**Amdahl's Law**

* [Amdahl's Law Defined](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html#amdahls-law-defined)
  + [A Calculation Example](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html#a-calculation-example)
* [Amdahl's Law Illustrated](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html#amdahls-law-illustrated)
* [Optimizing Algorithms](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html#optimizing-algorithms)
  + [Optimizing the Sequential Part](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html#optimizing-the-sequential-part)
* [Execution Time vs. Speedup](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html#execution-time-vs-speedup)
* [Measure, Don't Just Calculate](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html#measure-dont-just-calculate)

Amdahl法则可以被用于计算通过并行运行一部分计算可以提速多少计算。Amdahl法则被以Gene Amdahl命名，他在1967年提出这个法则。大部分并行或者并发系统工作的开发者对潜在加速有直观的感觉，甚至不用了解Amdahl法则。不管怎样，Amdahl法则仍然有用。

我将首先在算术上解释Amdahl法则，然后前进到使用图表说明Amdahl法则。

**Amdahl's Law Defined**

一个可以被平行化的程序(或者算法)能被拆分到两部分：

* 一部分不能被平行化
* 一部分能被平行化

想像一个程序处理来自硬盘的文件。程序的一小部分可能检测文件目录并在内存内部创建一个文件列表。在那之后，每个文件被传递给一个单独的线程来处理。扫描文件目录并创建文件列表的部分不能被平行化，但处理文件的部分可以。

序列化(不是平行化)执行程序花费的总时间被称作T。时间T包含非平行化和平行化部分的时间。非平行化的部分被称作B。平行化部分被指代为T - B。下面的列表将这些定义加和：

* T = 序列执行总时间
* B = 非平行化部分总时间
* T - B =平行化部分总时间(当序列化而非平行化执行)

由此得出如下结论：

T = B + (T-B)

起初看起来可能有点奇怪，等式中程序的平行化部分没有它自己的符号。然而，因为等式的平行化部分可以被表达使用总时间T和B(非平行化部分)，等式实际上已经在概念上被减少了，意味着它包含更少的不同变量在这个公式中。

正是平行化部分，T - B，可以被提速通过平行执行。它可以被提速多少取决于你应用多少线程或者CPU去执行它。线程或者CPU的数量被叫做N。平行化部分能被最快执行是如此：

(T - B) / N

另一种方式写这个是：

(1/N) \* (T - B)

维基使用这个版本在你读到关于Amdahl法则的地方。

依据Amdahl法则，当平行化部分被执行使用N个线程或CPU时，程序的总执行时间像这样：

T(N) = B + (T - B) / N

T(N)意思是总执行时间，带有一个平行因子N。因此，T可以被写作T(1)，意思是总执行时间，带有平行因子1.使用T(1)而不是T，Amdahl法则看起来这样：

T(N) = B + ( T(1) - B ) / N

不过它意思一样。

**A Calculation Example**

为了更好理解Amdahl法则，让我们过一遍一个计算例子。执行一个程序的总时间被设置为1.程序非平行化部分是1的总时间的40%，等于0.4。平行化部分因此等于1 - 0.4 = 0.6。

