稀疏矩阵的矩阵向量乘法的并行算法性能

作者：王舜　　指导老师：王小鸽

清华大学计算机科学与技术系

王舜：wangshun98@mails.tsinghua.edu.cn

王小鸽：wangxg@mail.tsinghua.edu.cn

摘 要：现代的科学计算中的最常用的基本算法就是矩阵向量乘法。所以一个快速高效的矩阵向量乘法的并行算法将给整个科学计算带来诸多的进步。但是随着处理器性能迅速提高和通信速度发展的相对滞后，并行算法的通信屏障显得越来越明显。而稀疏矩阵带来的问题是，计算量较小，通信量较大。本文将阐述稀疏矩阵矩阵向量乘法几种的并行实现，以及实现中的计算复杂度和通信复杂度。通过在计算机机群 ACI 上的测试结果显示出算法性能，并分析了通信量爆炸对并行算法性能的影响。

关键字：矩阵向量乘法，稀疏矩阵，并行算法，复杂度，通信

前言

并行计算的基本原理就是将计算任务分散到多个处理器同时进行计算，以获得成倍的计算速度。但是由于 p 个处理器并行工作的时候需要花费一些时间进行相互之间的协同和通信，所以并行程序最终获得的计算速度达不到串行程序的 p 倍。

如今，随着处理器性能的迅速提高，单台计算机的计算能力大大增强。而相比之下，多台计算机之间的通信速度的提高却相对滞后。这种计算能力和通信能力的不平衡日益增大，越来越妨碍了并行计算的发展。使得并行计算缩短下来的计算时间大多都用于通信，甚至可能通信时间比节约的计算时间还长，从而体现不出并行计算的优势。

矩阵向量乘法是现代科学计算中的一项基本运算。尤其是在线性代数算法中，如 Conjugate Gradient 迭代，它可以算得上是一项核心运算。由于数据规模的飞速扩大，这些迭代算法现在正在迅猛发展，代替直接法成为更加可行有效的算法。而对矩阵向量乘法的并行化的改进，也将直接影响迭代算法效率的提高。

本文中描述的矩阵向量乘法采用的数据结构都是基于矩阵的按列分布。原因是该文来源于我们正在做的稀疏对称非正定系统的双层迭代算法的并行实现。该项目中的其他算法决定了矩阵的数据结构最好采用按列分布。

我们在做这个项目的时候，发现并行程序的加速比上不去，做了大量实验和分析以后，发现问题的一个瓶颈在于稀疏矩阵的矩阵向量乘法上。这篇文章里，我们将要讨论的就是上面所说的计算机机群的计算能力和通信能力的相对不平衡对矩阵向量乘法的并行化的影响。其中第一部分描述了我们用到的稀疏矩阵的数据结构。第二部分介绍了一个最基本的算法，并对其性能进行了分析，这一部分是全文的重点。第三部分对算法进行了改进。最后给出结论和展望。

一、稀疏矩阵的数据结构

首先，为了讨论方便，我们给出稀疏矩阵和向量的数据结构。稀疏矩阵的数据结构由图 1 表示，具体 C 语言定义参见附录 1。整个矩阵按照列块分布在各个处理器上，即每个处理器上仅拥有稀疏矩阵的连续几列。按列分布的原因上面已经讲到，是在一个大的科学计算项目中，别的并行运算决定的。

而向量是密集的，其定义相对简单，用数组表示即可，在每个处理器上都保留完整备份。

二、基本的算法及其性能分析

按照矩阵向量乘法的定义，$y_i = \sum_j A_{ij} x_j$，如果记 $z_{ij} = A_{ij} x_j$，则有 $y_i = \sum_j z_{ij}$，如图 2 所示。

