Parallel Processing

性能评价方法：性能模型-并行程序性能评测

邵恩
高性能计算机研究中心

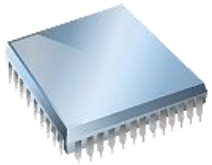
Outline

- **性能可扩展性**
- **分析性能评测——三大定律**
  - 阿姆达尔定律（Amdahl’ s law）
  - 古斯塔夫森-巴西斯定律（Gustafson-Barsis’ s Law）
  - 等效率定律 （Isoefficiency Law）
- **并行计算的编程模型**
  - PRAM模型
  - BSP模型
  - LogP模型

什么是性能?

- **在计算中，性能由两个因素决定**
  - 计算要求（需要做什么）
  - 计算资源（这样做的成本）
- **计算问题转化为：资源如何利用与使用的问题**
- **计算资源（或非计算资源）之间的相互作用和权衡（trade-off）**

$$\text{性能} \sim \frac{1}{\text{所需要的各项资源}}$$



Hardware Time Energy

……最终

Money

为什么我们关心性能?

- 性能本身是衡量计算需求得到满足程度的衡量标准
- 我们评估性能以了解需求和资源之间的关系
  - 决定如何改变“解决方案”以达到计算需求
- 性能指标（Performance measures）：反映了“解决方案”如何以及是否满足计算要求

“设计引擎（软件开发）最常见的困难是希望尽可能缩短执行计算的时间。”

查尔斯•巴贝奇（Charles Babbage），1791 – 1871

什么是并行性能?

- **这里我们关心的是使用并行计算环境时的性能问题**
  - 关于并行计算的性能
- **性能是并行的存在理由**
  - 并行性能 vs 串行性能
  - 如果“性能”不是更好，那么并行就没有必要了
- **并行处理包括并行计算所必需的技巧和技术**
  - 硬件、网络、操作系统、并行库、编程语言、编译器、算法、工具……
- **并行性必须带来性能提升**
  - 如何做？ 能有多好?

性能预期（损失）

- 如果每个处理器的额定值为 k MFLOPS，一个计算系统有 p 个处理器，那这个系统就一定有 k*p MFLOPS 的性能吗?
- 如果用 1 个处理器上求解一道题，需要花费 100 秒；那么用 10 个处理器求解这道题，就只需要花费 10 秒吗?
- 多种原因影响性能
  - 每个独立求解的计算任务，都必须单独执行
  - 但这些子任务，可以以复杂的方式相互作用
    - 完成一个子任务后，可能才会产生另一个子任务
    - 一个问题可能会影响另一个子任务的执行
- 改变规模 (系统规模或问题规模) 会影响性能满足的条件
- 需要了解性能空间

易并行计算问题 (Embarrassingly Parallel Computations)

- **易并行计算问题是指：可以明显划分为可以同时执行的完全独立的计算子任务**
  - 在”真正的”易并行计算问题中，独立进程之间没有交互
  - 在“近似的”易并行计算问题中，结果必须以某种方式分布和收集/组合（有极少数的非完全并行的部分）
- **易并行计算有可能在并行计算系统中，实现最大性能提升**
  - 如果按顺序需要T时间，就有可能在P个并行的处理器上，只需要花费T/P时间，就可以得到结果
  - 是什么导致情况并非总是如此?

可扩展性

- **一个程序可以扩展到使用多个处理器**
  - 这意味着什么?
- **如何评估可扩展性?**
- **如何评估可扩展性能够带来的优点?**
- **比较评价**
  - 如果处理器数量增加一倍，会发生什么?
  - 可扩展性是线性的吗?
- **使用并行效率进行度量（parallel efficiency）**
  - 随着问题规模的增加，计算效率是否保持不变?
- **应用的性能指标**

可扩展的并行计算

- **并行体系结构中的可扩展性**
  - 处理器数量
  - 内存体系结构
  - 互连网络
  - 避免关键硬件结构出现瓶颈
- **计算问题的可扩展性**
  - 问题的规模
  - 计算算法
    - 计算与内存访问比率
    - 计算与通信比率
- **并行编程模型和工具**
- **性能可扩展性**

为什么并行应用程序会有可扩展差的问题?

- **顺序执行**
- **关键路径**
  - 跨处理器的计算之间的依赖关系
- **计算业务带来的瓶颈**
  - 一个处理器处理一切
- **算法开销**
  - 有些事情只是需要对算法进行改造，调整对问题的求解，才能并行执行
- **通信开销**
  - 花在通信上的时间比例越来越大
- **负载不平衡**
  - 让所有处理器等待“最慢”的处理器
- **投机性的计算损失**
  - 并行做 A 和 B两个计算任务，但最终不需要 B

关键路径——长链

- **长链依赖**
  - 主要性能限制
  - 性能改进的阻力
- **诊断**
  - 性能停滞在一个（相对）固定值
  - 关键路径分析
- **解决方案**
  - 尽可能去除长链
  - 通过从关键路径中移除子任务，来缩短依赖链条

计算业务带来的瓶颈——一打多

- **如何检测?**
  - 一个处理器 A 忙，而其他处理器等待
  - 数据依赖于 A 产生的结果
- **典型情况：“一打多”**
  - **1-N的广播**
  - 一个处理器根据请求分配作业
- **解决方法：**
  - 更高效的通信方式
  - 主从的层次结构（多层）
- **程序可能不会长时间表达出瓶颈带来的不良影响**
- **瓶颈只会在问题规模扩展后出现**

算法开销——并行方式不当

- **解决同一个问题的不同求解方法（执行计算的顺序）**
- **在 1 个处理器上运行时，所有并行算法都是顺序执行**
- **所有并行算法都引入加法运算（为什么？最后算总账）**
  - 并行开销
- **并行算法的起点应该在哪里？**
  - 最佳顺序执行的算法，可能根本无法并行化
  - 或者，它不能很好地并行化（例如，不可扩展）
- **该怎么办？**
  - 选择并行开销最小的算法，去做并行
- **性能是关键**
  - 是否实现了更好的并行性能？
  - 必须与最佳性能的顺序执行算法进行比较

性能和可扩展性

- **测评**
  - 顺序执行时间 (T_{seq}) 的函数变量是
    - 问题规模和硬件体系结构
  - *并行执行时间* (T_{par}) 的函数变量是
    - 问题规模和并行体系结构
    - # 执行过程中使用的处理器（个数）
  - 受以下因素影响的并行性能
    - 算法+体系结构
- **可扩展性**
  - **并行算法实现的性能能够随：处理器数量和问题规模增长，同比例增长提升的能力**

性能指标和公式

- T_1 是使用1个处理器解一道题的执行时间
- T_p 是使用p个处理器的计算系统，解一道题的执行时间

- $S(p)$ 、(S_p) 是加速比

$$S(p) = \frac{T_1}{T_p}$$

- $E(p)$ 、(E_p) 是效率

$$Efficiency = \frac{S_p}{p}$$

- $Cost(p)$ 、(C_p) 是开销

$$Cost = p \times T_p$$

- 并行算法成本最优的
  - **并行实行的时间 = 串行执行的实际**；$T_p = T_1 \times p$
  - (当$C_p = T_1$, $E_p = 100\%$)

并行效率

- **并行效率分析是并行算法发展的一个关键点:**
  - 特定算法并行化的效率估计,
  - 估算对某类问题求解的最大并行加速比(求解一个问题的所有并行方法的效率估计)

“操作-操作数”图……(操作数是指运算的数据对象)

- **模型“操作-操作数”图可用于描述所选求解问题算法中的操作数之间的依赖关系**

- **为了简化问题，可以假设：**
  - 任何计算操作的执行时间都是相同的，并且等于 1（在某些测量单位中）
  - 计算设备之间的数据传输是即时进行的，没有任何时间消耗。

“操作-操作数”图...

让我们将操作数与操作之间存在的信息依赖关系，表示为无环有向图

$$G=(V,R),$$

$V = \{1,\ldots,|V|\}$ 是图顶点的集合，表示正在执行的算法操作

R 是边集合，r(i,j) 表示操作j依赖于操作i计算结果

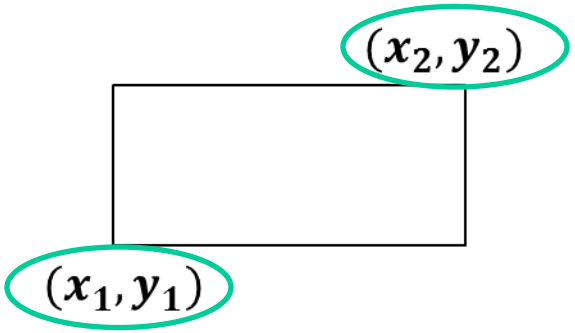
没有入度的点，表示“输入”操作，而没有出度的点表示“输出”操作。

$\bar{V}$ 表示没有入度的图顶点集合，

$d(G)$ 表示直径（即：最大路径的长度）。

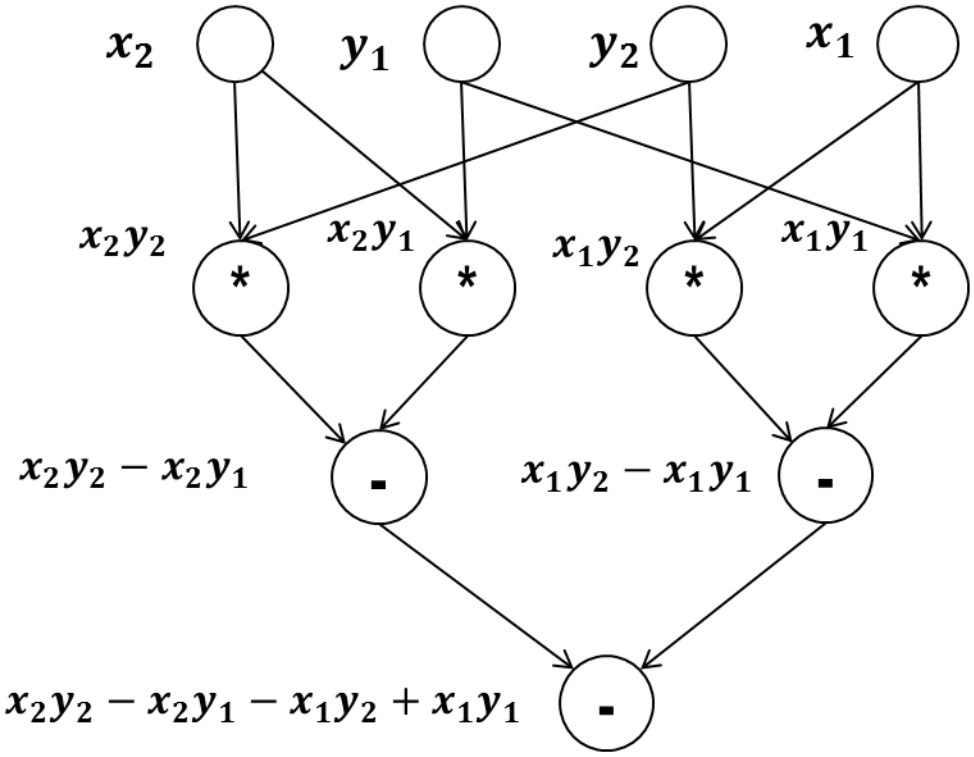
“操作-操作数”图...

Example: 图示算法表示，通过两个对角的坐标，求解矩形的面积



$$S = (x_2 - x_1)(y_2 - y_1)$$
$$= x_2y_2 - x_2y_1 - x_1y_2 + x_1y_1$$

算法对矩形面积的求解公式



“操作-操作数”图...

