第 4 章　程序性能评价与优化

给定并行算法，采用并行程序设计平台，通过并行实现获得实际可运行的并行程序后，一个重要的工作就是，在并行机上运行该程序，评价该程序的实际性能，揭示性能瓶颈，指导程序的性能优化。性能评价和优化是设计高效率并行程序必不可少的重要工作。本章主要介绍当前流行的并行程序性能评价方法，并讨论有效的性能优化方法。

4.1　并行程序执行时间

评价并行程序的性能之前，必须清楚并行程序的执行时间是由哪些部分组成的。众所周知，独享处理器资源时，串行程序的执行时间近似等于程序指令执行花费的 CPU 时间。但是，并行程序相对复杂，其执行时间（execution time）等于从并行程序开始执行，到所有进程执行完毕，墙上时钟走过的时间，也称之为墙上时间（wall time）。对各个进程，墙上时间可进一步分解为计算 CPU 时间、通信 CPU 时间、同步开销时间、同步导致的进程空闲时间。

计算 CPU 时间 进程指令执行所花费的 CPU 时间，它可以分解为两个部分，一个是程序本身指令执行占用的 CPU 时间，即通常所说的用户时间（user time），主要包含指令在 CPU 内部的执行时间和内存访问时间，另一个是为了维护程序的执行，操作系统花费的 CPU 时间，即通常所说的系统时间（system time），主要包含内存调度和管理开销、I/O 时间、以及维护程序执行所必需要的操作系统开销等。通常地，系统时间可以忽略。

通信 CPU 时间 包含进程通信花费的 CPU 时间。

同步开销时间 包含进程同步花费的时间。

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。

显然，进程的计算 CPU 时间小于并行程序的墙上时间，而并行程序的墙上时间才是用户真正关心的时间，是评价一个并行程序执行速度的时间。

以上讨论均假设并行程序在执行过程中，各个进程独享处理器资源。如果进程与其他并行程序的进程共享处理器资源，则该进程和其他进程只能分时共享处理器资源，因此，这样会人为地延长并行程序的墙上时间。在本章中，总是假设并行程序的各个进程是独享处理器资源的。

4.2 并行加速比与效率

在处理器资源独享的前提下，假设某个串行应用程序在某台并行机单处理器上的执行时间为 T_S，而该程序并行化后，P 个进程在 P 个处理器并行执行所需要的时间为 T_P，则该并行程序在该并行机上的加速比 S_P 可定义为：

$$S_P = \frac{T_S}{T_P} \tag{4.1}$$

效率定义为：

$$E_P = \frac{S_P}{P} \tag{4.2}$$

这里，需要说明的是，T_1 指处理器个数为 1 时，并行程序的执行时间。通常情形下，T_1 大于 T_S，因为并行程序往往引入一些冗余的控制和管理开销。

加速比和效率是衡量一个并行程序性能的最基本的评价方法。显然，执行最慢的进程将决定并行程序的性能。

在以上加速比和效率的定义中，有一个基本的假设，要求并行机的各个处理器是同构 (homogeneous) 的，即并行机各个处理器的结构完全一致（包含 CPU 类型、内存大小与性能、cache 特征等等），或者说，串行程序在各个处理器执行的墙上时间相等。

如果并行机的各个处理器功能不一致，称之为异构并行机。对此，以上加速比和效率的定义不是很合适。其中，两个突出的问题就是，串行程序的执行时间是选择最快的处理器运行，还是选择最慢的处理器运行？在效率定义中，处理器个数选择为 P 是否合适？一个比较好的方法就是，将所有处理器以最快的处理器为基准，进行归一化处理。

本章中，总是假设并行机是同构的。

4.3 并行程序性能评价方法

上节介绍的加速比和效率，只能反映并行程序的整体执行性能，但是，无法反映并行程序的性能瓶颈。性能评价的主要目的在于，揭示并行程序的性能瓶颈，指导并行程序的性能优化。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。这里，引入文献 [4] 介绍的性能评价方法。