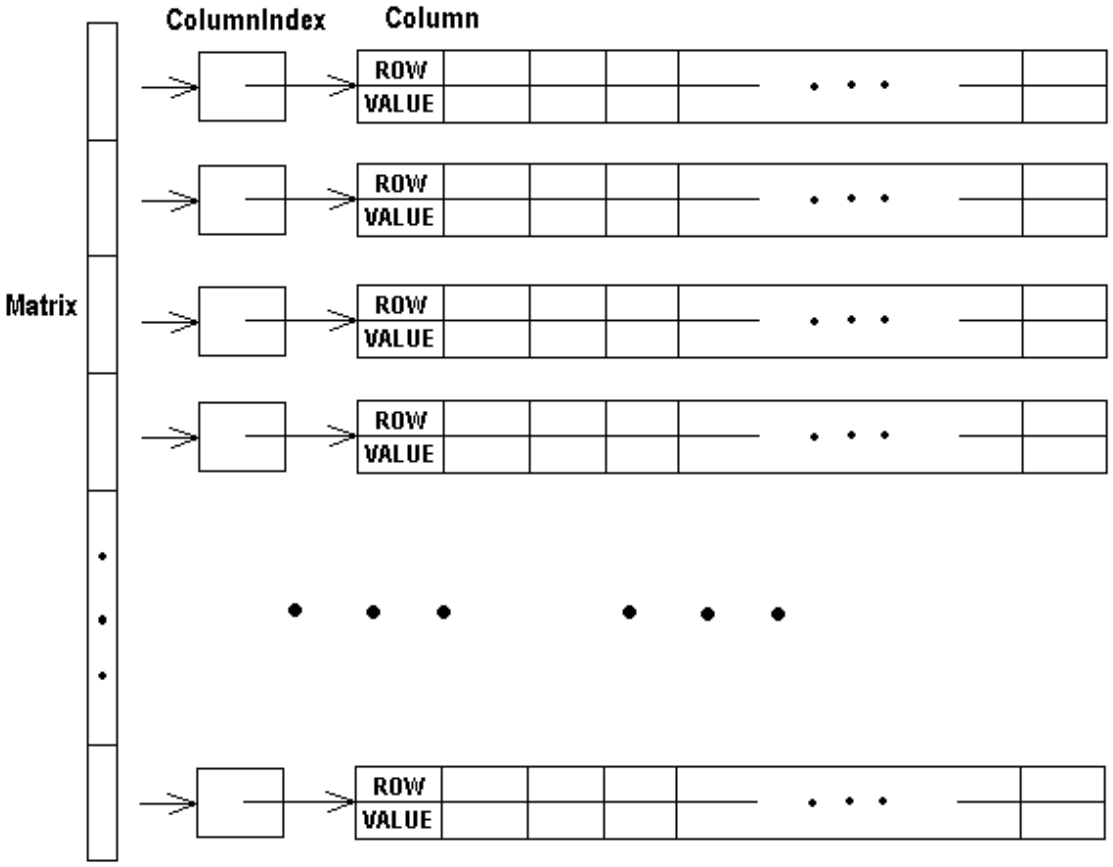


图 1

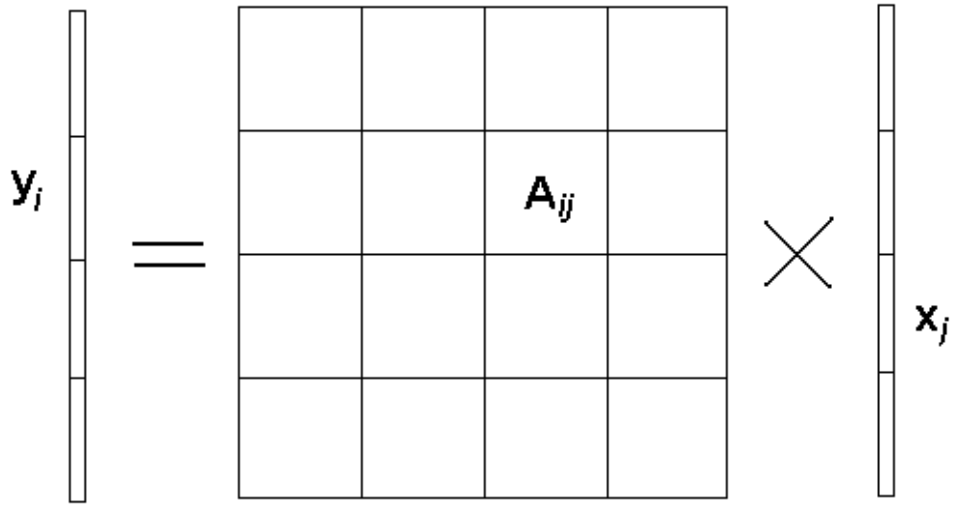
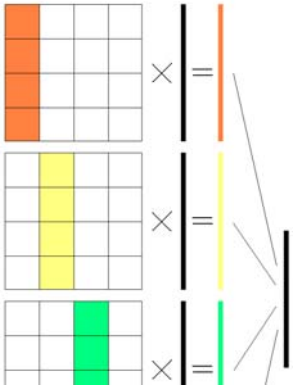


