并行算法科普向 系列之一：计算模型，调度器，和其它

Yan Gu
计算机科学 话题的优秀回答者

（本文作者为 Yihan Sun，转载请注明出处）

上次发出了一篇关于并行语言的科普贴（和招生广告）之后，很高兴看到大家对并行算法很有兴趣。许多人留言问，有没有相关的材料可以学习。并行算法和编程作为一个发展很快又相对比较新的领域，目前我们不知道非常合适的中文教材。许多有用的英文材料，也是以美国各大学校并行算法课的 lecture notes 为主。**所以我们就萌生了连载一个专栏，给大家介绍一些并行算法的背景知识的想法。希望可以引起更多同学们的兴趣 :)** 同时，这个连载的内容将会是我下学期在 UC Riverside 的课程（CS260）的一个缩略版。在 UCR 的同学们有兴趣也可以来选课哦。

今天我们简单的说说第一讲。首先，要研究一个并行算法，要考虑的第一个问题是，算法是怎么被并行的？如果我们有若干抽象的*处理器/核*（*processors/cores*）和一些*任务*（*tasks*），他们如何交互？如何访存？如何使用 cache？如何同步？从系统和实现的角度来说，这里有千千万万不同的情况。而我们这个系列的文章会更多的从理论的角度去理解这个问题，也就是说，在这么多不同的硬件setting下，我们首先如何对它们进行*建模*，如何*理论分析*？

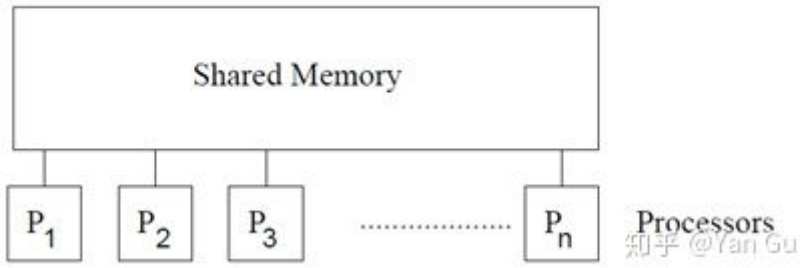
你可能会问，建模和理论分析，对于真正地去写并行算法，有什么用呢？从理论上去理解并行算法，正如你在大学第一门课里学习到的串行算法一样，它会告诉你，O(n log n) 的归并排序一般来讲比 O(n^2) 的冒泡排序效率高。它会告诉你，如果你想找到单源最短路，可以使用Dijkstra算法，如果想维护哈希表，要开出两倍你所要存的数的空间以避免频繁冲突，等等等等。学会了这些东西，不管你以后使用 C++ 还是 java，不管是自己实现还是去调用别人的实现，都可以写出对应的高效的程序来。同样的，学习并行计算不仅仅是学会使用 OpenMP，Cilk，TBB 等这些库的接口，更重要的是，应该怎么去写一个程序。**计算机语言和编程工具是不断发展更迭的，而算法的核心思想总体来说是一致的，一脉相承的。即便未来的并行语言变成了OpenABCDEFG，TAA，TCC, TDD，只要算法的思想对了，你也一样会把并行程序写好。**

我们这个系列文章主要讨论的是 **shared-memory multicore** 的并行算法。顾名思义，这是说多个处理器共用一片（通常来讲无限大的）内存。彼此之间可以通过内存“交流”。那么下一个问题就是，在使用多个核的情况下，我们如何去评价一个并行算法的时间复杂度呢？在传统串行算法设计中，大家对于 RAM（Random Access Machine） 模型应该并不陌生。简而言之，对于 RAM 模型来说，每一次计算和随机访存都花费单位时间。那么加起来总的时间就是这个算法的花费。因为这个模型比较简单，所以通常在大家本科算法课里用来分析复杂度。这个模型的简单性也是算法容易学习和普及的原因之一。当然，后来大家也提出了 I/O 模型等一系列别的计算模型，这里我们就不展开讨论了。总之，要研究并行算法，一个简单而行之有效的模型是非

