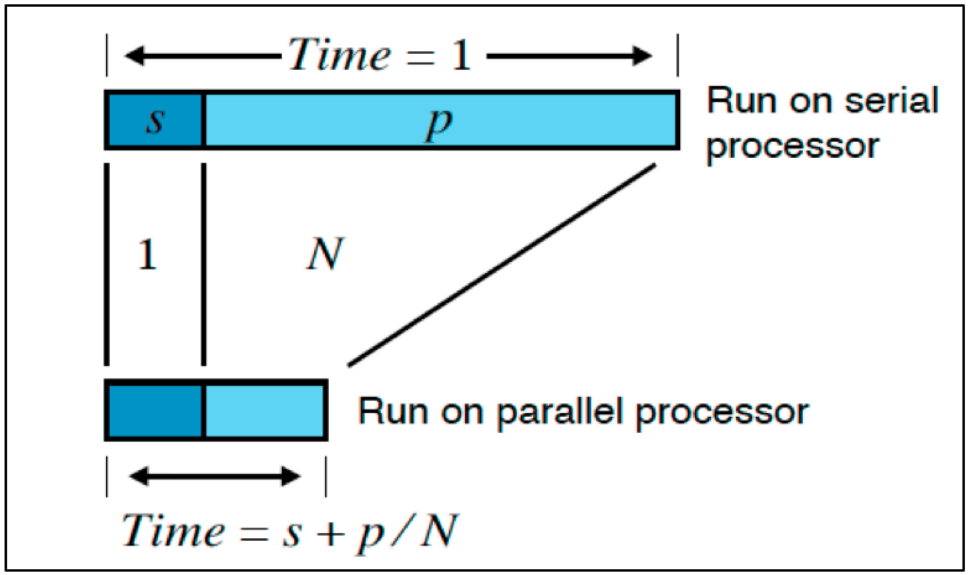
阿姆达尔定律 (Amdahl's Law, 1967)

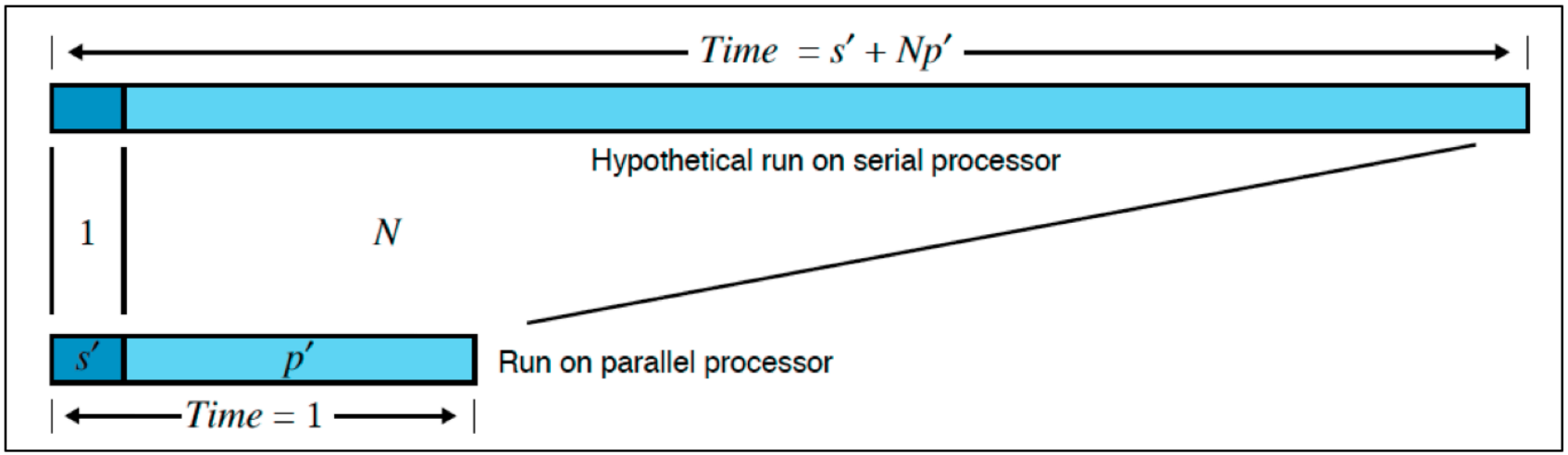
记 $\alpha \in [0,1]$ 是某任务无法并行处理部分所占的比例. 假设该任务的工作量固定，则对任意 n 个处理器，相比于 1 个处理器，能够取得的加速比满足：$S(n) < 1/\alpha$.

古斯塔法森定律 (Gustafson's Law, 1988)

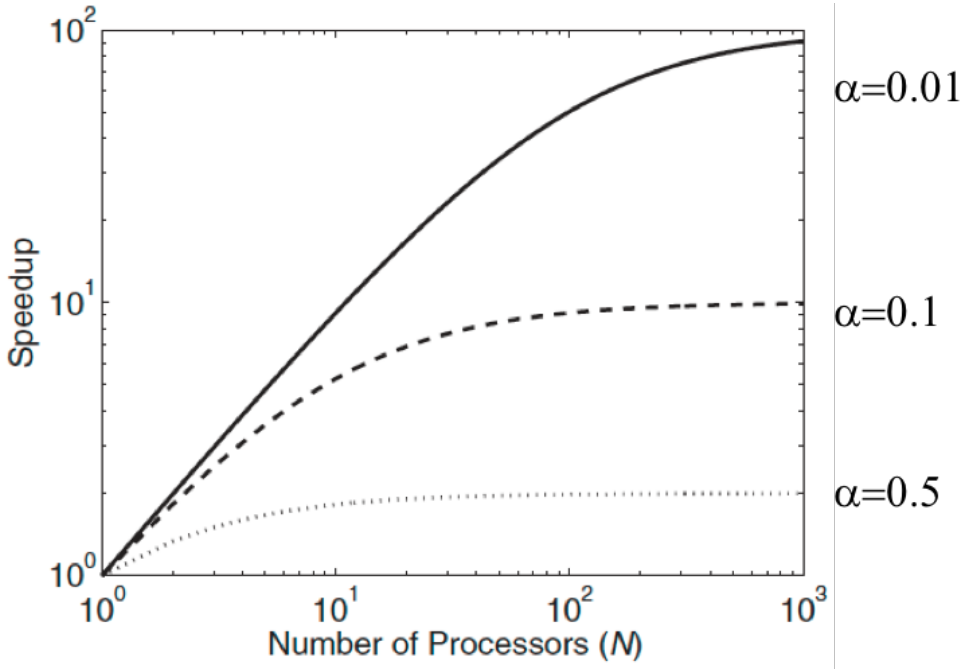
记 $\alpha \in [0,1]$ 是某任务无法并行处理部分所占的比例. 假设该任务的工作量可以随着处理器个数缩放，从而保持处理时间固定. 则对任意 n 个处理器，相比于 1 个处理器，能够取得的加速比 $S'(n)$ 不存在上界.