图 2



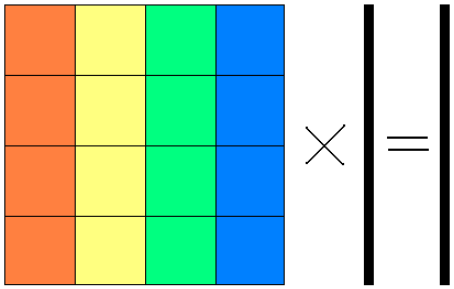


图 3

一个最基本简单的算法就是，各个处理器根据自己拥有的独立数据，进行矩阵非零元和向量中相应位置元素的乘法和累加，可以分别计算出结果向量的部分和。例如，第 j 个处理器上拥有 $A_{ij}(\text{对所有}i)$ 和整个向量，依据这些数据我们可以得到 $z_{ij}(\text{对所有}i)$ 。最后可以通过多个处理器的交互通信，把这些结果加到一起，如图 3 所示。

我们采用下面的通信方式，完成这样分布在多个处理器内存上的向量的加法。不妨设处理器个数 $p=2^k$ ，算法描述如下：

```
i=1;
while i<p do
    dest=(myid+i)%p;
   src =(myid-i+p)%p;
   send vector v to dest;
   receive vector v1 from src;
   v=v+v1;
    i=i*2;
end;
```

如此，一个 8 处理器的系统将按图 4 所示步骤完成向量求和。

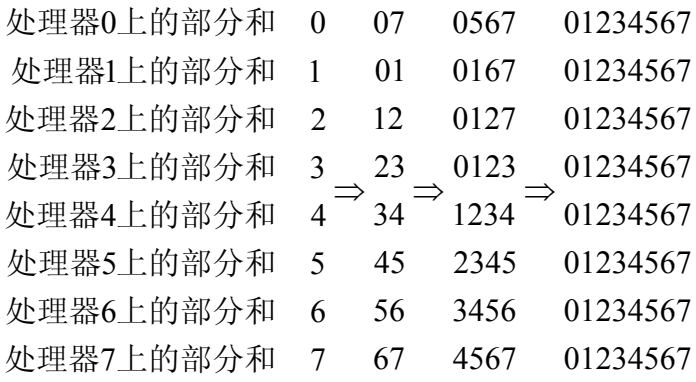


图 4

在不同的机群拓扑结构上，不同的通信方式会有不同的效率。在这里，我们尝试了三种不同的通信方式。

同步方式：每步通信时，所有处理器同时向目标处理器发送数据，然后进行接收。

交替方式：将处理器分成两组，一组先发送数据，后接收；对应的另一组先接收，后发送。例如 8 个处理器的时候，0、2、4、6 号处理器先发送后接收，而 1、3、5、7 号处理器先接收后发送。

