第三章
并行程序设计基础

哈尔滨工业大学
郝萌

2023, Fall Semester

目录

- PCAM设计原理
- 并行程序性能评价
- 并行程序编程模型

并行算法的一般设计方法

- 第一类：串行算法的直接并行化
- 第二类：从问题描述开始设计并行算法
- 第三类：借用已有算法求解新问题

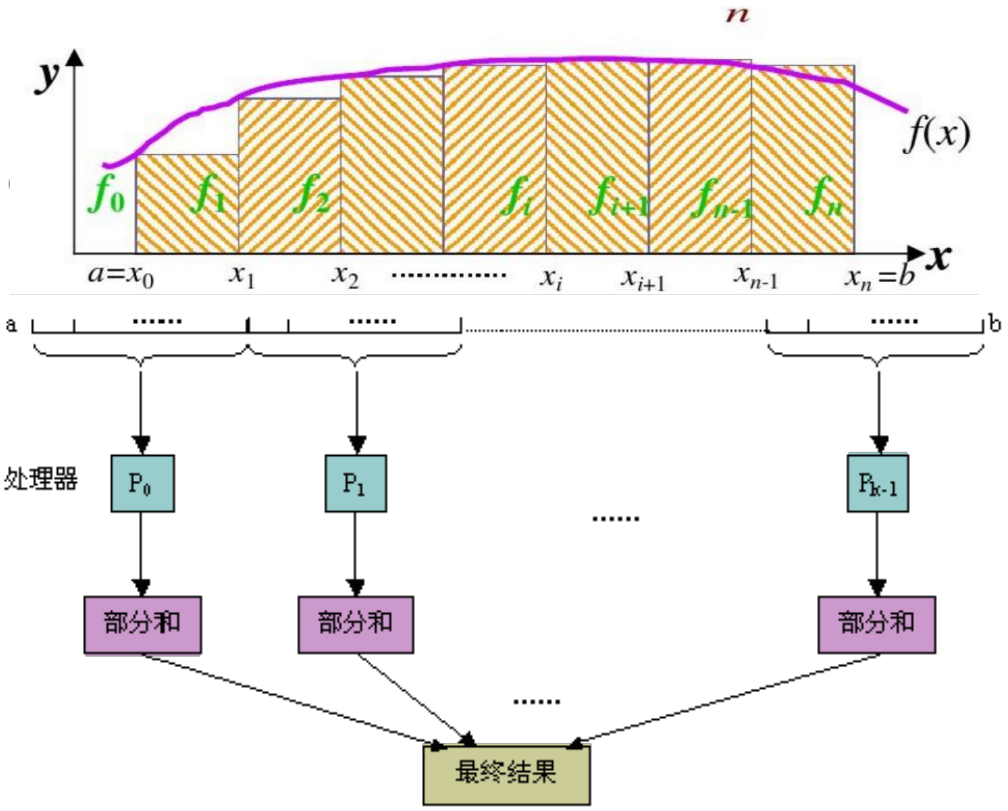
第一类：串行算法的直接并行化

- 方法描述
  - 发掘和利用现有串行算法中的并行性，直接将串行算法改造为并行算法
  - 许多并行编程语言都支持通过在原有的串行程序中加入并行原语（例如某些通信命令等）的方法将串行程序并行化
- 由串行算法直接并行化的方法是并行算法设计的最常用方法之一
- 不是所有的串行算法都可以直接并行化的（模拟退火算法）
- 一个好的串行算法并不能并行化为好的并行算法
- 许多数值串行算法可以并行化为有效的并行算法

第一类：串行算法的直接并行化

- 示例：用求和的方法进行数值积分；设被积函数为f(x)，积分区间为[a,b]
- 串行算法，可以用循环和叠加完成上述求和
- 可以直接进行并行化

科学和工程中有大量数值计算问题。针对这些问题，人们已经设计出了许多串行数值计算方法。在设计这些问题的并行算法时，大多采用串行算法直接并行化的方法。这样做的一个显著优点是，算法的稳定性、收敛性等问题在串行算法中已有结论，不必再考虑。

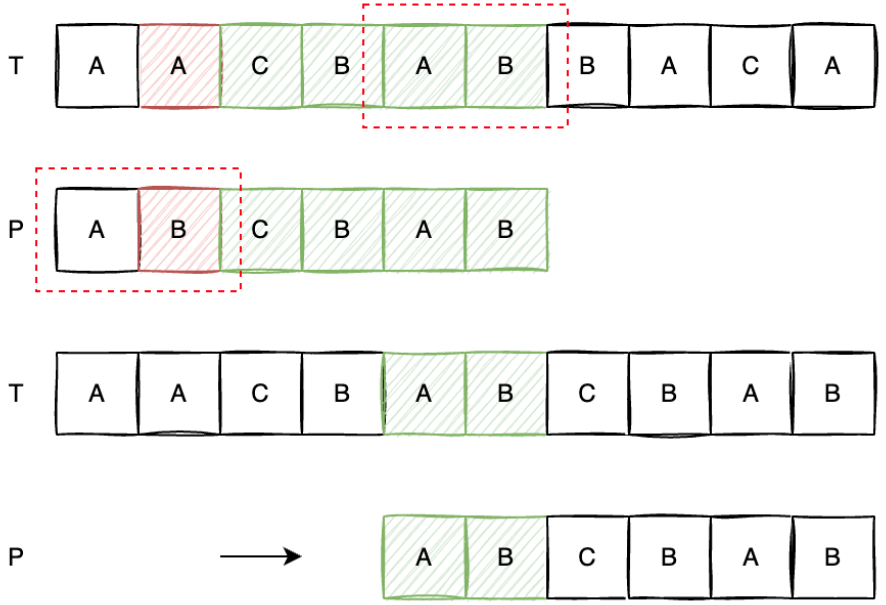


第二类：从问题描述开始设计并行算法

- 方法描述
  - 从问题本身描述出发，不考虑相应的串行算法，设计一个全新的并行算法
- 挖掘问题的固有特性与并行的关系
- 设计全新的并行算法是挑战性和创造性的工作

第二类：从问题描述开始设计并行算法

- **示例：串中包含的字符的个数称为串的长度。给定长度为n的正文串T和长度为m的模式串P，找出P在T中所有出现的位置称为串匹配问题**
- **目前已知的有效的串匹配算法均不易直接并行化，需要重新设计并行算法**

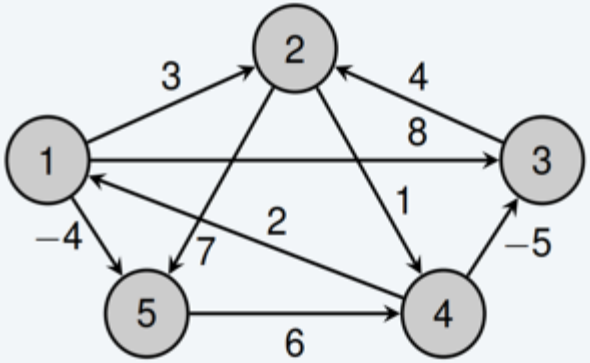


第三类：借用已有算法求解新问题

- 方法描述
  - 找出求解问题和某个已解决问题之间的联系
  - 改造或利用已知算法应用到求解问题上
- 这是一项创造性的工作
- 需要很高的技巧，同时算法设计者要有敏锐的观察力并且在并行算法方面有丰富的经验

第三类：借用已有算法求解新问题

- 示例：使用矩阵乘法算法求解所有点对间最短路径是一个很好的范例
- 问题描述：设在一有向图中，各弧都赋予了非负整数权。图中一条路径的长度定义为该路径上所有的弧的权的和。图中两结点之间的最短路径是指它们之间长度最短的路径



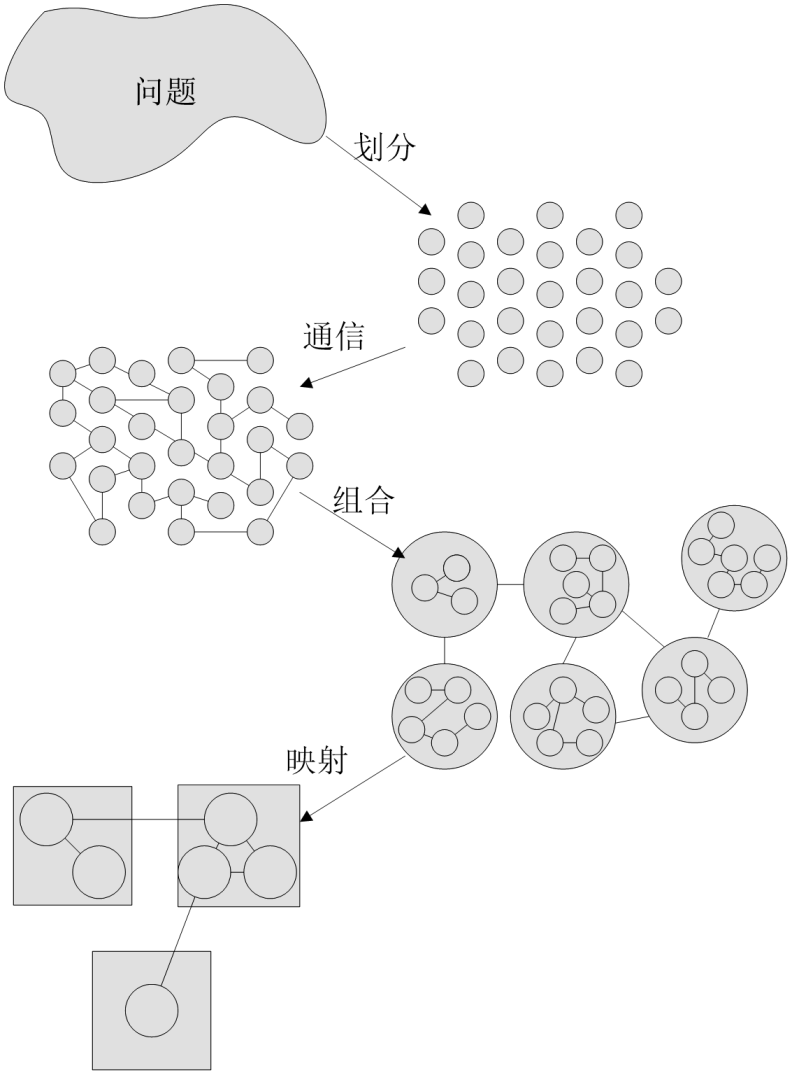
$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