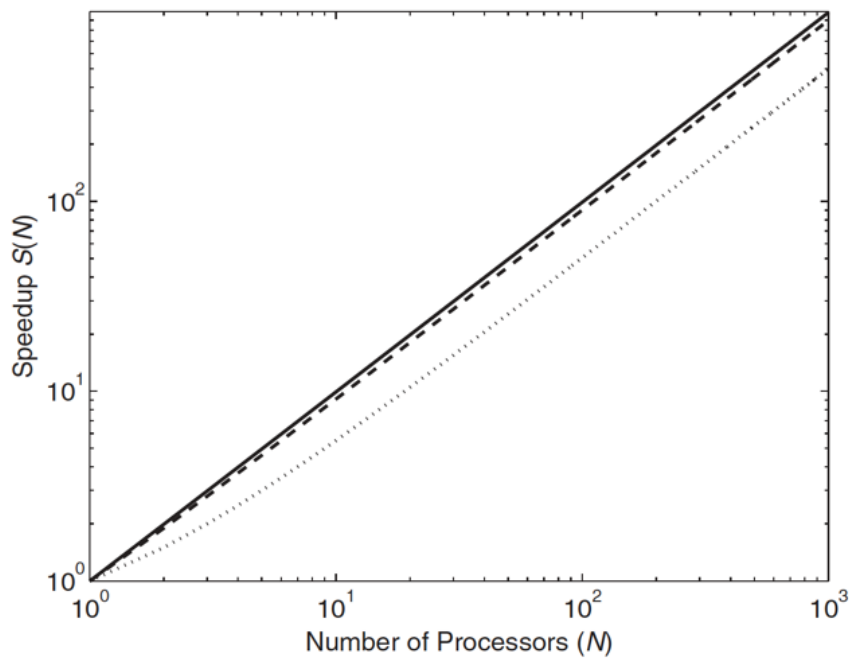
从阿姆达尔加速比到古斯塔法森加速比





从阿姆达尔加速比到古斯塔法森加速比





并行计算三大定律 (2)

孙-倪定律 (Sun-Ni's Law, 1990)

$$S^*(n) = \frac{\alpha + (1-\alpha)G(n)}{\alpha + (1-\alpha)\frac{G(n)}{n}} \begin{cases} = \dfrac{1}{\alpha + \frac{1-\alpha}{n}} & \text{if } G(n) = 1, \\ = \alpha + (1-\alpha)n & \text{if } G(n) = n, \\ > \alpha + (1-\alpha)n & \text{if } G(n) > n. \end{cases}$$

	加速比 $(n \to \infty)$	并行效率 $(n \to \infty)$
阿姆达尔	$S(n) \to \frac{1}{\alpha}$	$E(n) \to 0$
古斯塔法森	$S'(n) \to \infty$	$E'(n) \to 1-\alpha$
孙-倪 $(G(n) > O(n))$	$S^*(n) \to \infty$	$E^*(n) \to 1$

Roofline 模型 (2009)

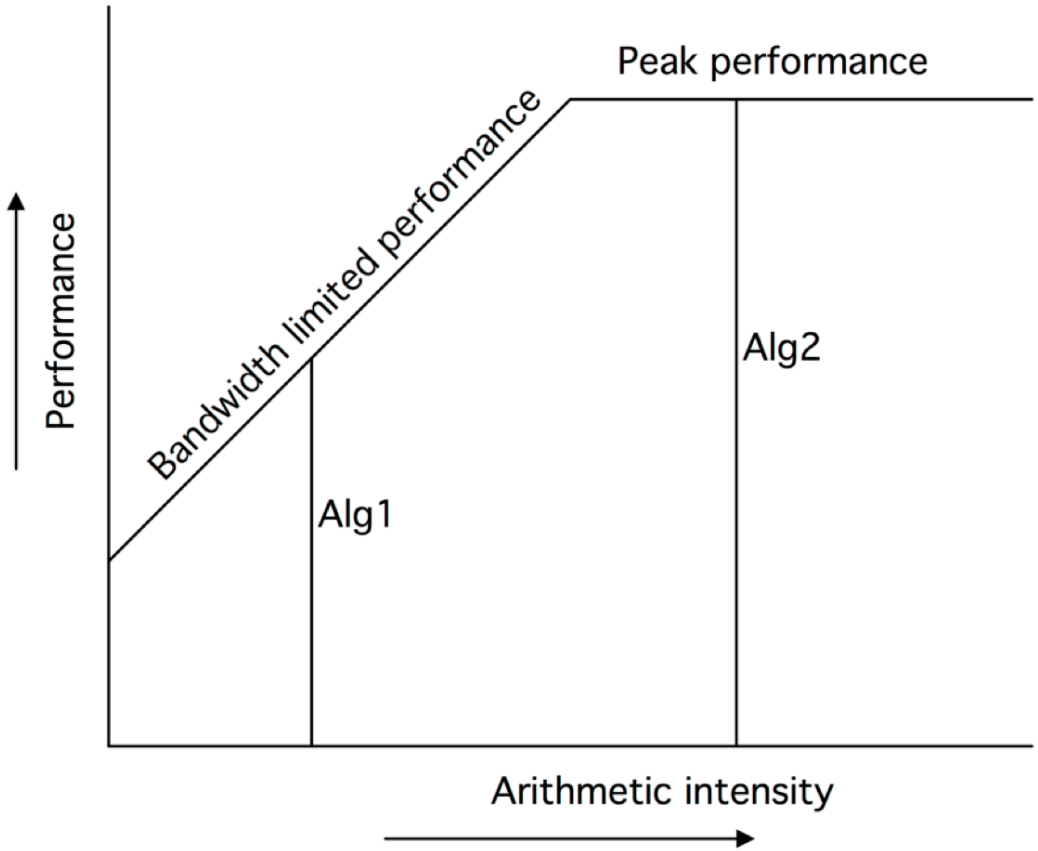
DOI:10.1145/1498765.1498785

The Roofline model offers insight on how to improve the performance of software and hardware.

