第八讲：程序性能评价与优化

陈俊清

jqchen@math.tsinghua.edu.cn

清华大学数学科学系

April 26, 2013

Part I

并行程序性能评价

Outline

Outline

并行程序的执行时间

- 计算CPU时间：可分为两部分，一部分为程序本身执行指令所占用cpu时间，即通常所说的用户时间（user time），包括在cpu内部的执行时间和内存访问时间。另一部分为维护程序的执行，操作系统花费的cpu时间，即系统时间（system time），包括内存调度和管理、I/O时间，以及维护程序执行所必须的操作系统开销。
- 通信CPU时间：指任务进程进行消息传递所需的处理机cpu时间，如数据包的形成、发送、接收、解包等。

并行程序的执行时间

- 同步开销时间：指并行算法在各进程的执行过程中，为确保各处理机的正确工作， 以及对共享可写数据的正确访问，程序员需在算法程序的适当位置放置同步点。 因此，同步开销即指在时间上强行使各执行进程在某一点相互等待而引起的时间开销。
- 进程空闲时间：当一个进程阻塞式等待其他进程的消息时，cpu通常是空闲的， 或者处于等待状态。进程空闲时间是指并行程序执行过程中，进程所有这些空闲时间的总和。

后三类时间统称为并行开销时间。

算法的并行度与粒度

算法的并行度

算法的并行度是指该算法中可并行执行的单位操作数。

例如，设a、b是两个长度为n的向量， 其对应的向量之和为：

$$a_i + b_i, i = 1, 2, , \ldots, n$$

因为向量a和b的各分量是独立的，加法可并行执行，所以该算法的并行度为n。
操作是计算的基本单位，可以指加、减、乘、除等基本算术运算，也可指某一任务级的作业，视具体应用问题而定。
注： 并行度仅与算法相关，与具体并行机的规模无关，是算法内在并行性的固有属性。

算法的并行度与粒度

算法的粒度

与并行度相关的一个概念，大粒度意味着能独立执行的是大任务，小粒度意味着能独立执行的是小任务。

粒度是一个相对概念，它与算法固有的并行度和具体的并行机相关。
如：内在并行度为100个单位操作的某个算法，相对于每秒只能执行一个单位操作的处理机来说是大粒度， 而相对于每秒能执行100个单位操作的处理机来说则是小粒度。
粒度大的并行算法称为粗粒度并行算法，粒度小的并行算法称为细粒度并行算法。

Outline

算法的加速比和并行效率

并行算法的加速比和效率是评价并行算法的两个重要指标。它们体现在并行机上使用并行算法求解问题获得的好处。

- 绝对加速比:

$$S_p(n) = \frac{t_s(n)}{t_p(n)}$$

其中，$t_s(n)$为求解一个规模为n的问题的最快串行算法在单处理器上的运行时间，$t_p(n)$为求解该问题的并行算法在p台处理器上的运行时间。

- 相对加速比:

$$S_p(n) = \frac{t_1(n)}{t_p(n)}$$

其中，$t_1(n)$为求解一个规模为n的问题的并行算法在一台处理器上的运行时间。一般情况下，并行程序会引入一些冗余开销，使得$t_1(n) > t_s(n)$

算法的加速比和并行效率

- 算法的效率：

$$E_p(n) = \frac{S_p(n)}{p}$$

如果并行算法的加速比$S_p(n)$与处理器数成正比，则称该并行算法在该条件下， 在该并行机上具有线性加速比，这是人们所期望的。
若$S_p(n) > p$，则称为该算法具有超线性的加速比，一般情况下，$S_p(n) \leq p$，所以$E_p(n) \leq 1$.
注：上述定义有一个隐含假定：并行计算机的各个处理器是同构的（homogeneous）。