- 不同的计算方案具有不同的并行化能力。
- 在构建计算模型时必须考虑选择**最适合并行执行的**计算方案（即：算法的实现）

- 在选定的计算方案中，算法操作之间如果没有**依赖边**，则可以**并行执行**多个操作

算法的并行执行方案

- **令 p 为执行算法的处理器数量。 然后要并行执行计算，必须指定集合（计划）：**

$$H_P = \{(i, P_i, t_i) : i \in V\}$$

  - **i** 是操作数,
  - P_i 是执行操作 i 的处理器数量,
  - t_i 是操作开始时间.
- **需要满足以下要求:**
  - **操作对处理器的独占**：同一个处理器不能同时分配给不同的操作

$$\forall i, j \in V : t_i = t_j \Rightarrow P_i \neq P_j,$$

  - **避免等待**：所有必要的数据必须在操作执行开始之前计算出来（j操作**依赖**于i操作的结果）

$$\forall (i, j) \in R \Rightarrow t_j \geq t_i + 1$$

并行执行时间的评估……

- 并行算法模型：

$$A_P(G, H_P)$$

- 并行算法执行的时间由调度中**最长执行时间的操作**决定：

$$T_P(G, H_P) = \max_{i \in V}(t_i + 1)$$

- 最佳调度下并行算法执行时间：

$$T_P(G) = \min_{H_P} T_P(G, H_P)$$

并行执行时间的评估……

- 可以通过适配最佳计算方案来减少执行时间：

$$T_P = \min_G T_P(G)$$

- 如果使用无限数量的处理器，并行算法执行的最短时间：

$$T_\infty = \min_{p \geq 1} T_P$$

- 具有无限数量处理器的计算机系统的概念通常称为：
  - 超电脑（paracomputer）

并行执行时间的评估……

- 如果使用一个处理器的算法执行时间（对于特定的计算方案）：

$$T(G) = |V|$$

- 顺序算法执行时间：

$$T_1 = \min_G T(G)$$

- 最佳顺序解的时间：

$$T_1^* = \min T_1$$

构建这样的估计是分析并行算法的一个重要方面，因为有必要评估并行性使用的效果。（是不是串行执行就够了？）

并行执行时间的评估……

- **定理 1**

  算法计算方案的最大路径长度（直径），决定了并行算法执行的最短执行时间，即

$$T_\infty(G) = d(G)$$

并行执行时间的评估……

- **定理 2**

  在算法计算方案中，让每个输入顶点到某个输出顶点都有一条路径。 另外让输入点的入度，不超过2。

  那么并行算法执行的最短时间由以下值限制：

$$T_\infty(G) = \log_2 n$$

  其中，n 是算法方案中的输入边数（依赖）。

并行执行时间的评估……

- **定理 3**

  如果使用的处理器数量减少，则算法执行时间与处理器数量的减少成比例增加，即.

$$\forall q = cp,\quad 0 < c < 1 \Rightarrow T_p \leq cT_q$$

并行执行时间的评估……

- **定理 4**

  对于使用的任意数量的处理器，以下对并行算法执行时间的**上限估计**：

$$\forall P \Rightarrow T_P < T_\infty + T_1 / p$$

并行执行时间的评估……

- **定理 5**

如果处理器较少，则算法执行时间不能超过给定处理器数的最佳计算时间的两倍以上，即

$$P < T_1 / T_\infty \Rightarrow \frac{T_1}{P} \le T_P \le 2\frac{T_1}{P}$$

最佳计算时间

并行执行时间的评估……

- **建议:**
  - 选择算法计算方案时必须使用最小”直径”的图（见定理1），
  - **并行执行的有效处理器数量**p由T1/T∞ 的值决定（见定理 5），
  - 并行算法执行时间受限于定理 4 和 5 中给出的值.

并行算法效率的特点...

- **加速比**

  与顺序执行相比，如果对 p 个处理器使用并行算法，则加速比由以下值决定：

$$S_p(n) = T_1(n) / T_P(n)$$

加速比是指并行算法的运行时间与顺序算法的运行时间之比。它衡量了在使用多个处理器时，算法运行速度的提高的倍数。

（n值用于对所求解问题的计算复杂度进行参数化，可以理解为输入问题数据的数量）

并行算法效率的特点...

- **并行效率（“性价比”）**

  并行算法在**求解问题时的处理器利用率（processor utilization）**由以下公式确定：

$$E_p(n) = T_1(n) / (pT_P(n)) = S_P(n) / p$$

并行效率是指并行算法的加速比与处理器数量之比。它衡量了在使用多个处理器时，每个处理器的利用率（性价比）

（效率值决定算法执行时间的平均值，单位还是“时间”。即，处理器实际用于计算的时间）

并行算法效率的特点...

- **Remarks:**
  - **在某些情况下可能会出现超线性加速情况 $S_p(n)>p$：**
    - “超线性加速比”（superlinear speedup），即加速比比处理器数更大的情况。
    - 超线性加速比很少出现。
    - 超线性加速比有几种可能的成因，
      - 例如，在并行计算中，不仅参与计算的处理器数量更多，不同处理器的高速缓存也被有效使用。
      - 集合的缓存足以提供计算所需的存储容量，因此算法执行时不必使用速度较慢的内存，从而存储器读写时间能够大幅降低，这就对实际计算产生了额外的加速效果。
  - **尝试针对其中提高一个指标（如：加速比或并行效率），可能会导致另一个标准的情况恶化**
    - **因为并行计算效率的指标之间可能存在相互冲突的.**

并行算法效率的特点...

- **计算的成本（P个处理器的执行时间）**

$$C_p = pT_P$$

- **成本最优的并行算法（cost-optimal parallel algorithm）**是指：其运行时间（计算开销）与最佳已知顺序算法的运行时间相匹配的并行算法。
  - 并行算法提供的加速比是最佳已知顺序算法的运行时间与并行算法的运行时间之比。

示例：求解数据集的部分和……

- 求数值序列部分和的简单问题（前缀和问题）

$$S_k = \sum_{i=1}^{k} x_i, 1 \le k \le n$$

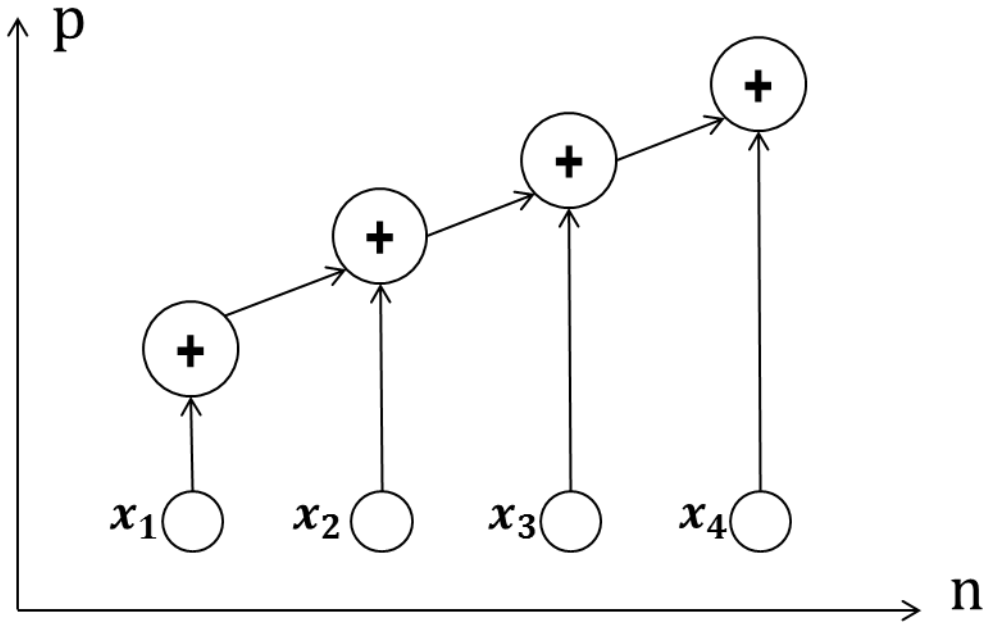
- 求解集合内总和的计算（一般归约问题的特殊情况）：

$$S = \sum_{i=1}^{n} x_i$$

示例：求解数据集的部分和……

- 对一系列值的元素进行**顺序求和**

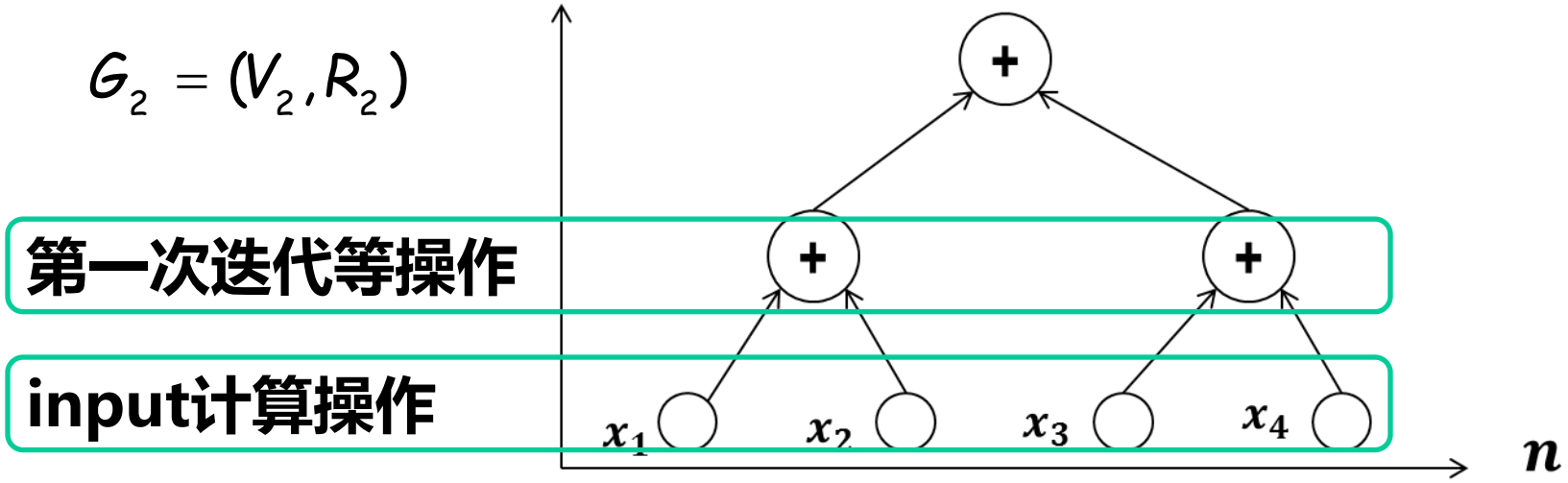
$$S=\sum_{i=1}^{n}x_i,\qquad G_1=(V_1,R_1)$$



这种“标准”顺序求和算法只允许严格串行执行，不能并行化。

示例：求解数据集的部分和……

- 级联累加的求解方案



- $V_2=\{V_{i1},\ldots,V_{ili},\ 0<=i<=k,\ 1<=l_i<=2^{-i}n\}$ 是各个求和操作
- $\{V_{01},\ldots,V_{0n}\}$ – 具有入度的input计算操作.
- $\{V_{11},\ldots,V_{1n/2}\}$ - 第一次迭代等操作，（会迭代执行多次）
- $R2=\{(V_{i-1,2j-1},V_{ij}),(V_{i-1,2j},V_{ij}),1<=i<=k,1<=l_i<=2^{-i}n\}$ - 图中的边，即各个计算操作间的依赖关系.

