MPI-漫谈并行计算思路入门

制作：一条咸鱼

写在前面

- 本文以python作为**伪代码**，旨在介绍并行计算的基本思想，为正式进行程序设计做铺垫。
- 本文经不断查阅资料并实验综合得到，如有错误，欢迎校正。
- 部分案例和代码修改自该网站

微分面积

从连续值计算开始说起

考虑这样一个题目：给定函数$y = f(x)$，求解其与$x = a$、$x = b$、$y = 0$所围成图形的面积。

一个可行的思路如下：

1. 将该图形沿平行y轴方向，划分为n块
2. 建立n个进程，每个进程分配一块，单独计算该块面积
3. 累计即可得到近似面积

通过调整进程的数量，我们可以轻易的控制计算结果的精度，进程越多，划分数量越多，计算精度越高。

那么，如何分配每个进程计算的部分呢？灵活使用每个进程的进程号是一个不错的选择。

```
rank = comm.Get_rank() # 获取进程号
l = (b-a)/n # 获取每块长度
start = rank*l
end = start + l
```

走向离散（一）

如果更深入一点，想一下这种思路的具体的应用，不难发现该算法存在着一些弊端：

1. 计算精度和并行进程数量有关，进程数量越多，精度越高
2. 计算精度要求又与a-b相关，a-b越大，如果不增加进程数，误差也会越大
3. 硬件设备的并行线程数是有限制的，不可以无限增加

那么，在遇到极高精度要求时，这种方法就行不通了。因此我们可以尝试离散映射的方法

1. 设根据精度要求，要把图形切成N块，机器同时最多运行n个进程，且N是n的倍数
2. 每次计算n个图形，一共通过N/n次计算，累加获取总面积

以3个进程，6个块为例，该方案将块号映射到每个进程如下：

块号	1	2	3	4	5	6
进程号	1	2	3	1	2	3

同样可以使用进程号分配块号：

```
for i in N/n:
    for j in n:
        start = i*n*h # 当前进行了i个循环，每个循环里有n个块，每块长度为h
        end = start + h
```

走向离散（二）

刚刚的算法仍然存在缺点：即N必须整除n，才能得到正确答案，如果N是一个大质数，改进的算法便退化回“连续值计算”（即：每块一个进程）。为了避免这个问题，再次改进，可令前rank-1个进程正常计算分配到的块，最后一个进程计算剩余所有块。

以3个进程，8个块为例，该方案将块号映射到每个进程如下：：

块号	1	2	3	4	5	6	7	8
进程号	1	2	3	1	2	3	3	3

一种判断方法如下

```
for i in N/n:
   for j in n:
       if rank!=size-1:
           start = i*n*h # 当前进行了i个循环，每个循环里有n个块，每块长度为h
           end = start + h
       if rank==size-1:
           while k in range((b-i*n*h)/h): # 让最后一个进程计算剩余所有块
               start = (i+k)*n*h # 当前进行了i+k个循环，每个循环里有n个块，每块长度为h
               end = start + h
```

负载均衡

虽然上述方法让算法具有了较好的鲁棒性，但在效率上仍然不高。试想这样一种情况：

一共有1099个块，并行进程为100个，那么按以上算法，前99个进程各需计算10个块，最后一个进程需要计算109个块。由于总结果是每个进程结果的总和，因此必须等待最后一个进程计算完109个块后才能得到最终结果。这样做显然效率很低。当进程数n很大，且总块数N为 k * n+(n-1) 时，计算开销最高。

为了解决这个问题，我们可以尝试把余数中的块均匀分配到各个进程中，使各进程之间，计算的块数差最大仅为1。

仍然以3个进程，8个块为例：

块号	1	2	3	4	5	6	7	8
进程号	1	2	3	1	2	3	1	2

矩阵乘法

就代数计算方法而言，对矩阵乘法的优化思路如下：

1. 将左矩阵切割为n行，或将右矩阵切割为n列
2. 广播剩余的那个矩阵
3. 各进程计算切割后矩阵的乘积，收集则为答案

以切割左矩阵为例

进程1	x	x	x	x

进程1	x	x	x	x
进程1	x	x	x	x
进程2	x	x	x	x
进程2	x	x	x	x

这种并行方法，除了可以减少计算时间以外，还可以降低存储需求。即只需要主进程所在的机器，读取全部的两个矩阵，其余机器只需接收并存储切割后的矩阵即可。

如果想进一步降低对存储空间的需求，可以将两个矩阵都进行切割（左按行，右按列），计算并收集答案即可。

PageRank

并行优化

PageRank的计算本质而言，可以用矩阵乘法解决。

对于网页节点之间，输入各个节点之间的连通关系，那么有向连通图可以用类似如下矩阵表示（右上为出节点，左下为入节点）：

↙	node1	node2	node3
node0	0	0	1
node1	1	0	1
node2	0	1	0

同理，可求出各节点向其他节点转移的概率P如下：

↙	node1	node2	node3
node0	0	0	1
node1	0.5	0	0.5
node2	0	1	0

为了防止链接陷阱出现，每个节点还要单独设定一个概率值P_out，向任意结点跳转。（详见搜索引擎课程）

可以算出每次跳转，向各节点转移的概率P_rand如下：

↙	node1	node2	node3
node0	0.33	0.33	0.33
node1	0.33	0.33	0.33
node2	0.33	0.33	0.33