并行算法的设计过程

- **目的：设计出一个具有并发性、可扩展性、局部性和模块性的并行算法**
- **PCAM设计方法**
  - **任 务 划 分 (Partitioning)**
  - **通 信 分 析 (Communication)**
  - **任 务 组 合 (Agglomeration)**
  - **处理器映射 (Mapping)**

PCAM设计方法

- **划分：将整个计算任务分解成一些小任务，其目的是尽量开拓并行执行的可能性**
- **通信：确定小任务执行中需要进行的通信，为组合做准备**
- **组合：按性能要求和实现的代价来考察前两阶段的结果，适当地将一些小任务组合成更大的任务以提高性能、减少通信开销**
- **映射：将组合后的任务分配到处理器上，其目标是使全局执行时间和通信开销尽量小，使处理器的利用率尽量高**



划分方法描述

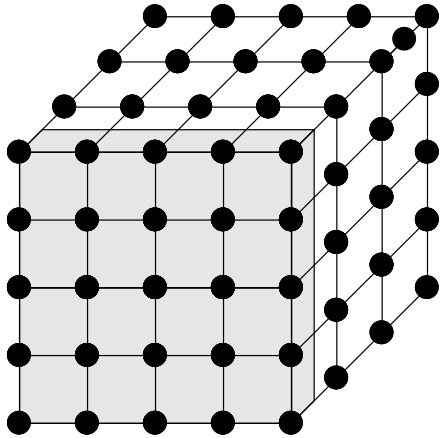
- 充分开拓算法的并发性和可扩展性
- 先进行数据分解（称域分解），再进行计算功能的分解（称功能分解）
- 划分阶段忽略处理器数目和目标机器的体系结构
- 能分为两类划分
  - 域分解（domain decomposition）
  - 功能分解（functional decomposition）

划分：域分解

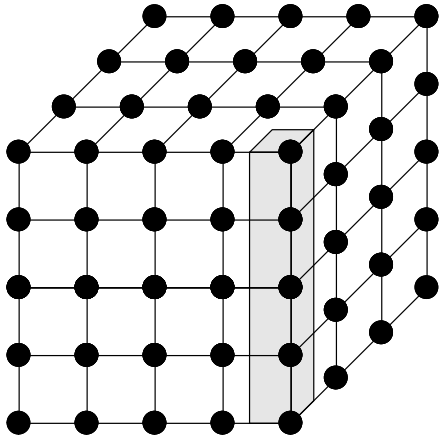
- 域分解也叫**数据划分**，划分的对象是数据。这些数据可以是算法（或程序）的输入数据、计算的中间结果或计算的输出数据
- 域分解的步骤
  - 分解与问题相关的数据，如果可能的话，应使每份数据的数据量大体相等
  - 再将每个计算关联到它所操作的数据上
  - 由此就产生出一些任务，每个任务包括一些数据及其上的操作
  - 当一个操作需要别的任务中的数据时，就会产生通信要求

划分：域分解

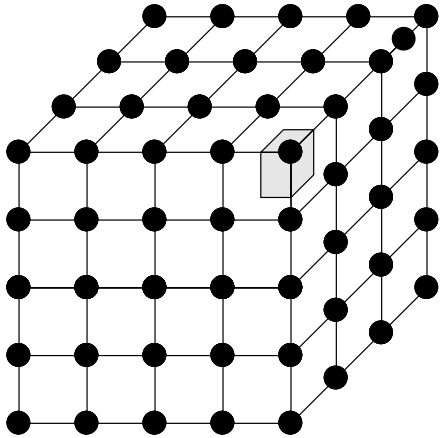
- **域分解示例：三维网格的域分解，各格点上计算都是重复的**



1-D



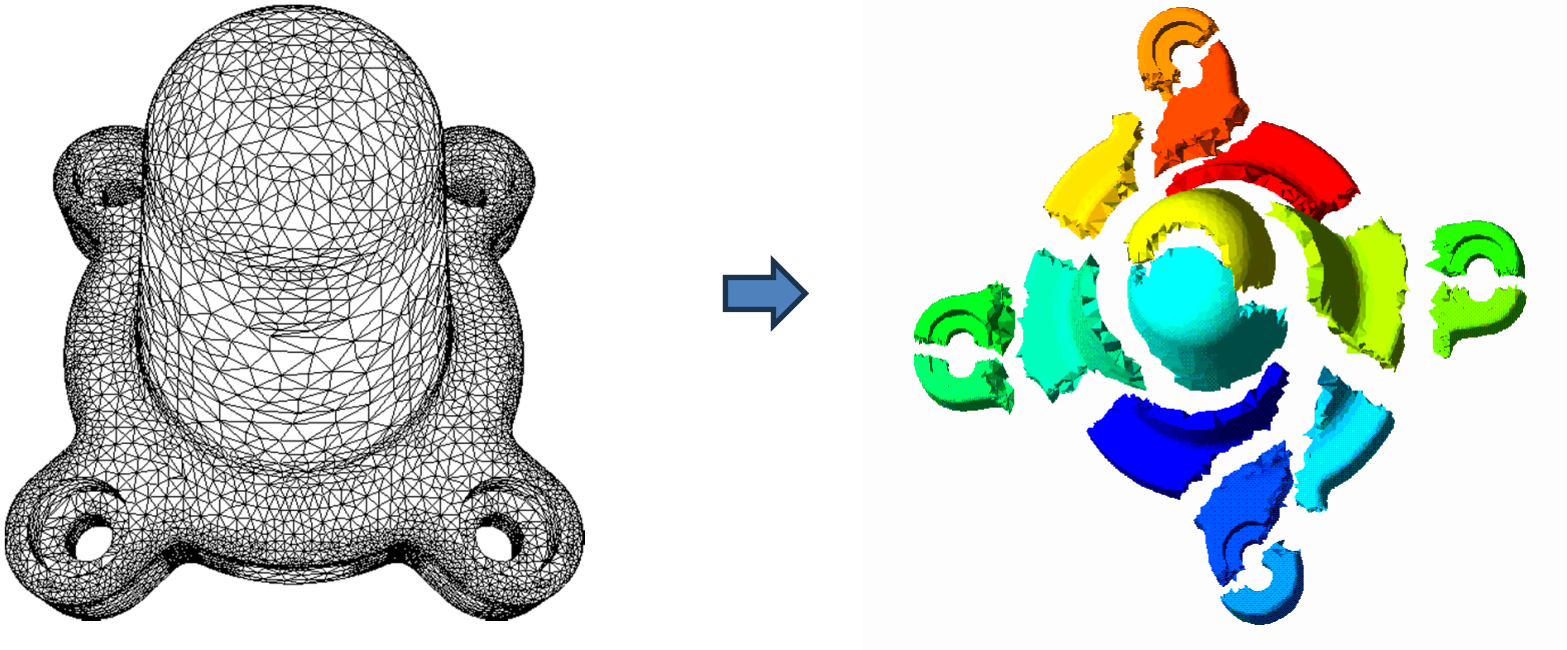
2-D



3-D

划分：域分解

- 不规则区域的分解示例

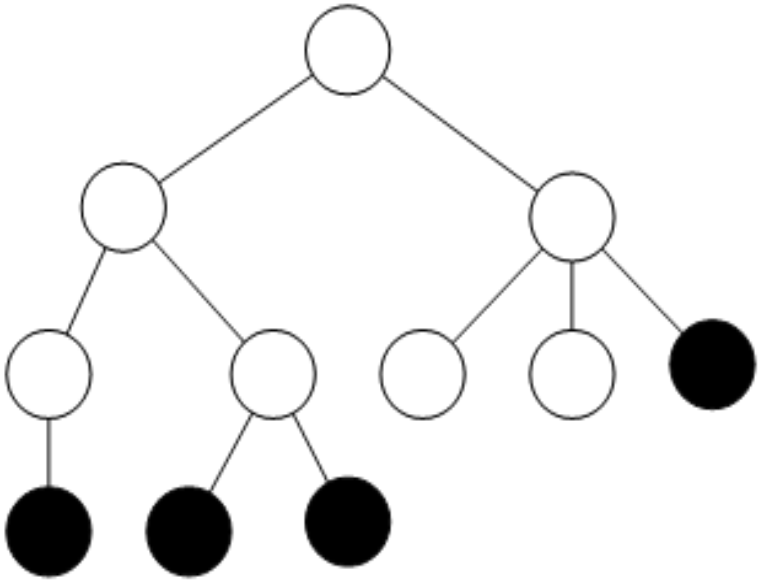


划分：功能分解

- **功能分解**划分的对象是计算，将计算划分为不同的任务，其出发点不同于域分解
- 功能分解的步骤
  - 首先关注被执行的计算的分解，而不是计算所需数据
  - 如果所作的计算划分是成功的，再继续研究计算所需的数据
  - 如果这些数据是不相交或相交很少的，就意味着划分是成功的；如果这些数据有相当的重叠，就会产生大量的通信，此时就暗示应考虑数据分解

划分：功能分解

- 功能分解示例：搜索树
- 搜索树没有明显的可分解的数据结构，但易于进行细粒度的功能分解



- 开始时根结点生成一个任务
- 对其评价后，如果它不是一个解，就生成若干叶结点
- 这些叶结点可以分到各个处理器上并行地继续搜索

划分：判断依据

- 下面的问题可以帮助检查所作的划分是否合理

检查项	问题
✓ 所划分的任务数是否高于目标机上处理器数目一个量级?	若不是，在后面的设计步骤中将缺少灵活性
✓ 划分是否避免了冗余的计算和存储要求?	若不是，则产生的算法对大型问题可能不是可扩展的
✓ 各任务的尺寸是否大致相当?	若不是，则分配处理器时很难做到负载平衡
✓ 划分的任务数是否与问题尺寸成比例?	理想情况下，问题尺寸的增加应引起任务数的增加而不是任务尺寸的增加。若不是这样，算法可能不能求解更大的问题，尽管有更多的处理器
✓ 是否采用了几种不同的划分法?	多考虑几种选择可以提高灵活性。同时既要考虑域分解又要考虑功能分解

