并行处理

L04: 性能模型与评价方法 (Performance Model and Evaluation)

——算法部分

叶笑春
中国科学院计算技术研究所

课程回顾：并行编程基础

- **如何设计并行程序**
  - 创建并行程序的各个环节
  - 如何识别并保障任务的依赖关系
- **并行程序优化：任务的负载平衡**
  - 减少同步成本
  - 静态分配与动态分配
- **并行程序优化：数据局部性与通信**
  - 固有与人为引发的通信
  - 如何降低通信开销
- **并行程序优化：竞争**
  - 通过复制竞争资源等方法来降低竞争开销

授课提纲

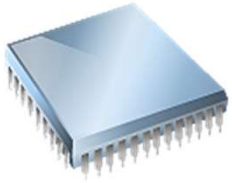
- **并行计算性能概述**
- **并行计算的算法模型和评价方法**
  - Operation-Operand Graph
  - 阿姆达尔定律 Amdahl’ s law
  - 古斯塔夫森定律 Gustafson’ s Law
  - 孙倪定律 Sun-Ni’ s law
  - 等效法则 Isoefficiency Law
- **并行计算的系统模型和评价方法**
  - PRAM模型
  - BSP模型
  - LogP模型

并行计算性能概述

什么是性能

- **在计算领域，性能由两方面因素决定**
  - 计算需求 (what needs to be done)
  - 计算资源 (what it costs to do it)
- **需要计算的问题转化为计算需求**
- **计算资源相互作用，需要权衡取舍**

$$Performance \sim \frac{1}{Resources\ for\ solution}$$



算力

带宽

能耗

…

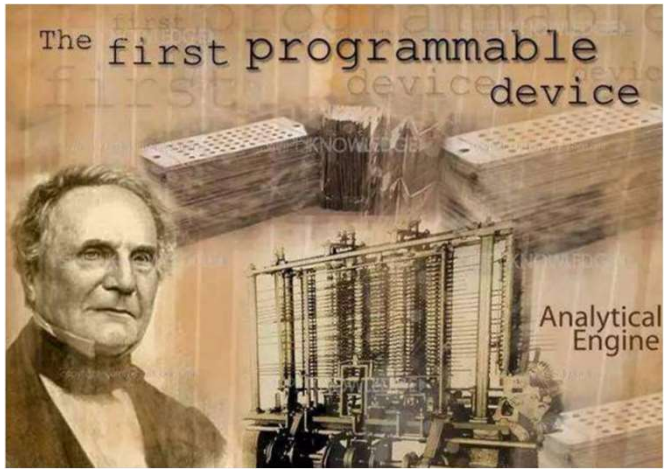
钱

我们为什么关注性能

- 性能表征了计算需求被满足的程度
- 通过分析性能，我们可以更加清楚的理解计算需求和资源之间的相互关系
- 性能一定程度反映了我们解决计算问题的方法的有效性
  - 如何调整方法来更好的解决问题

"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

Charles Babbage, 1791 - 1871



什么是并行性能

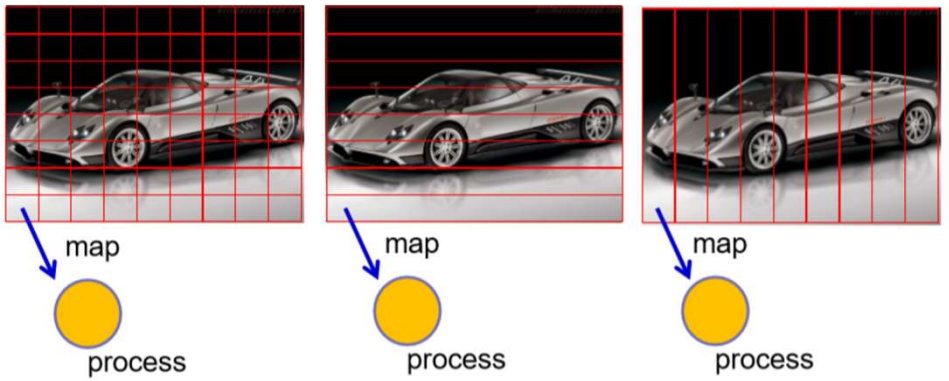
- **我们的课程主要关注在并行计算环境下的性能问题**
  - 与并行计算问题和并行计算资源相关
- **性能是并行存在的理由**
  - 并行方法的性能 vs. 串行方法的性能
  - 如果性能无法获得提升，使用并行就没有意义
- **并行处理使用各种方法和技巧实现计算的并行化（代价）**
  - 硬件, 网络, 操作系统, 并行编程库, 并行语言, 编译器, 算法, 工具, 等等...
- **并行一定要，或者最终期望获得性能优势**
  - 如何做到？如何做的更好？

性能期望

- 如果每个处理器具备 k MFLOPS 计算能力，我们有 p 个处理器，我们的并行程序能否获得 k*p MFLOPS 性能?
- 如果一个问题需要 1 个处理器执行 100 秒，10 个处理器是否可以用 10 秒完成?
- 通常情况下会有多种要素同时对性能造成影响
  - 每个要素需要独立分析
  - 要素之间相互作用为分析和优化带来困难
    - 解决一个问题有可能产生另一个问题
    - 问题之间可能会相互掩盖
- 扩展（系统规模、问题大小）千变万化
- 需要理解性能空间

理想情况：完美并行

- **一个并行计算过程如果可以显式的拆分成多个完全独立运行的子过程，那么该计算可以称为易并行计算（An embarrassingly parallel computation,** 也叫 **pleasingly parallel** or **perfectly parallel ）**
  - 各个计算进程之间完全没有交互（Truly embarrassingly parallel）
  - 只在计算进程划分和结果收集合并阶段交互（Nearly embarrassingly parallel computation）
- **易并行计算能够在并行平台上获得最好的加速效果**
  - 如果串行计算过程耗时 *T*，那么在 *P* 个处理器平台上，易并行计算过程的耗时就是 *T/P*
  - 在哪些情况下无法达到这样的效果?



性能和扩展性

- **性能度量**
  - **影响并行性能的核心因素**
    - algorithm + architecture
  - **串行执行时间：** *Sequential* runtime (T_{seq})
    - f (problem size, architecture)
  - **并行执行时间：** *Parallel* runtime (T_{par})
    - f (problem size, parallel architecture)
- **可扩展性（Scalability）**
  - 在确定的应用背景下，计算机系统（或算法或程序等）性能随处理器数量的增加、处理问题规模的增大，而按比例提高的能力

可扩展性 Scalability

- **如何评估性能Scalability的好坏?**
- **比较（加速比）评估法**
  - 增加处理器的数量，如增加一倍，性能发生了何种变化？
  - 是否是线性增长?
- **并行效率评估法**
  - 随着处理器数量的增加，效率是否保持不变?
  - 与加速比直接相关
- **并行开销等其他性能指标**

性能评估参数和公式

- T_1 是算法在单个处理器上执行的时间
- T_p 是算法在多个处理器上并行执行的时间

- ***S(p)*** **(**S_p**) is the *speedup* 加速比** $S(p) = \frac{T_1}{T_p}$
- ***E(p)*** **(**E_p**) is the *efficiency* 并行效率** $Efficiency = \frac{S_p}{p}$
- ***Cost(p)*** **(**C_p**) is the *cost* 并行开销** $Cost = p \times T_p$

- **最优可扩展性的并行算法**
  - $C_p = T_1$, $E_p = 100\%$, $S_p = p$

可扩展的并行计算

- **并行架构的可扩展考量**
  - **计算**：处理器数量和核数
  - **访存**：存储层次
  - **互连**：互连网络架构
  - 木桶效应：避免关键的体系结构瓶颈
- **计算问题的可扩展考量**
  - 计算问题的规模是否可扩展
  - 计算算法层面
    - 计算访存比（算法抽象）
    - 计算通信比（算法实现）
- **最终还得结合具体的并行编程模型和工具考量，来实现性能的可扩展**

导致应用可扩展性差的因素

- **应用本身的串行算法实现性能**
- **关键路径（Critical Paths)**
  - 分布在不同处理器上的计算过程之间的依赖关系
- **瓶颈问题（Bottlenecks)**
  - 某个或者某些处理器持有着程序推进的关键信息但执行太慢
- **算法开销（Algorithmic overhead)**
  - 某些算法步骤的并行化需要额外更多的开销
- **通信开销（Communication overhead)**
  - 额外花在不同进程之间通信的时间，随系统规模增大而增大
- **负载均衡（Load Imbalance)**
  - 所有处理器都在等待计算最慢的那个（**Straggler**）
- **无法预测的损失（Speculative loss)**
  - 并行计算 A 和 B，但B的结果并不需要

关键路径Critical Paths

- **过长的进程间依赖造成关键路径，是制约性能提升的主要问题**
- **如何发现**
  - 性能停滞在一个（相对）固定的值上
- **如何解决**
  - 从关键路径上移除一些任务，缩短依赖链条

处理瓶颈Bottlenecks

- **如何发现**
  - 处理器 A 正忙，而其他处理器在等待A的结果
  - 常见的场景：
    - N-to-1 reduction → computation → 1-to-N broadcast
    - 用单个处理器（or控制单元）处理请求分配工作
- **如何解决**
  - 使用更加有效的通信手段（Mellanox SHARP）
  - 使用层次化的主从计算架构
- **程序可能在很长一段时间内不会出现瓶颈问题，而当系统扩展的时候影响才会显现**

算法开销Algorithmic Overhead

- **解决同一问题可以使用不同的串行算法**
- **串行算法的选择一定程度上决定了并行算法的性能**
  - 所有并行算法在单个处理器上执行时就回退回串行算法
- **所有的算法并行化都会引入额外的并行开销(Why?)**
  - *Parallel overhead*
- **我们应该从哪里入手考虑算法并行化?**
  - 最好的串行算法可能根本不能并行化
  - 或者无法被很好的并行化（不可扩展）
  - 选择最小化开销的算法变体
- **性能是并行算法设计的难点**
  - 能否获得更优的并行性能?
  - 与最好的串行算法相比是否获得了性能提升?