BY SAMUEL WILLIAMS, ANDREW WATERMAN, AND DAVID PATTERSON

Roofline: An Insightful Visual Performance Model for Multicore Architectures

APRIL 2009 | VOL. 52 | NO. 4 | COMMUNICATIONS OF THE ACM 65



AMAT 模型

单层 AMAT (Average Memory Access Time) 模型

假设只有一级缓存，AMAT 模型预测的平均访存时间为：

$$\text{AMAT} = (1-r)T_\$ + r(T_\$ + T_M) = T_\$ + rT_M,$$

其中 $T_\$$ 为缓存访问时间 (也叫命中时间, hit time)，T_M 为内存访问时间 (也叫缓存失效损失, miss penalty)，r 为缓存失效率 (miss rate)。

两层 AMAT 模型

假设有两级缓存，T_1, T_2 分别为 L1, L2 缓存访问时间，T_M 为内存访问时间，r_1, r_2 分别为 L1, L2 缓存的局部失效率 (local miss rate)，则两层 AMAT 模型预测的平均访存时间为：

$$\text{AMAT}_2 = T_1 + r_1(T_2 + r_2T_M) = T_1 + R_1T_2 + R_2T_M.$$

其中 $R_1 = r_1$, $R_2 = r_1r_2$, 分别为 L1, L2 缓存的整体失效率 (global miss rate)。

共享内存并行计算模型 (2)

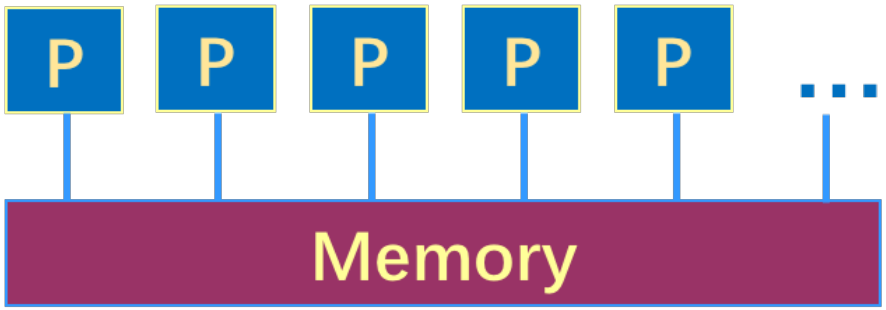
从多 (众) 核并行角度考虑

PRAM 模型 (Parallel Random Access Machine Model, 1978)

是 RAM (Random Access Machine) 模型在共享内存系统上的扩展。

- 所有处理器共享一个连续的内存空间;
- 每个处理器执行相互独立的指令;
- 处理器执行任意一种计算或访存操作的时间开销都相等。

模型参数：处理器个数 p，单位执行时间 τ.



PRAM 模型的并行访存策略

基于不同的处理访存冲突的策略，有四类 PRAM 模型

- Exclusive-read, exclusive-write (EREW) 模型；
- Concurrent-read, exclusive-write (CREW) 模型；
- Exclusive-read, concurrent-write (ERCW) 模型；
- Concurrent-read, concurrent-write (CRCW) 模型。

其中，concurrent-write 的处理又分为

- Common：所有处理器写的数值完全相同，没有冲突；
- Arbitrary：任意一个处理器完成写操作，其他处理器不操作；
- Priority：按照某种实现约定的原则确定处理器的优先级，优先级高的处理器写；
- Reduction：规约操作，如 SUM, MAX 等。

PRAM 模型的特点分析

- 正如串行算法设计者使用 RAM 来研究串行算法性能（比如，串行时间复杂度）一样，并行算法设计者使用 PRAM 来建模和量化分析并行算法性能（比如，在给定个数处理器上的时间复杂度）。
- 采用 PRAM 模型对并行算法进行理论分析的课程：
  `http://pages.cs.wisc.edu/~tvrdik/cs838.html`
- PRAM 模型可以用于共享内存并行计算机，也可以用于分布式并行计算机。
- 然而，PRAM 忽略计算机体系架构的诸多重要特性，比如访存、通信与计算开销的巨大差别，比如缓存、同步等机制，仅使用两个参数（单位时间 τ、处理器个数 p）来估计算法成本，往往难以预测真实性能。
- 更精细的模型：PHM (Parallel Hierarchical Memory) 模型等。