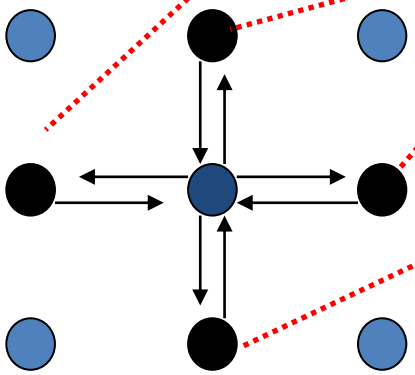
通信方法描述

- 由划分产生的各任务一般都不能完全独立地执行
- 各任务之间需要交换数据和信息，这就产生了通信的要求
- 通信就是为了进行并行计算，诸任务之间所需进行的数据传输
- 诸任务是并发执行的，通信则限制了这种并发性
- 通信的四种模式
  - 局部/全局通信
  - 结构化/非结构化通信
  - 静态/动态通信
  - 同步/异步通信

通信：局部通信

- **局部通信中，每个任务只与少数的几个近邻任务通信**

$$x_{i,j}(k)=\frac{4x_{i,j}(k-1)+x_{i-1,j}(k-1)+x_{i+1,j}(k-1)+x_{i,j-1}(k-1)+x_{i,j+1}(k-1)}{8}$$



局部通讯
通讯限制在一个邻域内

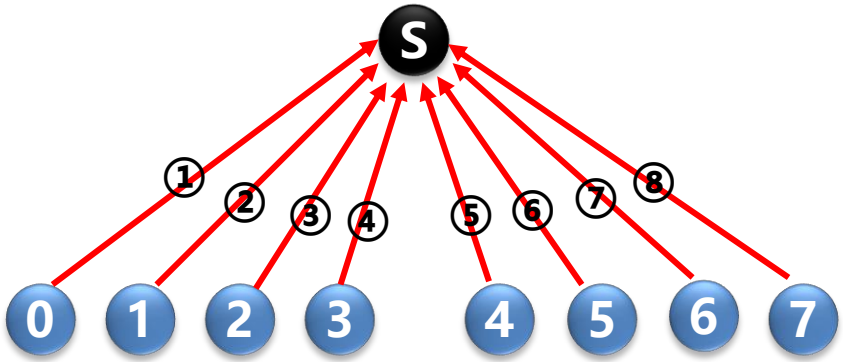
- 例：数值计算中的雅可比有限差分法
- 处于(i,j)位置上的处理器负责计算 $x_{i,j}$
- 计算每个$x_{i,j}(k)$时，(i,j)位置上的处理器只需与其上、下、左、右的邻居处理器通信
- 在获得数据的同时把自己的$x_{i,j}(k\text{-}1)$发送给邻居处理器

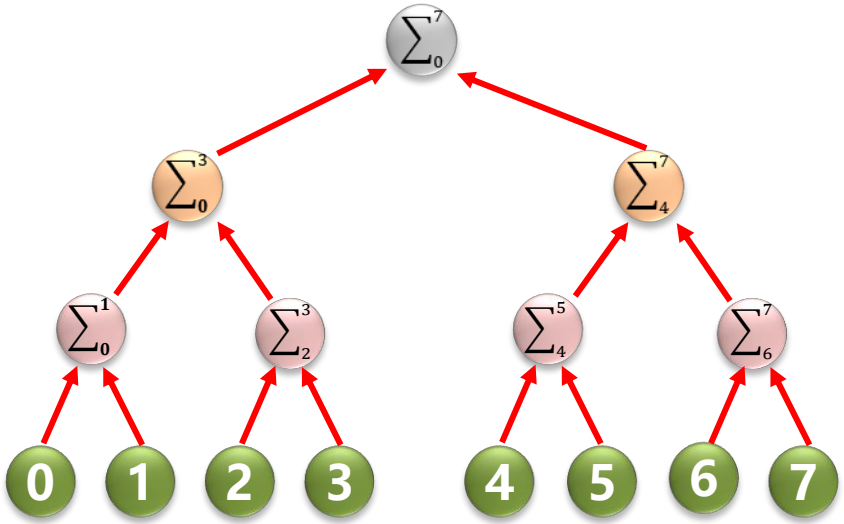
通信：全局通信

- **全局通信中，每个任务要与很多别的任务通信**

$$S = \sum_{i=0}^{N-1} x_i$$

使用一个根进程负责从各个进程一次接收一个值(x_i)，并进行累加，这时出现全局通信的局面



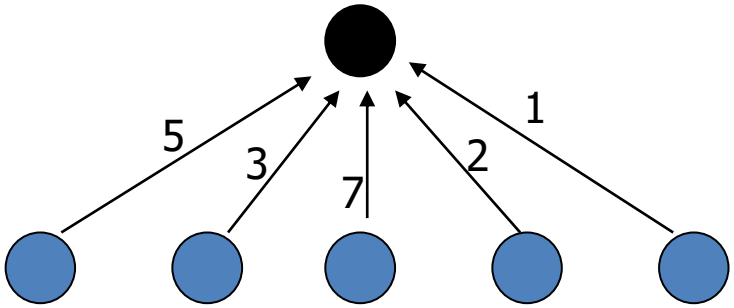


全局通讯
Master-Worker

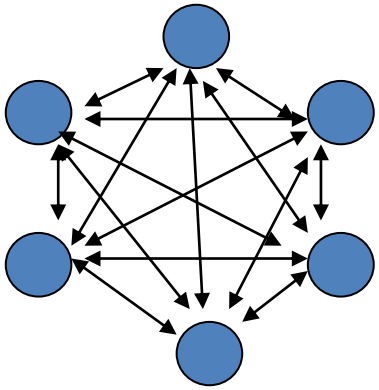
$$S = \sum_{i=0}^{2^n-1} x_i = \sum_{i=0}^{2^{n-1}-1} x_i + \sum_{i=2^{n-1}}^{2^n-1} x_i$$

