**一、pagerank简介**（参考书籍《推荐书籍实践》和csdn若干博客 可以跳到第二部分 需要解决的问题）

**1.PageRank的核心思想**

如果一个网页被很多其他网页链接到的话说明这个网页比较重要，也就是PageRank值会相对较高。如果一个PageRank值很高的网页链接到一个其他的网页，那么被链接到的网页的PageRank值会相应地因此而提高.

**2.Pagerank是什么**

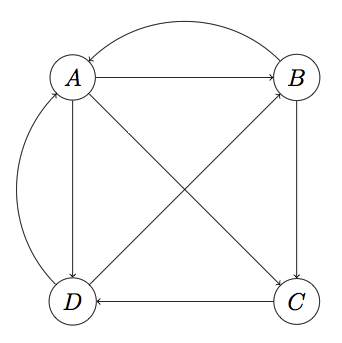
将各网页之间的链接关系按照幂次迭代矩阵A形式表示，那么各个网页的PageRank值将组成一个长度为URLNUM的列向量，该列向量就是矩阵A的特征值为1对应的特征向量。

即 PagerRank就是幂次迭代矩阵A的特征值为1的特征向量的元素值  
即 求解PageRank的值就是求解幂次迭代矩阵A的特征值为1对应的特征向量

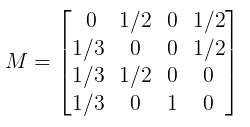
**3.简单PageRank计算**

首先，我们将Web做如下抽象：1、将每个网页抽象成一个节点；2、如果一个页面A有链接直接链向B，则存在一条有向边从A到B（多个相同链接不重复计算边）。因此，整个Web被抽象为一张有向图。

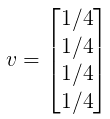
现在假设世界上只有四张网页：A、B、C、D，简单情况下假设这个图是强连通的（从任一节点出发都可以到达另外任何一个节点）：



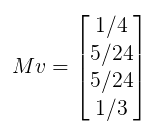
然后需要用一种合适的数据结构表示页面间的连接关系。其实，PageRank算法是基于这样一种背景思想：被用户访问越多的网页更可能质量越高，而用户在浏览网页时主要通过超链接进行页面跳转，因此我们需要通过分析超链接组成的拓扑结构来推算每个网页被访问频率的高低。最简单的，我们可以假设当一个用户停留在某页面时，跳转到页面上每个被链页面的概率是相同的。例如，上图中A页面链向B、C、D，所以一个用户从A跳转到B、C、D的概率各为1/3。设一共有N个网页，则可以组织这样一个N维矩阵：其中i行j列的值表示用户从页面j转到页面i的概率。这样一个矩阵叫做转移矩阵（Transition Matrix）。下面的转移矩阵M对应上图：



然后，设初始时每个页面的rank值为1/N，这里就是1/4。按A-D顺序将页面rank为向量v：



注意，M第一行分别是A、B、C和D转移到页面A的概率，而v的第一列分别是A、B、C和D当前的rank，因此用M的第一行乘以v的第一列，所得结果就是页面A最新rank的合理估计，同理，Mv的结果就分别代表A、B、C、D新rank：

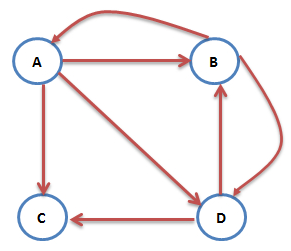


然后用M再乘以这个新的rank向量，又会产生一个更新的rank向量。迭代这个过程，可以证明v最终会收敛，即v约等于Mv，此时计算停止。最终的v就是各个页面的pagerank值。例如上面的向量经过几步迭代后，大约收敛在（1/4, 1/4, 1/5, 1/4），这就是A、B、C、D最后的pagerank。

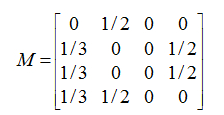
**二、需要解决的问题**

**1.dead ends**

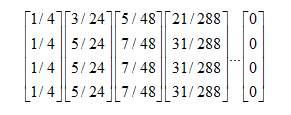
实际上，Web并不是强连通（甚至不是联通的）。下面看看PageRank算法如何处理一种叫做Dead Ends的情况。所谓Dead Ends，就是这样一类节点：它们不存在外链。看下面的图：