示例：求解数据集的部分和……

- 级联方案迭代次数：（n个数需要求和）

$$k = \log_2 n$$

- 串行执行：求和运算的总数为：

$$K_{seq} = n/2 + n/4 + \ldots + 1 = n - 1$$

- 并行执行：在级联方案的单独迭代的并行执行中，并行求和操作的总数等于：

$$K_{par} = \log_2 n$$

示例：求解数据集的部分和……

- 求和算法的级联方案的加速比和并行效率可以估计为：

加速比： $S_P = T_1 / T_P = (n-1)/\log_2 n,$

并行效率： $E_P = T_1 / pT_P = (n-1)/(p\log_2 n) = (n-1)/((n/2)\log_2 n),$

其中 p=n/2 是级联方案执行所需的处理器数量：

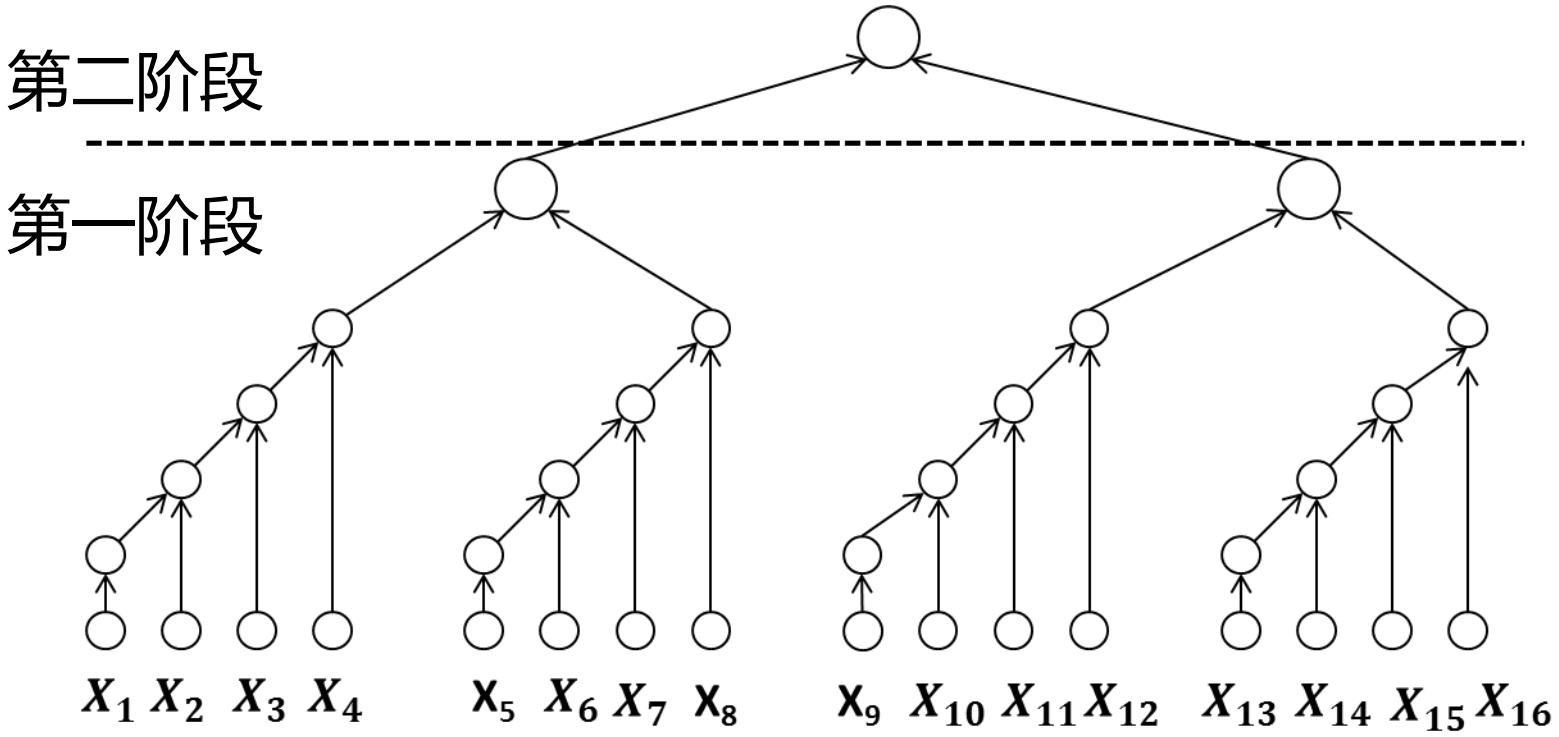
— 并行级联方案执行的时间与定理 2 中的超计算机估计一致，

—**可扩展性其实并不高：**当可求和值的数量增加时，处理器的并行效率会降低：

$$\lim E_p = 0 \quad npu \quad n \to \infty$$

示例：求解数据集的部分和……

- **修改级联方案：**
  - 所有汇总值被细分为(n/$\log_2$n)=16/4=4组，每组有$\log_2$n=4个元素。
  - 然后通过顺序求和算法为每个组计算值的总和
  - 在第二阶段，传统的级联方案用于获得的$\boldsymbol{log_2(n/log_2 n)}$ =2个独立组的求和。



示例：求解数据集的部分和……

- 如果使用$p = (n/log_2 n) = 4$ 个处理器，第一阶段会并行执行（$log_2 n$）=4个并行操作
- 第二阶段的执行需要 $log_2(n/log_2 n) \leq log_2 n$个并行运算（即，2个并行操作）
  - 使用到的处理器个数为： $p_2 = (n/log_2 n)/2 = 2$
- 这种求和方法，通过利用$(n/log_2 n)/2 = 2$个处理器，并行执行时间为:

$$T_p = 2\log_2 n$$

示例：求解数据集的部分和……

- 关于获得的估计，修改后的级联方案的加速和并行效率由以下关系定义：

$$S_P = T_1/T_P = (n-1)/2\log_2 n,$$

$$E_P = T_1/pT_P = (n-1)/(2(n/\log_2 n)\log_2 n) = (n-1)/2n$$

- **与修改前的算法相比，修改后的并行算法的加速比降低了两倍**
  - 对于新求和方法的并行效率，可以获得渐近非零估计（**并行效率能够提高**）：

$$E_P = (n-1)/2n \geq 0.25, \quad \lim E_P = 0.5 \text{ when } n \to \infty.$$

  - 修改后的级联算法是成本最优的，因为计算成本与顺序算法执行的时间成正比:

$$C_P = pT_P = (n/\log_2 n)(2\log_2 n) = 2n$$

示例：求解数据集的部分和……

- **计算所有部分和……**
  - 标量计算机上所有部分和的计算，可以通过**具有相同运算次数的常规顺序求和算法**来完成

$$T_1 = n$$

  - 在并行执行中，显式使用级联方案并不一定会带来理想的结果。

高效的并行优化需要新的面向并行的方法

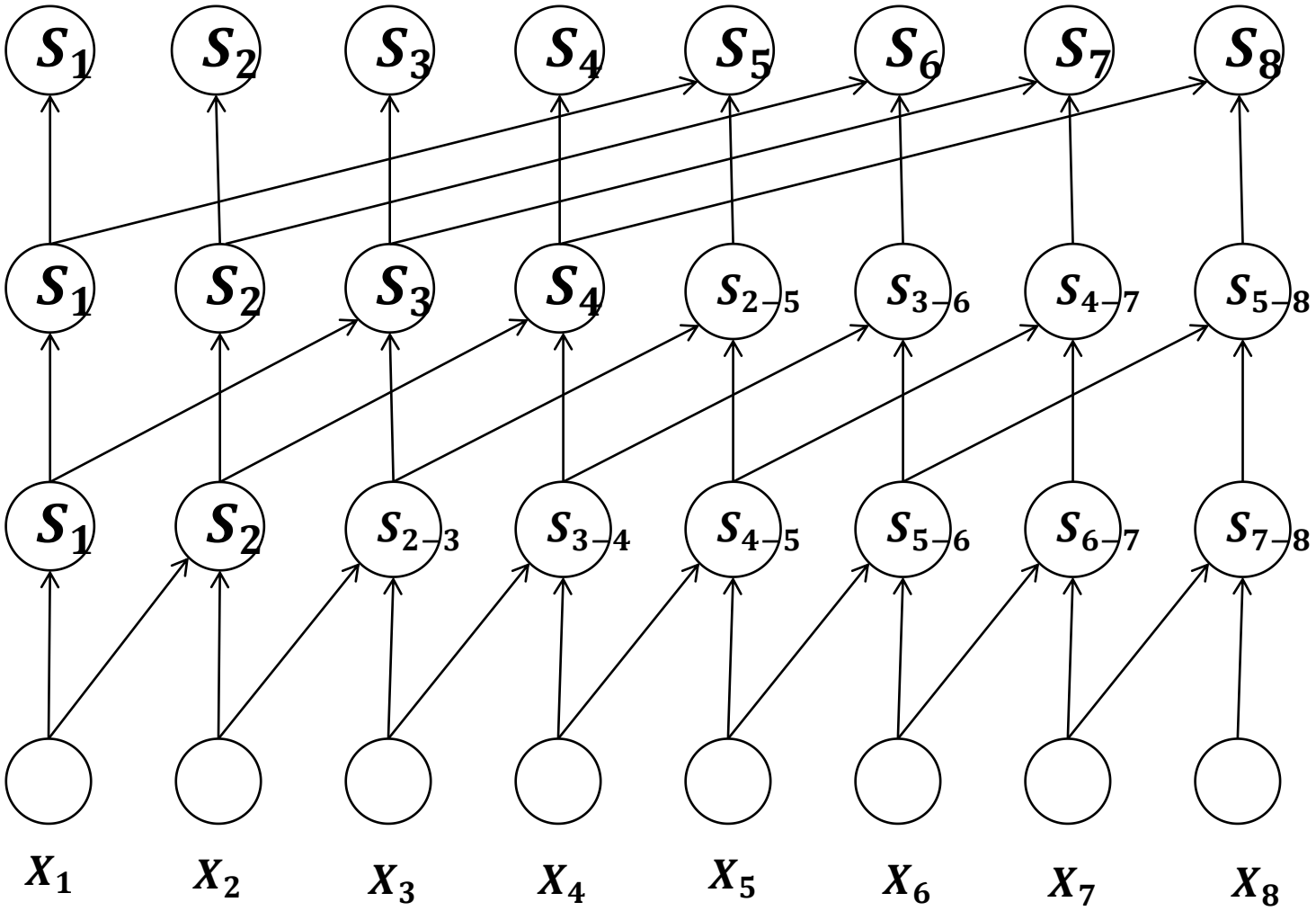
- **甚至需要打破原有思维方式，找出原本顺序编程中本不存在的方法，来解决问题**

示例：求解数据集的部分和……

- **计算所有部分和……**
  - 这个算法的并行操作个数是 $log_2 n$
    - 在计算开始之前，首先创建一个求和总值向量 S 的副本 (S=x),
    - 在每次求和迭代i，$\mathbf{1} \leq \boldsymbol{l} \leq \boldsymbol{log_2 n}$时，通过将向量S向右移动$2^{i-1}$个位置（由于移位而释放的左侧位置设置为零值）来形成辅助向量Q。算法迭代通过并行操作向量S和向量Q求和完成。

示例：求解数据集的部分和……

- 计算所有部分和……



示例：求解数据集的部分和……

- **计算所有部分和:**
  - 执行的标量操作的总数由值定义:

$$K_{seq} = n\log_2 n,$$

  - 必要的处理器数量由汇总值的数量定义:

$$p = n,$$

  - 并行算法的加速比（比前一种改进方法提高2倍）和并行效率（n趋近于无穷大时，并行效率比前一种方法更高）通过以下方式估算:

$$S_P = T_1/T_P = n/\log_2 n$$
$$E_p = T_1/pT_P = n/(p\log_2 n) = n/n\log_2 n = 1/\log_2 n$$

最大可能并行度的估计

- 并行计算效率的估计需要求解，任意计算问题的并行加速比和并行效率的最佳（最大可能）值

- 可能无法为所有耗时的计算问题提供理想的加速比和理想的并行效率
  - 理想的加速比： $S_P = P$
  - 理想的并行效率： $E_P = 1$

理想的表达式很简单，但其实很难达到！

Outline

- **性能可扩展性**
- **分析性能评测——三大定律**
  - 阿姆达尔定律（Amdahl’s law）
  - 古斯塔夫森-巴西斯定律（Gustafson-Barsis’s Law）
  - 等效率定律 （Isoefficiency Law）
- **并行计算的编程模型**
  - PRAM模型
  - BSP模型
  - LogP模型

最大可能并行度的估计

- **阿姆达尔定律（Amdahl’ s Law)**
  - 由于在执行的计算中存在顺序计算，因此可能无法实现最大可能加速 $S_P = p$。因为这些必须顺序执行的计算过程，无法通过并行方式执行。
  - 设 f 是应用数据处理算法中顺序计算的一部分
  - 如果使用 p 个处理器，计算加速比S受以下值限制:

$$S_p \le \frac{1}{f + (1-f)/p} \le S^* = \frac{1}{f}$$

最大可能并行度的估计

- **阿姆达尔定律：**
  - 如果我们选择更合适的并行优化方法，顺序计算的部分可能会大大减少
  - ***阿姆达尔效应（Amdahl’s Effect）：***
    - 对于许多问题，f=f(n) 部分是 ***n*** 的**递减函数**（串行部分，会随着n的数量增加而下降）。在这种情况下，由于增加了要解决的问题的计算复杂度，固定数量的处理器的加速比可能会增加。
    - 在这种情况下，加速比 $S_P = S_P(n)$ 是参数***n*** 的**递增函数**。