4.3.1 浮点峰值性能与实际浮点性能

在现代微处理器中，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。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。但是，由第 1 章中多级存储结构的讨论可知，任何一次 cache 访问失效均可能中断流

水线的执行，因此，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。

并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程序执行时间的比值。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。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。进一步定义，并行程序发挥并行机浮点峰值性能的比率为并行程序的实际浮点性能和并行机的峰值性能的比值。

实际浮点性能是衡量一个并行程序的绝对指标，而加速比和效率是相对于串行程序，衡量一个并行程序并行性的相对指标。显然，如果串行程序发挥的浮点峰值性能比率越高，它的执行时间就越短，获得的加速比就可能越低。

4.3.2 数值效率和并行效率

将并行程序的墙上时间分解为：

$$T_P = C_i + D_i \qquad i = 1, 2, \ldots, P \tag{4.3}$$

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：

$$C_i = L_i + O_i \qquad i = 1, 2, \ldots, P \tag{4.4}$$

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。

基于以上时间分解，引入如下几个概念。

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：

$$\gamma_i = \frac{L_i}{T_P} \qquad i = 1, 2, \ldots, P \tag{4.5}$$

非数值冗余 由于并行引入的额外非数值计算开销

$$W_i = D_i + O_i \qquad i = 1, 2, \ldots, P \tag{4.6}$$

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：

$$\eta_P = \frac{\sum_{i=0}^{P-1} C_i}{\max_{i=0}^{P-1} C_i \times P} \tag{4.7}$$

负载平衡对并行程序的性能影响很大。例如，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。

显然，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。

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

$$E_P = \frac{S_P}{P} = \frac{T_S}{C_T} \times \frac{C_T}{C_T + D_T}$$

分别定义数值效率和并行效率如下：

$$\begin{cases} \mathrm{NE}_P = \dfrac{T_S}{C_T} & \text{数值效率} \\ \mathrm{PE}_P = \dfrac{C_T}{C_T + D_T} & \text{并行效率} \end{cases}$$

则

$$E_P = \mathrm{NE}_P \mathrm{PE}_P$$

数值效率反映了并行计算引入的额外 CPU 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，各个进程的 cache 命中率将提高，有助于缩短总的计算 CPU 时间，从而提高数值效率；另一方面，并行计算可能引入额外的开销，例如并行算法增加计算量、并行通信 CPU 时间和同步开销等，它们将延长计算 CPU 时间，从而降低数值效率。如果缩短的 CPU 时间小于额外的开销，则数值效率将大于 1，否则，数值效率将小于 1。

并行效率反映了并行程序具体执行的并行性能，它依赖于并行机网络的通信性能，以及并行程序的负载平衡等方面，并行效率总是小于 1 的。

如果数值效率低，说明并行算法或者程序设计引入的额外 CPU 时间太大，有必要修改数值算法或者并行程序设计策略；如果并行效率低，说明并行计算网络通信同步导致的进程间空闲时间太长，有必要在负载平衡、通信结构等多个方面组织优化。

由于效率等于数值效率和并行效率的乘积，因此，如果数值效率大于 1，则可能效率将大于 1，也就是，并行程序的加速比将大于处理器的个数，此时，称之为超线性加速比。由以上的分析可知，需要辩证地看待超线性加速比。如果串行程序能够很好地发挥单处理器的峰值性能，则并行程序几乎不可能获得超线性加速比。反之，如果出现超线性加速比，说明串行程序需要进一步的性能优化。