费的高阶项，忽略常数和低阶项。

古老的 PRAM 模型

并行算法这个问题从上个世纪七八十年代就被数学家们和计算机科学家们提出和研究了，远远在还没有真正的多核电脑之前。科学研究的许多领域都是如此，理论的模型（和想象）通常会领先现实数十到数百年。这些东西可以活在理论家的脑子里，然后可能有一天一个相似的东西就实现了，还发现以前的人 YY 的那些东西挺有用的。对应于 RAM，一个叫做 **Parallel RAM （PRAM）** 的模型曾经在上世纪风靡一时。你去看看有点年纪的教授们，尤其是做理论的老师们，甭管现在干什么的，他们的 paper 历史里，多多少少都在二十多年前研究过那么几个 PRAM 的算法。PRAM 它和 RAM 一样，假设在每个单位时间，每一个 processor 可以做一个计算或者访存的操作。一共 P 个核共享一个 shared memory。像这样：



在这个模型中，通常设计算法就是要告诉所有的 P 个核，每一个单位时间做什么。每一个单位时间过去后，所有的processor们就完成了它们对应的操作，写内存的写内存，做算术的做算术。注意，这里暗含了一个十分强的假设，就是既然所有操作都是单位时间，memory又是共享的，那么这些 processor 们通过这个shared-memory是**高度同步**的，举个例子来讲，这里有两个 processor：

P1:	P2:
A = 3;	C = 1;
B = 2;	A = 6;
D = A + B;	X = 4;

在第三个时间单位，P1就会知道上一个时间里的A已经被P2改成了6，所以最后P1会把D的值改成8。惊不惊喜，意不意外？这件事在我们现在看来，自然是非常不现实的（这一点我们后面会展开讲）。但是大家在上个世纪，真情实感地在这个模型上设计了不计其数的算法……

研究过了。

在 PRAM 里，评价一个算法的的主要指标是它运行的时间 T，也就是一共需要几个并行的时间单位。另一个就是这个算法执行需要的 processor 的数量 P。另外，这个模型通常来讲有几种模式：

> Exclusive read exclusive write (EREW) — 多个processor不可以同时读或者同时写同一个内存位置
> Concurrent read exclusive write (CREW) — 多个processor 不可以同时写，但是可以同时读同一个内存位置
> Exclusive read concurrent write (ERCW) — 不能同时读但是竟然能同时写，这个没什么意义，所以没人用它
> Concurrent read concurrent write (CRCW) — 两个processor既可以同时读也可以同时写同一位置，花费单位时间。关于这个同时写最后结果是什么，也有许多说法，不过我们的重点不在这个模型上，就不展开了。

那么 PRAM 主要的不现实之处有哪些？我们又应该怎么改进呢？首先第一点，在PRAM里，一个算法需要几个 processor，这个 P 是固定的。在我们现代的多核电脑里，虽然你知道它有几个 processor，但是实际上，你不知道你能用几个。你的操作系统可能把其中的几个 processor 分给了的别的应用，即便你在程序运行的过程中，你能用的 processor 数也会动态变化。可能前半截你还有 8 个核可以用，跑着跑着有 4 个就被调度去干别的了，那你这个算法，还能跑不能跑？还对不对？还够不够快？第二点，就是上面说到的高度同步问题。这件事总体来说是非常难的，在现实中，访存比算术计算的花费要大得多，即便同样是访存，数据在不同层的cache里，在内存里，这些花的时间也都不一样。就算都是算术计算，也没有说全都一样快的。如果强行同步，这将可能花费比计算还长的时间。

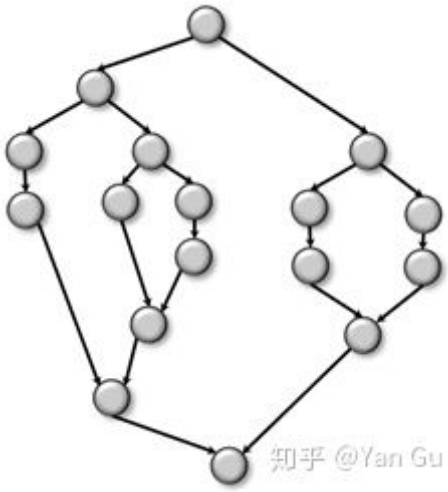
所以在现在，已经很少有人使用这个模型看待并行算法了。而且基于这个模型设计出来的并行算法，有很大可能在现实里也跑不太通。

