

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3选择2——雅可比迭代并行算法 | | |
| 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.学习并行化雅可比方法

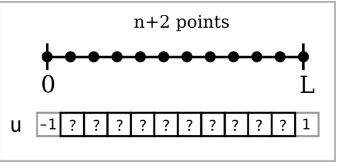
2.了解数据分区如何能够影响分布式算法的性能

3.学习如何利用延迟隐藏来提高并行性能

**二、实验过程**

**1. 问题陈述**

雅可比法（Jacobi method）是一种用于求解线性方程组的迭代数值方法。雅主要应用之一是求解边值问题（BVP） 所产生的方程：给定边界（网格）处的值，找到满足某一方程的内部值。它属于迭代法中的一种，特别适用于大规模稀疏线性系统的求解。



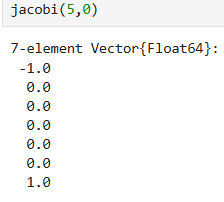
在求解一维拉普拉斯方程时，在迭代 t + 1 时，向量 u 的第 i 项时，雅可比方法需要用一下公式计算：：



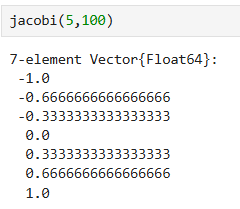
先考虑串行算法：

|  |
| --- |
| function jacobi(n,niters)     u = zeros(n+2)     u[1] = -1     u[end] = 1     u\_new = copy(u)     for t in 1:niters         for i in 2:(n+1)             u\_new[i] = 0.5\*(u[i-1]+u[i+1])         end         u, u\_new = u\_new, u     end     u end |

0次迭代看到初始态：

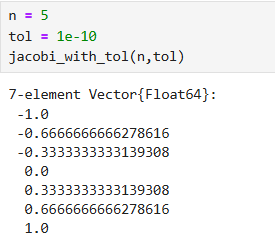


多次迭代看到预期解：

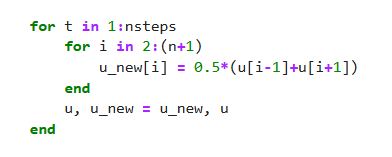


也可以在解的差距小于预期值的时候终止迭代：

|  |
| --- |
| function jacobi\_with\_tol(n,tol)     u = zeros(n+2)     u[1] = -1     u[end] = 1     u\_new = copy(u)     while true         diff = 0.0         for i in 2:(n+1)             ui\_new = 0.5\*(u[i-1]+u[i+1])             u\_new[i] = ui\_new             diff\_i = abs(ui\_new-u[i])             diff = max(diff\_i,diff)                     end         if diff < tol             return u\_new         end         u, u\_new = u\_new, u     end     u end |

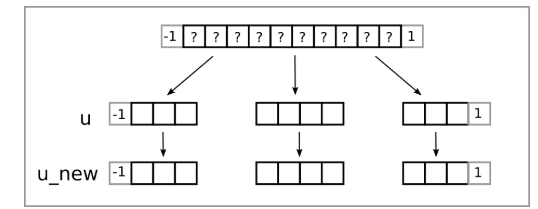


**2. 一维雅可比并行算法**

要实现并行算法，需要找到可并行的变量：

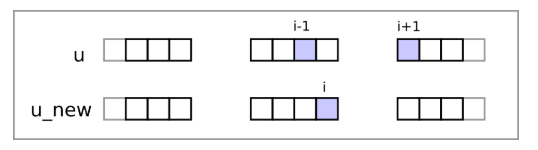
在两个循环中，外循环由于u的值会随每一个循环改变，无法并行化。内循环的循环迭代是相互独立的，因此可以并行。