对应的转移矩阵为：

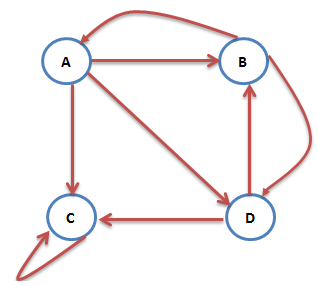


连续迭代下去，最终所有元素都为0：

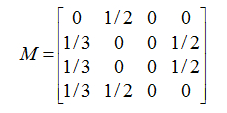


**2.spider trap**

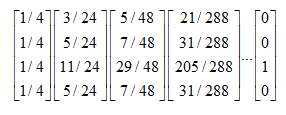
另外一个问题就是陷阱问题，即有些网页不存在指向其他网页的链接，但存在指向自己的链接。比如下面这个图：



上网者跑到C网页后，就像跳进了陷阱，陷入了漩涡，再也不能从C中出来，将最终导致概率分布值全部转移到C上来，这使得其他网页的概率分布值为0，从而整个网页排名就失去了意义。如果按照上面图对应的转移矩阵为：



不断的迭代下去，就变成了这样：



**3.稀疏矩阵**

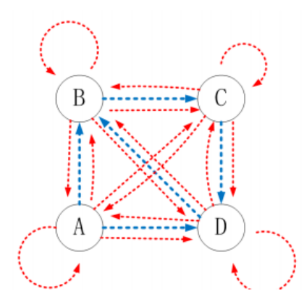
可以预见，如果把真实的Web组织成转移矩阵，那么这将是一个极为稀疏的矩阵，从矩阵论知识可以推断，极度稀疏的转移矩阵迭代相乘可能会使得向量v变得非常不平滑，即一些节点拥有很大的rank，而大多数节点rank值接近0。同时，稀疏矩阵的特性使得我们可以在存储和计算的时候进行巧妙的设计来减少存储的空间以及计算的复杂度。

**4.分块计算**

虽然本次的wikiData.txt还不需要分块操作，但现实中上千万上亿的页面绝对是需要的分块操作的，当然上面对稀疏矩阵的优化处理已经帮我们缓解了很多存储和计算的压力，但是真正的大数据面前，分块计算开始必要的。

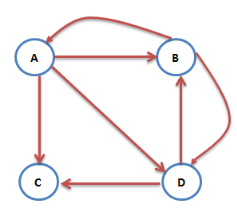
**三、解决思路及模型建立**

用户的浏览行为其实不仅仅只是浏览某个页面，然后点击该页面中的某个链接，进行下一个页面访问。有时候用户浏览完某个页面之后，会随机浏览另外的页面，而不是根据该页面的链接。所以为了解决上述几个问题，同时也更能模拟真实情况，我们可以建立如下的模型：



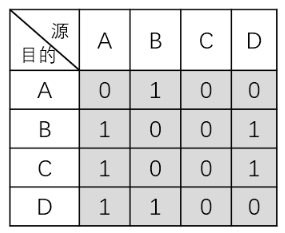
红色箭头代表每次用户有一定的几率随机输入一个网址进行跳转，当然因为不知道用户喜好所以每个点的跳转的概率都是一样的。蓝色箭头代表原始网页的链接，遵循刚刚我们简单模型的一切。所以我们的模型变为，在每一个页面的用户都有alpha的概率点击链接进入下一个页面，此时使用简单pagerank规则，然而还有（1-alpha）的概率随机跳转，此时每个页面的概率都为1/URLNUM。

所以根据这个思路我们就可以建立解决方案的模型了（以下图为例）：



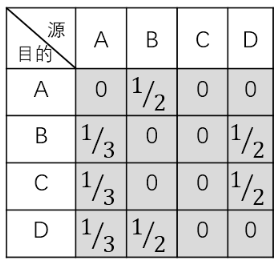
**（1）邻接矩阵G**

用邻接矩阵表示各个网页之间的链接关系，为了方便迭代中矩阵相乘的运算，要求G\_ij中的某一个元素是从ID= i 的源网页链接到ID= j 的目的网页的关系值。如果有链接关系值为1，如果没有链接关系值为0。换句话来说，行号对应的是目的网页ID，而列号对应的是源网页ID,且邻接矩阵的大小是URLNUM x URLNUM.