轮转方式：将处理器看成环状排列，起始节点先发送，后面的节点都是接收到前面节点的数据以后，再向后继节点发送数据，最后一步是由起始节点接收末尾节点的数据。

比较这三种方式，其中同步方式可能的通信冲突最多，而轮转方式则浪费的等待时间最多，交替方式的通信冲突和通信等待时间介于这两者之间。对于这三种方式，我们在 ACI 机群上做了很多实验（ACI 机群的配置参见附录 2），得到结果列于附录 3 的表格中，其中 MPI 方式是指利用标准的 MPI_Allreduce 接口函数实现通信。

从上述结果和图示可以看到，同步方式和轮转方式在处理器增多的时候效果都很差。前者是因为处理器增多造成通信阻塞增多；后者的通信实际上是按照串行方式进行的，处理器越多，浪费的等待时间也就越多。这样看来，交替方式是效率最高的方式，跟 MPI_Allreduce 接口函数的效率不相上下，甚至更好。

另外，上述结果中，我们可以看到很重要的一点就是，在 ACI 机群上，并行矩阵向量乘法的时间不仅和矩阵规模成正比，还随着处理器个数增加而变长，并没有达到并行运算的目的。

为了解释上述现象，下面我们分析一下算法的复杂度。假设矩阵大小为$n \times n$，稀疏矩阵平均每列的非零元个数为n_1，则稀疏矩阵的密集度为$\rho = n_1 / n$。首先，各个处理器上的计算总量为$\rho n^2 / p$次乘法，$\rho n^2 / p + n \log p$次加法。由于 n 相对于 p 很大，所以可以忽略$n \log p$项，于是各个处理器上的总计算量为$2\rho n^2 / p$次浮点运算。再来考虑通信复杂度，由上可见，整个通信过程分为$\log p$次完成。每次每个处理器都要发送和接受 n 个单位的数据，设一个单位的数据占 c 个字节，于是各个处理器上的总通信量为$cn \log p$。不妨设机群中处理器的运算速度为 s MFlops，通信速度为 u MB/s，。又假设每一步中各个处理器的通信可以同时进行，没有阻塞，互不影响，达到最高的通信效率。那么整个运算时间（微秒）可用下面的公式表示：

$$t = 2\rho n^2 / p \cdot 1/s + cn \log p \cdot 1/u$$

对 p 求偏导，得到

$$\frac{\partial t}{\partial p} = -\frac{2\rho n^2}{p^2} \cdot \frac{1}{s} + \frac{cn}{p} \cdot \frac{1}{u}$$

由$\partial t / \partial p = 0$，可以得出$p = \frac{2\rho n}{c} \cdot \frac{u}{s} = \frac{2n_1}{c} \cdot \frac{u}{s}$。也就是说，处理器个数的最优选择跟两个因素有关，一是矩阵规模和矩阵密集度，即稀疏矩阵平均每列的非零元个数；二是通信速度和计算速度的比。首先，对于稀疏矩阵，增大数据规模对于增大并行程序的加速比并没有直接好处，只要每列的非零元个数保持不变，并行程序的效率就提升不了。更重要的，我们发现了影响并行程序的加速比的第二个因素。当通信速度滞后计算速度太多，盲目的增加处理器的个数，反而会降低整个矩阵向量乘法的效率。另外，实际上由于机群中处理器的拓扑结构的限制，它们之间的多项通信任务不可能完全同时进行，从而实际的通信时间比上述公式里还长。

我们以 ACI 机群和本文前言中提到的项目为一个典型例子，验证上面的分析。首先，很容易测的 ACI 机群单机的运算能力，它的浮点运算速度可以达到 30MFlops。然后，用 Ping-Pong 法可以测出机群中两台机器间的通信速度，大概为 4-8MB/s。本文前言中提到的项目中，稀疏矩阵的每列非零元素个数大约在 7 左右。非零元数值用 float 类型表示，float 类型的长度一般为 4 个字

节。按照这些数据计算，并行计算处理器个数的最佳选择 $p^* < 1$。也就是说，这样的情况下，并行程序起不到加快运算的目的，只能在大量的通信中浪费时间，通信时间比计算时间还多很多。