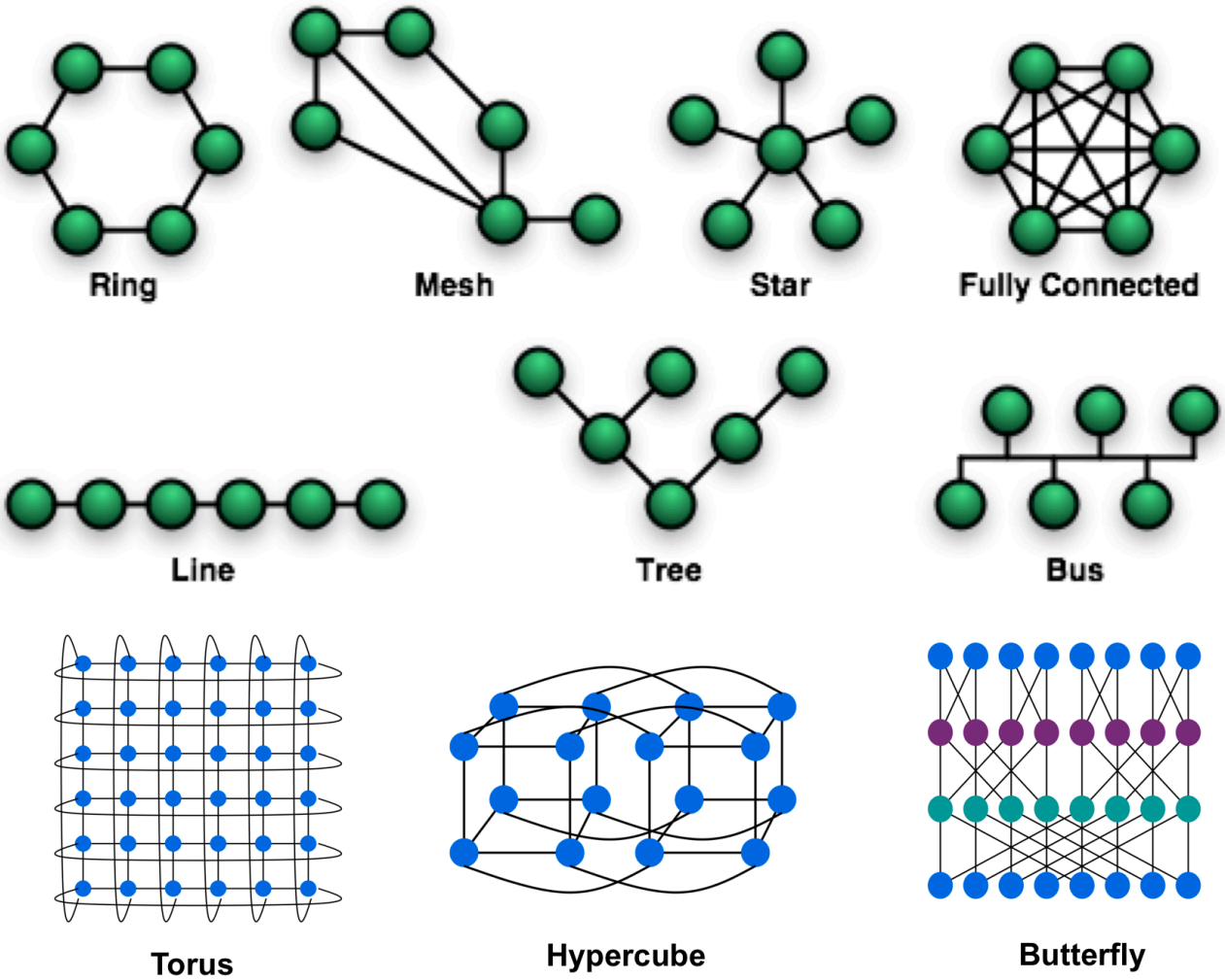
分布式并行计算模型

网络系统相关的一些基本概念

- 计算节点 (compute node)：使用高速互联网络连接的计算机，每个计算节点大多是共享内存架构。
- 网络连线 (link)：计算节点之间的高速互联网络，有连线表示存在直接互联。
- 路由 (router)：决定网络通信策略的算法或设备。
- 延迟 (latency)：从一个计算节点到另一个计算节点的数据传输时间。
- 带宽 (bandwidth)：单位时间的网络传输量，单位 GB/s。

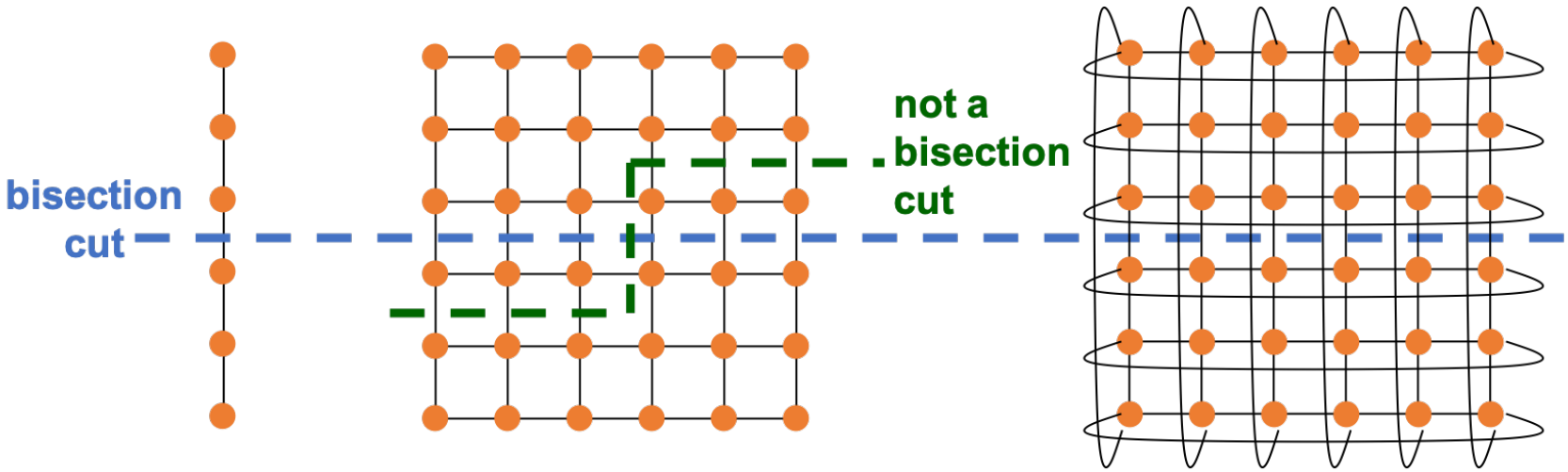
互联网络的拓扑架构

线/总线、环/环面、网、星、树、全连接、超立方、蝴蝶、蜻蜓等。



如何衡量不同网络拓扑的性能?

- 跳 (hop)：拓扑网络上一点到另一点的最短距离。
- 网络直径 (diameter)：拓扑网络上任意两个节点间的最大跳数。
- 二分宽度 (bisection width)：将拓扑网络平分为二的最小切割数。



思考：一个最优化问题

给定节点数 n，选取合适的总连线数，最小化直径、最大化二分宽度!

α-β 模型

网络通信时间由延迟 α，带宽 $1/\beta$，和消息长度 L 决定（忽略拓扑架构）：

$$T_{\text{comm}} = \alpha + \beta L.$$

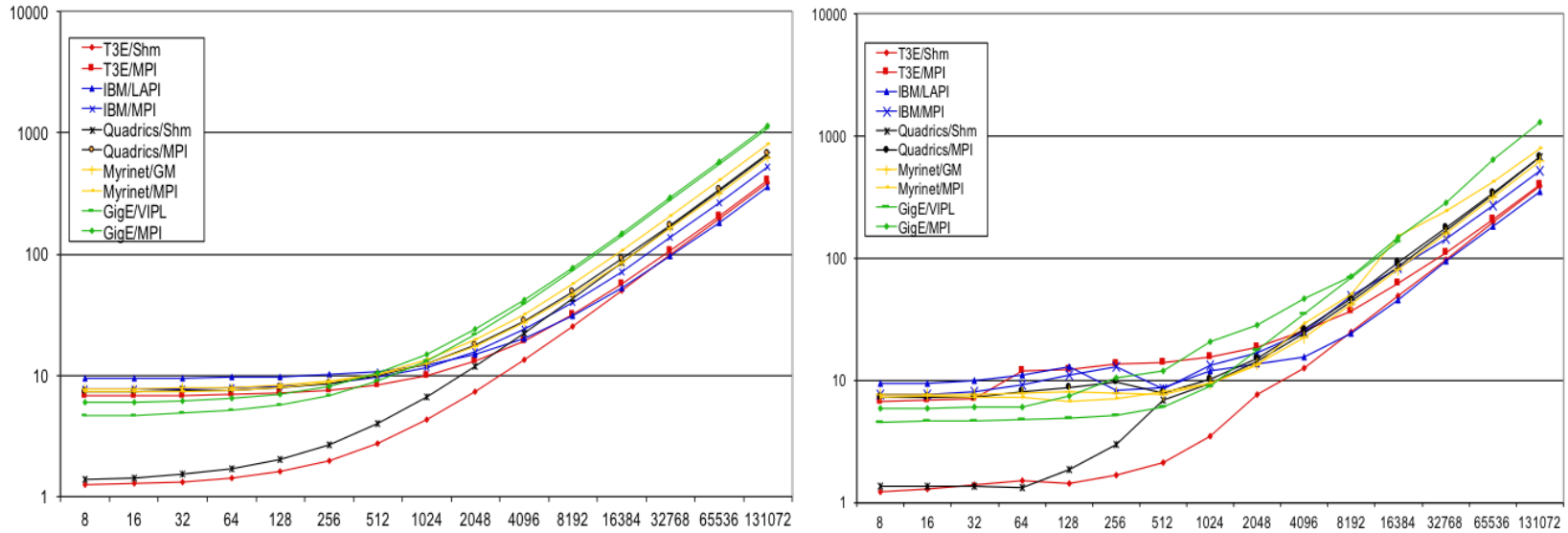
machine	α	β
T3E/Shm	1.2	0.003
T3E/MPI	6.7	0.003
IBM/LAPI	9.4	0.003
IBM/MPI	7.6	0.004
Quadrics/Get	3.267	0.00498
Quadrics/Shm	1.3	0.005
Quadrics/MPI	7.3	0.005