并行性能概述部分小结

- **性能是并行计算存在的基础**
- **并行计算性能主要关注可扩展性**
  - 加速比度量
  - 效率度量
  - 开销度量
- **制约可扩展性的一些要素**
- **分析并行性能是开发并行算法过程中的关键环节**
  - 预估某个特定并行算法的并行效率
  - 预估求解特定问题的最大加速比(评估求解该问题的每种并行方法)
  - **并行性能优化往往是一个需要反复迭代的过程，多个优化策略之间往往会相互影响**

并行计算的算法模型和评价方法

评估方法：“Operations-Operands” Graph…

- **操作-操作数 图模型（operations-operands graph） 用于描述解决特定问题的特定算法中的依赖关系信息**

- **为了简化分析过程，可以做以下合理假设:**
  - 把所有计算操作的开销定为1，度量单位可以是具体时间、时钟周期数等等；
  - 计算单元之间的数据传递不计入开销，假设通信即刻完成。

“Operations-Operands” Graph…

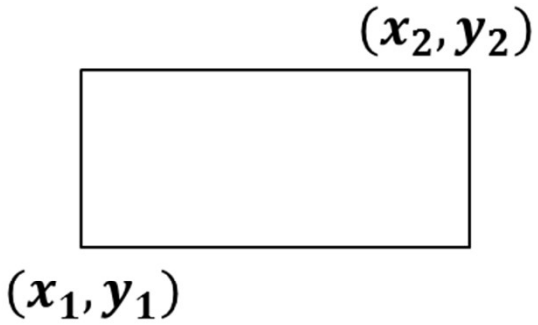
- **把解决特定问题的特定算法的所有操作的集合、以及操作之间的依赖关系表示为一个有向无环图**

$$G=(V,R)$$

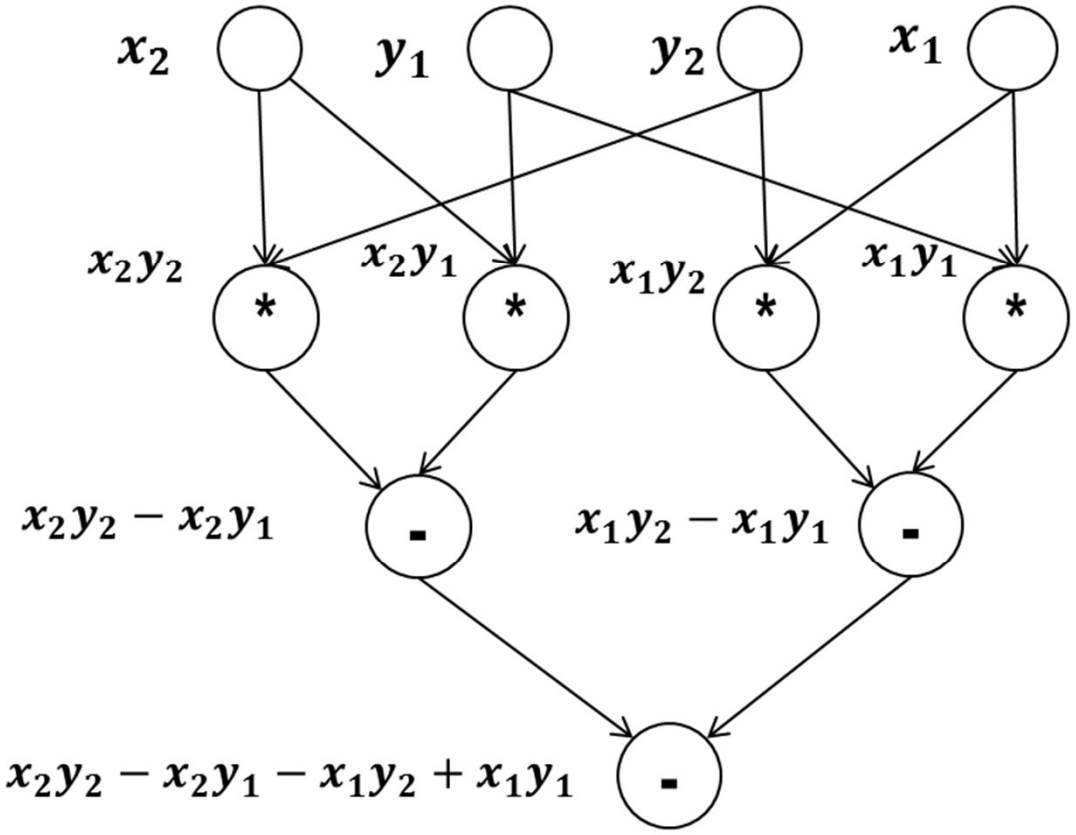
- **其中**:
  - $V = \{1, \ldots, |V|\}$ 是图中所有顶点的集合，表示算法中所要执行的所有操作
  - R 是图中所有弧的集合，只有操作 *j* 使用操作 *i* 的结果，*r(i, j)*才会存在
  - 没有入弧的顶点用于表示输入操作，没有出弧的顶点用于表示输出操作
  - *d(G)* 是图的直径（最大路径的长度）

“Operations-Operands” Graph…

例子: 求解矩阵面积（已知条件：对角顶点的坐标）算法的“Operations-Operands” Graph

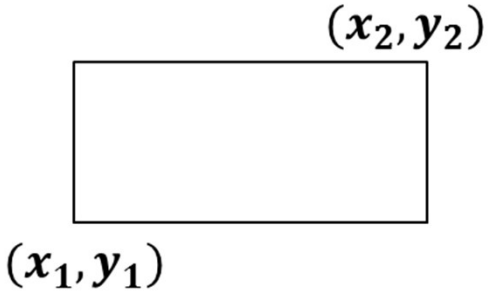


$$S = (x_2 - x_1)(y_2 - y_1)$$
$$= x_2y_2 - x_2y_1 - x_1y_2 + x_1y_1$$

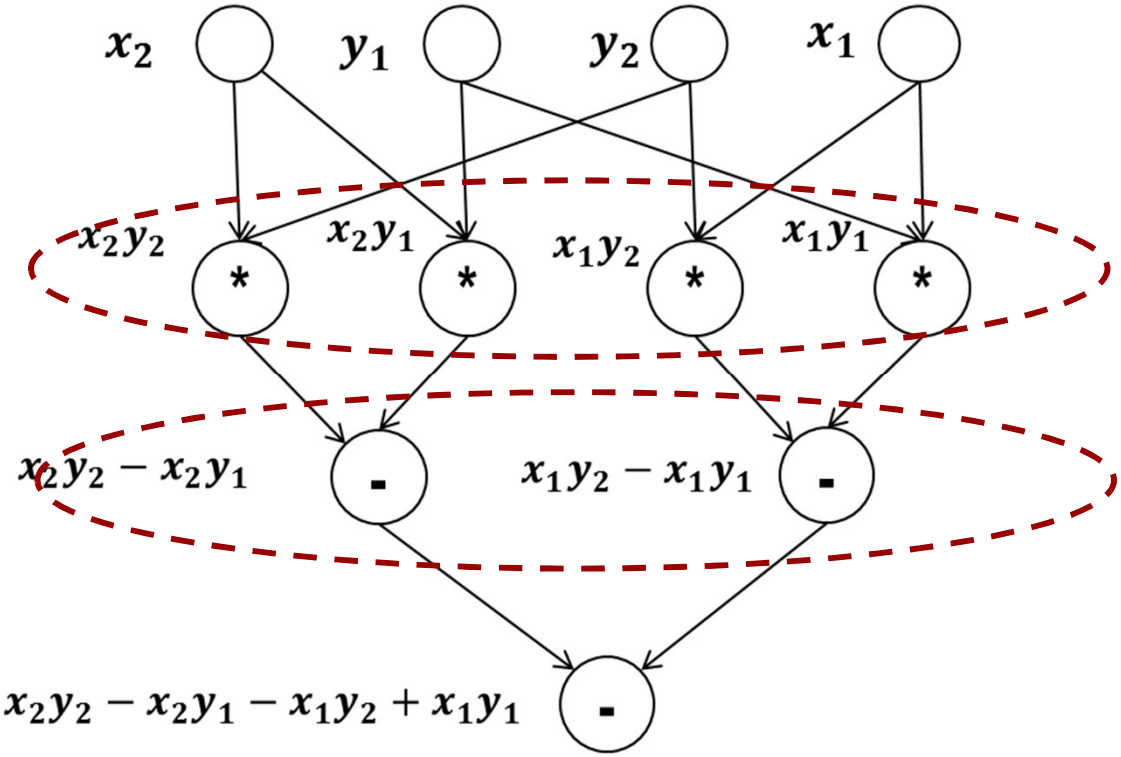


“Operations-Operands” Graph

- 不同的计算方案提供了不同的并行化能力，首先需要选择**最适合**并行执行的计算方案
- 在选定的计算方案中，在“Operations-Operands” Graph表示下，操作（顶点）之间没有路径（弧），就可以**并行执行**



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



基于O-O Graph的并行算法形式化表达

- **假设 p 为并行处理特定算法所使用的处理器数量，那么该处理所包含的所有操作可以使用下面的方式来表达（即为了实现该并行处理，需要按照下述调度集合来安排每个处理器上的操作）：**

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

  - **i** 表示第i个操作，i的集合V就是前面讲过的操作-操作数 图中顶点的集合；
  - P_i 表示执行第i个操作的处理器；
  - $\mathbf{t_i}$ 是第i个操作开始执行的时间。

- **上述表达必须满足以下条件：**
  - 同一处理器上不能同时执行两个不同的操作：

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

  - 每个操作的操作数必须在该操作执行时间开始之前准备好：

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

评估并行处理的耗时

- 给定一个并行算法:

$$A_P(G, H_P)$$

- 并行算法的执行时间由**调度集合中的最大时间值**决定:

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

- 最优调度下并行算法的执行时间是:

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

评估并行处理的耗时

- 通过拟定最佳计算方案，可以缩短执行时间：

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

- 如果使用的处理器数目不受限制，并行算法执行的最小可能时间是：

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

评估并行处理的耗时

- **观察1**

  算法实现方案（O-O Graph）的最大路径长度决定了并行算法执行的最小可能时间：

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

在选择算法计算方案时，必须尽可能使用直径最小的图

评估并行处理的耗时

- **观察2**

  在算法的特定实现方案中，假设每个输入顶点到一个特定的输出顶点都有一条路径。同时，每个顶点输入弧的数目（入度）不超过2，那么该算法并行执行的最小时间是：

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

  其中，n 是该算法实现方案的输入边的数量（输入操作数的数量）

评估并行算法的效率

- **加速比*Speedup***

  加速比是算法串行处理时间与使用 p 个处理器的并行处理时间的比值：

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

(其中，参数 n 用来表示所处理问题的计算复杂度，例如，可以理解为问题的输入数据规模)

评估并行算法的效率

- **效率*Efficiency***

  效率表征了并行算法在求解问题时处理器的利用率，由下述公式定义：

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

(效率值反应了处理器实际用于解决问题的比率)

思考：是否有可能出现$S_p(n) > p$的情况?

