CS2045M: 高级数据结构与算法分析 **2025-2026 学年秋冬学期**

Lecture 14: 并行算法

编写人：吴一航 yhwu_is@zju.edu.cn

穿过了 4 个困难而又精彩的的理论专题，我们来到了这门课程的尾声。在期末各大课程大作业等压力骤增的情况下，ADS 的最后两章难度的下降给了诸位温柔的怀抱。言归正传，本讲的主题在这门课程中相对比较独立：我们将进入一个全新的领域，讨论高性能计算在算法方面的基础优化方法，即设计所谓并行算法。本讲将在几个例子中讨论设计并行算法的几种基本思想与技巧。因为本讲在考试中也是比较基础的部分，因此重点将课件中的内容描述清楚，除了开头并行计算基础模型外，不会像前几章一样有过多的拓展。

14.1 何谓“并行”

【注】以下有较多内容参考了这个知乎页面，感兴趣的读者可以参考原链接学习更多。

规模最大且运算能力最强的计算机被称为“超级计算机”。在过去的二十年里，超级计算机的概念无一例外地指向拥有多个 CPU 且可同时处理一个问题的机器——并行计算机。

我们很难精确定义并行的概念，因为它在不同的层面上有着不同的含义。在计算机体系结构中，会见到如下三种并行模式：

1. 指令级并行（Instruction-Level Parallelism，ILP）：在一个 CPU 内部，多条指令可以同时执行；例如大家现在就已经学过的流水线 CPU，以及将来可能会学到的乱序、多发射以及超长指令字（VLIW，即把不相关的指令封装到一条超长的指令字中、超标量（例如有多个 ALU，可以同时运行没有相关性的多条指令）等；
2. 数据级并行（Data-Level Parallelism，DLP）：即将相同的操作同时应用于一些数据项的编程模型，例如经典的 SIMD（Single Instruction, Multiple Data）架构，即一条指令同时作用于多个数据，例如用一条指令实现向量加法，两个向量中每对对应的元素相加互不干扰，所以可以同时进行所有的加法；
3. 线程级并行（Thread-Level Parallelism，TLP）：一种显式并行，程序员要设计并行算法，写多线程程序，这也是本讲将要讨论的内容。线程级并行主要指同时多线程（SMT，是一种指令级并行的基础上的扩展，可以在一个核上运行多个线程，多个线程共享执行单元，以便提高部件的利用率，提高吞吐量）/超线程（HT，一个物理 CPU 核心被伪装成两个以上的逻辑 CPU 核心，看起来就像多个 CPU 在同时执行任务，是通过在一个物理核心中创建多个线程实现的）以及多核和多处理器。

这里需要澄清三个很相似但又不一致的词语：并发，并行和分布式。其中并行是指任务的真的在通过上述三种并行方式在一台机器的处理器上同时执行，分布式是指同一个网络里面的多台机器上同时完成多个任务。并发与上述两种有较大区别，实际上并发并非真的同时执行，而是通过时间片轮转的方式，让多个任务在一个处理器上交替执行，这样看起来就像是同时执行了，诸位在数据库和操作系统中学习的锁等都属于进程、线程调度并发控制的工具。

14.2 并行算法基本模型

说了这么多关于并行的基本内容，下面转向今天的主题：并行算法设计。使用并行算法显然是希望利用现代计算机多核的优势，同时执行一些可以不互相干扰的算法部分，加速算法的运行。因此有一个重要的参数称为加速比：

定义 14.1 *加速比（Speedup）是指在并行计算中，使用 p 个处理器时相对于使用一个处理器时的性能提升比例，即*

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

其中 T_1 是使用一个处理器时的运行时间，T_p 是使用 p 个处理器时的运行时间。

显然最理想的情况下，加速比应当是 p，即使用 p 个处理器时的运行时间是使用一个处理器时的 $1/p$。然而实际情况中，由于并行算法的设计和实现都有一定的困难：首先，使用多个处理器意味着额外的通信开销；其次，如果处理器并未分配到完全相同的工作量，则会产生一部分的闲置，就会造成负载不均衡（load unbalance），再次降低实际速度；最后，代码运行可能依赖其原有顺序，不能完全并行。

14.2.1 渐近论

如果我们忽略一些限制，比如处理器的数量，或者它们之间的互连的物理特性，就可以推导出关于并行计算效率极限的理论结果。本节将简要介绍这些结果，并简要讨论它们与现实中高性能计算的联系。

考虑一个很简单的例子，两个 n 阶方阵 A 与 B 相乘得到矩阵 C，显然总的操作数是 $2n^3$（所有需要的加法和乘法）。如果有 n^2 个处理器，那么有一个自然的并行策略，也就是让每个处理器负责结果矩阵 C 的一个元素的计算，而每个元素的计算显然互不干扰，这样并行需要的时间就是计算一个元素的时间，即 $2n$，因此这里的加速比就是 n^2，看起来非常完美。

当然可以用更多的处理器实现更快的算法，实际上矩阵乘法有一个步骤是要求 n 个对应元素乘法的和，这一步骤可以并行化，见 PPT 第 5 页，可以用 $n/2$ 个处理器实现在 $O(\log n)$ 的时间内完成这一步骤。因此如果一共有 $n^3/2$ 个处理器，即给每个元素的计算分配 $n/2$ 个处理器，那么 n 个乘法也可以并行到 2 步之内结束，因为乘法之间互不干扰，加法的时间则下降为 $O(\log n)$，因此总的时间为 $O(\log n)$，相比于 $O(n)$ 的时间有了提升。

