

高性能计算与云计算

实 验 报 告

(2024-2025学年第一学期)

提交日期： 2024年12月 15 日

|  |  |  |  |
| --- | --- | --- | --- |
| 实验题目 | 实验1——矩阵-矩阵乘法 | | |
| 学号 | 202230441138 | 姓名 | 黄鸿展 |
| 学院 | 计算机科学与工程学院 | 任课教师 | 何克晶 |
| 课程名称 | 高性能计算与云计算 | 课程编号 | 045101911 |

1. **实验目的**

1.并行化一个简单算法

2.研究不同并行化策略的性能

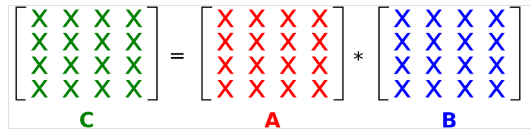
3.了解并行算法中 “粒度 ”的重要性

4.实现并测量并行算法的性能

**二、实验过程**

**1. 问题陈述**

将矩阵-矩阵乘法作为第一个要并行化的算法。我们要解决的问题定义如下。给定*A*和*B*两个*N* ×*N* 矩阵，要求矩阵-矩阵乘积 *C* =*AB*。计算该乘积并实现并行与高效。

****

1. 使用多个进程并行计算乘积（分布式实现）
2. 研究不同并行化方案的性能
3. 使用 Julia 基于任务的编程模型实现算法

首先考虑基于矩阵-矩阵乘积 Cij=∑k(Aik×Bkj)定义的序列算法。

|  |
| --- |
| @everywhere function matmul\_seq!(C,A,B)     m = size(C,1)     n = size(C,2)     l = size(A,2)     @assert size(A,1) == m     @assert size(B,2) == n     @assert size(B,1) == l     z = zero(eltype(C))     for j in 1:n         for i in 1:m             Cij = z             for k in 1:l                 @inbounds Cij += A[i,k]\*B[k,j]             end             C[i,j] = Cij         end     end     C end |

序列算法是朴素的串行算法，假设所有矩阵都是 N-by-N的矩阵，**复杂度为O(n³)。**

矩阵-矩阵乘法就是尴尬并行算法的一个例子。尴尬并行算法（也称微不足道的并行算法）是一种可以分割成并行任务的算法，它们之间没有（或只有很少）依赖关系。这类算法通常很容易并行化。如何看出算法是否可以被并行化？

我们可以从代码中分析：可以检查一下 for 循环，它们是潜在的并行性来源。如果迭代是相互独立的，那么它们就很容易并行化。检查迭代是否相互依赖的简单方法是改变它们的顺序（例如，将 for j in 1:n 改为for j in n:-1:1，即反向执行循环）。如果结果发生了变化，那么迭代就不是独立的。注意，浮点数运算由于有四舍五入舍去值，有部分迭代不完全独立。通过这种方式，我们就能从算法中尝试设计并行化优化。

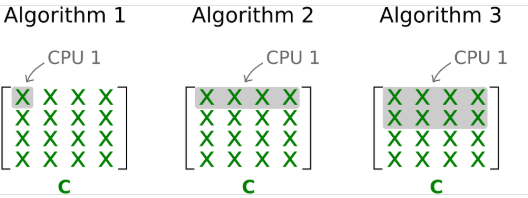
**2. 并行算法**

i和j上的循环是可并行的，这意味着矩阵 C 的所有条目都有可能并行计算。然而，在分布式系统中并行求解所有这些条目的最有效解决方案是什么？为此，将考虑不同的并行化策略：

算法 1：每个工作者计算 C 的一个条目（A一行与B一列）

算法 2：每个工作者计算 C 的一个单行（A一行与B整体）

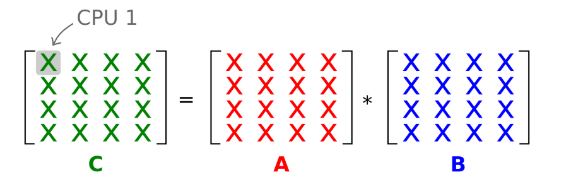
算法 3：每个工作者计算 N/P 行 C。（A的N/P行与B整体）

****

1. **算法1**

通过网络移动数据的成本很高，减少数据移动是设计高效分布式算法的关键点之一。为此，我们需要确定工作程序执行计算所需的最小数据量。这些数据称为 **数据依赖性** 。

在算法 1 中，每个工作者只计算结果矩阵 C 的一个条目。在这种情况下，负责计算条目 C[i,j] 的工作者所做计算的数据依赖关系是**关于A[i,:]一整行与B{:,j}一整列。**