如果考虑不同的进程中计算u\_new，虽然可以实现并行，但因为粒度不大，性能受通信开销影响，优化效果不佳。因此，可以考虑将数据分段，每段进行雅可比算法：

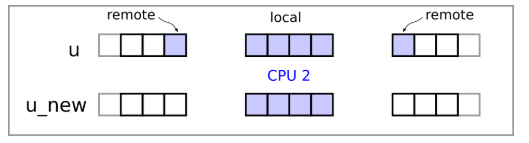


其中，雅可比算法核心式：u\_new[i] = 0.5\*(u[i-1]+u[i+1])

所以要更新本地存储向量**边界**上的条目，就需要存储在其他处理器上的条目。

****

那么，一段数据所需要的条目如图：



**通信开销**

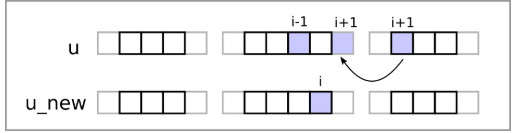
在每次迭代中，每个进程更新 N/P 项，其中 N 是向量的总长度,P 是进程的数量。因此，计算复杂度为 O(N/P)；

需要从两个邻居那里获取远程输入（每次迭代两条消息），需要每条消息传达 1 个条目，因此，通信复杂度为 O(1)；

通信/计算比率为 O(P/N)，如果 P << N，则该算法有可能具有可扩展性。

**Ghost单元**

幽灵单元（Ghost cells）是并行计算中用于处理边界条件的一种技术，特别是在有限差分方法中求解偏微分方程（PDEs）时。幽灵单元的概念是为了在并行环境中保持数据的连续性和边界条件的一致性。

