7-2004

Reduced Index Sparse Representation in a Parallel Environment

Pedro Alfonso Escallon

Reduced Index Sparse Representation in a Parallel Environment

by

Pedro Alfonso Escallon

Master of Science
Electrical Engineering
Florida Institute of Technology
Melbourne, Florida
1987

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
July 2004

We the undersigned committee
hereby approve the attached thesis

Reduced Index Sparse Representation in a Parallel Environment

by
Pedro Alfonso Escallon

Charles Fulton, Ph.D.
Professor, Mathematical Sciences
Thesis Advisor

R. G. Deshmukh, Ph.D., P.E.
Associate Professor, Computer Engineering

Eraldo Ribeiro, Ph.D.
Assistant Professor, Computer Sciences

William Shoaff, Ph.D.
Associate Professor and Head, Computer Sciences

Abstract

Title:	Reduced Index Sparse Representation in a Parallel Environment.
Author:	Pedro Alfonso Escallon
Advisor:	Charles Fulton, Ph.D.

Sparse-matrix/dense-vector multiplication algorithms are not as highly developed as algorithms for dense matrices. Dense matrix multiplication algorithms have been made efficient by exploiting data locality, parallelism, pipelining, and other types of optimization. Sparse matrix algorithms, on the other hand, encounter low or no data locality, indirect addressing, and no easy way to exploit parallelism. In an effort to achieve savings in storage and computation time, the topic of sparse matrix representation is often revisited.

The first contribution of this thesis is the introduction of a new representation for sparse matrices. This representation is called here the Reduced Index Sparse (RIS) representation because an ordered pair (j,v) with a single index j is used for every nonzero matrix element. This considerably reduces the required disk storage from that needed by other representations like the coordinate (COO) format, which uses an ordered triple (a_{ij}, i, j) with two indices, i and j for every nonzero matrix element.

The second contribution of this thesis is a modified block cyclic data distribution for sparse matrices with arbitrary nonzero structure. And the third contribution is the implementation of this distribution using RIS to perform a parallel sparse-matrix/ dense-vector multiplication on a distributed memory computer using MPI. The implementation achieved good load balancing for sparse matrices of different sparsity patterns.

Software was written to generate large random sparse matrices having well prescribed sparsity patterns. The desired number of nonzero elements and matrix density are user input. The sparsity pattern is also controlled by user input.

Performance of the new RIS storage scheme was measured against SPARSKIT routines. Efficiency and scalability of the parallel sparse-matrix/dense-vector multiplication was generally good and timing analysis shows the new RIS scheme is competitive.

Contents

1 A Matter of Representation **1**

1.1 Introduction 1
1.2 Notation 2
1.3 Dense Representation 2
1.4 Coordinate Representation 3
1.5 Compressed Sparse Row Representation 6
1.6 Reduced Index Sparse Representation 9
1.7 Reduced Index CSR Representation 13
1.8 Example 15
1.9 Comparison 16

2 Sparse Matrix Multiplication in Parallel **18**

2.1 The Column Balanced Partition 19
2.2 The Balanced Column Cyclic Partition 20
2.3 The Balanced Block Cyclic Partition 21
2.4 Sample Partitions 24

3 RIS Block Cyclic Implementation **30**

3.1 Introduction 30
3.2 Function Main 31
3.3 Function readMatrix 34
3.4 Function risMul 36
3.5 Function whichProcess 37
3.6 Function getColumnIndices 38
3.7 Validation Software 38

4 Sparse Matrix Generator **40**

4.1 Main 40

5 Results **42**

5.1 Validation Results 42

5.2 Load Balancing 44

5.3 Timing Results 48

6 Conclusion **57**

7 Future Work **59**

A Source Code **60**

A.1 *getColumnIndices* 60

A.2 *Program ftest.f* 61

B Support Software **62**

B.1 *Makefile* 62

Bibliography **64**

List of Tables

5.1 Validation Results 43
5.2 Load Balance Results with 16 Processors 45
5.3 Load Balance Results with 12 Processors 47
5.4 Sample Results 48
5.5 Column Balanced Results 49
5.6 Row Balanced Results 51
5.7 Block Cyclic Results 52
5.8 Myrinet Results 54

List of Figures

1.1 Sample Matrix - Dense Format 2
1.2 Sample Matrix in Coordinate format 4
1.3 Sample Matrix in CSR Format 7
1.4 Sample Matrix in RIS Format 10
1.5 RIS Representation Rules 11
1.6 Sample Matrix in RCSR Format 13
1.7 Sample Matrix 16
1.8 Human readable RIS 16
1.9 Grey Scale 16

2.1 Column Balanced Partition 20
2.2 Unpartitioned Matrix 25
2.3 Row Balanced Partition 26
2.4 Column Balanced Partition 27
2.5 Balanced Block Cyclic Partition 27
2.6 Balanced Column Cyclic Partition 28
2.7 Balanced Row Cyclic Partition 29

3.1 double RIS structure 30
3.2 Complex value structure 30
3.3 int RIS structure 30
3.4 complex RIS structure 30
3.5 Function Main 32

4.1 Sample matrix.mcfg file 40

5.1 RIS, CSR Performance 43
5.2 RIS Accuracy 43
5.3 Load Balance Results with 16 Processors 45
5.4 Load Balance Results with 12 Processors 47
5.5 Column Balanced Results 49
5.6 Column Balanced Efficiency 49
5.7 Row Balanced Results 52
5.8 Row Balanced Efficiency 52
5.9 Block Cyclic Results 53
5.10 Block Cyclic Efficiency 53
5.11 Multiplication Results 53
5.12 Overall Timing Results 53
5.13 Reduce Time Results 54
5.14 Averaged Column Balanced 54
5.15 Myrinet Timing Results 55
5.16 Myrinet Reduce Times 55
5.17 Speedup and Cost (Myrinet) 55
5.18 Efficiency (Myrinet) 55

Notation

Symbol	Name
cP	The current process
nP	The number of processors used
nCD	The number of column divisions
nRD	The number of row divisions
Pc	The number of column cycles
cD	The column division
rD	The row division
Pr	The number of row cycles
nPG	The number of process grids
nCG	The number of column groups
nRG	The number of row groups
pCC	The process column cycle
pc	The column process coordinate
pr	The row process coordinate
Y	Vector Y
(y_i)	Vector Y
y_i	i_{th} element of Y
A	Matrix A
(a_{ij})	Matrix A
a_{ij}	(i,j) entry of A
M	The number of rows
N	The number of columns
nnZ	The number of nonzero elements

Acknowledgments

I thank Dr. Howell for helping me in the initial stages of this thesis. I recognize Dr. Fulton for guiding me in finding the topic of my thesis and for the advise and direction he provided as my thesis advisor. I thank Dr. Deshmukh and Dr. Ribeiro for being part of my thesis committee. I recognize Dr. Shoaff for providing an excellent Computer Science Seminar on Fridays that is a good source of ideas for thesis and research. I thank Dr. Gang Qin for giving me room to work in the parallel computer so I could run my software. I thank Mr. Dale Means for helping me resolve a network problem.

I recognize and thank my mother for her support during the preparation of this thesis.

I recognize and thank God for answering my prayers, and for keeping me humble.

I dedicate this thesis to my parents.

Chapter 1

A Matter of Representation

1.1 Introduction

Sparse matrix representation is doomed to have index information attached to the matrix element value. In dense representation, indexing is implied by the position of the datum in whatever structure the matrix is represented (usually lists of arrays). For sparse matrices, the simplest representation is the coordinate format [1] in which each element has row and column indices attached to the nonzero element that is represented. All the representations previously known and proposed here have two aspects to consider. One is disk storage which is how the matrix is saved in a disk file for later retrieval. The other aspect is the way to store the matrix in random access memory for processing. This second aspect is more decisive in obtaining efficient algorithms.

According to Im [2], there are hundreds of sparse matrix formats. Of those we will consider only two, the coordinate (COO), and the compressed sparse row (CSR). And we will introduce two new ones, the reduced index sparse (RIS), and the reduced index compressed sparse row (RCSR). In this chapter we will treat each of these representations mathematically.

A small sample matrix (figure 1.1) will be used for illustration. All the representation examples of this chapter refer to this matrix.

$$\begin{bmatrix} 1.7 & 0 & 0 & 0 & 6.5 & 0 & 0 & 0 \\ 0 & 0 & 2.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6.2 & 0 & 0 & 0 & 0 & 0.5 & 5.3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7.8 & 0 & 6.7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3 \end{bmatrix}$$

Figure 1.1: Sample Matrix - Dense Format

1.2 Notation

The matrix and vector notation of Golub and Van Loan [12] is adopted here. Borrowing the introductory portion of this notation, Let $\mathbb{R}$ be the set of real numbers. Let $\mathbb{R}^{M\times N}$ be the vector space of all $M \times N$ real matrices,

$$A \in \mathbb{R}^{M\times N} \iff A = (a_{ij}) = \begin{bmatrix} (a_{11}) & \cdots & (a_{1N}) \\ \vdots & & \vdots \\ (a_{M1}) & \cdots & (a_{MN}) \end{bmatrix},\ a_{ij} \in \mathbb{R}$$

Matrices are denoted by uppercase letter, and their corresponding lower case letter with subscript ij refers to the (i,j) entry. $A(i,j)$ notation is also used to designate matrix elements.

Let $\mathbb{R}^n$ be the vector space of real n-vectors. Then,

$$X \in \mathbb{R}^n \iff X = (x_i) = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix},\ x_i \in \mathbb{R}$$

As in matrix notation, vectors are expressed in uppercase letters, their corresponding lower case letter with subscript i refers to the ith entry. X(i) notation is also used to designate vector elements.

1.3 Dense Representation

Let $Y = (y_i) \in \mathbb{R}^M$ be the product vector. Let $X = (x_j) \in \mathbb{R}^N$ be the right hand side vector. Let $A = (a_{ij}) \in \mathbb{R}^{M\times N}$ be a sparse matrix represented in dense format. The product $Y = AX$ can be expressed as

$$y_i = \sum_{j=1}^{N} a_{ij} x_j, \quad \forall i = 1 : M \tag{1.3.1}$$

Using C style syntax, equation (1.3.1) produces algorithm 1.1. In this algorithm each y_i should be initialized to 0 prior to calling the function *mul*, because in every iteration of the inner loop y_i is updated. In every such update, the memory of y_i is accessed to read the current value, and again to store the updated value.

Algorithm 1.1. *Dense Format Multiplication Algorithm*

```
void mul( int nnZ, double A[][N], double X[], double Y[] )
{
   int i,j;
   for (i=1;i<=M;i++) {
      for (j=1;j<=N;j++) {
         Y[i] += A[i][j] * X[j];
      }
   }
}
```

1.4 Coordinate Representation

To avoid re-describing established representations, quoted descriptions will be given when appropriate. Im [2] states that the "coordinate representation stores each nonzero matrix element with row and column integer indexes along with a floating point value" or floating point value pair if the matrix is complex[1]. "It consists of three arrays: a real array of size nnZ containing the real values of nonzero elements of A in any order, an integer array containing their row indices and a second integer array containing their column indices" according to Saad [9].

Let $A = (a_{ij}) \in \mathbb{R}^{M \times N}$ be a real sparse matrix with nnZ nonzero entries that we want to represent in coordinate format. The coordinate representation (COO) of A consists of three arrays:

- A real one-dimensional array $\tilde{A} = (j_k, \tilde{a}_k) \in \mathbb{R}^{nnZ}$ that contains only the nonzero entries a_{ij} of A in any order.
- An integer array $I = (i_k)$ also of length nnZ that contains the row indices of the corresponding a_{ij} elements stored in $\tilde{A}$.

[1] For simplicity, we only consider real matrices, but all representations treated here can be extended for use with complex matrices.

%%MatrixMarket matrix coordinate real general			
%	M	N	nnZ
	8	8	9
%	i	j	val
	1	1	1.7
	1	5	6.5
	2	3	2.0
	4	2	6.2
	4	7	0.5
	4	8	5.3
	6	5	7.8
	6	7	6.7
	8	8	0.3

Figure 1.2: Sample Matrix in Coordinate format

- An integer array $J = (j_k)$ also of length nnZ that contains the column indices of the corresponding a_{ij} elements stored in $\tilde{A}$.

Let $A' = \{(a_{ij}, i, j)\} \in (\mathbb{R} \times \mathbb{Z}^+ \times \mathbb{Z}^+)^{nnZ}$ be the set of triples corresponding to the nonzero entries of A. Let the triple consisting of the three arrays $(\tilde{A}, I, J)$ be the COO representation of A. Then there is a function $f : A' \to (\tilde{A}, I, J)$ as $f(t) = (\tilde{A}(k), I(k), J(k))$ for each triple $t = (a_{ij}, i, j) \in A'$. In other words, for each triple $t \in A'$ there is a k, $1 \leq k \leq nnZ$ such that

$$a_{ij} = \tilde{A}(k) = \tilde{a}_k$$

$$i = I(k)$$

$$j = J(k)$$

With this we can express A as an array of triples that can be mapped to a triple consisting of three arrays $(\tilde{A}, I, J)$. This is what the coordinate representation is. Using Y, X, and A as defined for equation (1.3.1), and replacing the index notation we obtain $Y = (y_i) = (y_{I(k)})$, $X = (x_j) = (x_{J(k)})$, and $A = (a_{ij}) = (\tilde{A}(k)) = (j_k, \tilde{a}_k)$ for $k = 1 : nnZ$. With these, equation (1.3.1) becomes,

$$y_{I(k)} = \sum_{k=1}^{nnZ} \tilde{a}_k x_{J(k)} \tag{1.4.1}$$

Using C style syntax, let $A[k]$, $I[k]$, and $J[k]$ be the k^{th} elements of $\tilde{A}$, I, and J respectively. Let $X[j]$ be the j^{th} element of X, and $Y[i]$ be the i^{th} element of Y. With these definitions algorithm 1.2 mm_SparseMul that corresponds to equation (1.4.1) is shown next.

Algorithm 1.2. *Coordinate Format Multiplication Algorithm*

```
void mm_SparseMul( int nnZ, double A[], int J[], int I[],
                   double X[], double Y[] )
{
   int k;
   for(k=0;k<nnZ;k++) {
      Y[I[k]] += A[k] * X[J[k]];
   }
}
```

Given the fact that $\tilde{A}(k)$ contains the nonzero elements of A in any order, the associated arrays of $\tilde{A}$, $I(k)$ and $J(k)$ are not necessarily nondecreasing functions of k as k varies from 1 to nnZ. In other word, the order of the triples in A' can be completely arbitrary. Algorithm 1.2 makes use of indirect addressing for the components of X and Y. So $Y[I(k)]$ and $Y[I(k+1)]$ for arbitrary k are with very high probability not contiguous in memory. The same argument applies to $X[J(k)]$ and $X[J(k+1)]$ for arbitrary k. Thus, this algorithm is very inefficient from the memory access point of view.

The storage of this representation is illustrated using the sample matrix in coordinate format in figure 1.2. The minimum storage required by the coordinate representation on an MxN real double precision sparse matrix with nnZ nonzero elements is

$$St_{mm} = sizeof\,(int) * (2 * nnZ) + sizeof\,(double) * nnZ \tag{1.4.2}$$

assuming a double has 8 bytes and an int has 4 bytes, $St_{mm} = 16 * nnZ$ bytes.