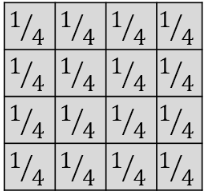
**（2）状态转移概率矩阵Gm**

设定每一个网页的pagerank初始值为1，那么由该网页链接出linkOutNum个网页，每个网页获得的pagerank值为1/linkOutNum。将邻接矩阵当中的非零元素值都进行这样的计算之后再替换原本所有的1。状态转移概率矩阵的大小也为URLNUM x URLNUM.



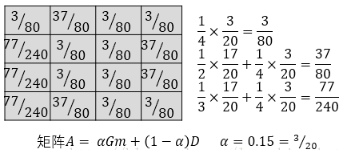
**（3）校正基础矩阵D**

为了避免spider trap等问题，需要根据公式进行校正。如果，那么就会有一个校正基础矩阵D，与邻接矩阵叠加形成最后参与幂次迭代矩阵A。校正基础矩阵的每一个元素都等于1/URLNUM，且大小为URLNUM x URLNUM



**（4）幂次迭代矩阵A**

校正基础矩阵D与概率转移矩阵Gm按照如下图中的公式叠加形成参与幂次迭代的矩阵A。且该幂次迭代矩阵A大小为URLNUM x URLNUM



**（5）迭代**

各个网页的pagerank值，其实是关于矩阵A的特征值为1的特征向量。要求pagerank值就是要求特征向量。求解特征向量的计算规模是O(n^3)，也就是求得慢同时还有可能求不准。所以只能采用迭代的方式，最终迭代出收敛的特征向量，也即待求的pagerank数组。

所以构造好上述矩阵后我们的做法就是：迭代初始向量PageRank[URLNUM]与幂次迭代矩阵A相乘，直到满足精度条件。即

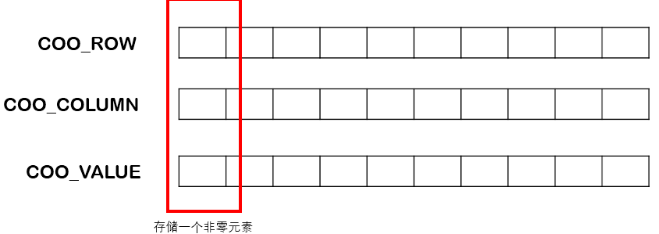
**a)LastRank[ URLNUM ] = PageRank[URLNUM]**

**b)PageRank[URLNUM] = A× LastRank [ URLNUM ]**

Until |PageRank[URLNUM] – LastRank[URLNUM]| < limitation

**（6）对稀疏矩阵的优化**

因为GM稀疏矩阵大部分都是零，导致A矩阵大部分都是（1-alpha）/ URLNUM，所以我们可以考虑对其存储和计算下手进行优化。对于存储方式上课PPT中介绍了一种，但是为了更加方便我采用COO稀疏矩阵模式，用行数组、列数组和value数组三个数组，将非零元素的位置(行、列)和值(value)存储下来。会占据更大的空间但是更好访问各个数据。



对应这种处理稀疏矩阵的方式，我们可以优化我们的计算方式。在程序具体实现矩阵相乘迭代过程中，首先将所有元素点都按照无链接关系处理，完成一部分累加。然后遍历COO数组，将有链接关系的点重新累加到迭代关系中，同时减去原本该位置按照无链接关系处理时的累加值。最终即可得到完成所有累加操作的pagerank数组。