到这里你可能会拍案而起：我看了这么多，你现在告诉我这个模型其实没用？先不要着急，首先，这是并行算法发展历史上影响比较大的一个模型。当你在读相关文献的时候，很可能会看到一些算法，它们和我们现在的设计思路不太切合，或者得到了一些匪夷所思的 bound。理解 PRAM 的一些简单原理（和局限性）有助于你们理解这些前人的算法，并且从中提取出对自己有用的部分。

那么下面我们讲一个现在大家用得比较多的计算花费模型（cost model），叫做 work-depth 模型。

Work-depth (work-span) 模型

画成一张 DAG（Directed acyclic graph，有向无环图）。其中每一个节点代表一个操作，每一条有向边 A->B 意味着B这个操作必须等待A操作进行完才可以执行。我们管这张图叫 computation graph 或者computation DAG。



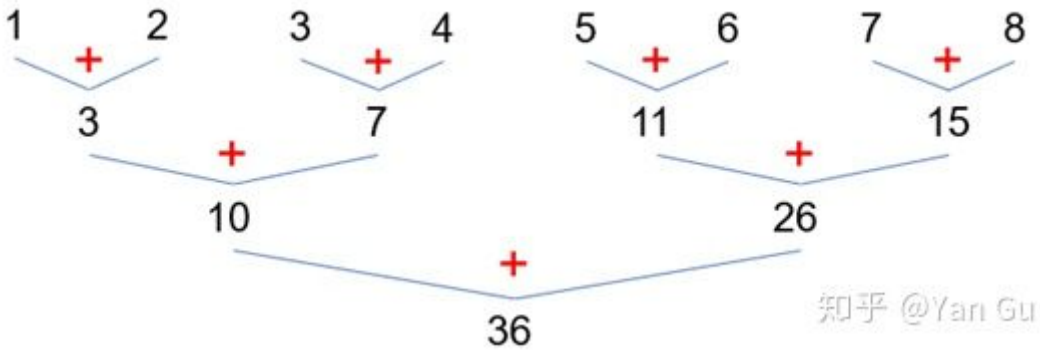
一个算法的代价用两个量来衡量，一个叫 work，用 W 表示，也就是这个图总的结点数。它告诉你这个算法如果你只有一个 processor，串行地跑，它的时间复杂度是什么。比如上图，这个数就是 17。第二就是这个图本身的深度，叫 Depth (D)，也叫 Span (S)，也就是最长的并行依赖链。在上图里，这个数是 8。**这个 depth 告诉你，如果你有无数个 processor，你这个算法需要多少时间**——因为即便你有无数个 processor，这些依赖关系还是要被按顺序一个等一个地执行的。

我们来用一个简单的算法作为例子。比如我现在有一个数组 A，我想计算出它里面所有元素的和。这个操作被称为 reduce 操作，是并行算法里最基础的一个。一个简单的方法大概也许是这样的：如果有 P 个 processor，就把数组拆成 P 份，计算出每个部分里的和，最后把所有的再串行加起来，也就是像这样：

```
Sum(A, n) {
  int B[p];
  start p threads with id 0 to p-1
  In each thread i {
    for (j = i*n/p to i*n/p+p) B[p] += A[j];
  }
  sync all threads;
  for (j = 0 to p) ret += B[p];
  return ret; }
```

不过，和上面提到过的 PRAM 算法类似，它也有这么几个问题：第一，你如何确定你其实有几个可以用的处理器？第二，如果你有三个五个处理器还好，如果你有 n 个处理器，上面这个不是和串行算法没有任何区别？咦？？怎么处理器越多感觉越慢呢？

这个过程，直到只剩一个数，就是我们要的结果。



上图既是这个算法的示意图，也可以直接看成是它的DAG，从上到下就是依赖关系：比如要计算左边的 3+7=10，必须要等 1+2=3 和 3+4=7 这两个操作都完成。写成伪代码可以有递归非递归两种形式：