算法的加速比和并行效率

影响加速比的因素：

- 问题的串行部分所占比例
- 大量的通信和同步操作
- 负载不平衡
- 资源冲突：对单一资源的争夺会限制算法的加速比（主要指数据的争夺）。
- 算法的有效性：算法的有效性对加速比影响很大。对一个给定的问题来说，要尽量研究有效的算法。一个有效的算法一定要做到进程间同步减至最少，即两个同步点之间的工作粒度要尽可能大；处理机工作要 负载平衡；所设计的算法的时间复杂性尽可能低。

Outline

并行计算加速比模型

并行计算的加速比是指对于一个给定的应用，并行算法（或并行程序）的执行速度相对于串行算法（或串行程序）的执行速度提高的倍数。
下面着重介绍三种加速比模型：

- 固定计算规模的加速比模型；
- 固定时间的加速比模型；
- 固定存储的加速比模型；

固定计算规模的Amdahl加速比模型

Amdahl加速比模型强调的是通过并行处理来缩短求解问题的时间。主要针对科学与工程计算中对实时性要求很高的问题， 时间是关键因素，而计算工作负载w(或问题规模、或工作量)是固定不变的。
在一定的工作负载下，通过增加处理机的台数来缩短求解问题的时间，从而达到加速的目的。
固定工作负载的加速比（Amdahl,1967）公式:

$$S_p = \frac{f+n}{f+n/p} = \frac{1}{f+n/p}.$$

其中整个负载w分为只能串行部分w_s和可以由任意多个处理器同时并行执行的部分w_p之和， $f = w_s/w$表示只能串行部分所占的比例，$n = w_p/w$表示可并行部分所占比例，$n = 1 - f$， p表示并行系统的处理器个数。

固定计算规模的Amdahl加速比模型

由Amdahl加速比公式，可知

$$S_p \to \frac{1}{f}, p \to \infty$$

即：对给定负载的问题，若串行部分所占比例为f，则不论增加多少个处理器，其加速比都不会超过$\frac{1}{f}$。

即问题的串行部分无法通过增加处理器来解决。

固定计算规模的Amdahl加速比模型

加入并行计算时的额外开销(w_0)，则固定问题规模的加速比为：

$$S_p = \frac{(f+n)w}{w(f+n/p)+w_0} \rightarrow \frac{1}{f+w_0/w}, p \rightarrow \infty$$

上式说明，要提高加速比，不仅要减少串行计算部分，而且要增加平均计算粒度，也就是说加速比不仅受串行部分限制， 而且还受限于平均开销。

去掉固定问题规模的限制，由于扩充问题规模的程度与计算机内存和求解问题的运行时间有关，受这两者的限制， 又产生了固定执行时间的加速比和固定存储量的加速比。

固定时间的Gustafson加速比

固定时间的加速比（Gustafson,1988）公式为：

$$S_p = \frac{w(f + n \times p)}{w(f + (n \times p)/p)} = f + n \times p.$$

时间固定的加速比是p的线性函数，而串行部分不再是瓶颈。

Gustafson模型强调的是：在同样时间里，通过并行计算能运行多大规模的问题， 也就是通过固定的运行时间来限制问题规模的增长程度。
加入并行计算的额外开销(w_0)，完整的时间固定加速比公式为：

$$S_p = \frac{(f + n \times p)w}{(f + (n \times p)/p)w + w_0} = \frac{f + (1 - f)p}{1 + w_0/w}$$

固定存储量的Sun-Ni加速比模型

Sun-Ni加速比模型是对前两个加速比模型的推广。其基本思想是:

在并行机存储空间有限的条件下求解尽可能大规模的应用问题（可能执行时间略有增加）， 以获得较高的加速比、较高计算精度和较好的资源利用率。

该模型要求问题规模的增长程度必须受存储量的限制，它的加速比更接近于线性加速比。 其公式为

$$S_p = \frac{fw + (1-f)G(p)w}{fw + (1-f)G(p)w/p} = \frac{f + (1-f)G(p)}{f + (1-f)G(p)/p}$$