对这些理论界线的一个反对意见是，它们隐含地假定了某种形式的共享内存。不难发现，上述矩阵相乘并行算法可能有很多个 CPU 同时访问了原始矩阵的同一个元素，比如同时得到 C 的第一行的所有元素需要同时访问 A 的第一行的所有元素。事实上，这种算法模型被称为 PRAM 模型（Parallel Random Access Machine），是 RAM 模型（Random Access Machine）在共享内存系统上的扩展。该模型假设所有处理器共享一个连续的内存空间。此外，模型还允许同一位置上同时进行多个访问。这在实际应用中，特别是在扩大问题规模和处理器数量的情况下是不可能的。对 PRAM 模型的另一个反对意见是，即使在单个处理器上，它也忽略了内存的层次结构（即忽略了 cache 等）。但在接下来的理论讨论中都会忽略这些问题，因为我们的目的是讨论并行算法的基本思想。我们使用 PRAM 模型，这一模型有规定如下三种内存共享方式：

1. EREW（Exclusive Read Exclusive Write）：每个内存位置在任意时刻只能被一个处理器读取或写入；
2. CREW（Concurrent Read Exclusive Write）：每个内存位置在任意时刻可以被多个处理器读取，但只能被一个处理器写入；
3. CRCW（Concurrent Read Concurrent Write）：每个内存位置在任意时刻可以被多个处理器读取或写入，因为写入涉及到同时写入不同值可能造成的冲突，因此写入策略又可以分为如下三种：
   (a) CRCW-C（Common）：所有处理器写入的值相同时才会写入；
   (b) CRCW-A（Arbitrary）：所有处理器写入的值可以不同，任意选取其中一个写入即可；
   (c) CRCW-P（Priority）：所有处理器写入的值可以不同，但是有一个优先级，只有优先级最高的写入才会生效。

如上讨论了内存模型与实际的差别，实际上还有一个具有较大争议的问题是处理器的个数。例如矩阵乘法问题，如果有无穷个处理器，能否设计出更高效的并行算法？当然这一问题的答案并不是今天要关心的，我们要关心的是，无穷个处理器，或者说很多个处理器真的是符合现实的吗？显然不是，处理器之间也是有通信时间的，电子的传输速度是有上界的，或者说目前相信光速是有界的，然后处理器阵列的排列方式也会影响通信时间，实际的布线如何实现也有很大的挑战。实际上，有人证明了，在三维世界和有限的光速下，对于 n 处理器上的问题，无论互连方式如何，速度都被限制在 $\sqrt[4]{n}$。这是因为有一个结论是，如果每个处理器占用一个单位体积的空间，在单位时间内产生一个结果，并且在单位时间内可以发送一个数据项。那么，在一定的时间内，最多只有半径为 t 的球中的处理器，即 $O(t^3)$ 的处理器可以对最终结果做出贡献；所有其他处理器都离得太远，这也意味着过多的处理器实际上是毫无意义的。那么，在时间 T 内，能够对最终结果做出贡献的操作数

$$\int_0^T t^3 \mathrm{d}t = O(T^4),$$

在时间 T 内，这意味着，最大的可实现的速度提升是串行时间的四次方根。当然，在我们的模型中，同样因为只是讨论基本的并行算法设计思想，因此不会过多关注这些细节，这里介绍只是希望强调一下实际情况与理想模型的区别。

在讨论了理想模型后，下面进一步讨论一些关于近似比的理论。前面矩阵乘法 n^2 个处理器的情况太过于理想，因为所有处理器之间的计算都互相不影响，因此达到了理想加速比。但现实中，部分代码依赖固有顺序执行，因此无法达到理想加速比。假设有 5% 的代码必须串行执行，那么这部分的时间将不会随着处理器的数量增加而减少。因此，对该代码的提速被限制在 20 的系数。这种现象被称为阿姆达尔定律（Amdahl's Law），下面对其进行表述：

定理 14.2 (阿姆达尔定律，1967) *设 $0 \leqslant f \leqslant 1$ 是一个程序中必须串行执行的部分的比例，那么使用 p 个处理器的最大加速比 $S(p)$ 满足*

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

证明： 设整个程序串行需要的时间为 t，那么串行部分的时间为 ft，并行部分的时间为 $(1-f)t$。使用 p 个处理器时，串行部分的时间不变，而并行部分的时间最少为 $(1-f)t/p$，因此总时间最小为 $ft + (1-f)t/p$，因此加速比最大为

$$S(p) = \frac{t}{ft + (1-f)t/p} = \frac{1}{f + (1-f)/p}.$$

■

当 $p \to \infty$ 时，$S(p) \to 1/f$，看起来这是一个很糟糕的结论，因为如果一个程序可并行的部分占到 90%，看起来非常多，但加速比最多只能达到 10 倍，是一个并不大的数字，因此一定程度上给研究并行计算的人们带来了挫败感，因为这个定律让人们认为并行可能不是一个很有前途的方向。

然而我们可以怀疑这一定律的合理性：阿姆达尔定律认为只增加处理器数量并不会对并行加速结果有明显的提升，其隐含假设是：越来越多的处理器上执行同一个固定计算。然而在实际中情况并非如此：人们根据可用处理器的数量来调整问题规模的大小，处理器很多的时候可以处理更多的数据，并且有一个现实的假设是，串行部分可以是独立于问题大小的，因此数据量越大，并行部分越大，而阿姆达尔定律则是假设数据集的大小是固定的。首个打破阿姆达尔定律这一假设的是 John L. Gustafson，他在 1988 年于 Communications of the ACM 上发表论文提出了如下定理：