阿姆达尔定律和可扩展性

- **可扩展性**
  - 并行算法实现与处理器数量和问题规模成正比的性能提升的能力
- **强扩展性——这个计算问题能够很好地被并行加速（阿姆达尔定律）**
  - 是指在**增加处理器数量**的同时**保持问题规模不变**，算法的运行时间能够线性减少。这意味着算法能够有效地利用额外的处理器来加速计算。
  - 衡量了算法在固定问题规模下处理更多数据的能力。
- **弱扩展性——大规模化的这个计算问题能够在有限时间内求解（古斯塔夫森定律）**
  - 是指在**增加处理器数量**的同时**增加问题规模**，算法的运行时间保持不变。这意味着算法能够有效地处理更大规模的问题，而不会增加运行时间。
  - 衡量了算法在增加问题规模时保持运行时间不变的能力。

阿姆达尔定律和可扩展性

- **可扩展性**
  - 并行算法实现与处理器数量和问题规模成正比的性能提升的能力
- **Amdahl定律适用于什么情况?**
  - **保持问题规模不变**
  - *强扩展性* ($p \to \infty$, $S_p = S_\infty \to 1/f$)，增加处理器数量
  - 加速比上限由计算中顺序执行操作的计算时间决定，而**不是处理器数量p**!!!
  - 呃，这不好……为什么?
  - 完美的并行效率很难实现
- **请参阅Amdahl在网页上的两篇原始论文**
  - 单处理器方法实现大规模计算能力的有效性：G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities. Proceedings AFIPS spring joint computer conference (1967), pp. 483-485
  - 计算机体系结构与Amdahl定律：G.M. Amdahl, Computer architecture and Amdahl's law. Computer, 46 (12) (2013), pp. 38-46

古斯塔夫森定律（Gustafson）

- **可扩展性**
  - 并行算法实现与处理器数量和问题规模成正比的性能提升的能力
- **弱扩展性——大规模化的这个计算问题能够在有限时间内求解（古斯塔夫森定律）**
  - 是指在**增加处理器数量**的同时**增加问题规模**，算法的运行时间保持不变。这意味着算法能够有效地处理更大规模的问题，而不会增加运行时间。
  - 衡量了算法在增加问题规模时保持运行时间不变的能力。

古斯塔夫森定律（Gustafson）：固定时间

- **许多应用问题更注重求解精度（增加问题规模）**
  - 例子一、有限元方法：进行结构分析或者用有限差分方法求解天气预报中的计算流体力学问题，粗网格要求较少的计算量。而**细网格要求有较多的计算量，但可获得较高的精度。**
  - 例子二、天气预报中的求解四维PDE：每个物理方向（x, y, z）的格子距离减少10倍，并以同一幅度增加时间步，相当于格点增加了10,000倍，也就是**计算量增加了10,000倍**
- **John Gustafson提出了固定时间的概念，即当机器规模增大时，使问题规模也扩大的方法来获取加速比的改善**
  - 规模扩大的问题可使扩增的资源处于忙碌状态，从而可有**较高的系统利用率**

最大可能并行度的估计

- **古斯塔夫森-巴西斯定律...**

  让我们根据执行的并行计算中现有的顺序计算部分来估计**最大可能的加速比**:

$$g = \frac{\tau(n)}{\tau(n) + \pi(n)/p}$$

  其中 τ(n) 是顺序部分的时间，π(n)是并行部分的计算时间，即.

$$T_1 = \tau(n) + \pi(n), \qquad T_p = \tau(n) + \pi(n)/p$$

  考虑到引入的值g，我们可以得到:

$$\tau(n) = g \bullet (\tau(n) + \pi(n)/p), \ \pi(n) = (1-g)p \bullet (\tau(n) + \pi(n)/p),$$

  这使我们能够构建加速比估计:

$$S_p = \frac{T_1}{T_p} = \frac{\tau(n) + \pi(n)}{\tau(n) + \pi(n)/p} = \frac{(\tau(n) + \pi(n)/p)(g + (1-g)p)}{\tau(n) + \pi(n)/p}$$

最大可能并行度的估计

- **古斯塔夫森-巴西斯定律**
  - 简化后的加速比

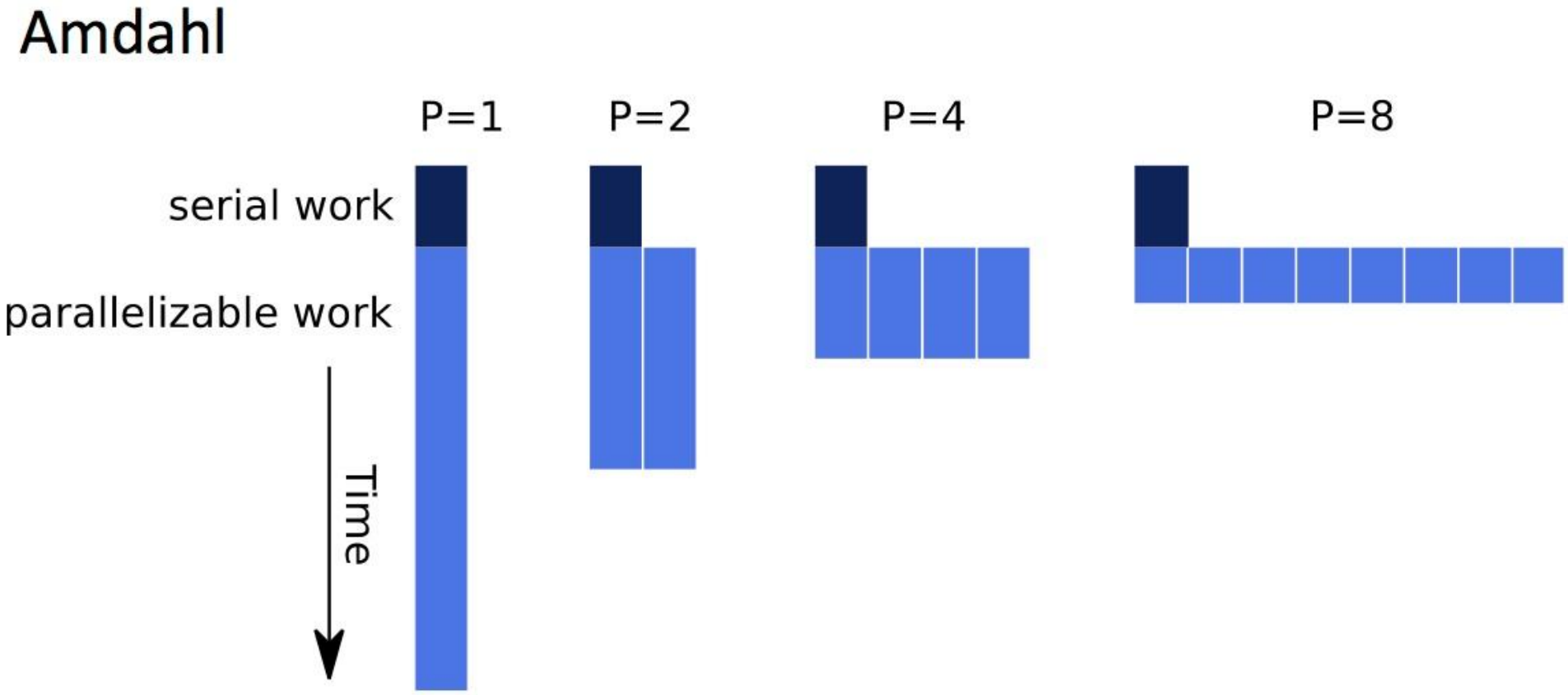
$$S_p = g + (1-g)p = p + (1-p)g$$

  - 考虑到这些情况，根据古斯塔夫森-巴西斯定律获得的加速比估计也被称为**可扩展性加速比（scaled speedup）**。
  - 事实上，这个特征可以显示如果问题的复杂度增加，如何并行计算的效率如何变化。---->弱可扩展性

古斯塔夫森-巴西斯定律和可扩展性

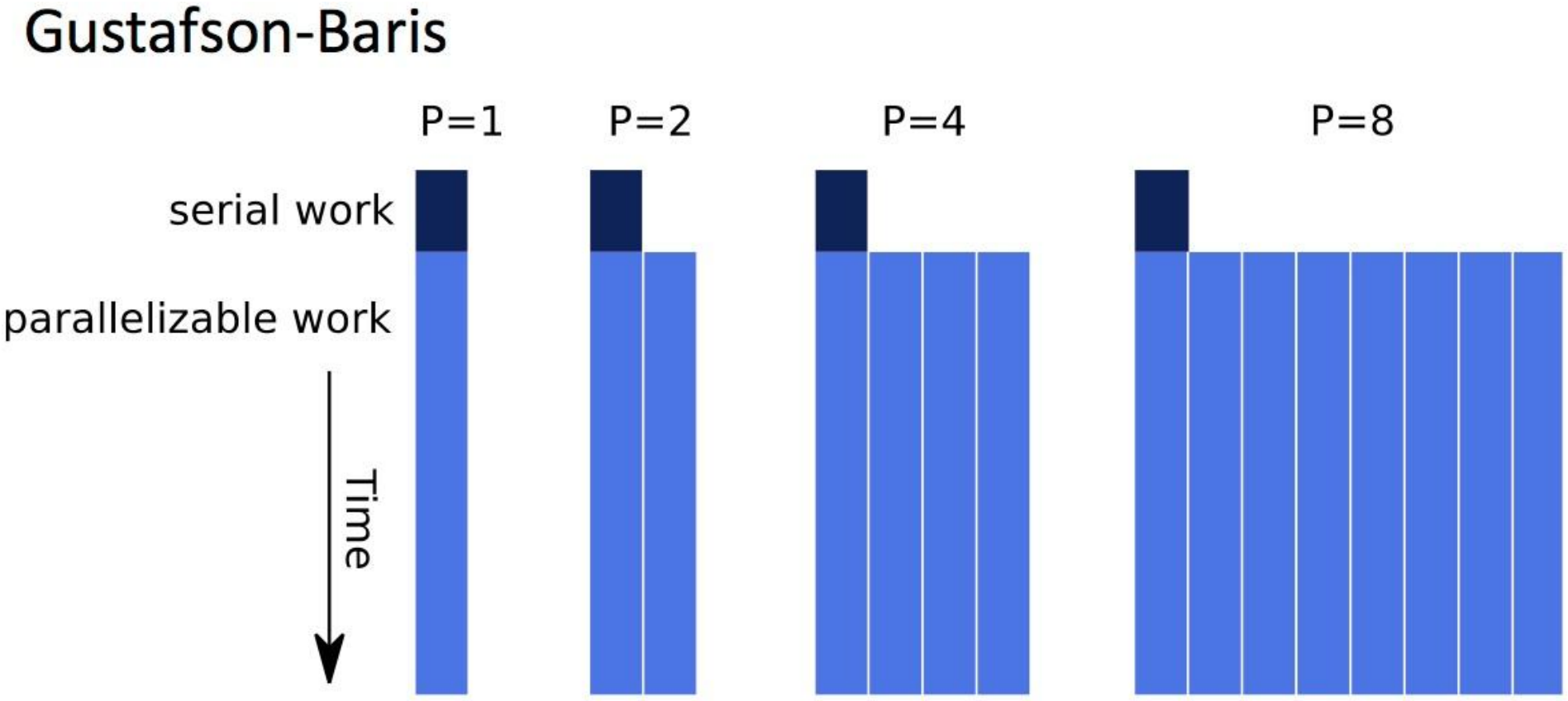
- **可扩展性**
  - 并行算法实现与处理器数量和问题大小成比例的性能增益的能力
- **古斯塔夫森定律何时适用?**
  - 当问题的大小随着处理器数量的增加而增加时
  - **弱可扩展性** $S_p = g + (1-g)p = p + (1-p)g$
  - 加速的能力与处理器数量相关！！！
  - 随着问题规模的扩大，可以保持或提高并行效率
- **在网页上查看 Gustafson 的原始论文**
  - http://johngustafson.net/glaw.html

阿姆达尔 (Amdahl) vs 古斯塔夫森-巴西斯 (Gustafson-Baris)



阿姆达尔（Amdahl）：强可扩展性，问题规模有限

Amdahl versus Gustafson-Baris



古斯塔夫森-巴西斯（Gustafson）：弱可扩展性，问题规模等比例扩展

并行计算可扩展性分析

- 一个并行算法能称为**可扩展性算法**的条件：
  - 条件一：随使用的处理器数量增加，能够提供**等比例增加的并行加速比**
  - 条件二：保持的**并行效率恒定**，即各个处理器利用率恒定，不会随着使用处理器个数增长而降低。