我们按照上述公式估算出理论值，和实际测得的结果进行比较。图 5 就是对附录 3 中 size=25210 的结果作出的图，这里的实际值采用的是最优的交替方式的结果，通信速度取 5MB/s。可以看出，实际结果和理论值非常接近。

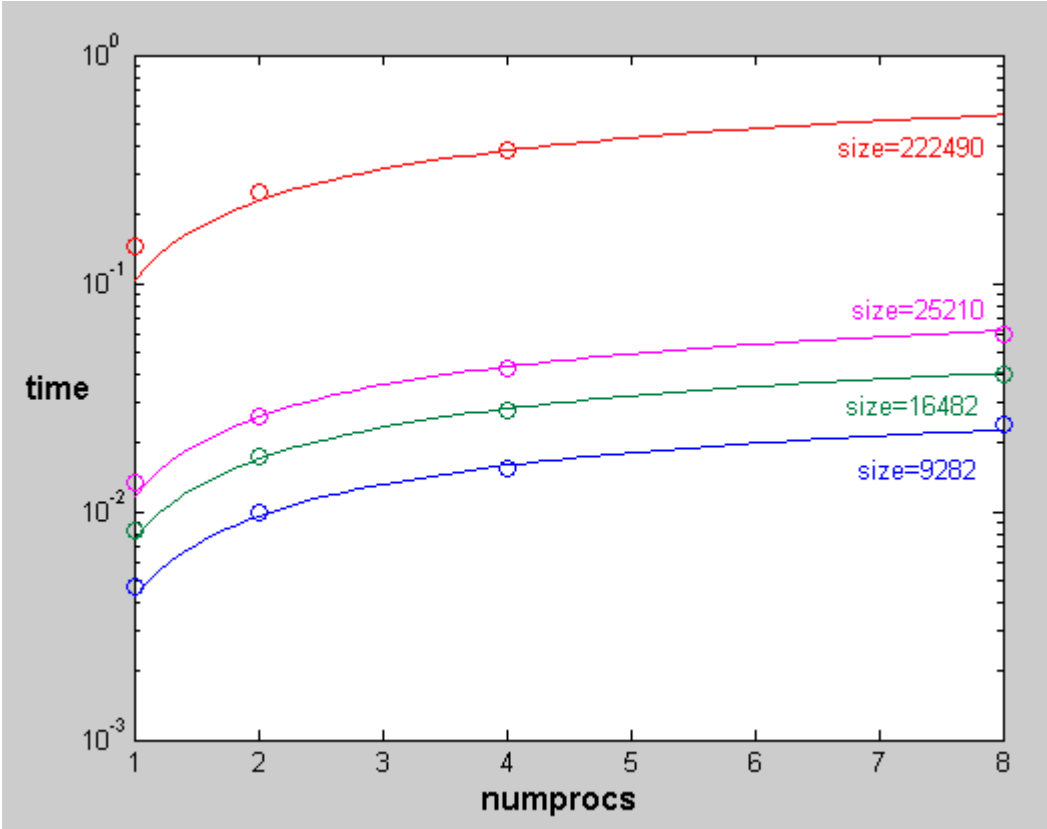


图 5

三、改进的算法

考虑到上述计算中，第 j 个处理器需要的数据实际上只有 A_{ij}(对所有i) 和 x_j，我们可以只在各个处理器上保留部分的向量，不仅对于原向量，也对于结果向量。这样一来，上述的向量求和过程中，各个处理器只需要的其他处理器上相应的一段向量即可。

另外，为了进一步提高效率，可以将通信安排在计算过程中间进行，使得与计算获得最大的重合。算法描述如下：

```
t=0;
j=myid;
while t<p do
      i=(myid+t)%p;
      k=(myid-t+p)%p;
      compute z_ij = A_ij x_j ;
    if t<>0
send z_ij to i;
receive z_jk from k;
y_j = y_j + z_jk ;
      else
y_j = z_jk ;
      end;
      t=t+1;
end;
```