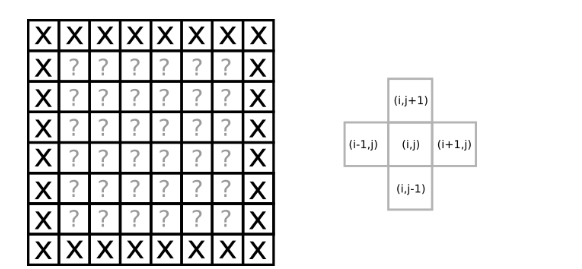


因此，算法核心在以下两个步骤：

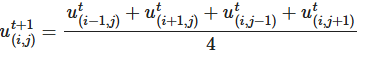
1. 用通信内容填补这些幽灵般的条目
2. 在每个进程中依次进行雅可比更新

**3. 二维雅可比并行算法**

目标是给定边界上的值，找出二维网格的内部点：



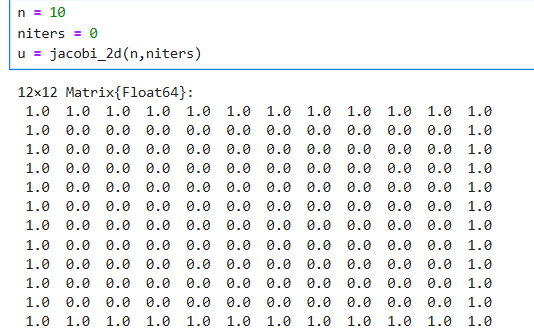
二维拉普拉斯方程，在迭代 t+1 时，（i，j）的计算公式为：



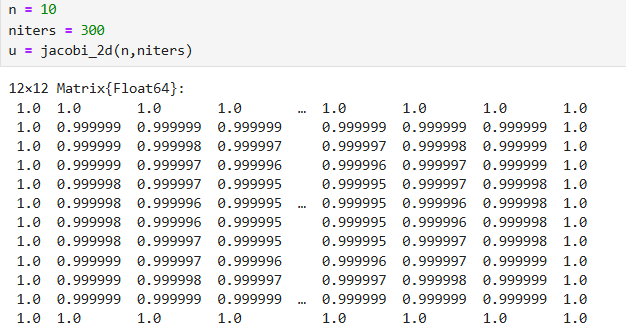
串行方法实现：

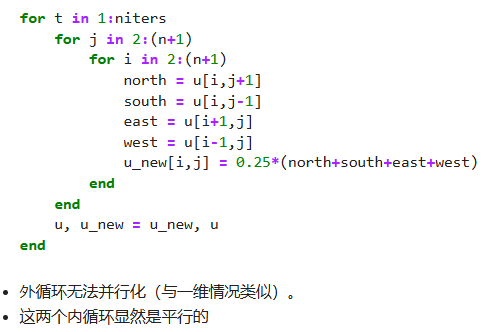
|  |
| --- |
| function jacobi\_2d(n,niters)     u = zeros(n+2,n+2)     u[1,:] = u[end,:] = u[:,1] = u[:,end] .= 1     u\_new = copy(u)     for t in 1:niters         for j in 2:(n+1)             for i in 2:(n+1)                 north = u[i,j+1]                 south = u[i,j-1]                 east = u[i+1,j]                 west = u[i-1,j]                 u\_new[i,j] = 0.25\*(north+south+east+west)             end         end         u, u\_new = u\_new, u     end     u end |

零次迭代：



多次迭代：



接下来考虑并行性，同一维雅可比的外循环无法并行。

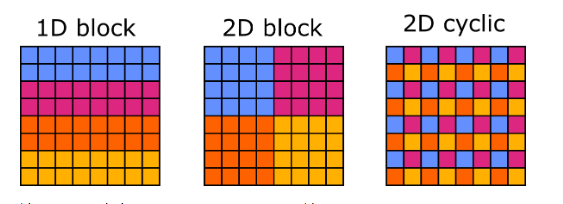
并行化策略：

二维为了更好的利用进程，有三种方案：

①1D 块行分区（工作节点处理多行以及所有列）

②二维块划分（工作节点处理连续多行多列）

③二维循环分区（工作节点处理行列的交替）



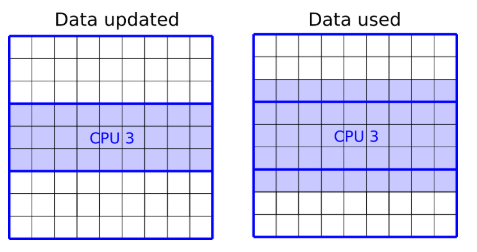
要考虑这三个方案的最优，需要计算通信复杂度与计算比：

**①一维按行分块**

假设进程数为P，矩阵长度为N，那么每次迭代计算量为N²/P；

每次通信从2个另外进程获取信息，每个进程获取N条，通信量为2N；

通信与计算比复杂度为2N÷（N²/P）=O(P/N)

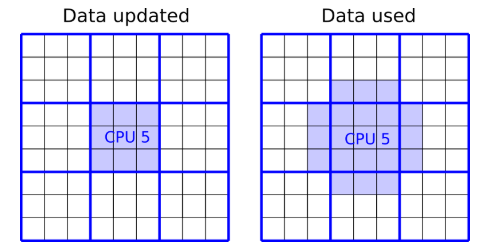


**②二维块划分**

假设进程数为P，矩阵长度为N，那么每次迭代计算量为N²/P；

每次通信从4个进程获取，每个进程获取条（用P占满N²），通信量为；

通信与计算比复杂度为÷（N²/P）=O(/N)

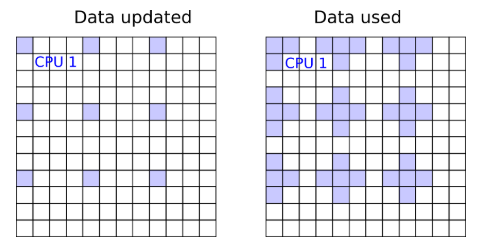


**③二维循环划分**

假设进程数为P，矩阵长度为N，那么每次迭代计算量为N²/P；

每次通信从4个另外进程获取信息，每个进程获取N²/P条，通信量为4N²/P；

通信与计算比复杂度为4N²/P÷（N²/P）=O(1)