Ignoring cache efficiency issues, the relative cost of algorithm 1.2 is calculated. In order to do that, the algorithm is re-written below using the style of Brassard and Bratley [6]. Let c be the upper bound of the time required to perform a loop test $k < nnZ$, an assignment (as in $k \leftarrow 0$), and an integer addition (as in $k \leftarrow k+1$). Indirect addressing as in $X[J(k)]$ is handled at the machine level by first calculating $J(k)$ into a temporary memory location j. This is shown below as $j \leftarrow J(k)$. Then this temporary internal variable is used to calculate $X[j]$. This has the same effect as calculating $X[J(k)]$. Likewise, to perform a multiplication and an addition as in $A + B \times C$, the processor first computes $B \times C$ into a temporary memory location, let us call it bc ($bc \leftarrow B \times C$). Then $A + B \times C$ is calculated by $A + bc$. In some architectures, a product such as $A + B \times C$ is called a fused add-multiply which is more efficient than performing both operations separately. Using a worst case scenario, we will consider them as separate operations. Let t be the upper bound of the time required to perform a floating point operation (as in $tmp \leftarrow A(k) \times X(j)$). Then the loop time is bounded above by $\ell \leq (2nnZ)t + (4nnZ + 2)c$ as seen below

Loop	**Cost**
$k \leftarrow 0$	c
while $k < nnZ$ **do**	$(nnZ + 1)\, c$
$\quad i \leftarrow I(k)$	$(nnZ)\, c$
$\quad j \leftarrow J(k)$	$(nnZ)\, c$
$\quad tmp \leftarrow A(k) \times X(j)$	$(nnZ)\, t$
$\quad Y(i) \leftarrow Y(i) + tmp$	$(nnZ)\, t$
$\quad k \leftarrow k + 1$	$(nnZ)\, c$
	$(2nnZ)t + (4nnZ + 2)c$

1.5 Compressed Sparse Row Representation

The Compressed Sparse Row (CSR) format consists of three arrays:

- A real array $\tilde{A} = (j_k, \tilde{a}_k)$ of length nnZ that contains only the nonzero values a_{ij} of A in row order.
- An integer array $J = (j_k)$ also of length nnZ that holds the column indices of the corresponding a_{ij} elements stored in $\tilde{A}$.
- An integer array R of length $M + 1$. For each row i in A, $R(i)$ contains the position in the $\tilde{A}$ and J arrays of the first occurrence of a nonzero element for that row. If a row is zero, $R(i)$ contains the position of the first occurrence of a nonzero element in the next nonzero row. This implies that $\tilde{A}$ and the corresponding J must be ordered by row. The last element, $R(M + 1)$ holds the location of a fictitious row $M + 1$. The fictitious row $M + 1$ is necessary to simplify the matrix-vector multiplication algorithm.

The sample matrix in CSR format is shown in figure 1.3. Notice how the zero rows of the matrix are represented in CSR format. In CRS, an element of R that corresponds

$$
\begin{aligned}
R = &\quad \begin{bmatrix} 1 & 3 & 4 & 4 & 7 & 7 & 9 & 9 & 10 \end{bmatrix} \\
J = &\quad \begin{bmatrix} 1 & 5 & 3 & 2 & 7 & 8 & 5 & 7 & 8 \end{bmatrix} \\
\tilde{A} = &\quad \begin{bmatrix} 1.7 & 6.5 & 2.0 & 6.2 & 0.5 & 5.3 & 7.8 & 6.7 & 0.3 \end{bmatrix}
\end{aligned}
$$

Figure 1.3: Sample Matrix in CSR Format

to a zero row is equal to the entry that corresponds to the first nonzero row. For example, the third row of our example matrix is zero but the fourth row is not. So $R(3)$ is equal to 4. We also have $R(4)$ equal to 4 because the first nonzero entry of the next nonzero row is stored in position 4 of $\tilde{A}$ and J.

Let $A = (a_{ij}) \in \mathbb{R}^{M \times N}$ be the real sparse matrix with nnZ nonzero entries to be represented in CSR format. Let the triple consisting of the three arrays $(\tilde{A}, J, R)$ be the CSR representation of A. Let $\tilde{A}(k) = \tilde{a}_k$ and $J(k)$ be the k^{th} entries in $\tilde{A}$ and J. Let $R(i)$ be the i^{th} entry in R and $R(i+1)$ be the next entry. Then the set of all k for which

$$R(i) \leq k < R(i+1)$$

is the range of k for which $\tilde{a}_k$ is in the i^{th} row. Given that $\tilde{A}$ and J are ordered by the rows of A, all elements k that obey the relation above, are contiguously represented in $\tilde{A}$ and J between positions $R(i)$ and $R(i+1)-1$ in their respective arrays and belong to row i. So for row i we can replace j by $J(k)$, and a_{ij} by $\tilde{a}_k$ for $k = R(i) : R(i+1)-1$ in equation (1.3.1) to obtain

$$y_i = \sum_{k=R(i)}^{R(i+1)-1} \tilde{a}_k x_{J(k)}, \quad \forall i = 1 : M \tag{1.5.1}$$

Using C notation, let $A[k]$ and $J[k]$ be the k^{th} elements of $\tilde{A}$ and J respectively. Let $X[j]$ be the j^{th} element of X, and $Y[i]$ be the i^{th} element of Y. The algorithm that corresponds to equation (1.5.1) is shown next.

Algorithm 1.3. *Compressed Sparse Row Multiplication Algorithm*

```
void csr_SparseMul( int M, double A[], int J[], int R[],
                    double X[], double Y[] )
{
  int i,k;
  for(i=0;i<M;i++) {
```

```
    for(k=R[i];k<R[i+1];k++) {
      Y[i] += A[k] * X[J[k]];
    }
  }
}
```

The minimum storage required by the CSR representation on an MxN double precision sparse matrix with nnZ nonzero elements is

$$St_{csr} = sizeof\,(int) \times (M + nnZ) + sizeof\,(double) * nnZ \tag{1.5.2}$$

or, $St_{csr} = 4 \times M + 12 \times nnZ$ bytes. The storage savings of CSR over COO are $SS_{csr} = St_{mm} - St_{csr}$

$$SS_{csr} = sizeof\,(int) \times (nnZ - M) \tag{1.5.3}$$

or, $SS_{csr} = 4 \times (nnZ - M)$ bytes.

Algorithm 1.3 addresses X indirectly, while Y and A have contiguous storage throughout. The reason for this is that the CSR representation requires that the matrix data be ordered by rows. So sorted matrix data can have a positive effect on efficiency. Matrix re-organization is a way to increase efficiency of matrix algorithms. Sorting can be one of them. Notice that the CSR multiplication algorithm has two nested loops while the COO one has only one loop. To calculate the relative cost of this algorithm we re-write it as we did in the COO format case. The outer loop runs M times but it is tested $M + 1$ times. To calculate how many times the inner loop runs, we recognize that in the i^{th} iteration of the outer loop, all the elements of the i^{th} row are processed. If we add the number of elements of each row over all M rows of the matrix we obtain nnZ. Let nR_i be the number of nonzero elements in the i^{th} row. Let nK be the total number of inner loop iterations for the i_{th} loop. Then $nK = \sum_{i=1}^{M} nR_i = nnZ$. Therefore the inner loop runs nnZ times.

Let c be the upper bound of the time required to perform the loop tests $k < M$ and $k < K(i + 1)$, the assignment instructions, and the integer additions. Let t be the upper bound of the time required by a floating point multiplication or addition. Then the loop time is bounded above by $\ell \leq (2nnZ)t + (3nnZ + 3M + 3)c$ as seen below

Loop	Cost
$i \leftarrow 0$	c
while $i < M$ **do**	$(M+1)\,c$
$\quad k \leftarrow K(i)$	$(M)\,c$
$\quad$ **while** $k < K(i+1)$ **do**	$(nnZ+1)\,c$
$\quad\quad tmp1 \leftarrow J(k)$	$(nnZ)\,c$
$\quad\quad tmp2 \leftarrow A(k) \times X(tmp1)$	$(nnZ)\,t$
$\quad\quad Y(i) \leftarrow Y(i) + tmp2$	$(nnZ)\,t$
$\quad\quad k \leftarrow k+1$	$(nnZ)\,c$
$\quad i \leftarrow i+1$	$(M)\,c$
	$(2nnZ)t + (3nnZ + 3M + 3)c$

1.6 Reduced Index Sparse Representation

This thesis introduces the new Reduced Index Sparse (RIS) representation that consists of a single array of ordered pairs (j, v) for every nonzero value in the matrix. A nonzero matrix element is expressed by a pair (j, v) where j is a positive integer that indicates its column index and v is the nonzero value that can be an integer, a real or a complex number. Only real valued matrices are discussed here. To make it possible to use pairs instead of triples the new storage scheme uses row markers. A row marker $(-r, 0)$ is a pair consisting of the negative integer $-r$ that indicates the row r and a 0 value that is ignored. All pairs (j, v) following the row marker $(-r, 0)$ belong to row r. The last element of the matrix is the pair $(0, 0)$ which is called the end-of-matrix pair. This single array representation of a sparse matrix introduced here contrasts with the COO and CSR representations that require three arrays to process the matrix. The savings in storage are paid back with extra processing to extract the row information.

Definition 1.1. *$\mathbb{P}$ is the set of pairs (j, v) where $j \in \mathbb{Z}^+$ and $v \in \mathbb{R}$. $\mathbb{P} = \mathbb{Z}^+ \times \mathbb{R}$. Each element of $\mathbb{P}$ has a "j" part that is an integer and a "v" part that is a real.*

Definition 1.2. *$\mathbb{P}^L$ is the vector space of all vectors of length L consisting of (j, v) pairs where $j \in \mathbb{Z}^+$ and $v \in \mathbb{R}$.*

```
%%Reduced Index matrix coordinate real ris
%  M  N   nnZ
   8  8   9
%  j  v
  -1  0.0
   1  1.7
   5  6.5
  -2  0.0
   3  2.0
  -4  0.0
   2  6.2
   7  0.5
   8  5.3
  -6  0.0
   5  7.8
   7  6.7
  -8  0.0
   8  0.3
```

Figure 1.4: Sample Matrix in RIS Format

Definition 1.3. *Let $V \in \mathbb{P}^L$. Let $\alpha(k) = (j_k, v_k)$ be the k^{th} element of V. Then the j part of $\alpha(k)$ is j_k and is expressed as $\alpha(k).j$ and the v part of $\alpha(k)$ is v_k and is expressed as $\alpha(k).v$*

This definition is adopted from some common notation from C/C++. Let $A = (a_{ij}) \in \mathbb{R}^{M \times N}$ be a sparse matrix with nnZ nonzero entries that we want to represent in RIS format. The RIS representation of A consists of:

- A single one dimensional array $P = (P(k)) = (p_k) = (j_k, v_k) \in \mathbb{P}^L$ with length $L \geq nnZ + M$ that contains (j, v) pairs corresponding to the nonzero values a_{ij} of A in row order together with a set of embedded row markers $(-r, 0)$ according to the rules given in figure 1.5

Let $Y = (y_i) = (Y(i)) \in \mathbb{R}^M$ be the product vector. Let $X = (x_j) = (X(j)) \in \mathbb{R}^N$ be the right hand side vector. We want to express $Y = AX$ in terms of P. To calculate Y with P, it is required to test $P(k).j$ for every $P(k) \in P$. If $P(k).j$ is 0, $P(k)$ is the end-of-matrix element and the loop terminates. If $P(k).j$ is negative, $P(k)$ is a row marker and the algorithm begins processing the new row $i = -P(k).j$. If $P(k).j$ is positive, $P(k)$ is a data element and is used to process $Y(i)$. Then

$$y_i = \begin{cases} \sum_{k=0}^{P(k).j \neq 0} P(k).v \times x_{P(k).j} & P(k).j > 0; \\ \\ i = -P(k).j, & P(k).j < 0. \end{cases} \tag{1.6.1}$$

Rules 8 and 9 in figure 1.5 state that the representation does not require the rows to be ordered and that rows can be segmented. For increased efficiency of the RIS multiplication algorithm however, it is advantageous to restrict the rows to be unsegmented and that they be ordered. Ordered, unsegmented rows imply more efficient access to the elements of Y. Using C style syntax, the algorithm that corresponds to equation (1.6.1) is shown next.

Algorithm 1.4. *RIS Multiplication Algorithm*

```
void risMul( dSparS_t *P, double *X, double *Y )
{
  int i, k;
  for( k=0; 0 != P[k].j; k++ ) {
    if(P[k].j < 0)
      i = -1-P[k].j;
    else {
      Y[i] += P[k].v * X[P[k].j-1];
    }
  }
}
```

1. Only nonempty rows are shown.
2. Row indices are negative.
3. Column indices are positive.
4. The first entry must be a row marker.
5. The end-of-matrix $(0, 0)$ is the last element.
6. Rows do not have to be ordered.
7. Rows can be segmented, that is parts of a row can appear in different places.

Figure 1.5: RIS Representation Rules

The minimum RIS storage required for an MxN sparse matrix with *nnZ* nonzero elements and no empty rows is

$$St_{ris} \;=\; sizeof\,(int) * (M + nnZ) + sizeof\,(double) * (M + nnZ) \qquad (1.6.2)$$

or, $St_{ris} = 12 * (M + nnZ)$ bytes.

The storage savings of RIS over COO are $SS_{ris} = St_{mm} - St_{ris}$ or,

$$SS_{ris} \;=\; sizeof\,(int) * (nnZ - M) - sizeof\,(double) * M \qquad (1.6.3)$$

or, $SS_{ris} = 4 * nnZ - 12 * M$ bytes.

The minimum required RIS storage is less if the matrix has empty rows because the RIS representation does not mark empty rows. Matrix operations with the RIS format do not require explicit knowledge of empty rows.

To calculate the relative cost of this algorithm we re-write it as we did in the previous cases. Let c be the upper bound of the time required to perform loop and if tests, jumps, assignments, and integer additions. Let t be the upper bound of the time required to perform floating point multiplications and additions. Then the loop time for this algorithm is bounded above by $\ell \leq (2nnZ)t + (5nnZ + 2M + 3)c$ as seen below

	Loop	**Cost**
	$k \leftarrow 0$	c
	while $P(k).j \neq 0$ **do**	$(nnZ + M + 1)\,c$
	if $P(k).j < 0$ **then**	$(nnZ + M + 1)\,c$
	$i \leftarrow -P(k).j - 1$	$(M)\,c$
	goto $L1$	$(M)\,c$
	else	0
	$tmp1 \leftarrow P(k).j - 1$	$(nnZ)\,c$
	$tmp2 \leftarrow P(k).v \times X(tmp1)$	$(nnZ)\,t$
	$Y(i) \leftarrow Y(i) + tmp2$	$(nnZ)\,t$
$L1$	**continue**	$(nnZ)\,c$
	$k \leftarrow k + 1$	$(nnZ)\,c$
		$(2nnZ)t + (5nnZ + 2M + 3)c$

1.7 Reduced Index CSR Representation

The CSR representation can be modified to have only two arrays. One such array is R as defined for CSR and the other one is P as defined for RIS. Unlike RIS, the modified CSR representation does not require row markers. Since this representation is a hybrid between RIS and CSR, it is called the Reduced Index Compressed Sparse Row, or RCSR representation.

$$R = \begin{bmatrix} 1 & 3 & 4 & 4 & 7 & 7 & 9 & 9 & 10 \end{bmatrix}$$
$$P = \begin{bmatrix} (1,1.7) & (5,6.5) & (3,2.0) & (2,6.2) & (7,0.5) \\ (8,5.3) & (5,7.8) & (7,6.7) & (8,0.3) & \end{bmatrix}$$

Figure 1.6: Sample Matrix in RCSR Format

Let $A = (a_{ij}) \in \mathbb{R}^{M \times N}$ be a real sparse matrix with nnZ nonzero entries that we want to represent in RCSR format. The RCSR representation of A consists of:

- A one dimensional array $P = (P(k)) = (p_k) = (j_k, v_k) \in \mathbb{P}^{nnZ}$ that contains (j, v) pairs corresponding to the nonzero values a_{ij} of A in row order.
- An integer array R of length $M + 1$. For each row i in A, $R(i)$ contains the position in the P array of the first occurrence of a nonzero element for that row. If a row is zero, $R(i)$ contains the position of the first occurrence of a nonzero element in the next nonzero row. This implies that P must be ordered by row. The last element, $R(M + 1)$ holds the location of a fictitious row $M + 1$. The fictitious row $M + 1$ is necessary to simplify the matrix-vector multiplication algorithm.