**（7）deadend的解决**

上面的算法之所以能成功收敛到非零值，很大程度依赖转移矩阵这样一个性质：每列的加和为1。在没有Dead Ends的情况下，每次迭代后向量v各项的和始终保持为1，而有了Dead Ends，迭代结果将最终归零

处理Dead Ends的方法如下：迭代拿掉图中的Dead Ends节点及Dead Ends节点相关的边（之所以迭代拿掉是因为当目前的Dead Ends被拿掉后，可能会出现一批新的Dead Ends），直到图中没有Dead Ends。对剩下部分计算rank，然后以拿掉Dead Ends逆向顺序反推Dead Ends的rank。

**（8）矩阵分块计算**

以上都是在较小规模的page时候的情况，当URLNUM远大于内存容量的时候，我们就需要进行分块计算。当然，第一步把稀疏矩阵处理成COO存储的模式我们已经在前面做过了，这一点也大大帮我们减少了内存的压力以及计算的压力。然而如果URLNUM还是太大，那我们就要使用block\_strip的方法来进行分块处理。



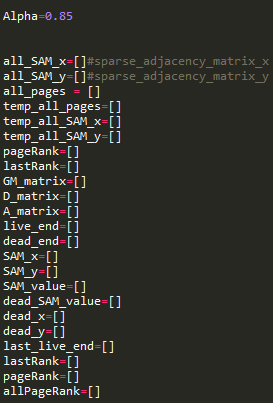
思路很简单，每次需要做的就是将block\_size大小的pagerank加载进内存，由于我们的问题在上边被我们模型化成了矩阵和向量的乘法，所以结果向量就是等于A矩阵的每一行乘以lastrank的对应值即可，所以每个block我们会遍历一次lastrank以及A矩阵对应的行，所以总的来说每次迭代我们需要加载一次pagerank向量，URLNUM/block\_size次的lastrank以及一次的A矩阵即可。

**（9）额外工作**

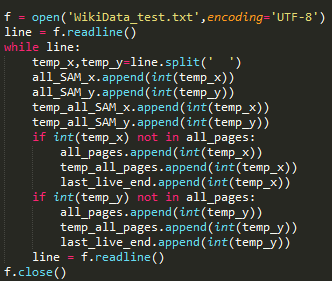
结合这学期并行课程为没做稀疏化处理的矩阵向量乘法做并行计算处理，包括SSE、OPENMP、Pthread等方法的对比。

**四、核心代码介绍**

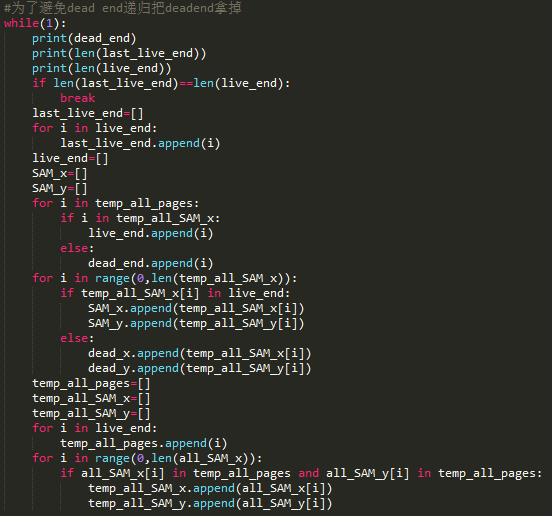
实现了四种方式，一是矩阵向量乘法的方式，二是coo稀疏矩阵优化以后的方式，三是block\_strip是在coo的基础上进行的，最后结合本学期并行课程对第一种方式进行了并行化处理



这是代码中的重要变量，SAM\_x和SAM\_y和SAM\_value是用来存储我们live\_end的coo稀疏矩阵。dead\_x和dead\_y和dead\_value是用来存储我们dead\_end的coo稀疏矩阵。All\_SAM\_x和all\_SAM\_y是用来存储我们allpages的coo稀疏矩阵。all\_pages是统计所有出现的页面，pagerank和lastrank存储着live\_end的score向量,allpagerank是所有页面的rank。GM D A三个matrix则是存储上文说的对应核心矩阵。所有带temp的都是在一开始为分开live和dead操作而设置的。