①如果 P << N,那么一维和二维的分块分区都有可能扩展；

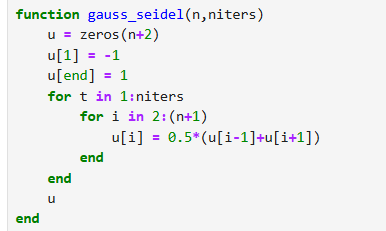
②二维块划分具有最低的通信复杂度；

③一维块划分需要发送更少的消息；

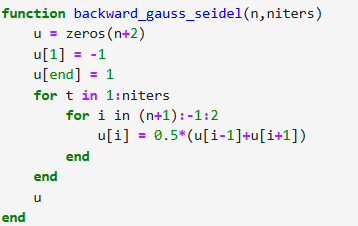
④通信量太多了，循环分区不切实际。

**3.高斯-赛德尔法**

通过采用雅可比的顺序实现，去除 u\_new，只使用 u。这种方法优化了内存。

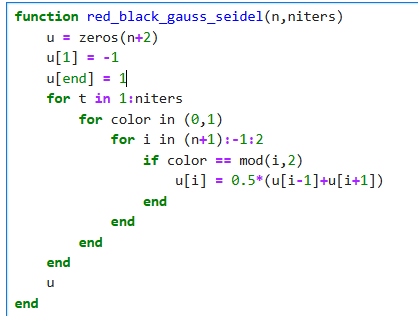
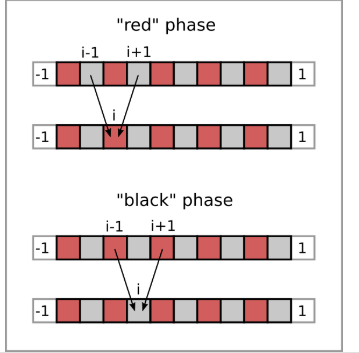


如果内循环从后往前，就得到了逆向的方法：



都会收敛到相同的结果，但在迭代过程中它们会导致略有不同的数值。

红黑高斯-赛德尔法首先，更新索引为偶数的项，然后更新索引为奇数的项。



**4.MPI实现：**

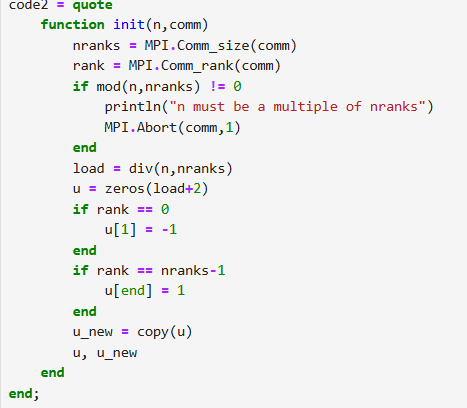
并行实现需要一个MPI通信器实例。例如，可以直接采用MPI.COMM\_WORLD。

函数jacobi\_mpi将在多个MPI进程中被执行。因此，进程应具有相同的n和niters值。

得到的u与顺序执行的jacobi函数的结果不一致。它仅包含每个进程存储的u的局部部分，要重建与顺序执行相同的向量，需要在一个进程中聚合这些部分。

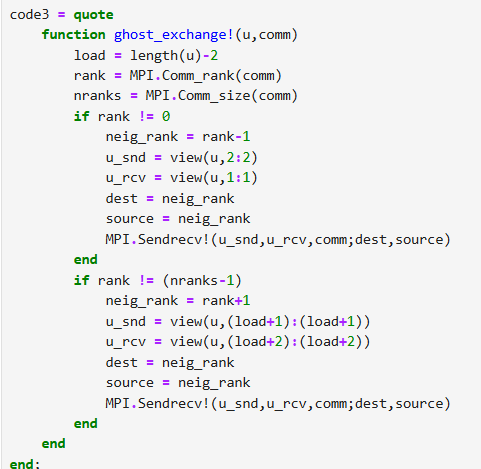
①开始阶段

建立并设置向量u和辅助向量u\_new，并输入边界值。仅创建当前进程负责的部分，为此，首先确定当前进程需要处理的条目数load。在u（和u\_new）中额外预留两个元素以容纳幽灵单元和边界条件。