评估并行算法的效率

- **注意：**
  - **超线性加速比 $S_p(n)>p$ 有可能在以下场景中出现：**
    - 并行和串行的实现方法不同
    - 串行程序执行和并行程序执行之间存在差异（例如：使用单处理器执行时，内存无法容纳全部的数据，需要内存和外存之间数据交换，而使用多处理器时，不存在该问题）
    - 处理问题的复杂度和问题的数据规模之间存在非线性依赖关系

评估并行算法的效率

- **计算开销（代价） *Computation cost***

$$C_p = pT_P$$

- 代价最优（***cost-optimal***）并行算法是指：计算开销（代价）与最优串行算法执行时间成正比的并行算法，也即并行算法的执行时间正比于最优串行算法执行时间

举例: 部分和计算

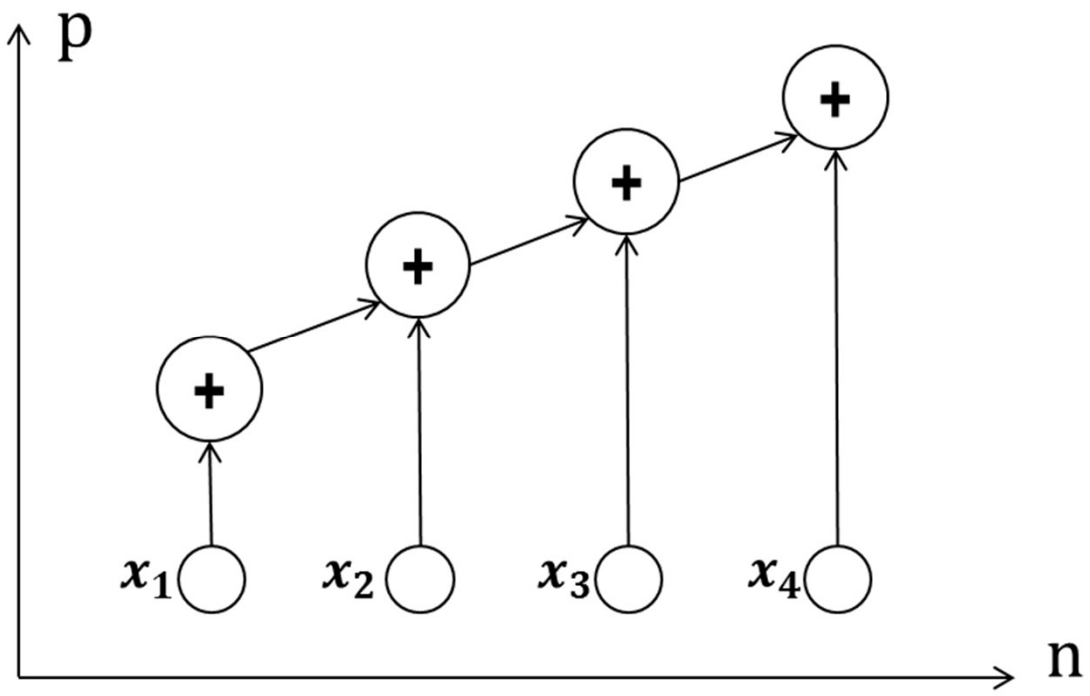
- 求数值序列部分和（partial sums or prefix sum problem）：

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

举例: 部分和计算

- 标准串行算法实现：对所有元素顺序求和

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

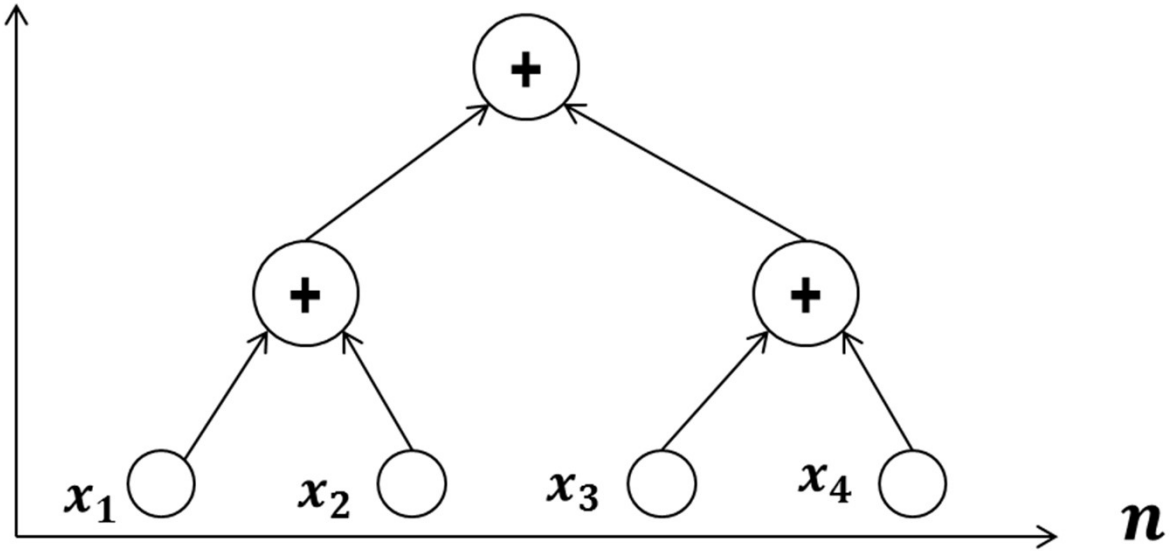


这种标准的串行求和算法
只允许严格串行执行，无法被并行化

举例: 部分和计算

- 级联求和方案

$G_2 = (V_2, R_2)$



- $V_2=\{V_{i1},...,V_{ili}, 0<=i<=k, 1<=l_i<=2^{-i}n\}$ 是图的顶点的集合
- $\{V_{01},...,V_{0n}\}$ – 输入操作
- $\{V_{11},...,V_{1n/2}\}$ – 第一层操作，其他层类推
- $R_2=\{(V_{i-1,2j-1},V_{ij}),(V_{i-1,2j},V_{ij}),1<=i<=k,1<=l_i<=2^{-i}n\}$ – 图中弧的集合.

举例: 部分和计算

- 该方案中所有的求和操作次数等于（串行执行所需步骤）：

$$K_{seq} = \frac{n}{2} + \frac{n}{4} + \ldots + 1 = n\left(\sum_{m=1}^{log_2 n - 1} (\frac{1}{2})^m\right) + 1 = n - 1$$

- 并行执行级联求和方案，所需要的并行步骤等于：

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

举例: 部分和计算

- 下面是级联求和方案的并行加速比和效率:

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

其中 p=n/2 是级联求和方案并行处理所需的处理器数量:

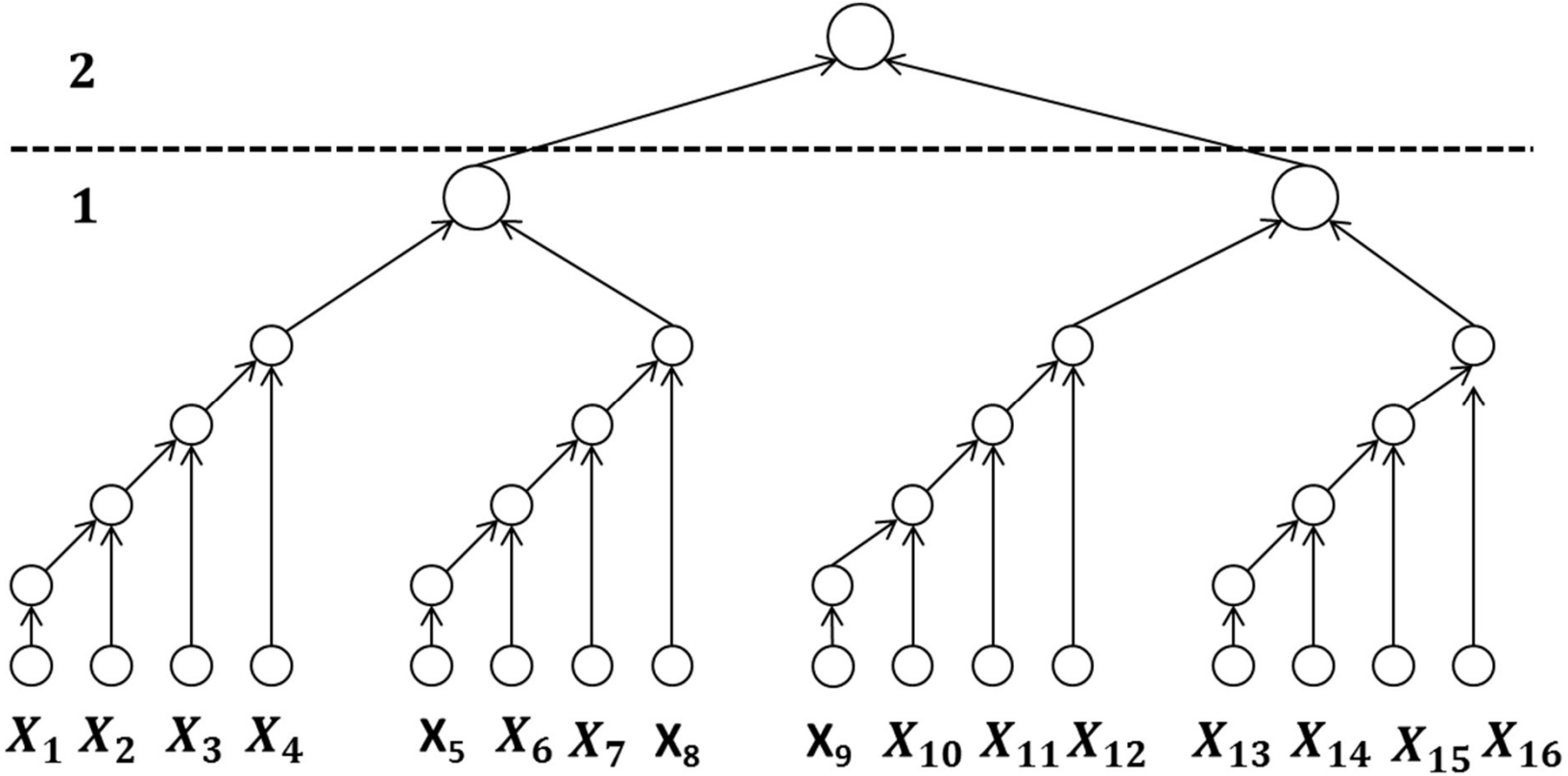
— 级联方案的并行处理时间与前面观察2的估计吻合

— 处理器的效率随求和数据规模的增加而下降，如何优化?

$$\lim E_p = 0 \qquad n \to \infty$$

举例: 部分和计算

- **修改的级联方案:**
  - 把需要求和的序列拆分成 (n/log_2n) 个组，每个组中有 log_2n 个元素。在第一阶段，每个组内部使用串行算法求和
  - 在第二阶段，使用级联算法对各组总计 n/log_2n 个结果进行求和 。



举例: 部分和计算

- 如果第一阶段使用$\boldsymbol{p = (n/log_2 n)}$ 个处理器，那么第一阶段的并行执行时间就是 $\boldsymbol{(log_2 n)}$
- 第二阶段使用 $\boldsymbol{p_2 = (n/log_2 n)/2}$ 个处理器，并行执行时间是 $\boldsymbol{log_2(n/log_2 n) \leq log_2 n}$
- 整个算法使用$\boldsymbol{p = (n/log_2 n)}$ 个处理器，执行时间（估算值）为:

$$\boldsymbol{T_P \leq 2\, log_2\, n}$$

举例: 部分和计算

- 修改的级联方案的加速比和效率:

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

- 与原来的级联方案相比，修改后的级联方案加速比缩减为原来的1/2
- 修改后的级联方案在数据规模不断增大时，效率不再趋于0:

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

- 改进的级联算法是代价最优的，因为计算开销（代价）与串行算法执行的时间成正比:

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

举例: 部分和计算

- **计算所有的部分和**
  - 在标量计算机上，所有的部分和的计算可以用传统的顺序求和算法来完成，运算次数为：

$$T_1 = n$$

  - 上述问题的并行执行中，直观地使用级联方案并不会带来理想的结果

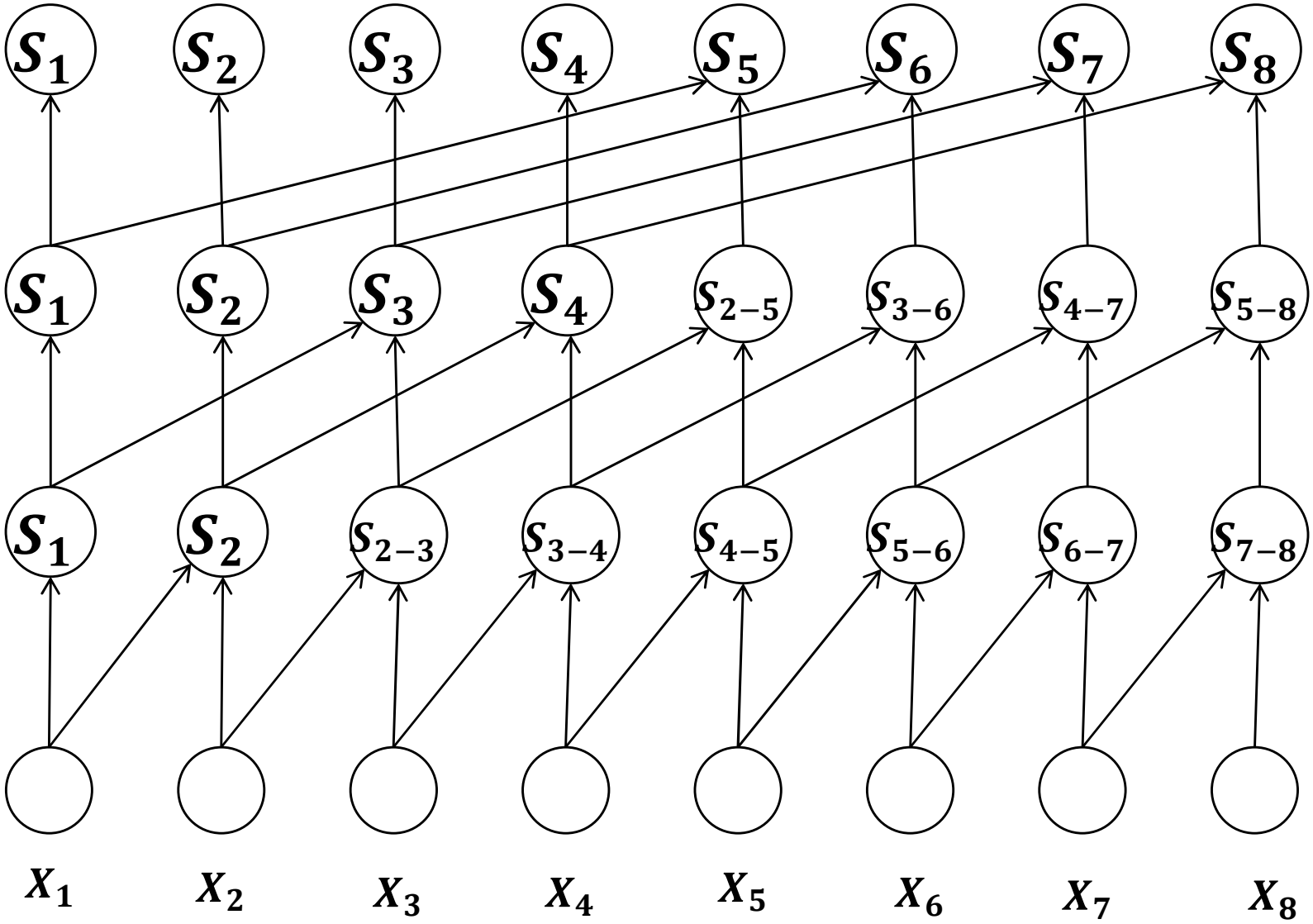
为了更好的解决问题，有效的并行化需要新的方法(甚至是那些在串行编程中没有类似的方法)，开发新的专门面向并行的算法

举例: 部分和计算

- **计算所有的部分和**
  - 能够实现 $log_2 n$ 步骤完成的并行算法:
    - 在计算开始前，创建向量 S 用于存储每步的求和值 (初始化，S=x)
    - 随后在每个求和迭代步骤 $\boldsymbol{i}$，$\boldsymbol{1 \le i \le log_2 n}$, 增加一个辅助向量 Q， Q 是由 S 向右位移 2^{i-1} 个单元得到（右移后，左边空出来的单元补0）.
    - 在该迭代步骤将 Q 和 S 相加.

举例: 部分和计算

- 所有的部分和计算并行算法过程



举例: 部分和计算

- **所有的部分和计算:**
  - 上述算法所需的总的标量操作数（注意与串行对比差异）：

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

  - 所需总的处理器的数量与数列的元素个数相同:

$$p = n,$$

  - 上述算法的加速比和效率计算:

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

最大可能并行性的估计

- 并行计算效率的估计需要知道加速比和效率的最佳(最大可能)值
- 大部分耗时的计算问题都很难达到理想的加速比$S_P = P$和效率 $E_P = 1$，可能会引入额外的计算量

并行三定律之一：阿姆达尔定律

- **阿姆达尔定律 Amdahl’ s Law**
  - 大多数问题处理过程中存在无法被并行化的部分串行计算部分，因此无法达到理想的最大加速比 $T_P = p$
  - 设 f 是原应用处理算法中需要顺序计算的部分占比（归一化处理）
  - 假设使用 p 个处理器执行该算法的并行版本，则计算加速比受下述公式制约:

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

阿姆达尔定律和可扩展性

- **可扩展性**
  - 在确定的应用背景下，计算机系统（或算法或程序等）性能（加速比）随处理器数量的增加，而按比例提高的能力
- **什么时候使用阿姆达尔定律?**
  - 在特定的问题规模下When the problem size is fixed
  - **强扩展**($p \to \infty$, $S_p = S_\infty \to 1/f$)
  - 加速上限由计算中串行执行时间的长短决定, 而不是 # processors!!!
  - **Uhh, this is not good ... Why?**
  - 完美的效率是难以实现的
- **感兴趣可以去读一下 Amdahl 的原始论文**
  - 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
  - G.M. Amdahl, Computer architecture and Amdahl's law. Computer, 46 (12) (2013), pp. 38-46

阿姆达尔定律

- **阿姆达尔定律要点:**
  - 串行计算部分制约了加速比，如果我们选择更合适的并行化方法，串行计算的部分可能会大大减少
  - **阿姆达尔效应*:***
    - 对于大多数问题来说，求解过程中的串行执行部分 ***f=f(n)*** 随着问题规模 ***n*** **的增大而递减**。在这种情况下，由于需要解决的问题的计算复杂性增加，固定数量处理器下加速比可能会增加。

二：Gustafson定律

- **许多应用问题更注重求解精度**
  - 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
  - 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，也就是工作量增加了10,000倍
- **John Gustafson提出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

Gustafson定律

- 任务规模随着处理器核数可扩展
- 单核执行时的任务量：a+b
- p个核执行时的任务量：a+pb
- 另f表示单核程序中可并行的部分

$$f = \frac{b}{a+b}$$

- 将f代入上面的公式

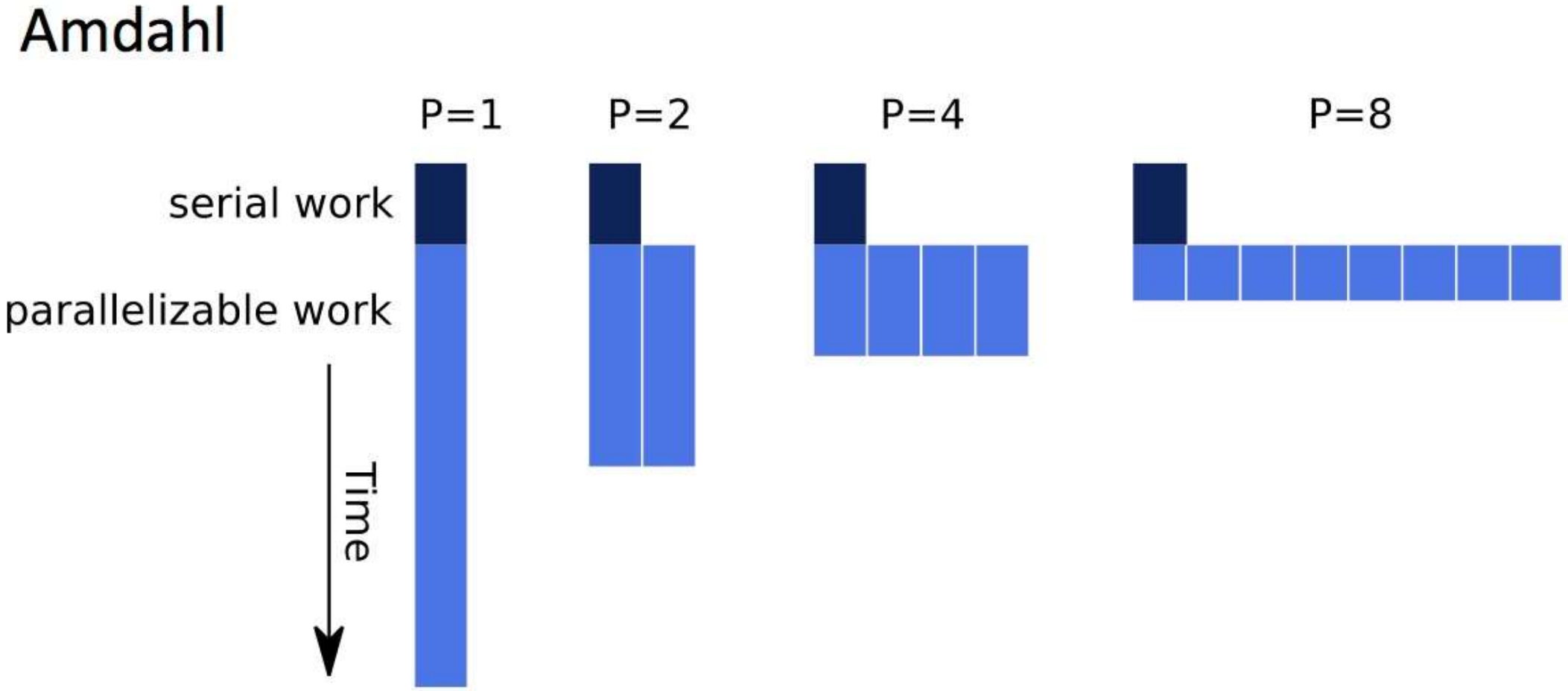
$$\text{加速比} = \frac{a+pb}{a+b} = 1 - \frac{b}{a+b} + \frac{pb}{a+b}$$