非递归写法：

```
reduce(A, n) {
  int B[n], B2[n];
  parallel_for i=1 to n B[0][i]=A[i];
  for i = log(n)-1 downto 1 {
    parallel_for j = 1 to 2^i
      B2[j] = B[2*j] + B[2*j+1];
    parallel_for j = 1 to 2^i B[i]=B2[i];}
  return B[0]; }
```

递归写法：

```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A+n/2, n-n/2);
    return L+R;
}
```

这个算法的 work 是 O(n)，这是因为它不管怎么分叉，一共都是要算 O(n) 个加法。它的 depth 是 O(log n)。也就是说，即使你有无穷多个核，你也需要 O(log n) 的时间执行完里面最长的链。

反而是 work（这一点我们马上会展开讲）。第二，work 作为操作总数，它还反映了许多其它的东西，比如能耗，比如占用的总资源。所以让一个算法 work-efficient 是非常重要的。通常来讲，我们设计一个好的并行算法，它的 work 在渐进意义上不应该超过最优的（或者已知最好的）的串行算法的复杂度。这意味着即便你只有一个处理器，或者很少的处理器数量，也可以跑出令人满意的结果。而不是为了并行，引入了更多别的overhead。

那既然我们没有足够多的处理器，depth 又有什么用呢？Depth 这个量主要反映的是，当我们的核，从三个增加到三十个，增加到无穷多个，这个算法的 scalability 如何。如果 depth 很大，有更多的核很快就没有意义了。也就是说，depth 意味着当我们拥有更多的处理器时，算法的性能可以提升的潜能。Depth 我们通常的目标是什么呢？我们希望它是 poly-log的，也就是说，是关于 log n 的一个多项式。这表明这个depth和输入规模 n 比起来不值一提，基本上现实中有的处理器数量都能期望看到好的加速比。即便不行，也至少是 O(n^{\epsilon}), \epsilon < 1 的。

这里多提一点，work-depth 是一个 cost model。它没有指明我在并行的前提下，可以做什么以及怎么做（如并发写，原子操作，新建线程，等等）。它只是告诉你，你有一个算法，在它的依赖关系下，它可能的花费（cost）是什么。

说到这里，你可能还有一个疑问：我是写出了一个并行算法，可是那是伪代码，我怎么让我的处理器去认领这些任务，怎么知道哪些核做哪些加法，怎么保证 load-balancing 呢。如果要把这些东西都写在代码里，难道不是会让实现变得十分复杂吗？这就引入了我们的下一个话题：调度算法。

调度算法

没错，在设计上面的 reduce 的算法设计当中，我们没有具体地说，这些并行的任务，应该怎么分配给处理器。没有像一开始那么具体地讲“分成P份，第 i 个处理器去加第 i 份的和”这样的话。这是因为，在我们的算法设计和真正的底层实现中，我们假设了一个**调度器（Scheduler）**。

调度算法可以说是过去15年来并行计算中最天才的概念。很多人觉得并行很复杂、很系统、很难debug，都是因为没有很好的利用调度算法或者调度器。有了调度算法，并行才能上升到理论的高度，设计简洁又复杂度低的算法才能真正在实际中提升程序的性能。不然，算法再简单，实现的时候也要和复杂的底层系统打交道。这就仿佛给你台电脑让你写个快排，结果你发现得从操作系统写起一样。

调度器就像是一个在我们的并行算法和处理器之间的一个黑盒子。并行算法告诉它：现在我这两个（或者多个）task（我们有时候管这些task们叫线程，threads）可以并行地跑。调度器记住之后，会把这些 task 排队分给当前有空的处理器。为什么说它是一个黑盒子呢？是因为作为算法的设计者，你从此就可以只用关心哪些任务之间能并行，哪些之间有依赖关系，而不用具体操心任务被映射到哪个处理器上了。而这个黑盒子里的具体的调度算法本身，也是有很

调度器对于算法设计者来说，无疑是做了一层有用的抽象。这个东西有效地隔离了算法设计和具体的任务调度，让算法设计变简单。在我们之后的算法设计中，我们都假设有一个有效的调度器帮我们调度任务。

说到这里，你可能又会问：使用调度器多一层操作，不是会比我亲自操作每个核执行什么任务，增加了 overhead 吗？这样不就让我的程序变慢了？其次，调度器它行不行啊？我不亲手把任务精巧地划分好，让每个处理器都接到差不多的任务，总是不大放心。会不会有 load-balancing 的问题啊？

