



第8章 并行计算模型

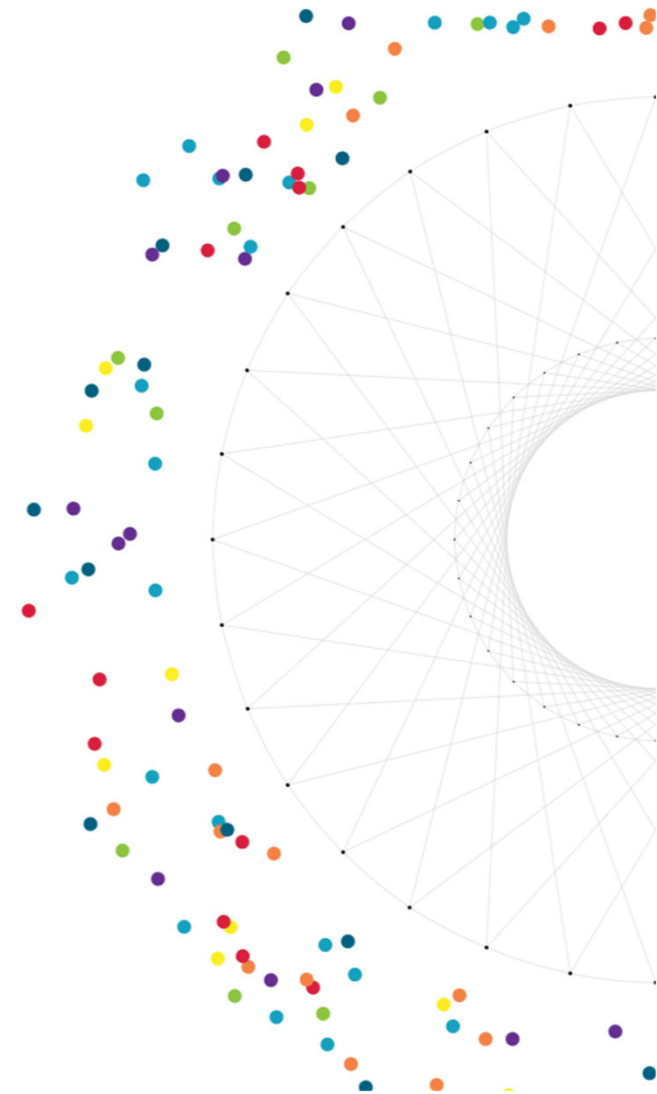
授课教师：胡俊成

jchu@jlu.edu.cn

Outline

1 并行算法基础

2 并行计算模型



并行算法的定义

- 算法：算法是解题方法的精确描述，是一组有穷的规则，它们规定了解决某一类特定类型问题的一系列运算。
- 并行算法：适合于在各种并行计算机上求解问题和处理数据的算法，它是一些可同时执行的诸进程的集合，这些进程相互作用和协调动作从而达到对给定问题的求解。

并行算法的分类

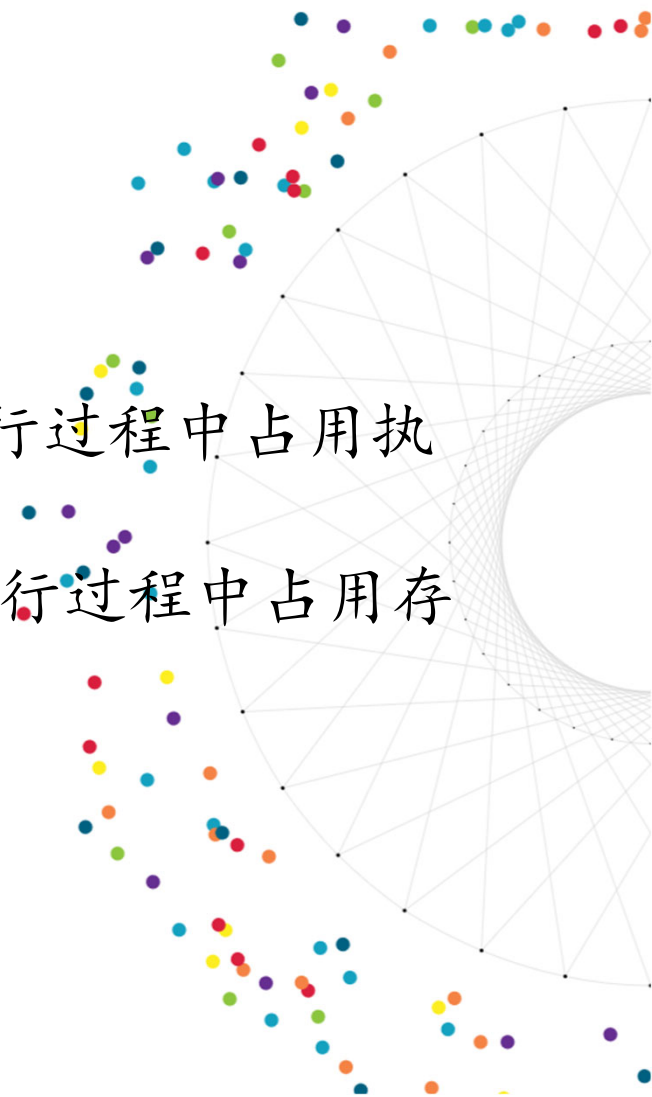
- 数值计算 (Numerical Computation) : 基于代数关系的一类运算。如矩阵运算, 多项式求解。
- 非数值计算 (Non-Numerical Computation) : 基于比较关系的一类运算。如排序、选择、搜索、匹配、图论。
- 同步运算 (Synchronized) : 某些进程的执行必须等待别的进程 (结果) 的一类运算。
- 异步运算 (Asynchronized) : 某些进程的执行不必等待别的进程 (结果) 的一类运算。
- 确定运算 (Deterministic) : 算法的每一步都能明确的指明下一步应该如何行进。
- 随机运算 (Randomized) : 算法的某一步随机的从指定范围内选取若干参数, 由其来确定下一步操作。

算法的复杂度

- 算法的**时间复杂度**和**空间复杂度**合称为算法的复杂度。对于同一个算法，时间复杂度和空间复杂度往往是相互影响的。当追求一个较好的时间复杂度时，可能会使空间复杂度的性能变差，即可能导致占用较多的存储空间；反之，当追求一个较好的空间复杂度时，可能会使时间复杂度的性能变差，即可能导致占用较长的运行时间。

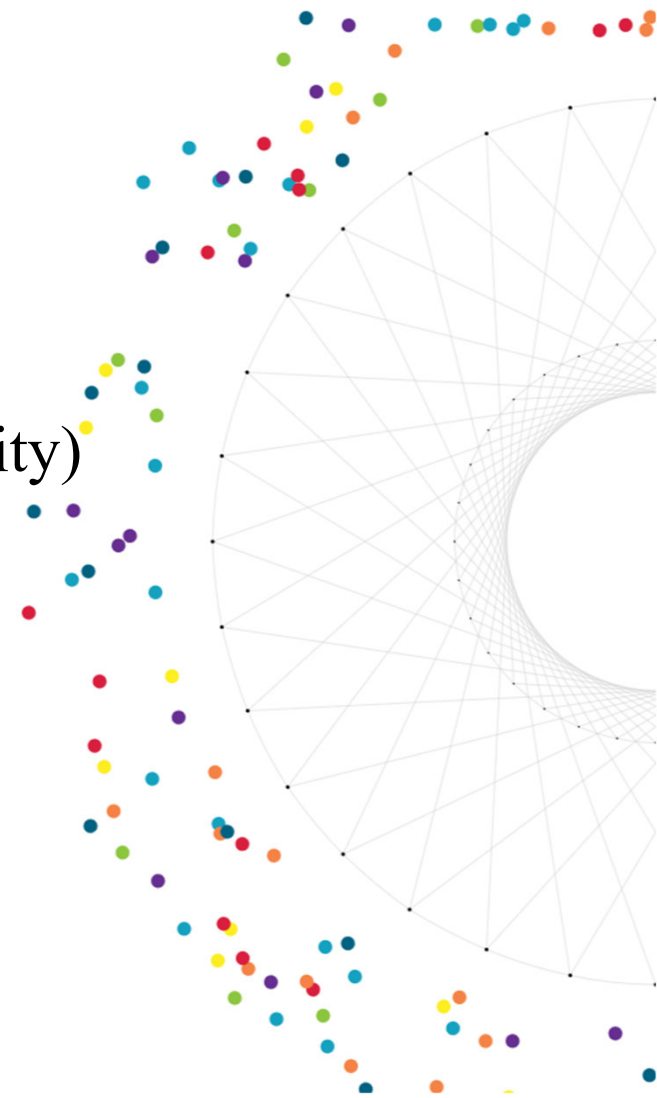
算法的复杂度

- n 问题规模, 求解函数 $f(n)$
- 时间复杂度(Time Complexity)是对一个算法在运行过程中占用执行时间大小的量度, 记做 $T(n)=O(f(n))$;
- 空间复杂度(Space Complexity)是对一个算法在运行过程中占用存储空间大小的量度, 记做 $S(n)=O(f(n))$;



串行算法的复杂性度量

- 最坏情况下的复杂度(Worst-CASE Complexity)
- 期望复杂度(Expected Complexity)

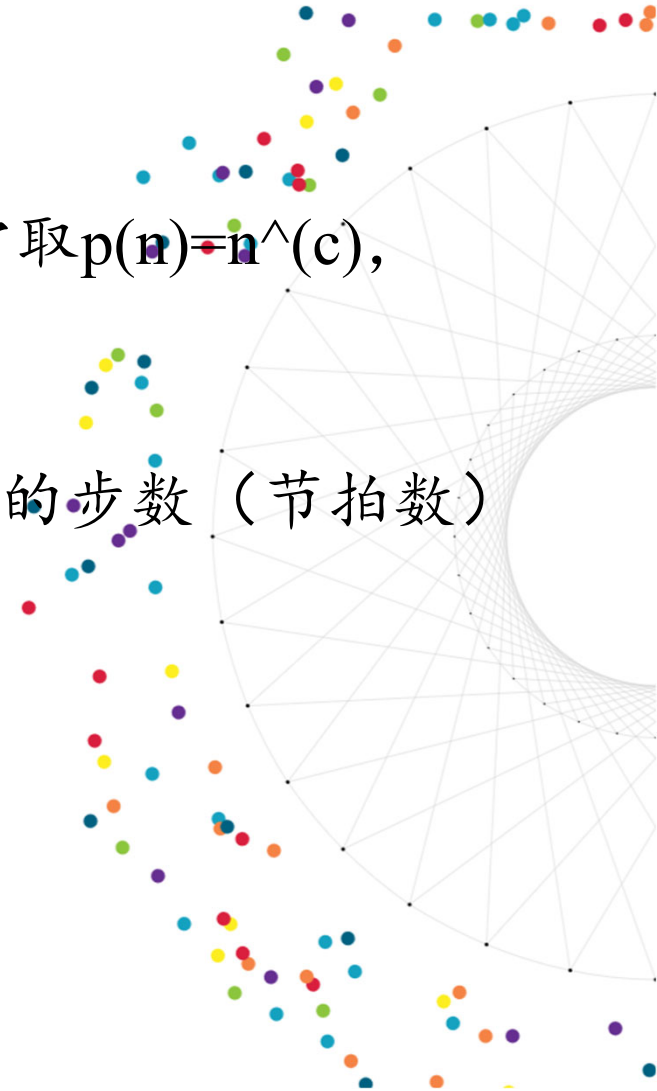


并行算法的复杂性度量指标1

- 运行时间 $t(n)$:在给定的模型上求解输入规模为 n 的给定问题所需时间,即算法从第一台处理机开始执行到最后一台处理机执行中止所需时间。
- $t(n)$ 包括两个部分:
- 计算时间 t_c :在某一处理器执行运算所需时间;
- 通信时间 t_r :数据从源处理机到目的处理机所需传送时间,算法在整个执行期间能传送的报文总数称为通信复杂度。