其中$G(p)$表示存储量增加p倍后，问题规模的增加量。

固定存储量的Sun-Ni加速比模型

增加并行计算的额外开销（w_0）后，完整的公式为：

$$S_p = \frac{fw + (1-f)G(p)w}{fw + (1-f)G(p)w/p + w_0} = \frac{f + (1-f)G(p)}{f + (1-f)G(p)/p + w_0/w}$$

与前两种模型之关系：

- $G(p) = 1$,Sun－Ni加速比⇒ Amdahl加速比。
- $G(p) = p$,Sun－Ni加速比⇒ Gustafson加速比。
- $G(p) > p$,表示问题规模增加的速度比存储规模增加的快，Sun－Ni加速比要比另外两个高。

Outline

并行程序性能评价

加速比和效率只能反映并行程序的整体执行性能，无法反映并行程序的性能瓶颈。 因此有必要进一步分解加速比和效率，提出更细致的性能评价方法。

性能评价的目的：揭示并行程序的性能瓶颈，指导并行程序的性能优化。

浮点峰值性能与实际浮点性能

- 微处理器的浮点峰值性能等于CPU 内部浮点乘加指令流水线的条数、 每条流水线每个时钟周期完成的浮点运算次数、处理器主频三者的乘积。
- 并行机的峰值性能等于处理器峰值性能和处理器个数的乘积。
- 并行(串行)程序的实际浮点性能等于并行程序的总的浮点运算次数和 并行（串行）程序执行时间的比值。

为了获得处理器的浮点峰值性能，所有浮点乘加指令流水线必须不间断运行。 但通常情形下，程序是无法达到峰值性能的。一般的串行程序也只能发挥峰值性能的几个到十多个百分点。 同样，并行程序的实际浮点性能是无法达到并行机峰值性能的。

数值效率和并行效率

将并行程序的墙上时间分解为：

$$T_p = C_i + D_i, i = 1, 2, \ldots, p$$

其中，C_i 为第i个进程花费的CPU 时间，D_i 为第i 个进程的 空闲时间。进一步分解，有：

$$C_i = L_i + O_i, i = 1, 2, \ldots, p$$

其中，L_i 为第i 个进程数值计算指令执行花费的CPU 时间，O_i 为第i个进程通信、同步花费的CPU 时间。

数值效率和并行效率

- 并行计算粒度： 进程指令数值计算时间与墙上时间的比值，即：

$$\gamma_i = \frac{L_i}{T_p}, i = 1, 2, \ldots, p$$

- 非数值冗余：由于并行引入的额外非数值计算开销

$$W_i = D_i + O_i, i = 1, 2, \ldots, p$$

- 负载平衡：为了减少无谓的空闲时间，各个进程分配的CPU 时间尽 量相等，为此，定义负载平衡效率如下：

$$\eta_p = \frac{\sum_{i=1}^{p} C_i}{\max_{i=1}^{p} C_i \times p} = \frac{\frac{1}{p}\sum_{i=1}^{p} C_i}{\max_{i=1}^{p} C_i}$$

数值效率和并行效率

基于以上时间分解公式，$C_T = \sum_i C_i, D_T = \sum_i D_i$,则$C_T + D_T = p \times T_p$，效率公式可以写成:

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p} = \frac{T_s}{C_T} \times \frac{C_T}{C_T + D_T}$$

分别定义数值效率和并行效率如下:

$$\begin{cases} NE_p = \frac{T_s}{C_T} & \text{(数值效率)} \\ PE_p = \frac{C_T}{C_T + D_T} & \text{(并行效率)} \end{cases}$$

则

$$E_p = NE_p \times PE_p$$

数值效率与并行效率

- 数值效率反映了并行计算引入的额外CPU计算时间开销，这种开销来自两个方面。一方面，并行执行时，随着处理器个数的增长，各进程的cache命中率将提高，有助于缩短总的计算时间。另一方面，并行计算可能引入额外的开销：增加计算量，通信、同步等，延长计算cpu时间，降低数值效率。若缩短的cpu时间小于额外开销，数值效率会大于1。
- 并行效率反映并行程序具体执行的并行性能，它依赖于并行计算机网络的通信性能， 以及并行程序的负载平衡等方面，并行效率总是小于1。