算法分为 p 步，每一步是在各个处理器上根据本地的数据计算出一个 z_{ij} 。除了第一步以外，将算出的 z_{ij} 发送到相应的处理器上加到结果向量中去，一次一个处理器的通信量为 n/p 。这样，各个处理器的平均通信量降为 $(p-1)\cdot n/p$ 。具体的通信方式可以参见图 6。

考虑到稀疏矩阵的特殊数据结构，当上述改进算法利用到稀疏矩阵上的时候，需要事先对稀疏矩阵进行分块定位，找到稀疏子矩阵 A_{ij} 在原矩阵中的列起始位置。在这里我们的作法是对每个矩阵在所有运算开始前，预先计算出一个伴随结构，记录每列对于各个分块的起点，终点和长度，方便后面的计算。

改进后的方法的效率同样列在附录 3 的表格中，并行速度有了明显的提高，比未改进时用标准的 MPI_Allreduce 接口函数和交替方式都快很多。但是加速比上不去的缺陷还是存在。我们仍然对改进后的方法进行理想情况下的复杂度分析。得到

$$t = 2\rho n^2 / p \cdot 1/s + cn \cdot (p-1)/p \cdot 1/u ,$$

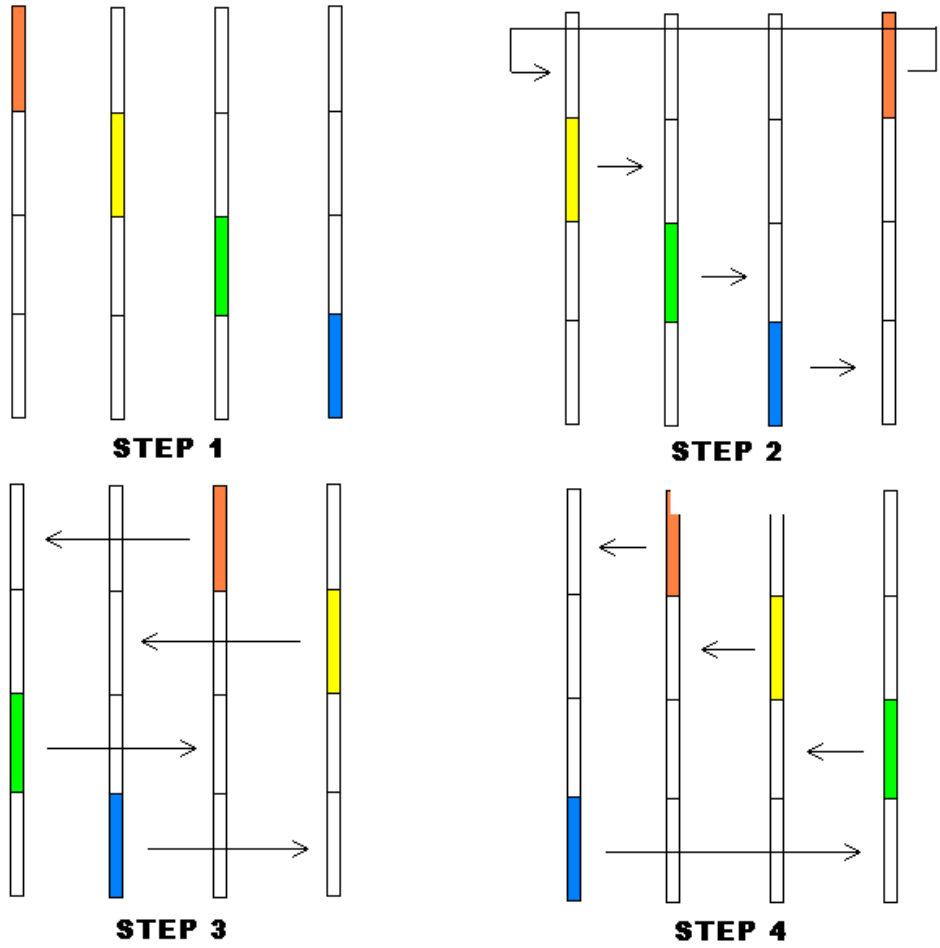


图 6

于是

$$\frac{\partial t}{\partial p} = -\frac{2\rho n^2}{p^2}\cdot\frac{1}{s} + \frac{cn}{p^2}\cdot\frac{1}{u} = \frac{n}{p^2}\left(\frac{c}{u} - \frac{2n_1}{s}\right)。$$