并行算法的复杂性度量指标2

- 处理机数 $p(n)$:求解给定问题所需的处理机数量, 通常取 $p(n)=n^c$, $0 < c < 1$ 。
- 并行算法的成本 $c(n) = t(n) \times p(n)$
- 成本最优 (Cost Optimal) : $c(n) = t(n) \times p(n) = \text{串行计算的步数 (节拍数)}$
- 工作量最优: 功耗低、环保



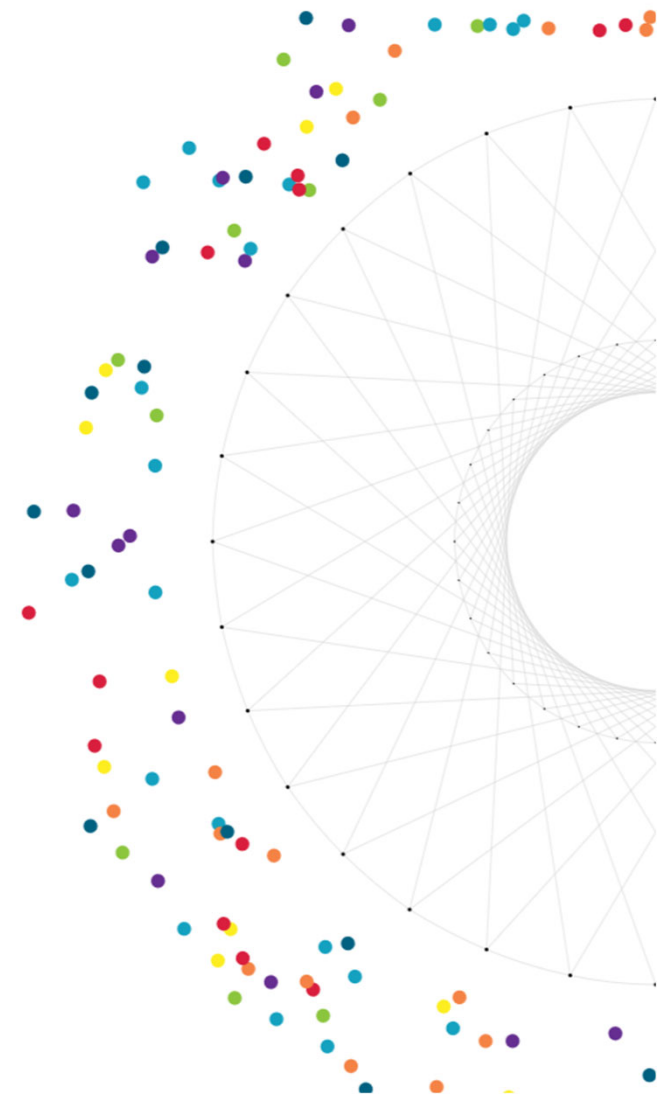
并行算法的复杂性度量指标3 (Brent定理)

- 令 $W(n)$ 是某并行算法A在运行时间 $T(n)$ 内所执行的运算量, 则A使用 p 台处理器可在 $t(n)=O(W(n)/p+T(n))$ 时间内执行完毕。
- $W(n)$ 和 $c(n)$ 密切相关: $c(n) = t(n) \times p = O(W(n)/p+T(n)) \times p = O(W(n) + p \times T(n))$, 因此当 $p=O(W(n)/T(n))$ 时, $c(n) = O(W(n))$, $W(n)$ 和 $c(n)$ 两者是渐近一致的;
- 推论: 对于任意的 p , $c(n) > W(n)$, 说明算法在运行过程中不一定能够充分利用处理器。

Outline

1 并行算法基础

2 并行计算模型



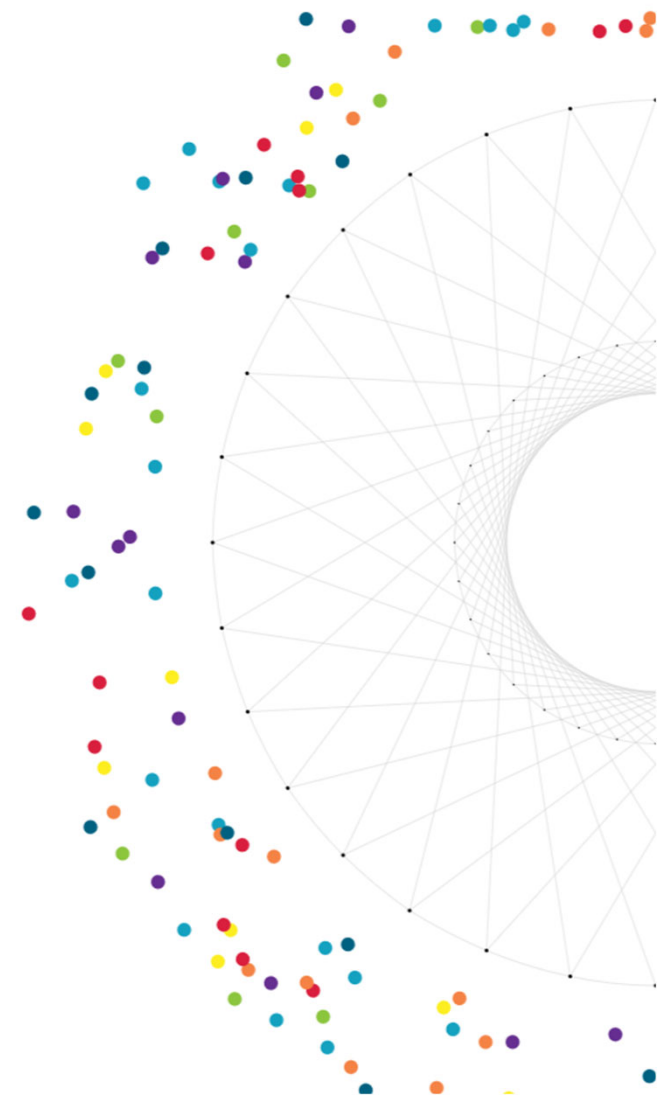
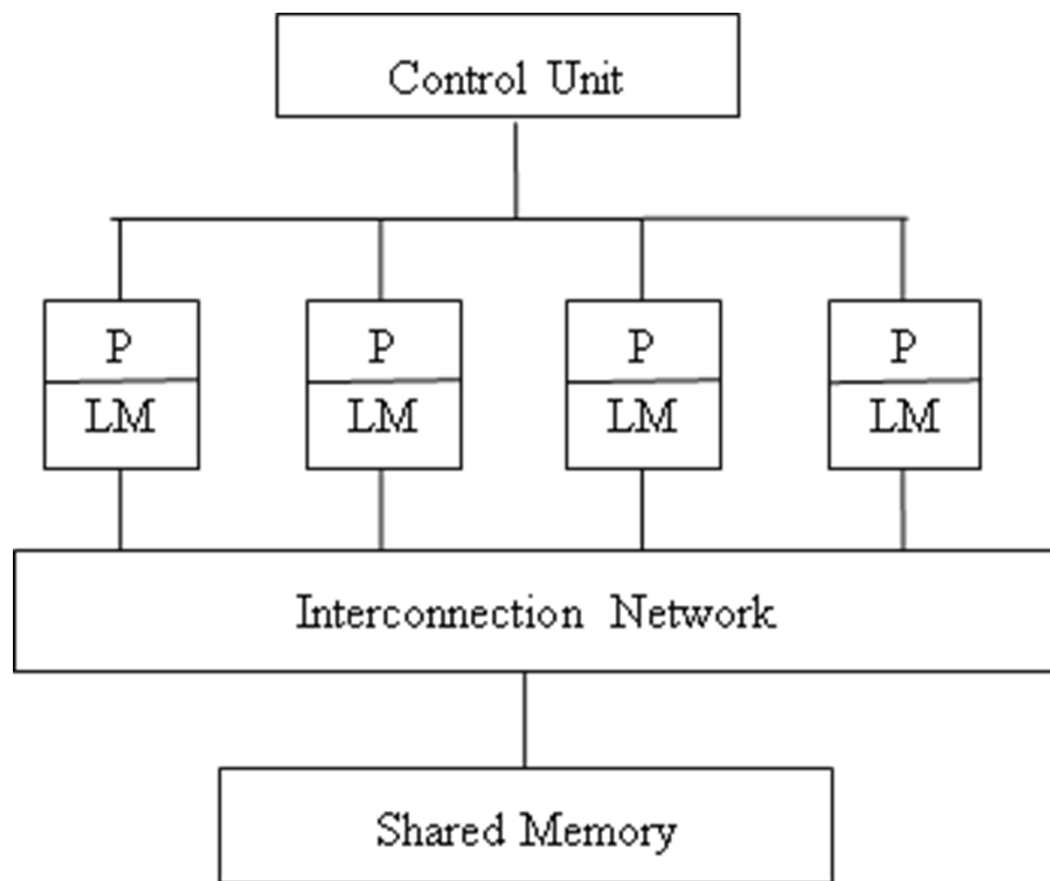
并行计算模型

- 并行计算模型通常指从并行算法的设计和分析出发，将各种并行计算机（至少某一类并行计算机）的基本特征抽象出来，形成一个抽象的计算模型。从更广的意义上说，**并行计算模型为并行计算提供了硬件和软件界面**，在该界面的约定下，并行系统硬件设计者和软件设计者可以开发对并行性的支持机制，从而提高系统的性能。

PRAM (SIMD-SM)模型

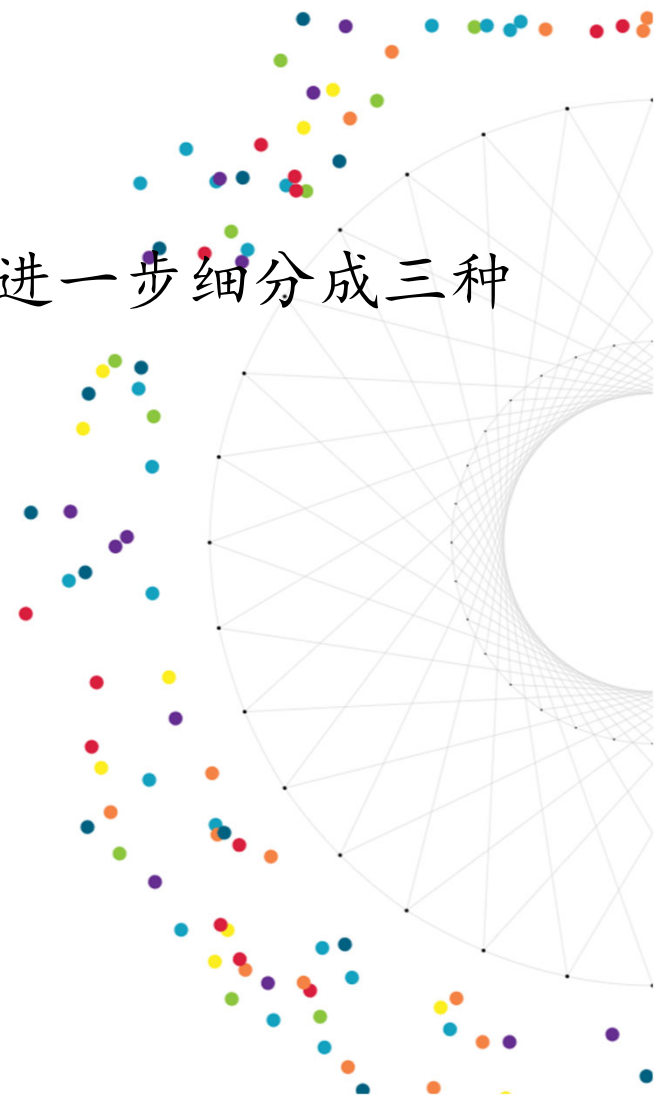
- 并行随机存取机器 (Parallel Random Access Machine, PRAM) 模型, 也称为 SIMD-SM模型 (共享存储的SIMD模型)。特点:
- (1) 一个集中的共享存储器, 假设其容量 M 无限大;
- (2) N 个功能相同的处理器, 每个处理器具有简单的算术运算和逻辑判断能力, 每个处理器都具有局部存储器 LM 。
- (3) 所有指令均按照锁步方式操作, 在每一个计算步内, 任一个处理器均可通过对共享存储器的某共享单元读写, 同其他任一处理器交换数据, 因此要求所有处理器共享单一的数据地址空间;
- (4) 每个处理器内部都没有控制器组件, 所有的处理器共享一个控制器, 因此要求所有处理器共享单一的程序地址空间;
- (5) 采用隐式同步机制, 诸如处理器间通讯、存储管理、进程同步、存储带宽和延迟等并行机的低级操作细节均隐含于模型中。