$$\text{加速比} = 1 - f + pf = 1 + (p-1)f$$

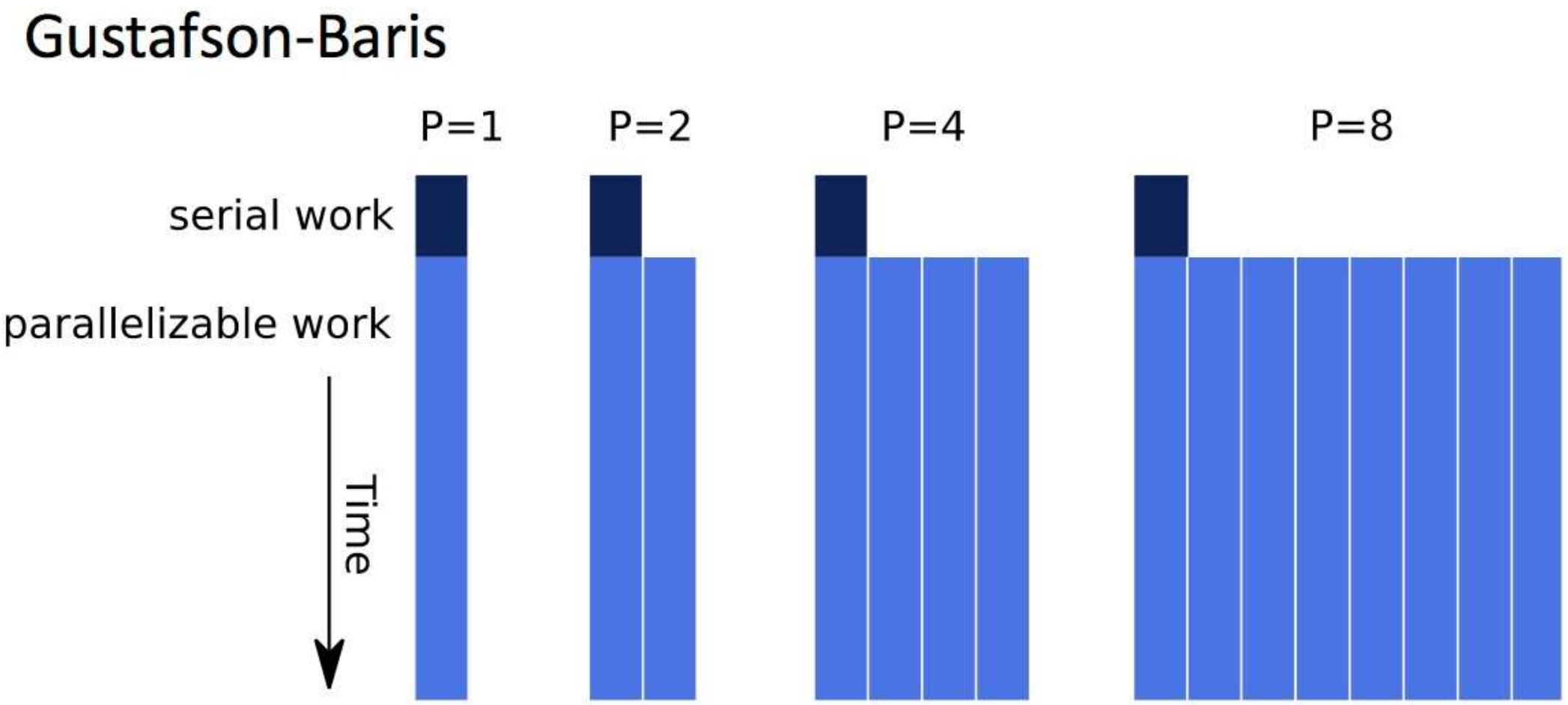
Gustafson定律和可扩展性

- **可扩展性**
  - 在确定的应用背景下，计算机系统（或算法或程序等）性能随处理器数量的增加和**处理问题规模**的增大，而按比例提高的能力
- **什么时候应用Gustafson定律?**
  - 当问题大小随着处理器数量的增加而增加时
  - **弱扩展** $(S_p = 1 + (p-1)f_{par})$
  - 加速比函数中包含处理器的数量!!!
  - 当问题扩展时，可以保持或增加并行效率
- **感兴趣可以去读一下 Gustafson的原始论文**
  - http://johngustafson.net/glaw.html

Amdahl versus Gustafson



Amdahl versus Gustafson



三、孙倪定律(1990)

- 任务的扩展性受限于内存容量
- Suppose f is the portion of the workload that can be parallelized and (1-f) is the sequential portion of the workload.
- Let y=g(n) be the function that reflects the parallel workload increase factor as the memory capacity increases

$$Speedup_{\text{memory-bounded}} = \frac{(1-f)+f\cdot G(n)}{(1-f)+\frac{f\cdot G(n)}{n}}$$

和Amdal定律和Gustafson定律分别是什么关系?

孙贤和教授，IEEE Fellow
IIT杰出教授，计算机系主任
2018年CCF海外杰出贡献奖
中科院计算所客座研究员

并行计算可扩展性分析

如果一个并行算法，在增加所使用的处理器数量时，一方面能够提高加速比，另一方面还能将处理器的利用率保持在一个相对稳定的水平，那么该算法就可以称为**可扩展的并行算法**

并行计算可扩展性分析

由于需要增加处理器之间的数据交互，以及不同处理并行进程之间的同步等额外增加的问题，我们定义并行计算引入的**额外开销**：

$$T_0 = pT_P - T_1$$

基于上述额外开销的定义，重新表示并行处理所需的时间和加速比:

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

同样，处理器的效率可以表示为：

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

并行计算可扩展性分析

- 对于固定复杂度的问题($T_1 = const$)，随着处理器数量的增加，由于并行带来的额外开销T_0增加，效率会降低
- 对于固定数量的处理器，问题复杂度T_1的提高可以提高处理器利用率（即效率）
- 在大多数情况下，如果处理器的数量增加，可以通过增加问题复杂度来维持必要的效率水准。

可扩展性

- **强可扩展**
  - 增加线程数，问题规模不变，并行性能可扩展
- **弱可扩展**
  - 增加线程数，同时以相同倍率增加问题规模，并行性能可扩展

思考：哪种扩展性更难实现？

并行计算可扩展性分析

- 假设 $E = const$ （我们期望的处理器利用率保持在一个不变的水平），通过效率的开销表达公式可以得到：

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

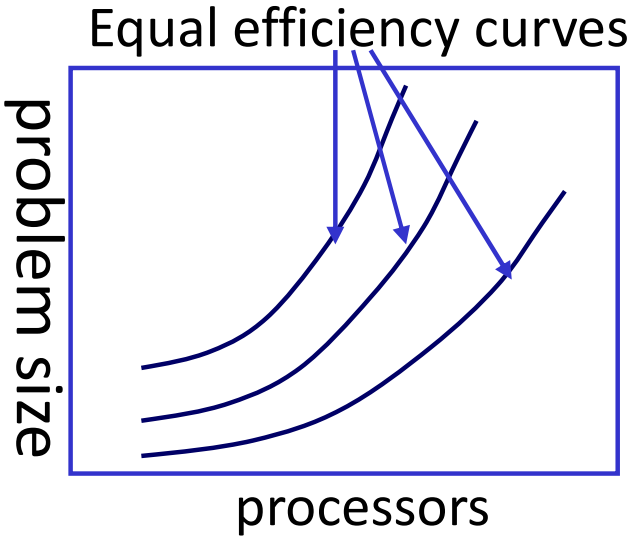
W表示给定问题在一个处理器上的串行执行时间，也即T_1

$$W = K(pT_P - W)$$

- 在保持效率为常量的条件下，问题规模和处理器数量之间的关系函数 n=F(p) 被称之为**等效率函数 isoefficiency function.**

等效率Isoefficiency

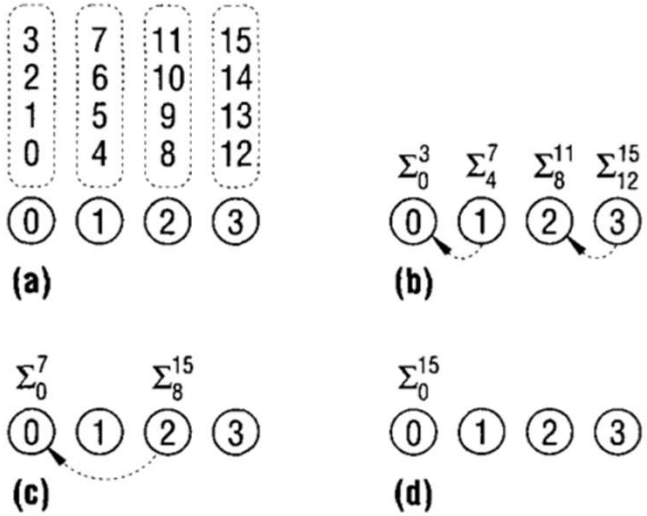
- 使用等效率的目标是量化可扩展性
- 在大型机器上保持同样的效率需要增加多少问题规模?
- 效率 Efficiency
  - $E = T_1 / (p * T_p)$
  - T_p = computation+communication+idle
- 等效率 Isoefficiency
  - 等效率曲线方程
  - 如果曲线是线性：系统可扩展
  - 如果曲线是指数的：系统不可扩展



- 感兴趣可以读一下 Kumar的原始论文
  - Ananth Y. Grama, Anshul Gupta, and Vipin Kumar, Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures, IPDT, 1993