②通信

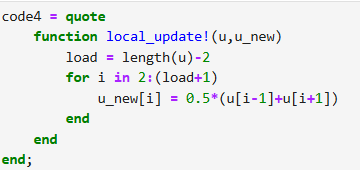
发生在函数 ghost\_exchange! 中。使用MPI点对点通信修改输入向量的幽灵单元。

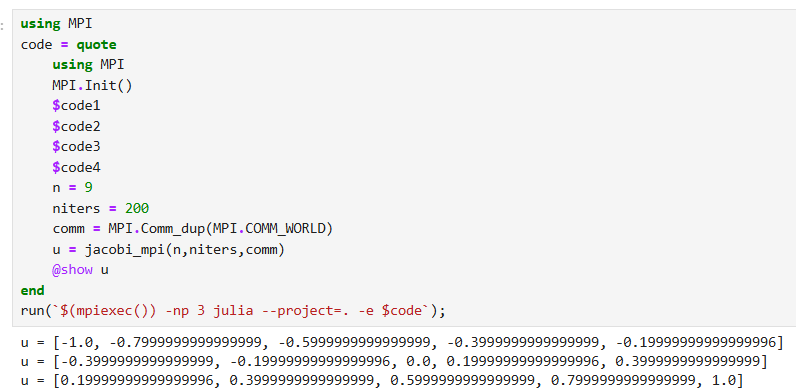


注意这里用的是Sendrecv，如果用send和Recv需要改进顺序，避免死锁。

③本地计算

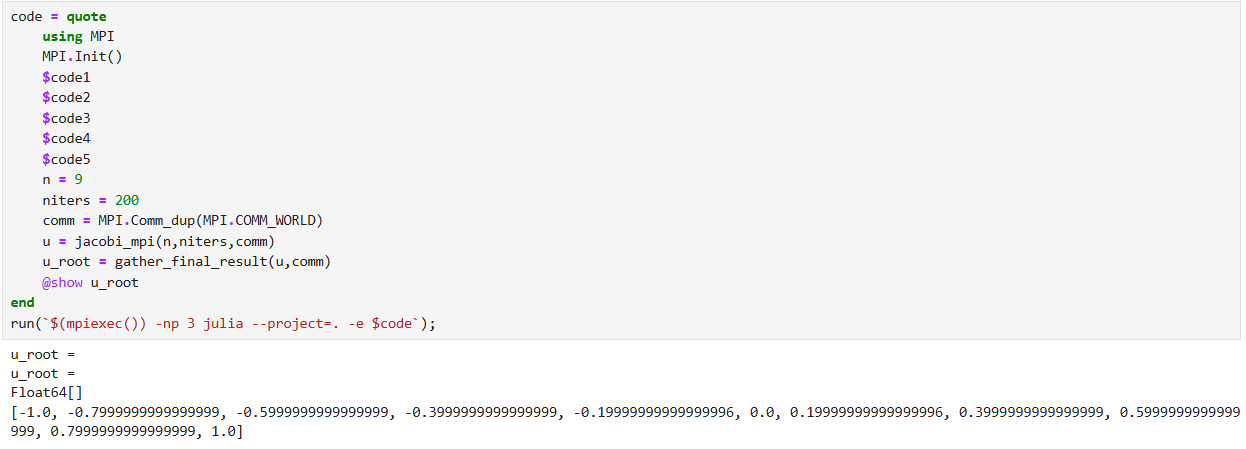
更新当前rank拥有的数据，即不修改幽灵值。





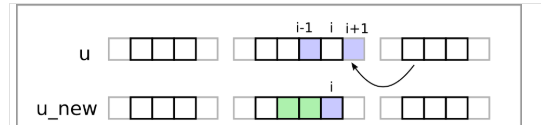
运行代码，可以看到解在奇数进程是正确分布的，但是对于偶数进程会报错。

将并行结果收集到一个秩中，与顺序执行比较，可以看到结果无误：



④延迟隐藏

更新每个进程的边界值。通过计算重叠来隐藏通信延迟。



要实现延迟隐藏，需要使用Isend和Irecv。

**下面是练习题代码的实现：**

**Exercise1：**

|  |
| --- |
| using MPI MPI.Init() function jacobi\_mpi\_latency\_hiding(n, niters，comm)     u, u\_new = init(n, comm)  *# 初始化网格 u 和 u\_new*     local\_load = length(u) - 2  *# 计算每个进程的局部负载（不包括边界）*     rank = MPI.Comm\_rank(comm)  *# 获取当前进程的 rank*     num\_ranks = MPI.Comm\_size(comm)  *# 获取进程总数*     nreqs = 2 \* ((rank != 0) + (rank != (num\_ranks - 1)))     reqs = MPI.MultiRequest(nreqs)  *# 初始化通信请求数组*      for t in 1:niters  *# 迭代 niters 次*         ireq = 0  *# 初始化通信请求索引*         if rank != 0             neig\_rank = rank - 1  *# 左侧邻居的 rank*             u\_snd = view(u, 2:2)  *# 发送的数据（边界值）*             u\_rcv = view(u, 1:1)  *# 接收的数据（边界值）*             dest = neig\_rank  *# 发送目标*             source = neig\_rank  *# 接收源*             ireq += 1             MPI.Isend(u\_snd, comm, reqs[ireq]; dest)  *# 异步发送*             ireq += 1             MPI.Irecv!(u\_rcv, comm, reqs[ireq]; source)  *# 异步接收*         end         if rank != (num\_ranks - 1)             neig\_rank = rank + 1  *# 右侧邻居的 rank*             u\_snd = view(u, (local\_load + 1):(local\_load + 1))  *# 发送的数据（边界值）*             u\_rcv = view(u, (local\_load + 2):(local\_load + 2))  *# 接收的数据（边界值）*             dest = neig\_rank  *# 发送目标*             source = neig\_rank  *# 接收源*             ireq += 1             MPI.Isend(u\_snd, comm, reqs[ireq]; dest)  *# 异步发送*             ireq += 1             MPI.Irecv!