通信：全局通信

- **全局通信中，每个任务要与很多别的任务通信**



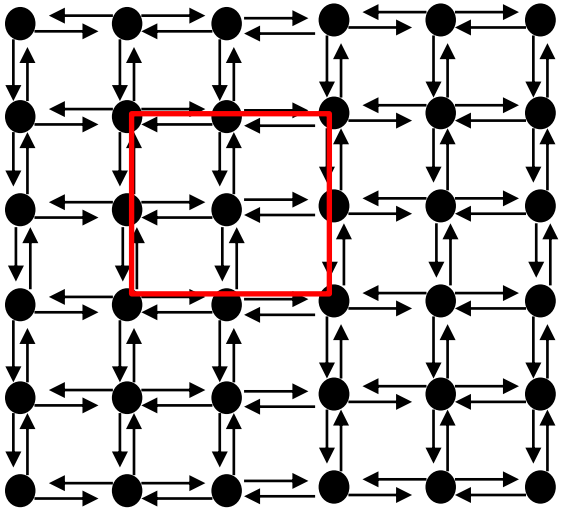
全局通讯
Master-Worker



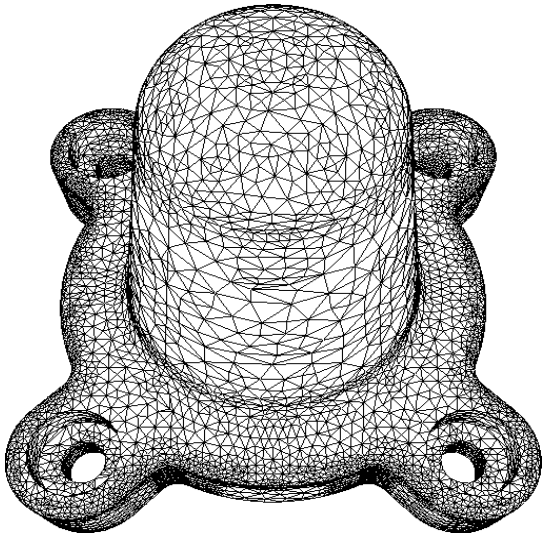
全局通讯
All-to-All

通信：结构化/非结构化通信

- **结构化通信中，一个任务和其近邻形成规则的结构（如树、网格等）**
- **非结构化通信中，通信网可能是任意图**



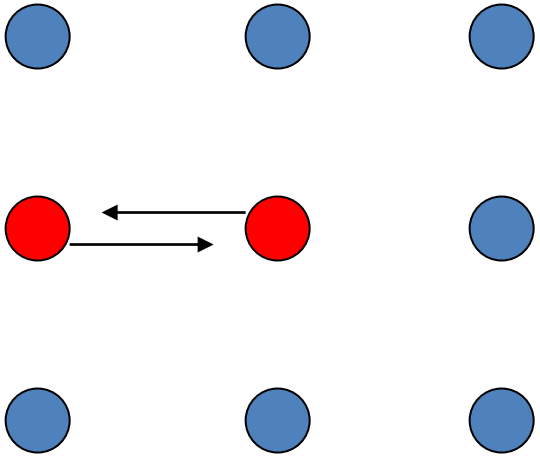
结构化通讯



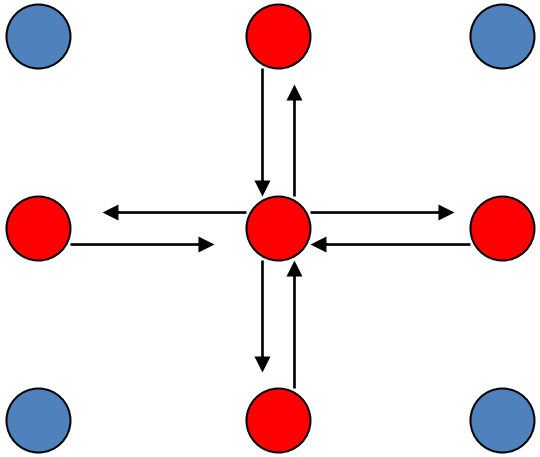
非结构化通讯

通信：静态/动态通信

- **静态通信中，通信伙伴不随时间变化**
- **动态通信中，通信伙伴可能动态变化，可以由运行时计算的数据确定**



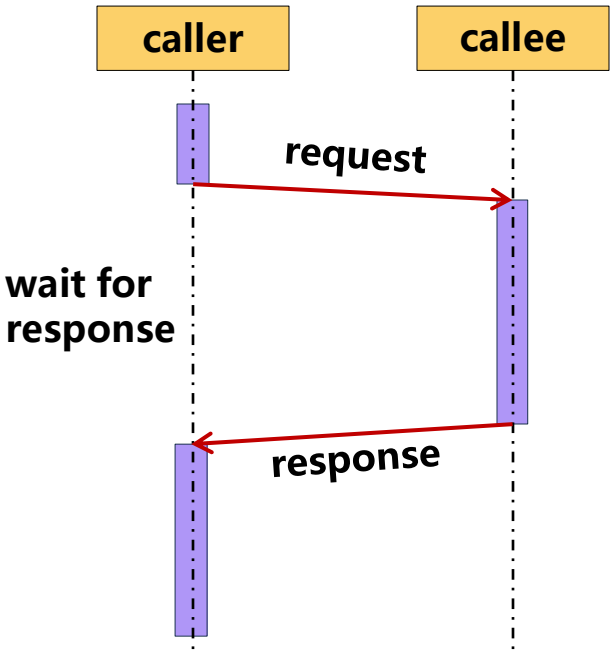
静态通讯



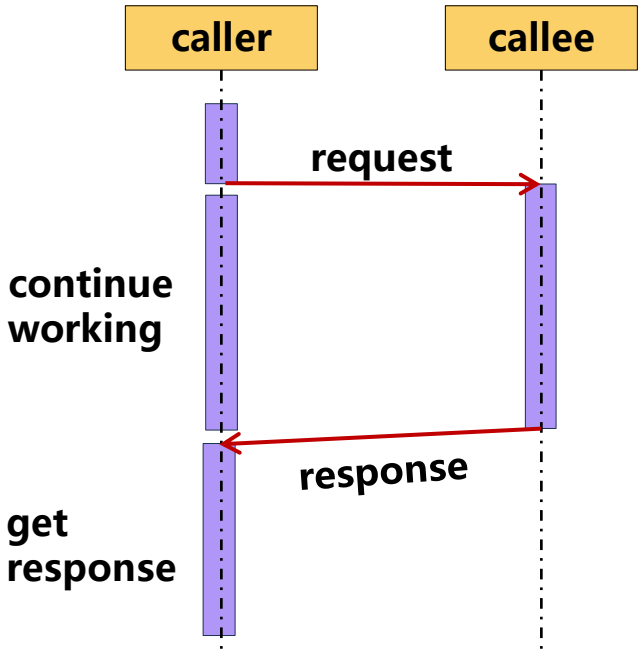
动态通讯

通信：同步/异步通信

- **同步通信中，接收方和发送方协同操作**
- **异步通信中，接收方获取数据无需与发送方协同**



同步通讯



异步通讯

通信：判断依据

- 下面的问题可以帮助检查通信设计是否合理

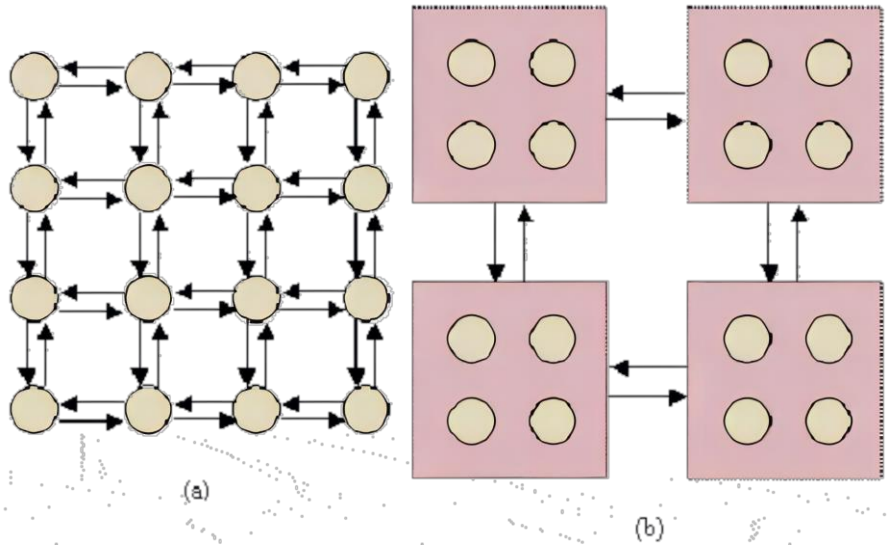
检查项	问题
✓ 所有任务是否执行大致同样多的通信?	若不是，所设计的算法的可扩展性可能会不好
✓ 每个任务是否只与少数的近邻通信?	若不是，则可能导致全局通信。此时应设法将全局通信换成局部通信
✓ 诸通信操作能否并行执行?	若不能，所设计的算法可能是低效的和不具可扩展性的。此时可试用分治策略来开发并行性
✓ 不同任务的计算能否并行执行?	若不能并行执行，所设计的算法可能是低效的和不具可扩展性的。此时可考虑重新安排通信和计算的顺序以改善这种情况
✓ 是否会因为等待数据而降低并行度?	

组合方法描述

- 组合是由抽象到具体的过程，是将组合的任务能在一类并行机上有效的执行
- 目的:合并小尺寸任务，减少任务数量和通信开销
- 通过增加任务的粒度和重复计算，减少通讯成本
- 保持映射和扩展的灵活性，降低软件工程成本
- 组合需要考虑两个因素
  - 表面-容积效应
  - 重复计算

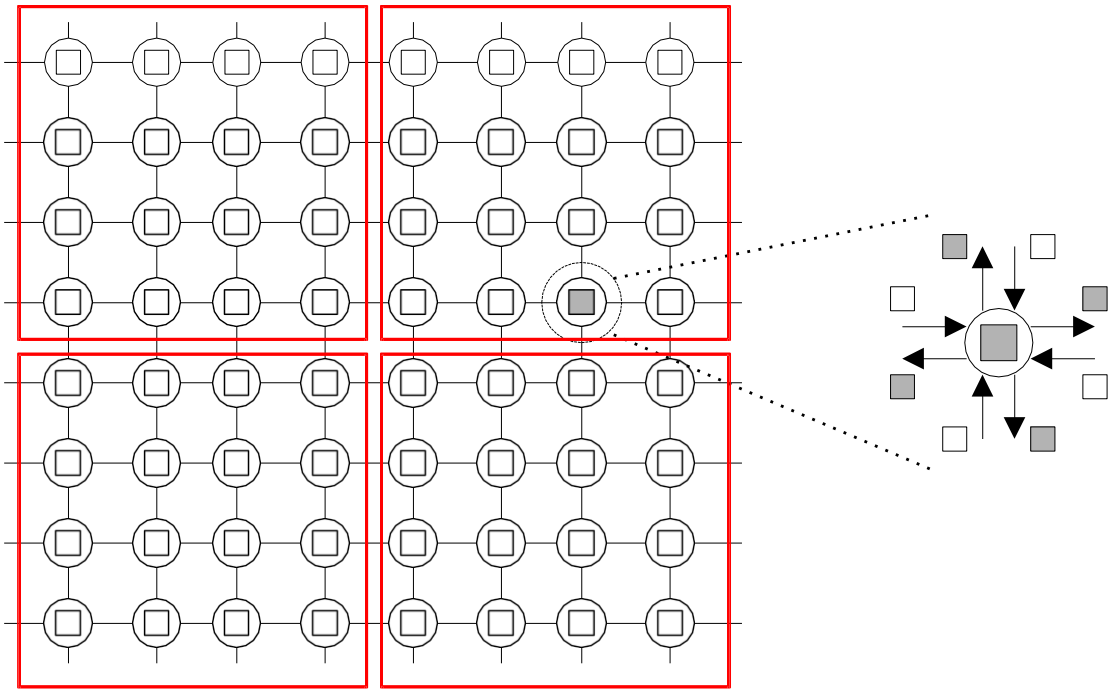
组合：表面-容积效应

- 任务的通信需求正比于它所操作的数据域的表面积，计算需求正比于它所操作的数据域的容积
- 计算单元的通信与计算之比随任务尺寸的增加而减小



- 假设需要计算的数据是4×4矩阵。如果把计算每个元素算作一个任务，则有16个任务。每轮迭代中，每个任务都需要与其上下左右的任务通信，共需48次通信
- 将相邻的四个元素的计算作为一个任务则只需8次通信

组合：表面-容积效应



- 细粒度二维网（8*8网络）
  - 8*8=64个任务
  - 64*4=256次通信
- 每个任务通信4次，共传输256个数据

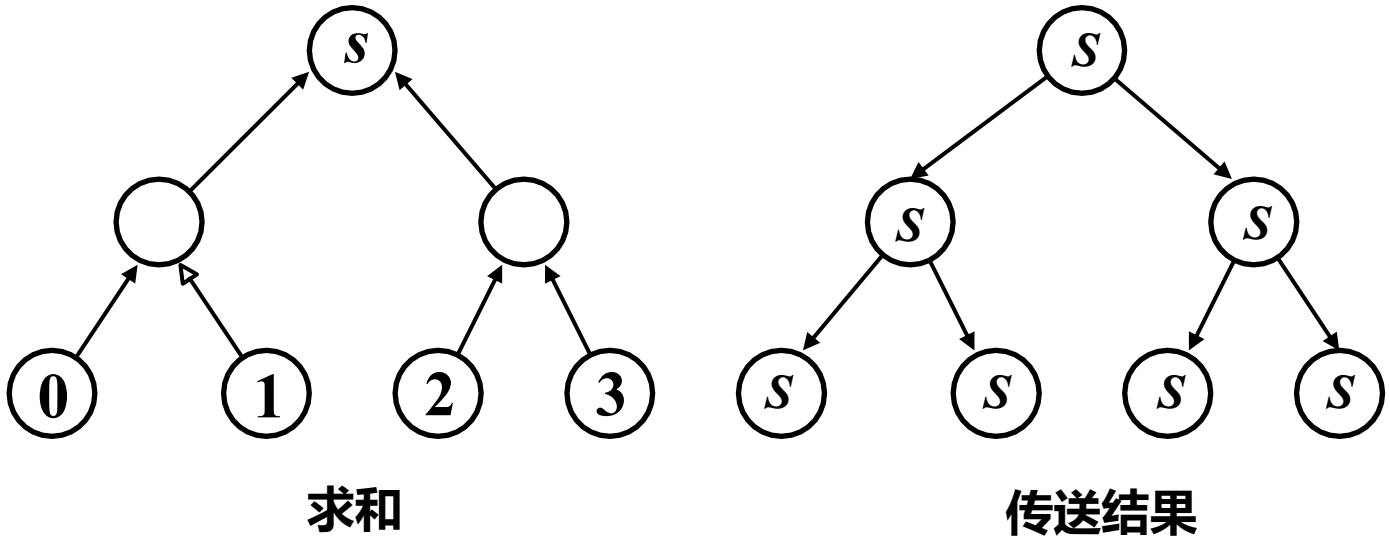
- 粗粒度二维网
  - 2*2=4个任务
  - 4*4=16次通信
- 每个任务通信4次，共传输16个数据