并行计算可扩展性分析

由于需要组织处理器之间的交互，因此会产生额外的**非计算开销，T_o**（Overhead）。需要执行一些额外的操作，例如：同步操作等。

$$T_0 = pT_P - T_1$$

- 结合非计算开销T_o，并行问题求解的总体时间和相应的加速比会有新的表达式:

$$T_P = \frac{T_1 + T_0}{P}, \quad S_P = \frac{T_1}{T_P} = \frac{pT_1}{T_1 + T_0}.$$

同样的，表示处理器利用率的并行效率可以表示为：

$$E_P = \frac{S_P}{P} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0 / T_1}.$$

并行计算可扩展性分析

- 问题规模不变：对于固定问题复杂度的并行程序（即：$T_1 = const$），由于总开销T_0增长，并行效率通常会随着处理器数量的增加而降低
- 计算资源不变：如果处理器数量固定，处理器利用效率可能会随着复杂度T_1的增加而提高
- 如果处理器的数量增加，则在大多数情况下，可以通过提高求解问题复杂度，来增加并行计算效率，提高处理器资源利用率

$$T_P = \frac{T_1 + T_0}{P}, \quad S_P = \frac{T_1}{T_P} = \frac{pT_1}{T_1 + T_0}.$$

$$E_P = \frac{S_P}{P} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0/T_1}.$$

并行计算可扩展性分析

- **令 E=const 为所执行计算的理想并行计算效率，即为常数。使用我们可以获得的并行效率方程式如下：**

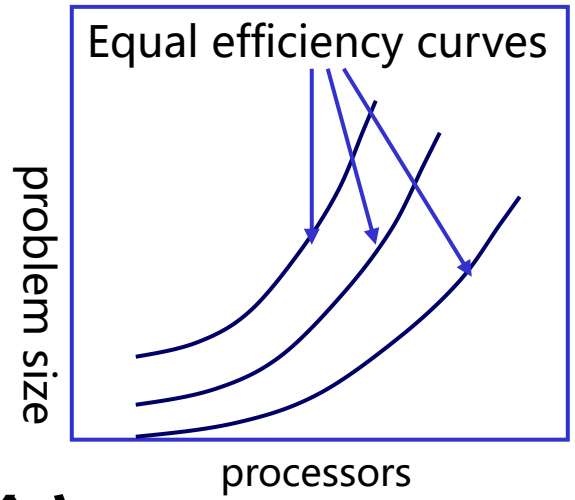
$$\frac{T_0}{T_1} = \frac{1-E}{E}, \text{ or } T_1 = KT_0, K = E/(1-E)$$

- 问题复杂度n与处理器数量p之间的依赖关系 n=F(p) 称为**等效率函数（isoefficiency function）**
  - 等效性函数是并行性能度量之一，它衡量了问题规模与维持系统并行效率所需的处理器数量之间的关系，让我们能够确定处理器数量、速度和互连网络通信带宽方面的可扩展性。
  - 等效性函数决定了**并行系统保持恒定效率的容易程度**，从而实现与处理器数量成比例增长的加速

等效率（Isoefficiency）

- **目标是量化可扩展性**
- **问题规模需要增加到多少，才能在更大规模的计算系统上保持相同的并行效率?**
- **并行效率**
  - $T_1 / (p * T_p)$
  - T_p = 计算时间 + 通信时间 + 空闲时间
- **等效率**
  - 等效率曲线方程
  - 如果一个计算问题无法绘制等效率曲线方程
    - 结合等效率定义，说明这个计算问题是不可扩展的

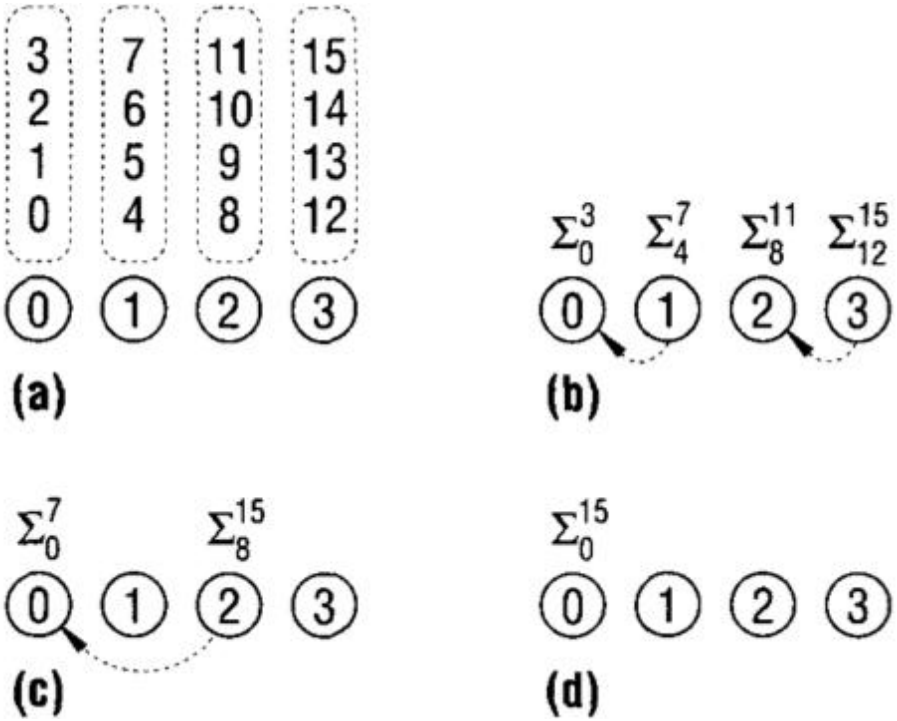
等效率曲线方程



- **查看 Kumar 关于等效率方程相关的原始论文**
  - 测量并行算法和架构的可扩展性：Ananth Y. Grama, Anshul Gupta, and Vipin Kumar, Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures, IPDT, 1993

通过增加问题规模n来提高可扩展性

- 并行系统的可扩展性，是衡量在求解问题过程中，其使用更多处理器来提高速度的能力
- 在带分区的 p 个处理器上添加问题规模 n ：



示例：求和问题的四个步骤

通过增加问题规模n来提高可扩展性

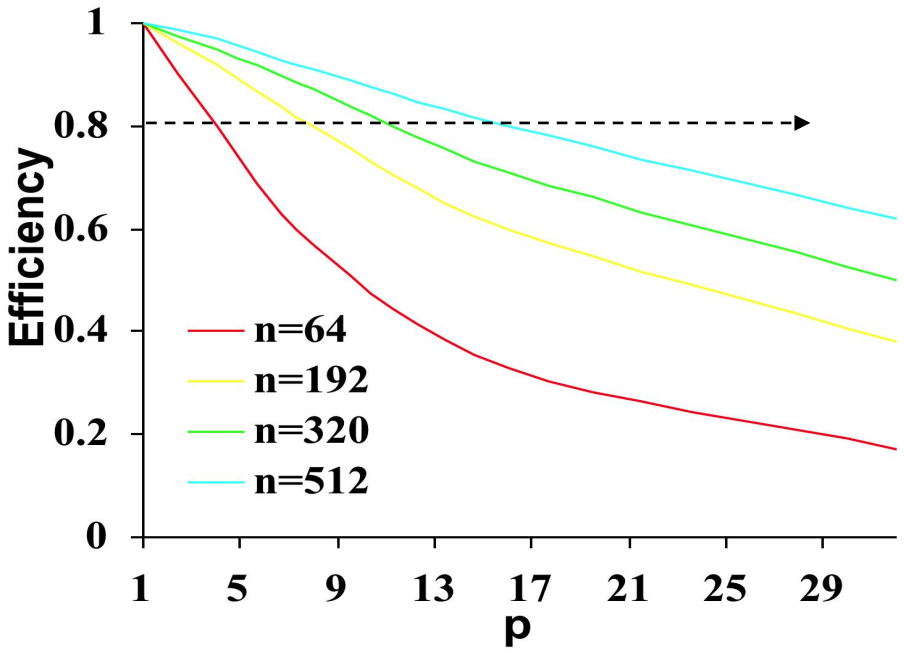
- 并行系统的可扩展性，是衡量在求解问题过程中，其使用更多处理器来提高速度的能力
- 在带分区的 p 个处理器上添加问题规模**n**:

$$T_{par} = \frac{n}{p} - 1 + 2\log p$$

$$Speedup = \frac{n-1}{\frac{n}{p} - 1 + 2\log p} \approx \frac{n}{\frac{n}{p} + 2\log p}$$

$$Efficiency = \frac{S}{p} = \frac{n}{n + 2p\log p}$$

相关度量参数计算公式



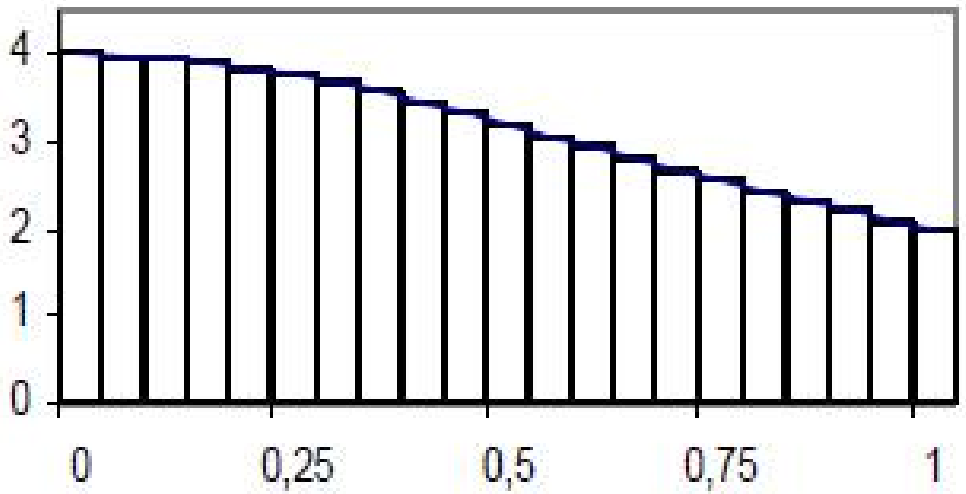
n表示求和数量的个数

示例：常数 π 的计算……

- 常数 π 的值可以通过求解积分进行计算

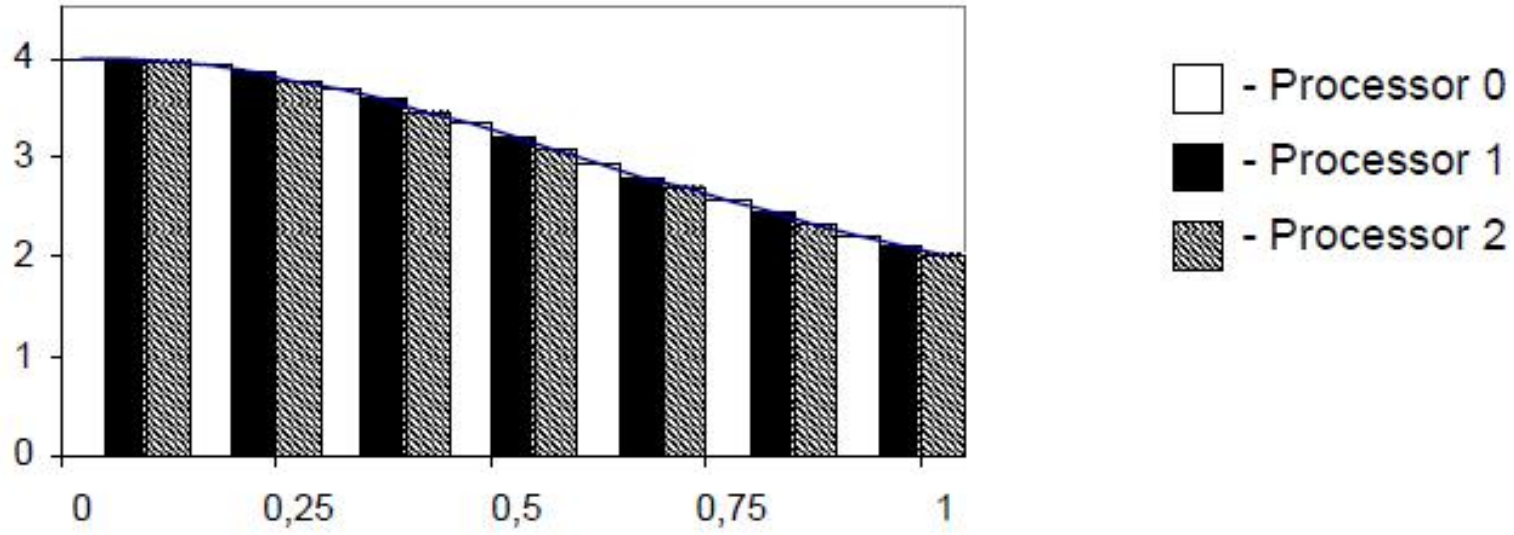
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