(u\_rcv, comm, reqs[ireq]; source)  *# 异步接收*         end         for i in 3:local\_load             u\_new[i] = 0.5 \* (u[i - 1] + u[i + 1])  *# 使用 Jacobi 迭代公式更新*         end         MPI.Waitall(reqs)         *# 更新边界单元*         for i in (2, local\_load + 1)             u\_new[i] = 0.5 \* (u[i - 1] + u[i + 1])  *# 使用 Jacobi 迭代公式更新*         end         *# 交换 u 和 u\_new*         u, u\_new = u\_new, u     end     return u end  function init(n, comm)     nranks = MPI.Comm\_size(comm)     rank = MPI.Comm\_rank(comm)     if mod(n, nranks) != 0         println("n must be a multiple of nranks")         MPI.Abort(comm, 1)     end     load = div(n, nranks)     u = zeros(load + 2)     if rank == 0         u[1] = -1     end     if rank == nranks - 1         u[end] = 1     end     u\_new = copy(u)     u, u\_new end  function gather\_final\_result(u, comm)     load = length(u) - 2     rank = MPI.Comm\_rank(comm)     if rank != 0         u\_snd = view(u, 2:(load + 1))         MPI.Send(u\_snd, 0, comm)         u\_root = zeros(0)     else         nranks = MPI.Comm\_size(comm)         n = load \* nranks         u\_root = zeros(n + 2)         u\_root[1] = -1         u\_root[end] = 1         lb = 2         ub = load + 1         u\_root[lb:ub] = view(u, lb:ub)         for other\_rank in 1:(nranks - 1)             lb += load             ub += load             u\_rcv = view(u\_root, lb:ub)             MPI.Recv!(u\_rcv, other\_rank, comm)         end     end     return u\_root end function jacobi(n, niters)     u = zeros(n + 2)     u[1] = -1     u[end] = 1     u\_new = copy(u)     for t in 1:niters         for i in 2:(n + 1)             u\_new[i] = 0.5 \* (u[i - 1] + u[i + 1])         end         u, u\_new = u\_new, u     end     u end n = 12 niters = 100 comm = MPI.Comm\_dup(MPI.COMM\_WORLD) u = jacobi\_mpi\_latency\_hiding(n, niters, comm) u\_root = gather\_final\_result(u, comm) rank = MPI.Comm\_rank(comm) if rank == 0     u\_seq = jacobi(n, niters)     if isapprox(u\_root, u\_seq)         println("Test passed 🥳")     else         println("Test failed 😢")     end end |