数值效率低，说明并行算法或程序设计引入额外cpu时间太大；并行效率低， 说明计算机网络通信同步导致的进程空闲时间太长。

负载平衡对并行程序的性能影响

假设并行程序总共需要计算100 个时间单位，并取P=4。

- 如果负载是平衡的，每个进程分配25 个单位，则可获得加速比4，负载平衡 效率为100%。
- 假设负载不平衡，某个进程分配50 个单位，2 个 进程各自分配20 个单位，而另外一个进程分配10 个单位，则 并行执行时间依赖于执行最慢的进程，因此，负载平衡效率仅为50%，导致加速比仅为2。

为了缩短并行程序的墙上时间，应该极小化非数值冗余 W_i，极大化并行计算粒度，保证负载平衡。

Outline

可扩展性的基本概念

可扩展性(scalability)是人们设计高性能并行机和并行算法所追求的一个重要目标。 它描述了能用增加处理机来计算更大规模问题的能力，也描述了增长的问题规模对增加的处理机资源的利用程度。

- 在给定的并行系统上运行一个给定的应用问题（并行算法）时，如果随并行系统规模增大而适当增加问题的规模， 使并行系统的性能与其规模成线性比例增长，则称该并行系统是可扩展的。
- 运行于一台给定的并行系统上的并行算法（并行应用程序），当系统的规模与算法的规模按比例增加时，其性能也按一定的比例提高， 则该并行算法（并行应用程序）是可扩展的（或称算法与系统的组合是扩展的）

可扩展性的基本概念

研究可扩展性的目的:

- 确定某类问题的何种并行算法与何种并行系统的组合，可有效地利用系统大量处理机;
- 由算法在小规模并行机上的运行性能，来预测其移植到大规模并行机上的运行性能;
- 对于一个固定规模的问题，能确定其运行在某类并行系统上时，所需的最优处理机台数和获得的最大加速比;
- 指导并行算法和并行机体系结构的改进。

可扩展性的基本概念

可扩展性的参数：处理机台数p，问题规模w。 通常增加p和w，能提高加速比，但p的增加会引起额外开销以及超过问题的并行度。因此在增加p和w的时候，事先要研究如下问题：

- 额外开销的增加是否快于计算的增加；
- 较大的w是否能提供较高的并行度；
- 算法中串行部分所占的比例是否随w的增大而缩小。

可扩展性分析方法

等效率度量（Kumar 1987）：
对于某类算法和并行机，如何保持问题规模w 与处理器个数p 之间的关系$w(p)$，使得随着处理器个数p 的增长，保持并行计算的效率不变。
也就是求出等效率函数(ISO-efficiency function)：

$$w = f_E(p), (\text{E固定})$$

等效率值越小，则当处理器个数增加时为保持相同效率所需增加的问题规模就越小，因此就有更好的可扩展性。

可扩展性分析方法

等效率是指系统规模（p）增加时，测量增加多少运算量（问题规模w）会保持并行算法的效率不变。

$$E_p = \frac{T_1}{pT_p} = \frac{t_c \cdot w}{t_c \cdot w + T_0(p,w)} = \frac{1}{1 + T_0(p,w)/(t_c \cdot w)}.$$

由此可得:

$$w = \frac{1}{t_c}(\frac{E_p}{1-E_p})T_0(p,w) = KT_0(p,w).$$

其中

$$K = \frac{1}{t_c}(\frac{E_p}{1-E_p}).$$

t_c为并行机上每个基本操作的平均时间，$T_0(p,w)$为并行计算时总的非数值冗余（并行的开销），$t_c \cdot w$ 为算法的串行时间。

可扩展性分析方法

当E_p不变时，可得到$w = f_E(p)$，当$f_E(p)$为线性函数时，称并行算法是线性可扩展的，等效率函数$f_E(p)$反映下面几个方面的问题:

- 效率不变时，问题规模与系统规模之间的匹配关系。
- 固定问题规模，效率随系统规模p增加而降低，由于系统开销随p增大而增大，处理器的利用率因此降低。
- 对于一定规模的并行系统，效率会随问题规模w的增加而提高。所以随系统规模的增加，适当增加问题规模，可以保持效率不变。

弱点：没有反映出p增加时，体系结构的计算与通信能力变化对算法内在并行性与通信需求的影响。 有时为了维持效率不变，随p的增加，w往往以多项式甚至以指数规模增长，从而超出并行机的有限存储容量。

可扩展性分析方法

等速度度量(ISO-speed Metrics)（Sun 1994）：

对于运行在并行机上的某个算法，当处理器个数增加时，需要增加多大的计算量，才能保持并行程序的平均速度不变。 定义平均速度$\bar{V} = \frac{V}{p} = \frac{w}{pT_p}$,V为并行程序的执行速度，问题规模从 $\{w, p\}$变化到$\{w', p'\}$，则等速度可扩展度量公式可写为：

$$\Psi(p, p') = \frac{w/p}{w'/p'} = \frac{\bar{V} \times T_p}{\bar{V} \times T_{p'}} = \frac{T_p}{T_{p'}}$$

$0 < \Psi(p, p') < 1$，$\Psi(p, p')$越接近与1，说明可扩展性越好。

可扩展性分析方法

优点:

- 包含了问题规模和执行时间两个因素，问题规模描述了应用问题的性质，执行时间反映机器性能和程序效率，包括计算时间和开销时间;
- 速度对不同机器来说是公平的;
- 问题规模由计算中的浮点操作数决定，容易估计。

缺点:

- 忽略了非浮点操作所引起的性能变化;
- 算法开销包含在整个执行时间中，而在总浮点数w中没有包含;
- 没有反映出并行算法与体系结构的匹配程度。

可扩展性分析的例

例子：下面考虑一个矩阵乘法的简单并行算法的扩展性分析；
设A和B都是$n \times n$阶矩阵，A，B的乘积矩阵为C，传统串行算法为：

$$A \times B = C = (c_{ij})_{n \times n}, c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$A \times B$最好的串行算法需要n^3个基本操作（这里一个基本算法包括1个乘法和一个加法），即问题的规模 $w = n^3$,设执行一次基本操作的时间为1，则串行程序的运行时间为 $T_1 = n^3$.

可扩展性分析的例

设p为某一整数m的平方（$p = m^2$），将A、B与C以适当的方式分为p块$A_{i,j}, B_{i,j}, C_{i,j}(0 \le i, j \le m-1)$，每块为 $n/m \times n/m$阶子矩阵，将这些子矩阵映射到$m \times m$个逻辑处理机阵列上，每个处理机记为$p_{i,j}$,$p_{i,j}$上存储$A_{i,j}, B_{i,j}$，并计算$C_{i,j}$。
在计算$C_{i,j}$时，要用到$A_{i,k}, B_{k,j}, k = 0, 1, \ldots, m-1$，因此A的块要在处理器阵列的每一行作多对多广播，B的块要在处理器阵列的每一列作多对多广播。
当$p_{i,j}$接收完$A_{i,k}, B_{k,j}, k = 0, 1, \ldots, m-1$后，执行子矩阵乘法和加法运算。