Let $P(k) = p_k$ be the k^{th} entry in P. Let $R(i)$ be the i^{th} entry in R and $R(i + 1)$ be the next entry. Then the set of all k for which

$$R(i) \leq k < R(i + 1)$$

is the range of k for which p_k is in the i^{th} row. Given that P is ordered by the rows of A, all elements k that obey the relation above, are contiguously represented in P between positions $R(i)$ and $R(i + 1) - 1$ and belong to row i. So for row i we can replace j by $P(k).j$, and a_{ij} by $P(k).v$ for $k = R(i) : R(i + 1) - 1$ in equation (1.3.1) to obtain

$$y_i = \sum_{k=R(i)}^{R(i+1)-1} P(k).v \times x_{P(k).j}, \quad \forall i = 1:M \tag{1.7.1}$$

Using C style notation, let $P[k]$ be the k^{th} element of P. Let $Y[i]$ be the i^{th} element of Y. Let $X[j]$ be the j^{th} element of X. The algorithm that corresponds to equation (1.7.1) is shown next.

Algorithm 1.5. *Reduced Index CSR Multiplication Algorithm*

```
void rcsr_SparseMul( int M, dSparS_t *P, int *K,
                     double *X, double *Y )
{
  int i,k;
  for(i=0;i<M;i++) {
    for(k=R[i];k<R[i+1];k++) {
      Y[i] += P[k].v * X[P[k].j];
    }
  }
}
```

In this algorithm the index of X becomes available while fetching $P[k]$. One fetch to $P[k]$ suffices to obtain $P[k].v$ and $P[k].j$. This is the major advantage over CSR.

To obtain the relative cost of the algorithm let c be the upper bound of the time required to perform the loop tests $k < M$ and $k < K(i+1)$, the assignment instructions, and the integer additions. Let t be the upper bound of the time required by floating point multiplications and additions. We calculate the relative cost of algorithm 1.5 by rewriting the algorithm below as was done in the previous cases. The loop time is found to be bounded above by $\ell \leq (2nnZ)t + (3nnZ + 3M + 3)c$ as seen below.

Loop	Cost
$i \leftarrow 0$	c
while $i < M$ **do**	$(M+1)\,c$
$\quad k \leftarrow K(i)$	$(M)\,c$
$\quad$ **while** $k < K(i+1)$ **do**	$(nnZ+1)\,c$
$\quad\quad tmp1 \leftarrow J(k)$	$(nnZ)\,c$
$\quad\quad tmp2 \leftarrow P(k) \times X(tmp1)$	$(nnZ)\,t$
$\quad\quad Y(i) \leftarrow Y(i) + tmp2$	$(nnZ)\,t$
$\quad\quad k \leftarrow k+1$	$(nnZ)\,c$
$\quad i \leftarrow i+1$	$(M)\,c$
	————————————————
	$(2nnZ)t + (3nnZ + 3M + 3)c$

The RCSR and CSR algorithms come up with the same cost analysis. This is expected because the loops are the same. But this analysis is based on estimates and upper bounds. The actual value of c (an upper bound) as defined in both algorithms could be different for each. Nevertheless, the two algorithms have the same cost. Having both $P(k).j$ and $P(k).v$ in a single fetch can be exploited for optimization, however.

1.8 Example

The RIS format for this matrix can expressed in a human readable format (figure 1.8), where each line starts with a row marker followed by one or more column indexed entries. An empty row is identified by the absence of the corresponding row marker. For example, the sample matrix in figure 1.8 does not have row 3 because there is no row marker {3,0.0}. The end-of-matrix marker {0,0.0} is the last entry. Our sample matrix (figure 1.1) is depicted graphically in figure 1.7. The graphic was obtained with *matview* [2]. It is the same matrix in figure 1.2 (COO storage) and figure 1.3 (CSR storage).

The coded values of figure 1.7 range from light Grey (minimum) to dark Grey (maximum) as seen in figure 1.9. Graphic representation is important to make it easy to visualize the actual matrix.

[2] matview can be found at http://www.csm.ornl.gov/ kohl/MatView/

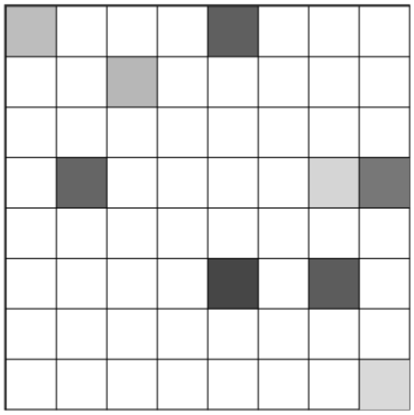

Figure 1.7: Sample Matrix

{-1,0} {1,1.7} {5,6.5}
{-2,0} {3,2.0}
{-4,0} {2,6.2} {7,0.5} {8,5.3}
{-6,0} {5,7.8} {7,6.7}
{-8,0} {8,0.3}
{0,0}

Figure 1.8: Human readable RIS

Figure 1.9: Grey Scale

1.9 Comparison

A sparse representation can have an impact on the efficiency of sparse matrix algorithms. An analysis of these algorithms as presented here shows the relative cost of each. Of the algorithms presented in this chapter the Reduced Index CSR seems to be the best one. Using the COO algorithm as reference, and assuming that c (as defined in previous sections) is the same for all three algorithms, we find that CSR and RCSR improve the performance by $(nnZ - 3M - 3)c$, and the RIS worsens the performance by $(nnZ + 2M + 1)c$.

Algorithm efficiency has many variables and aspects to be considered. The best way to find out is by actual time measurements using the same data on all algorithms. This type of comparison will be presented in a later chapter.

The RIS and RCSR storage schemes guarantee that all the information necessary to describe the next nonzero matrix element is available with a single access to the memory. Current computer technology organizes memory in a hierarchy of levels with conflicting properties [3]. Closest to the CPU and fastest access memory is the set of CPU registers. This memory is very limited in size. Furthest from the CPU, disk memory is very large but extremely slow. Cache memory is an intermediate stage of memory that ranks next in speed to the registers. This memory is also limited but on the order of a few Kilobytes. When accessing the memory for the next nonzero matrix element, cache memory is where the operating system first looks. If it does not find it there it goes to the next level in the hierarchy with a much longer access time. This is known as a cache miss. With a single array to access, RIS and RCSR offer a higher likelihood of finding the data in cache memory during a read. Cache misses will be fewer accessing a single array than accessing three arrays because having to access three arrays to obtain the information of one nonzero matrix element requires three reads. This increases the probability of getting a cache miss.

Chapter 2

Sparse Matrix Multiplication in Parallel

The proving ground of the RIS representation is the parallel processor environment. One goal is to arrive at scalable, accurate implementation with excellent load balance and efficiency. Another goal is to compare the sparse matrix multiplications of the different representations with numerical results.

In order to improve efficiency, three different partition schemes are implemented and tested with large matrices to obtain good numerical results. The matrices found at the Matrix Market website are excellent sparse matrices for testing, because they represent real world matrices. But they are not big enough for parallel processor testing. One of the most expensive parts of a parallel implementation is the communication between processors. The communication cost is fairly independent from the size of the data to be crunched. So it is important to increase the amount of computation to offset the cost of communication. On one hand, one could ignore the communication timing and say that only computation timing is to be compared. But that is not realistic because message passing between processors is an integral part of the algorithm. With that in mind, the choice is to produce very large matrices for testing the resulting implementations. In order to generate such large matrices a program named genMatrix was created to generate matrices of a target number of nonzero elements, with a target density and a specific distribution. This program will be described later. The following sections describe three partition schemes, the column balanced partition, the balanced column cyclic partition, and the balanced block cyclic partition.

2.1 The Column Balanced Partition

Let nP be the number of processors to execute a sparse-matrix/dense-vector multiplication of a sparse matrix $A \in \mathbb{R}^{M \times N}$ with nnZ nonzero elements by a vector $X \in \mathbb{R}^N$. The question is how to balance the processing load among the nP processors. Assume that processing an equal amount of nonzero elements in each processor implies that each processor performs the same number of operations to process its share of the data. With that assumption we would like to partition the data into nP equal parts.

RIS representation of a sparse matrix consists of a single array of (j, v) pairs of length $aSize = nnZ + M$. Exploiting that fact, it is safe to say that dividing the the array into nP number of sub-arrays of size roughly equal to $targetSize = aSize/nP$ also divides the matrix into nP parts of roughly the same number of elements. The reason for using the word roughly is because the row markers are embedded in the arrays and sub-arrays. With nP being the number of processors, this would be one way to balance the load among them. But the question arises whether to respect row boundaries or not. Respecting row boundaries implies that in some cases the load will be thrown too far out of balance to consider this a good load balancing scheme. On the other hand, ignoring row boundaries could leave part of the elements of one row in one processor and the other part in another. This would require additional post processing message passing where communication is expensive, offsetting any efficiency gained by the balancing scheme. This partition scheme is called here the *Row Balanced* partition.

The RIS representation can be used to implement a variable column size decomposition of the matrix where each process would get close to equal number of nonzero elements. The first step is to count the elements of each column into an array of size N. Let us call this array *colCount*. Suppose that the matrix is to be partitioned into nCD column divisions, and the total number of nonzero elements in the matrix is nnZ. Then the target size of the sub-matrices is $targetSize = nnZ/nCD$. The column partition is performed by finding nCD column divisions of close to $targetSize$ elements each. For each column division cD find the number of columns $cnc[cD]$ by

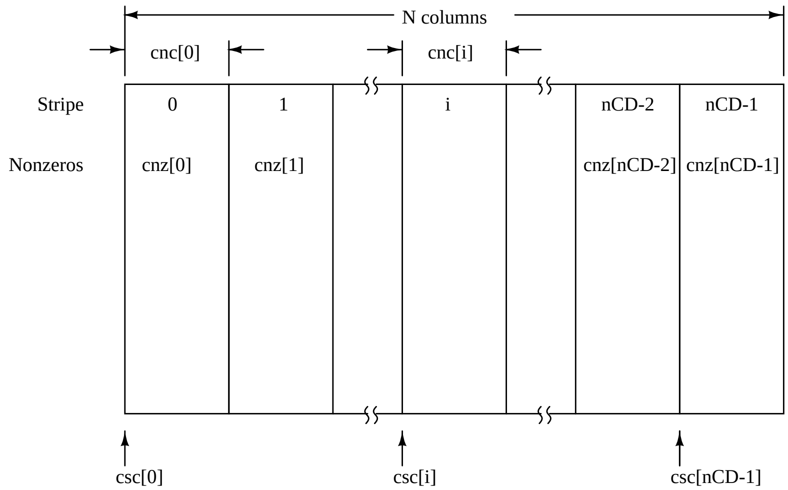

Figure 2.1: Column Balanced Partition

finding the

$$sum = \sum_{col=C_0}^{min(|sum-targetSize|)} colCount[col]$$

which is the summation of the column counts in the $colCount$ array until the sum is closest to $targetSize$. Suppose that C_L is the column number in the above summation that $|sum - targetSize|$ is minimum for the column division cD considered, then the number of columns in column division cD is $C_L - C_0 + 1$. This value is stored in the cnc array which stands for column division number of columns. So $cnc[cD] = C_L - C_0 + 1$. Two more arrays are populated in the column partition, the cnz and the cns arrays. The cnz array stores the actual number of nonzero elements in each column division, and the cns array contains the starting column of each column division. In the column partition, columns are not broken. Each column is part of one and only one column division. All theses arrays are of size nCD.

2.2 The Balanced Column Cyclic Partition

The column cyclic partition uses the Column Balanced Partition algorithm described in the previous section. Then it distributes the column divisions among a number nP of processors. One requirement for this partition is that $nCD \geq nP$ as there should be a minimum of one cycle. It is desirable that nCD be an integral multiple of nP

to achieve a balanced distribution. So the column cyclic partition is really a column balanced partition that cyclically assigns column divisions to the processors. Column division *0* is assigned to process P_0, column division *1* is assigned to process P_1, and so on. For an arbitrary process P_x, and an arbitrary column division cD,

$$P_x \longleftarrow cD \iff P_x == (cD \mod nP)$$

where the subscript x in P_x represents the rank of the process ($0 \leq x < nP$). In the extreme case where the number of processors is one, all the column divisions are assigned to P_0. When the number of column divisions is the same as the number of processors, each column division is assigned to exactly one process.

2.3 The Balanced Block Cyclic Partition

This partition occurs in two steps, column partition and then row partition. The balanced block cyclic partition uses a column cyclic partition, but the assignment of blocks to processes is different. The row partition consists of dividing the nonzero elements of each column division into nRD blocks of nearly the same number of elements where nRD is the number of row divisions. In section 2.1 there is a description of how column divisions can be partitioned into nRD blocks ignoring row boundaries. Ignoring row boundaries is a good choice in this case because in this partition, the range of rows that will be assigned to a particular processor P_x varies from one column cycle to another. To avoid complicated partitioning of Y, each processor will generate a Y_{local} that is the same size of Y (namely, $|Y_{local}| = |Y|$). Each processor's Y_{local} will contribute partial results in many of the rows. The final result will be added into processor P_0 using MPI_Reduce.

The blocks are then distributed among the different processors in the same manner they are in the dense (traditional) block cyclic algorithm. The big difference between the balanced and the traditional block cyclic distributions is that the row partitioning is implicit (or predefined) for the traditional case, while it is calculated for the balanced case. As in the traditional case, there is a column cycle Pc, and there is a row cycle Pr. In the traditional case the column cycle is a specific number of columns, and the row cycle is a specific number of rows. In the balanced block cyclic partition case, the column cycle Pc is a specific number of column divisions, and the row cycle Pr is defined by a specific number of row divisions. The number of processors needed for a balanced block cyclic partition is $nP = Pc \times Pr$. A complete process grid is a contiguous set of blocks each of which is assigned to one of the nP processors. Numbering the processor assignment of the blocks from left to right and top to bottom in a complete process grid, a block cyclic assignment is mapped as follows:

	0	1	...	Pc-1
0	0	1	...	Pc-1
1	Pc	1+Pc	...	(Pc-1)+Pc
⋮	⋮	⋮	⋱	⋮
(Pr-1)	(Pr-1)*Pc	1+(Pr-1)*Pc	...	(Pc-1)+(Pr-1)*Pc

The previous mapping is a more abstract description of the process grid defined in the ScalaPACK User's Guide [17] section 4.1.1. The single top row is the column process coordinate heading and shows the column process coordinates pc. The single left column is the row process coordinate heading and shows the row process coordinates pr. The process coordinate headings are optional, they can be excluded from the assignment map.