**Exercise2：**在雅可比法的并行实现中，我们假设该方法运行给定数量的迭代。在笔记本开头的函数jacobi\_with\_tol 中展示了当迭代之间的差异较小时如何停止雅可比迭代。实现这个函数的并行版本：

|  |
| --- |
| using MPI MPI.Init() function jacobi\_with\_tol(n, tol)     u = zeros(n + 2)       u[1] = -1      u[end] = 1      u\_new = copy(u)       while true         diff = 0.0          for i in 2:(n + 1)              ui\_new = 0.5 \* (u[i - 1] + u[i + 1])              u\_new[i] = ui\_new              diff\_i = abs(ui\_new - u[i])               diff = max(diff\_i, diff)           end         if diff < tol              return u\_new           end         u, u\_new = u\_new, u       end     u  end function jacobi\_mpi\_with\_tol(n, niters, tol, comm)     u, u\_new = init(n, comm)       local\_load = length(u) - 2       rank = MPI.Comm\_rank(comm)      num\_ranks = MPI.Comm\_size(comm)       nreqs = 2 \* ((rank != 0) + (rank != (num\_ranks - 1)))     reqs = MPI.MultiRequest(nreqs)       for t in 1:niters          ireq = 0          *# 处理与左侧邻居的通信*         if rank != 0             neig\_rank = rank - 1               u\_snd = view(u, 2:2)               u\_rcv = view(u, 1:1)               dest = neig\_rank              source = neig\_rank               ireq += 1             MPI.Isend(u\_snd, comm, reqs[ireq]; dest)  *# 异步发送*             ireq += 1             MPI.Irecv!(u\_rcv, comm, reqs[ireq]; source)  *# 异步接收*         end         *# 处理与右侧邻居的通信*         if rank != (num\_ranks - 1)             neig\_rank = rank + 1               u\_snd = view(u, (local\_load + 1):(local\_load + 1))              u\_rcv = view(u, (local\_load + 2):(local\_load + 2))              dest = neig\_rank              source = neig\_rank               ireq += 1             MPI.Isend(u\_snd, comm, reqs[ireq]; dest)  *# 异步发送*             ireq += 1             MPI.Irecv!(u\_rcv, comm, reqs[ireq]; source)  *# 异步接收*         end         *# 等待所有通信完成*         MPI.Waitall(reqs)         mydiff = 0.0           for i in 2:local\_load + 1             u\_new[i] = 0.5 \* (u[i - 1] + u[i + 1])              diff\_i = abs(u\_new[i] - u[i])               mydiff = max(mydiff, diff\_i)  *# 更新当前进程的最大差值*         end         diff\_ref = Ref(mydiff)          MPI.Allreduce!(diff\_ref, max, comm)  *# 全局归约，计算最大差值*         diff = diff\_ref[]           if diff < tol             return u\_new           end         *# 交换 u 和 u\_new*         u, u\_new = u\_new, u     end     return u end function init(n, comm)     nranks = MPI.Comm\_size(comm)       rank = MPI.Comm\_rank(comm)      if mod(n, nranks) != 0           println("n must be a multiple of nranks")         MPI.Abort(comm, 1)      end     load = div(n, nranks)       u = zeros(load + 2)       if rank == 0           u[1] = -1      end     if rank == nranks - 1          u[end] = 1       end     u\_new = copy(u)       u, u\_new   end function gather\_final\_result(u, comm)     load = length(u) - 2       rank = MPI.Comm\_rank(comm)       if rank != 0           u\_snd = view(u, 2:(load + 1))           MPI.Send(u\_snd, 0, comm)           u\_root = zeros(0)       else           nranks = MPI.Comm\_size(comm)           n = load \* nranks           u\_root = zeros(n + 2)          u\_root[1] = -1           u\_root[end] = 1           lb = 2           ub = load + 1           u\_root[lb:ub] = view(u, lb:ub)          for other\_rank in 1:(nranks - 1)               lb += load               ub += load               u\_rcv = view(u\_root, lb:ub)               MPI.Recv!(u\_rcv, other\_rank, comm)          end     end     return u\_root  end  function jacobi(n, niters)     u = zeros(n + 2)       u[1] = -1       u[end] = 1       u\_new = copy(u)       for t in 1:niters           for i in 2:(n + 1)              u\_new[i] = 0.5 \* (u[i - 1] + u[i + 1])          end         u, u\_new = u\_new, u       end     u  end n = 12   tol = 1e-6   niters = 1000   comm = MPI.Comm\_dup(MPI.COMM\_WORLD)  u = jacobi\_mpi\_with\_tol(n, niters, tol, comm)   u\_root = gather\_final\_result(u, comm)  rank = MPI.Comm\_rank(comm)   if rank == 0       u\_seq = jacobi\_with\_tol(n, tol)       if isapprox(u\_root, u\_seq)           println("Test passed 🥳")       else         println("Test failed 😢")       end end |