这是文件读取的内容，将wikiData的txt文件内容转化为了all\_SAM\_x all\_SAM\_y



这里看过去一大串但实现的就是将live以及dead end分开的操作，每次都在temp的全集里找到没有出度的点，把他们以及对应的边加到dead那边去，每次比较两次迭代之间live end有没变化，没有的话说明我们的递归找dead完成了

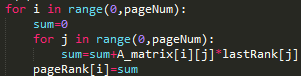
**接下来我们分四种方法逐个介绍实现情况（主要以（2）为主）**

**（1）A矩阵lastRank向量暴力乘法方式**



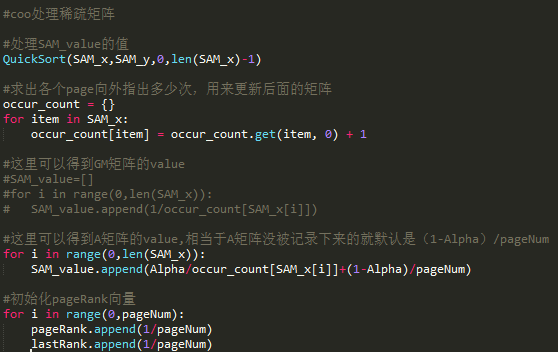
暴力矩阵乘法不做过多说明，我们主要的思路讲解在（2），矩阵乘法方式我们的侧重点在（4）并行化加速部分，而（3）则是按照要求模拟大批量数据内存无法放下时的情况，主要在（2）上进行的修改。

这里思路简单，就是按照思路里的内容分别初始化GM D A三个矩阵以及lastRank和pageRank两个向量

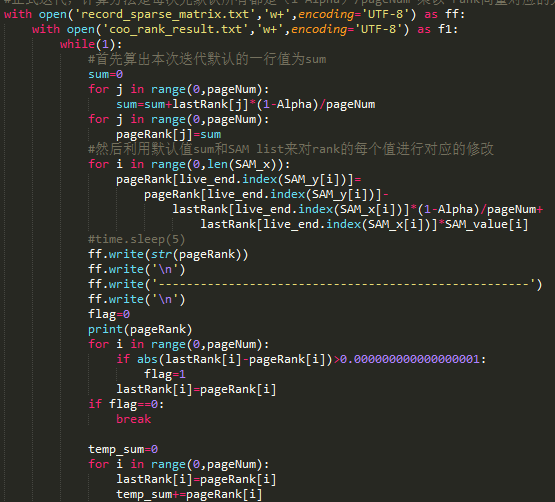


进入正式迭代以后最关键的步骤就是这里，相当于一个简单的矩阵乘法，不在赘述。

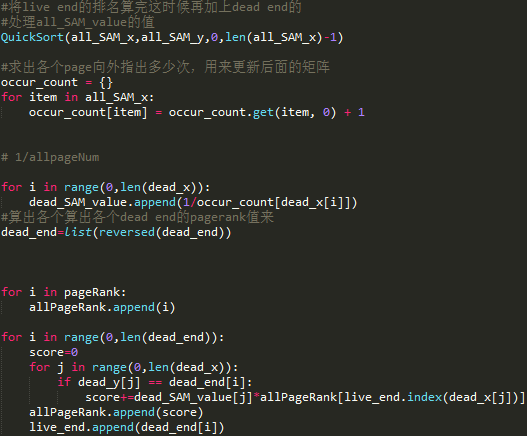
**（2）coo稀疏矩阵优化后迭代计算**



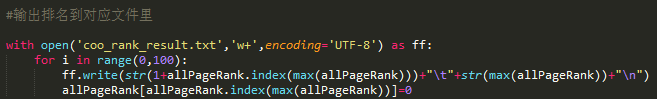
首先是要处理SAM\_value的内容，为了方便处理我们先调用一个自己写的快速排序的代码把两个list从小到大排序。之后统计live\_end中的每个pages的出度数，这样我们就可以转化为对应A矩阵里的对应位置的值了，通过我们的公式以及GM和D矩阵以及alpha系数。然后我们初始化pagerank。