考虑到数据依赖关系，并行算法 1 可以按以下步骤高效执行：

1. 工作进程从主进程接收数据依赖关系，即相应的行 A[i，:]和列 B[:,j]
2. 工作进程在本地计算 A[i,:] 和 B[:,j] 的点积
3. 工作进程将 C[i,j] 的结果发回主进程

该算法在 Julia 中的可能实现如下：

|  |
| --- |
| function matmul\_dist\_1!(C, A, B)     m = size(C,1)     n = size(C,2)     l = size(A,2)     @assert size(A,1) == m     @assert size(B,2) == n     @assert size(B,1) == l     z = zero(eltype(C))     @assert nworkers() == m\*n     iw = 0         @sync for j in 1:n         for i in 1:m             Ai = A[i,:]             Bj = B[:,j]             iw += 1             w = workers()[iw]             ftr = @spawnat w begin                 Cij = z                 for k in 1:l                     @inbounds Cij += Ai[k]\*Bj[k]                 end                 Cij             end             @async C[i,j] = fetch(ftr)         end     end     C end |

在这里，**@sync用于确保在继续执行代码之前，所有在它的作用域内启动的异步操作都已完成**：@sync 块确保了在处理下一行之前，当前行的所有元素都已经被计算并存储在 C 中。这对于保持结果矩阵 C 的正确性至关重要，因为矩阵乘法的计算顺序会影响到最终结果。

**@async用于异步地执行一个操作，允许主线程在等待结果的同时继续执行其他任务。**在这个函数中，@async 与 fetch 一起使用，用于从工作进程异步地获取计算结果。这样可以提高效率，因为主进程不需要等待每个单独的计算完成，而是可以继续处理其他任务，直到所有结果都准备好后再进行下一步。

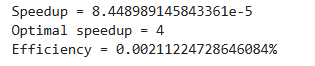
在这种情况下，我们将使用所谓的最佳并行加速。在 PP 进程上的并行加速定义为：

其中，T1 表示顺序算法的运行时间，TP表示并行算法在 P进程上的运行时间。如果我们用 P进程运行一个最优的并行算法，我们希望它的运行速度是顺序算法的 p 倍。也就是说，并行算法在 P进程上的 *最优* 加速等于 P ：

并行效率：

用上述方法检测并行效率，可以看到算法一的优化效果：

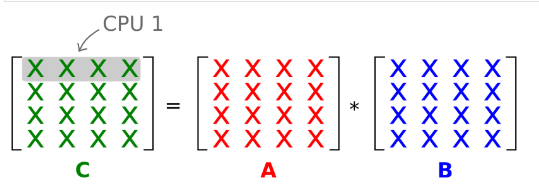
|  |
| --- |
| N = 2 A = rand(N,N) B = rand(N,N) C = similar(A) T1 = @belapsed matmul\_seq!(C,A,B) C = similar(A) TP = @belapsed matmul\_dist\_1!(C,A,B) P = nworkers() println("Speedup = ", T1/TP) println("Optimal speedup = ", P) println("Efficiency = ", 100\*(T1/TP)/P, "%") |



可以看到算法优化并不佳，是因为有通信开销：每名工人通信量为O(N)，计算量也为O(N)，因此**通信与计算之比（通信开销）为 O(1)**，通信成本与计算成本的数量级相同。由于在实际系统中，通信速度要慢上几个数量级，使得优化效果不佳。

1. **算法2**

在并行算法 2 中，每个工作者计算 C 的一整行。



在这里负责计算C[i,:]行的工作者计算的数据依赖关系是A的一行与B的整体。

以下是算法 2的主要执行步骤：

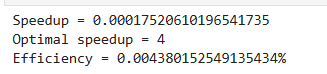
1. 工作进程从主进程接收相应的行 A[i,:] 和矩阵 B
2. 工作进程计算 A[i，:]行与 B 的乘积
3. 工作进程将 C[i,:]行的结果发回主进程

代码实现如下：

|  |
| --- |
| function matmul\_dist\_2!(C, A, B)     m = size(C,1)     n = size(C,2)     l = size(A,2)     @assert size(A,1) == m     @assert size(B,2) == n     @assert size(B,1) == l     z = zero(eltype(C))     @assert nworkers() == m     iw = 0     @sync for i in 1:m         Ai = A[i,:]         iw += 1         w = workers()[iw]         ftr = @spawnat w begin             Ci = fill(z,n)             for j in 1:n                 for k in 1:l                     @inbounds Ci[j] += Ai[k]\*B[k,j]                 end             end             Ci         end         @async C[i,:] = fetch(ftr)     end     C     end |