α: latency (us)
β: 1/BW (us/Byte)

- 推论：延迟、带宽分别是影响短消息和长消息通信性能的主要因素，多个短消息不如一个长消息，因为 $n(\alpha + \beta L) >> \alpha + \beta nL$。

α-β 模型的准确度

- 左：预测时间，右：实测时间。

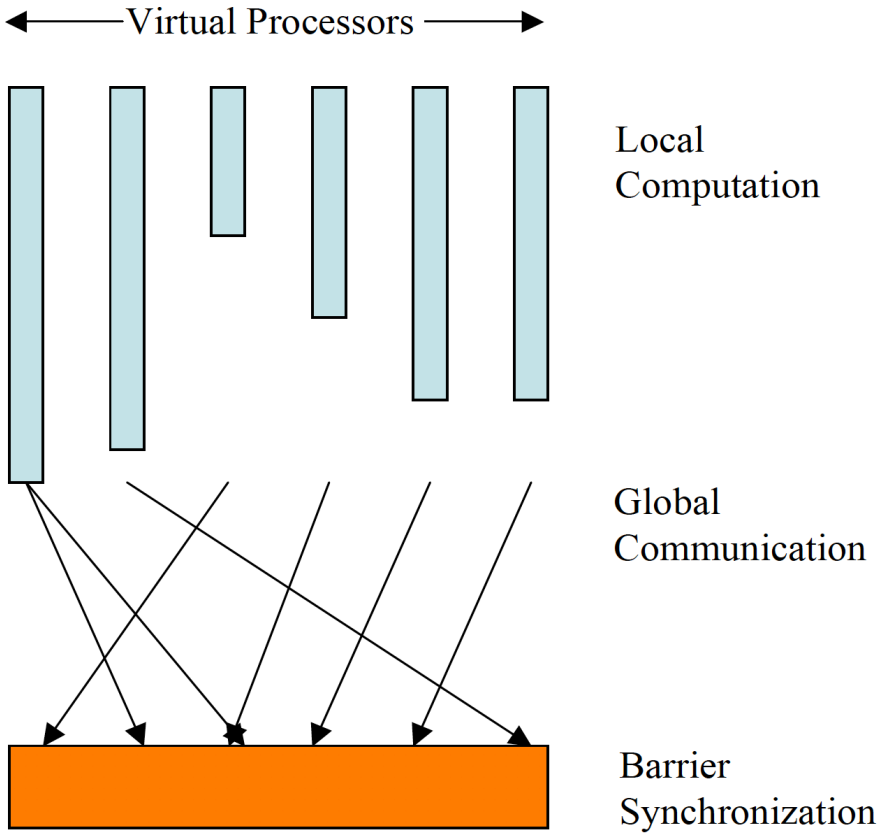


BSP (Bulk Synchronous Parallel) 模型

基本假设/约定

- 每个处理器拥有一个独立的内存空间；
- 所有处理器可以通过一个公共网络采用点对点方式通信；
- 所有处理器可以通过该网络实现同步；
- 程序以超步 (superstep) 为单位并行执行；
- 每个超步末进行栅栏同步，从而保证所有处理器同时进行下一个超步。

每个超步执行本地计算、全局通信两种操作：



BSP 模型的量化评估

BSP 模型参数

- p：处理器个数；
- S：总超步数；
- g：每单位消息的单边通信时间（$1/g$= 通信带宽，由硬件决定）；
- ℓ：每次栅栏同步的时间（由硬件决定）；
- w_s：第 s 超步本地计算的最大时间；
- h_s：第 s 超步单边通信的最大消息量。

采用 BSP 预测程序执行总时间：

$$\text{Time}_{BSP} = \sum_{s=1}^{S} w_s + g \sum_{s=1}^{S} h_s + \ell S.$$

BSP 模型的评价

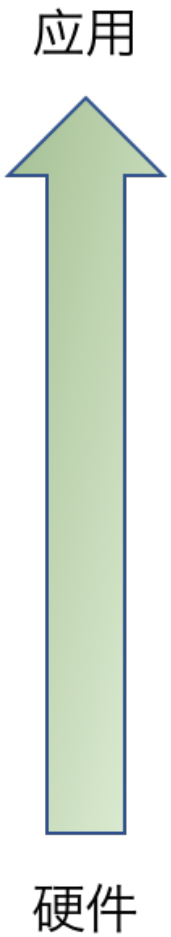
- 在一些情况下，本地计算可以与全局通信重叠，甚至进一步与栅栏同步重叠，此时

$$\text{Time}_{BSP} = \sum_{s=1}^{S} \max\{w_s, gh_s\} + \ell S, \quad \text{Time}_{BSP} = \sum_{s=1}^{S} \max\{w_s, gh_s, \ell\}.$$

- 基于 BSP 模型的算法受计算机硬件体系架构的制约较小，容易编程实现，性能也相对容易预测。
- BSP 模型忽略了通信的延迟，因此传输 m 个长度为 1 的消息的开销等于传输 1 个长度为 m 的消息（事实上应该是大于）。
- 更精细的模型：LogP 模型 (Latency/overhead/gap/Proc) 等。