举例: 对n个数求和的系统可扩展性

- 在 *p* 个处理器上用分段（级联）的方法对 *n* 个数求和

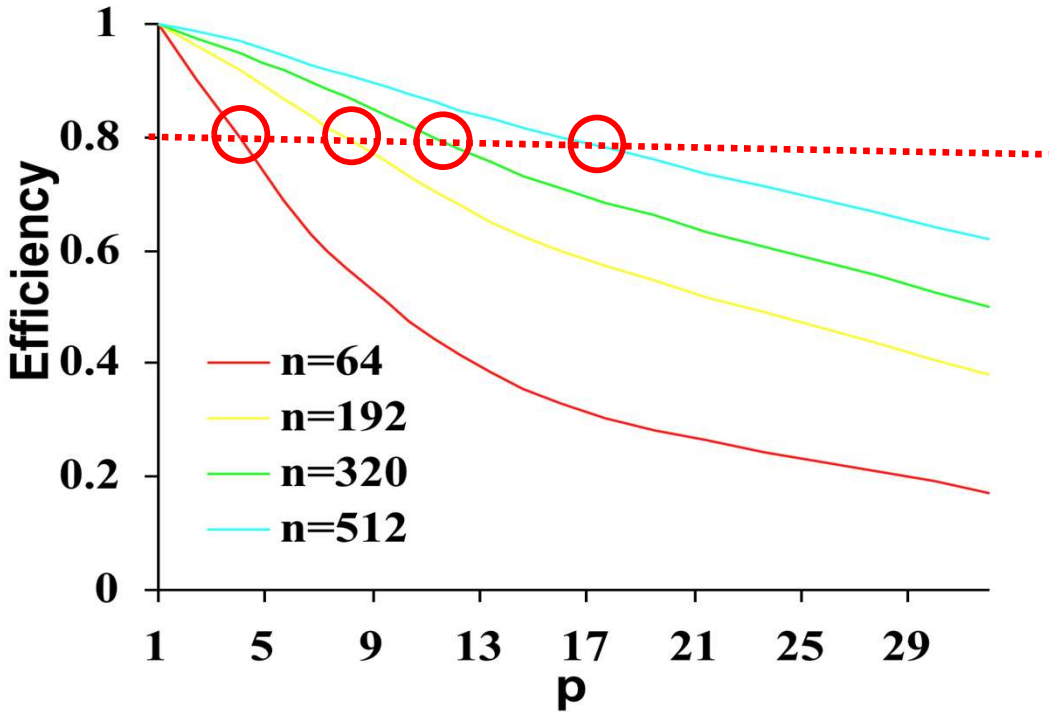


$$T_{par} = \frac{n}{p} - 1 + 2\log p$$

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

$$\approx \frac{n}{\frac{n}{p} + 2\log p}$$

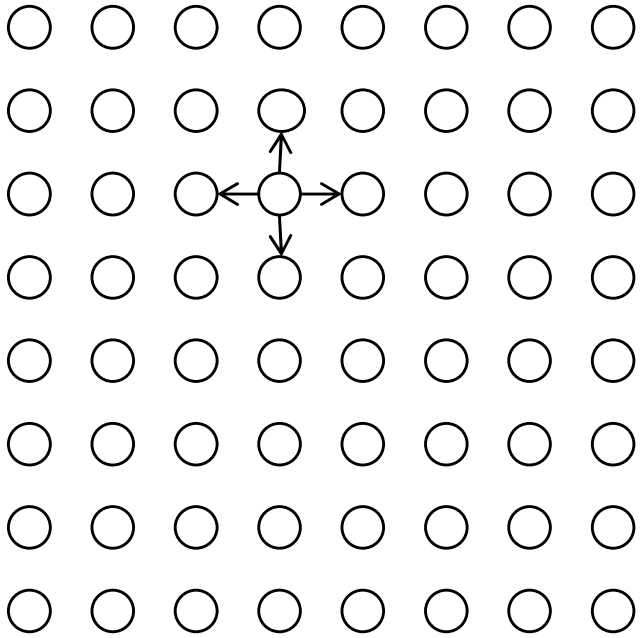
$$Efficiency = \frac{S}{p} = \frac{n}{n + 2p\log p}$$



举例: 有限差分法的系统扩展性

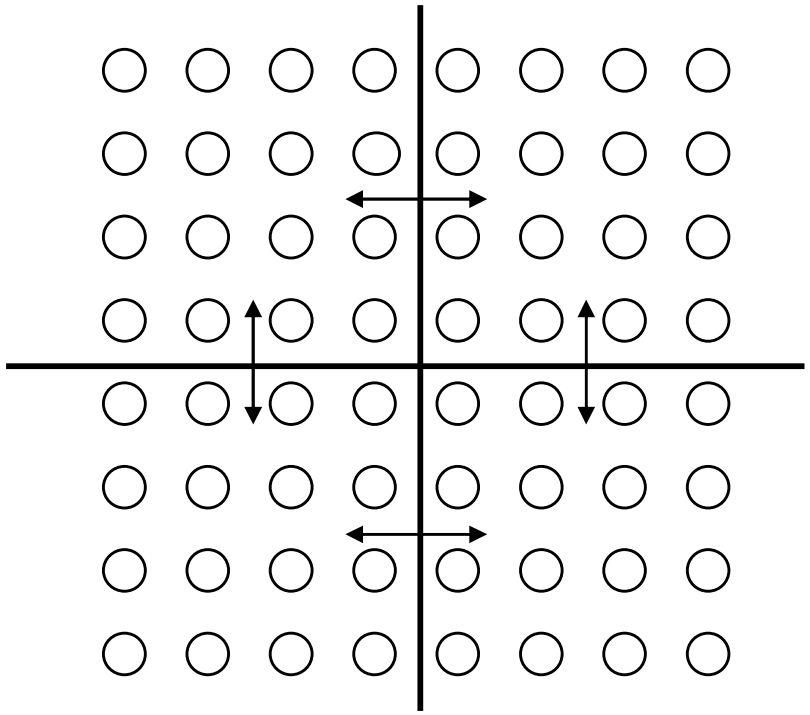
- 有限差分法被广泛用于偏微分方程的数值求解
- 来看下述算法 (N = 2)

$$X_{i,j}^{t+1} = w(X_{i,j-1}^{t} + X_{i,j+1}^{t} + X_{i-1,j}^{t} + X_{i+1,j}^{t}) + (1-w)X_{i,j}^{t}$$



举例: 有限差分法的系统扩展性

- 每个处理器使用网格点$(n/\sqrt{p}) * (n/\sqrt{p})$ 对矩形子区域进行计算
- 在每次迭代之后，p个处理器都需要执行同步



举例: 有限差分法的系统扩展性

- **效率分析（这里仅考虑平均到单次迭代时的情况）**
  - $T_1 = W = 6n^2$
  - $T_P = 6\frac{n^2}{p} + log_2 p$
  - **效率**

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

  - **等效率函数**

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

总结

- 利用 “operations-operands” graph描述计算模型，能够反映所选算法中的依赖关系，进而挖掘可并行部分
- 使用加速比、效率、开销等指标评价并行计算方法的有效性
- 并行算法分析的阿姆达尔定律、古斯塔夫森定律、孙倪定律、等效率法则

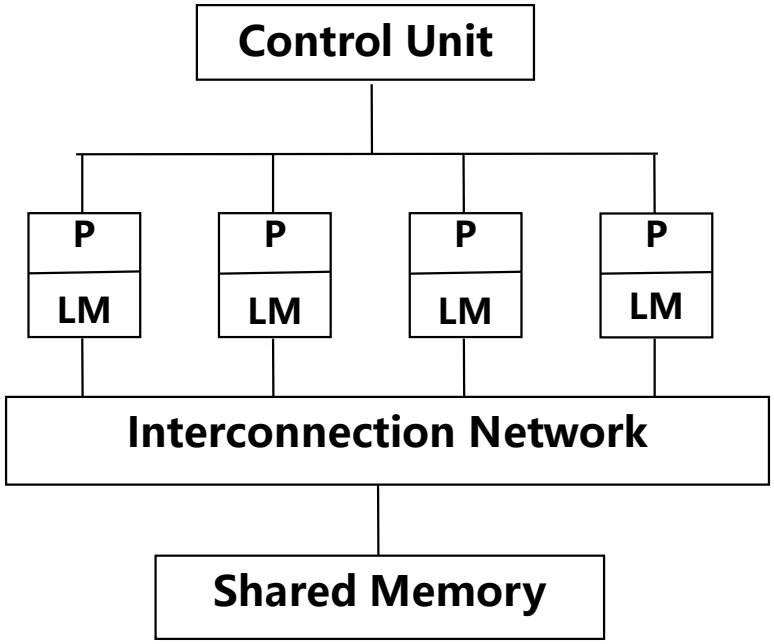
并行计算的系统模型和评价方法

并行计算的系统模型

- **PRAM (并行存储 parallel RAM)**
  - 基础并行系统
- **BSP (整体（块）同步并行 Bulk Synchronous Parallel)**
  - 将计算与通信隔离
- **LogP**
  - 用于研究分布式内存系统
  - 主要着眼于互连网络的影响
- **Roofline（屋顶模型）**
  - 主要分析“计算”和“访存”（供数）之间的关系

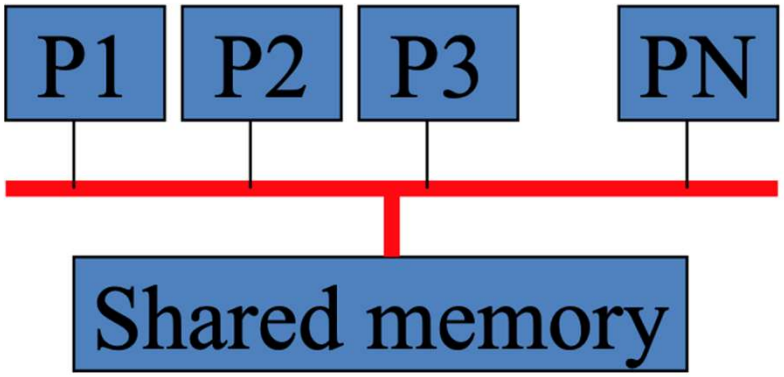
PRAM模型总览

- Parallel Random Access Machine 并行随机访存系统 (PRAM)
- 是共享内存多处理器模型（Shared-memory multiprocessor model)
- 由Fortune和Wyllie 1978年提出，又称SIMD-SM模型。有一个集中的共享存储器和一个指令控制器，通过共享内存的读写交换数据，隐式同步计算



PRAM模型的细节

- 处理器组编号(Pi)： 一组通过全局共享内存单元通信的同步处理器
- 内存单元组编号(M[i])： 每个处理器都可以在单位时间内访问共享内存单元，共享存储器容量无限大
- 一个PRAM 指令包含3 个同步步骤: read (获取输入数据), computation（计算）, write (将结果写入共享内存单元).
- 处理器之间通过往共享内存单元写入和读出数据实现通信，交换数据



按照读写限制对PRAM模型的分类

- ***EREW*** **(互斥读 互斥写 Exclusive Read Exclusive Write)**
  - 对相同内存位置不能并发读，或者并发写
- ***CREW*** **(并发读 互斥写 Concurrent Read Exclusive Write)**
  - 在同一个指令步骤中，多个处理器可以同时读取同一个全局内存单元
- ***ERCW*** **(互斥读 并发写 Exclusive Read Concurrent Write)**
- ***CRCW*** **(并发读 并发写 Concurrent Read Concurrent Write)**
- **性能：CRCW > (ERCW,CREW) > EREW**

并发读并发写(CRCW)的进一步分类

- 通常的方式COMMON:同时写入同一地址的所有处理器必须写入相同的值
- 随机方式ARBITRARY:如果多个处理器同时写入同一地址，则随机选择一个竞争获胜的处理器，并将其值写入该地址
- 基于优先级的方式PRIORITY: 如果多个处理器并发写入同一地址，则具有最高优先级的处理器仲裁获胜将其值写入
- 基于合并的方式COMBINING: 通过某种方式将不同处理器的值合并后写入, e.g., sum, min, or max
- COMMON-CRCW 模型最常使用