带有平行因子2(2哥线程或CPU执行平行化部分，因此N是2)的程序的执行时间将会是：

T(2) = 0.4 + ( 1 - 0.4 ) / 2

= 0.4 + 0.6 / 2

= 0.4 + 0.3

= 0.7

做相同的计算带有平行因子5而不是2将看上去这样：

T(5) = 0.4 + ( 1 - 0.4 ) / 5

= 0.4 + 0.6 / 5

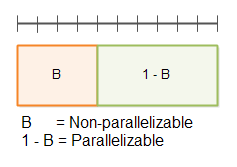
= 0.4 + 0.12

= 0.52

**Amdahl's Law Illustrated**

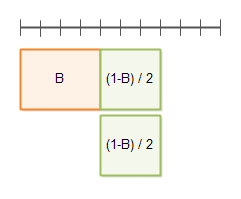
为了更好理解Amdahl法则，我将会试着说明法则怎么被被衍生。

首先，一个程序能被拆散进一个非平行化部分B，和一个平行化部分1-B，正如在图表中说明的：

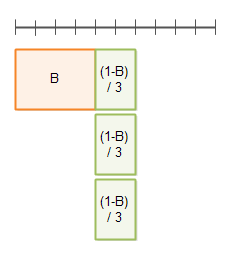


在顶部有定界符的行是总时间T(1)。

在这里，您可以看到并行化因子为2的执行时间：



这里你可以看到并行化因子为3的执行时间：



**Optimizing Algorithms**

从Amdahl法则来看，它自然遵循，并行化部分可以通过在其上扔硬件来更快地执行。更多的线程/CPU。非平行化部分，然而，只能通过优化代码被执行更快。因此，你能提升你的程序的速度和平行程度通过优化非平行化部分。你甚至可以改变算法来得到一个较小的非平行化部分，通过移动一些工作到平行化部分内(如果可能)。

**Optimizing the Sequential Part**

如果你优化一个程序的序列部分，你也可以使用Amdahl法则来计算程序优化之后的执行时间。如果非平行化部分B被一个因子O优化，那么Amdahl法则看起来这样：

T(O,N) = B / O + (1 - B / O) / N

记住，程序的非平行化部分现在花费B/O时间，因此平行化部分花费1- B/O时间。

如果B是0.4，O是2，N是5，那么计算看上去这样：

T(2,5) = 0.4 / 2 + (1 - 0.4 / 2) / 5

= 0.2 + (1 - 0.4 / 2) / 5

= 0.2 + (1 - 0.2) / 5

= 0.2 + 0.8 / 5

= 0.2 + 0.16

= 0.36

**Execution Time vs. Speedup**

到现在我们只用了Amdahl法则计算一个程序或者算法优化后或者平行化后的执行时间。我们也能使用Amdahl法则计算speedup，意思是新算法或者程序比旧的版本快多少。

如果旧版本程序或者算法的时间为T，那么提速将为

Speedup = T / T(O,N)

我们通常设置T为1，只为了计算执行时间和提速作为旧时间的一小部分。等式然后看起来这样：

Speedup = 1 / T(O,N)

如果我们插入Amdahl法则计算而不是T(O,N)，我们得到以下的公式：

Speedup = 1 / ( B / O + (1 - B / O) / N )

随着B=0.4、O=2、N=5，计算变成：

Speedup = 1 / ( 0.4 / 2 + (1 - 0.4 / 2) / 5)

= 1 / ( 0.2 + (1 - 0.4 / 2) / 5)

= 1 / ( 0.2 + (1 - 0.2) / 5 )

= 1 / ( 0.2 + 0.8 / 5 )

= 1 / ( 0.2 + 0.16 )

= 1 / 0.36

= 2.77777 ...

那意味着，如果你优化非平行化(序列化)部分通过因子2，平行平行化部分通过因子5，程序或算法的新的优化版本将运行最大2.77777倍快于旧版本。

**Measure, Don't Just Calculate**

当Amdahl法则让你能够计算一个算法的平行化的理论提速，不要太依赖于这种计算。实际上，许多其它的因素可能起到作用当你悠哈或者平行化一个算法时。

内存、CPU缓存、硬盘、网卡等(如果硬盘或网络被用到)的速度也可能是一个限制因子。如果一个新版本的算法是平行化的，但导致更多的CPU缓存丢失，你可能甚至不能得到期望的使用x N 个CPU的x N倍提速。如果你最终内存总线、硬盘或网卡或网络连接饱和也会这样。

我的建议可能是使用Amdahl法则来得到一个关于优化哪里的想法，但使用一种测量手段确定优化的真实提速。记住，有时一个高序列化(单CPU)的算法可能比一个平行算法表现出色，仅仅因为序列版本没有合作开销(打碎工作，再搭建整体)，并且因为单CPU算法可以更好地符合底层硬件的工作方式（CPU流水线、CPU缓存等）。