MPI 编程 (1)：入门

并行计算研究的几个主要视角



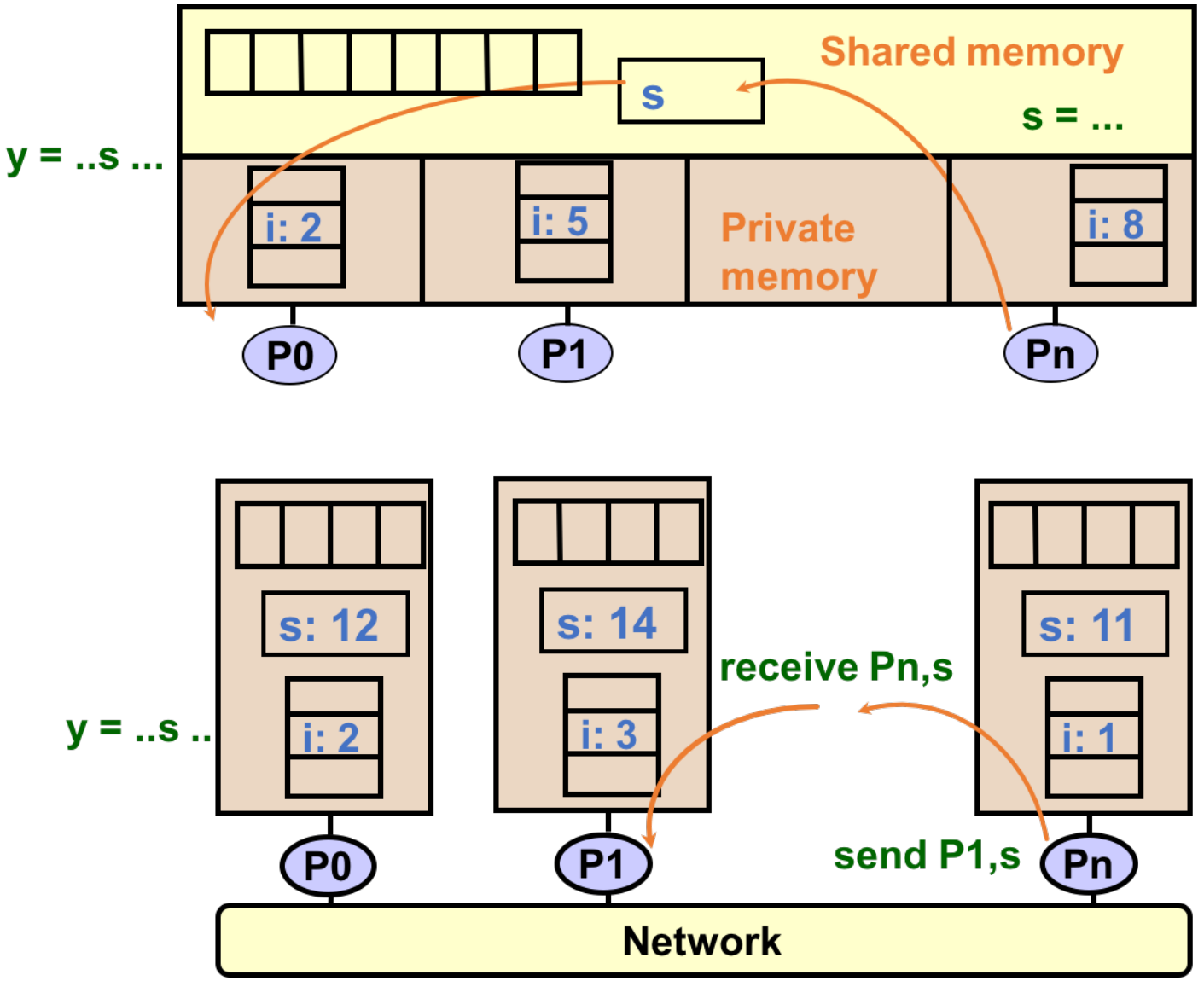
- 应用视角：数学建模，算法设计与分析等。
- 算法视角：算法的并行化以及相关的主要原则等。
- 编程视角：采用何种方式编程实现并行算法。
- 性能视角：通过建立并行计算模型，指导算法设计、编程实现、性能优化等。
- 硬件视角：并行计算机体系架构。

课程从应用与硬件切入，通过讨论算法与性能，最终聚焦并行编程。

并行编程模型

- 自动并行
  - 提交串行程序，由编译器自动实现并行；
  - 希望十分渺茫。
- 共享内存并行编程
  - 适用于共享内存并行机，以线程为并行单位；
  - pthreads、OpenMP、CILK、TBB等。
- 消息传递
  - 适用于共享内存和分布式并行机；
  - PVM、MPI等。
- 数据并行
  - CUDA/OpenCL、 Fortran 90、 PGAS、MapReduce等。
- 混合并行
  - MPI + OpenMP、MPI + CUDA 等。

共享内存与消息传递



共享内存并行编程

消息传递并行编程

什么是 MPI?

MPI = Message Passing Interface

是一组由学术界和工业界联合发展的、面向主流并行计算机的、标准化和可移植的消息传递接口标准。

- 定义了若干核心库函数的语法和涵义;
- 独立于编程语言，支持 C/C++、Fortran 语言的绑定;
- 独立于平台，学术界和厂商发展了若干高效、可靠的实现版;
- 支撑和推动了高性能计算软硬件生态的发展。

为什么用 MPI?

- 目前几乎所有主流并行计算机上都提供了 MPI 的支持;
- MPI 没有依托于任何标准化组织，但已经成为事实上的工业标准;
- 除了 MPI，没有其他更好的选择!