PRAM (SIMD-SM)模型结构图



PRAM (SIMD-SM)计算模式

- 根据各个处理器对共享存储器的竞争如何处理，又进一步细分成三种模式：
- PRAM-EREW互斥读互斥写（最弱）
- PRAM-CREW并发读互斥写（居中）
- PRAM-CRCW并发读并发写（最强）
- C代表Concurrent，允许并发操作
- E-代表Exclusive，禁止并发操作



PRAM (SIMD-SM) 复杂度分析:

- 同一个算法的运行时间

-

$$T_{EREW} \geq T_{CREW} \geq T_{CRCW}$$

$$T_{EREW} = O(T_{CREW} \cdot \log p) = O(T_{CRCW} \cdot \log p)$$

- 含义: 如果一个算法在PRAM-CREW或PRAM-CRCW上的时间复杂度为T, 则这个算法可以在PRAM-EREW上以时间复杂度为 $\log p \times T$ 运行。

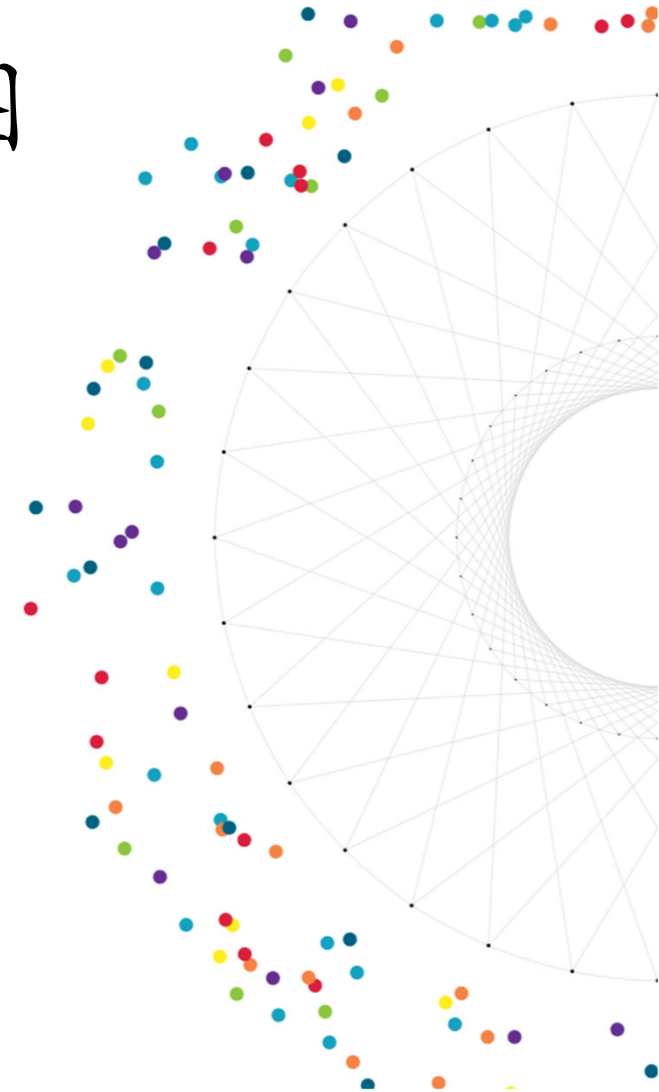
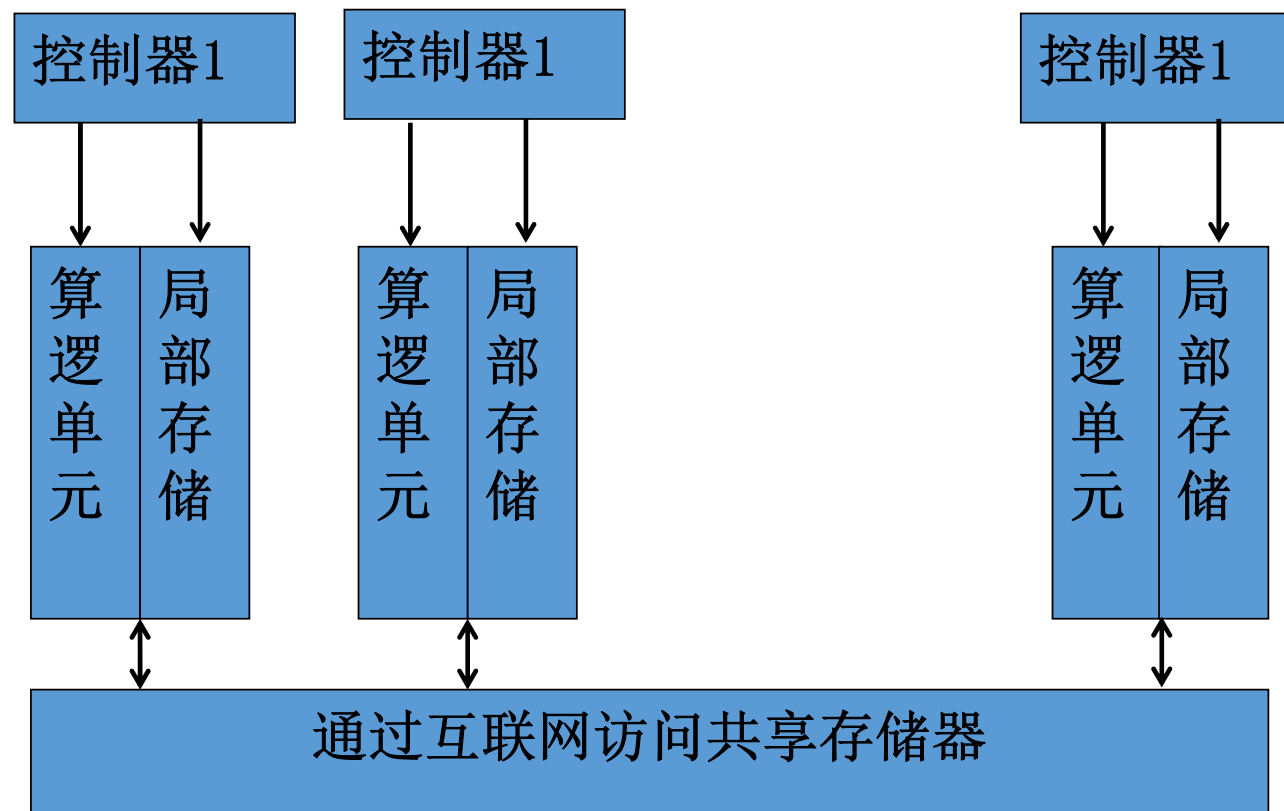
PRAM-CRCW的分类

- Common PRAM-CRCW: 仅允许所有处理器同时写入相同数据
- Priority PRAM-CRCW: 仅允许优先级最高的处理器写入
- Arbitrary PRAM-CRCW: 允许任意处理器自由写入

APRAM (MIMD-SM)模型

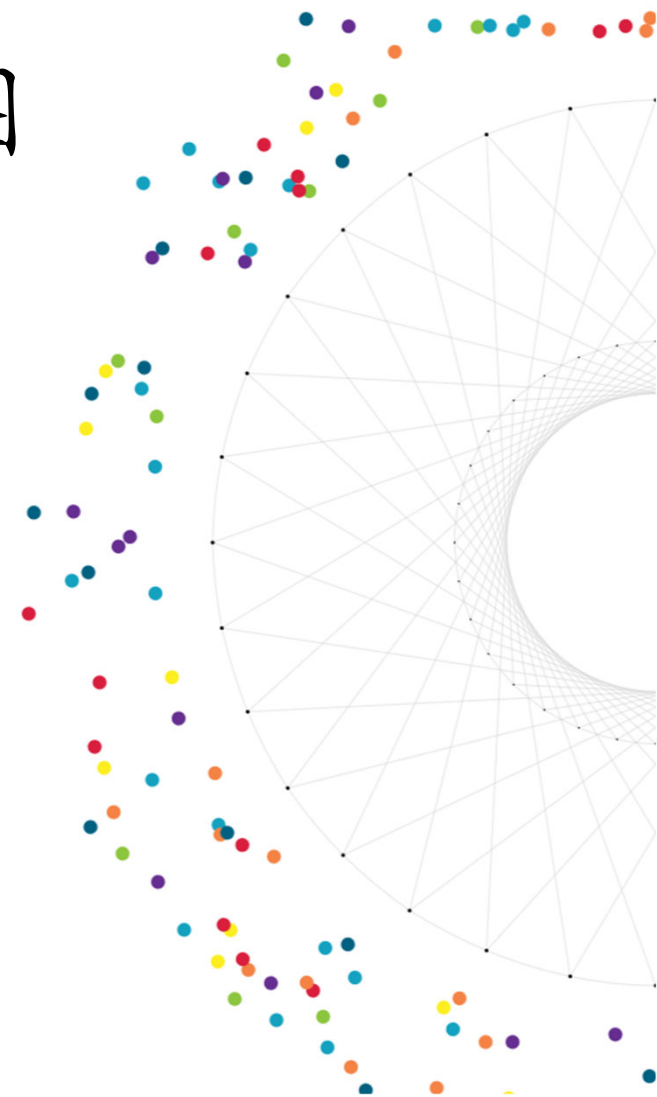
- 异步并行随机存取机器 (Asynchronous Parallel Random Access Machine, APrAM) 模型, 也称为MIMD-SM模型, 也称为分相 (Phase) PRAM模型。
特点:
- (1) 一个集中的共享存储器, 假设其容量M无限大;
- (2) N个处理器, 每个处理器内部, 有简单的算术运算和逻辑判断电路, 有独立的控制器, 有局部存储器LM, 有局部时钟;
- (3) 处理器之间的通信要通过共享存储器来完成, 因此要求所有处理器共享单一的数据地址空间;
- (4) 每个处理器内部都有独立的控制器组件, 因此要求所有处理器都有独立的程序地址空间;
- (5) 异步操作+显式同步。系统没有全局时钟, 每个处理器异步的执行局部程序, 处理器之间的时间依赖关系 (例如, 通过共享变量的读写操作实现通信等) 需要在各个处理器的局部程序中加入同步路障 (Synchronization Barrier) 来实现, 两次同步路障的时间间隔称为一个相 (phase)

APRAM (MIMD-SM)模型结构图



APRAM (MIMD-SM) 计算模式图

	处理器 1	处理器 2	...	处理器 p
	read x_1	read x_3	...	read x_n
phase1	read x_2	*		*
	*	write to B		*
	write to A	write to C		write to D
同步障	<hr/>			
	read B	read A		read C
phase2	*	*		*
	write to B	write to D		
同步障	<hr/>			
	*	write to C		write to B
	read D			read A
				write to B
同步障	<hr/>			



APRAM (MIMD-SM)计算模式分析

- (1) 整个计算过程系由一系列同步路障分开的多个全局相所组成。
- (2) 在每个全局相内，每个处理器异步的运行其局部程序；
- (3) 每个局部程序中的最后一条指令是一条同步路障指令；
- (4) 各个处理器均可异步地读取和写入全局存储器，但在同一个全局相内不允许两个处理器访问同一存储单元。
- (5) 程序内指令的分类：全局同步，全局读/写，局部操作*