正式进入迭代以后，我们要做的工作就是之前思路里捣鼓的，先按默认的值为pagerank计算值，这是利用了稀疏矩阵大部分都是0（当然这里大部分都是（1-alpha）/pageNum）的特点减少计算量，然后对应稀疏矩阵中非“0”值逐个操作把他们找到对应的value然后改变对应的pagerank值。这之后我们设置了一个阈值10e-10，如果两次pagerank的所有元素差距都小于次，那么说明程序收敛了。于是乎我们就能得到所有live\_end的pagerank



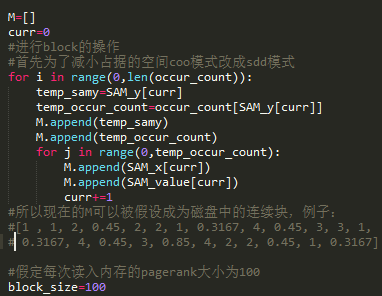
计算完live\_end的值我们需要计算dead\_end的值，还是先用快排把所有点的按顺序排列一下，然后计算每个点的出度数并转化为（1/出度数）的各页面value值供之后计算dead\_end的pagerank使用。这边我们需要注意的是为了让每次新的dead\_end加入时候指向它的end都是live的，我们需要把dead\_end翻转一下，和加入dead\_end的顺序相反就可以保证这一点。最后遍历deadend所有点，按照思路中提供的公式计算并加入pagerank向量即可。



最后把pagerank前一百名输出到对应result.txt文件里

**（3）block\_strip 模拟超大数据量处理**

Block strip部分我们主要讲怎么从（2）的情况下模拟大数据量下对矩阵以及向量都进行分块后的处理。

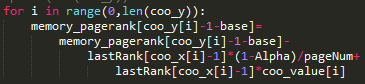


我们看到M相当于一个大磁盘，所以为了模拟这个大磁盘我们先要在已有的coo稀疏矩阵的基础上制作这个存储的内容，这里我们选用的存储方式是src dgree和destination的方式，即源头点，指出的数量，所有目的点。如上图可以看出一番操作以后M变成了上述存储方式的磁盘，大大节省了空间。这里curr我把他叫做磁盘探针，也是模拟真实情况。

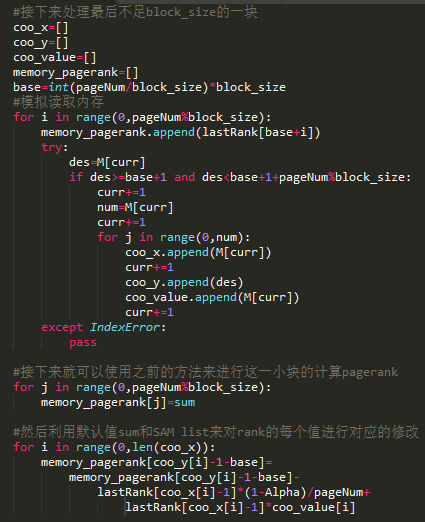
这之后一堆初始化操作都和（2）中一样不在赘述



进入迭代以后，虽然大致思路和（2）一样，不过要分成block和strip后还是麻烦了很多，首先假设block\_size以后我们得为pagerank分pageNum/blocksize+1次操作，除了最后一次外每次都是把对应blocksize大小的pagerank加载进memory pagerank里，当然这些加载结束后的操作其实就是缩小版的（2）



可以看到核心操作还是和（2）一样，只不过每次对象的size小了很多



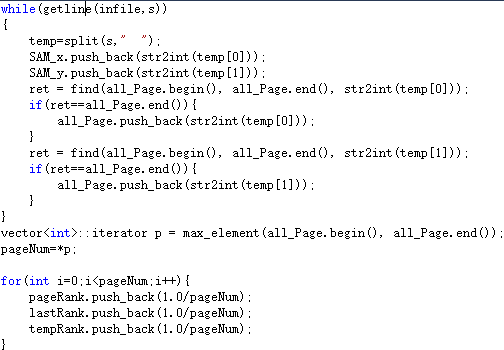
当然不要忘记最后一次不够block size大小的操作。这之后完成的效果就是（2）中live\_end的rank排序，之后deadend的部分和（2）一样不做赘述。