As an example, with $Pc = 4$ and $Pr = 4$ the assignment map with process coordinate headings in a complete process grid is

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

With the same values of $Pc = 4$ and $Pr = 4$ and with the number of column divisions $nCD = 8$ and the number of row divisions $nRD = 8$ the assignment map without process coordinate headings is

0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
12	13	14	15	12	13	14	15
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
12	13	14	15	12	13	14	15

Definition 2.1. *Let $x \in \mathbb{R}$ be an arbitrary real. Let $Ceil \in \mathbb{Z}$ be the smallest integer such that $Ceil \geq x$. Then $Ceil$ can be expressed by the notation*

$$Ceil = \lceil x \rceil$$

In this mapping, the column divisions are grouped into column division groups, each with Pc column divisions or less. The row divisions are also grouped into row division groups, each with Pr row divisions or less. The number of column division groups nCG, and the number of row division groups nRG are given by

$$nCG = \left\lceil \frac{nCD}{Pc} \right\rceil$$

$$nRG = \left\lceil \frac{nRD}{Pr} \right\rceil$$

In the previous example, with $nCG = 2$, $nRG = 2$, the number of process grids is $nPG = 4$. All four of the process grids are complete, that is, all the processors get a block assignment. This could have been predicted as

$$nPG = nCG \times nRG$$

To expand these concepts let us use another example with the same values of $Pc = 4$ and $Pr = 4$ but having $nCD = 10$ and $nRD = 10$. The assignment map with process coordinate headings is

	0	1	2	3	0	1	2	3	0	1
0	0	1	2	3	0	1	2	3	0	1
1	4	5	6	7	4	5	6	7	4	5
2	8	9	10	11	8	9	10	11	8	9
3	12	13	14	15	12	13	14	15	12	13
0	0	1	2	3	0	1	2	3	0	1
1	4	5	6	7	4	5	6	7	4	5
2	8	9	10	11	8	9	10	11	8	9
3	12	13	14	15	12	13	14	15	12	13
0	0	1	2	3	0	1	2	3	0	1
1	4	5	6	7	4	5	6	7	4	5

In this example, the number of column division groups is $nCG = \lceil \frac{10}{4} \rceil = 3$, the number of row groups is $nRG = \lceil \frac{10}{4} \rceil = 3$, and the number of process grids is $nPG = 9$ of which four are complete and five are incomplete.

The process in a process grid can be derived from the process coordinates pr and pc. But the process coordinates can be determined from the block parameters as well. For an arbitrary column division cD, the column process coordinate pc is the column position of the column division in the process grid. For an arbitrary row division rD, the row process coordinate pr is the row position of the row division in the process grid. The column process coordinate pc and the row process coordinate pr are determined by

$$pc = cD \mod Pc$$

and

$$pr = rD \mod Pr.$$

And for an arbitrary process P_x, an arbitrary row division rD in an arbitrary column division cD

$$P_x \longleftarrow (rD, cD) \iff P_x == pc + pr \times Pc \tag{2.3.1}$$

2.4 Sample Partitions

The partition implementation achieved in support of this thesis happens to include all the possible partitions discussed in this chapter. The parameters given to partition a matrix are the number of column divisions nCD, the number of row divisions nRD, the column cycle Pc, and the row cycle Pr. Depending on how these parameters are combined, the partition software can generate any partition desired. Using a generated 100×100 matrix with about 500 nonzero elements, several sample partitions were produced using the program *partMat*.

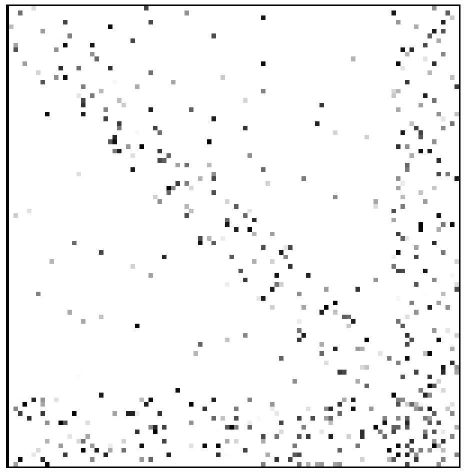

Figure 2.2: Unpartitioned Matrix

The example matrix is not uniformly distributed as we want to observe how the partition algorithms work on odd distributions. Concentrations of nonzero elements to the right and the bottom of the matrix are purposely included to create a visible effect of the partitioning. This test matrix matrix101.mtx has the following specifications.

- Values range from -1 to +1.
- The matrix size is 100×100.
- It has 506 nonzero elements.
- 30% of the nonzero elements are concentrated along the diagonal with a density of 8%.
- 25% of the nonzero elements are concentrated along the right edge with a density of 9%.
- 25% of the nonzero elements are concentrated along the bottom edge with a density of 9%.
- 20% of the nonzero elements are spread out.

The first example (figure 2.3) is a row balanced partition that was generated by issuing the command "*partMat* mat101 1 2 1 2". That is a partition with one column division ($nCD = 1$) and one column cycle ($Pc = 1$), two row divisions ($nRD = 2$) and two

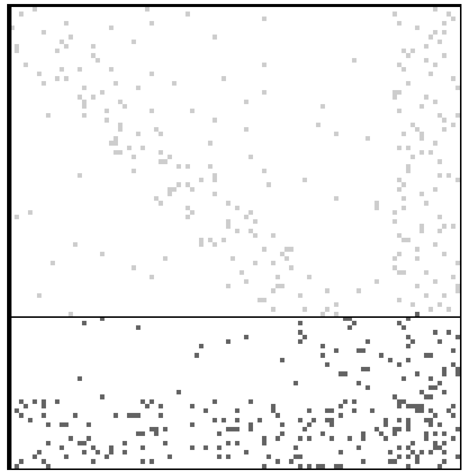

Figure 2.3: Row Balanced Partition

row cycles ($Pr = 2$). A gray scale map of the partition was produced using matview [1]. In order to highlight the partition without the use of colors, each partition block has its own level of Grey according to the gray scale of figure 1.9. Each block of the partition has roughly the same number of nonzero elements. But because of density variation of the sparsity pattern in a sparse matrix, the area that comprises the block looks different. The lower the density of a partition block, the larger the area of the block will appear in the graphic. The top block (block 0) spans from row 1 to row 67. Block 1 ranges from row 67 to row 100. Notice that one element in row 67 belongs to block 1. All the other ones belong to block 0. Row 67 is shared between the two blocks. In the graphical representation generated by *matview*, rows are counted from top to bottom starting at 1, and columns are counted from left to right starting at 1.

The second example (figure 2.4) is a column balanced partition that was generated by issuing the command "partMat mat101 2 1 2 1". That is a partition with two column divisions ($nCD = 2$) and two column cycles ($Pc = 2$), one row division ($nRD = 1$) and one row cycle ($Pr = 1$). In this partition the first 66 columns of matrix mat101.mtx is on block 0, and block 1 is the right side area of columns 67 through 100. Notice that there are no shared columns. Column partitions respect boundaries; row partitions do not.

[1] matview can be found at http://www.csm.ornl.gov/ kohl/MatView/

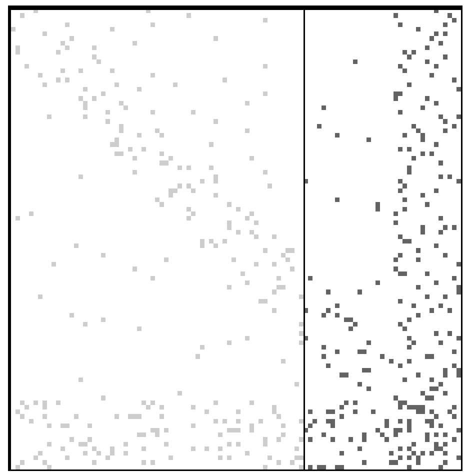

Figure 2.4: Column Balanced Partition

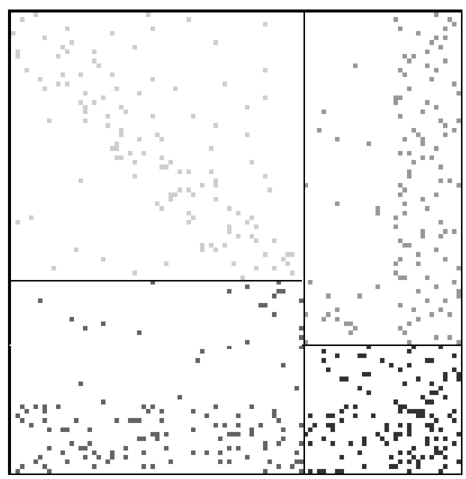

Figure 2.5: Balanced Block Cyclic Partition

The third example (figure 2.5) is a balanced block cyclic partition that was generated by issuing the command "partMat mat101 2 2 2 2". That is a partition with two column divisions ($nCD = 2$) and two column cycles ($Pc = 2$), two row divisions ($nRD = 2$) and two row cycles ($Pr = 2$). The left blocks (blocks 0 and 2) encompass columns 1 through 65. Block 0 ranges from row 1 through row 58. Block 2 goes from rows 59 to 100. The right side blocks (blocks 1 and 3), are divided at row 72. Block 1 comprises rows 72 and below, and block 3 rows 73 and above.

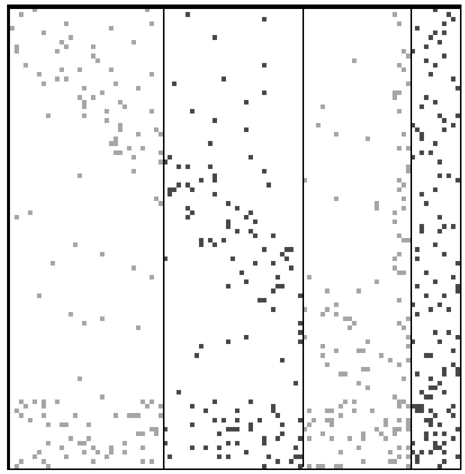

Figure 2.6: Balanced Column Cyclic Partition

The fourth example (figure 2.6) is a balanced column cyclic partition that was generated by issuing the command "partMat mat101 4 1 2 1". That is a partition with four column divisions ($nCD = 4$) and two column cycles ($Pc = 2$), one row division ($nRD = 1$) and one row cycle ($Pr = 1$). This partition has two column groups ($nCG = 2$). For the left column division group, block 0 spans columns 1 through 34, and block 1 spans columns 35 through 65. On the right column division group, block 0 spans columns 66 through 89, and block 1 ranges from column 90 to the last column. Observe that the areas become smaller at the right of the matrix where the density is higher.

The last example (figure 2.7) is a balanced row cyclic partition that was generated by issuing the command "partMat mat101 1 4 1 2". That is a partition with one column division ($nCD = 1$) and one column cycle ($Pc = 1$), four row divisions ($nRD = 4$) and two row cycles ($Pr = 2$). This partition has two row groups ($nRG = 2$). For the top row group, block 0 spans rows 1 through 35, and block 1 spans rows 35 through 67. On the bottom row group, block 0 spans rows 67 through 90, and block 1 ranges from column 90 to the last column. Observe that the areas become smaller at the bottom of the matrix where the density is higher. Also notice that rows 35, 67, and 90 are shared between blocks.

The program *partMat* that was used to produce the visual representations of the

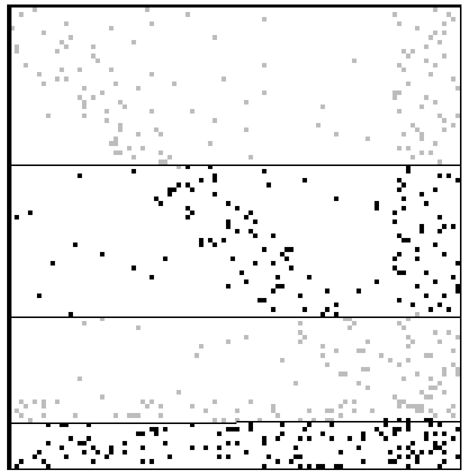

Figure 2.7: Balanced Row Cyclic Partition

different partitions uses the same algorithm for partitioning the data as the program *RIS_Block_Cyclic*. Furthermore, the partition occurs during matrix read. This fact could be a very nice feature if the program is capable of concurrently executing useful work during a disk read. Asynchronous I/O and multiple threads come to mind [16]. In such scenario, partitioning can be practically cost free because it can occur while the operating system acquires the data from disk storage. Such concurrency was not implemented in *RIS_Block_Cyclic* but is proposed as a future work item. As a matter of interest, and to compare the effect of a particular partition type on performance, timing measurements for the different types of partition are presented in a later chapter.

Chapter 3

RIS Block Cyclic Implementation

3.1 Introduction

The (j, v) pairs are populated into variables of type dSparS_t (see figure 3.1) for matrices with data of type double. Integer matrices use type iSparS_t, and complex matrices use cSparS_t. As stated before, only real valued matrices are discussed here. But the other types are shown to indicate the way to structure other types of matrices.

```
typedef struct {
  int    j;
  double v;
} dSparS_t;
```

Figure 3.1: double RIS structure

```
typedef struct {
  double re;
  double im;
} cmplx_t;
```

Figure 3.2: Complex value structure

```
typedef struct {
  int j;
  int v;
} iSparS_t;
```

Figure 3.3: int RIS structure

```
typedef struct {
  int     j;
  cmplx_t v;
} cSparS_t;
```

Figure 3.4: complex RIS structure

The implementation of the RIS block cyclic concept involves several programs. A program called RIS_Block_Cyclic implements the block cyclic algorithm in a parallel computer using MPI. To be able to partition the input matrix while reading it is necessary to order the matrix by row, and by column within the rows. A program called $prepMat$ does that and saves the matrix into a new file with a "_o" appended to its name. For instance, if the input matrix is foo.mtx, the ordered matrix is foo_o.mtx. This step is required for matrices that are downloaded from Matrix Market (http://math.nist.gov/MatrixMarket/) or some other source. Program *prepMat* adds a column count array to the header section of the matrix. This array is expected by the program *RIS_Block_Cyclic*.

A program called *genMatrix* generates matrices large enough to get decent time measurements of the implementation. The matrices produced by *genMatrix* are specified to have a target number of nonzero elements with a specific target density, and a specific sparsity pattern. Files generated by *genMatrix* are not required to be run through *prepMat* because they are fully ordered, and the column count array in the header section is included. A program called *ris_Val* independently validates *RIS_Block_Cyclic* via a FORTRAN program called *ftest.f* that uses the SPARSKIT library libskit.a. This chapter will present the principal program of the implementation *RIS_Block_Cyclic*.

The validation software *ris_Val* is discussed in section 3.7. The matrix generation software *genMatrix* is discussed in chapter 4.

3.2 Function Main

The function *main* performs four tasks amid necessary calls to MPI. The four tasks are reading the matrix from disk, broadcasting the X vector from the root processor, performing the local multiplication, and assembling the results back in the root node using *MPI_Reduce*. The main parts of the main block are presented in the section 3.2. The function calls (including the function *main*) are simplified to avoid clutter. The four tasks mentioned above are in boldface. Around each one of the tasks there are timers to measure the performance of that part of the implementation. In reality, each timer requires a separate compilation because of a restriction of the wall time function *MPI_Wtime* that can only be used once in each program. Two other tasks that occur in separate compilations are the determination of the time it takes to perform the test multiplication by an independent code. This test gathers the CSR multiplication time of a function called *amux_* from the SPARSKIT archive. The

```
int main(int argc, char *argv[])
{
        MPI_Init(&argc, &argv);                    // Start MPI
        MPI_Comm_size(&nP);                        // get # of processors
        MPI_Comm_rank(&my_rank);                   // get current process
        startTime6 = MPI_Wtime();
        readMatrix( );                             // Read Time
        stopTime6 = MPI_Wtime();
        MPI_Barrier();                             // Sync processes
        startTime1 = MPI_Wtime();
        // Preparation                             // Prep Time
        stopTime1 = MPI_Wtime();
        startTime2 = MPI_Wtime();
        MPI_Bcast(X,)                              // Broadcast Time
        stopTime2 = MPI_Wtime();
        startTime5 = MPI_Wtime();
        startTime3 = MPI_Wtime();
        risMul(A_local, X_local, Y_local );
        stopTime3 = MPI_Wtime();
        startTime4 = MPI_Wtime();
        MPI_Reduce(Y_local, Y);                    // Reduce Time
        stopTime4 = MPI_Wtime();
        stopTime5 = MPI_Wtime();
        startTime7 = MPI_Wtime();
        amux_(N,X,Ycsr,Ac,jC,RS);                  // CSR Time
        stopTime7 = MPI_Wtime();
        valmat_(X,Y,Ac,iR,jC);                     // Validation
        MPI_Finalize();                            // Finalize MPI
}
```

Figure 3.5: Function Main

other task is the accuracy validation also by SPARSKIT library functions.