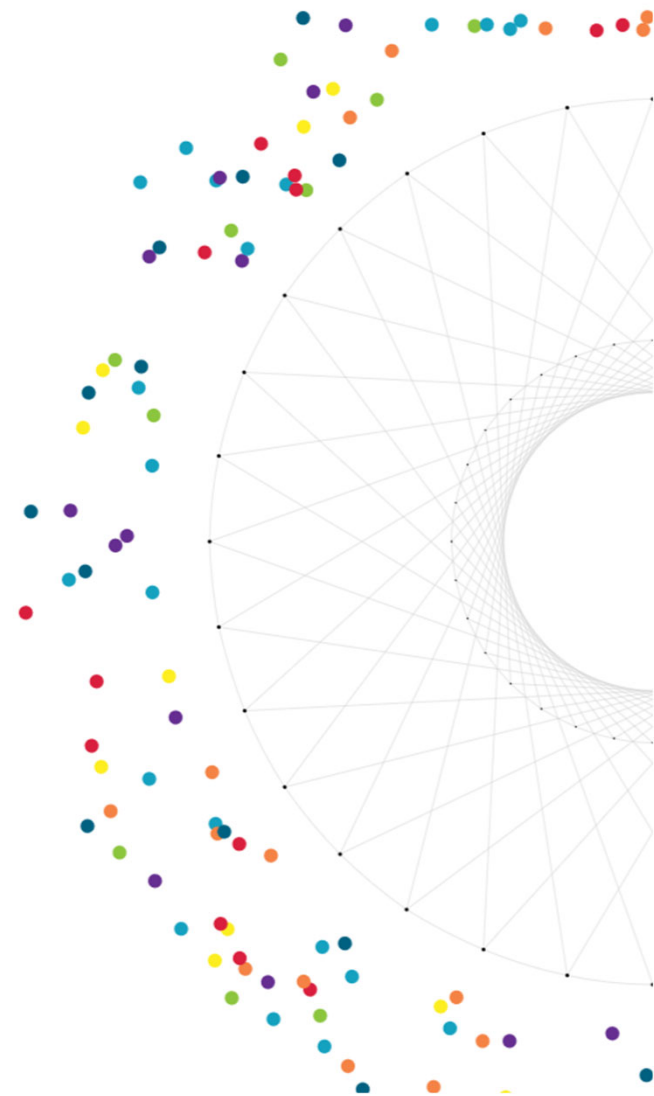
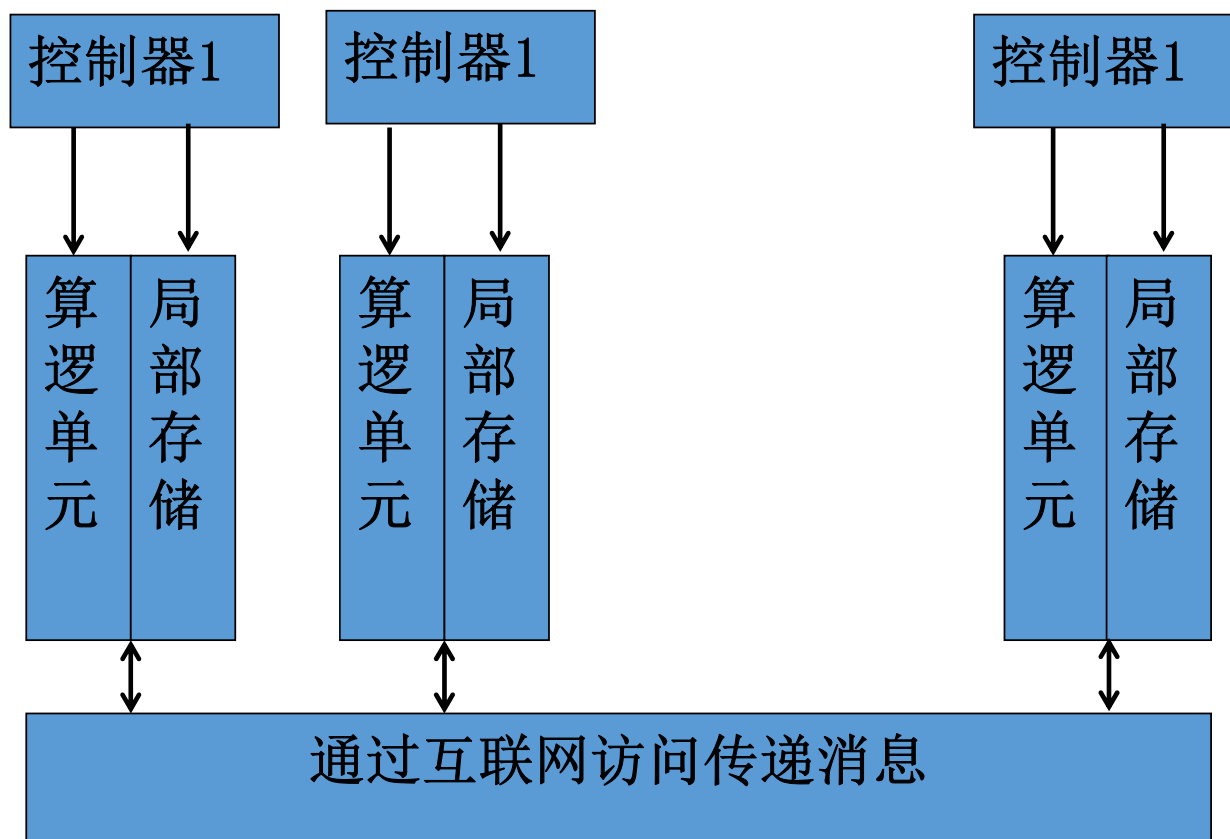
APRAM (MIMD-SM) 复杂度分析:

- 设局部操作为单位时间;
- 全局读/写平均时间为 d , d 随着处理器数目的增加而增加;
- 同步路障时间为 $B=B(p)$, 它是处理器数 p 的非降函数, 通常处理器越多导致同步时间越长。
- 设 T_i 为第 i 个全局相内各处理器执行时间最长者;
- 则 $T=T_1+T_2+\dots+T_n+ B \times (n-1)$

BSP (MIMD-DM)模型

- 块同步 (Bulk Synchronization Parallel) 模型, 也称为MIMD-DM (distributed memory) 模型, 也称为大同步模型、**桥模型**, 桶同步并行模型。特点:
- (1) 系统中有 p 个计算节点, 每个计算节点内部有独立的运算器, 独立的控制器, 独立的存储器, 独立的局部时钟; 因此, 每个处理器有独立的数据地址空间和程序地址空间, 可以异步的执行局部程序;
- (2) 系统中有一个路由器, 该路由器连接所有计算节点, 实施计算节点之间点到点的消息传递, 该路由器的吞吐率和延迟是有限的, 吞吐率 th (字节每秒), 传输延迟 tl (秒);
- (3) 系统中有一个路障同步器, 路障同步器每隔时间间隔 L 完成一次全局路障同步操作, 以实现计算节点之间的时间依赖关系 (例如, 通过共享变量的读写操作实现通信等)。

BSP (MIMD-DM) 模型结构图



BSP (MIMD-DM) 计算模式图

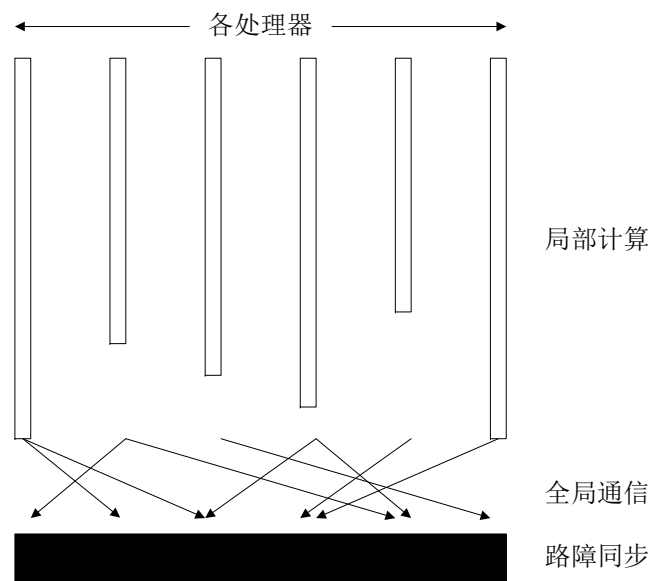
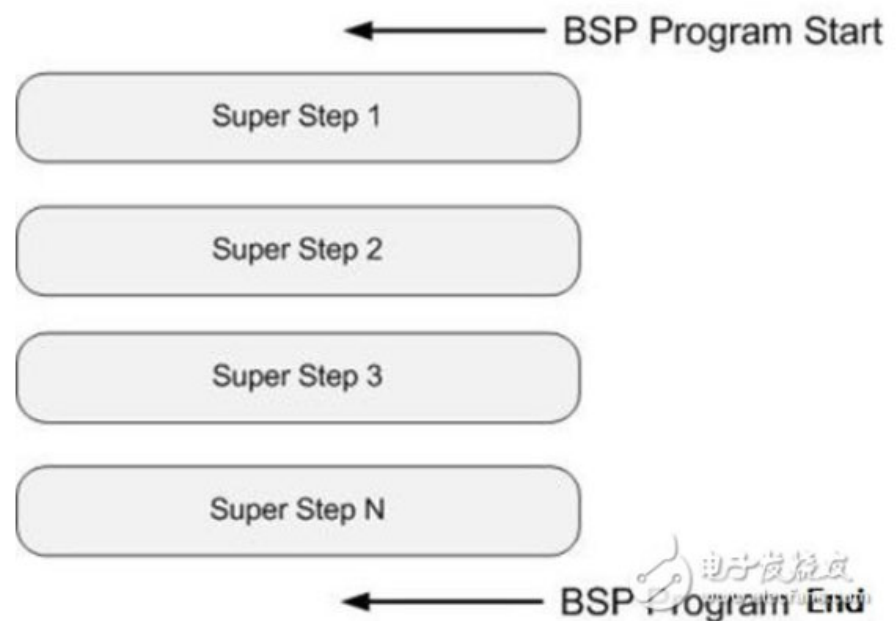


图4.3

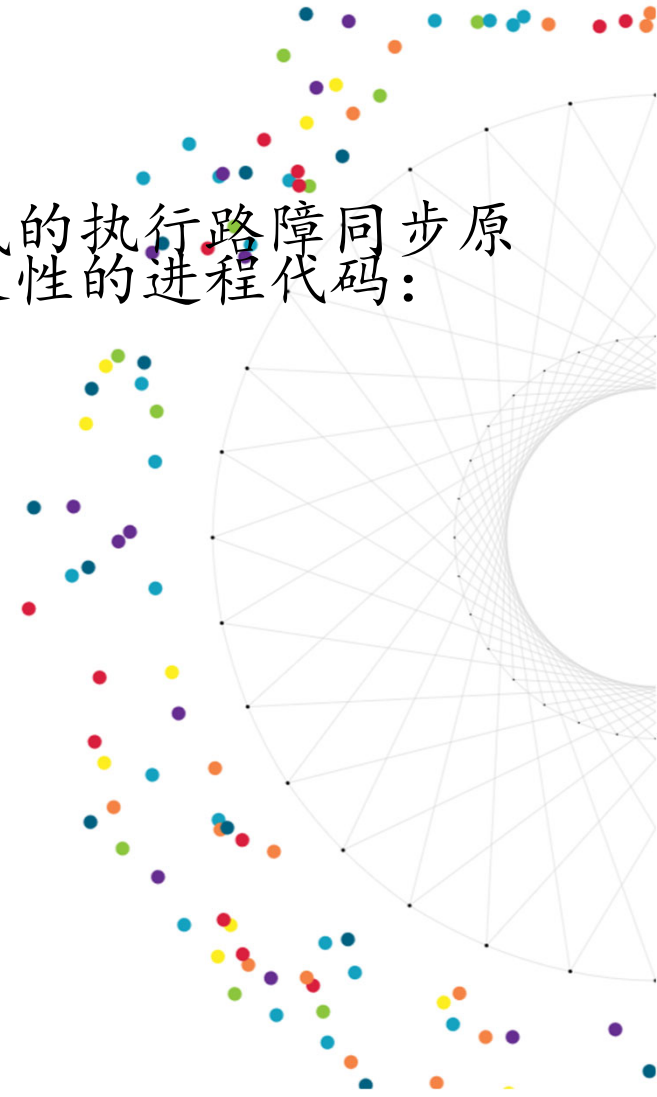


BSP (MIMD-DM) 计算模式分析

- (1) 整个计算过程由一系列用全局同步操作分开的、同步周期为 L 的多个超级步 (super step) 组成;
- (2) 在每个超级步内, p 个计算节点异步的运行其局部程序, 并通过路由器接收和发送消息;
- (3) 路由器仅施行点到点的消息传递, 不能提供组播、广播这类的功能, 路由器的消息收发能力为, h 为每个处理器至多发送或接收 h 条消息/超级步 (称为 h -relation), 如果某个超级步内某节点收发信息的总量超过 h 个, 则会导致路由器拥塞, 因此要将该超级步扩展成多个超级步; 当前超级步中接收的消息, 在下一个超级步中的计算中才可以使用;
- (4) 当每个超级步的时间片 L 用完时, 路由器会发起一次全部路障同步检查, 如果所有节点都已经完成本超级步内要做的操作, 则打开路障, 让所有节点一起进入下一个超级步; 如果有节点还没完成, 则保持路障, 让所有节点等待, 让所有还没到达路障位置的节点再获得时间片 L 。