①初始化：网格u,u\_new 以及负载local\_load

②获取进程信息：获取当前进程的 rank 和总进程数 num\_ranks。

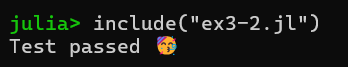
③计算需要处理的通信请求数 nreqs，取决于当前进程是否为边界进程。

④初始化通信请求数组 reqs，用于异步通信。

⑤迭代更新：

* + - 处理与左侧邻居的通信（当前进程不是第一个进程）：
      * 计算左侧邻居的 neig\_rank。
      * 准备发送和接收的数据 u\_snd 和 u\_rcv。
      * 异步发送 u\_snd 到左侧邻居。
      * 异步接收 u\_rcv 从左侧邻居。
    - 处理与右侧邻居的通信（当前进程不是最后一个进程）：
      * 计算右侧邻居的 neig\_rank。
      * 准备发送和接收的数据 u\_snd 和 u\_rcv。
      * 异步发送 u\_snd 到右侧邻居。
      * 异步接收 u\_rcv 从右侧邻居。
    - 调用 MPI.Waitall(reqs) 等待所有异步通信完成。
    - 更新内部单元：
      * 计算当前进程的最大差值 mydiff。
      * 使用 Jacobi 迭代公式更新内部单元 u\_new[i]。
    - 用 MPI.Allreduce! 计算全局最大差值 diff。
    - 如果全局差值diff小于容差tol停止迭代并返回u\_new。
    - 交换 u 和 u\_new。

⑥返回最终结果

****

**三、总结**

我了解了雅可比方法，串行实现方式以及，如何有效地并行化雅可比方法，一维分块与二维分块比较好，如果使用块分区，当问题规模较大于处理器数量时，通信开销会较小，还可以利用延迟隐藏来进一步减少通信所导致的开销。

除此之外，我们还学习了高斯-赛德尔法，作为简单变法有了正向逆向两种方法，以及特殊的红黑方法。最后，我学习了如何使用 MPI 实现一维雅可比方法。

在使用阻塞指令时需要小心，以避免死锁。