The program uses several structures defined as types. The *matParams_t* type contains M, N, and nnZ. The *partitionParams_t* type holds the arguments given to the program nCD, nRD, Pc, and Pr. The *cycleData_t* type has pointers to four integer arrays (*cnc*, *csc*, *cnz*, and *Pcnt*) that are used to partition the matrix while reading it from disk. The *dSparS_t* is the type used to hold the (j, v) pairs in the RIS format. Some of these structures simplify the argument lists in some of the program's functions.

```
typedef struct {
        int M; // Number of rows
        int N; // Number of columns
        int nnZ; // Number of nonzeros
} matParams_t;

typedef struct {
        int nCD; // column divisions
        int nRD; // row divisions
        int Pc; // column cycles
        int Pr; // row cycles
} partitionParams_t;

typedef struct {
        int *cnc; // column division # of columns array
        int *csc; // column division starting column array
        int *cnz; // column division nonzeros array
        int *Pcnt; // count of elements read in cycle
} cycleData_t;
```

Besides measuring the elapsed time of the four main tasks of the implementation, it is desirable to know how long it takes to prepare the arrays for processing (memory allocation and initialization). So timing data was gathered as well for preparation. In this parallel implementation, each processor reads its own portion of the matrix. The function *readMatrix* partitions the matrix as it reads it. The *readMatrix* function will be discussed in section 3.3. Because the time of reading from disk is not very predictable, it is necessary to synchronize the end of matrix reading on all processors with a call to *MPI_Barrier*. All other MPI function calls form the basic MPI multi-processor program skeleton.

The matrix multiplication function *risMul* uses the arrays *A_local* and *X_local* to perform the local matrix-vector product. The result is returned in *Y_local*. The function *risMul* will be discussed in section 3.4. A special compilation that does not perform any timing measurements, calls *valmat_* to validate the results. This function belongs to a program called *ftest.f*. The program *ftest.f* will be discussed in section 3.7.

3.3 Function readMatrix

The function *readMatrix* performs three main tasks. The three tasks are reading the header, obtaining partition information, and reading the matrix data. While reading the matrix data, *readMatrix* only stores the data that belongs to the current process partition. The function shown below has been grossly simplified. Only the data read loop is shown in some detail. After reading a nonzero matrix element the current process cP is compared to the return value of a function called *whichProcess*. If the return value of *whichProcess* is equal to cP, then the data that was read is stored, otherwise it is ignored. The function *whichProcess* passes by reference a *newCol* value that is used to replace the column value if the matrix element is accepted. This will all be discussed in section 3.5.

The argument list of this function is fairly large. Most of the arguments are passed by reference back to the calling routine for its use. But because readMatrix allocates memory for all the arrays passed by reference, it is the responsibility of the calling routine to deallocate the memory from those variables when no longer needed. the complete parameter list is

- char *mmFile, // The matrix file name
- int cP, // The current process
- partitionParams_t *pp, // Program argument list
- double **Y_test, // A matrix to test the results - only during validation
- int **iR, // COO format I (rows) array - only during validation
- int **jC, // COO format J (cols) array - only during validation
- int **Ac, // COO format A (a_{ij}'s) array - only during validation
- matParams_t *mp, // Matrix dimension information - M, N, nnZ
- MM_typecode *matcode, // MM format typecode
- cycleData_t *cycl, // Partition support data
- dSparS_t **A_local, // The local A matrix in RIS format

The first three parameters are inputs, the rest are outputs. Y_test, iR, jC, and Ac are passed only in the validation compilation. When they are defined, the calling program must free the memory for these variables. *A_local*, *cycl*, *matcode*, and *mp* are either read from a file or created in *readMatrix*. A_local must be deallocated by the calling routine.

```
readMatrix( theMatrix,..., A_local )
{
    mat = fopen(theMatrix, "r");
    // Read header
    getColumnIndices();
    for(k=0:nnZ-1) {                                  // data read loop
        li=0;
        prevRow=0;
        // read iRow, iCol, and rVal
        if( whichProcess(icol,newCol) == cP ) {
            if(iRow ≠ prevRow) {
                A_local[li].j = -iRow;                // row marker
                A_local[li].v = 0.0;
                li++;
                prevRow = iRow;
            }
            A_local[li].j = newCol;
            A_local[li].v = rVal;
            li++;
        }
    }
    fclose(mat);
    return li;
}
```

While *readMatrix* is reading the header, it fills the *colCount* array found in the header section. This column count is placed in the header section of the matrix file by *prepMat*, a program that orders the matrix by row, then by column so that *RIS_Block_Cyclic* can partition the matrix during read. The program *prepMat* counts the nonzero elements for each matrix column. Knowing that this information is required by *RIS_Block_Cyclic*, it was decided to add this to the matrix file.

The function getColumnIndices fills the partition support data into the variable *cycl*. Variable *cycl*, which is passed by reference to and from *readMatrix*, is a structure that holds four arrays. One is the column division number of columns *cnc*. Another one is the column division starting column *cns*. Yet another one is the column division number of nonzero elements *cnz*. And the last one is *Pcnt* that is used by function *whichProcess* to determine the row cycle a nonzero belongs to. The whole structure is used by function *whichProcess* to partition the matrix.

3.4 Function risMul

This function is the implementation of algorithm 1.4. The inputs are A_local, the right hand side vector X. And the output is the product vector Y.

```
void risMul(dSparS_t *A_local, double *X, double *Y )
{
    int theRow, i, index;
    for( i=0; 0 != (index = A_local[i].j); i++) {
        if((index = A_local[i].j) < 0) {
            theRow = -(index + 1);
        }
        else {
            Y[theRow] += A_local[i].v * X[index - 1];
        }
    }
}
```

As has been pointed out previously, the *for* loop contains a test of the "j" part of the A_local element. Because the "j" part is tested again inside the loop to find row markers, it is assigned to a private variable called *index*. This is done to avoid accessing $A\_local[i].j$ again in the same iteration of the for loop. The *for* loop terminates when $index = 0$. Inside the *for* loop, if *index* is found to be negative, it is used to process a new row by the assignment $theRow \leftarrow -(index+1)$. Otherwise, $Y[theRow]$ is accumulated by the product $A\_local[i].v * X[index-1]$. A good algorithm for math intensive processing is a simple algorithm. Then, to take advantage of parallelism, it can be made complicated with loop unrolling, software pipelining, and other optimization schemes. Because of the internal test to identify row markers, this algorithm does not lend itself well for loop unrolling using current hardware technology. New technology is being developed at present to be able to optimize if loops (one example is the Intel Itanium processor). When such technology is readily available, this algorithm should be revisited for optimization.

3.5 Function whichProcess

```
int
whichProcess( int col,
              int *newCol,
              cycleData_t *cycl,
              partitionParams_t *pp )
{
    // Determine column division
    Pcnt[coldiv]++;      // coldiv's element cnt

    // Determine block within the coldiv

    // Determine process
    pc = coldiv % Pc;
    pr = block % Pr;
    cP = pc + pr * Pc;

    // Determine newCol

    return cP;     // return process
}
```

The function *whichProcess* performs four tasks. The four tasks are to find the column division that *col* belongs to, determine the row division within the column division found, and thereby determine which process *cP* owns *col*, and lastly to calculate a *newCol* value for *col*. The inputs to the function are *col*, *cycl* that was previously populated by the function getColumnIndices in main, and the argument list of the program in *pp*. The function returns the process that *col* belongs to, and passes by reference *newCol* back to the calling routine. The column division and block are necessary to determine the process *cP* and the *newCol*.

The function determines the column division by testing if *col* is between *coldivStart* and *coldivEnd* for each column division until found. To determine the block, the function increments the column count of the column division *Pcnt[coldiv]* every time the function is called. Using *Pcnt[coldiv]*, the function determines the block by testing if *Pcnt[coldiv]* is between blockStart and blockEnd for every block in the column division until found. Process determination is shown in the simplified function shown above, where *Pc* is the number of column cycles and *Pr* is the number of row cycles (both supplied as program arguments).

The determination of $newCol$ is due to a choice made on how to store X_local. This implementation populates X_local with only the elements of X that are necessary for the local matrix-vector product. In doing that, indices are readjusted for X_local. So the A_local column indices must be readjusted accordingly.

3.6 Function getColumnIndices

The function $getColumnIndices$ is only an accounting function. First, it establishes the target number of nonzero elements $eqLoad = nnZ/nCD$ that each column division should have. Then, for each column division, it accumulates the column counts until the accumulated sum is closest to the target $eqLoad$. Upon reaching this goal for each column division cD, the results are stored in $cnc[cD]$, and $cnz[cD]$. At this point $csc[cD+1]$ is also filled. The first element of csc is set before the loop starts as $csc[0] = colStart$. The function is void, it returns nothing.

The routine whichProcess is thus quite analogous to the ScalaPACK routine NUMROC which determines the number of rows and columns that each process receives in the A_local matrix.

3.7 Validation Software

Program $ftest.f$ (see appendix A.2) has two subroutines. One is $valmat$ that receives the matrix size n, the number of nonzero elements nnz, the RHS vector x, the vector to be validated y_t, and the matrix to use in coordinate format (a, ir, jc). $amux$ is a function form the SPARSKIT archive that performs matrix-vector products. But since it requires CSR format, a call to $coicsr$ (also from the SPARSKIT library) converts the input matrix to CSR format in place. After the call to $amux$, the subroutine calls subroutine $errpr$ that was copied from a sample program in the SPARSKIT archive to maintain the same look and feel.

Subroutine $errpr$ calculates

$$t = \sqrt{\sum_{k=1}^{n} (y_k - y_{tk})^2}$$

where n is the length of y and y_t, and y and y_t are the vectors to be compared. Then it prints the message, " 2-norm of difference in msg = t", where msg is a message supplied to the subroutine to identify the results.

The routine valmat in $ftest.f$ calls errpr with y_t being the output of *RIS_Block_Cyclic* for the global vector $y_t = A * X$, after assembling it on node P_0, and $y0$ being the output of routine $amux$ for computing the same matrix-vector product.

Chapter 4

Sparse Matrix Generator

4.1 Main

The program $genMatrix$ requires as argument the name of a configuration file. The argument is not required to include the extension, but the configuration file name is expected to have the extension ".mcfg". The program reads and displays the configuration information for the user to accept, reject, or quit. The configuration file is expected to be as in the sample shown in figure 4.1.

minValue	-1.0	
maxValue	1.0	
densityReq	0.01	
nonZeros	25e6	
groupDiagonal	0.20	0.15
groupRight	0.30	0.15
groupLeft	0.30	0.15
groupTop	0.0	0.0
groupBottom	0.0	0.0
groupCenterRow	0.0	0.0
groupCenteProl	0.0	0.0
spreadout	0.20	

Figure 4.1: Sample matrix.mcfg file

The $minValue$ and $maxValue$ entries specify the random data bounds of the matrix elements. The requested density is specified by *densityReq*. The desired number of nonzero elements is the $nonZeros$ entry. Each *groupN* entry has two numbers to the right. The first one is the nnZ share of $groupN$, and specifies the fraction of the $nonZeros$ entry that is to be applied to group N. For instance, if 20% of the desired nonzero elements is to be grouped near the diagonal, then the first number to the right of *groupDiagonal* should be 0.20. The second number specifies the group density. If, for example, the concentration around the diagonal is desired to have a density of 15%, then the second number to the right of *groupDiagonal* should be 0.15. The entry for *spreadout* indicates the fraction of the nonzero elements that should be spread throughout the matrix. All the first entries of *groupN* and the *spreadout* entry should add up to 1.0 (100%). While reading the configuration file, the program finds the boundaries between which the new matrix elements will be randomly assigned using a function called $findBounds$. With a call to the function $prepareMatrixHeader$ the program writes the heading to the ".mhdg" file.

For each row, and for each group with nnZ share grater than zero, the program determines the maximum number of data elements uB that the program will produce for that group with a function called *upperBound*. Then, with uB and the group boundaries, a column number is randomly assigned using the function *getColumn*. The matrix element value is randomly assigned with a call to $getValue$. These steps produce one single matrix element that is pushed to a linked list. When all the elements of the row are produced, the linked list contents are written to the output file with a call to $recordMatrixRow$ and empties the list for the next row with $cleanRowList$.

Chapter 5

Results

All the results and source code, are available on-line. Currently they can be found at http://my.fit.edu/~pescallo or http://my.fit.edu/beowulf under Publications. This chapter presents only the information necessary for interpretation of the results. Each test requires from three to eight separate runs depending on the purpose of the test. Each one of the plots in section 5.3, for instance, requires at least 16 tests with 3 runs each. That is a minimum of 48 runs. Each run produces nP results, one for each processor. If all the raw data were included, it would consume many pages that nobody is interested in reading. Some of the tables in this chapter contain the information to produce the plots is presented here.

5.1 Validation Results

To test the validity of the RIS implementation, SPARSKIT functions were used to test a total of ten randomly generated matrices of 50000 nonzero elements each. The matrices, named test00.mtx through test09.mtx were generated with different density patterns. The runs were made with a variety of partitions. The results are found in table 5.1. The column under the heading "Mult Time" is the RIS time at the root node. The "CSR Time" data came from timing a call to function $amux\_$ from the SPARSKIT library. The "2-nom(Diff)" column is from the output of $ftest.f$. The last four columns show the partition information.