4.4 可扩展分析

给定并行算法（程序）和并行机，如何调整参与并行计算的处理器个数 P 和求解问题的计算规模 W，使得随着处理器个数的增长，

并行计算的效率可以保持不变，称之为并行程序和并行机相结合的可扩展分析。

可扩展分析是并行计算一个重要研究课题，被广泛应用于描述并行算法（程序）能否有效利用可扩展的处理器个数的能力。通常地，它具有四个目的：

选择合理的算法与结构组合 确定求解某类问题的何种并行算法与何种并行机的组合，它可以有效地利用所期望的处理器规模。

性能预测 对于运行在某台并行机上的某种算法（程序），根据算法（程序）在小处理器规模上的运行性能，预测该算法（程序）移植到大处理器规模上后运行的性能。

最优性能选择 对某类算法，假设问题规模固定，确定在某类并行机上最优的处理器个数和可获得的最优的加速比。

指导性能优化 指导改进并行算法（程序），使得并行算法充分利用可扩展的处理器规模。

下面介绍两种常见的可扩展分析方法。

等效率度量（Kumar 1987[6]） 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：

$$W = f_E(P) \qquad E \text{ 固定} \tag{4.8}$$

等效率值越小，则当处理器个数增多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。

等速度度量（Sun 1994[7]） 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：

$$\Psi(P,P')=\frac{W/P}{W'/P'}=\frac{W}{W'}\times\frac{P'}{P}=\frac{T}{T'} \tag{4.9}$$

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

4.5 程序性能优化

一个程序的实际执行性能取决于程序的实现方式及所使用的高性能计算机的体系结构。本节介绍编写高性能计算程序时的一些注意事项以及一些常用的程序性能优化技巧。在编写程序时注意这些方面的问题，有助于得到合理的性能。

4.5.1 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。在基于微处理器的高性能计算机上，提高程序单机性能的关键是改善程序的访存性能、提高 cache 命中率、以及充分挖掘 CPU 多运算部件、流水线的处理能力。

1. 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。许多著名的高性能数学程序库如优化的 BLAS、FFTW

(参看附录 A) 等，由于经过厂商或第三方针对特定处理机进行的专门优化，其性能一般大大优于用户自行编写的同样功能的程序段或子程序。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。

2. 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的目标代码的性能。这些优化功能通常通过一组编译选项来控制。比较通用的优化选项有 "-O"、"-O0"、"-O2"、"-O3" 等，"-O0" 表示不做优化，"-O1"、"-O2"、"-O3" 等表示不同级别的优化，优化级别越高，生成的代码的性能可能会越好，但采用过高级别的优化会大大降低编译速度，并且可能导致错误的运行结果。通常，"-O2" 的优化被认为是安全的，它可以保证程序运行的正确性。对于一般程序的编译而言，使用优化选项 "-O2" 或 "-O3" 就可以了，进一步的优化可以参考所使用的编译器的手册，通过实验比较来找出一组理想的优化选项组合 (参看习题 4)。

3. 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。对于组关联的 cache 结构也有类似的问题，应该使被访问的数据均匀地分布在尽可能多的 cache 组中才能获得好的执行性能。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。以 Fortran 为例，假设内存访问的字长为 8 字节，内存体数为 S，当对二维数组 `REAL*8 A(M,N)` 第二维上的数据进行顺序访问时：

```
DO J = 1, N
   ... A(I,J) ...
ENDDO
```

数据的增量等于第一维的维数 `M`。当 `M` 是 S 的倍数时，所访问的数据集中在一个存储体中，此时的访存性能是最差的。而当 `M` 与 S 互素时，所访问的数据均匀分布在所有的存储体中，此时访存性能是最好的。对于前一种情况，即 `M` 恰好是 S 的倍数时，一个有效的优化方法是将数组声明成 `REAL*8 A(M+1,N)`，即给数组增加一条“边”，这样虽然浪费了少量存储空间，但是经常可以大幅度地提高程序的执行效率 (参看习题 5)。

4. 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。在嵌套的多重循环中，循环顺序往往对循环中数据访问的局部性有很大的影响，例如下面的循环：

```
DO I = 1, N
DO J = 1, M
   A(I,J) = D
ENDDO
ENDDO
```

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。如果交换上述内外层循环的顺序，将

“DO I” 做为内层循环，“DO J” 做为外层循环，则数据的访问是连续的，空间局部性好，因而程序的性能会大幅度提高。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，这一点不难作到，而且往往可以大幅度提高程序的性能，是编写高性能计算程序时首先要注意的问题。(参看习题 6)。