用以下代码检测算法2优化效果：

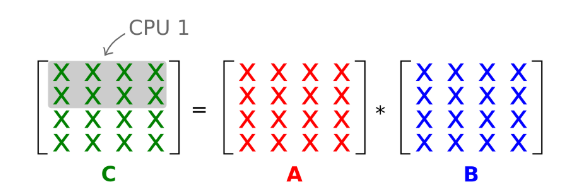
|  |
| --- |
| N = 4 A = rand(N,N) B = rand(N,N) C = similar(A) T1 = @belapsed matmul\_seq!(C,A,B) C = similar(A) TP = @belapsed matmul\_dist\_2!(C,A,B) P = nworkers() println("Speedup = ", T1/TP) println("Optimal speedup = ", P) println("Efficiency = ", 100\*(T1/TP)/P, "%") |



可以发现优化以后的并行加速更快，而且效率是原来并行算法1的2倍，效果大大提高，但还是达不到最佳效果。计算通信开销可知，计算量和通信量均为O(N²)，仍然是一个数量级。为此，需要考虑增加计算和通信比。

**③ 算法3**

在一个工作者中计算 C 的若干行。每个工作者连续计算 N/P 行 C，其中 P 是工作者的数量。



计算行范围 C[rows,:] 的工作者所做计算的数据依赖关系是A的整行与B整体。

以下是算法 3 的主要执行步骤：

1. 工作者从主进程接收相应的行 A[rows,:] 和矩阵B
2. 工作者计算 A[行,:]乘以 B的乘积
3. 工作者将结果 C[行,:]发回主进程

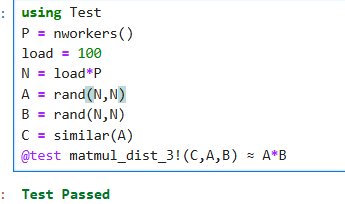
我们先行计算算法3的通信开销：

一个工人每次的通信量依然不超过一个矩阵，为O(N²)，但由于提高了一个进程能处理的行数，使得计算量变为O(N²·N/P)=O(N³/P)，这样计算量的数量级在N>>P时可以忽略通信开销O(P/N)，这样就减小了通信开销给算法带来的影响。

**下面，补充练习题的代码，实现算法3（假设C 的行数是工作进程数的整数倍）：**

|  |
| --- |
| using Base.Threads using Distributed using BenchmarkTools using Printf function matmul\_dist\_3!(C, A, B)     *# 获取矩阵的维度*     m = size(C, 1)  *# 结果矩阵 C 的行数*     n = size(C, 2)  *# 结果矩阵 C 的列数*     l = size(A, 2)  *# 矩阵 A 的列数，同时也是矩阵 B 的行数*     *# 检查矩阵维度是否匹配*     @assert size(A, 1) == m  *# 矩阵 A 的行数必须等于结果矩阵 C 的行数*     @assert size(B, 2) == n  *# 矩阵 B 的列数必须等于结果矩阵 C 的列数*     @assert size(B, 1) == l  *# 矩阵 B 的行数必须等于矩阵 A 的列数*     *# 确保结果矩阵的行数可以被工作线程数整除*     @assert mod(m, nworkers()) == 0  *# 行数必须是工作线程数的整数倍*     *# 计算每个工作线程需要处理的行数*     nrows\_w = div(m, nworkers())  *# 每个工作线程处理的行数*     *# 同步所有工作线程*     @sync for (iw, w) in enumerate(workers())         *# 计算当前工作线程处理的行的范围*         lb = 1 + (iw - 1) \* nrows\_w  *# 当前工作线程处理的行的起始索引*         ub = iw \* nrows\_w            *# 当前工作线程处理的行的结束索引*         *# 提取当前工作线程需要处理的子矩阵 A*         A\_w = A[lb:ub, :]  *# 从矩阵 A 中提取当前工作线程需要处理的子矩阵*         *# 在工作线程上异步执行矩阵乘法*         ftr = @spawnat w begin             *# 创建与子矩阵 A 相同大小的子矩阵 C*             C\_w = similar(A\_w)             *# 顺序执行矩阵乘法*             matmul\_seq!(C\_w, A\_w, B)             *# 返回计算结果*             C\_w         end         *# 异步将计算结果写回结果矩阵 C*         @async C[lb:ub, :] = fetch(ftr)     end     *# 返回最终的结果矩阵 C*     C end |