Two plots were produced from these results. The first one is a comparison of the RIS and CSR times. To make the comparison more realistic, the plot has an "adjusted RIS" curve. This is the cost function that is the product of the "Mult Time" figure and

nP	Mult Time	CSR Time	2-nom(Diff)	nCD	nRD	Pc	Pr
1	3.67E-03	2.15E-03	2.11E-11	1	1	1	1
1	3.69E-03	2.12E-03	2.11E-11	1	1	1	1
2	2.01E-03	2.10E-03	2.11e-11	1	2	1	2
2	2.23E-03	2.11E-03	1.93E-11	12	1	2	1
3	1.76E-03	2.14E-03	2.06E-11	1	12	1	3
4	1.11E-03	2.09E-03	2.15E-11	2	2	2	2
4	1.21E-03	2.08E-03	2.24E-11	4	6	2	2
4	1.24E-03	2.06E-03	2.08E-11	16	16	2	2
4	1.20E-03	2.08E-03	2.53E-11	16	16	2	2
4	1.38E-03	2.09E-03	2.49E-11	16	16	2	2
5	1.50E-03	2.13E-03	1.70E-11	10	1	5	1
6	8.90E-04	2.03E-03	2.58E-11	8	6	2	3
6	8.22E-04	2.04E-03	2.07E-11	12	15	2	3
8	6.69E-04	1.97E-03	1.85E-11	10	8	2	4
9	6.88E-04	2.03E-03	2.09E-11	15	15	3	3
10	5.59E-04	2.01E-03	1.80E-11	10	10	2	5
11	4.69E-04	2.05E-03	2.57E-11	1	11	1	11
12	5.94E-04	2.00E-03	2.08E-11	12	15	4	3
13	5.90E-04	2.05E-03	2.21E-11	13	1	13	1
14	4.31E-04	2.05E-03	2.16E-11	14	14	2	7
15	4.60E-04	2.02E-03	2.12E-11	20	21	5	3
16	4.04E-04	2.10E-03	1.94E-11	16	16	4	4

Table 5.1: Validation Results

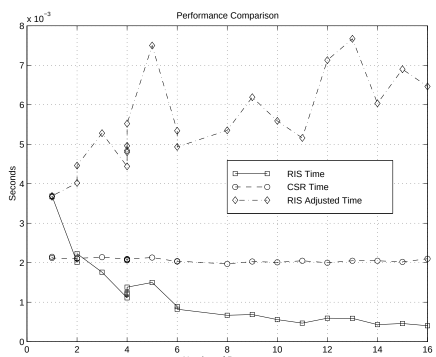

Figure 5.1: RIS, CSR Performance

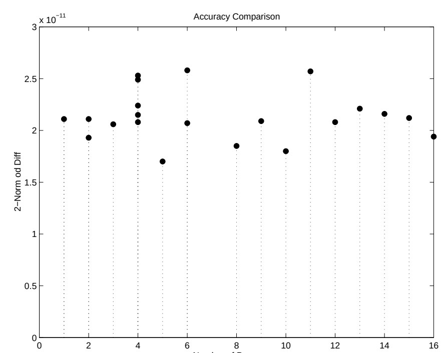

Figure 5.2: RIS Accuracy

the number of processors nP. For $nP > 1$, $T_{CSR} = 2.15e-3$ and $T_{Mult} = 3.67e-3$. Speed is proportional to $1/T$, so the relative speed of CSR over RIS is

$$relSpeed = \frac{1/T_{CSR}}{1/T_{Mult}} = \frac{T_{Mult}}{T_{CSR}}$$

or $relSpeed = 3.67e-3/2.15e-3 = 1.707$. Therefore, the single run result indicates that CSR runs about 70% faster than this RIS implementation. This result was expected from the analysis in sections 1.5 and 1.6. In the cost analysis of algorithm 1.3, an upper bound of the CSR loop time is found to be

$$\ell_{CSR} \leq (2nnZ)t + (3nnZ + 3M + 3)c.$$

And in the cost analysis of algorithm 1.4, an upper bound of the RIS loop time is found to be

$$\ell_{RIS} \leq (2nnZ)t + (5nnZ + 2M + 3)c$$

Certainly, assuming that t and c are the same for both algorithms, the CSR loop time upper bound is less than the RIS loop time upper bound by $(2nnZ - M)c$. Observing both algorithms, the big difference between them is the test for the row markers in the RIS loop. This overhead proves to be costly.

For $nP > 1$, the adjusted performance of RIS gets worse because of communication overheads. The second plot shows the accuracy deviation of RIS versus CSR using program $ftest.f$. The results prove that this sparse-matrix/dense-vector multiplication using this RIS implementation agrees with sparse-matrix/dense-vector multiplication using SPARSKIT functions.

5.2 Load Balancing

Using a generated matrix of 25 million nonzero elements, timing was gathered for several partitions. The matrix has a target number of nonzero elements of 25 million. The sparsity is specified as 30% of the nonzero elements along the diagonal with a density of 15%, 30% of the nonzero elements are concentrated on the right edge of the matrix with a density of 15%, 30% of the nonzero elements is concentrated on the left edge with a density of 15%, and 20% of the nonzero elements are spread throughout the matrix. The contents of the $genMatrix$ input file mat25M.mcfg are shown in figure 4.1.

Tests were conducted in order to prove that the partition algorithm can be used to balance the processing load among the processors. Two tests are presented here. The first test involves 16 processors (table 5.2). Eight runs were made for that test, each with a different partition. Four of the runs have balanced partitions, and the other ones have unbalanced partitions. The characteristic of a balanced partition is that nCD is divisible by Pc and nRD is divisible by Pr.

cP	Run1	Run9	Run10	Run11	Run12	Run13	Run14	Run15
0	108	148	113	172	114	198	105	264
1	113	124	108	124	109	134	119	144
2	114	127	108	124	107	127	130	173
3	108	123	110	124	114	134	103	144
4	107	118	110	124	111	128	104	132
5	110	096	108	092	108	087	118	070
6	110	098	107	092	108	085	121	077
7	108	098	109	091	110	086	103	073
8	107	117	109	123	109	127	104	132
9	110	097	107	092	107	088	122	073
10	110	097	108	093	108	085	120	077
11	107	097	110	091	112	088	105	073
12	108	117	111	124	115	128	105	132
13	113	098	109	091	107	086	132	073
14	112	097	109	091	109	084	121	078
15	107	096	113	092	113	087	105	070
nCD	16	17	12	13	8	9	4	5
nRD	16	17	12	13	8	9	4	5
Pc	4	4	4	4	4	4	4	4
Pr	4	4	4	4	4	4	4	4

Table 5.2: Load Balance Results with 16 Processors

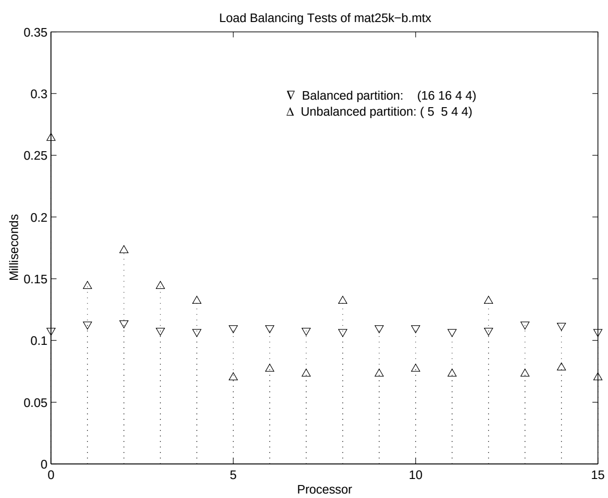

Figure 5.3: Load Balance Results with 16 Processors

Higher values of the ratios nCD/Pc and nRD/Pr in a balanced partition produce finer distributions of the data. A finer distribution implies a closer balance. The partition parameters of each test is shown at the bottom of table 5.2. From that it is easy to see which tests are balanced and which tests are not. The timing results are in milliseconds. From the partition parameters and the timing results, the most balanced run is Run1. This test's results are used in figure 5.3. The load balance plots include the most balanced and the most imbalanced results in the test to show a good contrast.

With respect to imbalance, lower values of the ratios nCD/Pc and nRD/Pr in a imbalanced partition produce greater imbalance. In the test under discussion, the partition parameters of Run15 are (ref. section 2.3) $nCD = 5$, $nRD = 5$, $Pc = 4$, $Pr = 4$, $nCG = \lceil\frac{5}{4}\rceil = 2$, $nRG = \lceil\frac{5}{4}\rceil = 2$, and $nPG = 2 \times 2 = 4$. The total number of process grids is four. One of them is complete, and the other three are incomplete as seen in the assignment map below.

0	1	2	3	0
4	5	6	7	4
8	9	10	11	8
12	13	14	15	12
0	1	2	3	0

It is easy to see from the assignment map that process 0 is assigned four blocks, processes 1, 2, 3, 4, 8 and 12 are assigned two blocks each, and processes 5, 6, 7, 9, 10, 11, 13, 14 and 15 are assigned only one block. Therefore, the expectation is that in Run15 process 0 performs its share of the sparse-matrix/ dense-vector multiplication in about twice the time as processes 1, 2, 3, 4, 8 and 12. And they, in turn, do it in about twice the time as processes 5, 6, 7, 9, 10, 11, 13, 14 and 15. The results of figure 5.3 show that indeed this is the case.

The second test (table 5.3) presented here involves twelve processors. Eight runs were performed with a different partition each. The runs chosen to generate the plots in figure 5.4 are Run16 and Run23. Run16 is the most balanced one, and Run23 is the most imbalanced one.

The partition parameters of Run23 are (see section 2.3) $nCD = 5$, $nRD = 4$, $Pc = 4$, $Pr = 3$, $nCG = \lceil\frac{5}{4}\rceil = 2$, $nRG = \lceil\frac{4}{3}\rceil = 2$, and $nPG = 2 \times 2 = 4$. The total number of process grids is four. One of them is complete, and the other three are incomplete as seen in the assignment map below.

cP	Run16	Run17	Run18	Run19	Run20	Run21	Run22	Run23
0	149	216	151	236	152	256	139	331
1	146	155	144	161	144	172	157	179
2	144	154	143	152	143	162	173	208
3	147	156	151	161	151	173	138	178
4	147	168	149	172	148	164	138	164
5	143	123	144	118	143	112	157	089
6	143	122	144	114	144	109	158	096
7	146	121	148	120	148	111	138	091
8	148	168	151	172	150	164	138	165
9	144	121	143	120	144	112	173	091
10	146	122	144	114	145	109	157	098
11	148	123	151	118	152	113	138	089
nCD	12	13	8	9	8	9	4	5
nRD	12	13	9	10	6	7	3	4
Pc	4	4	4	4	4	4	4	4
Pr	3	3	3	3	3	3	3	3

Table 5.3: Load Balance Results with 12 Processors

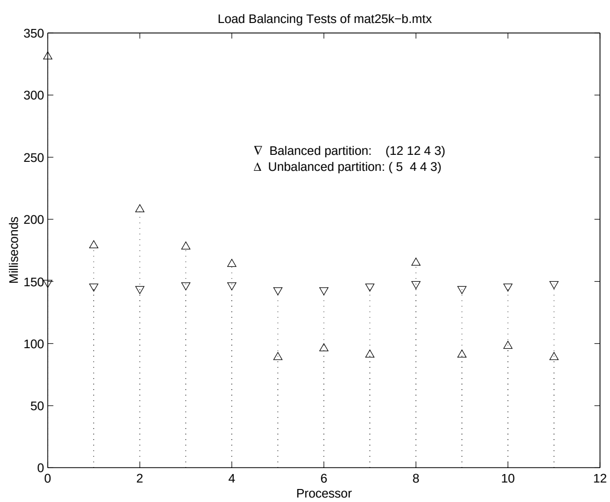

Figure 5.4: Load Balance Results with 12 Processors

0	1	2	3	0
4	5	6	7	4
8	9	10	11	8
0	1	2	3	0

From the assignment map we see that process 0 is assigned four blocks, processes 1, 2, 3, 4 and 8 are assigned two blocks each, and processes 5, 6, 7, 9, 10 and 11 are assigned only one block. Therefore, the expectation is that in Run15 process 0 performs its share of the sparse-matrix/ dense-vector multiplication in about twice the time as processes 1, 2, 3, 4 and 8. And they, in turn, do it in about twice the time as processes 5, 6, 7, 9, 10 and 11. The results of figure 5.4 show that indeed this is the case.

5.3 Timing Results

The timing results are voluminous. So instead of showing all the collected numerical data in this document, selected samples of the data are shown, and plots of the results are provided. All the timing tests were performed on the same 25 million nonzero sparse matrix. Each column of each table and each curve of each plot presented here proceed from independent sets of runs. Therefore, the numbers do not necessarily "add up". For instance, on table 5.5, one would expect that for each row the value under the "Overall" heading would be equal the sum of the "Mult" and "Reduce" readings on the same row. This is not necessarily the case because each column was produced with a separate, independent run.

```
Test 1 (mat25M-b.mtx  8 8 2 2) results:
          Prep    Bdcast   Read
Process   Time     Time    Time
-------  ------  -------  -------
   P0    8.2e-5  7.61e-2  1.41e+2
   P1    7.0e-5  7.88e-2  1.41e+2
   P2    5.0e-5  8.01e-2  1.41e+2
   P3    7.3e-5  8.16e-2  1.41e+2

Test 2 (mat25M-b.mtx  12 8 3 2) results:
          Prep    Bdcast   Read
Process   Time     Time    Time
-------  ------  -------  -------
   P0    8.3e-5  1.11e-1  1.44e+2
   P1    7.1e-5  1.12e-1  1.45e+2
   P2    7.3e-5  1.10e-1  1.45e+2
   P3    7.4e-5  1.15e-1  1.45e+2
   P4    4.8e-5  7.87e-2  1.44e+2
   P5    5.6e-5  8.28e-2  1.45e+2
```

Table 5.4: Sample Results

The speedup factor, according to Wilkinson and Allen [13] is

$$S(n) = \frac{t_s}{t_p}$$

where t_s is the time using one processor, and t_p is time using a multiprocessor with n processors. The system efficiency is defined as

$$E = \frac{t_s}{t_p \times n}$$

given as a percentage. Cost is defined as the execution time times the total number of processors used.

$$cost = \frac{t_s \times n}{S(n)} = \frac{t_s}{E} = t_p \times n$$

nP	Mult	Reduce	Overall	S(nP)	Effic	Cost	nCD	nRD	Pc	Pr	
1	1.840	0.003	1.850	1.00	100.0%	1.85	1	1	1	1	Test2
2	0.911	0.047	0.959	1.93	96.5%	1.92	2	1	2	1	Test8
3	0.559	0.240	0.799	2.32	77.2%	2.40	3	1	3	1	Test11
4	0.413	0.985	0.578	3.20	80.0%	2.31	4	1	4	1	Test17
5	0.334	0.193	0.525	3.52	70.5%	2.63	5	1	5	1	Test21
6	0.277	0.176	0.455	4.07	67.8%	2.73	6	1	6	1	Test34
7	0.238	0.167	0.401	4.61	65.9%	2.81	7	1	7	1	Test39
8	0.209	0.151	0.365	5.07	63.4%	2.92	8	1	8	1	Test61
9	0.186	0.188	0.375	4.93	54.8%	3.38	9	1	9	1	Test72
10	0.168	0.181	0.354	5.23	52.3%	3.54	10	1	10	1	Test77
11	0.153	0.179	0.332	5.57	50.7%	3.65	11	1	11	1	Test76
12	0.141	0.177	0.322	5.75	47.9%	3.86	12	1	12	1	Test53
13	0.130	0.181	0.310	5.97	45.9%	4.03	13	1	13	1	Test41
14	0.121	0.175	0.301	6.15	43.9%	4.21	14	1	14	1	Test33
15	0.113	0.177	0.294	6.29	42.0%	4.41	15	1	15	1	Test18
16	0.107	0.182	0.289	6.40	40.0%	4.62	16	1	16	1	Test1

Table 5.5: Column Balanced Results

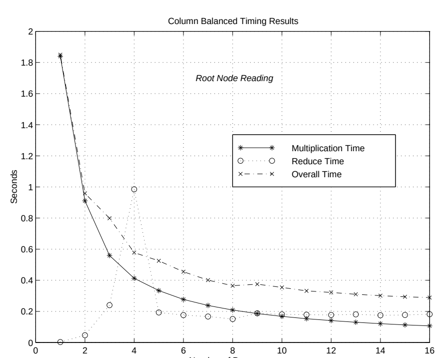

Figure 5.5: Column Balanced Results

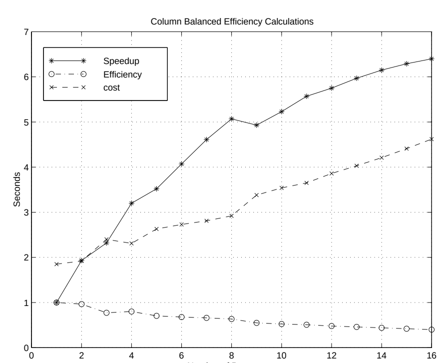

Figure 5.6: Column Balanced Efficiency

In the data sample in table 5.4, P0, P1, etc., indicate the processor that produced the results were obtained. In parenthesis at the top of each test is the matrix used for the test along with the partition parameters nCD, nRD, Pc, and Pr in that order. Each column of each test requires a complete run because MPI_Wtime can only be used once per program. Refer to figure 3.5 for definitions of these tests. Given the size of the matrix in these tests, the $PrepTime$ measurement turned out to be negligible. It only proves that memory allocation and such does not affect performance. Although measurements of the $BdcastTime$ were gathered for the population of X and X_local, they were ignored because population of the right hand side vector is not part of the algorithm. $ReadTime$ measurements were also ignored for the same reason, population of the matrix is not part of the algorithm. These parameters will not be considered any further.