**（4）并行加速**

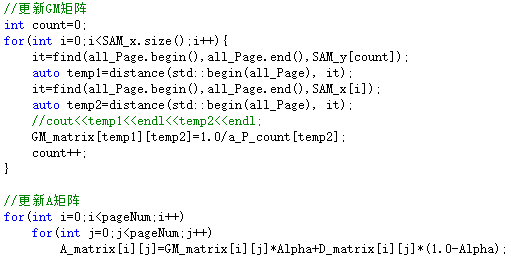
**首先将python的暴力法改为C++**

为了在本学期以及熟悉的windows+vs环境下实现上述工具的并行化，我们把对应的代码改成了C++版本

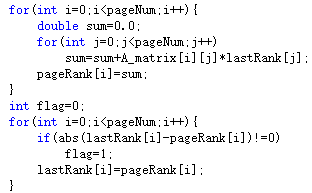
1.对应的读取文件操作



2.对应的三个矩阵初始化操作

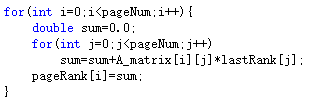


3.迭代部分核心矩阵向量乘法部分



**将问题抽象成普通的矩阵向量乘法**

对于我们代码中最可以改进的部分我们将其提出来并首先抽象成一般性问题进行探讨：



矩阵乘法串行算法如下面伪代码所示：

**procedure** Matrix\_Mul (*A,B,C*)

**begin**

**for** i := 1 **to** col **do**//row

*tmp=0*;

**for** *j* := 1 **to** row **do**//col

tmp += A[i\*col + j]\*B[j]

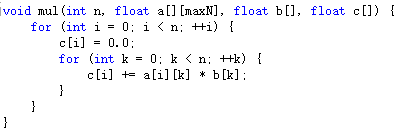
**end for**

C[i] = tmp

**end for**

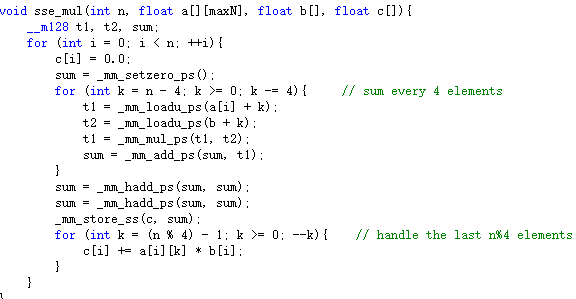
通过分析矩阵相乘的伪代码我们可以发现，其中可以实现并行的是其中的第六行部分。

所以抽象后的矩阵向量基础代码如下：



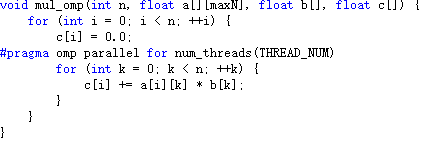
**SSE 算法设计与实现**

用SSE指令将串行矩阵相乘中的乘法加法进行改写，主要思路为：将A的行和B的列的点分为每四个（对）为一组进行点乘和并行加法运算，余下的不足四个的组额外处理。主要代码如下，编程环境为VS2010：