可扩展性分析的例

- 计算时间： $p_{i,j}$计算$C_{i,j}$时，需要进行m次$n/m \times n/m$阶子矩阵乘法，故$p_{i,j}$的计算时间为：

$$m(\frac{n}{m})^3 = \frac{n^3}{m^2} = \frac{n^3}{p}$$

- 通信时间：设t_s为发送消息的启动时间，t_w为传送每个浮点数时的通信时间， 由于算法需要在由m台处理器组成的组中作二次多对多广播，每次包含处理器阵列上所有行或列的每次并发广播， 发送消息的个数由n^2/p个元素的子矩阵组成，所以整个通信时间为：

$$2(mt_s + m\frac{n^2}{p}t_w) = 2(mt_s + \frac{n^2}{m}t_w)$$

可扩展性分析的例

从而整个并行运行时间为：

$$T_p = \frac{n^3}{m^2} + 2(mt_s + \frac{n^2}{m}t_w) = \frac{n^3}{p} + 2(\sqrt{p}t_s + \frac{n^2}{\sqrt{p}}t_w)$$

系统开销时间为：

$$T_0 = pT_p - T_1 = 2p(\sqrt{p}t_s + \frac{n^2}{\sqrt{p}}t_w).$$

可扩展性分析的例

按等效率度量方法:

$$\begin{aligned} w &= \frac{E_p}{(1-E_p)} \cdot T_0 = \frac{E_p}{(1-E_p)} \cdot 2p(\sqrt{p}t_s + \frac{n^2}{\sqrt{p}}t_w) \\ &= K(p\sqrt{p}t_s + n^2\sqrt{p}t_w) \end{aligned}$$

其中$K = \frac{2E_p}{(1-E_p)}$，同时又有

$$w = n^3$$

于是一定存在正常数α，使得当$n = \alpha K t_w \sqrt{p}$时上述两式相等。此时

$$w = (\alpha K t_w)^3 p^{3/2}. \tag{1}$$

可扩展性分析的例

按等速度度量方法:
设w为使用p台处理器时的问题规模，w'为使用p'台处理器时维持平均速度不变时的问题规模。从而

$$\frac{w}{pT_p} = \frac{w'}{p'T_{p'}}$$

根据可扩展性度量

$$\Psi(p,p') = \frac{w/p}{w'/p'} = \frac{T_p}{T_{p'}}$$

可得

$$\Psi(p,p') = \frac{\frac{n^3}{p} + 2(\sqrt{p}t_s + \frac{n^2}{\sqrt{p}}t_w)}{\frac{n'^3}{p'} + 2(\sqrt{p'}t_s + \frac{n'^2}{\sqrt{p'}}t_w)}$$

$\Psi(p,p')$越小，可扩展性越好。

Part II

程序性能优化

Outline

Outline

串行程序性能优化

串行程序性能的优化是并行程序性能优化的基础。一个好的并行程序首先应该拥有良好的单机性能。 影响程序单机性能的主要因素是程序的计算流程和处理器的体系结构。

调用高性能库

充分利用已有的高性能程序库是提高应用程序实际性能最有效的途径之一(比如基本线性代数库blas,fftw)

BLAS：basic linear algebra subroutines, 是一组高质量的基本向量、矩阵运算子程序库。其官方网址为http://www.netlib.org/blas,国内镜像为：http://netlib.amss.ac.cn/blas。 其中包含Level1 BLAS， Level2 BLAS和Level3 BLAS。

fftw：http://www.fftw.org，c语言写的一个快速fourier变换的子程序库。

合理地调用这些高性能库中的子程序，可以成倍、甚至成数量级地提升应用程序的性能，达到事半功倍的效果。

选择适当的编译器优化选项

现代编译器在编译时能够对程序进行优化，从而提高所生成目标代码的性能。 这些优化功能通常可以通过编译选项来控制。（第二次课上讲过）比较通用的优化选项有：－O0，－O1，－O2，－O3等。 通常－O2被认为是安全的，可以保证程序运行的正确性。(参看例子程序AxB.c)