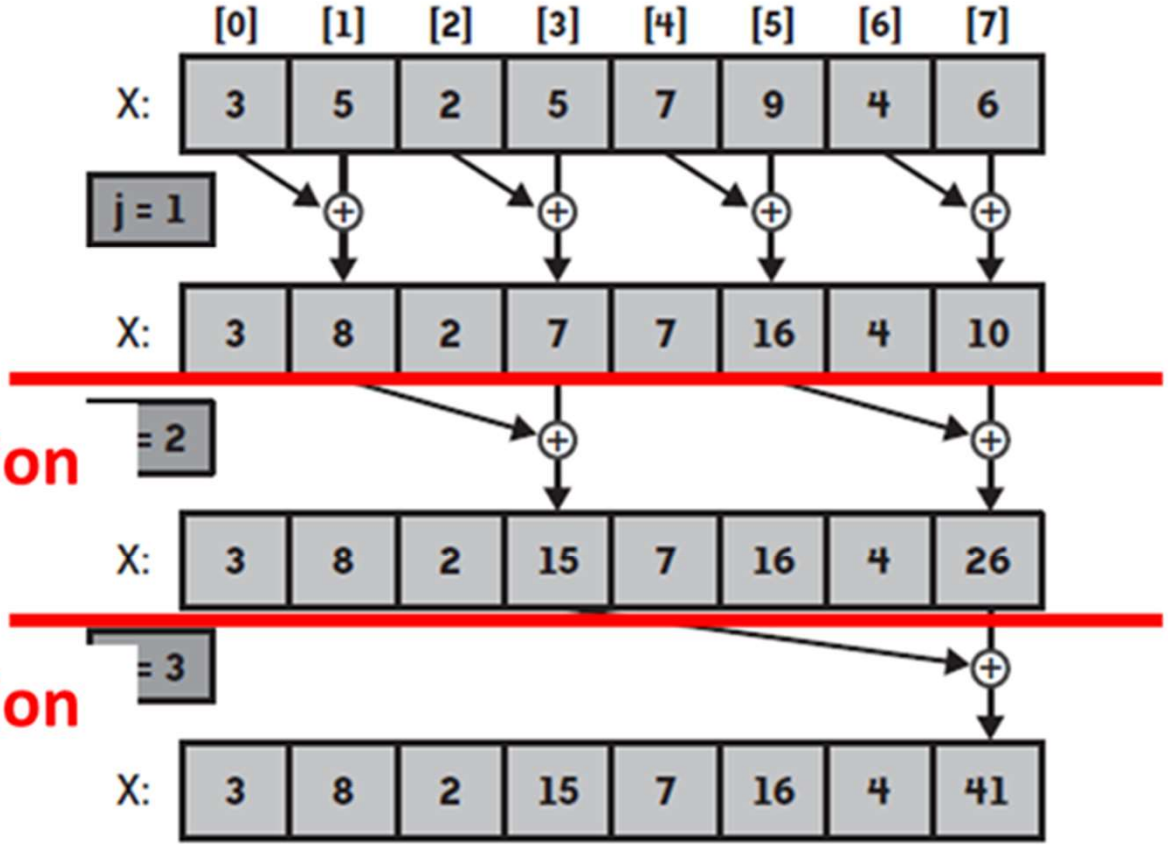
示例：基于PRAM的并行加法

EXAMPLE 6-2. PRAM algorithm for parallel sum

```
for j := 1 to log_2(n) do
  for all k in parallel do
    if (((k + 1) mod 2^j) = 0) then
      X[k] := X[k - 2^(j-1)] + X[k]
    fi
```

Synchronization

Synchronization



PRAM是一个理论模型(Unfeasible)

- 处理器和存储器之间的互连网络有很大的实现开销
- 互连网络上的消息路由所需时间与网络大小相关
- 可以设计PRAM模型的通用算法，并在一个可行的网络上进行仿真

- 优点：隐藏了通信、同步等细节，适合并行算法表示以及复杂度分析，易于使用
- 缺点：不适合MIMD并行机，也无法反应共享存储的竞争、通信延迟等因素

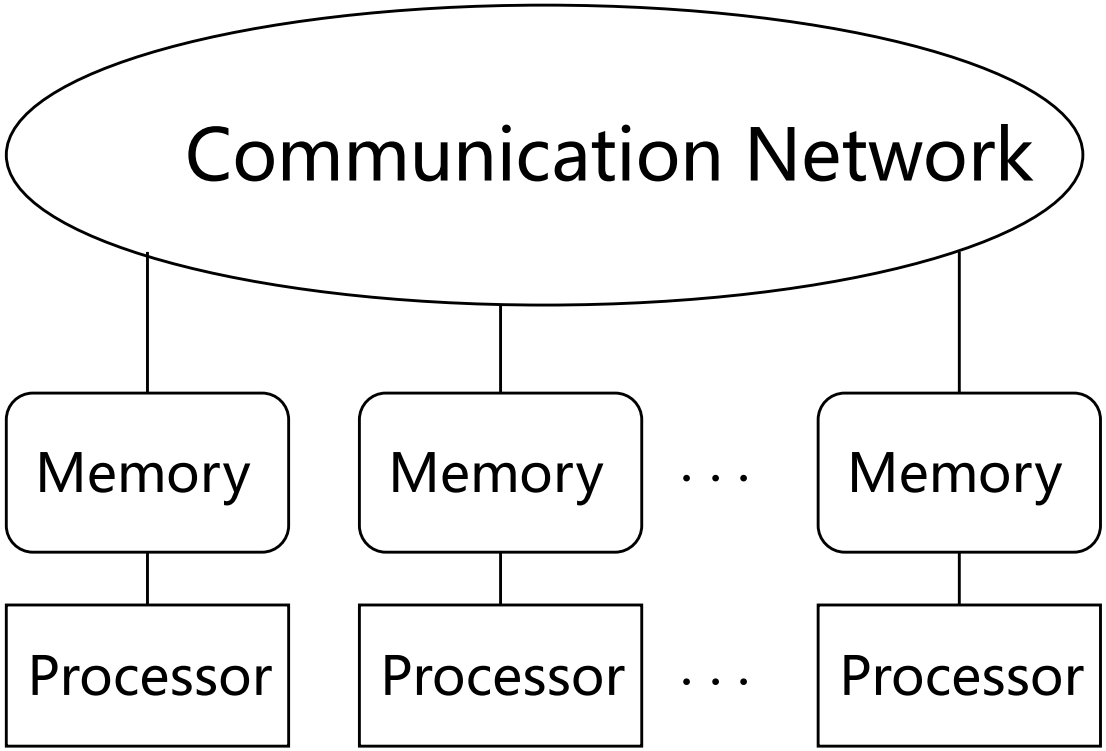
PRAM 直观地符合程序员对并行计算机的看法: **它忽略了较低层次的体系结构约束**，以及诸如内存访问争用和开销、同步开销、互连网络吞吐量、连接性、速度限制和链路带宽等细节。

BSP（Bulk Synchronous Parallel）模型

- “块(bulk)”同步并行模型
- “块”内异步并行，“块”间显式同步
- 由Leslie Valiant 1990年在Harvard提出，用于系统的性能预测
- 可以支持SPMD (Single Program Multiple Data) or MIMD模型
- 支持直接内存访问或者消息传递通信语义

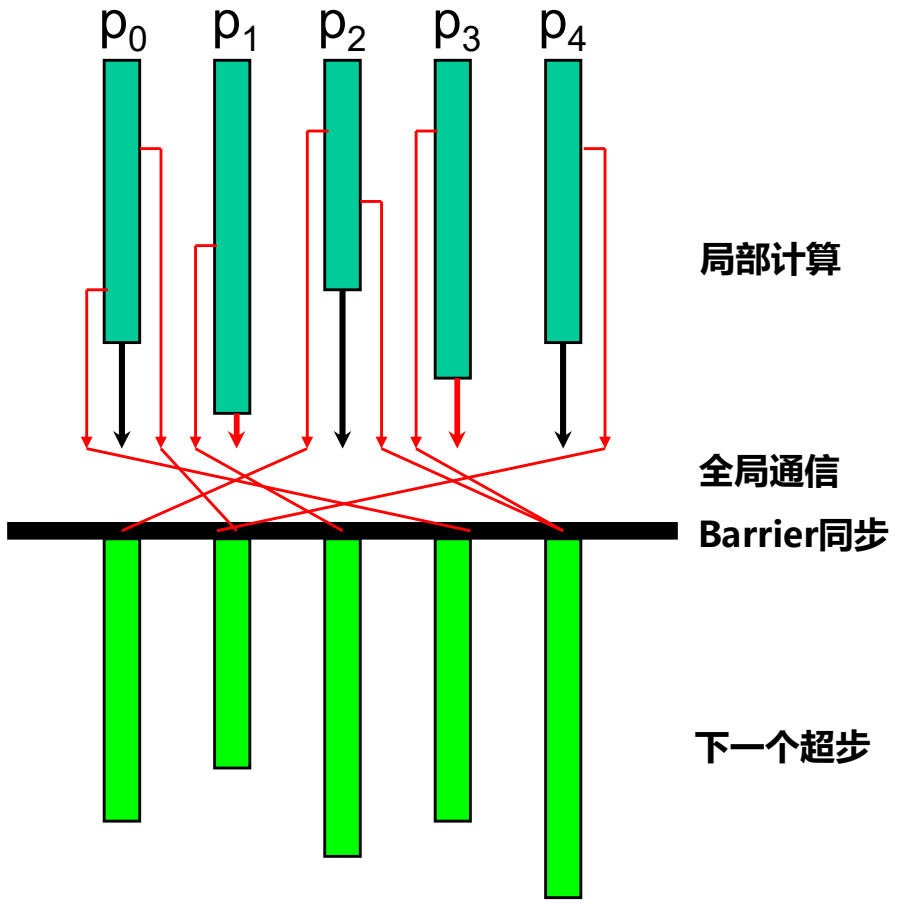
BSP计算机的基本组件

- 一组处理器（带有存储器）
- 一套点对点通信网络（系统互连）
- 一套在可以在不同处理器之间高效实现栅栏同步的机制（显式同步）



BSP模型的超步概念

- 一个BSP模型下的计算过程由一系列的超步组成（*supersteps*）
- 在每个超步中，处理器使用本地内存数据进行计算computations，然后发出通信请求communication
- 进程间在每个超步结束时同步synchronized，所有通信请求都被处理完成后才进入下一超步



BSP性能模型参数

- *p* = 处理器数量
- *l* = 栅栏同步开销
- *g* = 每个消息的通信开销（1/bandwidth）
- *s* = 处理器速度
- 任一处理器在每个超步最多收发 *h* 条消息（called *h*-relation communication）