从理论上讲——当然，调度器的方法不会比你用手写出来的*最好*的方法好，至少你总可以（理论上讲）手动分配出一个和调度器一样的方法吧。可是，随着调度器算法的完善，各个并行工具，库等东西的完善，手动操作超过调度器的方法，将会变得越来越难且越来越没有必要。想要理解这个，我们做一个简单的类比。当人类刚开始写串行算法的时候，是写机器码，汇编语言，这些东西都要和底层体系结构直接打交道。那时候，写代码是个很难的事情。全世界能干这件事的人寥寥无几。后来随着高级语言的出现，编译器，解释器的普及，曾经几千行的机器码可能只要几十行的高级语言就能实现了。一个高中生也完全可以自学两天，就写出简单的程序来。当然，就在十几年前，写汇编还是程序员的重要自我修养，因为一旦非常需要高性能的代码，就要用汇编来一段 free，啊不，assembly style，以避免编译器给你编译出来的一些overhead，但是在如今这个年代，这件事的重要性已经大大降低，因为你用手写的大量汇编代码，已经很难赛过发展了几十年的C语言编译器了。更何况对于复杂的大型的工程来讲，把它们用汇编写出来都近乎不可能了，更不要提考虑 debug 这样的问题了。

所以，做一个不一定非常精确的类比，如今我们看待并行算法和并行编程，也是这么个道理。大家总觉得并行编程难，尤其是并行代码debug起来如同愤怒的男/女朋友一样蛮不讲理。这是因为我们看待并行编程的时候，一眼就看到了和底层体系结构打交道的那一步。cache，memory，处理器，同步，通信，并发，冲突，内存管理，load-balancing……这些东西搅和在一起，并行算法当然难了。可是，这时候你应该问问自己：**凭什么？？我要干这些？？？我写串行算法的时候受这些委屈了吗？？我像舔狗一样为我的代码费了这么大劲处理这些东西，有用吗？？值得吗？？** 正如编译器把我们写串行算法的时候把我们和底层隔开一样，调度器这个时候就应该起到同样的作用。设计算法只需要纯粹地从问题出发，电脑有几个处理器，每个处理器什么时候干些什么，会不会运行到一半少了两个能用的处理器，都不再重要了。你就可以优雅地看看算法的 work，depth，写写算算，问题就解决了。**只有让编程这件事变得简单，才能写出功能更复杂的程序。**

那让我们来看看，一个调度器可以怎么调度上面的程序。我们回顾一下这张图。

显然，如果有 p 个核执行这 n 个加法，一个最简单的方法就是——按顺序从上到下执行。按照拓扑排序（topological sort）的顺序，先拿出最早的，相互不依赖的，最多p个任务一起执行，然后再拿最多 p 个，以此类推。在这个例子里，大致来说，就要 O(n/p) 的时间。想象一下 p=2，那就是每次从上到下从左到右按顺序用这两个处理器执行两个加法，需要 n/2 的时间。如果 p 越变越大呢？最快也不能超过O(log n)吧，因为 1->3->10->36 这条链必须花四轮进行（这就是 depth 的意义）。总体来说，你可以证明这种分配任务的方法在 O(n/p+log n) 的时间里可以把这 n 个加法做完。

这就是最简单的一个调度算法。可以证明，给定一个 DAG，W 的 work 和 D 的 depth，你总是可以通过一定的调度算法，在 W/p+O(D) 的时间内给它执行完。最简单的方法就是像上面那样（大家可以自行证一证，上面的算法其实是 W/p+D）。这个结果可以说是相当棒的，为什么呢，因为它也是这样一个 DAG 执行时间的下界（lower bound）。W 的 work，p 个处理器，就算任何时候每个处理器都忙得团团转，完全平衡地分配所有任务，也得要 W/p 的时间吧。O(D) 这一项就不用说了，你就算是有无穷个处理器，也要这么多时间吧？所以就这么一个简单的调度算法，渐进程度上就已经是最优的了。所以，放心把这些事情都交给它们吧！你再怎么使劲用手调度，划分任务，也就是和调度器在渐进程度上一样的。