定理 14.3 (古斯塔夫森定律，1988) *设某个程序使用 p 个处理器并行执行时，串行部分使用的时间为 f_1（不是比例），并行部分使用的时间为 f_2，那么使用 p 个处理器的最大加速比 $S(p)$ 满足*

$$S(p) = \frac{f_1 + f_2 \cdot p}{f_1 + f_2}.$$

证明： 由假设可知，使用 p 个处理器并行执行的时间为 $T_p = f_1 + f_2$，使用一个处理器串行执行的时间最大为 $T_1 = f_1 + f_2 \cdot p$，因此加速比为

$$S(p) = \frac{T_1}{T_p} = \frac{f_1 + f_2 \cdot p}{f_1 + f_2}.$$

■

实际上这里的关键就是没有规定串行和并行时间所占的比例，而是可以根据处理器的数量调整它们各自的占比。显然，这里当 $f_2 \to \infty$ 时，$S(p) \to p$，这就与前面的理想加速比相符合了，因此做并行计算看起来还是有前途的！

阿姆达尔定律和古斯塔夫森定律看起来似乎水火不容，但 1990 年，密西根州立大学的博士生孙贤和（现为伊利诺理工大学教授）与倪明选（现为澳门大学教授）在 Supercomputing 会议上提出了一种 Memory-Bounded 并行加速模型，成功地统一并扩展了 Amdahl 定律与 Gustafson 定律。该模型后被称为 "Sun-Ni's Law"，写入并行计算教科书：

定理 14.4 (孙-倪定律，1990) *设某个程序串行部分占比为 f，并行部分占比为 $1-f$，内存受限函数为 $G(p)$（后面会解释），那么使用 p 个处理器的最大加速比 $S(p)$ 满足*

$$S(p) = \frac{f + (1-f)\cdot G(p)}{f + \dfrac{(1-f)\cdot G(p)}{p}}.$$

假设 $G(p) = 1$，则问题大小是固定的或与资源增加无关，内存限制加速模型简化为 Amdahl 定律；假设 $G(p) = p$，则内存限制加速模型简化为 Gustafson 定律，这意味着当内存容量增加 p 倍，工作负载也增加 p 倍。

然而，前面所说的定律都只是非常粗糙的估计，因为很多实际的程序并不能很干净地分成串行和并行部分，我们需要一个更精细的模型，也就是关键路径（Critical Path）和相应的布伦特定理（Brent's Theorem）。我们将关键路径定义为最长度的依赖关系链（可能是非唯一的），这个长度有时被称为跨度（span），在本课程中统一记为 D，即深度 depth。由于关键路径上的任务需要一个接一个地执行，关键路径的长度是并行执行时间的一个下限，因此这自然也就是理想情况下有无穷个处理器的时候需要的执行时间，即有

$$D = T_\infty.$$

另外有一个参数——工作量（work），记为 W，就是并行中所有任务的总时间，实际上就是只用一个处理器的时候的执行时间（忽略不同处理器之间的通信等），即

$$W = T_1.$$

有了这些概念，可以将算法的平均并行度（average parallelism）定义为 $W/D = T_1/T_\infty$。

PPT 第 9 页给出了这两个参数在求和问题中更直观的展示，实际上 D 就是二叉树的高度，W 就是二叉树的所有节点的数量，即 $D = O(\log n)$，$W = O(n)$。

现在我们讨论了 T_1 和 T_∞，但没有像前面的定理那样讨论 T_p，即对于一般的处理器数量，一个任务的处理时间如何。那么接下来的布伦特定理就是要讨论这一问题：

定理 14.5 (布伦特定理，Brent's Theorem) *设一个并行计算问题的工作量为 W，关键路径长度为 D，那么使用 p 个处理器的并行时间具有如下上下界：*

$$\frac{W}{p} \leqslant T_p \leqslant \frac{W-D}{p} + D.$$

证明: 下界是显然的，因为 $T_p \geqslant T_\infty$，而 T_∞ 也是受到理想加速比的限制的，因此 $T_p \geqslant W/p$。

对于上界，因为关键路径长度为 D，因此整个算法的执行可以分为 D 个内部并行，互相之间串行的阶段，设每个阶段的工作量为 W_i，则有

$$W = \sum_{i=1}^{D} W_i,$$

使用 p 个处理器时，每一个阶段所需的时间为 $\lceil W_i/p \rceil$，向上取整的原因在于，每个阶段的工作量可能不能被 p 整除，比如大任务被分成了 7 个子任务，7 个任务分配给 3 个处理器，那么每个处理器分配到的任务数应当是 2，2，3。

因此需要的总时间就是

$$T_p = \sum_{i=1}^{D} \left\lceil \frac{W_i}{p} \right\rceil = \sum_{i=1}^{D} \left(\left\lfloor \frac{W_i - 1}{p} \right\rfloor + 1 \right) \leqslant \sum_{i=1}^{D} \left(\frac{W_i - 1}{p} + 1 \right) = \frac{W - D}{p} + D.$$

其中使用了等式

$$\left\lceil \frac{x}{y} \right\rceil = \left\lfloor \frac{x-1}{y} \right\rfloor + 1.$$

■

根据这一结论，你也可以得到加速比 T_1/T_p 的具体公式，事实上前面的定理都只是在不断地给出加速比的上界，但这里引入关键路径这一更加贴合实际程序运行的理论模型同时给出了上下界，因此是一个较强的结论。

14.3 前缀和问题

前面的讨论介绍了并行计算的基础理论，引入了 PRAM 等简化计算模型，以及基于关键路径的理论，接下来将在这些理论的基础上讨论一些具体的并行算法设计思路。通常有五种设计并行算法的范式，包括平衡二叉树、划分范式、双对数范式、加速级联范式以及流水线范式，其中流水线范式不展开介绍，感兴趣的读者可以阅读 PPT 最后给出的参考资料，其中给出了 2-3 树的插入和删除的流水线算法（笔者并没有仔细阅读，觉得有些许抽象）。而平衡二叉树在求和问题中已经见到了，因此接下来相当于给出一个更复杂的例子进行应用。