- 要计算此积分，可以使用矩形法（the method of rectangles）进行数值积分



示例：常数 π 的计算……

- 下图中的循环调度方案，可用于在多个处理器之间分配计算。
- 最后，需要对各个处理器上所计算出来的部分和，最终再进行一次求和。
  - 最终进行一次求和，无法与各个处理器的“部分和”计算过程并行执行。



示例：常数 π 的计算……

- **效率分析…**
  - **n** – [0,1]之间的分段数，共n段
  - 算法的计算复杂度：

$$W = T_1 = 6n,$$

  - 每个处理器要处理的网格节点数

$$m = \lceil n/p \rceil \le n/p + 1,$$

  - 每个处理器要处理的计算复杂度

$$W_p = 6m = 6n/p + 6.$$

示例：常数 π 的计算……

- **效率分析…**
  - 并行算法**执行时间**

$$T_P = 6n/p + 6 + \log_2 p,$$

  - **加速比**

$$S_P = T_1/T_P = 6n/(6n/p + 6 + \log_2 p),$$

  - **并行效率**

$$E_P = 6n/(6n + 6p + p\log_2 p),$$

  - 等效率函数

$$W = K(pT_P - W) = K(6p + p\log_2 p)$$

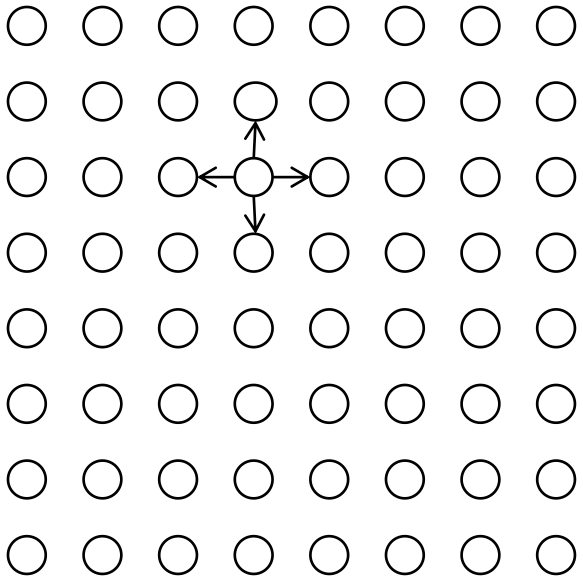
$$\Rightarrow n = [K(6p + p\log_2 p)]/6, \quad (K = E/(1-E))$$

$$Ex.\ E = 0.5 \ \text{when}\ p = 8 \rightarrow n = 12$$

$$\text{when}\ p = 64 \rightarrow n = 128$$

示例：有限差分法...

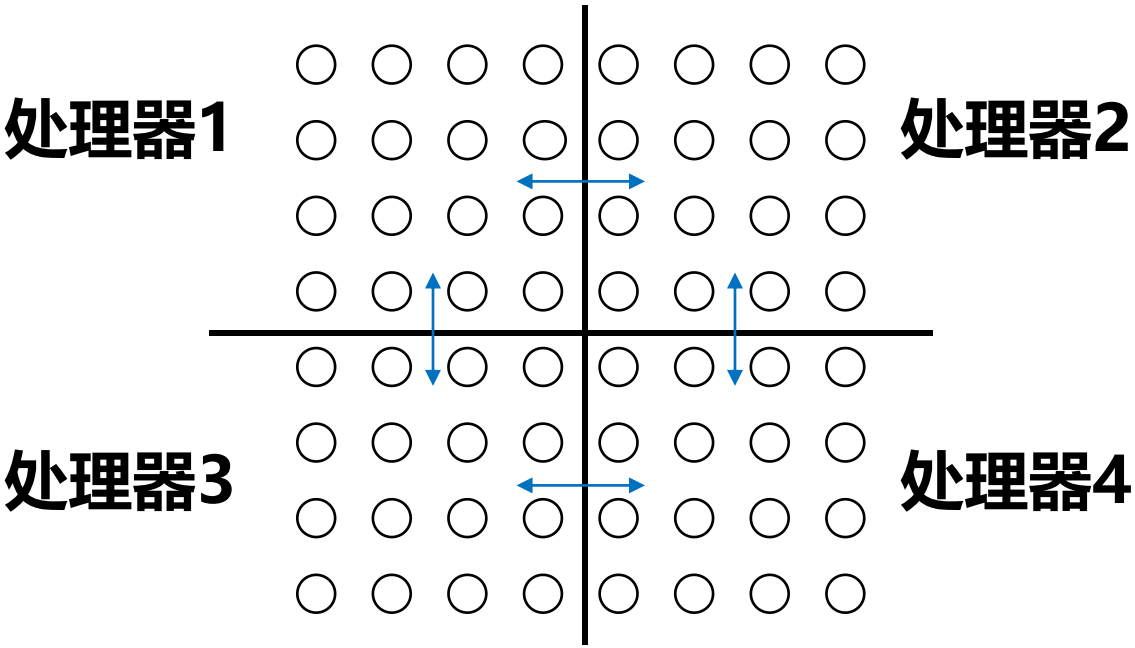
- 有限差分法被广泛应用于偏微分方程的数值求解之中
- 求解偏微分方程时，网格计算是一种常用的数值方法
  - 网格计算通过将求解域离散化为许多小的网格单元，然后在每个单元上应用数值方法来近似求解偏微分方程。
  - 这种方法可以将连续的偏微分方程转化为离散的代数方程组，从而可以使用计算机进行求解。
- 网格计算的精度取决于网格大小和形状
  - 网格越细，精度越高。
  - 但是，随着网格变得更加精细，计算量也会增加。
  - 在实际应用中需要在精度和计算量之间进行权衡。



在网格计算中，每个节点的计算结果通常需要与相邻节点进行同步

示例：有限差分法...

- 每个处理器对网格的$(n/\sqrt{p}) * (n/\sqrt{p})$个节点的矩形子区域进行计算，
- 每次迭代后都需要执行同步，注意体现在邻近的处理器之间进行结果同步



示例：有限差分法...

- **效率分析**
  - $T_1 = W = 6n^2M$ (M – 迭代的次数),
  - $T_P = 6M + Mlog_2p$,
  - **加速比**

$$S_P = T_1/T_P = 6n^2/(6n^2/p + \log_2 p),$$

  - 等效率函数

$$W = K(pT_P - W) = K(p\log_2 p),$$

$$\Rightarrow n^2 = [K(p\log_2 p)]/6,\ (K = E/(1-E))$$

有限差分法比矩形法更具可扩展性

数据通信方法

处理器之间传输数据所需的时间，就是并行算法执行过程中的**通信开销** ***(communication overheads)***

并行算法的数据通信时间，由以下各分项开销构成:

- ***启动开销*** (t_s) 即：start。
  - 表征准备传输消息的持续时间，网络中路由的搜索等.
  - 这个时间通常被称为通信操作的延迟 *(latency)*
- ***每跳的传输时间*** (t_h) 即hop。
  - 两个相邻处理器之间传输数据的时间中， 对数据包头的处理过程，包含对系统信息、错误检测等处理
- ***每字节传输时间*** (t_b)即byte。
  - 沿着数据通信通道传输一个字节的时间; 此传输的持续时间由通信信道带宽 *(bandwidth)* 来决定.

通信操作的复杂度估算

- 在实际应用中，为了使用上述模型，有必要估计所使用通信操作开销的参数值。
- 在这方面，有时使用更简单的方法来计算数据通信的时间开销是合理的。
- 其中最著名的方案之一是霍克尼模型(Hockney model)，它根据方程来估计两个处理器之间通信的持续时间:

$$t_{comm}(m) = t_s + mt_b$$

通信操作的复杂性估计

- 还应注意的是，上述针对通信操作的时间开销分析已经可以初步对通信操作的特征进行描述了。
- 此外，该模型是所有模型中最简单的模型。

- 我们将在所有后续部分中使用霍克尼模型来估算通信时间； 模型的记录格式为：

$$t_{comm}(m) = \alpha + m/\beta,$$

- 其中α是数据通信网络的延迟 (即 $a = t_s$), β 是网络带宽 (即： $\beta = R = 1/t_b$).

总结…

- 本节将计算模型描述为“操作-操作数”图，可用于描述所选问题求解算法中，所存在的信息依赖关系。
- 超计算机(paracomputer)作为具有”无限”数量处理器的并行系统的概念，被认为是为了简化对并行算法进行评价和估计的理论模型。
- 为了估计并行计算方法的效率，讨论了诸如加速比（speedup）、并行效率（efficiency）、开销成本（cost ）和等效率（isoefficiency ）等广泛使用的评价标准

Outline

- **性能可扩展性**
- **分析性能评测——三大定律**
  - 阿姆达尔定律（Amdahl’ s law）
  - 古斯塔夫森-巴西斯定律（Gustafson-Barsis’ s Law）
  - 等效率定律 （Isoefficiency Law）
- **并行计算的编程模型**
  - PRAM模型
  - BSP模型
  - LogP模型

讲新东西前，咱先热热身

- 为了演示模型的应用和并行算法分析的方法，前面的课程内容，已经介绍了求部分和的计算问题、数值积分问题和有限差分法的示例
- 为了获得效率特性的最大可能值的估计，讨论了 阿姆达尔(Amdahl)定律和古斯塔夫森-巴西斯(Gustafson-Barsis) 定律
- 最后给出等效率函数的概念

并行计算（系统）模型

- **PRAM模型（并行内存）**
  - 全称：parallel RAM
  - 最基本的并行计算系统
- **BSP模型（批量同步并行模型）**
  - 全称：Bulk Synchronous Parallel
  - 将计算与通信进行隔离
- **LogP模型**
  - 用于研究分布式存储系统
  - 专注于互连网络
- **Roofline模型**
  - 分析资源的“供给”和计算的“速度”

并行内存：PRAM模型

- **并行随机存取机**
  - 全称：Parallel Random Access Machine (PRAM)
- **是一种基于共享内存的多处理器模型**
- **假设一：无限数量的处理器**
  - 无限本地内存
  - 每个处理器都有自己的 ID
- **假设二：无限容量的共享内存**
  - 输入/输出的数据，都统一放置在共享内存中
  - 存储单元可以存储任意大的整数
- **假设三：每条指令占用单位时间**
  - 每条指令需要跨处理器进行同步 (SIMD)

对PRAM 复杂性的度量

- **对于每个单独的处理器**
  - 时间开销：执行的指令数
  - 空间开销：访问的内存单元数
- **PRAM计算机（服务器）**
  - 时间开销: 运行时间最长的处理器所用的时间
  - 硬件开销: 需要占用的处理器的数量
- **技术问题**
  - 处理器是如何被激活使用的?
  - 共享内存是如何被访问的?