当然具体实现一个调度器的时候，没有这么简单，调度器本身自己的实现也是很 tricky 的，因为调度器本身就是一个和底层操作打交道的，你可能并不想写的，并行算法啊！不过这里就不展开了。还有一个事情是，现实中的任务都是动态生成的，不是程序一运行，就把这个 DAG 交给调度器的。不过放心，调度器也能处理这种情况。Again，这里不细讨论调度器的实现，只是让大家理解一下它的原理，有机会我们开个专题讲一讲~

回到这个 W/p+O(D) 的值上来——目前来讲，我们的多核系统所能使用的处理器数 p 和我们的 work W（通常至少是O(n)的吧）相比，可以说小得可以当成一个常数，所以一般来讲，这个数是被 W/p 所主导的。这也就是前面所说的，为什么在实际中 W 比 D 对并行时间的影响还大，也是为什么我们要设计算法，work-efficiency 是很重要的。

很多已有的并行语言库都支持调度器，很多人也喜欢自己写自己的调度器，不管怎么说，都是调度器和真正的算法隔离开，做到互不影响。我个人用 cilk 用得比较多，这里用 cilk 的接口给大家举个例子：

```
  cilk_for (int i=1; i<n; i++) B[i]=A[i];
  for (int i = ceil(log(n)); i>1; i--) {
    cilk_for (int j = 1; j < power(2, i); j++) {
      B2[j] = B[2*j] + B[2*j+1];
    }
    cilk_for (int j = 1; j < power(2, i); j++) B[j] = B2[j];
}
  return B[0];
```

分支递归写法：

```
int reduce(int* A, int n) {
    if (n == 1) return A[0];
    L = cilk_spawn reduce(A, n/2);
    R = reduce(A+n/2, n-n/2);
    cilk_sync;
    return L+R;
}
```

这里 cilk_for 就是一个并行的 for 循环，这是告诉调度器，新建出 n 个这样的任务（每一个有一个 i 的值做参数）等着去执行。下面的例子里，cilk_spawn 的意思是告诉调度器，下面这句话，是新建一个并行的任务去执行它的。也就是告诉调度器：计算 L 的那句话，可以给另一个处理器去做。cilk_sync 是说，等上面的任务都执行完了在这里同步一下。这时候调度器就知道了：这句话执行到这儿的时候，要去看看L算完没？然后安排好这一切，你作为程序员，就高枕无忧地等着它在 O(n/p+log n) 的时间里跑完就好了，哪个处理器做了哪些加法，你才不用管呢。

如果你使用OpenMP，也可以用类似的接口写出这样的程序。这里就不赘述了。

此外注意，这两个算法直接写成这样都不并不能达到最好的并行效果。比如第一个程序对 B 数组多了一次复制增加了很多 I/O。而对于第二个程序，我们还可以里再加一个粒度控制（granularity control），当比如子问题规模小于 1000，而不是等于 1 的时候我们就切换到串行算法以避免递归、cilk_spawn 和 cilk_sync 的开销。这个合适的粒度控制和你的系统参数和平台都会有关系，不过这也是整个实现中唯一需要额外考虑的部分。这里两个代码只是示例，要是想写得再好一点还要动动脑筋哟。

另一个值得一提的事情是，上面的两个写法，都是race-free的，不存在两个处理器同时处理一个内存单元的情况。此外，第一个写法中的B数组，在任何一个指定的轮（某一个 i 值）它的值确定的，如果需要debug，它的表现应该是非常确定性的，可预测的。倒不是说这么简单一个算法还需要de多少bug，而是在更复杂的算法中，我们也要怀着这种思想，设计出不仅单，编程友好的算法。

在第一讲的最后，还想多说几句，那就是，说了这么多，我们为什么要care并行？既然这个系列的文章缘起招生广告，那么……来干这个有前途吗？？**首先，并行的唯一目标，就是让算法跑的比谁都快。**回忆一下上一篇文章里的第一张图，在二十年前，单核处理器的性能还一路狂飙的时候，每 18 个月处理器性能翻一番的故事，大家想必还记得。那时候，不仅并行程序十分难写，而且处理器性能提高飞快。**要是我写个并行程序花了一年半，性能提高了一倍，我还不如坐在家里等一年半买台新电脑跑我的串行代码。**这也是为什么 PRAM 算法大家研究了十几年，结果最后都纷纷转行干别的去了——大家既看不到它们到底怎么能很好的实现，也不知道实现了有没有意义。