Since the RIS multiplication algorithm gathers the results of all processors at the root node to produce the final answer, only the root node results are considered here. Averaged results over the processors in each test generally look better than those at the root node. Compare figures 5.5 and 5.14. To find out whether a partition type has significantly different results from the others, the results were separated into column partition, row partition, or a block cyclic partition results. For each partition, we provide one data table and two plots. One plot is for the timing results, the other one is for efficiency calculations that include speedup, efficiency, and cost as defined above. The efficiency results are also included in the data tables.

The column balanced test results of the test matrix mat25M-b.mtx are shown on table 5.5. The plots corresponding to those results are shown in figures 5.5 and 5.6. The characteristic of these tests is that there is no row partitioning and that the number of column divisions (nCD), the number of cycles (Pc), and the number of processors (nP) are the same for each test. The graphical results of figure 5.5 show that (except for $nP = 4$, the "Overall Time" curve appears to be the sum of the "Multiplication Time" and "Reduce Time" curves even if they come from separate, independent runs. The spike in the "Reduce Time" curve is unexpected under ideal conditions. In reality, since the MPI_Reduce function is an MPI function that uses a single communicator for all the nodes involved, it is possible to experience network contention. In this test, the Y_local arrays in each node is of size $50,000 \times sizeof(double)$, or $400,000$

bytes. With 16 nodes, the throughput required for the *MPI_Reduce* operation is 51.2 megabits. On principle, communication response times should be of the same order of magnitude as the matrix multiplication times. Tests with excessively large reduce or overall times were ignored.

Figure 5.6 shows the speedup, efficiency and cost curves of the column balanced test results of the test matrix mat25M-b.mtx. The efficiency curve starts out at 1.0 for one processor ($nP = 1$), and steadily decreases to about 40% for $nP = 16$. Scalability is defined as the ability of a multiprocessor to perform nP times the workload done in a single processor in the same amount of time as the single processor. In other words, if W is the work load done by a single processor in t_0 time, a scalable multiprocessor can execute a workload of size $nP \times W$ with nP processors in t_0 time. The graphical results show that for $nP = 2$, this implementation is almost scalable as seen by the speedup curve. For $nP > 2$ the $S(nP)$ seems to approach asymptotically a level no greater than seven. The maximum number of nodes available for this work was 16.

nP	Mult	Reduce	Overal	S(nP)	Eff	Cost	nCD	nRD	Pc	Pr	
1	1.840	0.003	1.850	1.00	1.00	1.85	1	1	1	1	Test2
2	0.923	0.048	0.986	1.88	0.94	1.97	1	4	1	2	Test4
2	0.924	0.047	0.978	1.89	0.95	1.96	1	2	1	2	Test7
3	0.614	0.087	0.701	2.64	0.88	2.10	1	3	1	3	Test9
4	0.460	0.091	0.549	3.37	0.84	2.20	1	4	1	4	Test16
5	0.382	0.123	0.501	3.69	0.74	2.51	1	5	1	5	Test19
6	0.309	0.127	0.435	4.25	0.71	2.61	1	6	1	6	Test29
6	0.312	0.124	0.436	4.24	0.71	2.62	1	6	1	6	Test32
7	0.267	0.129	0.394	4.70	0.67	2.76	1	7	1	7	Test36
8	0.231	0.129	0.366	5.05	0.63	2.93	1	8	1	8	Test56
9	0.204	0.168	0.370	5.00	0.56	3.33	1	9	1	9	Test71
10	0.184	0.171	0.355	5.21	0.52	3.55	1	10	1	10	Test84
11	0.169	0.167	0.337	5.49	0.50	3.71	1	11	1	11	Test75
12	0.155	0.168	0.325	5.69	0.47	3.90	1	12	1	12	Test58
13	0.142	0.176	0.316	5.85	0.45	4.11	1	13	1	13	Test44
14	0.133	0.174	0.302	6.13	0.44	4.23	1	14	1	14	Test35
15	0.123	0.174	0.299	6.19	0.41	4.49	1	15	1	15	Test20
16	0.116	0.183	0.303	6.11	0.38	4.85	1	16	1	16	Test15

Table 5.6: Row Balanced Results

The row balanced test results of the test matrix mat25M-b.mtx are shown on table 5.6. The plots corresponding to those results are shown in figures 5.7 and 5.8. The characteristic of these tests is that there is no column partitioning and that the

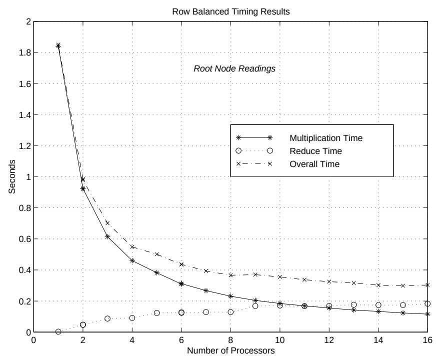

Figure 5.7: Row Balanced Results

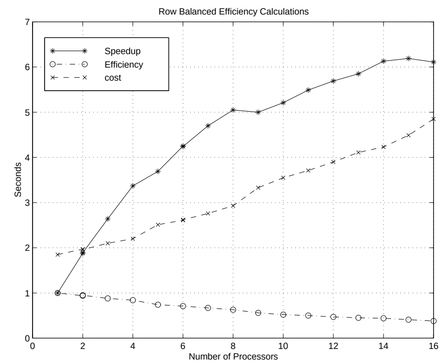

Figure 5.8: Row Balanced Efficiency

number of column divisions (nRD), the number of cycles (Pr), and the number of processors (nP) are the same for each test. The graphical results of figure 5.7 and of figure 5.6 are very similar to those of the column balanced partition tests. No further discussion is necessary.

nP	Mult	Reduce	Overal	S(nP)	Eff	Cost	nCD	nRD	Pc	Pr	
1	1.840	0.003	1.850	1.00	1.00	1.85	1	1	1	1	Test2
2	0.923	0.048	0.986	1.88	0.94	1.97	1	4	1	2	Test4
2	0.910	0.050	0.957	1.93	0.97	1.91	4	1	2	1	Test5
4	0.459	0.085	0.546	3.39	0.85	2.18	4	4	2	2	Test13
6	0.312	0.122	0.442	4.19	0.70	2.65	6	4	3	2	Test25
6	0.306	0.128	0.435	4.25	0.71	2.61	4	6	2	3	Test26
8	0.223	0.125	0.349	5.30	0.66	2.79	12	6	4	2	Test43
8	0.230	0.133	0.364	5.08	0.64	2.91	4	8	2	4	Test48
8	0.228	0.132	0.361	5.12	0.64	2.89	8	4	4	2	Test52
9	0.200	0.166	0.365	5.07	0.56	3.29	9	9	3	3	Test64
10	0.182	0.169	0.345	5.36	0.54	3.45	10	4	5	2	Test80
10	0.184	0.168	0.351	5.27	0.53	3.51	4	10	2	5	Test82
10	0.180	0.166	0.348	5.32	0.53	3.48	10	4	5	2	Test83
12	0.140	0.198	0.331	5.59	0.47	3.97	6	2	6	2	Test62
12	0.148	0.169	0.314	5.89	0.49	3.77	18	6	6	2	Test65
12	0.150	0.169	0.317	5.84	0.49	3.80	6	18	2	6	Test73
14	0.126	1.280	0.296	6.25	0.45	4.14	14	4	7	2	Test37
14	0.131	0.171	0.302	6.13	0.44	4.23	4	14	2	7	Test40
15	0.119	0.184	0.288	6.42	0.43	4.32	10	9	5	3	Test22
15	0.119	0.174	0.293	6.31	0.42	4.40	10	9	5	3	Test23
15	0.119	0.171	0.294	6.29	0.42	4.41	9	10	3	5	Test27
16	0.110	0.185	0.297	6.23	0.39	4.75	16	16	8	2	Test10
16	0.113	0.192	0.290	6.38	0.40	4.64	16	16	2	8	Test12
16	0.107	0.176	0.304	6.09	0.38	4.86	16	16	4	4	Test14

Table 5.7: Block Cyclic Results

The block cyclic test results of the test matrix mat25M-b.mtx are shown on table 5.7.

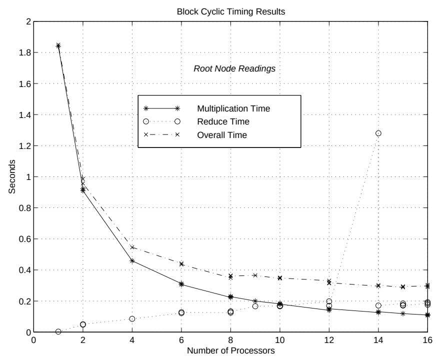

Figure 5.9: Block Cyclic Results

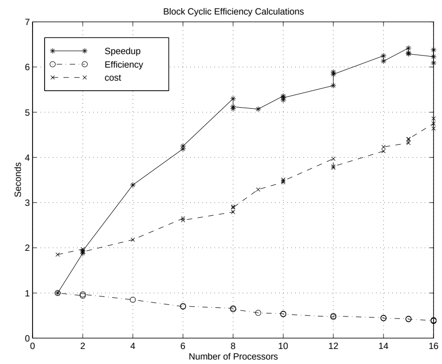

Figure 5.10: Block Cyclic Efficiency

The plots corresponding to those results are shown in figures 5.9 and 5.10. The characteristic of these tests is that they are neither column balanced nor row balanced partitions. The graphical results of figure 5.9 and of figure 5.10 are very similar to those of the column balanced and row balanced partition tests. No further discussion is necessary.

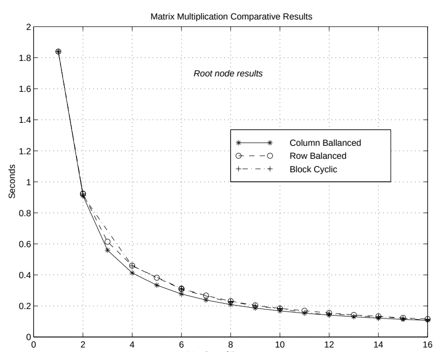

Figure 5.11: Multiplication Results

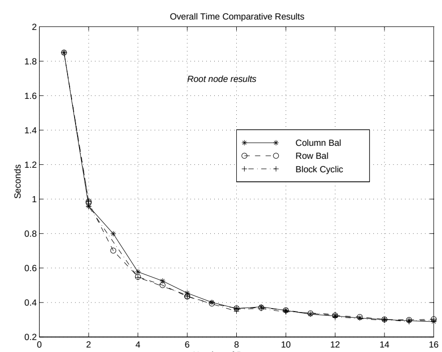

Figure 5.12: Overall Timing Results

Figures 5.11 and 5.12 show the comparative results of the multiplication and overall timings respectively for the three partition types (column balanced, row balanced, and block cyclic). The curves are almost on top of each other. This helps us conclude that the partition choice does not impact the performance. This may also prove that this is a good load balancing algorithm.

Figure 5.13 shows the comparative results of the call to *MPI_Reduce* for the three partition types (column balanced, row balanced, and block cyclic). Ignoring the spikes

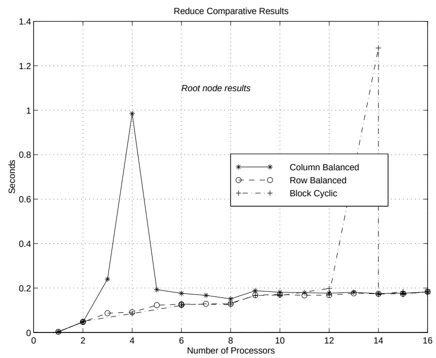

Figure 5.13: Reduce Time Results

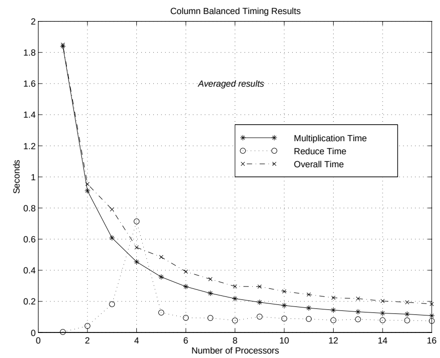

Figure 5.14: Averaged Column Balanced

in those curves, they match very well with each other.

In view of the spikes found in the previous tests, a decision was made to duplicate some of the tests on a different computer that uses myrinet technology. The tests above were run on a parallel computer that uses ethernet.

nP	Mult	Reduce	Overal	S(nP)	Eff	Cost
1	2.540	0.00690	2.570	1.000	100.0%	2.570
1	2.520	0.00650	2.530	1.016	101.6%	2.530
2	1.420	0.00750	1.430	1.797	89.9%	2.860
2	1.440	0.00760	1.440	1.785	89.2%	2.880
2	1.490	0.02520	1.500	1.713	85.7%	3.000
2	1.480	0.02470	1.510	1.702	85.1%	3.020
2	1.420	0.02370	1.430	1.797	89.9%	2.860
3	0.985	0.01870	1.010	2.545	84.8%	3.030
3	0.852	0.12900	0.981	2.620	87.3%	2.943
3	0.837	0.13900	0.992	2.591	86.4%	2.976
4	0.718	0.02680	0.737	3.487	87.2%	2.948
5	0.475	0.06460	0.570	4.509	90.2%	2.850
6	0.490	0.04450	0.500	5.140	85.7%	3.000
7	0.342	0.08260	0.420	6.119	87.4%	2.940
8	0.346	0.02640	0.379	6.781	84.8%	3.032
9	0.304	0.04090	0.331	7.764	86.3%	2.979
10	0.270	0.03810	0.305	8.426	84.3%	3.050
11	0.230	0.06140	0.281	9.146	83.1%	3.091
12	0.219	0.05040	0.261	9.847	82.1%	3.132
13	0.195	0.05560	0.243	10.576	81.4%	3.159
14	0.181	0.04410	0.228	11.272	80.5%	3.192
15	0.176	0.06100	0.238	10.798	72.0%	3.570
16	0.155	0.06180	0.222	11.577	72.4%	3.552

Table 5.8: Myrinet Results

Table 5.8 shows the results on the parallel computer that uses myrinet. Figure 5.15 is the plot of the myrinet timing results. The reduce times in these tests are much smaller than in the ethernet tests. For that reason, the matrix multiplication times and the overall times are almost coincident. But because the reduce curve is so small, is also shown in figure 5.16.

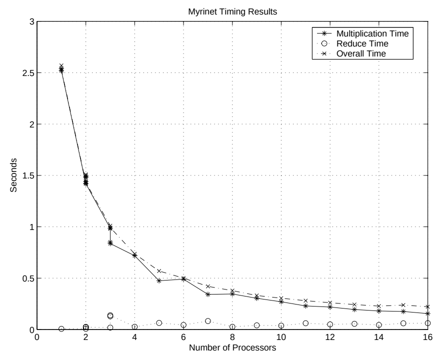

Figure 5.15: Myrinet Timing Results

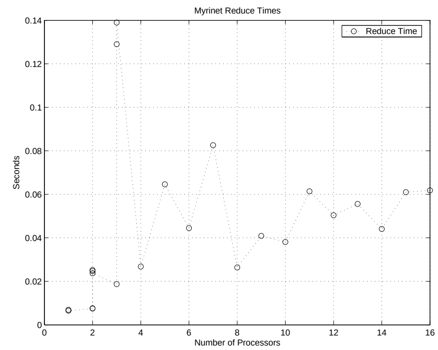

Figure 5.16: Myrinet Reduce Times

Observing the reduce time curve on figure 5.16 helps us understand the spike problem better. In the ethernet case, the reduce times were very large compared to those in the myrinet tests so the spikes are even more noticeable. Myrinet results also show spikes, but because the response is so much faster, those spikes are insignificant compared to the times to perform the matrix-vector multiplication.