下面要讨论的是前缀和问题，即给定一个长度为 n 的数组 A，需要设计并行算法，求出 A 的前 k 个元素的和，其中 k 取遍 1 到 n。PPT 第 12-13 页给出了这一算法，关键步骤就在于设

$$C(h,i) = \sum_{k=1}^{\alpha} A(k),$$

其中 α 是点 (h,i) 为根的子树最右下角的叶子的 i 值，那么可以从上往下计算这个 C，每一层计算都可以并行，直到算完 $C(0,i)$ 实际上就是 1 到 i 的前缀和。因此关键就是怎么从上往下计算，并且保证每一层都能并行。事实上非常简单，因为很容易观察到根结点 C 值就是整个数组的和，因此这只需要把自底向上求和的结果（存在数组 B 中）保留即可。然后再往下走的过程中，有如下三个观察：

1. 如果 $i=1$，那么 $C(h,i)=B(h,i)$，这是因为 $i=1$ 的时候，根据定义，$C(h,1)$ 是从第 1 个元素加到以这一点为根的子树右下角的叶子，而 $B(1,h)$ 事实上也就是从第一个元素加到以这一点为根的子树右下角的叶子（看图可以看得很清楚）；
2. 如果 i 是偶数，这表明这一点是某个点的右儿子，因此它和它的父亲最右下角的叶子是同一个，因此有 $C(h,i)=C(h+1,i/2)$；
3. 如果 i 是奇数且不是 1，这表明这一点是某个点的左儿子，首先它自己的值 $B(h,i)$ 不是从 1 开始加的，所以要选一个左边的点，把从 1 开始加到这个点对应的之前的部分补上，即 $C(h,i)=C(h+1,(i-1)/2)+B(h,i)$。需要注意的是，如果不是并行算法，$C(h+1,(i-1)/2)$ 可以替换为 $C(h,i-1)$，但因为并行算法要求每一行是一起算出来的，所以要用上一行的才合理。

算法的伪代码见 PPT 第 13 页，显然这一并行算法的深度是 $O(\log n)$，总工作量是 $O(n)$，因为无非就是在完全二叉树上先从下往上，然后再从上往下分层遍历了两轮。

14.4 归并问题

接下来将介绍第二种并行算法的设计思路，因为二叉树对于更复杂的问题而言有较大的局限性。讨论的第一个例子是归并问题，PPT 第 14 页给出了问题的描述，事实上与归并排序中 “归并” 步骤是一致的，除了增加了几条比较平凡的假设。下面将讨论如何设计一个并行算法来解决这个问题。

14.4.1 简单的想法

既然可以设计并行算法，那么如果能同时将所有 A 和 B 中的元素在最后的数组 C 中的位置找到，然后同时将它们放到正确的位置上，那么就可以以很快的速度解决这一问题。

这里的关键步骤就是同时找到 A 和 B 中的元素在 C 中的位置，这是一个并不困难的问题，例如对 A 中的元素 $A[i]$，它在 A 中是第 $i+1$ 个元素（下标从 0 开始），即在它前面有 i 个元素。如果能找到它在 B 中处于

$$B[j]<A[i]<B[j+1]$$

的位置，那么 B 中有 $j+1$ 个元素在合并后的 C 中应当在 $A[i]$ 前面，因此最后 $A[i]$ 在 C 中的位置就是 $C[i+j+1]$，这样它前面就有 $i+(j+1)$ 个元素了。对 B 中的元素 $B[k]$ 也有类似的分析。

由此将归并问题转化为了找到某个数组中的一个元素在另一个数组中的位置的问题。显然有两种策略：

1. 逐个元素二分查找：使用二分查找找到一个元素在另一个数组中的位置只需要 $O(\log n)$ 的时间，即深度 $D=O(\log n)$，总工作量为 $O(n\log n)$；然后需要将所有元素放到正确的位置上，这一深度为 $O(1)$，总工作量为 $O(n)$（如果数组长度不同则是 $O(n+m)$），两步合起来深度为 $O(\log n)$，总工作量为 $O(n\log n)$；

2. 整体线性查找：PPT 第 16 页给出了伪代码，实际上和归并排序的操作并无本质差别，就是两个数组的元素从头依次向后比较，因此完全没有并行，深度和总工作量都是 $O(n)$（或者数组长度不同就是 $O(m+n)$）。

但这两种方法都不令人满意，因为第一种方法的总工作量太大，而第二种方法的深度太大。我们希望能够找到一个更好的方法，这就需要引入一种更加高级的并行算法设计思路，即划分范式。

14.4.2 划分范式（Partitioning Paradigm）

划分范式的想法非常简单，且实际运用起来非常有效，分为如下两个步骤：

1. 划分（partitioning）：把原问题划分为很多（设为 p 个）很小的子问题，每个子问题大小大致为 n/p；
2. 真实工作（actual work）：同时对所有子问题进行处理，得到最终结果。

有心的同学应当观察到了，前面的二分查找方法相当于将原问题划分为了 n 个子问题，即 $p=n$，而线性的方法则完全没有划分，即 $p=1$。因此这两种方法分别对应于划分范式的两个极端，所以有理由相信合理的 p 可以得到一个更好的结果，能让两个极端取长补短。