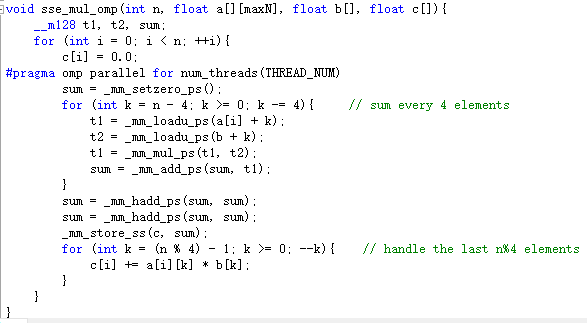


**openmp算法设计与实现**

先对最初的矩阵乘法进行在OpenMP 并行化，只需用parallel 语句声明了需要并行的代码段将用omp 加速：

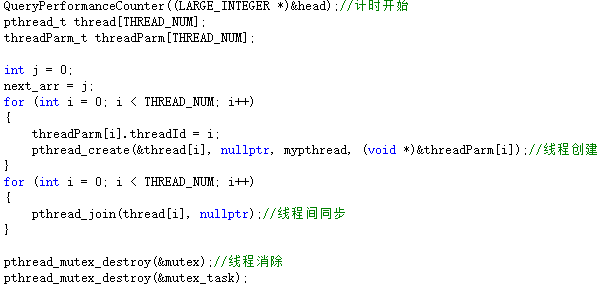


然后SSE算法的基础上进行OpenMP 并行化，同样用parallel 语句声明代码将用omp 加速，如下：

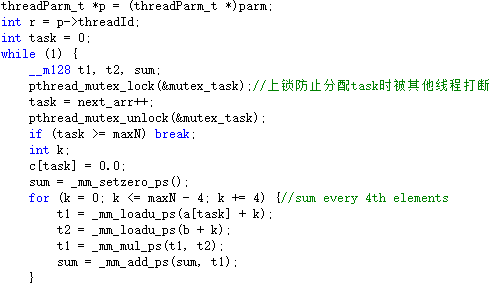


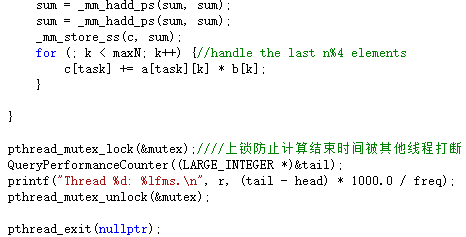
**pthread算法设计与实现**

线程的创建和维护主要是通过pthread 的函数进行线程创建、线程间同步、线程消除等工作，主要代码如下：



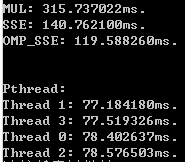
由于需要计算的数据都是以行作为整体，所以在分配数据时，我们将每一行整体分配给一个线程。然后确定分配方法，因为计算量分布无规律，为了能让每个线程的负载较为均衡，运行时间大致相同，采用随机分配的方法，即每当有线程空闲时，将需要先处理的数据行随机分配给线程。在mypthread 中，我们完成了对任务的划分，实现了线程内的具体执行函数，其中两处需要引入互斥量（通过加/解锁）保证线程同步，代码如下：





**在7000数据量下的结果对比**

由于我们正式运用的情况是7000\*7000的矩阵和7000\*1的向量这个量级的乘法，所以我们做一下各个方法的对比



可以看出应该还是结合了SSE算法的pthread在这个量级表现最优，时间大约是原来的暴力法的4~5倍，表现优异。

最后再将pagerank暴力法中的矩阵向量乘法带入到pthread SSE方法中去即可。

**五、结果展示**

最后我们对我们的结果进行展示和说明。

首先需要注意的是，我们的pagerank.py里面包含着（1）（2）（3）三种方式的代码，pagerank.cpp和parallelization.cpp是并行化这边的代码。Py文件默认执行的是（2）coo稀疏化矩阵优化的代码。如果想要跑（1）暴力法，请把pagerank.py文件的132~237的注释符号去掉，并且为266~341加上注释；同样如果要跑block\_strip分块化矩阵的代码，把344行之后的注释符号去掉并且把266~341加上注释即可。而cpp文件方面，pagerank.cpp是改写后的暴力法，在parallelization.cpp是SSE openmp和pthread的实现和对比。



首先是运行python代码的运行示意图，在.py文件目录下输入python pagerank.py即可运行，最终达到一定精度以后得到如上结果所示代表运行完成。可以得到result和record两类文件分别记录着迭代结果的前100名排名和分数，以及迭代过程中每次迭代的各pagerank向量的值。结果如下两图。