合理定义数组维数

现代计算机提高内存带宽的一个重要手段是采用多体交叉并行存储系统， 即使用多个独立的内存体，对它们统一编址，将数据以字节为单位采用循环交替方式均匀地分布在不同的内存体中。 为了充分利用多体存储，在进行连续数据访问时应该使得地址的增量与内存体数的最大公约数尽量小， 特别要避免地址增量正好是体数的倍数的情况，因为此时所有的访问将集中在一个存储体中。 由于内存体数和cache 组数通常是2 的幂，因此连续数据访问时应该避免地址增量正好是2 的幂的情形。 很多情况下，合理地声明数组维数有助于避免或缓解内存体或cache 组冲突的问题。(参看例子conflicts.c)

注意嵌套循环的顺序

提高cache 使用效率的一个简单原则是尽量改善数据访问的局部性。 数据访问的局部性分为空间局部性和时间局部性。 空间局部性指访问了一个地址后，应该接着访问它的邻居， 而时间局部性则指对同一地址的多次访问应该在尽可能相邻的时间内完成。下面的循环:

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        A[j][i]=1.;
```

内层循环中数据访问是跳跃的，地址增量为N个数， 因此当N较大时数据访问的空间局部性较差。在编写嵌套的多重循环代码时， 一个通用的原则是尽量使得最内层循环的数据访问 连续进行。(参看程序loop.c,loop1.c)

数据分块

当处理大数组时，对数组、循环进行适当分块有助于同时改善访存的时间和空间局部性。 下面是一个典型的例子

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        A[i]=A[i]+B[j];
```

如果对数组B 进行分块，可以将循环改写成下面的形式:

```
for(j=0;j<N;j+=S)
    for(i=0;i<N;i++)
        for(ij=j;ij<min(j+S,N);ij++)
            A[i]=A[i]+B[ij];
```

根据cache 的大小选择适当的S值，使得B(j:j+S) 能够被容纳在cache 中， 可以改善对数组B的访问的时间局部性。参看block.c，block1.c

循环展开

循环展开是另一个非常有效的程序优化技术。它除了能够改善数据访问的时间和空间局部性外， 还由于增加了每步循环中的指令与运算的数目，亦有助于CPU 多个运算部件的充分利用。
下面是一个一维循环的例子:

```
for(i=0;i<N;i++)
    sum = sum + A[i];
```

将它进行4 步循环展开的代码如下:

```
for(i=0;i<N%4;i++)
    sum = sum +A[i];
for(i=N%4+1;i<N;i+=4)
    sum = sum + A[i]+A[i+1]+A[i+2]+A[i+3];
```

上面的代码中第一个循环用于处理N 除以4 的余数，第二个循环是展开后的循环。

循环展开

再给出一个二重循环的例子，它计算一个矩阵与一个向量的乘积

```
for(i=0;i<N;i++){
     t=0.0;
     for(j=0;j<M;j++)
          t = t+A[i][j]*X[j];
     Y[i]=t;
}
```

循环展开

对i 循环展开2步、j 循环展开3 步，再对内层循环进行合并后的结果如下， 这里为了简化循环展开后的代码，假设N 是2的倍数、M是3 的倍数:

```
for(i=0;i<N;i+=2){
    t0=0.0;
    t1=0.0;
    for(j=0;j<M;j+=3){
        t0=t0+A[i][j]*X[j]+A[i][j+1]*X[j+1]+A[i][j+2]*X[j+2];
        t1=t0+A[i+1][j]*X[j]+A[i+1][j+1]*X[j+1]+A[i+1][j+2]*X
         }
    Y[i]=t0;
    Y[i+1]=t1;
}
```

参看例子程序 Ax.c Ax1.c Ax2.c。

其他程序优化方法

前面主要介绍的优化技术主要是针对访存的优化。除此之外， 还有许多其他的优化方法，如针对CPU 的指令调度、分支预测等等的优化。另外，有许多通用的或由CPU 厂商开发的、针对特定CPU的性能分析工具，如Intel VTune等。

Outline

并行程序性能优化

并行程序的性能优化相对于串行程序而言更加复杂，其中最主要的是选择好的并行算法及通信模式。 在并行算法确定之后，影响并行程序效率的主要因素是通信开销、 由于数据相关性或负载不平衡引起的进程空闲等待、以及并行算法引入的冗余计算。