激活处理器

- **P_0（0号处理器）将处理器的个数数p，存入指定的共享内存单元中**
  - 每个可被调用的处理器 P_i, 其中$i < p$, 开始被占用，执行计算任务
  - 在$O(1)$ 时间激活
  - 在 P_0 停止工作时，所有处理也都会被终止

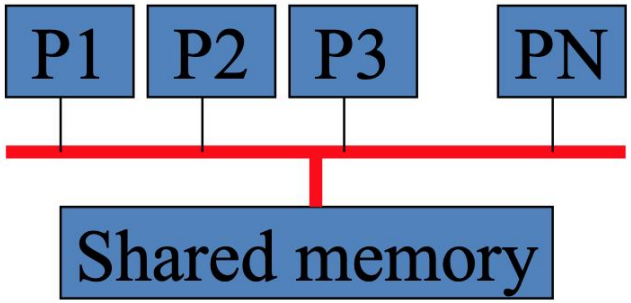
- **活动处理器通过 FORK 指令显式地激活其他处理器**
  - 以树状结构，逐层激活多个处理器
  - 激活时间：O(log p)

PRAM 是一个理论（实际不可行）模型

- 处理器和内存之间的互连网络需要无限高的网络带宽。
- 互连网络上的消息路由所需要花费的时间，需要与网络的规模大小成正比例。
- 算法设计人员可以理想化地忽略通信延迟问题，只关注并行计算性能。
- 在有边际范围的网络上，可以基于PRAM模型模拟任意并行算法
- 为PRAM模型设计通用的并行算法，并在实际网络上进行验证。

PRAM Model

- 一组同步处理器，它们与全局共享内存单元通信
- 一组有编号的处理器（Pi）
- 一组索引内存单元(M[i])
- 每个处理器Pi都有自己的无限的本地内存（寄存器）并知道它的索引（rank）。
- 每个处理器可以在单位时间内访问任何共享内存单元。
- PRAM算法的输入和输出由N个不同的数据单元组成。
- PRAM指令包括3个同步步骤：读取（获取输入数据），计算，写入（将数据保存回共享内存单元）。
- 通过写入和读取内存单元来实现数据交换。



PRAM Model

- **算法复杂度**
  - 时间开销 = P0 计算所用的时间
  - 空间开销 = 访问的内存单元数
- **最简化的一种系统模型**
  - 在N个处理器上每个周期的操作数最多为N。
  - 所有访问都在一个时间单位内实现。
  - 通过抽象所有通信或同步开销，降低复杂性和正确性开销。

PRAM 以最直观地和最简化的程序员视角，来看待并行计算机。

- **它忽略了较低级别的体系结构约束和细节**
- **例如：内存访问争用和开销、同步开销、互连网络吞吐量、连接性、速度限制和链路带宽等。**

PRAM Model具有多种模式

- ***独占读独占写：EREW* (Exclusive Read Exclusive Write)**
  - 没有对同一内存位置的并发读/写
- ***并发读独占写：CREW* (Concurrent Read Exclusive Write)**
  - 多个处理器可以在同一指令中，对同一个全局内存位置进行读取
- ***独占读并发写：ERCW* (Exclusive Read Concurrent Write)**
  - 允许并发地写入同一个全局内存位置
- ***并发读并发写：CRCW* (Concurrent Read Concurrent Write)**
  - 允许并发读写
- **各种模式的性能差别：CRCW > (ERCW,CREW) > EREW**

基于并发读并发写(CRCW)的PRAM Models

- **共有四种细分模型，如下：**
  - 共同性写入（COMMON）:同时写入同一地址的所有处理器必须写入相同的值
  - 武断性写入（ARBITRARY）:如果多个处理器同时写入该地址，则随机选择一个处理器并将其值写入寄存器
  - 按照优先级写入（PRIORITY）：如果多个处理器同时写入该地址，则具有最高优先级的处理器成功将其值写入内存位置
  - 合并式写入（COMBINING）：存储的值是写入值的某种组合，例如 sum、min 或 max
- **最常用的 COMMON-CRCW （共同性写入）模型**

PRAM 算法的复杂性

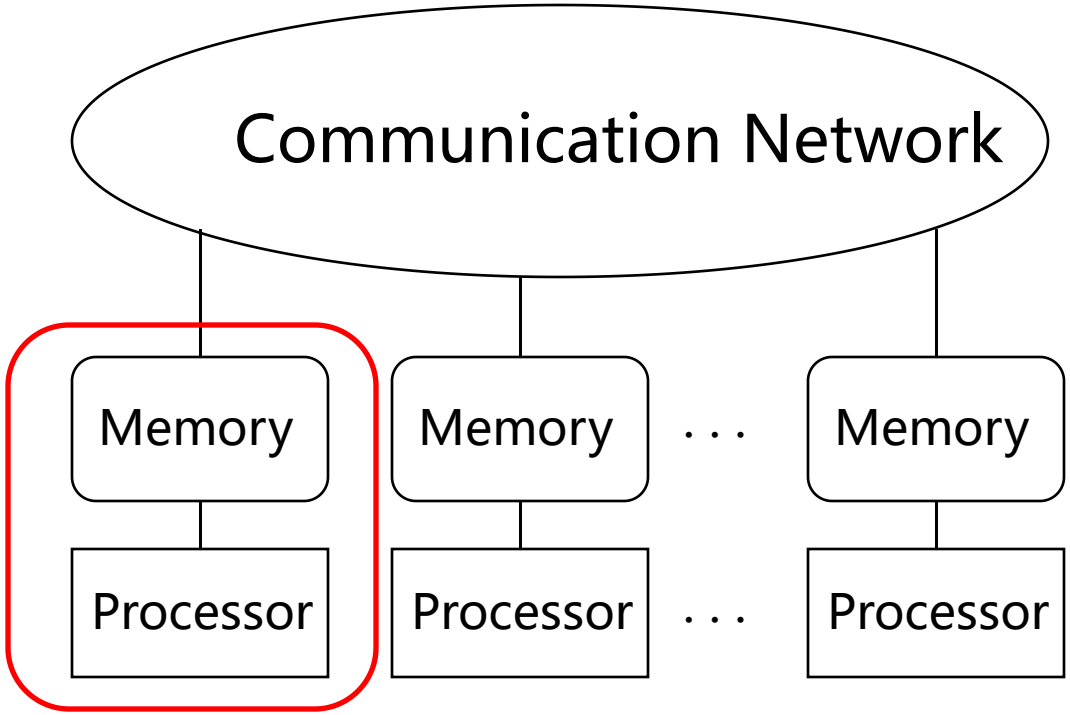
Problem / Model	独占读独占写 EREW（最坏）	并发读并发写 CRCW（最好）
Search	O(log n)	O(1)
List Ranking	O(log n)	O(log n)
Prefix	O(log n)	O(log n)
Tree Ranking	O(log n)	O(log n)
Finding Minimum	O(log n)	O(1)

BSP 模型概述

- **批量同步并行**
  - 并行程序设计模型
  - 由 莱斯利·瓦利安特（Leslie Valiant ）在哈佛大学发明
  - 实现了对并行算法的性能预测
  - 适用于SPMD（单程序多数据）
- **支持直接内存访问（DMA，direct memory access ）和消息传递语义（message passing semantics）**
  - BSPlib 是由牛津大学实现的 BSP 库

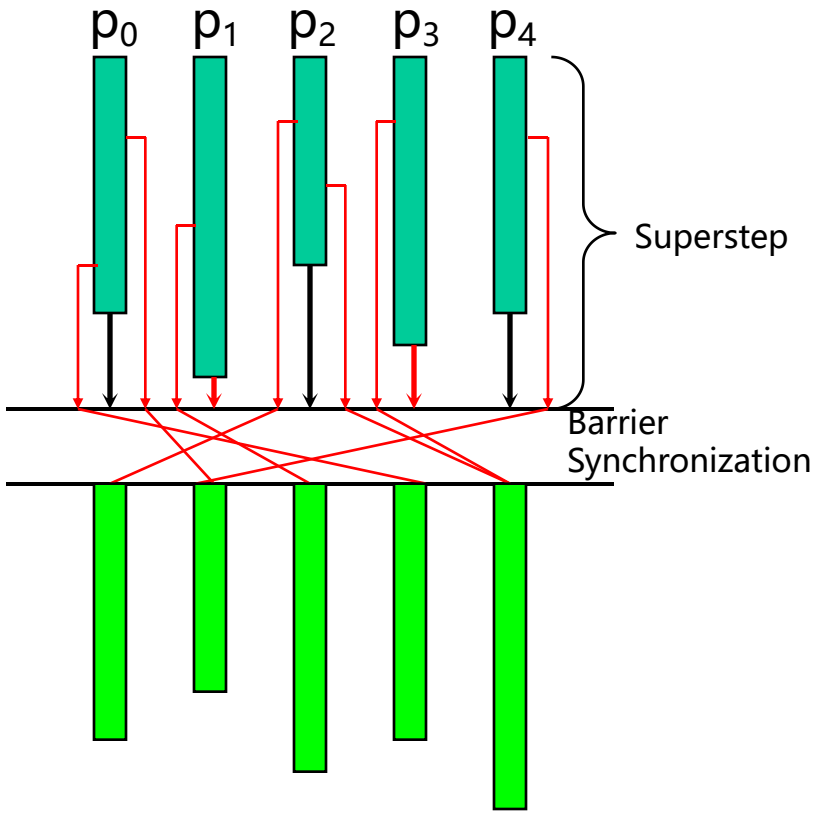
BSP计算机的组成部分

- 一组“处理器-内存”对儿
- 通信点对点（point-to-point）网络
  - 两个节点之间直接进行通信，不经过任何中间节点
- 对所有的处理器具有高效的屏障同步机制（barrier synchronization）



BSP 的超步（Supersteps）

- BSP 计算由一系列超步组成
- 在每个超步中，进程使用本地可用数据执行**计算**，并向远程处理器发出**通信**请求
- 进程在超步结束时进行**同步**，此时所有的通信操作都已完成



BSP 性能模型参数

- **p = 处理器数量**
- **l = 屏障延迟，实现barrier同步的时间开销**
  - l 以秒为单位进行测量
- **g = 每条同步信息的通信时间开销**
  - g 以秒为单位进行测量
- **s = 处理器速度**
  - s 以 FLOPS 为单位进行测量
- **任何处理器在单个超步中最多发送和接收 h 条消息（称为 h-关系通信）**
- **超步的时间开销 = 任何一个处理器执行的最大本地计算操作的时间 + g*h + l**

同步开销（Synchronization Cost）

- **同步成本用 l 表示，一般根据经验确定**
- **随着计算资源规模的增加，同步成为主要瓶颈**
  - 删除它们可能会引入死锁或活锁
  - 模型复杂度升高

计算开销

$$T_{superstep} = \max_{i=1}^{p}(w_i) + g * \max_{i=1}^{p}(h_i) + l$$
$$= w + g * h + l$$

Where:

w = 最大计算时间
g = 1/(网络带宽)
h = 最大消息数
l = 同步时间

$$T_{total} = \sum_{s=1}^{S} T_{superstep}$$

BSP - 举例：拉普拉斯方程（Laplace）

- 拉普拉斯方程是一个偏微分方程，它描述了U在空间中的变化情况。
- 在拉普拉斯方程中，U通常表示一个标量函数，它描述了某个物理量在空间中的分布情况。这个物理量可以是温度、压力、电势等，具体取决于问题的背景。
- 拉普拉斯方程的一般形式为：

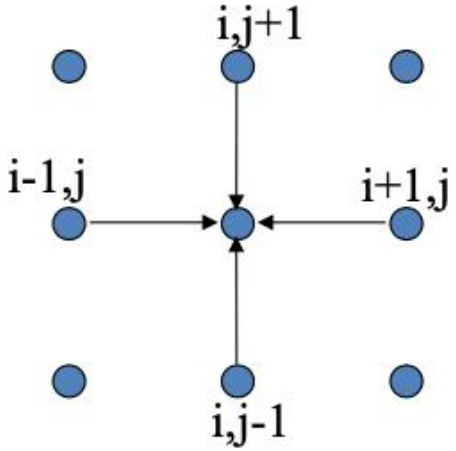
$$\nabla^2 U = 0$$

- 其中，∇^2表示拉普拉斯算子，它是一个二阶微分算子。
- U表示一个标量函数，它描述了某个物理量在空间中的分布情况。

BSP - 举例：拉普拉斯方程（Laplace）

- 求解拉普拉斯（Laplace）方程的数值解

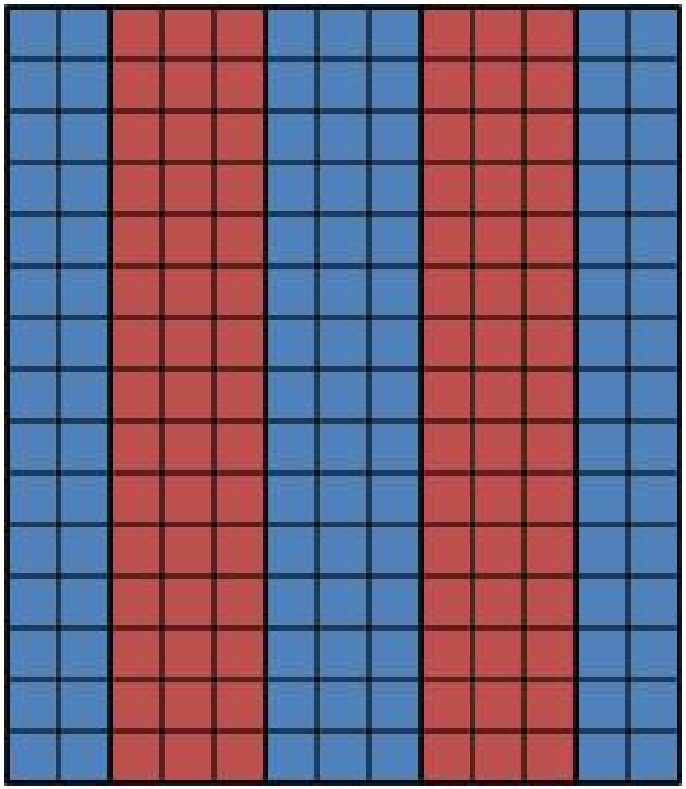
$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$




```
for j = 1 to jmax
  for i = 1 to imax
    Unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                       +   U(i,j-1) + U(i,j+1))
  end for
end for
```