BSP (MIMD-DM) 模型实现

- 实际实现中，每个超步结束后由各个计算节点显式的执行路障同步原语完成全局同步，来代替周期性的同步检测。代表性的进程代码：
-
- ... //局部程序代码
- Send(source,target,message);
- Receive(source,target,message);
- Global-Syn(k); //第k次全局路障同步
-
- ... //局部程序代码
- Send(source,target,message);
- Receive(source,target,message);
- Global-Syn(k+1); //第k+1次全局路障同步



BSP (MIMD-DM) 复杂度分析:

- 设 W_k 为第 k 个超级步内各节点局部程序中本地计算运行时间最长者;
- 设 H_k 为第 k 个超级步内各节点局部程序中收发消息个数最多者的收发操作执行时间;
- 设每次全局同步路障的开销为 B ;
- 设每个数据包大小 q 个字节;
- 则 $H_k = h * (t_l \text{ (传输延迟)} + q / t_h \text{ (吞吐率)})$
- 一个超级步内的总时间开销为 $T_k = W_k + H_k + B$; (T_k 需要小于等于 L , 效率为 T_k / L)
- s 个超级步内的总时间开销为 $T = W_1 + W_2 + \dots + W_s + H_1 + H_2 + \dots + H_s + s \times B$; (T 需要小于等于 $s * L$, 效率为 $T / (s * L)$)

BSP (MIMD-DM) 模型小结

- 有效避免死锁 (计算划分为超级步)
- 延迟通信
 - BSP把所有的计算和通信视为一个整体行为, 而不是一个单独的进程的个体活动; 它让每个进程尽可能的将超级步内的所有消息收发尽量押后, 组成一个尽可能大的通信, 其好处是延迟的通信能够提供更多优化通信的机会, 其坏处是延迟发送就意味着延迟接收;
- 同步开销
 - BSP需要依赖路障同步器, 通过硬件方式实现横跨所有节点的路障同步器其实现成本过大, 通过软件方式实现同步库函数, 其全部同步开销随着节点个数增加而增大;
- 硬件气泡
 - 节点执行计算操作时网络资源空闲, 节点执行消息收发时计算资源空闲, 到达同步路障的快速进程必须在路障处等待慢速进程造成节点空转; 可以使用节点内多进程/多线程来填充气泡
- Tradeoff
 - 编程的结构性v.s.资源的有效性

BSP与MapReduce对比

- 执行机制：MapReduce是一个数据流模型，每个任务只是对输入数据进行处理，产生的输出数据作为另一个任务的输入数据，并行任务之间独立地进行，串行任务之间以磁盘和数据复制作为交换介质和接口。
- BSP是一个状态模型，各个子任务在本地的子图数据上进行计算、通信、修改图的状态等操作，并行任务之间通过消息通信交流中间计算结果，不需要像MapReduce那样对全体数据进行复制。
- MapReduce的设计初衷：解决大规模、非实时数据处理问题。“大规模”决定数据有局部性特性可利用（从而可以划分）、可以批处理；“非实时”代表响应时间可以较长，有充分的时间执行程序。而BSP模型在实时处理有优异的表现。这是两者最大的一个区别。

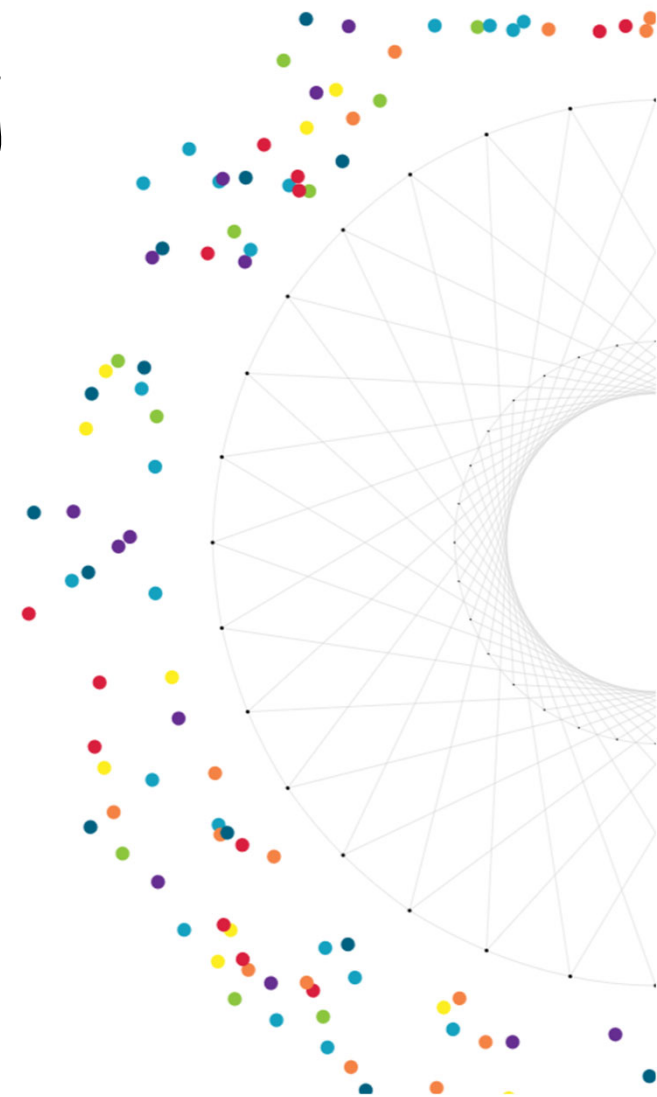
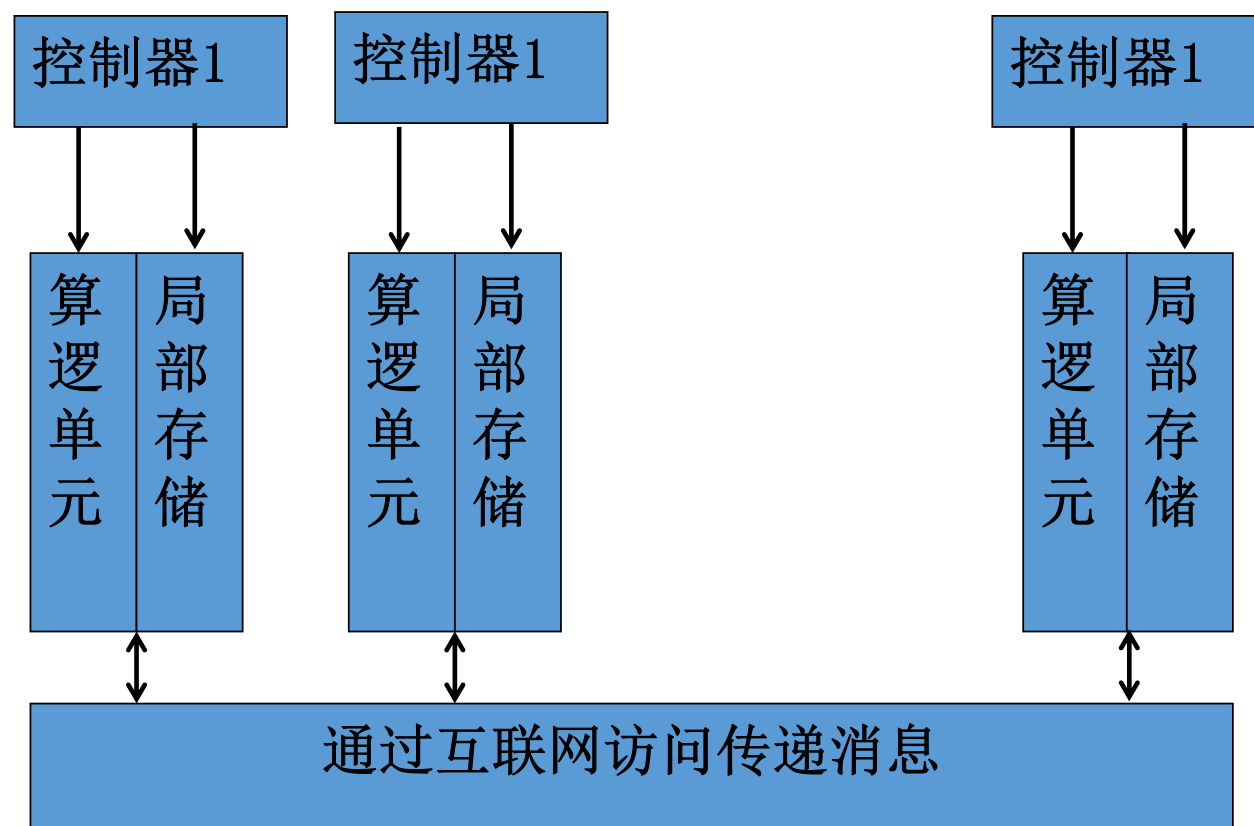
LogP (MIMD-DM) 模型

- LogP是使用L,O,G,P四个参数来描述的一种面向分布式存储器、点对点通信的多计算机系统的并行计算模型。
- L (Latency) : 表示信息从源节点到目的节点所需的时间;
- O (Overhead) : 表示节点接收或发送一条消息所需额外开销 (操作系统开销和网络软件开销), 并且在此期间节点不能做任何操作;
- G (Gap): 表示节点连续进行两次发送或接收消息之间必须有的时间间隔, 超过此间隔则数据包会因拥塞而被网络丢弃, 因此实际上此参数是对网络的限制而非对节点的限制;
- P (Processor) : 表示节点的数目。

LogP (MIMD-DM) 模型的特点

- (1) 系统中有 p 个计算节点，每个计算节点内部有独立的运算器，独立的控制器，独立的存储器，独立的局部时钟；因此，每个处理器有独立的数据地址空间和程序地址空间，可以异步的执行局部程序；
- (2) 系统中有一个大型互连网络（多个路由器节点组成的数据交换阵列），该互连网络连接所有节点，实施节点之间点到点的消息传递，该互连网络的带宽和延迟是有限的，带宽为 g 的倒数（字节每秒），传输延迟 L （秒）；
- (3) 通过节点间的个体消息传递实现同步操作，一旦消息到达了处理器就可以立即使用。

LogP (MIMD-DM) 模型结构图



LogP (MIMD-DM) 模型实现

- 实际实现中，由各个计算节点显式的执行通信原语完成局部同步，来代替周期性的同步检测。代表性的进程代码：
- ... //局部程序代码
- Send(source,target,message);
- Receive(source,target,message);
- ... //局部程序代码
- Receive(source,target,message);
- Send(source,target,message);
- ... //局部程序代码



LogP=BSP+Overhead-GlobalBarrier

- 以BSP为基础，将所有并行的进程按照快、慢程度分成若干个子集，以每个子集内的小规模子集同步代替全局大同步，如果子集小到每个子集只包含两个进程（一个发送，一个接收），则BSP就变成了LogP。