减少通信量、提高通信粒度

在消息传递并行程序中，花费在通信上的时间是纯开销，因此，如何减少通信时间是并行程序设计中首先要考虑的问题。 减少通信时间的途径主要有三个：减少通信量、提高通信粒度和提高通信中的并发度(即不同结点对间同时进行通信， 要注意的是，这些手段都是相对于特定条件而言的，例如，在网络重负载的情况下，提高通信并行度并不能改善程序的性能)

减少通信量、提高通信粒度

例如，在求解PDE 的区域分解算法中，为了减少通信量，应该尽量将通信局限在相邻的子区域之间， 避免整个数据场的全局通信。在划分子区域时，应该极小化各子区域内边界点的数目。 对于规则区域而言，通常采用高维块划分比一维条划分子区域内边界点数更少。
提高通信粒度的有效方法是减少通信次数，即尽可能将可以一起传递的数据合并起来一次传递。 在收发不同类型的数据时，定义适当的MPI数据类型来避免内存中的数据拷贝。

全局通信尽量利用高效聚合通信算法

当组织多个进程之间的聚合通信时，使用高效的通信算法可以 大大提高通信效率、降低通信开销。对于标准的聚合通信，如广播、 归约、数据散发与收集等，尽量调用MPI 标准库中的函数，因为这 些函数往往经过专门优化。但使用标准库函数的一个缺点是整个通 信过程被封装起来，无法在通信的同时进行计算工作，此时，可以自 行编制相应通信代码，将其与计算过程结合起来，以达到最佳的效 果。

挖掘算法的并行度，减少CPU 空闲等待

一些具有数据相关性的计算过程会导致并行运行的部分进程空 闲等待。在这种情况下，可以考虑改变算法来消除数据相关性。某些 情况下数据相关性的消除是以增加计算量做为代价的，这方面的典 型例子有用Jacobi 迭代替换Gauss–Seidel 或超松弛迭代、三对角线 性方程组的并行解法等。当算法在某个空间方向具有相关性时，应 该考虑充分挖掘其他空间方向以及时间上的并行度，在这类问题中 流水线方法往往发挥着重要的作用。

负载平衡

负载不平衡是导致进程空闲等待的另外一个重要因素。在设计 并行算法时应该充分考虑负载平衡问题，必要时使用动态负载平衡 技术，即根据各进程计算完成的情况动态地分配或调整各进程的计 算任务。动态调整负载时要考虑负载调整的开销及由于负载不平衡 而引起的空闲等待对性能的影响，寻找最优负载调整方案。

通信、计算的重叠

通过让通信与计算重叠进行，利用计算时间来屏蔽通信时间，是 减少通信开销的非常有效的方法。实现通信与计算重叠的方法一般 基于非阻塞通信，先发出非阻塞的消息接收或发送命令，然后处理 与收发数据无关的计算任务，完成这些计算后再等待消息收发的完 成。通信与计算的重叠能否实现，除了取决于算法和程序的实现方 式之外，还取决于并行机的通信网络及通信协议。

通过引入重复计算来减少通信

有时通过适当引入一些重复计算，可以减少通信量或通信次数。 由于当前大部分并行机的计算速度远远快于通信速度，并且一些情 况下，当一个进程计算时，其他进程往往处于空闲等待状态，因而适 当引入重复计算可以提高程序的总体性能。 例如一些公共量的计算，可以由一个进程完成然后再发送给其 他进程，也可以各进程分别独立计算。后一个做法在性能上通常要 好于前一个做法。

作业

1，修改程序AxB.c，将计算结果填入下表中：

循环顺序	运行时间（cpu时间）/s	性能/Mflops
i,j,k		
i,k,j		
j,k,i		
j,i,k		
k,i,j		
k,j,i		

2，修改block1.c程序，测试不同分块（NB取值）下的cpu运行时间，画出曲线。