MPI 的发展历史

- 1991 年夏季：首次讨论 MPI 的概念；
- 1992 年 4 月：首届 MPI 研讨会，MPI 标准工作组成立；
- 1992 年 11 月：MPI 1.0 初稿形成，MPI 论坛成立；
- 1993 年 1-9 月：MPI 标准工作组会议每六周开展一次；
- 1993 年 11 月：MPI 1.0 初稿在 SC93 大会上公布，征集公众意见；
- 1994 年 6 月：MPI 1.0 发布。

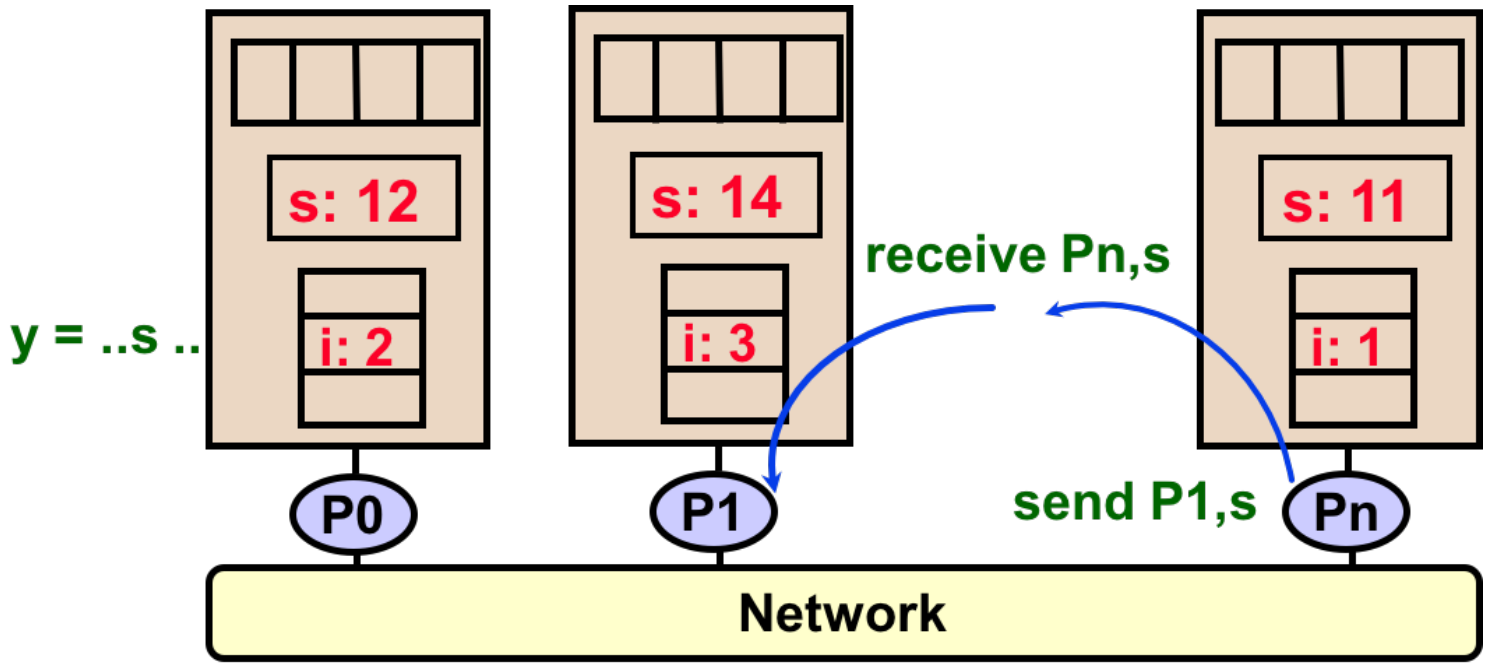
版本	最新版	增加主要功能	增加语言绑定
MPI-1	MPI 1.3	消息传递、静态运行空间	C/F77
MPI-2	MPI 2.2	并行 IO、动态进程等	C++/F90
MPI-3	MPI 3.1	容错、RMA 等 (主流版本)	F2008
MPI-4	MPI 4.0	混合编程等 (进行中)	-

MPI 的主要实现版本

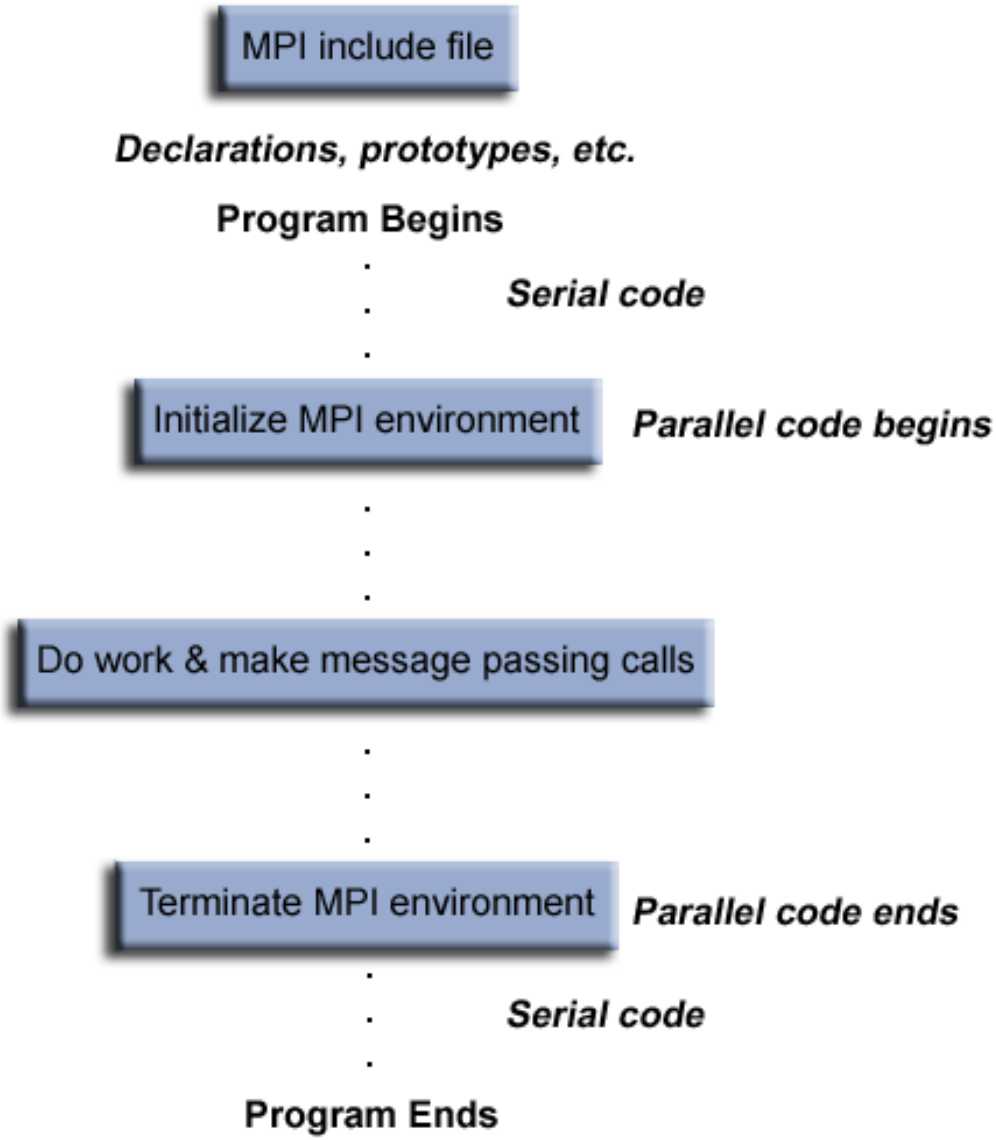
- MPICH
  - 由 Argonne 国家实验室与 Mississippi 州立大学联合开发；
  - 是最早的，也是目前最流行的 MPI 实现。
- MVAPICH
  - Ohio 州立大学开发；
  - 基于 MPICH 发展，强调对各类硬件和网络的个性化支持。
- OpenMPI
  - Stuttgart 大学等开发；
  - 是多个 MPI 开源实现的合并版。
- 商业版
  - Intel MPI、Cray MPI、HP-MPI、MS-MPI、.....
- ABI (application binary interface) 兼容：由 MPICH 发起，保证各个 MPI 实现在底层数据类型上的兼容性。