可以看出，如果 $n_1 < \frac{cs}{2u}$，那么 $\partial t/\partial p > 0$，计算时间随处理器增多而加长。也就是说，这种情况下，加速比和稀疏矩阵平均每列的非零元素个数密切相关，每列的非零元达到一定数量，改进的并行算法才可能出现较好的加速比。

当然，上述分析是在比较理想的情况下讨论的。改进的算法的一个最大优点就是能够让计算和通信尽可能的重合起来，减少通信阻塞对计算的影响。但是，我们也要看到，改进的算法使得下面两个问题变得明显起来：一是因为当处理器增多的时候，由于稀疏矩阵的非零元的不规则分

布，各个处理器上的多个分块之间本身就存在非零元个数相差较大的问题，于是分步计算的时候，处理器之间的负载很难达到平衡，可能出现较多等待的情况；二是因为处理器增多的时候，单个消息的长度变短，消息头的比例增大，增加了冗余的通信量，给通信任务增加了更多的负担。

小结

在这里，我们已经看到稀疏矩阵的矩阵向量乘法的瓶颈所在，一是稀疏度高导致计算量相对较小，通信量相对很大，二是机群系统的低速通信和高速计算之间的不平衡。可见现在大规模计算基于的硬件系统，应该投入更多精力在高速的通信设施和结构的研究上。

附录一：数据结构的 C 语言定义

```
typedef struct {
   int row;
   float val;
} Column;

typedef struct {
   int col;
   int nonzeros;
   int colsize;
   Column *index;
} ColumnIndex;

typedef struct {
   int rowsize,colsize;
   int nonzeros;
   int col,from;
   ColumnIndex *column;
} Matrix;

typedef struct {
   int size;
   float *element;
} Vector;
```

附录二：ACI 机群的配置

```
CPU    : PIII 733MHz
Cache   : 256K
Networks: Ethernet
```

附录三：ACI 机群上的实验结果

size	nonzeros	numprocs	time (sec)				
			同步	交替	轮转	MPI	PART
9282	63926	1	0.004765	0.004713	0.004651	0.004680	0.005573
9282	63926	2	0.010043	0.009949	0.009766	0.009800	0.004983
9282	63926	4	0.014210	0.015427	0.022323	0.015596	0.007401
9282	63926	8	0.062890	0.024222	0.050307	0.02522	0.010398
16482	114010	1	0.008339	0.008278	0.008273	0.008337	0.009939
16482	114010	2	0.014981	0.017355	0.017535	0.017426	0.009491
16482	114010	4	0.035804	0.027700	0.039853	0.027885	0.021572
16482	114010	8	0.069530	0.039580	0.088247	0.044943	0.016142
25210	174726	1	0.013431	0.013409	0.013328	0.013355	0.015433
25210	174726	2	0.024147	0.026250	0.026463	0.027088	0.015500
25210	174726	4	0.060149	0.042073	0.060868	0.042559	0.027846
25210	174726	8	0.113717	0.059829	0.135810	0.068129	0.022796
222490	1552294	1	0.146757	0.145358	0.145497	0.145393	0.164053
222490	1552294	2	0.255025	0.252532	0.252111	0.256043	0.174237

222490	1552294	4	0.410589	0.386611	0.550810	0.395246	0.212587

参考文献

Bruce Hendrickson, Robert Leland and Steve Plimpton, *An Efficient Parallel Algorithm for Matrix-Vector Multiplication*, Sandia National Laboratories, Albuquerque, NM 87185

L.M. Romero and E.L. Zapata, *Data Distributions for Sparse Matrix Vector Multiplication*, University of Malaga, J. Parallel Computing, vol. 21, no. 4, April 199