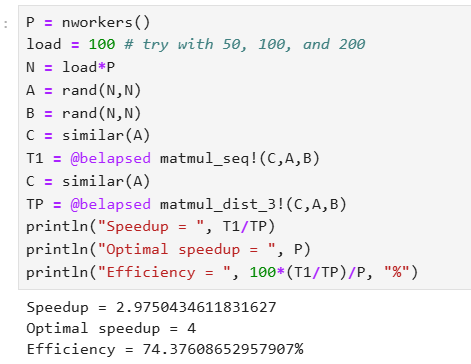
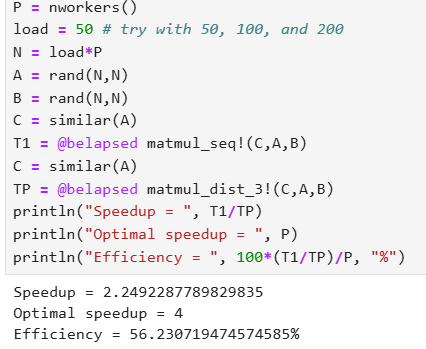
嵌套的三个 for 循环计算矩阵乘法的结果。外层循环遍历矩阵 C 的每一列，中间循环遍历矩阵 A 的列（即矩阵 B 的行），内层循环遍历当前工作进程负责的每一行。每个工作进程的结果存储在 C\_w 中，使用 @async 将每个工作进程的局部结果 C\_w更新到全局矩阵 C 的对应行。更新范围由 lb:ub 指定，即当前工作进程负责的行的起始和结束索引。

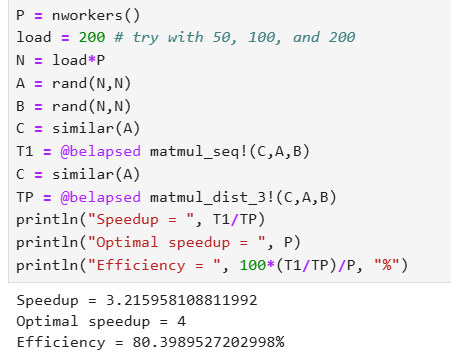
****

**这段代码成功通过测试，下面检测其并行效果：**

|  |
| --- |
| P = nworkers() load = 200 *# try with 50, 100, and 200* N = load\*P A = rand(N,N) B = rand(N,N) C = similar(A) T1 = @belapsed matmul\_seq!(C,A,B) C = similar(A) TP = @belapsed matmul\_dist\_3!(C,A,B) println("Speedup = ", T1/TP) println("Optimal speedup = ", P) println("Efficiency = ", 100\*(T1/TP)/P, "%") |

**以下分别为load=50、100、200时的情况，可以看到并行效率与加速均远高于算法1、2，而且随着load数据量的增大效率和速度不断提升，这样就接近了我们希望的最优并行算法。**

****

****

**三、总结**

下表对三种并行算法进行了比较：

| **Algorithm** | **Parallelism (#workers)** | **Communication per worker** | **Computation per worker** | **Ratio communication/ computation** |
| --- | --- | --- | --- | --- |
| 1 | N² | O(N) | O(N) | O(1) |
| 2 | N | O(N²) | O(N²) | O(1) |
| 3 | P | O(N²) | O(N³/P) | O(P/N) |

矩阵-矩阵乘法具有微不足道的并行性（至少在理论上，结果矩阵中的所有条目都可以并行计算）。矩阵乘法并行化算法旨在将矩阵乘法分解为多个并行执行的任务，从而减少等待时间，提高算法速度。

**算法1**是比较普通的乘法算法，计算量为O(N)，通信与计算比为O(1)，由于现实通信开销的影响，使得速度较慢，并行效果不佳。

**算法2**尝试将矩阵乘法并行化，但由于每个工作进程需要访问整个矩阵，导致通信量与计算量均为O(N2)，通信与计算比保持为O(1)。这种算法在实际应用中可能并不高效，因为通信开销较大。

**算法3**通过将矩阵分配给不同的工作进程来减少通信量。每个工作进程只负责计算结果矩阵的一部分，从而减少了通信量。这种算法的通信量仍然是O(N²)，但由于计算量分散在P*P*个工作进程中，每个工作进程的计算量减少到O(N³P)，通信与计算比降低到O(P/N)，提高了并行效率。

并行算法的性能不仅取决于算法本身的计算复杂度，还受到通信开销的影响。在设计并行算法时，需要平衡计算和通信，以达到最佳的并行效率。算法3通过减少每个工作进程的计算量和通信量，提高了并行计算的效率，尤其是在大规模并行计算环境中。

在网上进行搜索，我还找到更多有效的并行算法，比如Cannon算法与Fox算法，他们运用并行的思想不同，采用了分治的策略转化为子矩阵乘法，从而减小问题解决的规模。而本实验采用的算法是通过增加每次计算的粒度提高计算量，同时减小通信开销，因此可以将几个算法结合起来，在小问题中运用提高粒度的方法，以更好地提高并行速度。