综上，设P_out=0.1，最终的状态转移矩阵如下：

计算公式为：$P \times (1 - P\_out) + P\_rand \times P\_out$

↙	node1	node2	node3
node0	0.033	0.033	0.9+0.033
node1	0.45+0.033	0.033	0.45+0.033
node2	0.033	0.9+0.033	0.033

各节点分值向量初始化如下：

	mark
node0	0.333
node1	0.333
node2	0.333

每轮迭代，相当于计算：状态转移矩阵 × 各节点初始分值向量

更新各节初始分值向量，即可进行下一轮迭代。

串行优化

PageRank算法除了在并行部分可以优化以外，串行部分也值得好好思考：

设最终状态转移矩阵为M，分值向量为V，迭代数为N，不难发现，整个迭代过程可以写为

$$M \times M \times \ldots \times M \times M \times V = \Pi_{i=1}^{N} M \times V \tag{1}$$

即：V左乘N次M。时间复杂度为$O(n)$

如果应用结合律，将累乘部分结合，如果N为奇数，则单独留一个M在外面最后计算。得到：

$$(M \times M) \times \ldots \times (M \times M) \times V = \Pi_{i=1}^{N/2} (M \times M) \times V \tag{2}$$

$$((M \times M) \times (M \times M)) \times \ldots \times ((M \times M) \times (M \times M)) \times V = \Pi_{i=1}^{N/4} ((M \times M) \times (M \times M) \times V \tag{3}$$

...

即：每轮迭代，优先计算累乘的部分，那么时间复杂度可以降低至$log(n)$

消失的分值

以上方案还有一个缺陷：当连通矩阵存在全0列时，计算值会收敛到接近全0。

实际情况对应为：至少存在一个节点，有其他节点连接进该节点，但该节点不连接出任意一个节点（包括自己本身）

原因如下：

- PageRank每轮循环的本质是将自己的节点分值交给其他节点，再收集其他节点的分值，从而实现节点分值的再分配。
- 此处"再分配"表现为：无论怎样循环，所有节点的分数总和必为1，不会增加也不会减少。因为分值在该集合内只是流动、转移，而不会增加或消失。
- 但是，如果一个节点只接收其他节点的分值，而不把自己节点的分值交给其他节点（甚至不交给自己）其他节点给与它的分值相当于被直接浪费（经过随机游走分散出去的分值除外）
- 因此每轮循环后，总分值越来越少，对应导致各节点分值不断收敛至0.

以总计5个网络节点，其中234节点均指向0节点，0节点指向5节点为例，可以画出如下的连通图，且第五列为全0：

	node0	node1	node2	node3	node4
node0	0	1	1	1	0
node1	0	0	0	0	0
node2	0	0	0	0	0
node3	0	0	0	0	0
node4	1	0	0	0	0

该连通矩阵经过计算结果如下：

	node0	node1	node2	node3	node4
第1轮循环	0.0704	0.0164	0.0164	0.0164	0.5204

	node0	node1	node2	node3	node4
第2轮循环	0.05708	0.0128	0.0128	0.0128	0.07616
第3轮循环	0.0379928	0.0034328	0.0034328	0.0034328	0.05480
...	...	...	...	...	
第100轮循环	1.30911492e-33	1.94594479e-34	1.94594479e-34	1.94594479e-34	2.693869 33

对此，存在两种可能的改进方法：

方案一：使连通图中主对角线恒为1

此方案实际意义为，认为每个网站一定连接到自己。这样就可以保留每次其他节点给予该节点的分值，避免最终结果收敛至全0的情况。

经过修改后，邻接矩阵如下所示：

	node0	node1	node2	node3	node4
node0	1	1	1	1	0
node1	0	1	0	0	0
node2	0	0	1	0	0
node3	0	0	0	1	0
node4	1	0	0	0	1

连通矩阵经过此方案计算结果如下：

	node0	node1	node2	node3	node4

	node0	node1	node2	node3	node4
第100轮循环	0.06909091	0.03636364	0.03636364	0.03636364	0.82181818

方案二：使全0列变为全1

此方案实际意义为：将没有任何外部链接的网络节点，所拥有的分值，平均分给所有其他节点，以此保证分值不消失。

经过修改后，邻接矩阵如下所示：

	node0	node1	node2	node3	node4
node0	0	1	1	1	1
node1	0	0	0	0	1
node2	0	0	0	0	1
node3	0	0	0	0	1
node4	1	0	0	0	1

连通矩阵经过此方案计算结果如下：

	node0	node1	node2	node3	node4
第100轮循环	0.33544878	0.09066183	0.09066183	0.09066183	0.39256573

相对于上一个方案，该方案好处为：最终分值的分布相对更加平均。也更能客观的表现网站之间的权重关系。

修正项

由于每个节点分值总和必为1，我们可以发现，随着节点数增加，各个节点的平均分数会相应降低。

这样会导致不同大小的局域网络计算出的节点分值，不具有可比性。阅读上也相当的不适。

十个节点的某次计算结果如下：

	node0	node1	node2	...	node7	node8	node9
第100轮循环	0.09848767	0.04922033	0.11370036	...	0.09530293	0.0497375	0.07028376

一百个节点的某次计算结果如下：

	node0	node1	node2	...	node97	node98	node99
第100轮循环	0.01867948	0.0089862	0.00314754	...	0.01728934	0.00682841	0.00925973

为了阅读上的便捷，以及使各个大小的网络具有可比性，我们可以增加一个修正项：将最终结果均扩大“节点数”倍。