14.4.3 运用划分范式

目标非常明确，就是取长补短。因此算法应当按照如下方式套用划分范式：

1. 划分：将每个数组划分为 p 份（此时 p 未知，分析后可以选取到最优的 p），p 是一个很大的值，每个子问题很小，如 PPT 第 17 页图所示。首先对每个子问题的第一个元素求出它们在另一个数组的位置，这一步应当使用二分查找，否则因为子问题很多，使用线性查找会导致总工作量达到 $O(p\cdot n)$，这几乎是平方级别的工作量，显然是不可接受的；因此使用二分查找，这样深度为 $O(\log n)$，总工作量为 $O(p\log n)$；
2. 真实工作：如 PPT 第 17 页图所示，现在剩下的工作就是相邻两个箭头之间的部分需要在这一很小的区域内确认相对位置，因为这些位置确定后直接可以根据上一步划分中得到的结果确定在整个最终的数组中的位置。

   注意，很显然的一点是，这些箭头不会交叉，这是由箭头代表的大小关系决定的。另一方面，相邻两个箭头之间的距离一定不超过 n/p，因为一旦超过 n/p，那其中一定会有一个箭头（箭头出现的频率是每 n/p 个元素一次），并且一个数组上有 p 个出发的位置和 p 个到达的位置，所以在任意一个数组上，出发点和到达点一共最多有 $2p$ 个，然后两个数组的这些划分区域按图中形式一一对应进行计算，每个区域实际上就是一个子问题，即现在还剩下 $2p$ 个大小为 $O(n/p)$ 的子问

题。这时并行对每个子问题直接使用线性查找即可，因为每个子问题都非常小（大小为 $O(n/p)$），此时深度为 $O(n/p)$ 也很小，总工作量为 $2p \cdot O(n/p) = O(n)$ 是符合要求的。而如果使用二分查找，总工作量为

$$2p \cdot \frac{n}{p} \cdot O(\log \frac{n}{p}) = 2n \log \frac{n}{p},$$

这是因为有 $2p$ 个子问题，每个子问题都至多有 n/p 个元素要用至多 $\log(n/p)$ 的时间确定位置，这并无法保证是 $O(n)$ 的，因此不使用二分查找。

现在的问题就在于，还有两个时间复杂度未定：第一步划分的总工作量为 $O(p \log n)$，第二步真实工作的深度为 $O(n/p)$。我们希望找到一个最优的 p，使得第一步总工作量是 $O(n)$ 的，第二步深度是 $O(\log n)$ 的。这样就可以得到一个深度为 $O(\log n)$，总工作量为 $O(n)$ 的并行算法，成功做到取长补短。事实上很容易看出，$p = \dfrac{n}{\log n}$ 就是满足条件的，因此就选取这个 p，得到了一个深度为 $O(\log n)$，总工作量为 $O(n)$ 的并行算法。

总而言之，划分范式就是通过划分（在大数据量下使用二分查找）和真实工作（在小工作量下使用线性查找）统一了两种极端情况，然后希望找到一个折中的合适的 p，使得两个极端取长补短，得到一个更好的结果。

14.5 寻找最大元素

下面希望设计并行算法找到一个数组中的最大元素。我们将会讨论第三种并行算法的设计范式，即双对数范式，这是二叉树的一种扩展；然后讨论加速级联范式，它可以综合不同的并行算法得到更好的并行算法。

14.5.1 简单的想法

在讨论复杂的范式前先来讨论一些简单的想法。第一种最简单的想法就是类似于前面求和问题的做法，构造一棵二叉树，初始元素两两分组比较，然后逐层上递，最终得到最大值。这样的算法深度为 $O(\log n)$，总工作量为 $O(n)$。

然而我们可以更激进：为什么不一次性把所有元素之间都比较一次呢？这样深度只需要 $O(1)$。如果不考虑总工作量会很大的危害，这种方式是可行的：并行比较每一对元素 $A[i]$ 和 $A[j]$，只要 $A[i]$ 在比较中是较小的一方，就往新的数组 $B[i]$ 处写 1（B 的所有位置初始化为 0），最后那个最大的元素肯定从来没被写过 1，因此还保持着 0，所以并行判断 B 中哪个元素是 0，那它就是最大值。

这里有一个需要注意的细节是，在并行比较每一对元素时，可能会有多个线程同时往数组 B 中进行写入，实际上只需要使用前面提到的 CRCW 策略允许同时写入，然后按 common rule 写入即可，因为这里所有线程往任意一个 $B[i]$ 只会写入 1 这个数字，所以写入的内容都是一样的，故用 common 就能实现写入。

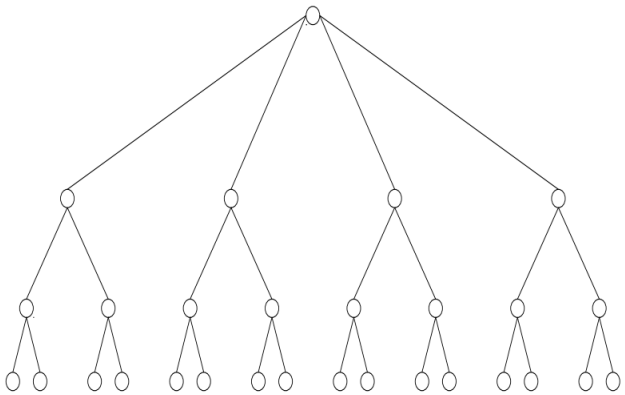
很显然，这样的算法深度非常完美，是 $O(1)$ 的，但总工作量重点需要比较每一对元素的值，因此是 $O(n^2)$ 的。上面的两种方法在总工作量和深度上如果能取长补短，必定是非常完美的，接下来就来逐步尝试优化它们。