- 每个超步所需时间= 任一处理器所需执行的最大本地操作数量+ *g***h* + *l*

执行时间计算

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

Where:

w = max of computation time

g = 1/(network bandwidth)

h = max of number of messages

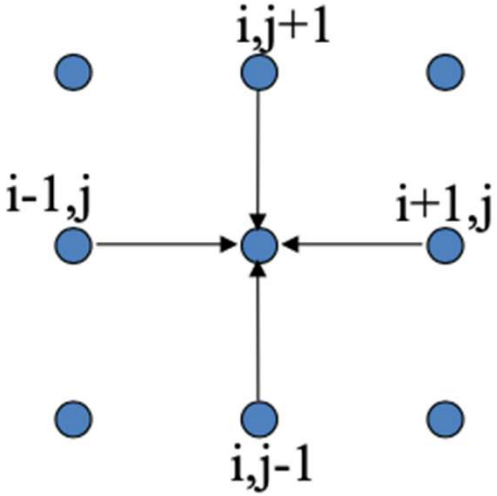
l = time for the synchronization

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

BSP模型算法示例

- 拉普拉斯方程的数值解法

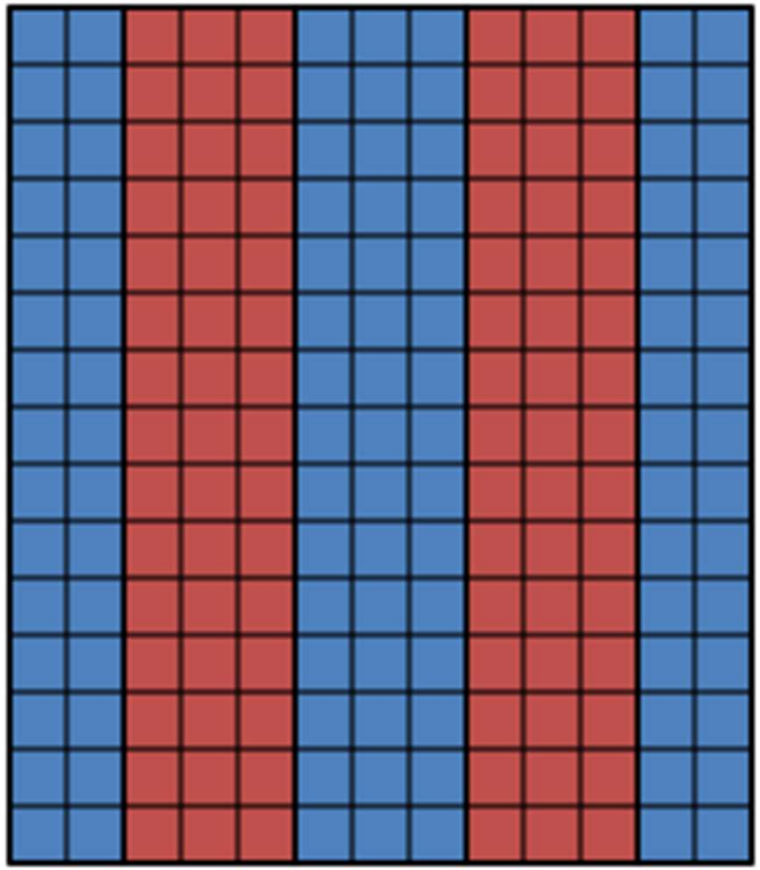
$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^{n} + U_{i+1}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n}\right)$$




```
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
```

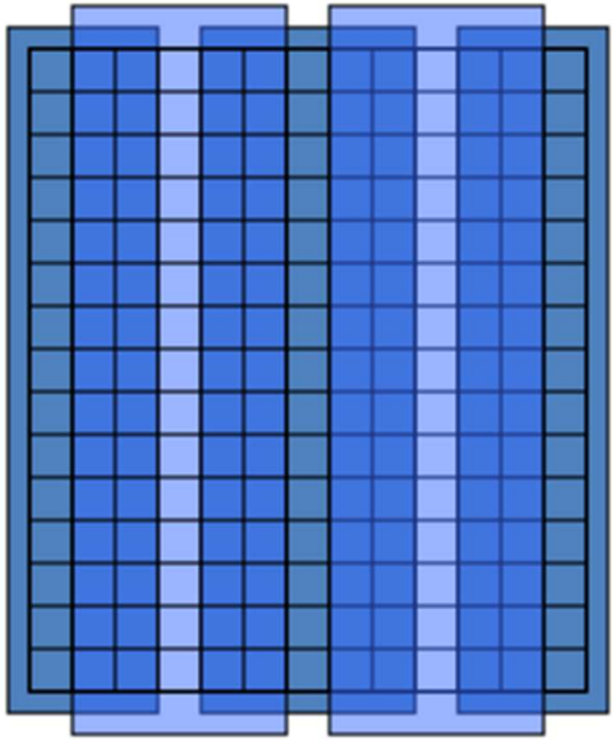
BSP模型算法示例

- 对数据进行分区实现并行

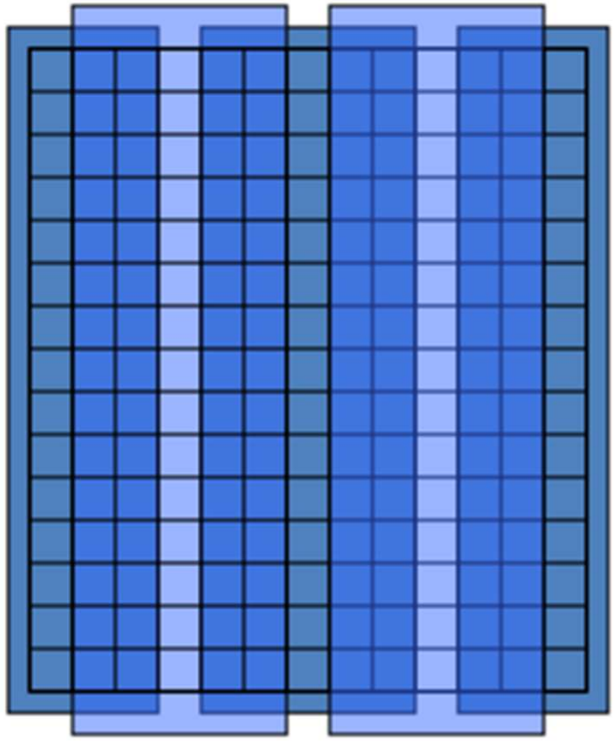


BSP模型算法示例

- **The approach to make it parallel is by partitioning the data**
  - Overlapping the data boundaries allow computation without communication for each superstep
  - On the communication step each processor update the corresponding columns on the remote processors.



BSP模型算法示例



```
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模型算法示例

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

h = max number of messages

= N values to the left +

N values to the right

= 2 * N (ignoring the inverse communication!)

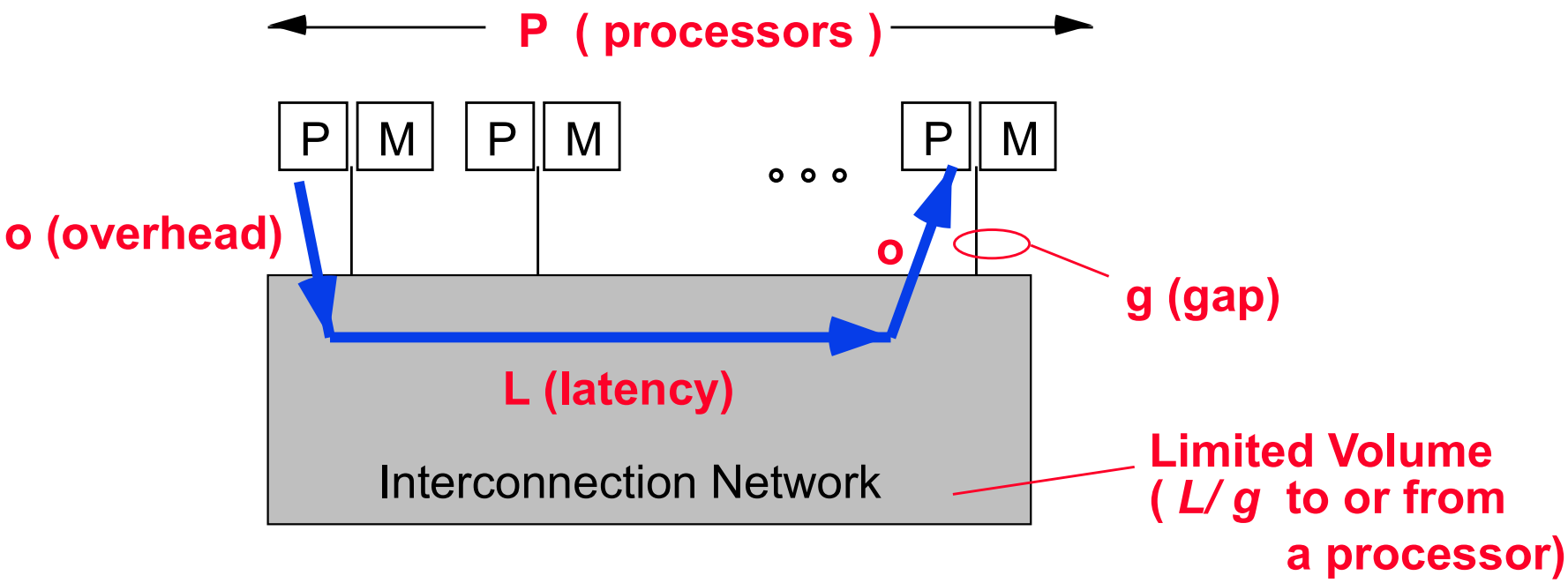
w = 4 * N * N / p

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

LogP模型

- **由Culler (1993, Berkeley) 提出，是一种分布存储的、点到点通信的多处理机模型，模型分为处理和通信两部分**
  - PRAM和BSP模型均未考虑分布存储的情况，也未考虑通信同步等的实际开销（BSP虽然引入了通信，但是只考虑一个带宽因子g）
- **处理**
  - 建模处理器, 大容量内存, cache等 => P
- **通信**
  - 建模网络通信延迟(100’s of cycles) => L
  - 建模受限的网络通信带宽(5%-10% of memory) => g
  - 建模通信启动开销(10's – 100's of cycles). => o
    - 在通信的两端都存在
    - 不和拓扑结构相关联
    - 不和具体架构相关联

LogP模型



- Latency in sending a (small) mesage between modules
- overhead felt by the processor on sending or receiving message
- gap between successive sends or receives (1/BW)
- Processors

LogP模型

- 一条消息从处理器 A 发送到处理器 B 的时间是:
  - L+2*o
- 有限的通信容量：同时支持不超过L/g条消息
- L、o、g共同刻画了通信网络的特性，但却又屏蔽了网络拓扑、路由算法和通信协议等具体细节
- 与具体的编程模型（共享存储、消息传递、数据并行）无关

- 思考：LogP模型是否支持通过多线程来隐藏网络通信延迟?

LogP vs. PRAM

- **思考：LogP模型能否表示PRAM模型?**
  - g=0, L=0, o=0时，LogP和PRAM什么关系?

LogP vs. BSP

- 思考：LogP和BSP模型有什么关系?
- BSP采用整体大同步，简化了算法设计，但是牺牲了运行时间
- 一种改进办法：采用子集同步。将所有线程按快慢程度分成若干个子集，子集内barrier同步
- 当子集小到只包含一对发送/接收者时，就变成异步的点对点同步，这就是LogP模型了

BSP+Overhead-Barrier=LogP

是否可以用BSP去模拟LogP，或者用LogP去模拟BSP?

并行模型对比

属性 \ 模型	PRAM	BSP	LogP
计算模式	同步计算	异步计算	异步计算
同步方式	指令自动同步	Barrier同步	点对点同步
模型参数	单位时间步	计算、带宽、同步	计算、通信（延迟、开销、带宽）
计算粒度	细粒度/中粒度	中粒度/粗粒度	中粒度/粗粒度
通信方式	读/写共享变量	发送/接收消息	发送/接收消息
地址空间	全局地址空间	单地址/多地址空间	单地址/多地址空间

总结

- **并行计算性能与可扩展性**
- **并行计算的算法模型和评价方法**
  - 使用Operation-Operand Graph描述并行算法
  - 阿姆达尔定律 Amdahl’ s law
  - 古斯塔夫森定律 Gustafson’ s Law
  - 孙倪定律 Sun-Ni’ s law
  - 等效法则 Isoefficiency Law
- **并行计算的系统模型和评价方法**
  - PRAM模型
  - BSP模型
  - LogP模型