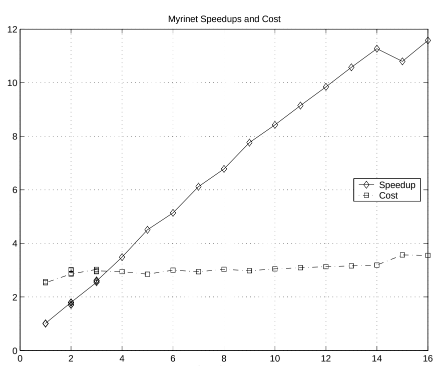

Figure 5.17: Speedup and Cost (Myrinet)

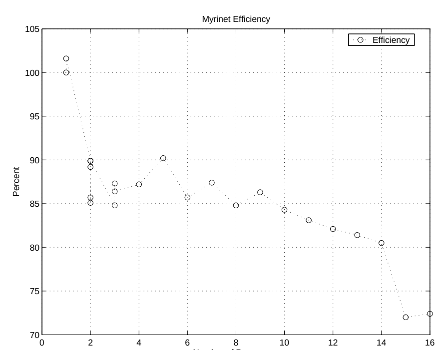

Figure 5.18: Efficiency (Myrinet)

The speedup and cost plot on figure 5.17 shows a very linear speedup curve, and a cost function that is quite flat. In this set of tests, the speedup slope is about 0.8 for a number of processors $np = 14$ or less. Additional tests were not conducted to statistically determine the degree of scalability. But looking at the results of figure 5.17, from the slope of the speedup curve, we can say that the RIS implementation is about 80% scalable with myrinet. The cost curve ranges from 2.6 to 3.6. That is pretty flat compared to the cost curve in the previous set of tests which ranges from about 1.9 to about 4.9

Figure 5.18 is the plot of the myrinet efficiency results versus the number of processors nP. This curve shows that efficiency is much higher with myrinet than with ethernet. For instance, the test with $nP = 14$ was 80% efficient. Due to resource limitations, the myrinet set of tests was limited. A larger sample could have been used to find a statistical interpolation curve. But even with the small sample, it is easy to see that efficiency is higher than 70% at $nP = 16$.

Chapter 6

Conclusion

This thesis introduces the new RIS representation for sparse matrices. This representation proved to be an improvement in storage over COO, but not over CSR. Analytical results proved that RIS has lower performance than COO and CSR. This analytical result was confirmed experimentally for CSR where it was found that the performance of the CSR representation is better than RIS by about 70% (refer to section 5.1).

This thesis develops a modified block cyclic data decomposition for sparse matrices (called here the balanced block cyclic partition) that achieves very good load balancing independently of the sparsity pattern. The theory of the balanced block cyclic partition scheme is then supplemented with graphical partition examples of a sample matrix to provide clearer insight.

The balanced block cyclic partition scheme was then implemented in a program called *RIS_Block_Cyclic* in a parallel computer using Message Passing Interface (MPI). The implementation was validated with the use of SPARSKIT functions. The validation results proved that this RIS implementation produces results that agree with those produced by the academically accepted functions of the SPARSKIT library. It was also found that the performance of the implemented RIS algorithm for sparse-matrix/dense-vector multiplication is comparable, albeit not as good, to that of function *amux_* of the SPARSKIT library.

It was explored whether the choice of partition would impact the results in this implementation. Many partitions where the number of divisions was an integral multiple of the cycle (for either row or column partitions) to achieve good load balance were tested with the conclusion that the choice of partition has no effect. Tests were run with partitions where the number of divisions is not an integral multiple of the cycle (for both row and column partitions) to obtain an imbalanced load distribution. The conclusion of these tests is that the balanced block cyclic partition is a good load balancing scheme (see section 5.2).

The multiplication algorithm with ethernet is about 40% efficient (see figures 5.6, 5.8 and 5.10). The myrinet tests, on the other hand, are better than 70% efficient. The algorithm is practically not scalable with ethernet, but with myrinet, it is estimated to be about 80% scalable.

The *MPI_Reduce* timing results show "spikes" on both ethernet and myrinet tests. The myrinet spikes were found to be insignificant with respect to the time to perform the matrix-vector multiplication, while the ethernet spikes were not insignificant. These spikes are suspected to be caused by network contention.

Chapter 7

Future Work

In the process of analysis of RIS, a hybrid representation between RIS and CSR, called here RCSR, was found. It promises to be an improvement over CSR. This would need to be explored. In fact, the balance block cyclic partition can easily be implemented in RCSR, and maybe even in CSR.

The random matrix generator suits the needs of this thesis, but the program has the potential of becoming a good matrix generator for software testing.

Program *RIS_Block_Cyclic* could be improved to concurrently partition the matrix as is being read from disk storage.

The *RIS_Block_Cyclic* algorithm should be revisited when new technology that allows optimization of the "if" loop becomes available. The upcoming Intel Itanium processor is one example of such new technology.

Appendix A

Source Code

A.1 *getColumnIndices*

```
void getColumnIndices(int N, int nCD, int nz,
int* colCount, int colStart, cycleData_t *cycl )
{
  int i=-1;
  int lastSum = nz;
  int eqLoad = nz/nCD;
  int cd, k, sumCurr, sumPrev;
  cycl->cnc[nCD-1] = N;  // All columns go to the last process
  cycl->csc[0]=colStart; // Start col for the first proccessor
  for(cd=0;cd<nCD-1;cd++) { // For each column division
    sumCurr=0, k=0;         // initialize sumCurr and k
    while( i<N && sumCurr<eqLoad) { // for each column i
      i++;
      k++;                  // increment k
      sumPrev = sumCurr;    // keep track of the previous sum
      sumCurr += colCount[i];  // incr. by elements in ith column
    }
    cycl->cnc[cd]=k;        // number of columns for CD

    if(sumCurr-eqLoad > eqLoad-sumPrev) { // closer to previous
      cycl->cnc[cd]--;      // decrement number of columns
      i--;                  // decrement column
      lastSum -= sumPrev;   // adjust elements of the last CD
      cycl->cnz[cd] = sumPrev;  // the CD nonzero elements
    }
    else {                  // otherwise ...
      lastSum -= sumCurr;   // adjust elements of the last CD
      cycl->cnz[cd] = sumCurr;  // the CD nonzero elements
    }
    cycl->csc[cd+1]=colStart+i+1; // the next cd's start column
    cycl->cnc[nCD-1] -= cycl->cnc[cd]; // adjust last nCD
  }
  cycl->cnz[nCD-1] = lastSum; // the last CD nonzero elements

  return;
}
```

A.2 *Program ftest.f*

```
C---------*---------*---------*---------*---------*-------*---
C
C     Subroutine valmat(n,nnz,x,yt,a,ir,jc)
C
C---------*---------*---------*---------*---------*-------*---
      subroutine valmat(n,nnz,x,yt,a,ir,jc)
      real*8  yt(*), a(*), x(*)
      integer n, nnz, ir(*), jc(*)

      real*8  y0(n)
      integer iwk(n+1)
      integer job = 0

      call coicsr(n,nnz,job,a,jc,ir,iwk)

      write(*, '(a)' ) '-Comparing the supplied test vector Yt  '
      write(*, '(a)' ) ' with the generated vector Y0 using the '
      write(*, '(a)' ) ' SPARSKIT subroutine amux(). '
      write(*, '(a)' ) ' '

      call amux(n,x,y0,a,jc,ir)
      call errpr(n,yt,y0,'RIS   ')

      return
      end
C---------*---------*---------*---------*---------*---------*---
C     End of valmat
C---------*---------*---------*---------*---------*---------*---

C---------*---------*---------*---------*---------*---------*---
C
C     Subroutine errpr(n, y, y1,msg)
C
C---------*---------*---------*---------*---------*---------*---
        subroutine errpr(n, y, y1,msg)
        real*8 y(*), y1(*), t, sqrt
        character*6 msg
        t = 0.0d0
        do 1 k=1,n
           t = t+(y(k)-y1(k))**2
 1      continue
        t = sqrt(t)
        write (*,*) ' 2-norm of difference in ',msg,' =', t
        return
        end
C---------*---------*---------*---------*---------*---------*---
C     End of valmat
C---------*---------*---------*---------*---------*---------*---
```

Appendix B

Support Software

B.1 *Makefile*

```
MY_BIN = $(HOME)/bin/
CC = mpicc
F77 = mpif77
SBC_RIS1  = RIS_Bk_Cyc RIS_Bk_Cyc1 RIS_Bk_Cyc2 RIS_Bk_Cyc3
SBC_RIS2  = RIS_Bk_Cyc4 RIS_Bk_Cyc5 RIS_Bk_Cyc6
SBC_RIS   = $(SBC_RIS1) $(SBC_RIS2)
GEN_PROGS = genMatrix prepMat partMat
VAL_PROGS = ris_Val RIS_Bk_Cyc7

VAL_OBJ = RIS_Block_Cyclic.o ftest.o mmio.o libskit.a
PROGS = $(GEN_PROGS) $(SBC_RIS) $(LEGACY) $(VAL_PROGS)

FLAGS = -DKNC_OUTPUT=1  -DDEBUG=0
TIMER1 = -fast -DTIMING_OUTPUT=1  -DDEBUG=0
TIMER2 = -fast -DTIMING_OUTPUT=2  -DDEBUG=0
TIMER3 = -fast -DTIMING_OUTPUT=3  -DDEBUG=0
TIMER4 = -fast -DTIMING_OUTPUT=4  -DDEBUG=0
TIMER5 = -fast -DTIMING_OUTPUT=5  -DDEBUG=0
TIMER6 = -fast -DTIMING_OUTPUT=6  -DDEBUG=0
TIMER7 = -fast -DCSR_TIMER=1  -DDEBUG=0
VALDT = -fast -DVALIDATION=1
OPTM = -fast

all:     $(VAL_PROGS) $(SBC_RIS) $(SBC_RIS2) $(VAL_PROGS)

ris_Val:     RIS_Block_Cyclic.c ftest.f mmio.o libskit.a
             $(F77) $(OPTM) -c ftest.f
             $(CC) $(VALDT) -c RIS_Block_Cyclic.c
             $(F77) -Mnomain $(OPTM) -o $@ $(VAL_OBJ)
             cp $@ $(MY_BIN)

RIS_Bk_Cyc:  RIS_Block_Cyclic.c mmio.o
             $(CC) $(FLAGS) RIS_Block_Cyclic.c mmio.o -o $@
             cp $@ $(MY_BIN)

RIS_Bk_Cyc1: RIS_Block_Cyclic.c mmio.o
             $(CC) $(TIMER1) RIS_Block_Cyclic.c mmio.o -o $@
```

```
                cp $@ $(MY_BIN)

RIS_Bk_Cyc2: RIS_Block_Cyclic.c mmio.o
                $(CC) $(TIMER2) RIS_Block_Cyclic.c mmio.o -o $@
                cp $@ $(MY_BIN)

RIS_Bk_Cyc3: RIS_Block_Cyclic.c mmio.o
                $(CC) $(TIMER3) RIS_Block_Cyclic.c mmio.o -o $@
                cp $@ $(MY_BIN)

RIS_Bk_Cyc4: RIS_Block_Cyclic.c mmio.o
                $(CC) $(TIMER4) RIS_Block_Cyclic.c mmio.o -o $@
                cp $@ $(MY_BIN)

RIS_Bk_Cyc5: RIS_Block_Cyclic.c mmio.o
                $(CC) $(TIMER5) RIS_Block_Cyclic.c mmio.o -o $@
                cp $@ $(MY_BIN)

RIS_Bk_Cyc6: RIS_Block_Cyclic.c mmio.o
                $(CC) $(TIMER6) RIS_Block_Cyclic.c mmio.o -o $@
                cp $@ $(MY_BIN)

RIS_Bk_Cyc7: RIS_Block_Cyclic.c ftest.f mmio.o libskit.a
                $(F77) $(OPTM) -c ftest.f
                $(CC) $(TIMER7) -c RIS_Block_Cyclic.c
                $(F77) -Mnomain $(OPTM) -o $@ $(VAL_OBJ)
                cp $@ $(MY_BIN)

partMat:        partMat.c mmio.o
                $(CC) $(OPTM) partMat.c mmio.o -o $@
                cp $@ $(MY_BIN)

prepMat:        prepMat.c mmio.o
                $(CC) $(OPTM) prepMat.c mmio.o -o $@
                cp $@ $(MY_BIN)

genMatrix:      genMatrix.c mmio.o
                $(CC) $(OPTM) genMatrix.c -lm mmio.o -o $@
                cp $@ $(MY_BIN)

clean:
                $(RM) core *.o *.bak *~

veryclean:      clean
                $(RM) $(PROGS)
                (cd $(MY_BIN) ; $(RM) $(PROGS))
```

Bibliography

[1] R. F. Boisvert, R. Pozo and K. A. Remington, *The Matrix Market Exchange Formats: Initial Design*, (NISTIR 5935) NIST Computing and Applied Mathematics Laboratory, Gaithersburg, MD, 1996.

[2] Eun-Jin Im, *Optimizing the Performance of Sparse Matrix-Vector Multiplication*, PhD Dissertation, Graduate Division, University of California at Berkeley, 2000.

[3] D. A. Patterson, J. L. Hennessy, *Computer Architecture A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[4] P. Angeli, O. Basset, C. Fulton, G. Howell, et al, *Some Issues in Efficient Implementation of a Vector Based Model for Document Retrieval*, Florida Institute of Technology, Melbourne, Florida, June 23, 2001.

[5] W. Press, S. Teukolsky, W. Vetterling, B. Flannery, *Numerical Recipes in C The Art of Scientific Computing*, Second Edition, Cambridge University Press, Cambridge, 2002.

[6] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[7] P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.

[8] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI - The Complete Reference*, Second Edition, MIT Press, Cambridge, Massachusetts, 1998.

[9] Y. Saad, *SPARSKIT: a basic tool kit for sparse matrix computations*, Version 2, University of Minnesota, Minneapolis, MN, 1994.

[10] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second Edition with Corrections, ©Yousef Saad, 2000.

[11] S. Goedecker, A. Hoisie,*Performance Optimization of Numerically Intensive Codes*, SIAM, Philadelphia, PA, 2001.

[12] G. Golub, C. Van Loan, *Matrix Computations* Third Edition, The Johns Hopkins University Press, Baltimore, MD, 1996.

[13] B. Wilkinson, M. Allen, *Parallel Programming, Techniques and Applications using Networked Workstations and Parallel Computers*, Prentice Hall, Upper Saddle River, New Jersey, 1999.

[14] G. Almasi, A. Gottlieb, *Highly Parallel Computing*, Second Edition, The Benjamin/Cummins Publishing Company, Inc. Redwood City, California, 1994.

[15] J. Purdum, *C Programming Guide*, Second Edition, Que^{TM} Corporation, Carmel, Indiana, 1983.

[16] K. Robbins, S. Robbins, *Practical Unix Programming*, A Guide to Concurrency, Communication and Multithreading, Prentice Hall OTR, Upper Saddle River, New Jersey, 1996.

[17] L. Blackford et al, *ScalaPACK User's Guide*, Siam, Philadelphia, Pennsylvania, 1997.