14.5.2 双对数范式（Doubly-logarithmic Paradigm）

双对数范式是一种可以二叉树的扩展。在完全二叉树中，设叶子的数量为 n，那么树高是 $\log n$ 级别的，这里双对数则是希望构造一棵树，使得树高是 $\log\log n$ 级别的。为了构造这样一棵树，首先设树中每个节点的 level 为从根到该节点的距离（需要经过的边数），根的 level 为 0。接下来构造这棵树如下：

1. 设某个节点的 level 为 s，当 $s \leqslant \log\log n - 1$ 时，则它有 $2^{2^{h-s-1}}$ 个孩子；
2. 当 $s = \log\log n$ 时，它有 2 个孩子作为树的叶子。

不难验证这样一棵树的叶子数量的确是 n，高度也是 $\log\log n$ 级别的。下图给出了一个 16 个叶子的双对数树的例子：



事实上还可以有一个观察，即一个不在倒数两行的结点的 level 为 s 时，根据定义它有 $2^{2^{h-s-1}}$ 个孩子，然后以它为根的子树的叶子数量可以算出来是 $2^{2^{h-s}}$，而我们知道

$$2^{2^{h-s-1}} = \sqrt{2^{2^{h-s}}},$$

也就是说在每个节点相当于把问题分成了 $\sqrt{m}$ 个子问题，其中 m 为该节点对应的子树的叶子数量。而我们知道，以双对数树某一层单个节点为根的子树的叶子数，其实是上一层某个节点为根的子树的叶子数开根号（因为层数为 s 的某个节点，以它为根的子树的叶子数量是 $2^{2^{h-s}}$，s 每增加 1 实际上就是开一次根号），因此层数越往下，就相当于子问题大小逐步开根号。假设原问题输入大小为 n，那么可以通过加一些 dummy 节点使得可以构造出以这 n 个节点为叶子的双对数树，这样就可以使用双对数范式来设计一个并行算法，即在每一层将问题分为 $\sqrt{m}$ 个子问题，m 为这一层的单个节点为根的子树的叶子数量，这样每个子问题（其实就是树上某个节点）对应的叶子数也就是 $\sqrt{m}$ 了，这就和双对数树结合起来了。我们熟知的二叉树则是每层分成两个子问题，是更简单的策略。

将这一思路套用在当前的问题上：首先将数组划分为 $\sqrt{n}$ 份，每一份的大小为 $\sqrt{n}$，然后并行递归找到每一份中的最大值，最后归并阶段再并行找到这 $\sqrt{n}$ 个最大值中的最大值即可。设整个算法深度为 $D(n)$，工作量为 $W(n)$，它们都是关于数组长度的函数。每一份中的最大值采用递归的策略，因此长度为 $\sqrt{n}$ 的数组的最大值的寻找需要 $D(\sqrt{n})$ 的深度和 $W(\sqrt{n})$ 的总工作量。以上是递归阶段，在归并阶段，返回的 $\sqrt{n}$ 个最大值中的最大值采用前面的两两比较的方法即可，深度为 $O(1)$，总工作量为 $O((\sqrt{n})^2) = O(n)$。于是按照分治法的方式可以写出递推式

$$D(n) = D(\sqrt{n}) + O(1),$$
$$W(n) = \sqrt{n}W(\sqrt{n}) + O(n).$$

第一个式子这样的类型在分治算法中应当已经见过，换元法即可破解。设 $n = 2^m$，则有

$$D(2^m) = D(2^{m/2}) + O(1),$$

设 $D(2^m) = S(m)$，则有

$$S(m) = S(m/2) + O(1),$$

肉眼即可解得 $S(m) = O(\log m)$，因此 $D(n) = O(\log\log n)$。

对于 W 的递推式，直接暴力展开找规律即可。同样设 $n = 2^m$，$S(m) = W(2^m)$，则有

$$S(m) = 2^{m/2}S(m/2) + O(2^m),$$

进一步展开 $S(m/2)$，有

$$S(m/2) = 2^{m/4}S(m/4) + O(2^{m/2}),$$

代入上式，有

$$S(m) = 2^{3m/4}S(m/4) + O(2 \cdot 2^m),$$

继续展开，有

$$S(m) = 2^{7m/8}S(m/8) + O(3 \cdot 2^m),$$

以此类推，直到 $S(1)$ 有

$$S(m) = 2^{m-1}S(1) + O(\log m \cdot 2^m),$$

代入 $S(1) = O(1)$，有

$$S(m) = O(\log m \cdot 2^m),$$

即 $W(n) = O(n\log\log n)$。因此得到了一个深度为 $O(\log\log n)$，总工作量为 $O(n\log\log n)$ 的并行算法。

当然可以用更直观的方式来理解上面两个递推式的结果。事实上 D 的递推式就对应于求叶子数为 n 的双对数树的高度，因为子问题大小每开一次根号，大小就加一，这和双对数树每往下一层，单个节点为根的子树的叶子数开根号完全对应。而 W 的递推式表明，在最高层需要多做 $O(n)$ 的工作，第二层应当多做 $O(\sqrt{n}\cdot\sqrt{n}) = O(n)$ 的工作（$\sqrt{n}$ 个子问题，每个子问题在解决时，综合出这个子问题的解都需要 $\sqrt{n}$ 的时间），以此类推，每一层都需要做 $O(n)$ 的工作，因此总工作量是 $O(n\log\log n)$，实际上这就是在利用递归树方法求解通项。

注：其实上面不一定要分成 $\sqrt{n}$ 份，分成 $n^{1/3}$ 份也可以用相同的方式证明最后的复杂度满足前面的结论。

14.5.3 加速级联范式（Accelerating Cascades Paradigm）

我们仍然希望改进上面的算法，那么加速级联范式通常是一个值得尝试的手段——即可以考虑将现有的并行算法，例如上面讨论的双对数范式的算法与其它策略结合，得到更好的算法。方法如下（至于怎么凑的，那也是一些智慧与尝试的结晶了）：