LogP模型小结

- 即时通信

- LogP不再把所有的计算和通信视为一个整体行为，而是一个单独的进程的个体活动；它让每个进程按需排布计算操作和消息收发，一旦收到消息立即可以使用；

- 同步开销

- LogP不需要路障同步器，将路障同步的开销降低为两个进程之间的信息收发时间差；

- 硬件气泡

- 节点执行计算操作时网络资源空闲，节点执行消息收发时计算资源空闲，非阻塞进程不必等待阻塞进程避免了节点空转；可使用节点内多进程/多线程来填充气泡

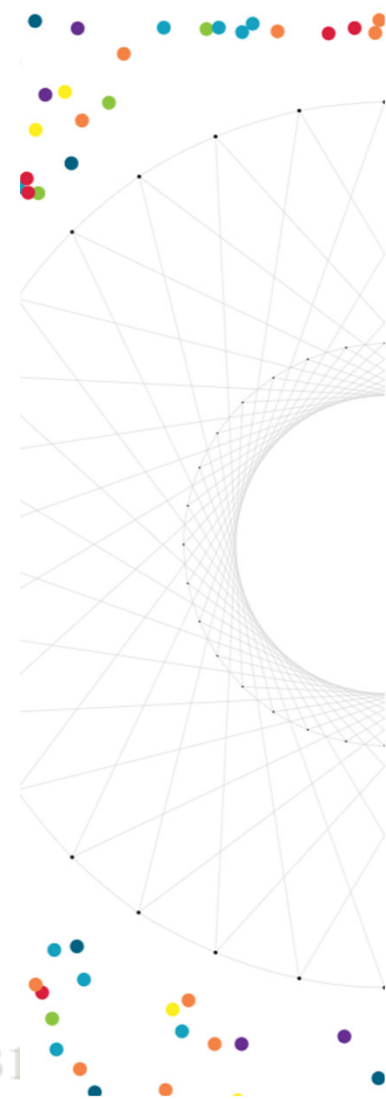
- Tradeoff

- 编程的随意性v.s.性能的可预测性

LogP (MIMD-DM) 复杂度分析:

- LogP的实质是将整个互连网络看做一个流水线部件，其启动率为 g ，延迟为 L ，端点处理器开销为零；
- 网络的容量假定是有限的，在任何时刻从源处理器到目的处理器的网络路径上，最多只能容纳 L/g 条消息；
- 点到点传输一条消息的时间为 $o+L+o$ ；
- 当 $g=0$ ， $L=0$ ， $o=0$ 时，LogP等同于PRAM；
- LogP和BSP已经被证明是等效的，可以相互模拟：用BSP模拟LogP通常会慢常数倍；用LogP模拟BSP通常会慢对数倍。
- SM可以用DM模拟

属性 \ 模型	PRAM	APRAM	BSP	logP
体系结构	SIMD-SM	MIMD-SM	MIMD-DM	MIMD-DM
计算模式	同步计算	异步计算	异步计算	异步计算
同步方式	自动同步	路障同步	路障同步	隐式同步
模型参数	1 (单位时间步)	d, B d: 读/写时间 B: 同步时间	P, g, l P: 处理器数 g: 带宽因子 l: 同步间隔	L, o, g, p L: 通信延迟 o: 额外开销 g: 带宽因子 p: 处理器数
计算粒度	细粒度/ 中粒度	中粒度/ 大粒度	中粒度/ 大粒度	中粒度/ 大粒度
通信方式	读/写共 享变量	读/写共 享变量	发送/接收 消息	发送/接收 消息
编程地址空间	全局地址空间	单一地址空间	单地址/多 地址空间	单地址/多 地址空间

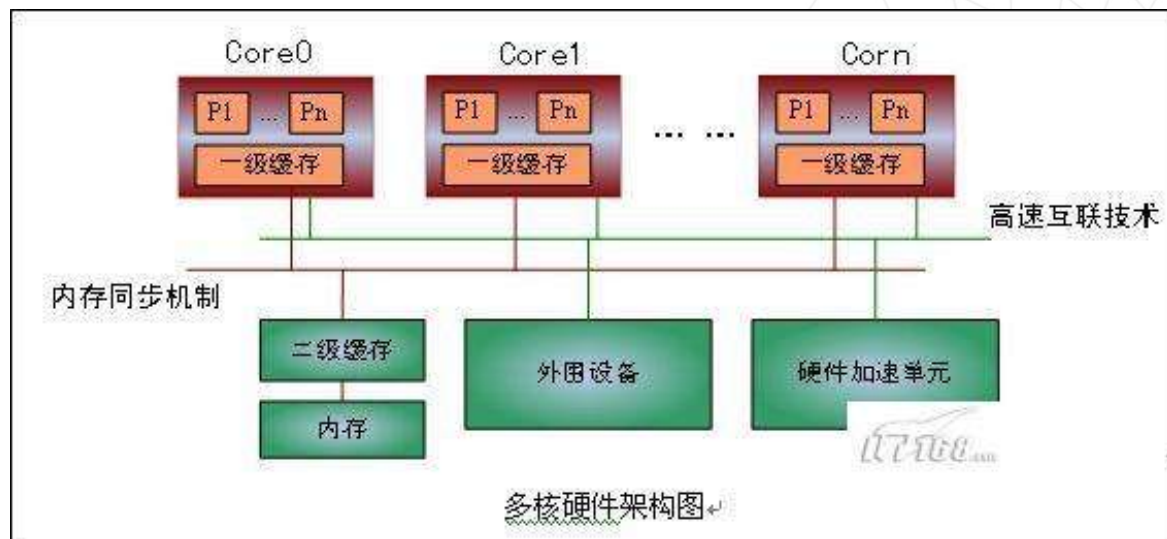


Multi-BSP (多层架构)模型

在多核架构中，有许多个处理器，多个本地cache，多个存储器。如图所示。由于多核架构的产生，导致并行算法的产生。并行算法在设计甚至实施时会有很多的难度。例如：

1. 并行底层的算法设计比串行底层的算法设计要复杂的多；
2. 设计出的并行算法要在性能上优于串行算法；
3. 并行算法设计结果可能只是简简单单的加多处理器的数量；
4. 由于硬件的缘故，很可能在本机上优化的并行算法，在另一台电脑上不起作用或者起反作用。

综上，如何创建和利用有效的多核算法是有很困难的。



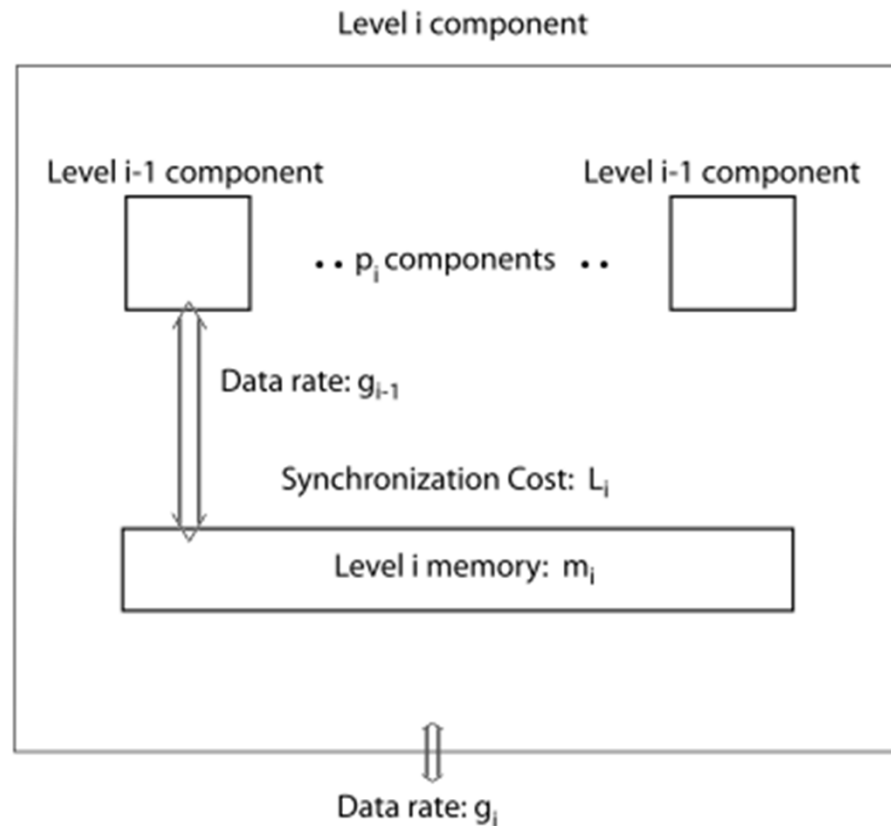
Multi-BSP (多层架构)模型的参数

- multi-BSP从两个方向拓展了BSP。
 - multi-BSP model是一个层次模型，它可以表示出单芯片或多芯片架构中的多个memory或多个cache。
 - 在每一层中，multi-BSP都将存储大小作为一个参数。
- Multi-BSP model实际上是一个深度为D的树，内部节点表示memory和cache，叶子节点表示处理器，每一层有四个参数 (p_i, g_i, L_i, m_i) ，
 - p_i 表示子组件的数量，
 - g_i 表示通信带宽，
 - L_i 表示同步成本，
 - m_i 表示memory或cache的大小。

Multi-BSP (多层架构)模型的作用

- Multi-BSP是一个综合的计算模型，它具有同步以及计算和通信的机制。
- Multi-BSP的定量计算证明，对于诸如矩阵乘法，快速傅立叶变换和比较排序的问题，即使有许多参数，算法也可以在近似无参数的意义上进行最优化。

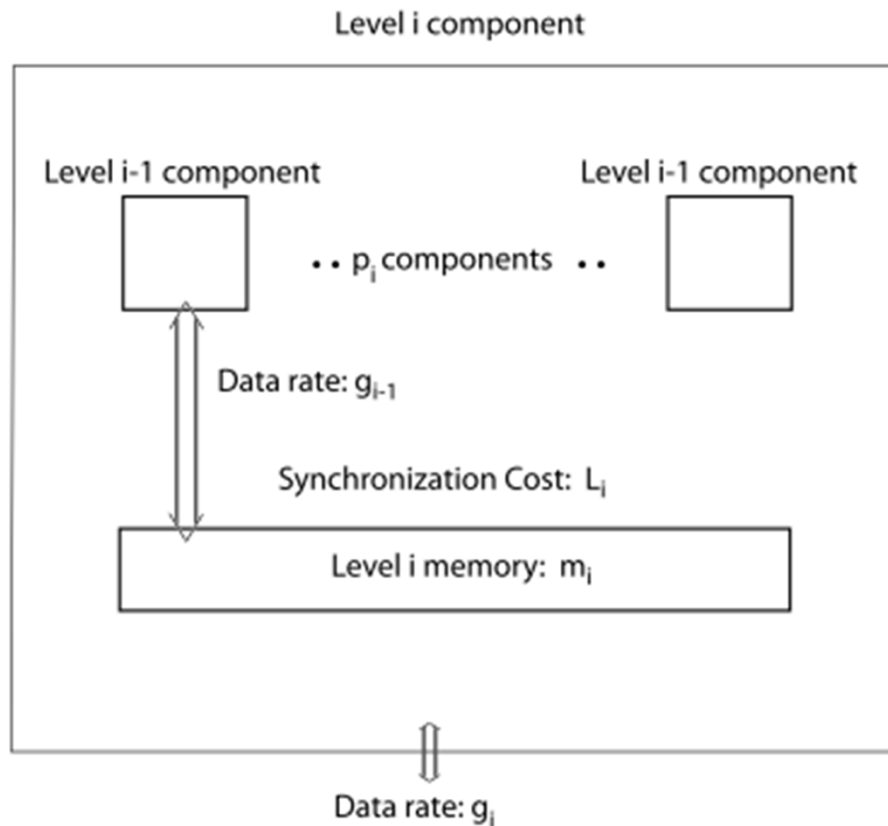
Multi-BSP (多层架构)模型的架构



Multi-BSP 模型是树形结构，其中叶子部分为处理器，内部节点为存储信息，这个模型不区别memory和cache。Multi-BSP模型每个节点有四个参数 (p_i, g_i, L_i, m_i) 。

Fig. 1. Schematic diagram of a level i component of the Multi-BSP model in terms of level $i - 1$ components.