MPI 的主要理念

- 机器由若干可以相互传递消息的进程 (process) 构成；
- 每个进程拥有私有的存储空间，进程间无共享存储；
- 进程间的消息传递采用显式的发送/接受 (send/receive) 机制完成；
- 程序的运行模式主要有松散同步和完全异步两种；
- 程序往往采用 SPMD (single program multiple data) 方式编写。



MPI 程序的执行流程


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

MPI 程序的结构

- 头文件 (header file)：包含了 MPI 库提供的函数接口定义

C include file	Fortran include file
`#include "mpi.h"`	`include 'mpif.h'`

- 函数接口的调用格式
  - MPI 中函数、变量、参数均以 `MPI_` 作为前缀

C Binding	
Format:	`rc = MPI_Xxxxx(parameter, ... )`
Example:	`rc = MPI_Bsend(&buf,count,type,dest,tag,comm)`
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	`CALL MPI_XXXXX(parameter,..., ierr)` `call mpi_xxxxx(parameter,..., ierr)`
Example:	`CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)`
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful

MPI 定义的函数

- MPI-3 有超过 430 个函数 (包含 MPI-1、MPI-2 大部分函数)；
- 从应用角度 (包括功能和性能) 上看，只有 20 多个函数是常用的；
- 单纯从功能上说，只需要 6 个函数就可以完成所有 MPI 功能。

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

MPI 库的启动和结束

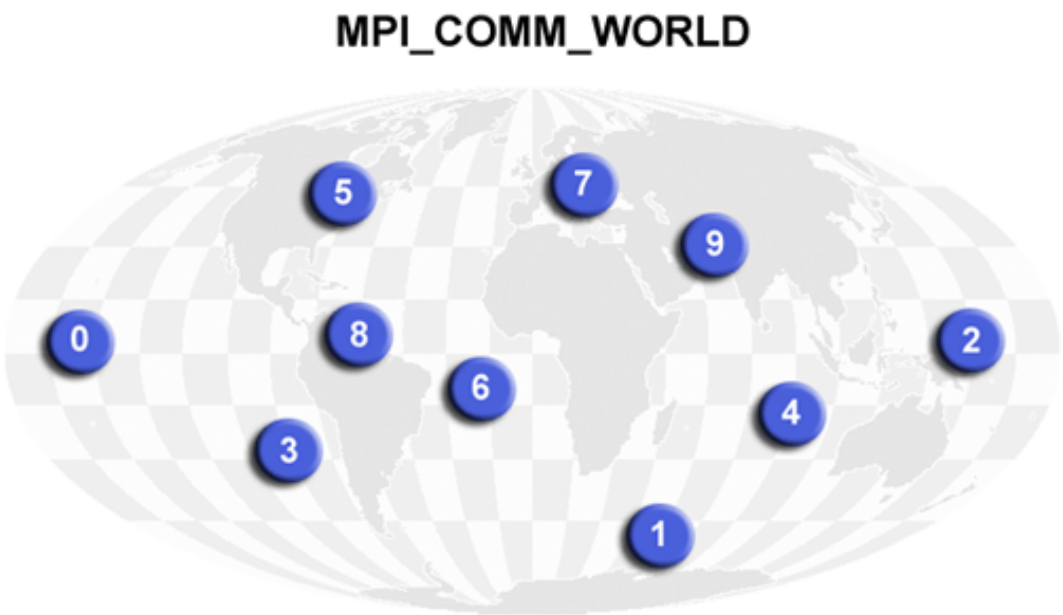
- MPI_Init 一般在主程序的开头，主要用于启动 MPI 环境；
- MPI_Finalize 一般在主程序末尾，主要用于终止 MPI 环境；
- 这两个函数的语法如下：

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- 如果运行正常，所有 MPI 函数的返回值都是 MPI_SUCCESS 常量。

我在哪?——通信器 (communicator)

- 定义了所有参与通信的进程的集合;
- 几乎所有的 MPI 函数都需要指定该函数所作用的通信器;
- 通信器变量的数据类型是 `MPI_Comm`;
- 默认通信器是 `MPI_COMM_WORLD` (所有进程)。



我是谁?——获取进程信息

- MPI_Comm_size 函数用来获得目标通信器的总进程数;
- MPI_Comm_rank 函数用来获得当前进程在目标通信器的进程号;
- 这两个函数的语法如下:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- rank 的范围: $0, 1, \cdots, \texttt{size} - 1$.
- 同一个进程可以属于不同的通信器，因此 rank 也可能不同!

第一个 MPI 程序：hello world!

mpi_hello.c

```c
#include "mpi.h" // mpi header file
#include <stdio.h> // standard I/O

int main(int argc, char *argv[]){
  int size, rank;

  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size); // get num of procs
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get my rank
  printf("From process %d of %d: Hello World!\n", rank, size);
  if (rank == 0) printf("That's all, folks!\n");
  MPI_Finalize(); // done with MPI
  return 0;
}
```

程序的编译与运行

- 加载 MPI 库环境：

```
$ module add mpich
```

- 编译：

```
$ mpicc mpi_hello.c -o hello
```

- 运行 (请正确使用 sbatch 或者 salloc)：

```
$ mpiexec -n 4 ./hello
```

运行结果

- 第一次运行:

```
From process 3 of 4: Hello World!
From process 1 of 4: Hello World!
From process 2 of 4: Hello World!
From process 0 of 4: Hello World!
That's all, folks!
```

- 第二次运行:

```
From process 2 of 4: Hello World!
From process 1 of 4: Hello World!
From process 0 of 4: Hello World!
That's all, folks!
From process 3 of 4: Hello World!
```

- 为什么?