1. 将数组分为 $n/\log\log n$ 份，即每一份的大小为 $\log\log n$，如 PPT 第 20 页；实际上每一份的大小都很小了，所以可以直接利用线性查找的方式找到每一份的最大值，则每一份的深度和工作量都是 $O(\log\log n)$ 的；
2. 然后对上面求出的 $n/\log\log n$ 个最大值使用双对数范式的算法。

于是总的深度为

$$D(n) = O(\log\log n) + O(\log\log(n/\log\log n)) = O(\log\log n),$$

总的工作量为

$$W(n) = O(n/\log\log n \cdot \log\log n) + O(n/\log\log n \cdot \log\log(n/\log\log n)) = O(n).$$

当然也可以尝试其它级联的方式，比如第一步用二叉树的方法等，但采用上面的步骤已经能实现一定的改进。同学们也可以探究不同的划分对于这一算法的影响，这里就不再深入讨论了。

14.5.4 随机算法

看起来还有改进的空间，毕竟当初暴力的两两比较方法能实现 $O(1)$ 的深度。接下来将介绍一种随机算法，它可以保证以非常高的概率在 $O(1)$ 的深度和 $O(n)$ 的工作量内找到最大值。

首先给出算法的描述，然后进行分析：

1. 第一步：从长度为 n 的数组 A 中，依照均匀分布取出 $n^{\frac{7}{8}}$ 个元素，得到新的数组记为 B；这一步需要 $n^{\frac{7}{8}}$ 个处理器各自负责抽一个然后放到内存中某个位置，深度为 $O(1)$，总工作量为 $O(n^{\frac{7}{8}})$；
2. 第二步：求出 B 中的最大值，使用确定性的算法实现，通过如下三个子步骤实现：
   (a) 把 B 分成 $n^{\frac{3}{4}}$ 个长度为 $n^{\frac{1}{8}}$ 的子数组，然后使用两两比较的暴力算法并行找到每个子数组的最大值，这一步深度为 $O(1)$，总工作量为 $O(n^{\frac{3}{4}} \cdot (n^{\frac{1}{8}})^2) = O(n)$，因为有 $n^{\frac{3}{4}}$ 个子数组，每个子数组用两两比较的办法的复杂度是数组长度平方级别的；然后将这 $n^{\frac{3}{4}}$ 个最大值放在新数组 C 中；

(b) 把 C 分成 $n^{\frac{1}{2}}$ 个长度为 $n^{\frac{1}{4}}$ 的子数组，然后使用两两比较的暴力算法并行找到每个子数组的最大值，这一步深度为 $O(1)$，总工作量为 $O(n^{\frac{1}{2}} \cdot (n^{\frac{1}{4}})^2) = O(n)$，然后将这 $n^{\frac{1}{2}}$ 个最大值放在新数组 D 中；

(c) 数组 D 直接使用两两比较的方法找到最大值，这一步深度为 $O(1)$，总工作量为 $O((n^{\frac{1}{2}})^2) = O(n)$。

综合上述方法，整个第二步相当于一个三轮的淘汰赛，整体而言第二步的深度为 $O(1)$，总工作量为 $O(n)$，并且能求出 B 中的最大值，也就是抽取的 $n^{\frac{8}{7}}$ 个元素中的最大值；

【注】实际上不难验证最开始抽 $n^{\frac{15}{16}}$ 或者其它类似值的也是可以的，当然不能取太高破坏复杂度。

3. 第三步：目前只得到了 $n^{\frac{7}{8}}$ 个元素中的最大值（记为 M），如何进一步提高概率呢？答案是再来一轮，但是这再来一轮是有讲究的。这一步首先均匀分布取出 $n^{\frac{7}{8}}$ 个元素，得到数组 B；然后用 n 个处理器，每个处理器放 A 的一个元素，与前一步得到的 M 进行比较，如果小于 M 则什么也不用做，大于 M 则往数组 B 的一个随机位置写入这一更大的值（为什么要随机写入呢？因为一共 n 个处理器，但只有 $n^{\frac{7}{8}}$ 个位置可供选择，不能一一对应位置供每个处理器使用，但又要在 $O(1)$ 时间内完成位置分配，只能是随机了）。写完后，再次求出更新后的 B 中的最大值，这一步的深度和工作量和第二步一样，因此是 $O(1)$ 的深度和 $O(n)$ 的工作量。

那么这样成功的概率如何呢？如果第二步得到的 M 本来就在原数组中排名靠前，例如位列前 $n^{\frac{1}{4}}$，那么第三步随机放入更大的元素时，出现冲突的概率就会降低，如果完全没有冲突，那么所有比 M 大的元素都会被放入 B 中，这样 B 中的最大值就是原数组的最大值。但如果出现冲突且导致真正的最大元素因为冲突没有写入 B，那么我们的算法就失败了。

总而言之，关于算法的正确性有如下定理：

定理 14.6 *以上算法以非常高的概率在 $O(1)$ 的深度和 $O(n)$ 的工作量内找到数组 A 中的最大值：存在常数 c，使得算法只有 $\frac{1}{n^c}$ 的概率无法在这一时间复杂度内找到最大值。*

在这一复杂度内找到最大值的含义可以简单理解为，上面的第三步只执行一次——事实上为了进一步提高正确概率，第三步可以被重复实施很多次，但很多次后将会破坏算法的时间复杂度，上面的定理就是表明，在找到最大值之前破坏这一时间复杂度的概率是很小的。