Multi-BSP (多层架构)模型的架构



p_i 是第 i 级(层)组件里的 $i-1$ 级处理器的数量, 对于 $i=1$ 的情况, 可以认为 p_1 表示的是假想的0级处理器的数量。

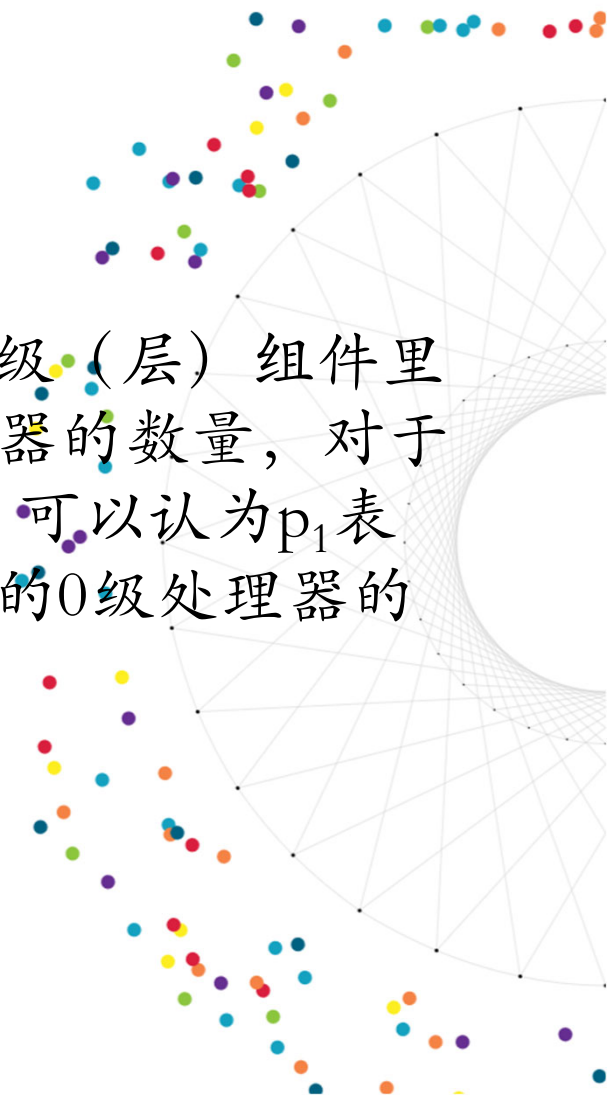
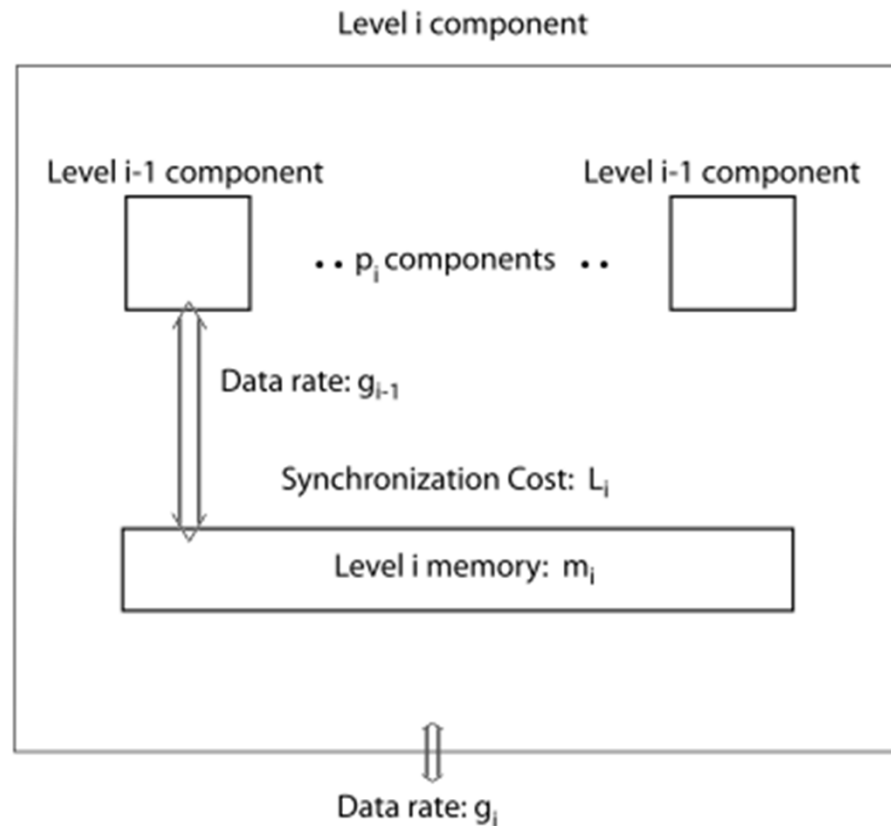


Fig. 1. Schematic diagram of a level i component of the Multi-BSP model in terms of level $i - 1$ components.

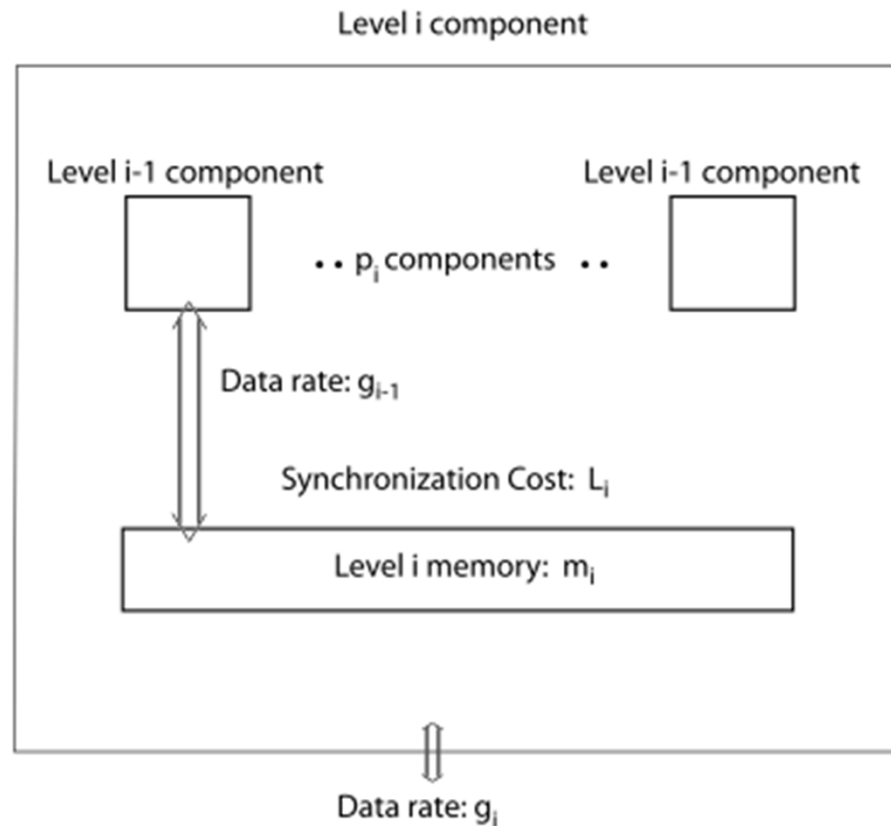
Multi-BSP (多层架构)模型的架构



g_i 表示通信带宽, $g_i = (\text{处理器在一秒内执行的操作次数}) / (\text{第}i\text{级和第}i+1\text{级存储器之间一秒内传输的字数})$ 这里的一个字是执行处理器操作的数据量 ($1\text{Word} = 2\text{Byte}$)。假设 $g_1=1$ 。

Fig. 1. Schematic diagram of a level i component of the Multi-BSP model in terms of level $i - 1$ components.

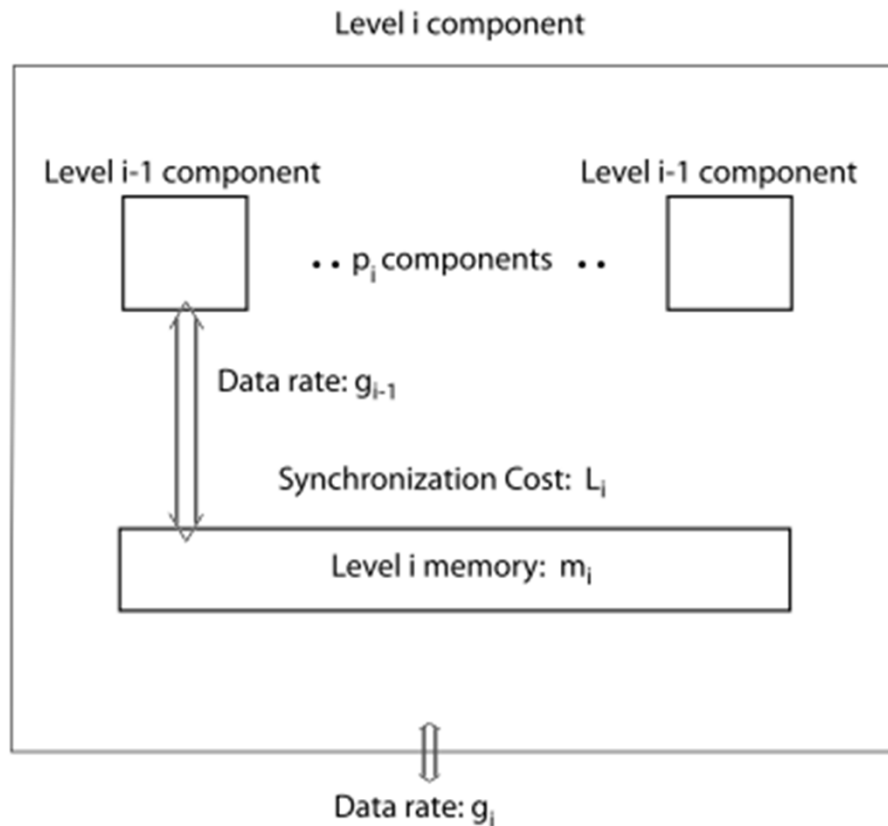
Multi-BSP (多层架构)模型的架构



第 i 层的superstep表示第 $i-1$ 层所有组件独立执行，直到到达同步栅栏。当 $i-1$ 层的组件全部到达barrier时，它们可以和第 i 层的存储器交换信息，通信成本通常由 mg_{i-1} 表示，其中 m 表示第 i 层组件的存储器与任何一个 $i-1$ 层子组件存储器之间传达的最大字数。然后下一层的superstep才开始执行。

Fig. 1. Schematic diagram of a level i component of the Multi-BSP model in terms of level $i - 1$ components.

Multi-BSP (多层架构)模型的架构



L_i 表示第 i 层superstep的栅栏同步的开销，这个栅栏表示的是组件与组件之间的，并不表示分支语句引起的。特别地， $L_1 = 0$ 。

m_i 表示第 i 层中存储器的数量。

Fig. 1. Schematic diagram of a level i component of the Multi-BSP model in terms of level $i - 1$ components.