组合：重复计算

- 采用冗余的计算来减少通信和/或整个计算时间
- 重复计算减少通讯量，但增加了计算量，应保持恰当的平衡，最终目标应减少算法的总运算时间

组合：重复计算

- 示例：假定在二叉树上求N个数的和，且要求最终在每个处理器上都有该结果

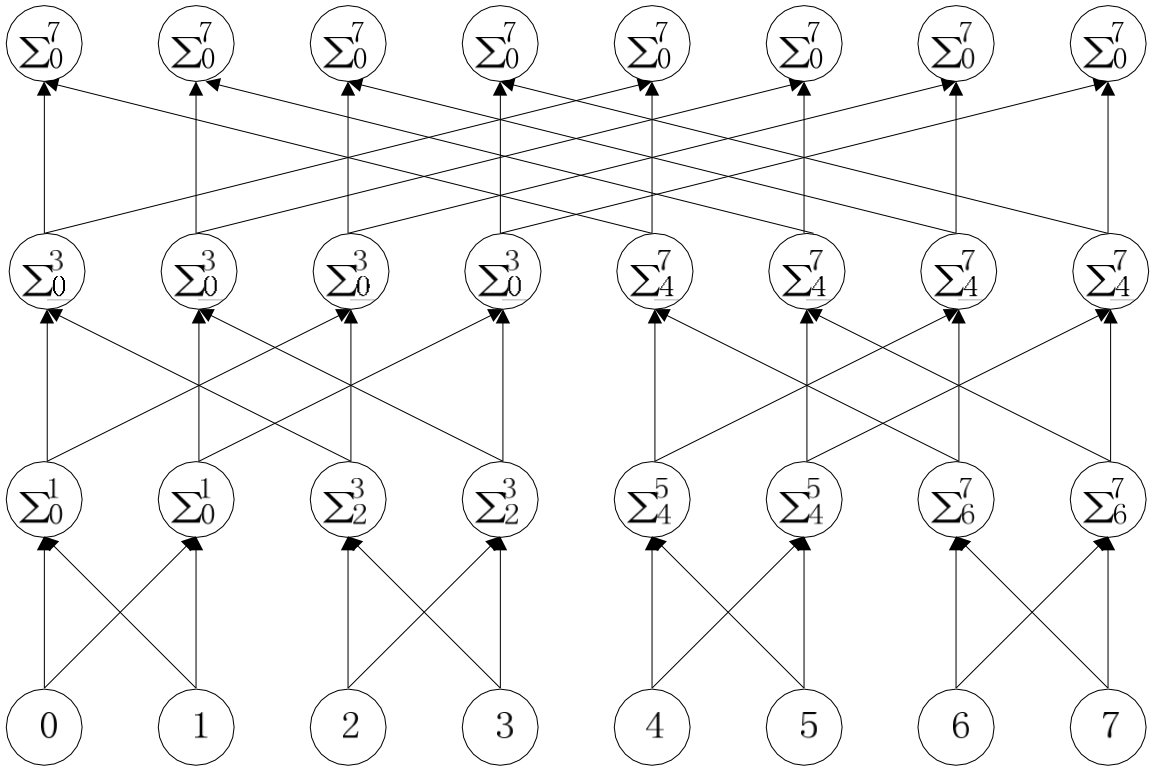


求和　　传送结果

先自叶向根求和，得到结果后再自根向叶播送，共需2logN步

组合：重复计算

- 示例：假定在二叉树上求N个数的和，且要求最终在每个处理器上都有该结果



蝶式结构求和，使用了重复计算，共需logN步

组合：判断依据

- 下面的问题可以帮助检查所进行的组合是否合理

检查项
✓ 增加粒度是否减少了通讯开销?
✓ 重复计算是否已权衡了其得益?
✓ 是否保持了灵活性和可扩展性?
✓ 组合的任务数是否与问题尺寸成比例?
✓ 是否保持了类似的计算和通讯?
✓ 有没有减少并行执行的机会?

映射方法描述

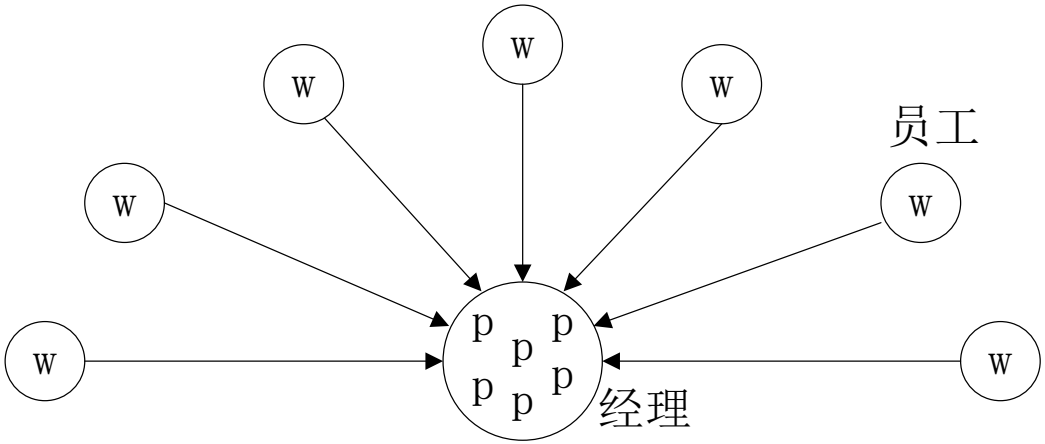
- 任务要映射到具体的处理器，定位到运行机器上
- 目标：最小化全局执行时间和通信成本，最大化处理器的利用率，减少算法的总执行时间
  - 把并发执行的任务放在不同的处理器上以增加并行度
  - 把需频繁通信的任务置于同一处理上以提高局部性
- 映射实际是一种权衡，属于NP完全问题
- 映射过程所用到的两类算法
  - 负载平衡算法
  - 任务调度算法

映射：负载均衡算法

- 任务的工作量不同，通信是非结构化的，可采用负载平衡算法
- **局部算法**：通过从近邻迁入任务和向近邻迁出任务来达到负载平衡
- **概率方法**：将任务随机地分配给处理器，如果任务足够多，则每个处理器预计能分到大致等量的任务
- **循环映射**：轮流地给处理器分配计算任务

映射：任务调度算法

- 任务放在集中的或分散的任务池中，使用任务调度算法将池中的任务分配给特定的处理器
- 经理/雇员模式：一个进程（经理）负责分配任务，每个雇员向经理请求任务，得到任务后执行任务
- 非集中模式：无中心管理者的分布式调度法



映射：判断依据

- 下面的问题可以帮助检查映射设计得是否合理

检查项
✓ 如果采用集中式负载均衡方案，是否检查了中央管理者不会成为瓶颈?
✓ 如果采用动态负载均衡方案，是否衡量过不同策略的成本?
✓ 如果采用概率或循环映射法，是否有足够多的任务？一般地，任务数应不少于处理器数的10倍。

PCAM设计原理小结

- 划分：域分解和功能分解
- 通讯：任务间的数据交换
- 组合：任务的合并使得算法更有效
- 映射：将任务分配到处理器，并保持负载均衡

目录

- PCAM设计原理
- **并行程序性能评价**
- 并行程序编程模型

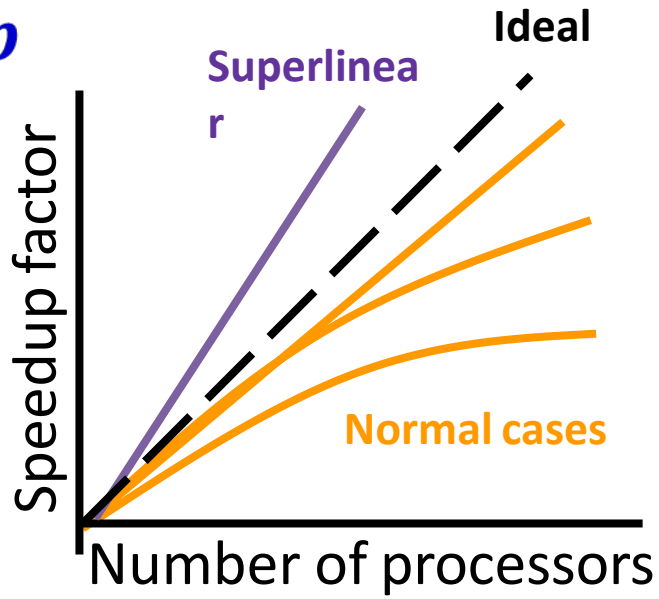
加速比（Speedup）

- **程序加速比计算**
  - T_s：使用最佳串行算法的执行时间
  - T_p：使用p个处理器时的执行时间

$$S(p) = \frac{T_s}{T_p}$$

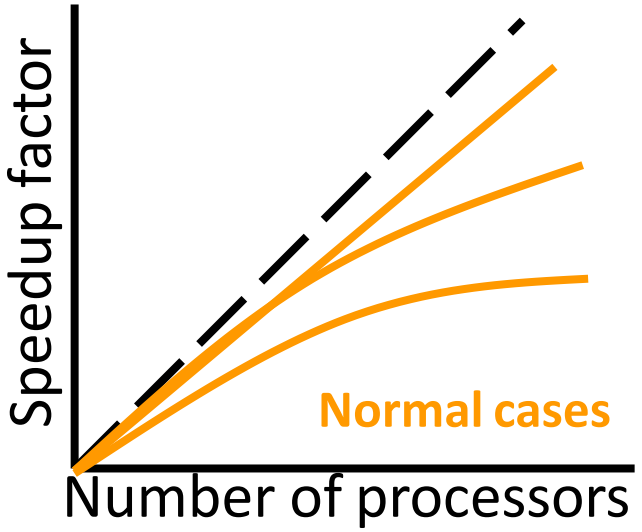
- **线性加速比**
  - 理论上的最大加速比（理想） $S(p) = p$
- **超线性加速比** $S(p) > p$
  - 实践中偶尔会发生
  - 额外的硬件资源（如内存）
  - 软/硬件深度优化（如缓存）
- **系统效率（Efficiency）**