最后以这一定理的证明来结束本讲的内容（但这一证明并不要求掌握）：

证明: 事实上，上述算法第三步一轮之内能找到最大值的概率大于等于如下两部分的乘积：

1. 第二步返回的结果排名在整个长度为 n 的数组的前 $n^{\frac{1}{4}}$；
2. 在上一条的条件下，第三步随机放入的元素都没有冲突。

大于等于的原因是，即使不满足上述条件也可能找到最大值，比如冲突的元素不是最大值，但这么粗略的估计就能够证明最后的结论。

首先来看第一部分的概率。考虑这一问题的补，即抽到的 $n^{\frac{7}{8}}$ 个元素中最大的那个元素排名在前 $n^{\frac{1}{4}}$ 之外的概率。这事实上等价于所有这 $n^{\frac{7}{8}}$ 个元素的排名都在前 $n^{\frac{1}{4}}$ 之外。因为是均匀分布抽取的，故每一个元素的排名在前 $n^{\frac{1}{4}}$ 之外的概率是

$$\frac{n-n^{\frac{1}{4}}}{n}=1-n^{-\frac{3}{4}},$$

并且不同的元素之间是独立的，因此所有元素的排名都在前 $n^{\frac{1}{4}}$ 之外的概率是

$$(1-n^{-\frac{3}{4}})^{n^{\frac{7}{8}}}=((1-n^{-\frac{3}{4}})^{n^{\frac{3}{4}}})^{n^{\frac{1}{8}}},$$

回忆 $(1-\frac{1}{n})^n$ 是单调递增趋于 $\frac{1}{e}$ 的，因此一定存在一个常数 $c_1<1$ 使得上式小于等于 $c_1^{n^{\frac{1}{8}}}$。由于这里求的是原问题的补问题的概率，故原问题的概率大于等于 $1-c_1^{n^{\frac{1}{8}}}$。

接下来求解第二部分概率，即仅有小于等于 $n^{\frac{1}{4}}$ 个元素会写入新数组的情况下，写入没有冲突的概率，这个概率的一个下界是不难得到的。虽然所有元素是并行写入的，但可以串行思考。第一个元素写入时，没有冲突的概率显然是 1；第二个元素写入时，不与第一个元素冲突的概率是

$$\frac{n^{\frac{7}{8}}-1}{n^{\frac{7}{8}}}=1-n^{-\frac{7}{8}},$$

当写入第 $i+1$ 个元素时，它不发生冲突的概率最低的情况就是前 i 个元素都没有冲突的情况，因为此时前 i 个元素都写入了不同位置，因此冲突的概率最高，故不冲突的概率最低。因此第 $i+1$ 个元素不发生冲突的概率大于等于

$$1-i\cdot n^{-\frac{7}{8}},$$

因此所有元素都不冲突的概率大于等于

$$(1-n^{-\frac{7}{8}})(1-2n^{-\frac{7}{8}})\cdots(1-n^{\frac{1}{4}}\cdot n^{-\frac{7}{8}})\geqslant(1-n^{\frac{1}{4}}\cdot n^{-\frac{7}{8}})(1-n^{\frac{1}{4}}\cdot n^{-\frac{7}{8}})\cdots(1-n^{\frac{1}{4}}\cdot n^{-\frac{7}{8}})=(1-n^{-\frac{5}{8}})^{n^{\frac{1}{4}}},$$

又有

$$(1-n^{-\frac{5}{8}})^{n^{\frac{1}{4}}}=((1-n^{-\frac{5}{8}})^{n^{\frac{5}{8}}})^{n^{-\frac{3}{8}}},$$

因为 $(1-\frac{1}{n})^n$ 单调递增趋于 $\frac{1}{e}\geqslant\frac{1}{3}$，故 n 较大时，上式大于等于 $(\frac{1}{3})^{n^{-\frac{3}{8}}}$.

为了得到最后的结果，需要考察 1 和 $(\frac{1}{3})^{n^{-\frac{3}{8}}}$ 之间的距离，设 $1-f(n)=(\frac{1}{3})^{n^{-\frac{3}{8}}}$，即 $(1-f(n))^{\frac{8}{3}}=\frac{1}{3}$。下面希望据此推出 $f(n)=O(\frac{1}{n^{c_2}})$，其中 $c_2>0$ 是一个常数。事实上，如果取 $f(n)=n^{-\frac{3}{16}}$，那么 $(1-n^{-\frac{3}{16}})^{\frac{8}{3}}$ 会趋于 0，因为这实际上就是 $(1-\frac{1}{n})^{n^2}$，而这一值趋向于 $\frac{1}{e^n}$，故趋近于 0。因此不可能等于 1/3，想要始终恒等于 1/3，$f(n)$ 的的下降速度必须比 $n^{-\frac{3}{16}}$ 更快，在 n 较大的时候值也就更小，因此 $f(n)$ 一定是 $O(\frac{1}{n^{c_2}})$ 的，其中 $c_2>0$ 是一个常数。

综合上面的讨论，第三步一轮之内不能找到最大值的概率就会小于等于 1 减去前面算出的两部分的乘积，即

$$P\leqslant 1-(1-c_1^{n^{\frac{1}{8}}})\cdot(1-f(n))=f(n)+c_1^{n^{\frac{1}{8}}}\cdot(\frac{1}{3})^{n^{-\frac{3}{8}}},$$

我们知道，$f(n)$ 是 $O(\frac{1}{n^{c_2}})$ 的，而 $c_1^{n^{\frac{1}{8}}}$ 在 n 较大时会趋向于 0，$(\frac{1}{3})^{n^{-\frac{3}{8}}}$ 会趋向于 1，因此 P 最大也就是 $f(n) = O(\frac{1}{n^{c_2}})$ 级别的，因此命题得证。 ■