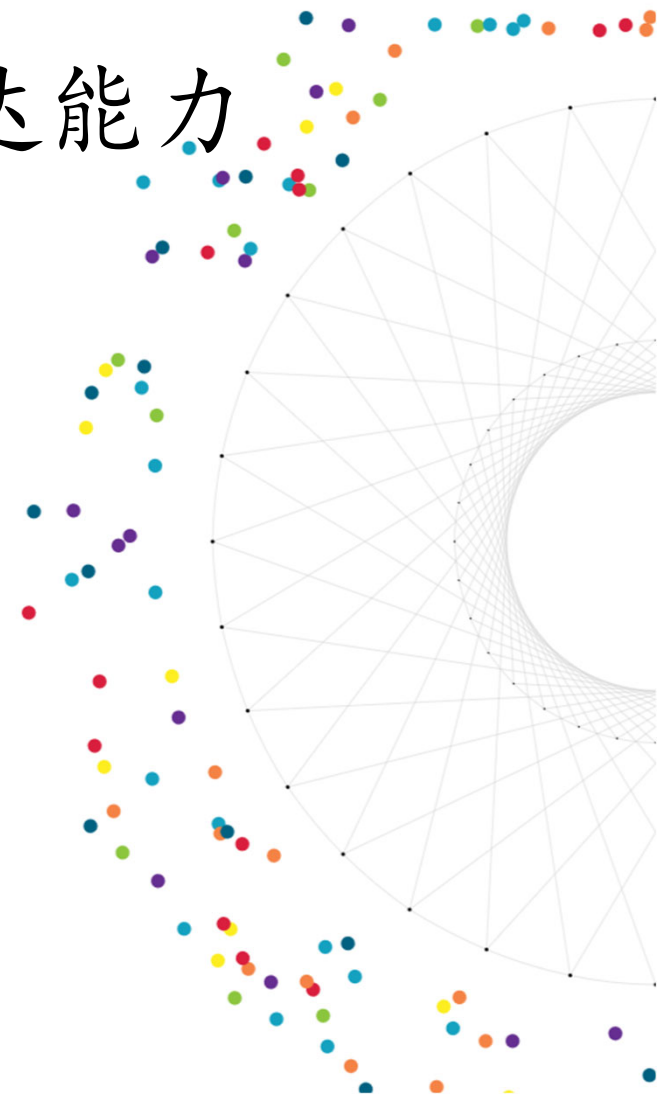
Multi-BSP (多层架构)模型的表达能力

冯·诺依曼模型 (d=1) : $(p_1 = 1, g_1 = \infty, L_1 = 0, m_1)$

PRAM模型 (d=1) : $(p_1 \geq 1, g_1 = \infty, L_1 = 0, m_1)$

BSP模型 (d=2) : $(p_1 = 1, g_1 = g, L_1 = 0, m_1 = m)$

$(p_2 = p, g_2 = \infty, L_2 = L, m_2)$



Multi-BSP (多层架构)模型的特征

第 i 层中节点的组件总数为 $P_i = p_1 \cdots p_i$

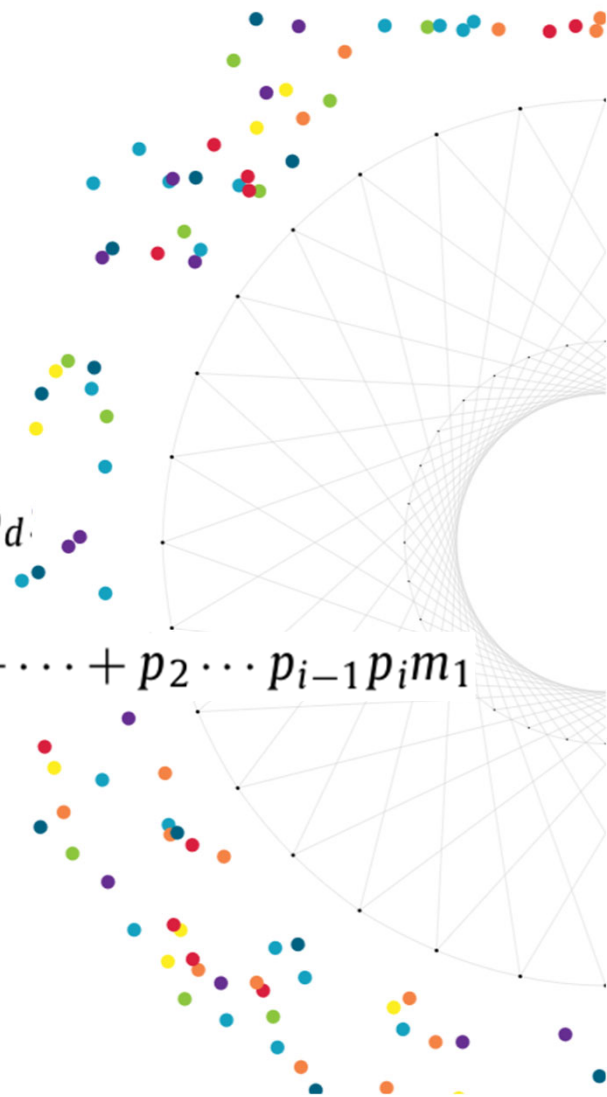
第 j 层节点中第 i 层节点的组件数量是 $Q_{i,j} = p_{i+1} \cdots p_j$

整个系统中第 i 层节点的组件数量是 (d 是树的深度) $Q_{i,d} = Q_i = p_{i+1} \cdots p_d$

第 i 层节点中可用的存储器总容量为 $M_i = m_i + p_i m_{i-1} + p_{i-1} p_i m_{i-2} + \cdots + p_2 \cdots p_{i-1} p_i m_1$

从第1层到第 i 层的带宽或通信代价为 $G_i = g_i + g_{i-1} + g_{i-2} + \cdots + g_1$

为了简化, 假设对所有的 i 都有 $m_i \geq m_{i-1}$



multi-BSP@Sun Niagara UltraSparc T1 multi-core

在Level 1 一个芯片有1个拥有4个线程的处理器，和一个1级cache，表示为：

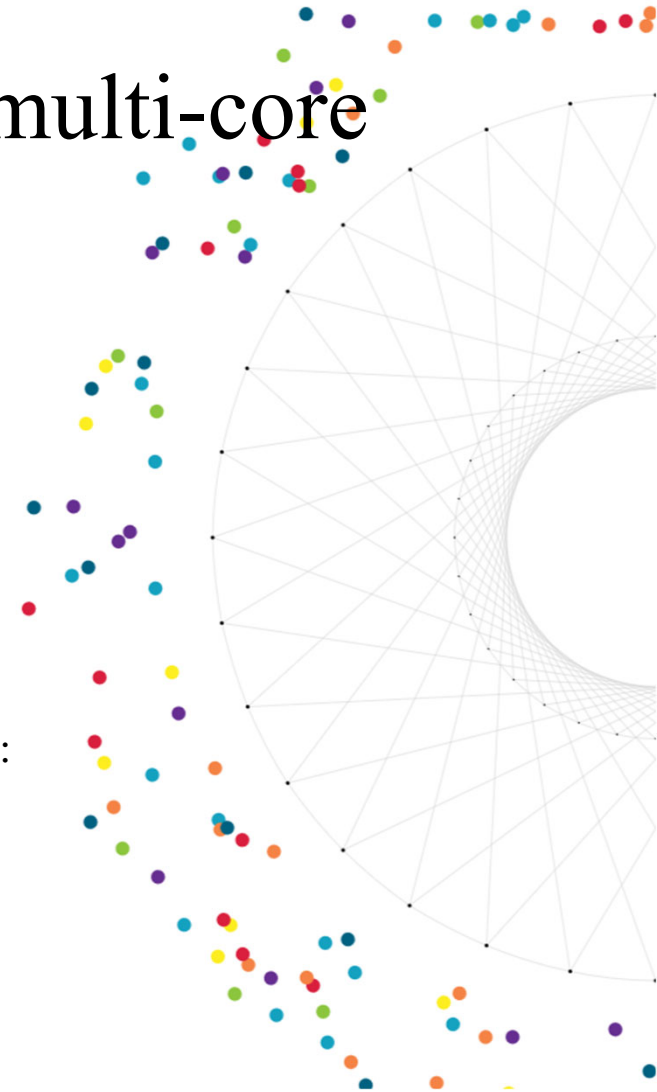
$$(p_1 = 4, g_1 = 1, L_1^* = 3, m_1 = 8 \text{ KB}).$$

在Level 2，一个芯片有8个内核和1个2级cache，表示为：

$$(p_2 = 8, g_2 = 3, L_2^* = 23, m_2 = 3 \text{ MB}).$$

在Level 3，p个mutil-core芯片，有外部存储器m3，通信带宽为g2，则表示为：

$$(p_3 = p, g_3 = \infty, L_3^* = 108, m_3 \leq 128 \text{ GB})$$



Multi-BSP (多层架构)模型的算法评价

设 $m = \min\{m_i \mid 1 \leq i \leq d\}$ 输入量 n 为一个很大的数。

F_1 F_2 表示 $\{p_i, g_i, L_i, m_i \mid 1 \leq i \leq d\}$ 的取值范围。

用 $F_1 \lesssim F_2$ 表示: 假设 m, n 足够大 对于所有的 $\varepsilon > 0$ 都有 $F_1 < (1 + \varepsilon)F_2$

用 $F_1 \lesssim_d F_2$ 表示: 对于某些依赖于 d 的 常量 c_d 假设 m, n 足够大, 则有: $F_1 < c_d F_2$

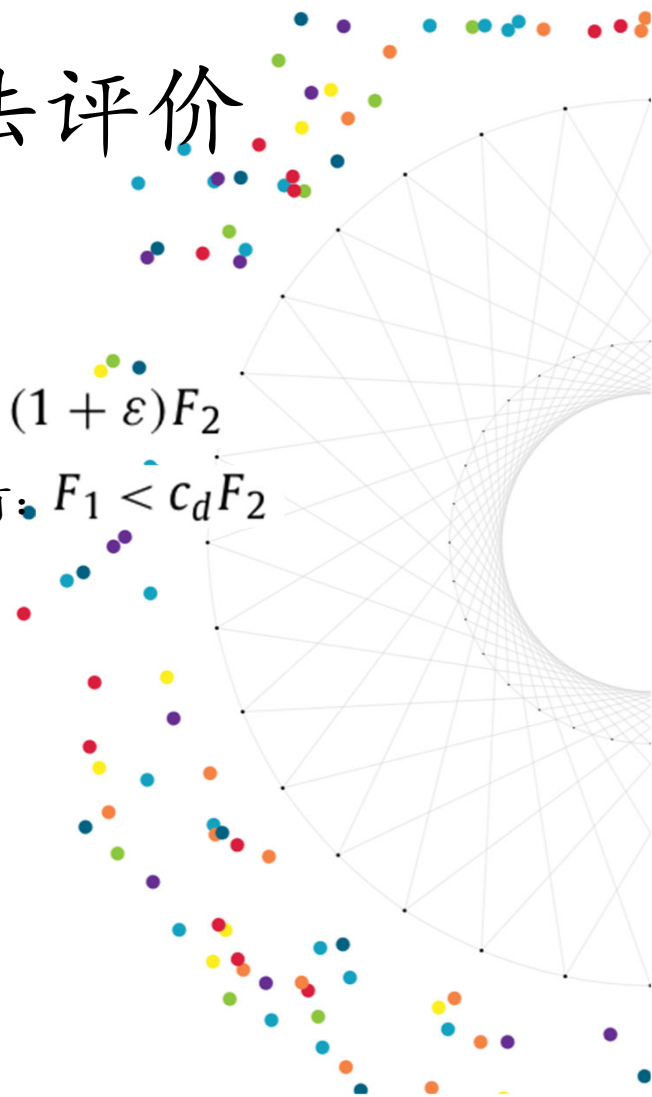
对于一个 Multi-BSP 算法 A^*

定义 $\text{Comp}(A^*)$ 表示并行计算的开销

定义 $\text{Comm}(A^*)$ 表示并行通讯的开销

定义 $\text{Synch}(A^*)$ 表示并行同步的开销

这三种开销均由处理器的基本时间单位表示。



Multi-BSP (多层架构)模型的算法评价

为了量化 A^* 的效率, 定义 A^* 的基线算法 A

用 $\text{Comp}_{seq}(A)$ 表示 A 执行的计算操作的总次数。

$\text{Comp}(A)$ 表示 $\text{Comp}_{seq}(A)/P_d$ 其中 P_d 表示 Multi-BSP machine H 中处理器的数量
即 $\text{Comp}(A)$ 表示每个处理器进行计算操作的平均次数

$\text{Comm}(A)$ 表示在 H 上运行的 A 算法的最小通信开销

$\text{Synch}(A)$ 表示在 H 上运行的 A 算法的最小同步开销

规定: 如果满足以下条件, 则 A^* 比 A 更有效

1. $\text{Comp}(A^*) \lesssim_d \text{Comp}(A)$, and $\text{Comp}_{seq}(A^*) \lesssim \text{Comp}_{seq}(A)$.
2. $\text{Comm}(A^*) \lesssim_d \text{Comm}(A)$
3. $\text{Synch}(A^*) \lesssim_d \text{Synch}(A)$