5. 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 (引自 [1])：

```
DO I = 1, N
DO J = 1, N
   A(I) = A(I) + B(J)
ENDDO
ENDDO
```

如果对数组 B 进行分块，可以将循环改写成下面的形式：

代码 4.1: 利用分块技术改进数据访问的时间局部性。

```
DO J = 1, N, S
DO I = 1, N
   DO JJ = J, MIN(J+S-1, N)
      A(I) = A(I) + B(JJ)
   ENDDO
ENDDO
ENDDO
```

代码 4.1 中 S 为分块大小。当 S ⩾ N 时，相当于原始循环；当 S = 1 时相当于交换 I 和 J 的循环顺序。根据 cache 的大小选择适当的 S 值，使得 B(J:J+S-1) 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。

数据分块是一项比较复杂的优化技术，好的分块方式与分块参数的确定需要对代码及 cache 结构进行非常细致的分析或通过大量

的实验才能得到，因此该项技术一般只在对一些关键代码段进行深层次优化时才使用 (参看习题 7 和习题 8)。

6. 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：

```
DO I = 1, N
   D = D + A(I)
ENDDO
```

将它进行 4 步循环展开的代码如下：

```
DO I = 1, MOD(N,4)
   D = D + A(I)
ENDDO
DO I = MOD(N,4)+1, N, 4
   D = D + A(I) + A(I+1) + A(I+2) + A(I+3)
ENDDO
```

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。

再给出一个二重循环的例子，它计算一个矩阵的转置与一个向量的乘积：

```
DO J = 1, M
   T = 0.0
   DO I = 1, N
      T = T + A(I,J) * X(I)
   ENDDO
   Y(J)=T
ENDDO
```

对 I 循环展开 3 步、J 循环展开 2 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 3 的倍数、M 是 2 的倍数：

```
DO J = 1, M, 2
   T0 = 0.0
   T1 = 0.0
   DO I = 1, N, 3
      T0 = T0 + A(I,J)*X(I)+A(I+1,J)*X(I+1)+A(I+2,J)*X(I+2)
      T1 = T1 + A(I,J+1)*X(I)+A(I+1,J+1)*X(I+1)+A(I+2,J+1)*X(I+2)
   ENDDO
   Y(J) = T0
   Y(J+1) = T1
ENDDO
```

手工编写多重循环展开代码往往非常麻烦，并且只能对固定的循环展开步数进行，不便于寻找最优的循环展开步数。现代编译系统亦提供编译选项或编译指导语句，实现自动循环展开的功能，例如 GNU 编译器 (gcc，g77 等) 提供了选项 "`-funroll-loops`"，用于指定对代码中的循环进行展开，但它们一般局限于使用固定的循环展开步数，通常达不到最好的性能。对于一些复杂的情况，可以借助于专门设计的工具来对代码进行自动或半自动的循环展开处理，例如，文献 [65] 中研究了借助 m4 宏语言进行 Fortran 程序的循环展开，其中定义了一套 m4 宏命令，可将一些常见的循环写成可以任意指定循环展开步数的形式，参看习题 9。

7. 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 [66] 等。

4.5.2 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。在设计并行程序时，可以采用多种技术来减少或消除这些因素对并行效率的影响。本节对常用的一些并行程序优化技术进行简单介绍与讨论，主要给出一些原则性的考虑。

1. 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)。

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。

2. 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。

3. 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，例如，参看第 9.5 节。

4. 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。

5. 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。第 339 页代码 8.3 提供了一个如何实现通信与计算重叠的实例。

6. 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。另外一个通过引入重复计算来提高通信粒度的例子参看第 8 章习题 7。