$$E(p) = \frac{T_s}{T_p \times p} = \frac{S(p)}{p} \times 100\%$$



加速比（Speedup）

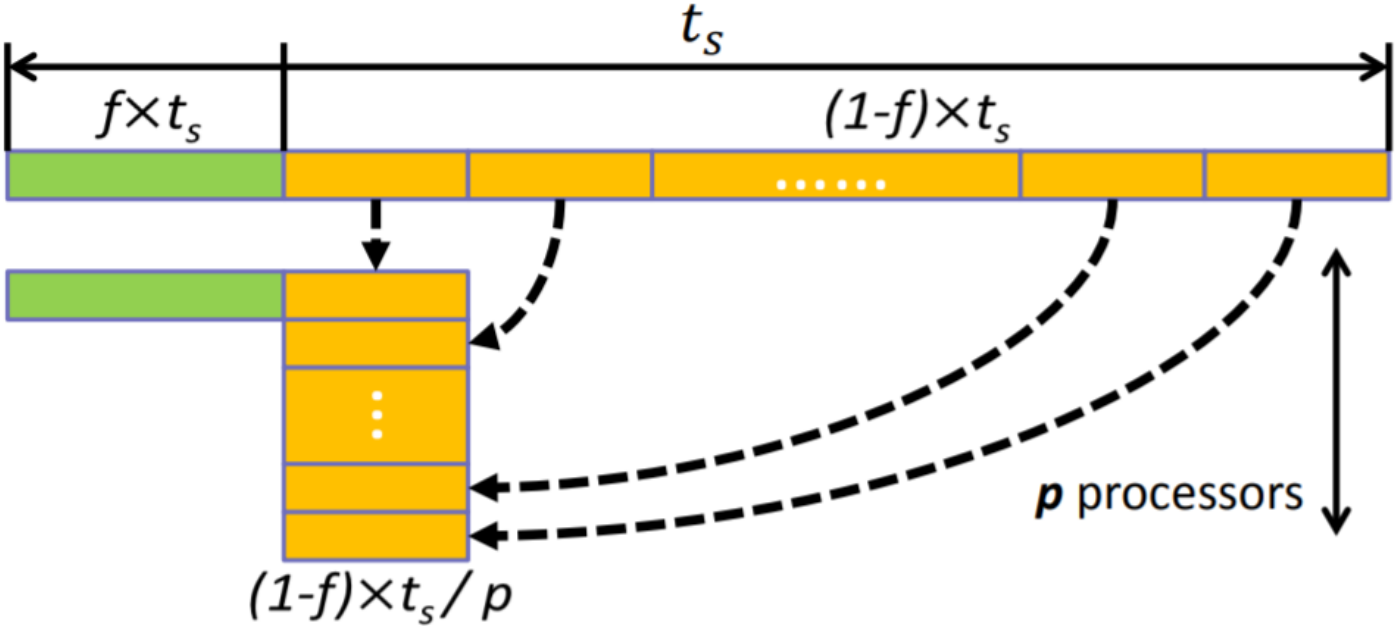
- 理想（最大）加速比很难达到 $S(p) = p$
- 程序中并非每个部分的计算都可以并行化，使得处理器空闲
- 并行版本中需要额外的计算（即用于的同步成本）
- 进程之间需要通讯（通讯成本，通常是主要因素）



Amdahl定律

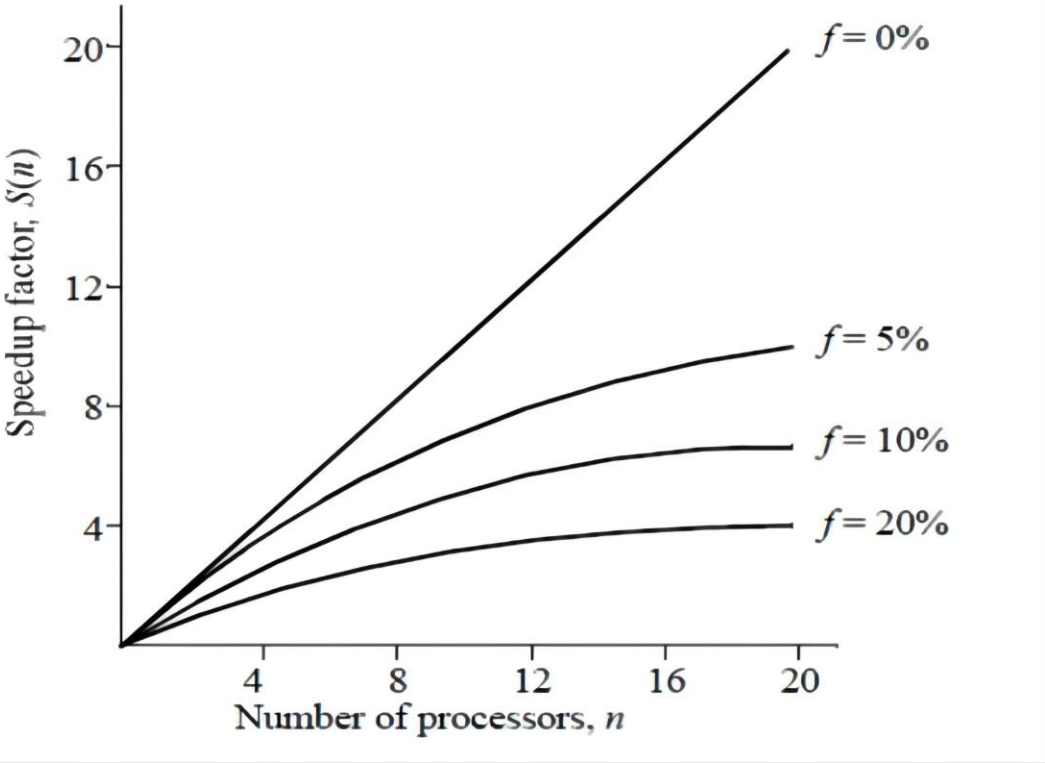
- Amdahl定律定义了串行程序并行化后加速比计算公式与理论上限
- f 表示程序中不可以被并行化的部分所占的比例

$$S(p) = \frac{t_s}{f \times t_s + (1-f) \times t_s/p} = \frac{p}{1 + (p-1) \times f}$$



Amdahl定律

- 处理器数量无穷大时，$S(p)_{p\to\infty}\frac{p}{1+(p-1)\times f}=\frac{1}{f}$
- 并行代码所占的百分比固定的情况下，随着处理器数量的增加，对并行效率的提升会固定在一定比例



Amdahl定律

- 固定不变的计算负载
- 固定的计算负载分布在多个处理器上的
- 增加处理器加快执行速度，从而达到加速的目的

Gustafson定律

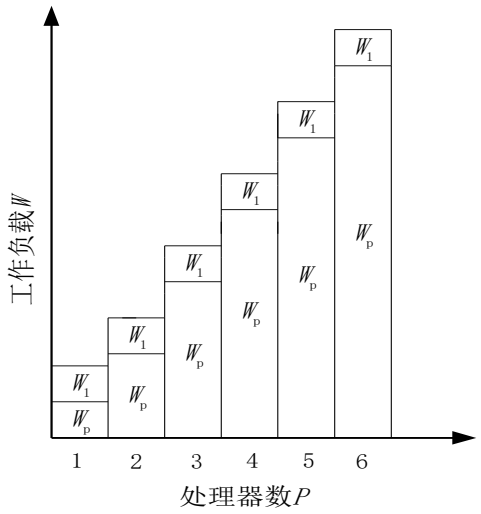
- Amdahl 定律有一个重要前提，就是处理的数据集大小是固定的，但是这在大数据计算的领域里，这个假设并不经常能达到，因为人们总是会为了在短时间内处理更多的数据
- 很多大型计算，精度要求很高，即在此类应用中精度是一个关键因素，而计算时间是固定不变的。此时为了提高精度，必须加大计算量，相应的也必须增加处理器的数目来完成这部分计算，以保持计算时间不变
- 研究在给定的时间内用不同数目的处理器能够完成多大的计算量是并行计算中一个很实际的问题