而那条性能提升的曲线，在 2005 年左右，就缓和了下来。这件事和并行计算机可以说是相互成就。首先，正是因为单核性能提高变得困难，硬件厂商研制多核计算机才变得迫在眉睫。其次，正是因为那时候Intel的双核计算机横空出世，才使得提升单核性能这件事，它变得不再重要了。于是大家也就不再死磕主频和单核性能这种事情，转而去研究怎么往电脑里多放几个核了。**看到了吧，以前你的串行算法每一年半自己跑快一倍的事情翻篇了，现在电脑的性能还在提升，但是你想利用这个提升，得想着怎么同时利用好几个核了。**

另一件重要的，有点老掉牙的理由是，毫无疑问，我们处理的数据越来越大。许多现有的数据光用单核读一遍就要好几小时甚至好几天。不并行根本无法处理和分析这些数据。所以，并行和许多领域都是息息相关的。**抽象地去研究并行算法问题，这个思想会对很多其它的应用领域起到帮助**，比如并行的数据库，数据挖掘，计算生物学，计算几何学，图形学，等等等等。将这些领域中一个成熟的串行算法的效率提高20倍很难，但是只要并行一下就能得到比这更好的效果。

另一个问题是比如最近ML、神经网络之类的 topics 很热门，这些和 reduce、prefix sum、排序的并行有关系么？如果你觉得学习算法有用，那么学习并行算法也就有用。前一阵国内某大厂的人和我们讨论过一些 ML 中的并行问题。这里的问题虽然 train 好了神经网络，但是怎么并行的 evaluate 很多 input 然后截取最好的输出，其实和如何并行的排序很多个整数在思想上是大同小异的，只不过对象从整数变成了神经网络而已。并行的思想，也就是 parallel thinking，他的用处是远远超出课上举例的几个算法的。

因此，计算机算法和编程未来的发展，并行是一个绕不过去的主题。更何况现在几乎所有的设备，哪怕是笔记本和手机，都变成多核了。想要利用计算资源，就要利用好多核。**我们相信，正如“写程序”这个事情变得越来越普及一样，“写并行程序”也会逐渐变成人们习以为常的事情。**如果你想从事这方面的研究，那就是为了要让这一天来得快一些。如果你只是想学习，那就是为了早一点体会到到了那一天，写并行程序得心应手的感觉 :)

非常感谢大家的阅读，这一篇是并行算法科普小文章的第一篇，所以尽量还是讲得比较浅显。其中有很多概念上的东西，没有涉及到太多的算法（不如说只讲了一个reduce……）。如果大家有什么看法或者好的建议，欢迎给我们留言。下一篇（虽然不知道什么时候才能写）会给大家简单地一些经典的并行算法，前缀和（prefix sum，又叫scan），矩阵乘法（matrix multiplicatio以及如何使用一些并行语言实现这些算法，等等。后面我们还希望能介绍其它有用

此外，本系列文章和我的课程中许多资源，图片，和思想来源于许多老师在别的学校的课程，课件和 lecture notes。虽然他们好像基本上都看不懂中文，我还是在这里向他们表示感谢。这些课程包括 CMU 的 15-210, 15-853, 15-897, 15-418（这门课也在清华暑期开过），和MIT 的 6.172, 6.886。

本系列其它文章：

Yan Gu：多核时代与并行算法
zhuanlan.zhihu.com

Yan Gu：并行算法科普向 系列之一：计算模型，调度器，和其它
zhuanlan.zhihu.com

Yan Gu：并行算法科普向 系列之二：前缀和，fork-join 和矩阵乘法
zhuanlan.zhihu.com

Yan Gu：并行算法科普向 系列之三：归并与归并排序，过滤与快速排序

编辑于 2019-11-22

并行计算 算法 计算机科学

文章被以下专栏收录

多核时代与并行算法 已关注

推荐阅读