习 题

1. 假设你使用的是一台异构并行机，请问如何评价一个并行程序的加速比和效率比较好？它相对于传统的同构并行机上的评价准则有何优点？

2. 在一台并行机上，为了测试一个并行程序的加速比和效率，必须做哪些准备工作？为什么？

3. 影响并行程序执行时间的一些主要因素是什么？如何合理地评价一个并行程序的性能？

4. 下面列出的是一个计算矩阵乘积的 Fortran 程序。统计采用不同优化选项编译该程序生成的代码的运行时间，根据程序的计算量和运行时间计算出程序的实际浮点性能（以 Mflops 为单位）和效率 (实际性能/处理器峰值性能)，并将结果填写在下表中 (根据需要加行)。

提示：用 “`time` 程序名” 可以得到程序的运行时间。

优化选项	运行时间 (秒)	性能 (Mflops)	效率 (%)

文件名: code/performance/AxB.f

```
      PARAMETER(N=1024)
      REAL*8 A(N,N), B(N,N), C(N,N)
*
      DO J = 1, N
         DO I = 1, N
            A(I,J) = 1.D0
            B(I,J) = 1.D0
            C(I,J) = 0.D0
         ENDDO
      ENDDO
*
      DO J = 1, N
      DO K = 1, N
      DO I = 1, N
         C(I,J) = C(I,J) + A(I,K) * B(K,J)
      ENDDO
      ENDDO
      ENDDO
*
      STOP
      END
```

5. 测试下面程序的运行时间，然后将其中的 "REAL*8 A(N,N)" 改为 "REAL*8 A(N+1,N)"，比较修改前后运行时间的差异。

文件名: code/performance/conflicts.f

```
      PARAMETER(N=2048)
```

```
2       REAL*8 A(N,N)
3       DO K = 1, 100
4          DO I = 1, N
5          DO J = 1, N
6             A(I,J) = 1.D0
7          ENDDO
8          ENDDO
9       ENDDO
10      STOP
11      END
```

6. 改变习题 4 的程序中三重循环 (11–19 行) 的顺序，统计不同循环顺序的运行时间、性能及效率，将结果填写在下表中。

循环顺序	运行时间 (秒)	性能 (Mflops)	效率 (%)
I,J,K			
I,K,J			
J,K,I			
J,I,K			
K,I,J			
K,J,I			

7. 编写代码 4.1 中关于循环分块的完整程序，选取一个适当的 N 值，测试不同分块大小下的程序运行时间，并画出曲线。

8. 网址 ftp://ftp.cc.ac.cn/pub/home/zlb/bxjsbook/code/performance/ 下的 A+Bt.f 是计算一个矩阵和另一个矩阵的转置相加的程序，block.f 是它的分块版本。试分析分块前后 cache 使用的差异，并测试不同分块大小 (NB) 对程序性能的影响。

9. 网址 `ftp://ftp.cc.ac.cn/pub/home/zlb/bxjsbook/code/performance/` 中有一个利用 m4 宏语言进行循环展开的例子。从该网址下载文件 `AxB.m4`、`defs.m4` 和 `m4post.c`，编译 `m4post.c` 得到可执行程序 `m4post`，然后便可用下述命令：

```
m4 -P defs.m4 AxB.m4 | ./m4post >tmp1.f
m4 -P -Dni=2 -Dnj=2 -Dnk=2 defs.m4 AxB.m4 | ./m4post >tmp2.f
... ...
```

得到对习题 4 中的矩阵乘法程序进行循环展开的代码，其中 `ni`、`nj` 和 `nk` 分别给出循环 `I`、`J` 和 `K` 的循环展开步数 (注意运行上述命令时必须使用 GNU m4，其他版本的 m4 程序不支持 "`-P`" 选项)。试比较习题 4 中的代码与循环展开后的代码，为了便于阅读，可以利用 Emacs 编辑器的 "`C-A-q`" 命令 (`Ctrl-Alt-q`) 或菜单项 "`Fortran->Ident subprogram`" 对代码进行缩进 (identation) 处理。测试不同循环展开步数的性能，没法找出达到最佳性能的循环展开步数，并将测试结果填写在下表中 (可以在表中加行)。

循环展开步数	运行时间 (秒)	性能 (Mflops)	效率 (%)
1 1 1			