Gustafson定律

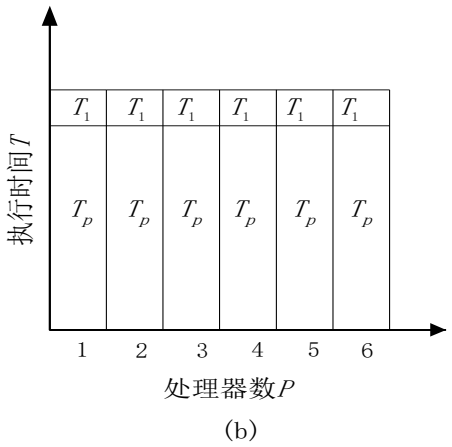
- t_s，t_p表示串行部分和并行部分执行的时间
- f表示程序中不可以被并行化的部分的所占的比例

$$f = \frac{t_s}{t_s + t_p}$$

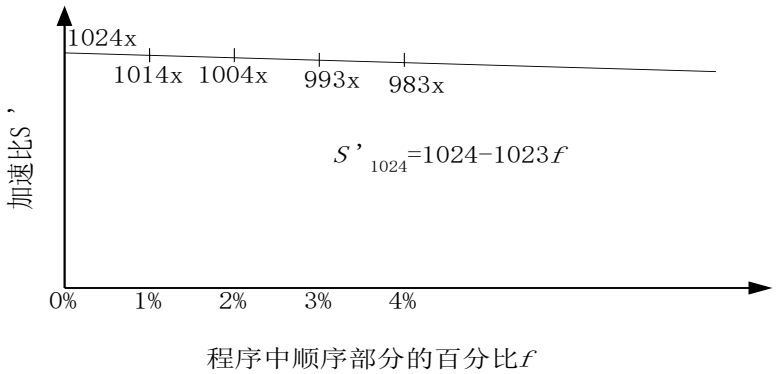
- $$S(p) = \frac{t_s + p \times t_p}{t_s + p \times t_p / p} = \frac{t_s + p \times t_p}{t_s + t_p} = f + p \times (1 - f)$$



(a)



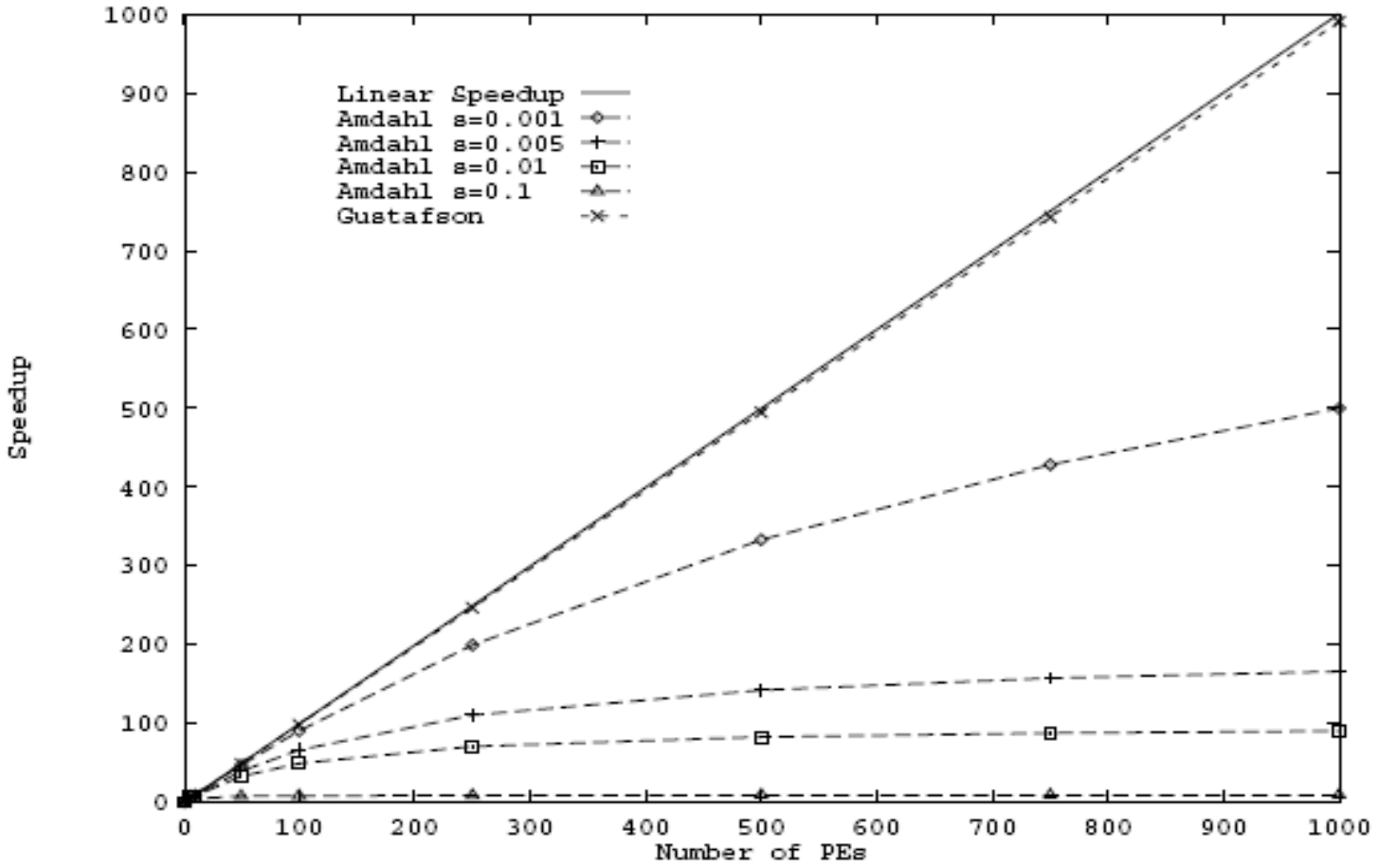
(b)



(c)

Amdahl定律 vs Gustafson定律

- 刻画的内容等价，可相互推导

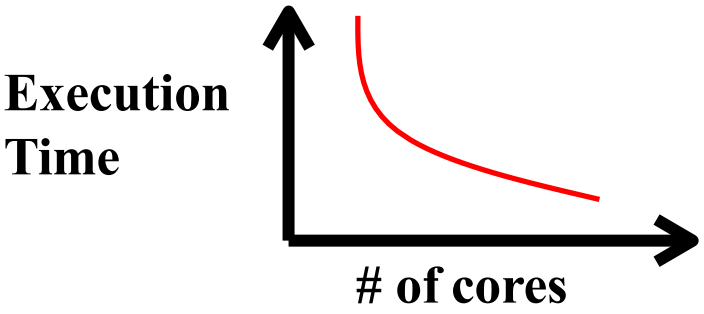


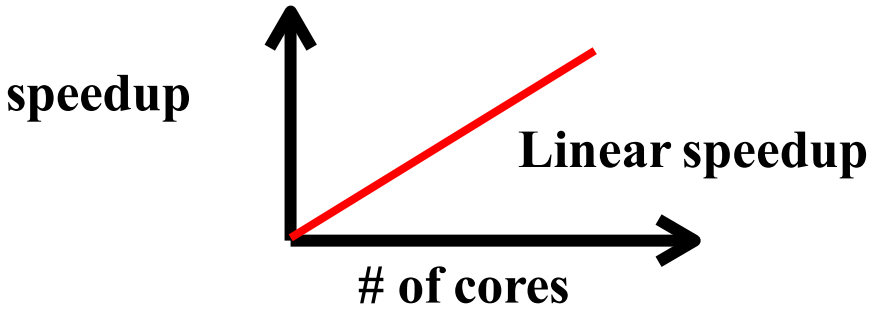
可扩展性（Scalability）

- 确定的应用背景下，计算机系统（或算法或程序等）性能随处理器数的增加而按比例提高的能力
- 可扩展性分类
  - 强可扩展性（Strong Scalability）
  - 弱可扩展性（Weak Scalability）

强可扩展性（Strong Scalability）

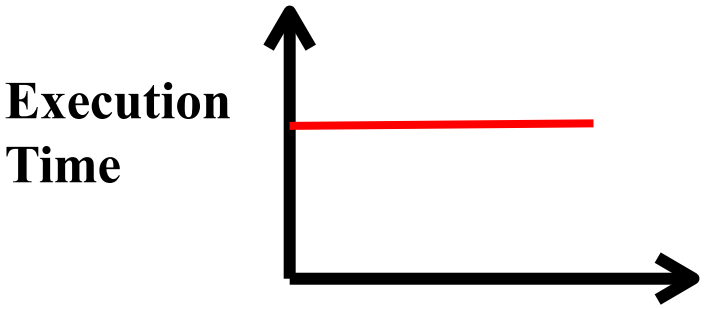
- 问题规模不变，处理器的数量增加
- 用于寻找一个平滑点，可以在可接受的时间内完成计算（用多少台机器解决这个问题是最好的）
- 如果加速比等于处理器的数量，则称为线性扩展（Linear Scaling）

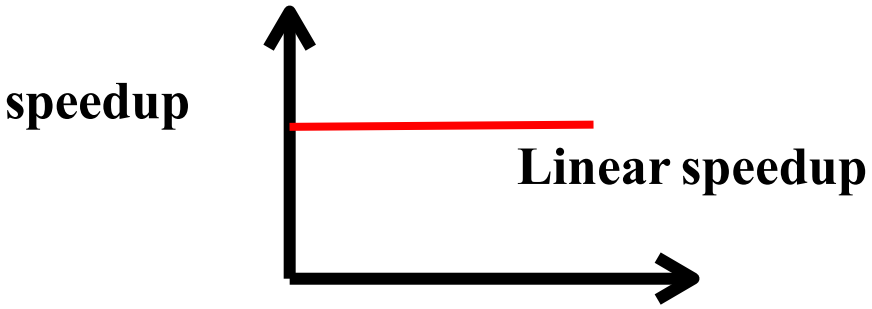




弱可扩展性（Weak Scalability）

- 随着处理器的增加，问题的规模同比增加
- 目标是在相同的时间内解决更大规模的问题
- 随着负载的增加，执行时间保持不变，则称为线性扩展





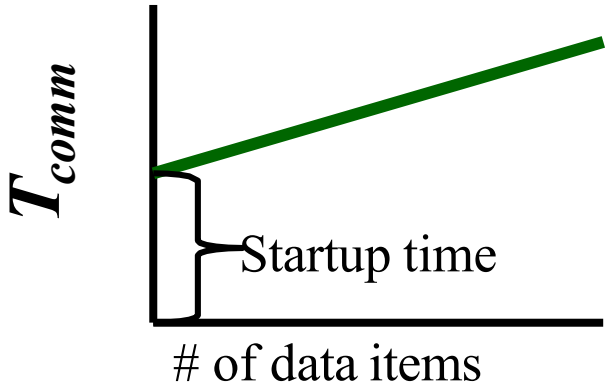
强可扩展性 vs 弱可扩展性

- 强可扩展性：线性扩展难实现，因为通讯开销可能与系统规模（节点数、处理器数）同比例增加
- 弱可扩展性：线性扩展更容易实现，因为程序通常采用最近邻通信模式，其中通信开销相对恒定，而与使用的进程数无关