BSP - 举例：拉普拉斯方程（Laplace）

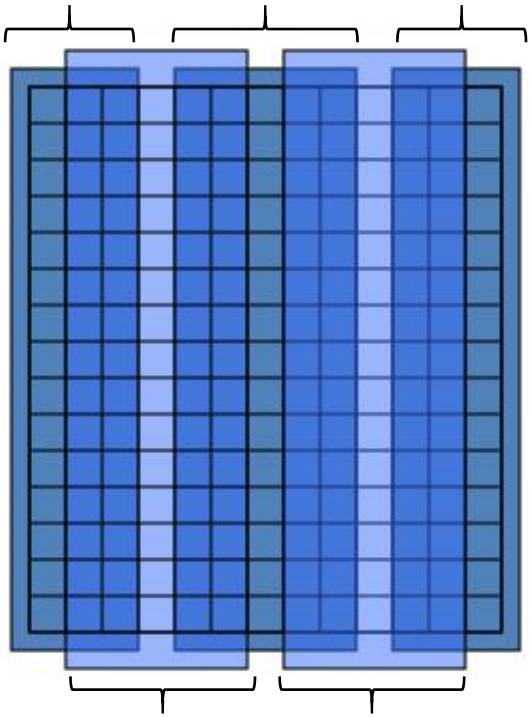
- 使其并行的方法就是对数据进行分区



BSP - 举例：拉普拉斯方程（Laplace）

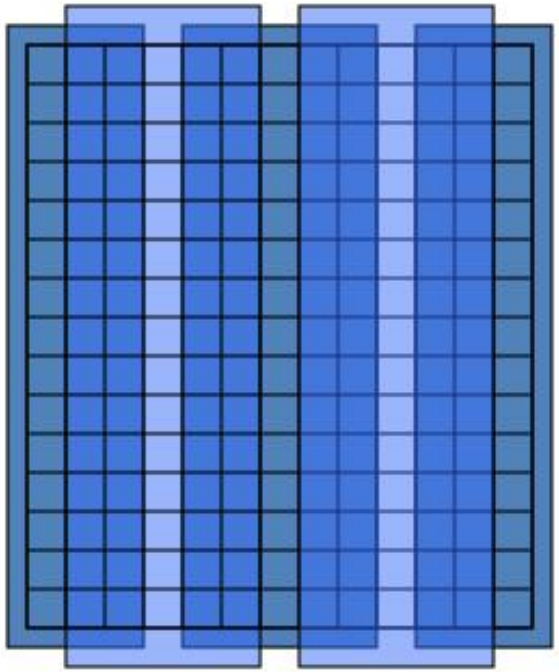
- 使其并行的方法是对数据进行分区
  - **重叠数据边界**允许每个超步无需通信即可进行计算
  - 在通信步骤中，每个处理器更新远程处理器上的相应**列**。

分区1　分区3　分区5



分区2　分区4

BSP - 举例：拉普拉斯方程（Laplace）



```
for j = 1 to jmax
  for i = 1 to imax
    unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                       +   U(i,j-1) + U(i,j+1))
  end for
end for
if me not 0 then
  bsp_put( to the left )
endif
if me not NPROCS – 1 then
  bsp_put( to the right )
Endif
bsp_sync()
```

BSP - 举例：拉普拉斯方程（Laplace）

$$T_{superstep} = w + g * h + l$$

h = 任何处理器在单个超步中最多发送和接收 h 条消息

= 左边的一列 N 个值 + 右边的一列 N 个值

= 2 N（忽略逆向通信！）

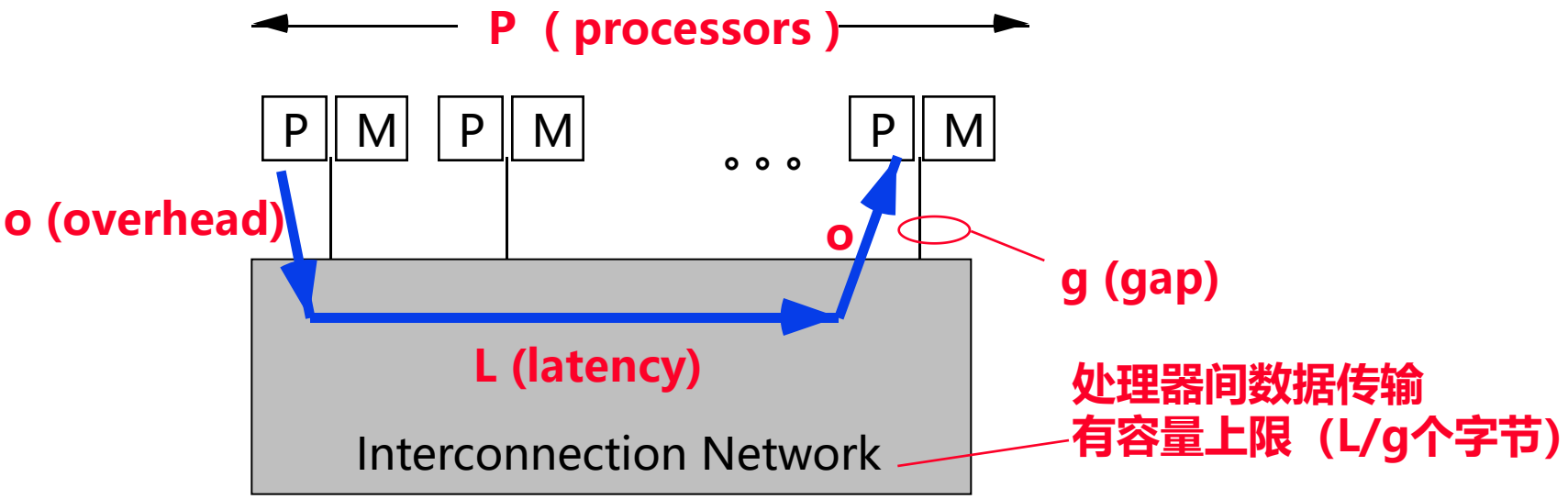
W = 最大计算时间=4*N*N/p

$$T_{superstep} = 4 * \frac{N^2}{p} + 2 * g * N + l$$

The LogP Model (Culler, Berkeley)

- LogP是一种用于评估并行计算机系统通信性能的模型。它由四个参数组成：
  - L（latency延迟）：表示从一个处理器发送消息到另一个处理器接收到该消息所需的最短时间。
  - o（overhead带宽带来的限制）：表示每个处理器在发送或接收消息时的开销。
  - g（gap间隔）：表示连续两次消息发送之间的最短时间间隔。
  - P（processors 处理器数量）：表示系统中处理器的数量。
- LogP模型通过这四个参数来描述并行计算机系统中**通信**的性能特征。
  - 用来评估并行算法在给定系统上的通信开销，从而为算法设计和优化提供参考。

LogP



- Latency 在模块之间发送（小）消息的延迟
- overhead 处理器在发送或接收消息的时间开销
- gap 连续发送或接收之间的间隔（1/BW)
- Processors 处理器

LogP

- 消息从处理器 A 到处理器 B 的总时间为：
  - L+2*o
- 没有针对应用程序的模型
- 我们可以使用与 BSP 相同的方法来描述应用程序：超步（supersteps）

$$T_{\text{superstep}} = w + h*(L + 2o) + l$$

w = 最大计算时间
h = 最大消息数
L = 在模块之间发送（小）消息的时间
o =处理器发送或接收消息的时间
l = 同步时间

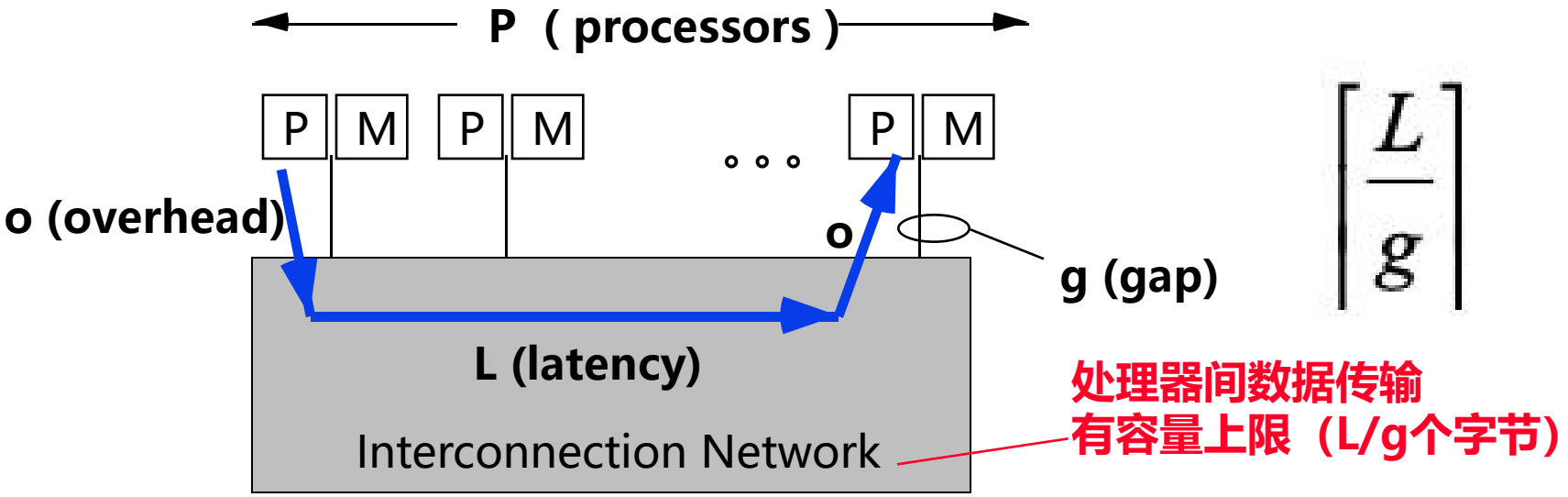
LogP

- **难道处理器的个数P，不影响超步的计算时间?**
- **当处理器数量不固定时，他们都是p的函数:**
  - 计算变化的时间 w(p)
  - 消息数量变化 h(p)
  - 同步时间变化 l(p)

$$T_{\text{superstep}} = w + h*(L + 2o) + l$$

LogP

- **有趣的概念：网络容量的限制**
  - 任何传输超过一定数据量的同步或通信，都会使处理器停止（宕机）



- **这个模型没有解决单个消息的数据量大小的问题，即使是最坏的假设是所有消息都是“小”的。**

LogP 的建模思路

- **Think about:**
  - 将 N 个词映射到 P 个处理器上
  - 处理器内的计算
    - 它的开销与平衡
  - 处理器之间的通信
    - 它的开销与平衡
- LogP模型通过这四个参数来描述并行计算机系统中通信的性能特征
  - 用来评估并行算法在给定系统上的通信开销，从而为算法设计和优化提供参考
  - 描述处理器和网络的性能
  - 不要去想网络中的细节

总结

- **性能可扩展性**
  - **并行算法实现的性能能够随：处理器数量和问题规模增长，同比例增长提升的能力**
- **分析性能评测——三大定律**
  - 阿姆达尔定律（Amdahl’s law）
  - 古斯塔夫森-巴西斯定律（Gustafson-Barsis’s Law）
  - 等效率定律 （Isoefficiency Law）
- **并行计算的编程模型**
  - PRAM模型
  - BSP模型
  - LogP模型

谢谢