时间复杂度（Time Complexity）

- $T_p = T_{comp} + T_{comm}$
  - T_p 并行算法的总体执行时间
  - T_{comp} 计算部分时间
  - T_{comm} 通讯部分时间

- $T_{comm} = q(T_{startup} + nT_{data})$
  - $T_{startup}$ 信息延迟（假设固定值）
  - T_{data} 单位数据的传输时间
  - n：信息中的数据量
  - q：信息的数量



时间复杂度——示例

- 算法步骤
  - Step 1：节点1发送$n/2$个数字给节点2
  - Step 2：两个节点同时对$n/2$个数字进行计算
  - Step 3：节点2发送部分和的计算结果给节点1
  - Step 4：节点1将部分和相加得出最终结果
- 时间复杂度分析
  - Computation(for step 2 & 4)

$$T_{comp} = \mathrm{n}/2 + 1 = O(n)$$

  - Communication(for step 1 & 3)

$$T_{comm} = (T_{startup} + n/2 \times T_{data}) + (T_{startup} + T_{data})$$
$$= 2T_{startup} + (n/2 + 1) \times T_{data} = O(n)$$

  - 算法整体复杂度：$O(n)$

目录

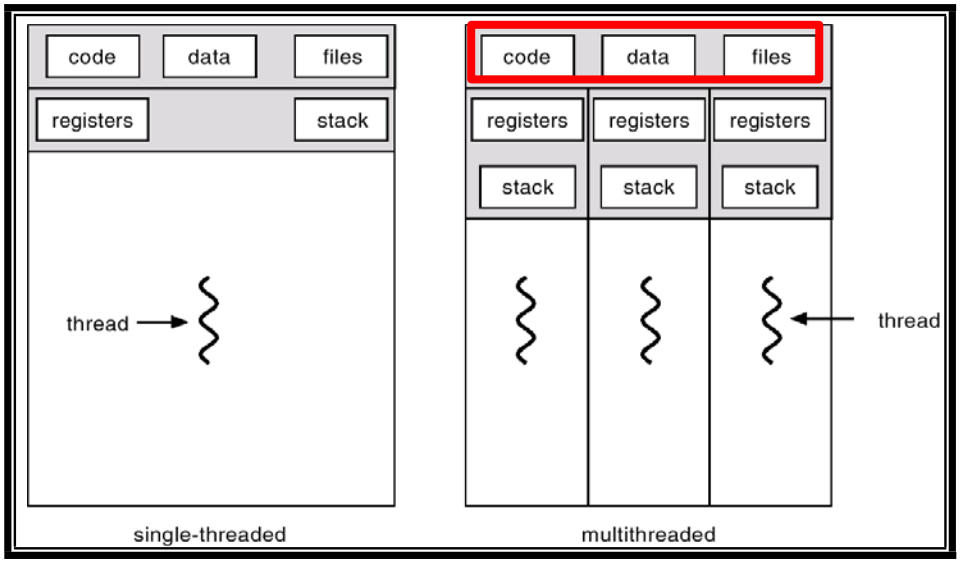
- PCAM设计原理
- 并行程序性能评价
- **并行程序编程模型**

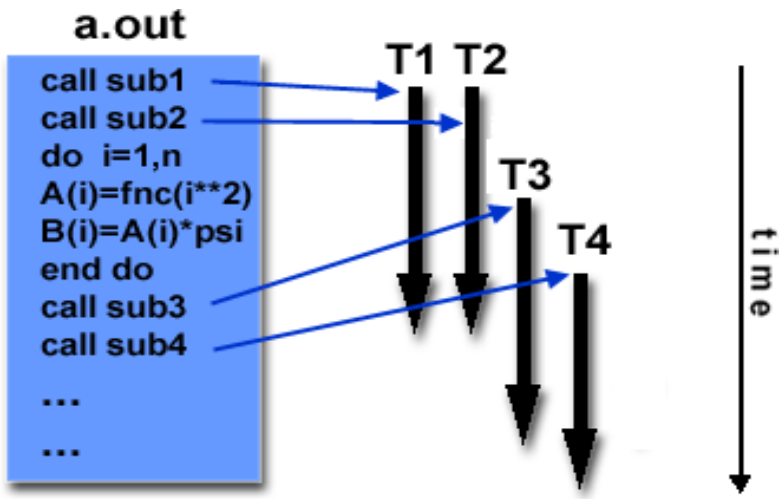
并行编程模型

- 硬件和内存架构之上的一种抽象
- 一般来说，编程模型的设计与计算机体系结构相匹配
  - 共享内存编程模型 → 共享内存系统
  - 消息传递编程模型 → 分布式内存系统
- 编程模型不受机器或内存体系结构的限制
  - 共享内存系统上可以支持消息传递编程模型：例如，单个服务器上的MPI
  - 分布式内存系统上支持共享内存编程模型：例如，Partitioned Global Address Space（PGAS）

共享内存编程模型

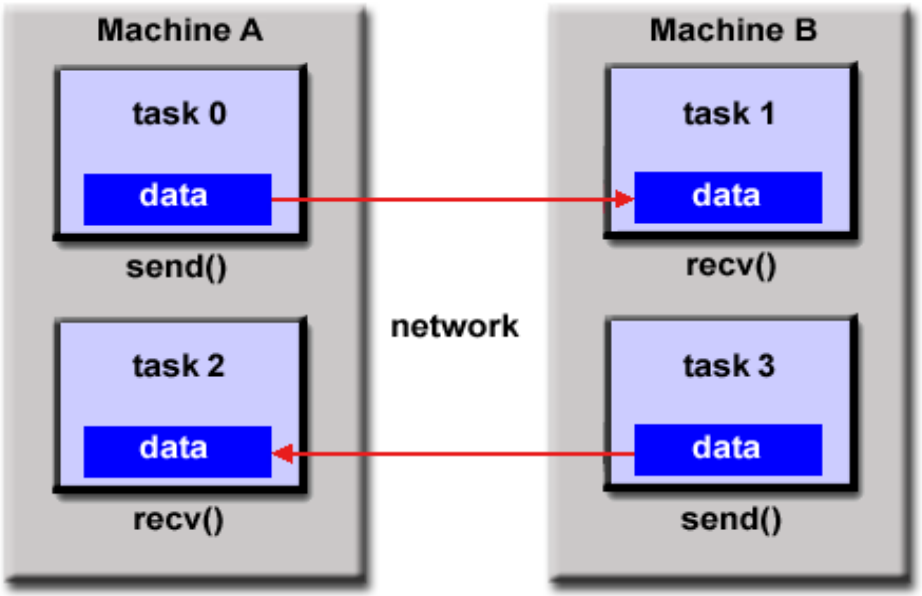
- 单个进程可以有多个并发执行路径
- 线程有本地数据，但也共享资源
- 线程通过全局内存（Global Memory）相互通信
- 线程可以产生和消失，但主进程始终存在
  - 提供必要的共享资源，直到应用程序完成





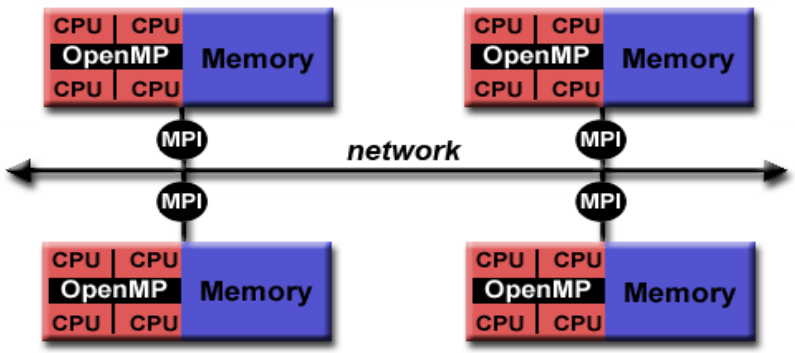
消息传递编程模型

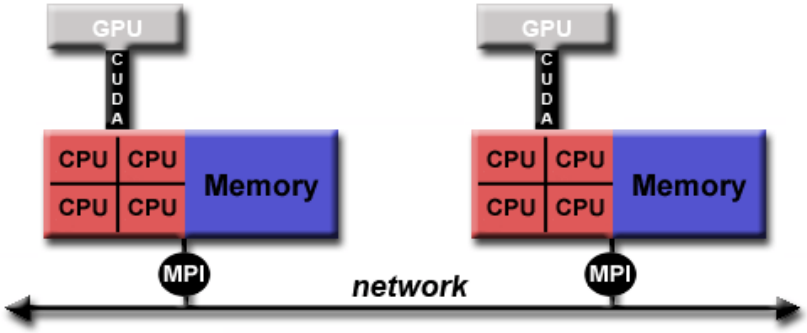
- 在计算过程中使用自己的本地内存的一组任务
  - 多个任务可以驻留在同一台物理机器上和/或跨任意数量的机器
- 任务通过发送和接收消息（内存副本）的通信过程来交换数据
- 典型代表：MPI
  - Send，Recv，Bcast
  - Gather，Scatter等



混合编程模型

- 混合模型结合了前面描述的多个编程模型
  - 消息传递模型（MPI）与共享内存模型（OpenMP）的组合
  - 消息传递模型（MPI）与CPU-GPU混合编程
- 混合模型适用于目前流行的多核/众核集群环境





并行程序编程模型小结

- 编程模型和并行系统的设计和流行是相互影响的
- OpenMP、MPI、Pthreads、CUDA等只是一些供用户进行并行编程的并行语言
- 理解什么是并行计算甚至比如何进行并行编程更重要，因为理解并行计算，可以
  - 快速学习新的并行编程方式
  - 了解程序的性能和瓶颈
  - 优化程序的性能